TU, PU, PoU, KU

Insights on EMBEDDED SYSTEM

Features

- INSTANT NOTES
- Solution to large number of numericals including that from past exams.

Scanned with CamScanner

Chapter 1

INTRODUCTION TO EMBEDDED SYSTEM

1.1 Embedded System Overview ....

1.2 Classification of Embedded Systems 1

1.2.1 Classification Based on Generation 4

1.2.2 Classification Based on Complexity and Performance 4

1.3 Hardware and Software in a System 5

1.4 Purpose and Application of Embedded Systems 6

1.4.1 Purpose of Embedded Systems 9

1.4.2 Applications of Embedded Systems 11

Chapter 2

HARDWARE DESIGN ISSUES

2.1 Combination Logic 12

2.1.1 Basic Combinational Logic Design 12

2.1.2 RT-Level Combinational Components 13

2.2 Sequential Logic 15

2.2.1 RT Level Sequential Components 15

2.2.2 Sequential Logic Design 16

2.3 Custom Single-Purpose Processor Design 18

2.3.1 Steps for Designing Single-Purpose Processor....20

2.3.2 Design Example of a Single-Purpose Processor..22

2.3.3 RT Level Custom Single-Purpose Processor Design 26

2.4 Optimizing Custom Single-Purpose Processors..........27

Solution to Important Questions..........31

Practice Design Questions 45

Chapter 3

SOFTWARE DESIGN ISSUES

3.1 Basic Architecture 46

3.2 Operation 48

3.3 Programmer's View 50

3.4 Development Environment 54

3.4.1 Tools for Implementation and Verification Phase....54

3.4.2 Design Flow 56

3.5 Application-Specific Instruction Set Processors 58

3.6 Selecting a Microprocessor 60

3.7 General-Purpose Processor Design 61

Chapter 4

MEMORY

4.1 Introduction 67

4.2 Memory Write Ability and Storage Permanence 68

4.3 Common Memory Types 70

4.4 Composing Memory..........80

4.5 Memory Hierarchy and Cache 83

4.5.1. Memory Hierarchy..........83

4.5.2. Cache Memory..........84

Chapter 5

INTERFACING

5.1 Communication Basics..........90

5.2 Microprocessor Interfacing 97

5.2.1. I/O Addressing..........97

5.2.2. Interrupts – Interrupt Driven I/O 99

5.2.3. Direct Memory Access – DMA controller 102

5.3 Arbitration 103

5.4 Multilevel Bus Architectures 108

Scanned with CamScanner

5.5 Advanced Communication Principles 10
5.6 Serial, Parallel and Wireless Protocols 11

Chapter 6

REAL TIME OPERATING SYSTEM

6.1 Operating System Basics 120
6.1.1. The Kernel 122
6.1.2. Real Time Kernel 124
6.1.3. Kernel Space and User Space 124
6.1.4. Types of Kernel 125
6.2 Task Process and Threads 126
6.2.1 Process 126
6.2.2. Threads 129
6.3 Multiprocessing and Multitasking 134
6.3.1. Context Switching 135
6.3.2. Types of Multitasking 136
6.4 Task Scheduling 136
6.5 Task Synchronization 140
6.5.1. Task Communication/Synchronization Issues...140
6.5.2. Task Synchronization Techniques 143
6.6 Device Drivers 146
Solution to Important Questions..................147

Chapter 7

CONTROL SYSTEMS

7.1 Introduction 158
7.2 Open-Loop and Closed-Loop Control Systems Overview .158
7.2.1 Open-Loop Control Systems 158
7.2.2 Closed-Loop Control Systems 159
7.2.3 Comparison of Open-Loop and Closed Loop Control Systems 160
7.2.4 Open and Closed Loop Control System Design Example 160
7.3 General Control Systems and PID Controllers 167
7.3.1 Control Objectives 167
7.3.2. Modeling Real Physical Systems 169
7.3.3. Controller Design 169
7.4 Software Coding of PID Controller 174
7.5 PID Tuning 176
7.6 Practical Issues Related to Computer-Based Control 176
7.7 Benefits of Computer-Based Control Implementation...178

Chapter 8

IC TECHNOLOGY

8.1 Introduction 179
8.1.1 IC Manufacturing Process 181
8.1.2 Photolithography 183
8.2 Full-Custom (VLSI) IC Technology 187
8.3 Semi-Custom (ASIC) IC Technology 189
8.3.1 Gate Array Semi-Custom IC Technology 189
8.3.2 Standard Cell Semi-Custom IC Technology 190
8.4 Programmable Logic Device (PLD) IC Technology 191
Solution to Important Questions..................193

Chapter 9

MICROCONTROLLERS IN EMBEDDED SYSTEMS

9.1 Intel 8051 Microcontroller Family, its Architecture and Instruction Sets 195
9.2 Assembly Language Programming 209
9.2.1 Assembly Language Programming Format 209
9.2.2 Delay Calculation in Assembly Program.........213
9.3 Interfacing with Seven Segment Display 214
Solution to Important Questions..................217

Scanned with CamScanner

10.1 Introduction ....
10.2 VHDL Code Structure..235
10.3 Data types, Data Objects and Operators..235
10.4 Statements in VHDL 238
10.5 Standard Architectures 243
10.6 FSM Design..246
Solution to Important Questions..250
....252

1 byte = 8 bit

Chapter 1

INTRODUCTION TO EMBEDDED SYSTEM

- Embedded System Overview
- Classification of Embedded Systems
- Hardware and Software in a System
- Purpose and Application of Embedded Systems

1.1 Embedded System Overview

1. Introduction

Embedded system is a microprocessor/microcontroller based hardware system with integrated software, designed to perform a dedicated functn within larger mechanical or electronic system.

An embedded system is a combination of hardware and software designed to perform a specific function. The hardware consists of mechanical parts and electronic circuits while the software represents the program instructions that cause embedded system to operate its functionality. The programs written for the embedded systems can also be referred as firmware which is stored in read-only memory. An example of embedded system can be a digital watch. A digital watch with simple configuration can contain 4-bit processor, registers, counters, real-times clocks as electronic components. And other hardware elements of the watch can be buttons/touch screen for inputs and screen & speaker for output.

Embedded systems are used to control, monitor or assist the operation of an equipment, machinery or plant. So, an embedded system may be designed for specific control functions within a larger system, often with real time computing constraints. Hence, in many cases, an embedded system is a component within some larger system. For example, modern cars contain embedded systems like embedded airbag system, navigation system, adaptive cruise control, and few others left unmentioned.

2. Characteristics

i. **Single functioned:** As embedded systems are designed for specific control functions, it usually executes a specific program to carry out the specific function repeatedly.

ii. **Tightly constrained:** In a way, tightly constrained means optimizing the embedded system in various system defining

Scanned with CamScanner

parameters. In terms of few attributes, an embedded syste must be economic, small in size, fast enough to process data real time and must consume minimum power.

iii. **Reactive and real time:** In general, embedded systems mu continually respond to changes in the system's environment an must perform instant data processing without delay. A delay i computation and slow response may result a failure in th operation of the system.

3. Design Metrics

A design metric is a measurable feature of the system's performance cost, time for implementation and safety, etc. Some of the common used metrics include:

i. **NRE cost (non-recurring engineering cost):** It represents the monetary cost for designing the system. Since the cost doesn't occur more than once for a particular system, it is termed a nonrecurring.

ii. **Unit cost:** It is the monetary cost of manufacturing each unit of the system excluding NRE cost.

iii. **Size:** It is the physical space required by the system. For software it is measured in terms of bytes and for hardware it is measured in terms of no of gates or transistors.

iv. **Performance:** It is measured in terms of the execution time of the system.

v. **Power:** The amount of power consumed by the system, which may determine the lifetime of a battery, or the cooling requirements of the IC.

vi. **Flexibility:** The ability to change the functionality of the system without incurring heavy NRE cost.

vii. **Time to prototype:** The time needed to build a working version of the system, which may be bigger or costlier than the final system implementation.

viii. **Time to market:** The time required to develop a system to the point that it can be released and sold to customers.

ix. **Maintainability:** The ability to modify the system after it initial release.

x. **Correctness:** We can check the functionality throughout the process of designing the system and we can insert test circuitry to check that manufacturing was correct.

xi. **Safety:** The system is supposed to cause no harm.

The Time to Market Design Metric

Introduction of an embedded system to the marketplace significantly affects the overall system profitability. The market window, period during which the product have highest sales, for products is getting shorter, so a short delay on introduction of product to the marketplace can render huge loss. Using a simplified model of revenue as shown in the figure below, we will deduce the loss of revenue that can occur due to delayed entry of a product in the market.

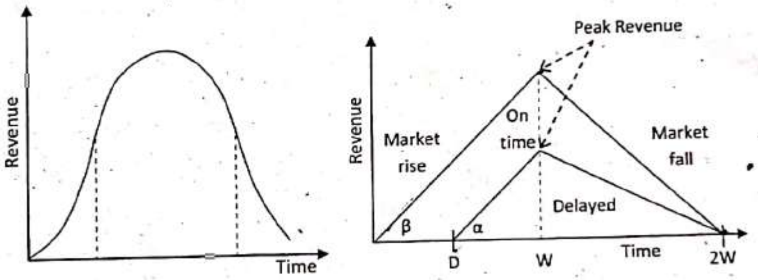


Figure 1.1: Market window and simplified revenue model for loss calculation for delayed entry

This model assumes the peak of the market occurs at the halfway point, denoted as W, of the product life. The peak is same for delayed entry. The revenue for an on-time market entry is the area of the triangle labeled On-time, and for delayed entry is the area of triangle labeled *Delayed*. The difference between the areas of two triangles gives the revenue loss for a delayed entry.

$$\text{Revenue Loss} = \frac{\text{On time} - \text{Delayed}}{\text{On time}} \times 100$$

$$\text{Area of on time triangle} = \frac{1}{2} \times \text{base} \times \text{height}$$

$$= \frac{1}{2} \times 2 \times W \times W \times \tan\beta$$

(Assuming, market rise angle is β)

$$= W^2 \tan\beta$$

Scanned with CamScanner

Area of delayed entry triangle = $\frac{1}{2} \times (2W - D) \times (W - D) \times \tan\alpha$

Assuming $\beta = \alpha$, and on solving we get,

$$\text{Revenue Loss} = \frac{D(3W - D)}{2W^2} \times 100\%$$

4. Example of an Embedded System – A Digital Camera

A digital camera can be taken as embedded system as it performs only a single function of capturing image. It is tightly constrained as it is affordable, portable, and consumes less power. And as it is fast enough to process numeral images in milliseconds, it exhibits real time feature. But however, a simple digital camera may not possess high degree of reactive attribute. On the contrary, few contemporary digital cameras are capable of detecting human expressions.

1.2 Classification of Embedded Systems

Embedded systems can be classified using various aspects like functionality, application, generation, and complexity & performance. But we will discuss the categorization of embedded systems based on generation and complexity & performance in this section.

1.2.1 Classification based on Generation

1. First Generation

Embedded systems were designed using 8-bit microprocessors or 4-bit microcontrollers. Hardware circuits were simple and the firmware was developed using assembly code. Motor controller using 8085 can be taken as an example of first generation embedded system.

2. Second Generation

The systems were built using 16-bit microprocessors and 8/16-bit microcontrollers. More complex and powerful instructions were available for the designer to work with. Some systems involved embedded operating systems for their operation. Data Acquisition Systems can be an example of second generation embedded systems.

3. Third Generation

The systems were designed with more advanced processor technology in the form of 32-bit processors and 16-bit microcontrollers. Along with complex and powerful instruction sets, instruction pipelining was introduced for better performance. Dedicated embedded real time operating system implementation was another important feature in this generation. Also, the concept of application specific processors like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASIC) came into existence.

4. Fourth Generation

Fourth generation was marked with the advent of System on Chips (SoC), reconfigurable processors and multicore processors. These embedded systems used high performance real time embedded operating systems for its operation.

1.2.2 Classification Based on Complexity and Performance

See the article for simpler explanatn

1. Small Scale Embedded Systems

These systems' are designed with a single 8-bit or 16-bit microcontroller (8051 family, PIC16F8X, Hitachi H8). They have little hardware and software complexities and involve board level design. They may be battery operated. While developing embedded software for these system, an editor, assembler and cross assembler specific to the microcontroller or processor are used as the main programming tool. Usually C language is used for developing these systems. Automatic vending machine, stepper motor controller for a robotics system, etc. can be the examples of small scale embedded systems.

2. Medium Scale Embedded Systems

These systems are designed with a single or few 16-bit or 32-bit microcontrollers (8051MX, PIC16F876) or DSPs or Reduced Instruction Set Computers (RISCS). It may also employ the readily available single purpose processors and IPs for various functions, for example: bus interfacing, encryption, deciphering and so on. These systems have both hardware and software complexities. For software design, the programming tools used is RTOS, source code engineering tool, simulator, debugger, and integrated development environment (IDE). Software tools also provide the solutions to the hardware complexities. Some of the examples of medium scale embedded systems are computer networking systems, signal tracking system, etc.

Scanned with CamScanner

3. Sophisticated Embedded Systems or Large Scale Embedded Systems (32/64 bit microcontrollers)

These systems have enormous hardware and software complexities and may need scalable processors or configurable processors and programmable logic arrays. They are used for cutting edge applications that need hardware and software co-design and integration in the final system. They are constrained by the processing speeds available in their hardware units. Certain software functions are implemented in the hardware to obtain additional speed by saving time. Some of the functions of the hardware resources in the system are also implemented by the software. These systems generally implement high performance real time operating system. Development tools for these systems may not be readily available at a reasonable cost or may not be available at all. In some cases, a compiler or retargetable compiler might have to be developed for these. (A retargetable-compiler is one that configures according to the given target configuration in a system). Embedded System for wireless LAN & for convergent technology devices is one of the sophisticated embedded systems.

1.3 Hardware and Software in a system

1. Single Purpose Processor

Single purpose processor (SPP) is a digital circuit designed to execute exactly one program. In other words, it is a circuit that represents a program or functionality of a specific task. So, it does not require a program memory in its configuration. In general, SPPs are used for simple and less computation intensive operations in which stored program concept is not required. Hence, this simple dedicated task oriented configuration makes SPPs small in size and consumes less power for operation. However, SPP lacks flexibility as same design configuration cannot be used to perform operations other than the specified one.

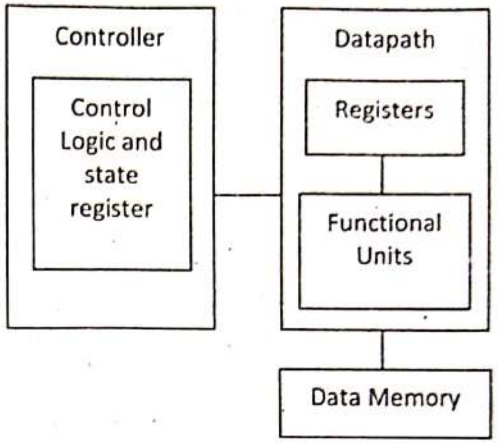


Figure 1.4: Block diagram of a single purpose processor

The single purpose processor contains controller, datapath and data memory. Controller is used to generate control signals to carry out operations in datapath. The datapath contains only the essential components for the specified task. And SPP contains data memory for temporary storage during computation. DMA controller can be taken as the examples of single purpose processor.

allows I/O devices to directly access memory with less participation of processor

2. General Purpose Processor

The general purpose processor (GPP) is a programmable device that supports wide range of functionality. The required functionality is carried out by programming the processor's memory. Microprocessors are the examples of general purpose processor. GPPs are highly flexible as it supports change of functionality based on requirement ~~to the extent~~ that the given configuration of the processor supports the operation. For complex and high computational operations, GPP can be effective as majority components of the system will be in operation. However, for simple operations, memory access will cause the operation to become slow and additional components might increase power consumption.

Scanned with CamScanner

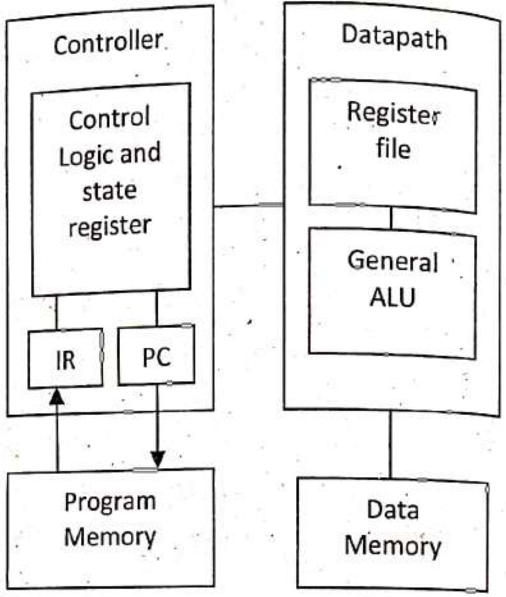


Figure 1.5: Block diagram of a general purpose processor

The general purpose processor includes controller, datapath, da and program memory.

i. **Controller:** It is used to generate control signals based on t instruction provided in the program memory. It consists instruction register (IR) and program counter (PC). IR is used hold the instruction that needs to be executed and PC is use to sequence through the instructions.

ii. **Program memory:** The program cannot be built or converte into an equivalent digital circuit in general purpose processo since the program likely to run on the processor will b unknown. Hence, program memory is used to store progra instructions.

iii. **General Datapath:** The datapath must be general enough t handle a variety of computations, so the datapath typically ha a large register file and one or more general purpos arithmetic logic units (ALUs).

3. Application Specific Processors

Application specific processors are programmable processor optimized for a particular class of applications. It generally include program memory, optimized datapath and special functional units. These processors provide optimum level of performance maintaining appropriate size and power consumption. Microcontrollers for controlling application and digital signal processors (DSPs) for huge data processing application are examples of application specific processors.

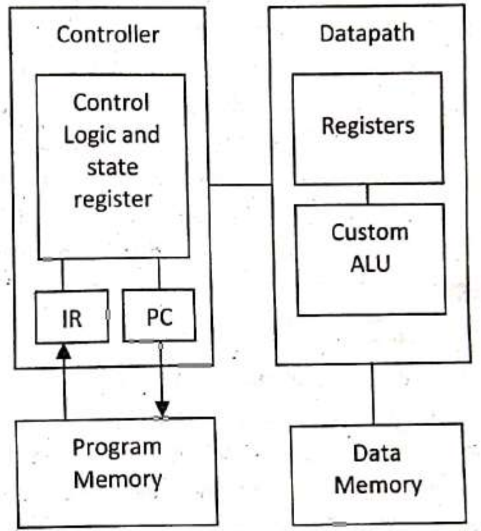


Figure 1.6: Block diagram of an application specific processor

1.4 Purpose and Application of Embedded Systems

The main purpose of an embedded system is to automate the human driven activities such that the task can be performed with higher reliability and efficiency. And regarding the application of embedded system, it has a wide range of application that varies from consumer electronics to industrial equipment, entertainment to academic devices and medical instruments to weapons and aerospace control systems.

1.4.1 Purpose of Embedded Systems

1. **Data collection**

In embedded systems, the data is collected from other external devices for storage, analysis, manipulation or transmission. Data may be in analog or digital form. Systems working with digital data require analog to digital converters if the collected data is in analog form.

The collected data can be used for meaningful purpose based on functionality of the embedded system. For instance, a digital cam collects data, stores it and finally provides graphical representatio data in the form of captured image.

2. **Data communication**

An embedded system is required to connect two or more devi which may be at close vicinity or at remote location. communication between devices can be done via wired line medi or wireless medium. Embedded systems are incorporated w different wireless modules or wire-line modules for communicat purpose. For example, if we have to transfer images captured us camera into Laptop, we can use either WIFI or serial communicati (using data cable) based on which mode of transmission is suppor by our devices.

3. **Data Processing**

The collected data in embedded system is subjected to some sort processing for which embedded systems are attributed with d processing modules. Speech coder, audio video codec, etc can be t examples of data processing unit. Data processing includes t manipulation of data for appropriate purpose.

4. **Monitoring**

Many embedded systems are incorporated with sensors to check t state of the different parameters. The parameters can be curre voltage, temperature, humidity, etc. which are continuou monitored and appropriate processing or controlling of devices done. However, the value of the parameters cannot be controlled the system itself. The values of parameters are used for so controlling purpose or for some graphical representation purpose simply stored for further analysis and processing.

5. **Control**

For control purpose, actuators along with sensors are present in t embedded systems. The sensor connected in input port detects t change in the desired parameter and the actuators at output po are controlled accordingly to implement the desired functionalit Electric Motors are examples of actuators. In an object avoid robot, ultrasonic sensor senses the presence of certain kind of object and the motor is rotated accordingly to avoid the collision.

6. **Application specific user interface**

To provide a better user interface based on application has been one of the concerns of contemporary embedded systems. Keypads, simple LCD modules, speakers, etc are basic and common interface for users. However, sensitive touch pad along with high definition display has been the sophisticated interface implemented in current scenario.

1.4.2 Applications of Embedded Systems

The applications of embedded systems are:

1. **Household appliances:** microwave ovens, television, DVD players and recorders.
2. **Consumer electronics:** camera, video games
3. **Office Utility:** fax machines, printers, scanners
4. **Business equipment:** alarm systems, card readers
5. **Automobiles:** engine controller, fuel injection, antilock brakes
6. **Networking:** modem, network cards, network switches and routers
7. **Medical equipment:** MRI scanner, sonography, blood pressure device and glucose test set
8. **Security**

Let us review one example of embedded system related to the household appliances.

Microwave Oven: Basically, microwave ovens use electromagnetic waves to generate heat by moving water molecules. When caught in electromagnetic waves, water molecules move very quickly in a counterclockwise then clockwise motion, alternating back and forth at extreme speeds. This movement generates heat energy which causes the food items to warm rapidly.

The microwave often has a display, keyboard, and a number of sensors and actuators. Sensors can be a temperature sensor or the sensor that detects whether the door is closed. Actuators can be the electronic switch that controls a microwave tube or a system that controls the rotational speed of a turntable within the microwave.

Scanned with CamScanner

HARDWARE DESIGN ISSUES

- Combinational Logic
- Sequential Logic
- Custom Single-Purpose Processor Design
- Optimizing Custom Single-Purpose Processors

2.1 Combination Logic

Combinational circuit is a digital circuit whose output is purely function of its present inputs. Combination logic circuits are made up fro basic gates or universal gates that are combined or connected together produce more complex switching circuits. In general, logic gates are t building blocks of combinational logic circuits. It has no memory block. Som of the examples of the combinational circuits are decoder, multiplexe adder, ROM, etc.

2.1.1 Basic Combinational Logic Design

In combinational logic design, output is purely a function of t present inputs and has no memory of past inputs. We can use basic log gates to design combinational circuits. In such design, outputs are describe in terms of inputs.

1. General Steps for Combinational Logic Design

- The problem description (question) is translated into a trut table with all possible combinations of input values.
- The input values lies on the left of the truth table and th corresponding output values of the inputs lies on the right o the truth table.
- For each output, we have to derive the equations. The equation may contain number of combinations of the inputs. The number of combinations depends on the number of high (1) value of each column of the output. Rows of the inputs are used to derive the equation corresponding to the high output of the column. And the equation must be further minimized.
- Another way to derive minimized equation directly is by using k map. It is always better to use k-map unless the design is too simple (when the output column consists of only one high value).
- The final equation is translated to an equivalent circuit diagram using logic gates.

2. Combinational Logic Design Example

Example 1:

In an alarm system of a bank, three sensors are implemented and the alarm is triggered when at least two sensors detect the change. Assuming sensors to output digital values, design a combinational logic circuit for alarm system.

Solution:

Let a, b, c represent the three sensors and y represents the buzzer for alarm. The output y should be high when two or more than two inputs are high. The truth table and its corresponding combinational design are shown below.

Truth Table

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

K-Map

y a \ bc	00	01	11	10
0	0	0	1	0
1	0	1	1	1

y = ac + bc + ab

Combinational Circuit

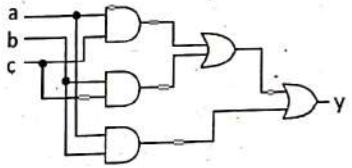


Figure 2.3: Truth table, K-map, and combinational circuit for bank alarm system

2.1.2 RT-Level Combinational Components

Register-transfer or RT level components are generally used when the design of the circuit becomes complex. As the number of input increase, the complexity of the design increase. One of the ways to reduce design complexities is by using RT-level components. Multiplexers, decoder, adder are the examples of RT-Level Components.

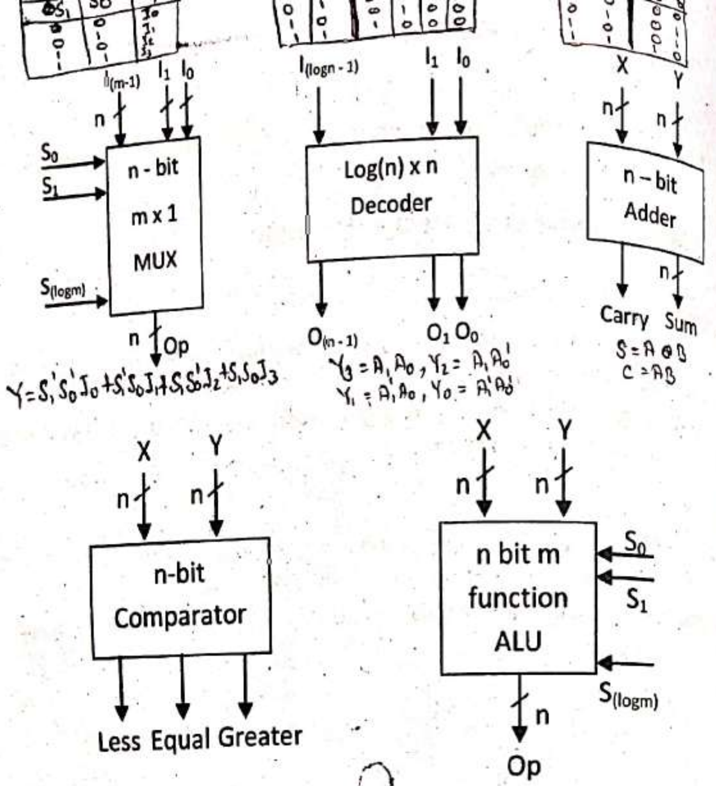


Figure 2.4: Few commonly used RT-level combinational components

i. **Multiplexer** allows only one of its data inputs to pass through to the output. For m×1 multiplexer, there are m data inputs and one data output with $\log_2 m$ select lines. The value of select lines determines which input data to pass through to the output. It can be used for parallel to serial conversion.

ii. **Decoder** allows exactly one of the output lines to be high at a given time for a particular input. For n input lines, there will be 2^n output lines. A decoder can be used for coding the addressing lines in the memory. It can be used to convert binary to a suitable form.

iii. (hots) **Adder** is used to add two n-bit inputs producing an n-bit sum along with a carry of 1 bit.

iv. **Comparator** allows to compare two n-bit binary inputs, generating the corresponding output based on whether one input is less than equal to, or greater than another input.

v. **Arithmetic-logic unit** (ALU) performs variety of arithmetic and logic functions on its n – bit inputs. The select line is used to select which function is to be carried out. If there are 2^m functions that can be done by ALU then there must be at least m select lines.

vi. **Shifter** is another example which is used to shift the bits of the input right or left. It can be used as a divider or multiplier. For example shifting 0110 (6) to the right would give 0011 (3).

- latches -> level sensitive enable signal
- flip flop -> edge sensitive " "

2.2 Sequential Logic

A sequential circuit is a digital circuit whose outputs are a function of not only the present inputs but also the past inputs. The output of a sequential logic depends on its present internal state and the present inputs. Hence, a sequential logic circuit has some kind of memory. Logic gates and flip-flops are the basic building blocks of sequential logic circuits.

Flip-flop is an example of sequential logic circuit. A flip-flop stores a single bit. The different types of flip flops are listed below.

- **D flip-flop**: It has two inputs D and clock, when clock is high, value of D is stored in flip flop and same will be the value of the output Q. When clock is low, previously stored bit is maintained ignoring the value of input D. (ensures S & R are never equal to 1 at the same time, eliminating drawback of SR-flip flop)
- **SR flip-flop:** It has three inputs S (set), R (reset), and clock. When clock is low, the previously stored bit is maintained ignoring the clock is low, the previously stored bit is maintained ignoring the values of input at S and R. When clock is high, the output varies with inputs S and R. If S is high, the output Q will be high and high bit (1) will be stored by the flip-flop. If R is high, then low bit (0) will be stored. The output will not change if both the inputs are low but the undefined condition will occur if both the inputs are high.
- **JK flip-flop:** Its operation is similar to that of SR flip-flop but when both the inputs J and K is high, the stored bit toggles either from high to low or low to high.
- T flip-flop: (toggle flip flop) Q will change state if T=1, and maintain its state while T=0. Made by shorting J & K inputs of JK flip-flop.

2.2.1 RT Level Sequential Components

(go to neso academy, if you want to understand these)

- **Register**

A register stores n bits from its n-bit data input which also appears at its output. A register usually has at least two control inputs, clock and load. For a rising edge triggered register, the inputs are only stored when load is high and clock is rising from 0 to 1. Another control

↳ flip flop is a 1-bit memory cell
↳ For storing n-bits, we use a n-bit register, which is built using n flip-flops

Scanned with CamScanner

input clear may be used to resets all bits to 0 regardless of the val of input. Since all n bits of the registers can be stored in parallel, w refer this type of register as a parallel load register.

- **Shift Register**

  A shift register stores n bits from its one bit data input with at lea two control inputs clock and shift. When clock is rising and shift is the nth bit of input is stored in the (n–1)th bit, and (n–1)th bit of inp is stored in the (n–2)th bit and so on down to the second bit bein stored in the first bit. The first bit is shifted out appearing as a output bit. It has one bit output and the input must be shifted in the register serially.

- **Counter**

  A counter is a register that adds binary 1 to its stored binary value. I general, a counter has a clear, and count as a control inputs. Clea resets all stored bits to 0 and a count input enables incrementing o each clock edge. A common counter feature is both up and dow counting which required an additional control input to indicate th count direction.

A small triangle in the block represents the clock input for an sequential logic.

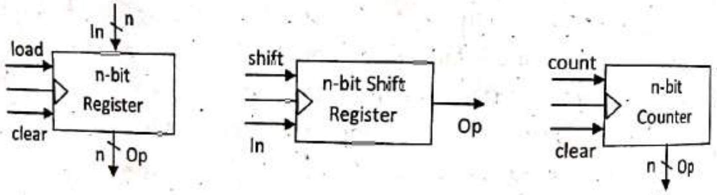


Figure 2.5: RT-level sequential components

2.2.2 Sequential Logic Design

For a sequential logic design, either Moore or Mealy model needs to be used. However, the design steps remain the same for any model used for designing.

1. **General steps for Sequential Logic Design**

- Translate the problem description to a state diagram, also called a finite state machine (FSM).

  In FSM, each circle represents a state where desired output values are listed with each state. Whereas, the input conditions which cause a transition from one state to another are listed next to each arc.

- Assign each state a unique binary value, and create a truth table for the combinational logic. The external inputs and the bits coming from the state registers are fed to the combinational logic as inputs. Whereas, the external output values along with the state bits to be loaded into the state register acts as the output of the combinational logic.

- The output values change only with the current state, so we list the external output values only for each possible state, regardless of the change in external input values.

- Now, we can have a truth table, with the help of which we can proceed with combinational design by generating minimized output equations using k-map. And finally, drawing the combinational logic circuit.

2. **Sequential Logic Design Example**

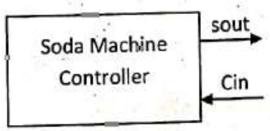
Example 1:

Design a soda machine controller, given that a soda costs 75 cents and your machine accepts quarters only. Draw a black-box view, come up with a state diagram and state table, minimize the logic, and then draw the final circuit.

Solution:

The coin must be entered three times to get a soda out of the machine. Throughout the design, Cin represents the coin input and sout indicates the soda output whereas Q1, Q0 represent current state and I1, I0 represent next state.

A. Black Box View



B. State Table

Inputs			Outputs		
Q1	Q0	Cin	I1	I0	Sout
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	0	0	1
1	1	1	0	1	

D. K-map

I1 \ Q1Q0	00	01	11	10
Cin 0	0	0	0	1
1	0	1	0	1

I1 = Q1Q0' + Q1'Q0Cin

I0 \ Q1Q0	00	01	11	10
Cin 0	0	1	0	0
1	1	0	1	1

I0 = Q1'Q0Cin' + Q1Cin + Q0'Cin

sout \ Q1Q0	00	01	11	10
Cin 0	0	0	1	0
1	0	0	1	0

sout = Q1Q0

E. Combinational Circuit

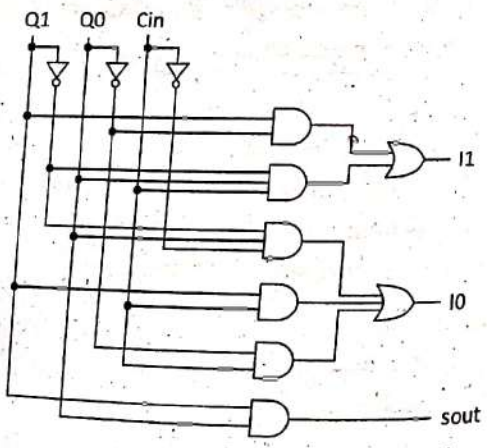


Figure 2.6: Soda machine controller design

2.3 Custom Single-Purpose Processor Design

A basic processor consists of a controller and a datapath.

1. **Datapath**

- It stores and manipulates a system's data.
- It contains register units, functional units and connection units like wires & multiplexors.
- The datapath can be configured to read data from particular registers, feed that data through functional units configured to carry out particular operations like add or shift, and store the operation results back into particular registers.
- Examples of data include binary numbers representing external conditions like temperature or speed, characters to be displayed on a screen.

2. **Controller**

- It sets the datapath control inputs (like register load and multiplexor select signals) of the register units, functional units, and connection units to obtain the desired configuration at a particular time.
- It monitors external control inputs as well as datapath control outputs, known as status signals, coming from functional units, and it sets external control outputs as well.

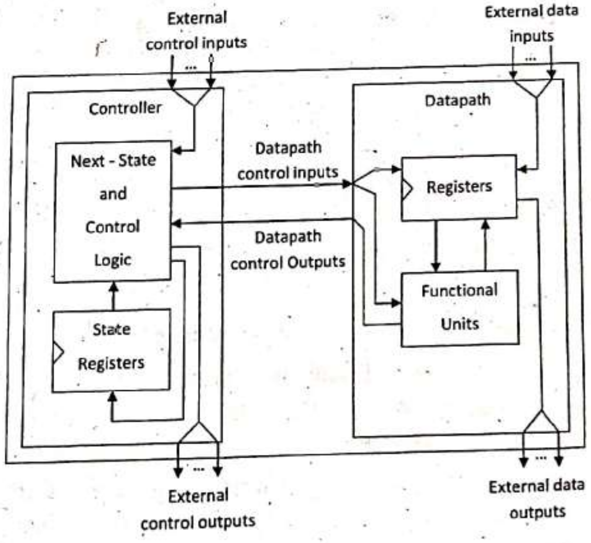


Figure 2.7: Internal view of controller and datapath of single purpose processor

2.3.1 Steps for Designing Single-Purpose Processor

1. **Draw a black box diagram:** Black box diagram is a simple box w external interfaces of a system. It generally includes input and out signals along with few control signals.

2. **Write the functionality or program:** The functionality or program code which provides the solution to the defined problem.
   - The input signals are assigned to a variable.
   - Number of temporary variables may be used based requirement.
   - The final result is assigned to the output port.

3. **Design a finite state machine with data (FSMD):** The code converted into equivalent complex state diagram which is known finite state machine with data. In FSMD, templates are used represent various constructs of program. The templates assignment, branch statement and loop statement are discuss below.

   i. **Assignment statement:** For this statement, a single state is us with statement representing its action. Generally, a single arr is used to connect to next state. The template used statement S = A + B is shown as an example.

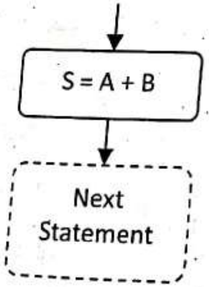
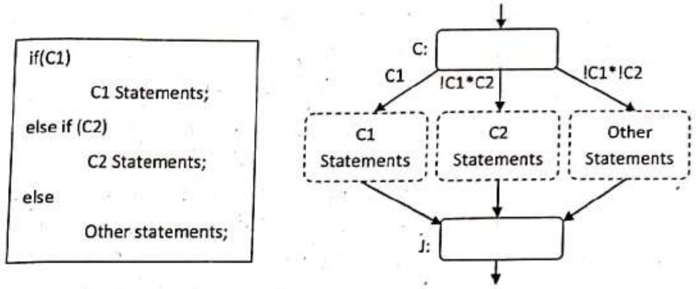


Figure 2.8: Template for assignment statement

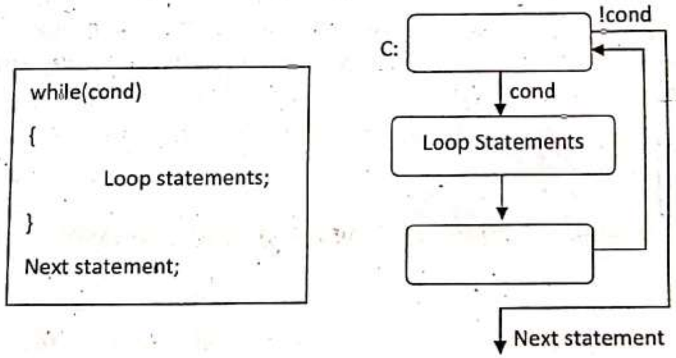
   ii. **Branch statement:** It can be represented by using conditi state C, join state J, and few other states in between C and state. State C and State J are with no actions, left empty. B states between C state and J state contain actions. Its templat can vary depending on number of conditions defined in th problem. However, for each true condition, there can be sever states representing actions. Conditions are written alongsid with the arrow that connects the C state and states of each branch. Last states of each branch are connected to the J state.



```
if(C1)
        C1 Statements;
else if (C2)
        C2 Statements;
else
        Other statements;
```

Figure 2.9: Template for branch statement

   iii. **Loop statement:** Its template consists of Condition State C, Join State J, and other states representing statements of loop. Condition is written alongside arrow connecting condition state and state of first statement of loop. The last state of loop is connected to the J state which is connected back to condition state. Complement condition is used alongside arrow connecting C state and next statement outside of loop. The template for the loop statement is shown in the figure below.



```
while(cond)
{
        Loop statements;
}
Next statement;
```

Figure 2.10: Template for loop statement

4. **Build a datapath:** The datapath is build based on functionality of the system. Following steps are needed to be taken into considerations while developing a datapath.

Scanned with CamScanner

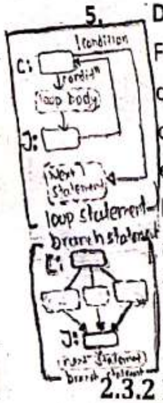
i. **Registers**: The number of registers to be used is defined by number of variables used in the functionality. Registers assigned to inputs, temporary variables and output.

ii. **Functional units**: Blocks representing arithmetic and log operations are defined within the datapath.

iii. **Connections**: The connections among ports, registers functional units are done based on operands used in vari assignments and comparison of functionality code. Appropri multiplexor is required when the value in register can assigned from more than one source. The sources may be input port, a functional unit, or another register.

iv. **Control inputs and outputs**: Input control signals are gene required by registers and multiplexor. Register load signa used in case of register while selection line signals multiplexor. Control output is produced by logical units of datapath. Each control singles are given a unique identifier.

5. **Develop a finite state machine (FSM):** The states and transitions FSM are same as that of FSMD. However, the complex actions conditions of FSMD are replaced by Boolean expressions using control signals defined within datapath. For every register w operations (assignment statement, arithmetic statements), regi load signal is asserted and corresponding multiplexor selection li activated if there are two or more sources for a given register. the logical operations are replaced by the control signals of corresponding functional block.



C -> condition state
J -> join state

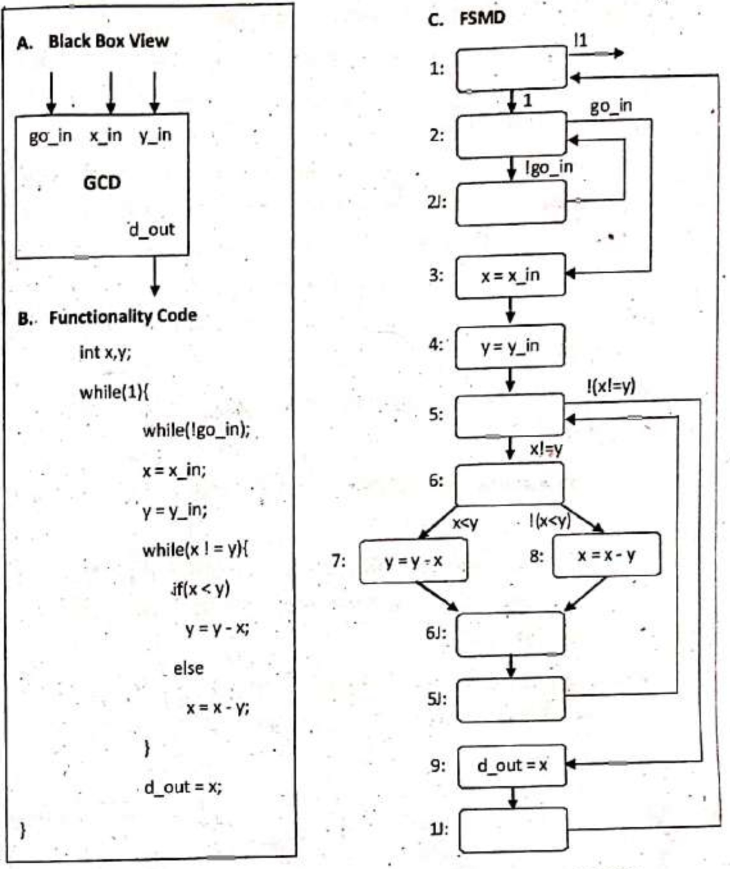
2.3.2 Design Example of a Single-Purpose Processor

Example 1:

Design a single purpose processor that calculates the great common divisor (GCD) of two numbers. Include FSMD, datap and FSM in the design.

Solution:

Initially, the black box view diagram is drawn and then followed the functionality which is converted into FSMD using appropri templates.



```
int x,y;
while(1){
        while(!go_in);
        x = x_in;
        y = y_in;
        while(x != y){
            if(x < y)
                y = y - x;
            else
                x = x - y;
        }
        d_out = x;
}
```

Figure 2.11: Black box view, functionality and FSMD diagram of GCD processor

1. **Datapath for GCD processor:** To construct the datapath, we need to determine the number of registers required, functional blocks for operations, mux and connection requirement, and control lines for register and mux.

Number of registers: Two inputs x_in and y_in assigned to variables x and y, and use a register d for generating output signal (d_out), and no other temporary variables are used. Hence, three registers x, y and d are required.

Functional blocks: The arithmetic and relational operation involved in the functionality are x-y, y-x, x!=y and x<y. Hence, two subtractors and two comparing blocks are required.

Scanned with CamScanner

MUX requirement and connections: The value in register x ha sources, x_in and 'x-y', so it requires a multiplexor of 2x1. Similar case for register y. For connections, the output of registers x a are connected to inputs of subtracting blocks and comparing b Also, the line representing x_in and x-y are connected to the inpu mux whose output is fed to register x. Similarly, y_in and y- connected to the register y through mux. And, the output register is connected to input of register d. All connections mu done so as to represent the corresponding operation in functionality.

Control signals: Unique identifier for various control signa assigned.

- **Load signal of registers:** x_ld for register x, y_ld for regis and d_ld for register d.
- **Selection lines of multiplexor:** x_sel for multiplexor associ with register x and y_sel for multiplexor associated register y.
- **Signals from logical block:** x_neq_y and x_lt_y are used for x equal to y and x less than y respectively.

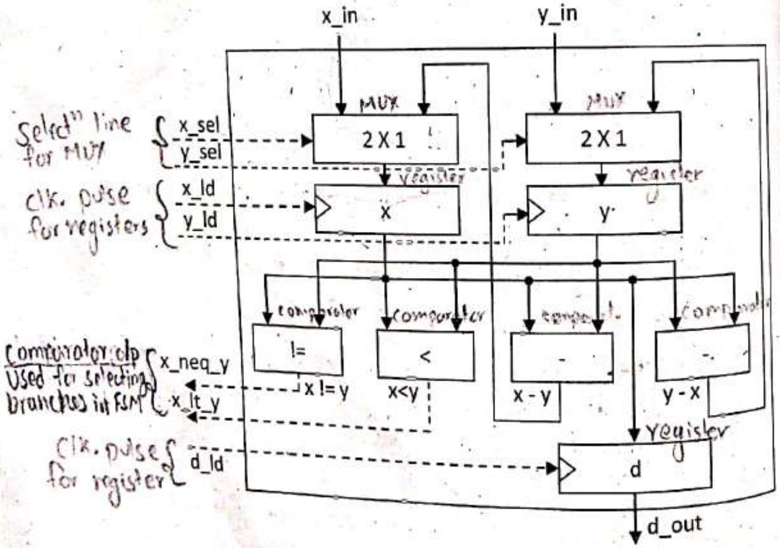


Figure 2.12: Datapath of GCD processor

2. Finite state machine for GCD processor

All actions and conditions are replaced by equivalent Boolean expressions as used in datapath. For example, action x := x_in is replaced by expressions x_sel = 0 and x_ld = 1. x_sel = 0 will connect the input line x_in through mux to register x and x_ld = 1 will load the value of x_in into x. In case of d_out = x, only d_ld = 1 is used as it has only one source and no multiplexor is used. And condition x< y is replaced by x_lt_y. The identifiers for control signals, however, used in FSMD must match with the one that is defined in datapath.

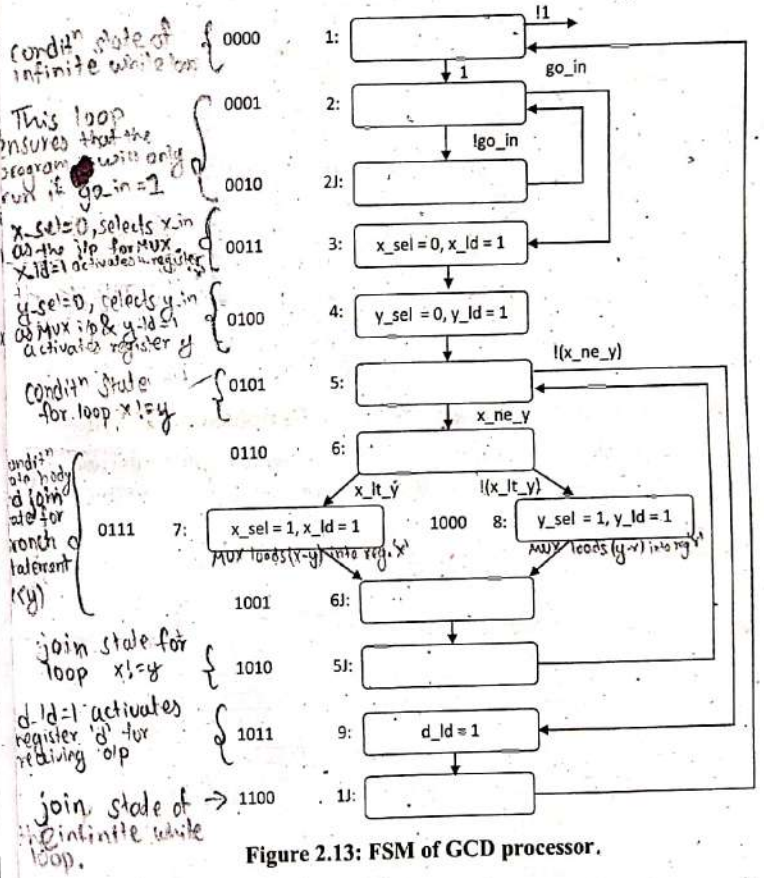

Conditⁿ state of infinite while loop
0000
1:
!1
1
go_in
This loop ensures that the program will only run if go_in = 1
0001
2:
!go_in
0010
2J:
x_sel=0, selects x_in as the i/p for MUX. x_ld=1 activates the register
0011
3:
x_sel = 0, x_ld = 1
y_sel=0, selects y_in as MUX i/p & y_ld=1 activates register y
0100
4:
y_sel = 0, y_ld = 1
!(x_ne_y)
Conditⁿ state for loop x!=y
0101
5:
x_ne_y
0110
6:
x_lt_y
!(x_lt_y)
Conditⁿ state body d join state for branch of tolerant (<y)
0111
7:
x_sel = 1, x_ld = 1
MUX loads (x-y) into reg. x
1000
8:
y_sel = 1, y_ld = 1
MUX loads (y-x) into reg. y
1001
6J:
join state for loop x!=y
1010
5J:
d_ld=1 activates register 'd' for receiving o/p
1011
9:
d_ld = 1
join state of the infinite while loop.
1100
1J:

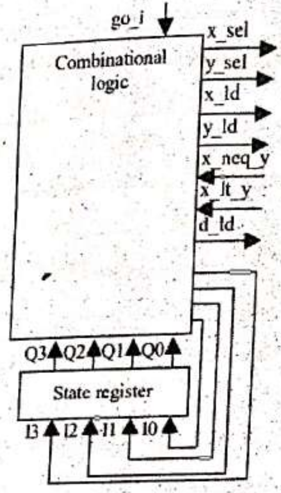
Figure 2.13: FSM of GCD processor.

Scanned with CamScanner

3. Building Controller implementation model

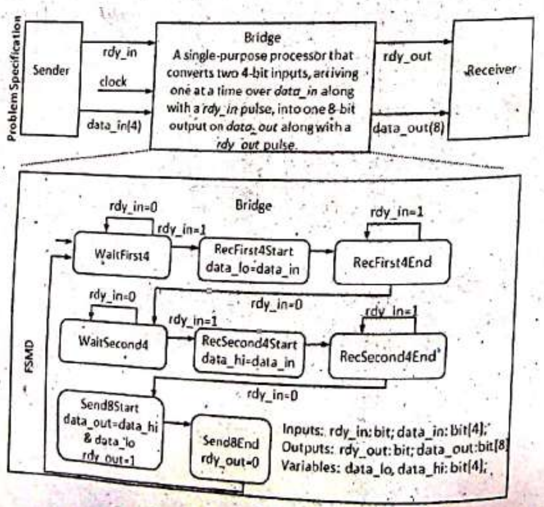
- Use all the control signal representations from datapath
- Number of state register is defined by number of states used to represent FSM. For example, if there are 13 states then we need 4 bits to represent 13 states. And hence we have to use 4 state registers.

Controller implementation model



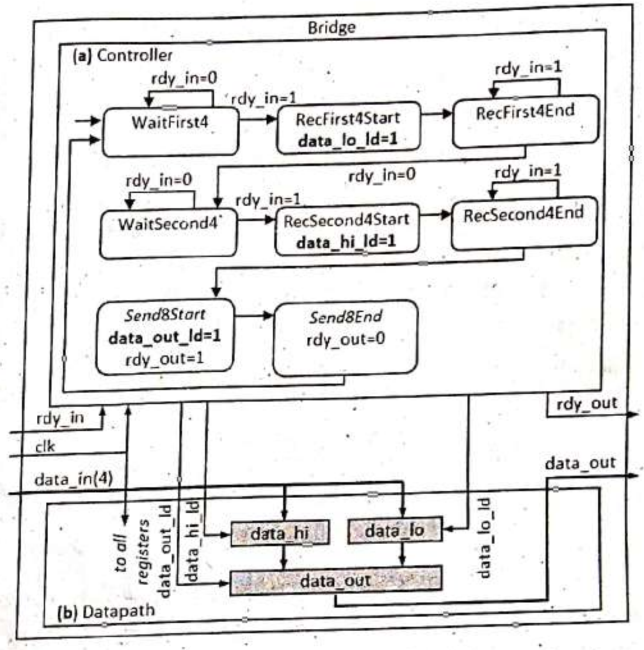
2.3.3 RT Level Custom Single-Purpose Processor Design

Let us consider an example of a bridge which combines two 4-bit inputs, arriving one at a time from the sender terminal into one 8-bit output to the receiving terminal.



A single-purpose processor that converts two 4-bit inputs, arriving one at a time over *data_in* along with a *rdy_in* pulse, into one 8-bit output on *data_out* along with a *rdy_out* pulse.

Inputs: rdy_in: bit; data_in: bit[4];
Outputs: rdy_out: bit; data_out: bit[8]
Variables: data_lo, data_hi: bit[4];



2.4 Optimizing Custom Single-Purpose Processors

Optimization is the task of making design metric values the best possible. Optimization can be done by simplifying the resulting design of any system utilizing various techniques. Different states in the FSM can be removed which does nothing and are redundant. Also, we can share a component for same operations in different states and hence minizing the size of the system as well as its cost. Other various factors can be considered for optimum design but some simple optimization that can be applied are discussed further.

i. Optimizing the Original Program

We should analyze different program attributes and try to develop alternative algorithm that are more efficient. We can analyze the algorithm in terms of time complexity and space complexity. Number of computations can be a form of time complexity whereas the size of variables required corresponds space complexity.

Lets compare two different logics to calculate GCD:

```
while(x ! = y){
```

Scanned with CamScanner

```
if(x < y)
y = y-x;
else
x = x-y;
}
```

To compute GCD of 42 and 8, it takes 9 iterations to complete operation, x and y will take different values as (42, 8), (34, 8), (26 (18, 8), (10, 8), (2, 8), (2, 6), (2, 4), (2, 2). And it requires variables.

```
while(y != 0){
r = x % y;
x = y;
y = r;
}
```

To compute GCD of 42 and 8, it takes 3 iterations to complete operation, x and y will take values as (42, 8), (8, 2), (2, 0). If y = 42 = 8, it will take 4 iterations, one more than previous. But it requi three variables.

ii. Optimizing the FSMD

Each state in an FSMD is assigned with operations from the desi program; this process is also termed as scheduling. The schedul process can be improved by following methods.

- **Merge states**: States with independent operations can merged.
- **Eliminate state**: States with constants on transitions can eliminated since transition to be taken will be fixed as defi by constants. And some states without any operation can be eliminated.
- **Separate states**: States which require complex operations be broken into smaller states to reduce hardware size.

Considering the example of GCD:

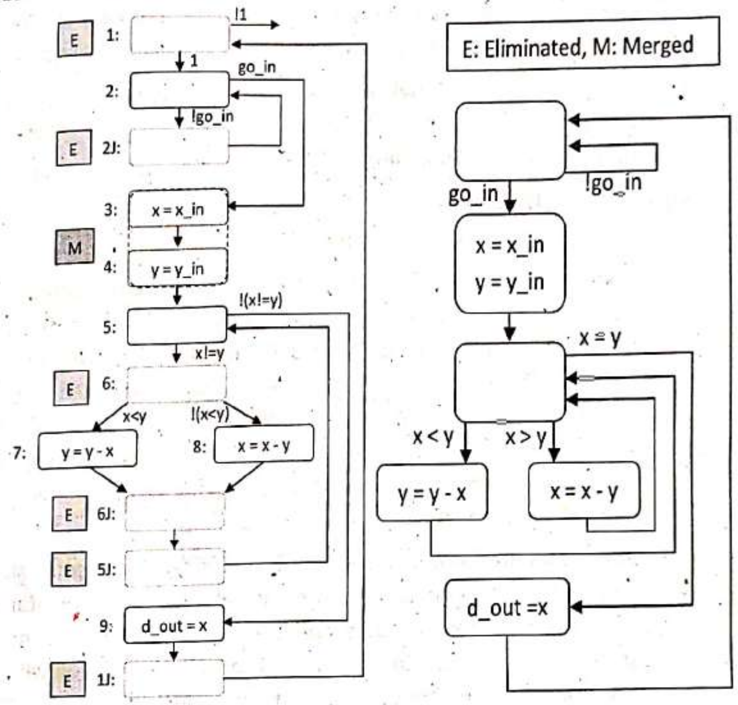


Figure 2.14: Optimized FSMD from original FSMD

Consider the operation p = a*b*c*d, if we use single state for this particular operation then three multipliers are required which makes system expensive and bulky. So the operation can be broken down as x= a*b, y = c*d and p = x*y with each operations having its own state. Thus, only one multiplier would be required in the system.

iii. Optimizing Datapath

During the datapath design, the task of selecting a RT components for particular operation is termed as allocation. Whereas the task of mappinig operations from the FSMD to allocated components is termed as binding. The optimization in datapath design can be done by following ways.

- **Sharing of functional units**: Single functional unit can be shared if same operations occur in different states. For example, in computation of GCD there were two subtractor used for subtraction, rather a single subtractor can be used with the of the multiplexor. Hence one to one mapping is not necess

- **Use of multi-functional units**: A variety of operations can performed by ALU hence it can be shared for diffe operations occuring in different states.

iv. Optimizing the FSM

Optimization in FSM can be done by:

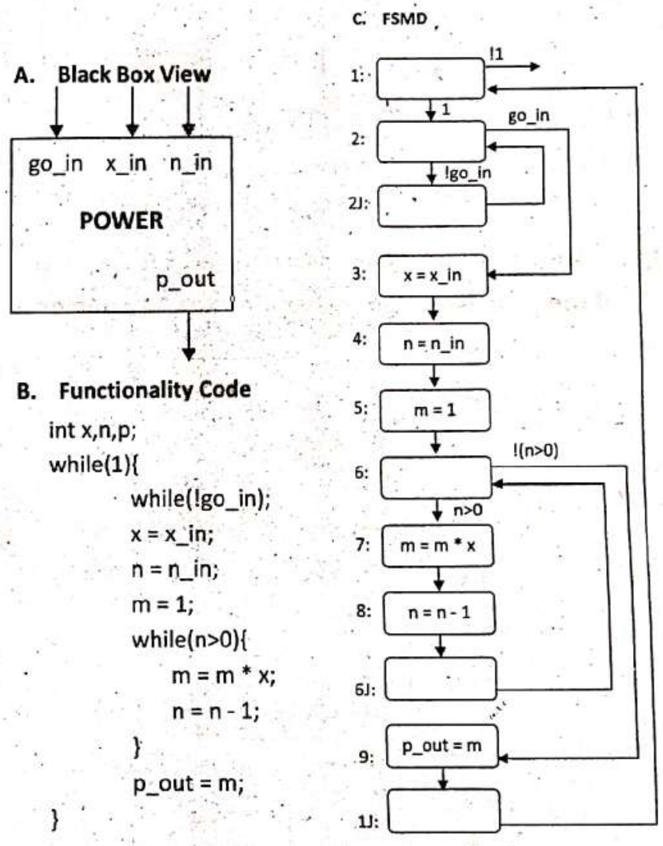
- **State encoding:** It is the task of assigning a unique bit patten each sate in an FSM. The size of the register as well as the of the combinational logic varies for different encodings. example, if we have four states then it can be encoded as 01, 10 ,11 but it can also be encoded as 11, 10, 01, 00. If number of state is large the number of ways of state encod will be very large, hence CAD tools are used to determine most efficient encodings.
- **State minimization:** It is the task fo merging equivalent sta into a single state. Two states are equivalent if those two sta generate the same outputs and transition to the same state, for any given input combinations. Merging equival states yield exactly the same output behaviour.

SOLUTION TO IMPORTANT QUESTIONS

Problem 1:

Design a single purpose processor that calculates x to the power n (x^n). Include FSMD, datapath and FSM in the design. [2074 Bhadra]

Solution:



```
int x,n,p;
while(1){
        while(!go_in);
        x = x_in;
        n = n_in;
        m = 1;
        while(n>0){
            m = m * x;
            n = n - 1;
        }
        p_out = m;
}
```

Figure: The black box view, functionality and FSMD for processor that calculates x^n.

Scanned with CamScanner

D. Datapath for the processor that calculates x^n

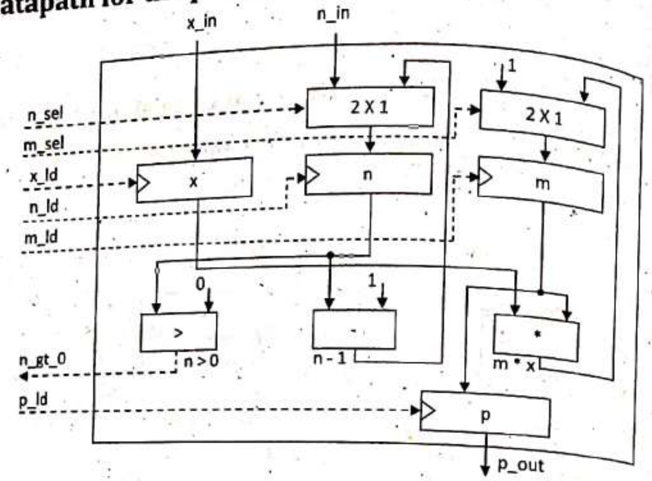


Figure: Datapath of the processor that calculates x to the power n

E. FSM of the processor that calculates x to the power n

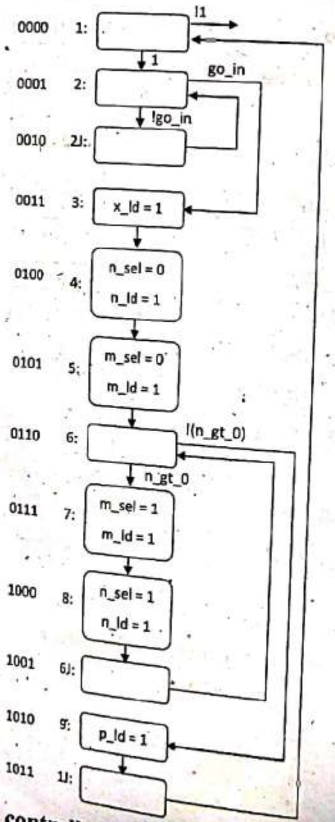


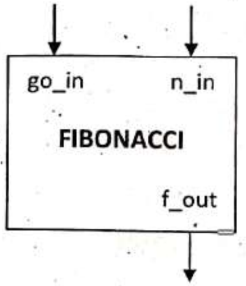
Figure: FSM controller of processor that calculates x^n.

roblem 2:

Design a single purpose processor that generates Fibonacci series up to n places. Start with a function that computes desired result, translate the function into a state diagram, sketch a probable datapath, and draw FSM diagram. [2075 Baishakh]

olution:

A. Black Box View



B. Functionality Code

```
int ft, st, nt, count, n;
while(1){
        while(!go_in);
        n = n_in;
        ft = 0;
        st = 1;
        count = 1;
        while(count <= n){
            f_out = ft;
            nt = ft + st;
            ft = st;
            st = nt;
            count++;
        }
}
```

C. FSMD

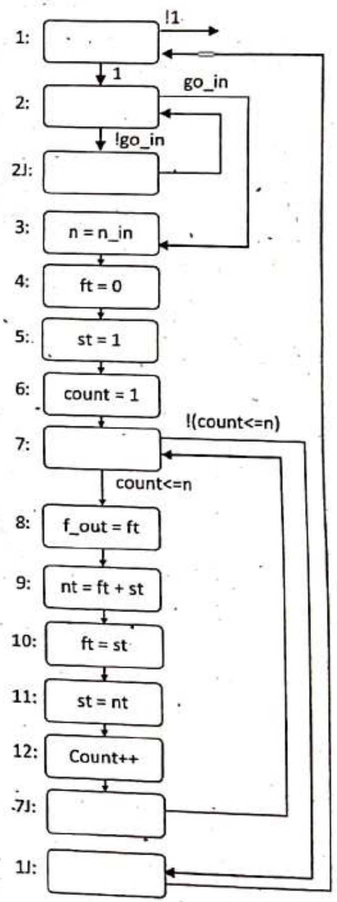


Figure: Fibonacci series generator – the black box view, functionality and FSMD

D. Datapath of the processor that generates Fibonacci ser

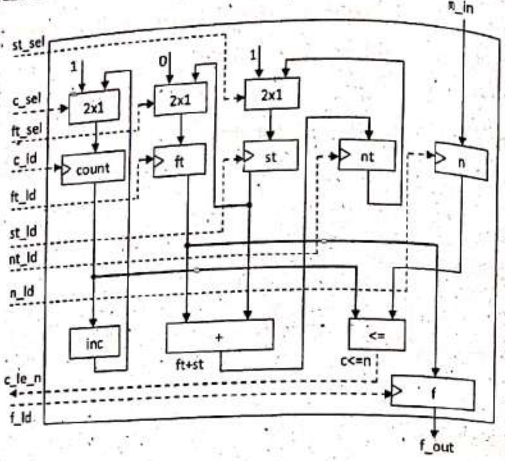


Figure: Datapath for Fibonacci series generator

E. FSM controller for Fibonacci series generator

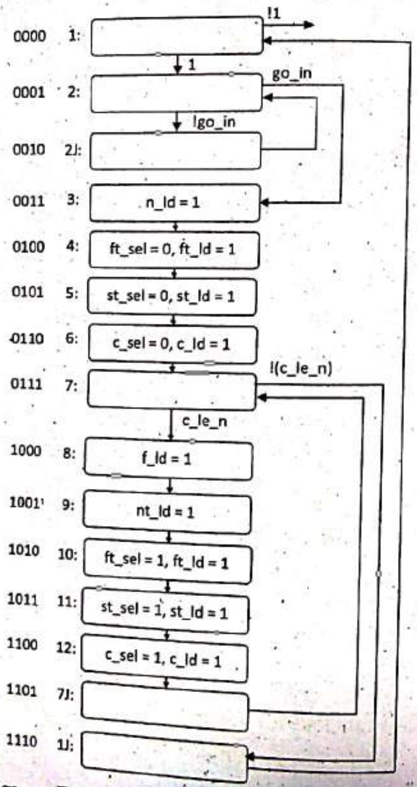


Figure: FSM of Fibonacci series generator

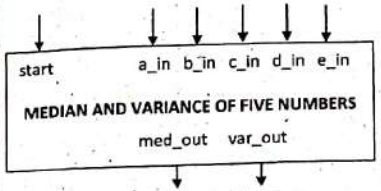
roblem 3:

Design a dual-purpose processor that calculates the median and variance of 5 numbers entered by the user by showing the algorithm, FSMD, FSM, datapath and controller design.

[2073 Magh]

olution:

A. Black Box



B. Functionality Code

```
int max, n[5], i, MD;
float mn, s, t, VR;
while(1){
while(!start);
   n[0] = a_in;
   n[1] = b_in; n[2] = c_in;
   n[3] = d_in; n[4] = e_in;
   s = 0, i = 0;
   while(i < 5){
         s = s + n[i];
         i = i + 1;
   }
   mn = s/5;
   s = 0, i =0;
   while(i < 5){
         t = n[i] – mn;
         s = s + t*t;
         i = i + 1;
   }
   VR = s/4, MD = n[2];
   m_out = MD, v_out = VR;
}
```

C. FSMD

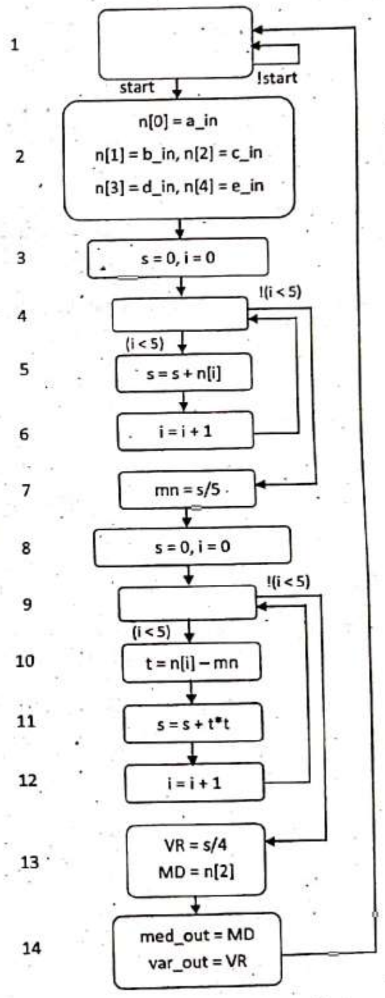


Figure: FSMD of processor that calculates the median and variance

D. Datapath

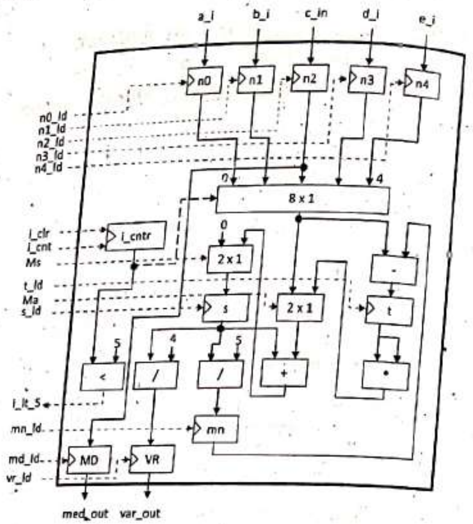


Figure: Datapath to calculate the median and variance of five numbers

E. FSM

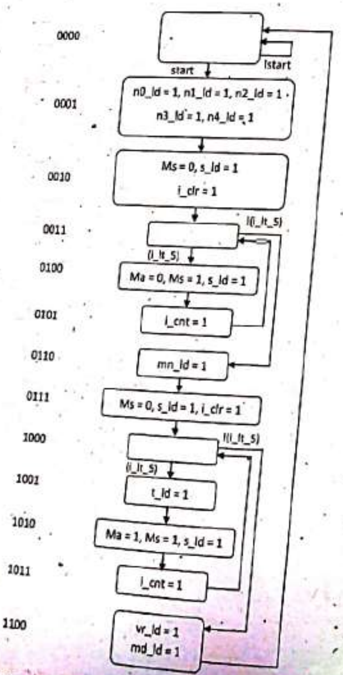


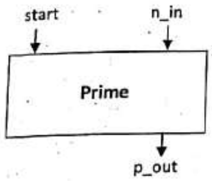
Figure: FSM of the processor that calculates the median and variance of five numbers

Problem 4:

Design a single purpose processor that checks whether an integer is prime number or not. Include FSMD, datapath, and FSM in the design. [2076, Bhadra]

Solution:

A. Black Box



B. Functionality Code

```
int n, i, c;
while(1){
 while(!start);
 n = n_in;
 c = 0;
 i = 1;
 while(i < n){
    if(n%i==0)
       c = c+1;
    i = i + 1;
 }
 p = (c==1);
 p_out = p;
}
```

C. FSMD

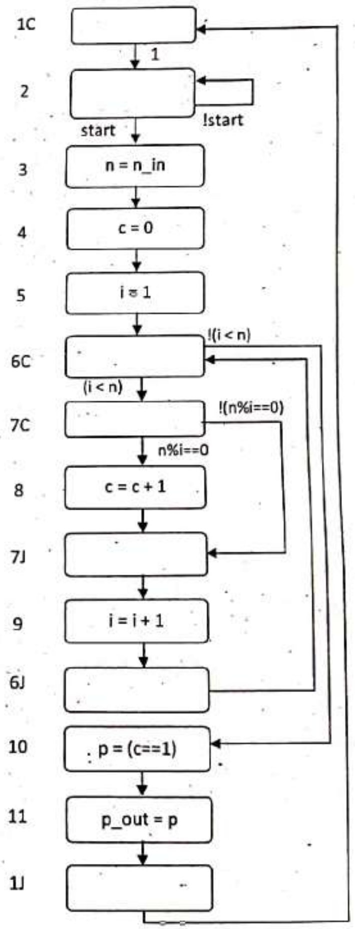
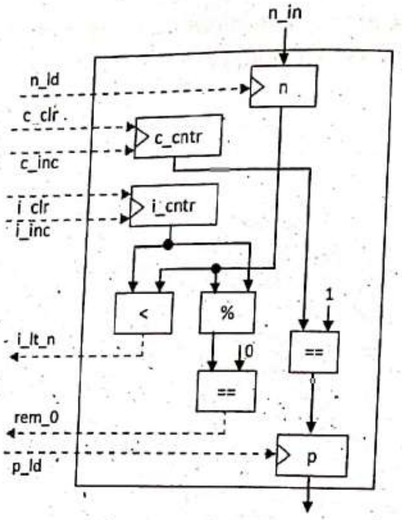


Figure: FSMD to check whether a number is prime or not

D. Datapath



E. FSM

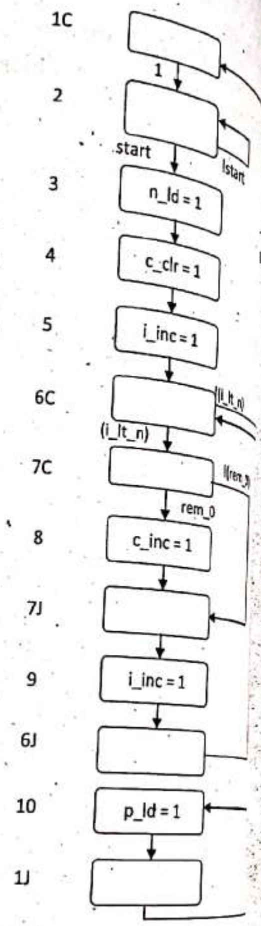


Figure: Datapath and FSM to check whether a number is prime or not

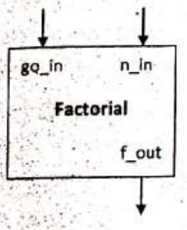
roblem 5:

Design a single purpose processor that calculates factorial of an integer. Include FSMD, datapath, and FSM in the design.

[2073 Magh]

olution:

A. Black Box View



B. Functionality Code

```
int n, f;
while(1){
        while(!go_in);
        n = n_in;
        f = 1;
        while(n>0){
            f = f * n;
            n = n - 1;
        }
        f_out = f;
}
```

C. FSMD

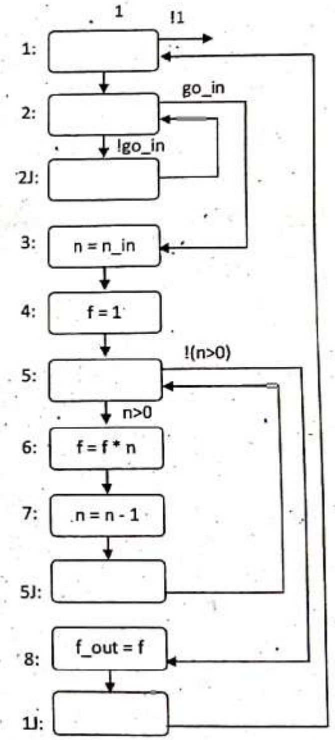


Figure: The black box view, functionality and FSMD for processor that calculates factorial of an integer.

D. Datapath for the processor that calculates factorial of an inte

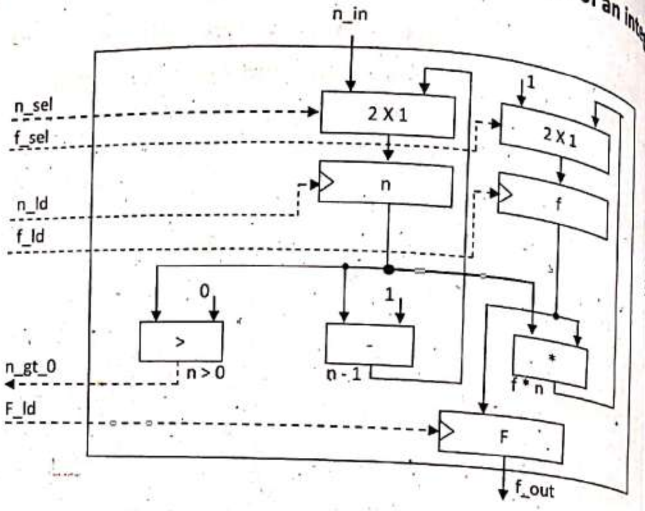


Figure: Datapath of the processor to calculate the factorial of an integ

E. FSM of the processor that calculates factorial of an integer valu

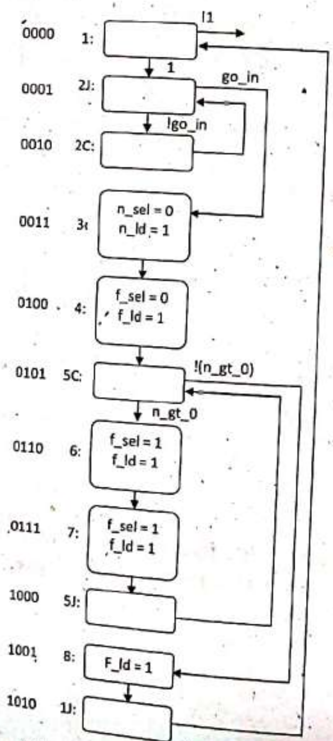
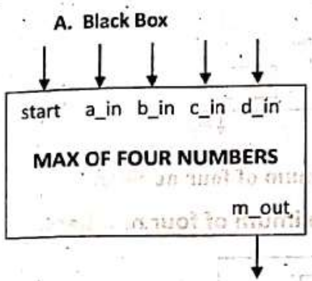


Figure: FSM controller

Problem 6:

Develop algorithm, draw the state diagram, and design the datapath of a custom single purpose processor that determines the largest of four integers. Propose the block diagram of its controller also. [2073 Bhadra]

Solution:



B. Functionality Code

```
int max, n[4], i;
while(1){
while(!start);
    n[0] = a_in;
    n[1] = b_in;
    n[2] = c_in;
    n[3] = d_in;
    max = n[0];
    i = 1;
    while(i < 4){
        if(n[i] > max)
            max = n[i]
        i = i + 1;
    }
    m_out = max;
}
```

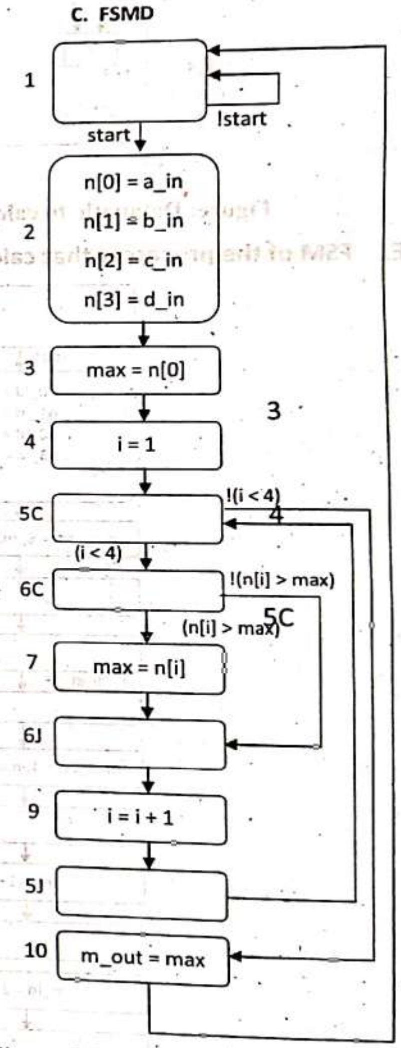


Figure: The black box view, functionality and FSMD for processor that calculates maximum of four numbers.

D. **Datapath of the processor that calculates maximum of four num**

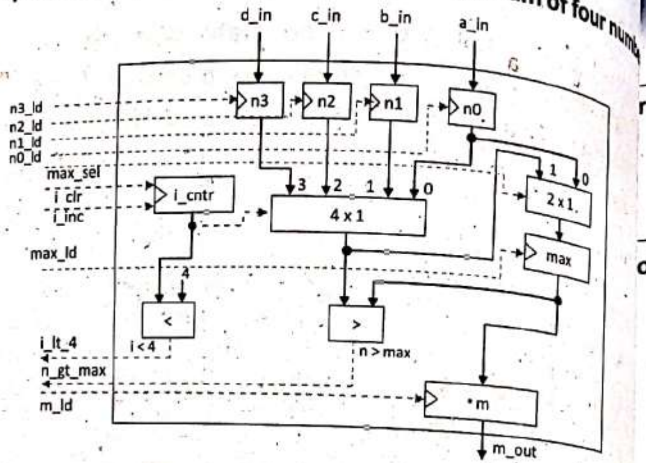


Figure: Datapath to calculate maximum of four numbers

E. **FSM of the processor that calculates maximum of four numbers.**

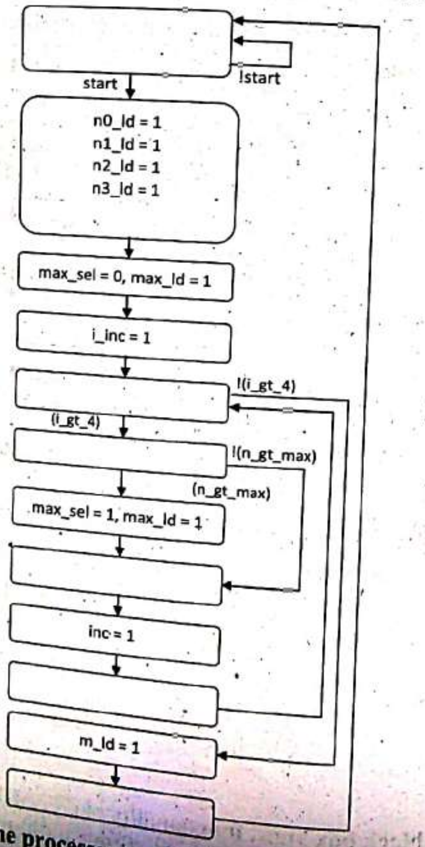


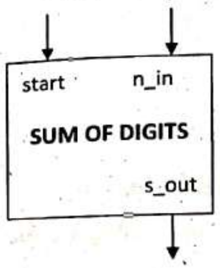
Figure: FSM of the processor to calculate maximum of four numbers

roblem 7:

Design a single purpose processor to determine the sum of digits of an integer. Start the design 'from the function computing the desired result, FSMD, datapath and controller. [2076 Baishakh]

olution:

A. **Black Box**



B. **Functionality Code**

```
int n, s, r;
while(1){
        while(!start);
        n = n_in;
        s = 0;
        while(n>0){
            r = n %10;
            s = s + r;
            n = n/10;
        }
        s_out = s;
}
```

C. **FSMD**

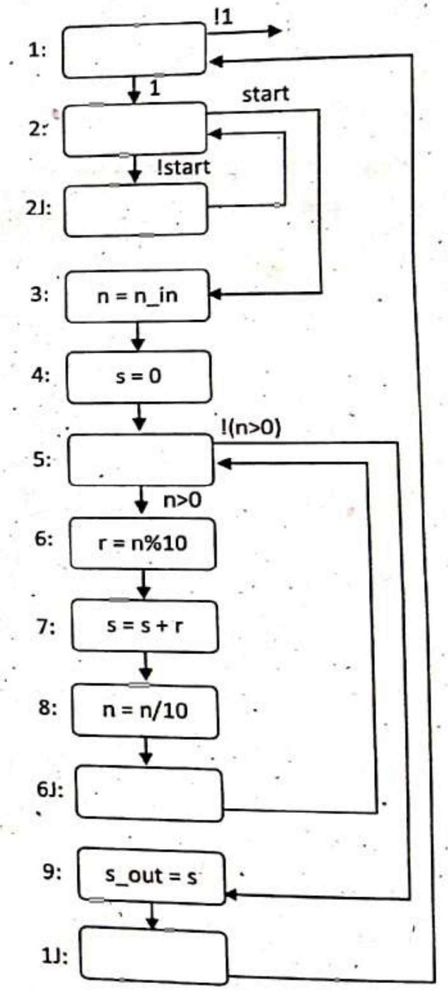


Figure: The black box view, functionality and FSMD for processor that calculates sum of digits of an integer value.

Scanned with CamScanner

D. Datapath of the processor that calculates sum of digits of an integer value.

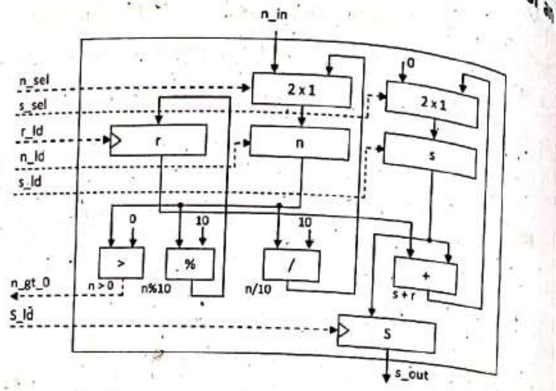


Figure: Datapath of the processor to calculate sum of digits of an integer

E. FSM of the processor that calculates **sum of digits of an integer val**

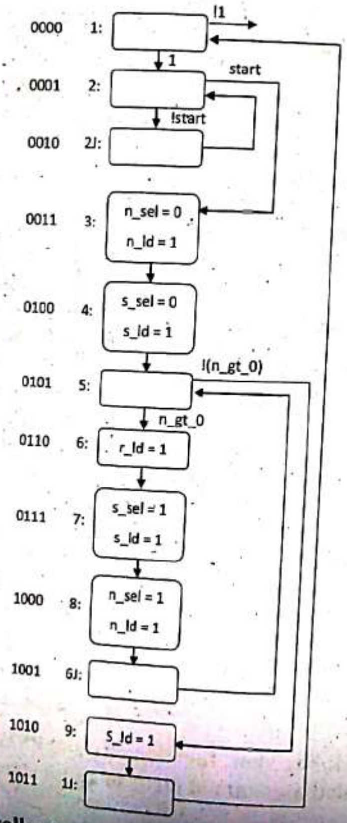


Figure: FSM controller to calculate sum of digits of an integer value

ractice Design Questions

Design a single purpose processor for the following problems. Start the design from the function computing the desired result, FSMD, datapath and controller

- to determine the largest among three numbers
- to calculate the multiplication table of integer n upto m. For example: if n = 4 and m = 15 then we need to generate the multiplication table of 4 from 1 to 15 consecutive computations.
- to generate the sequence: 1, 5, 10, 17, 26 upto n terms
- to check whether an integer number is perfect or not. (A number is said to be perfect, if the sum of all the factors excluding the number itself equals to the original number.)

Design a dual purpose processor for the following problems. Start the design from the function computing the desired result, FSMD, datapath and controller. Use optimization wherever applicable.

- to calculate the area and perimeter of an rectangle
- to determine the smallest and highest of three numbers
- to determine the maximum and minimum of five integers
- to calculate the sum of odd digits of an integer. And also check whether the calculated sum is even or odd.
- to calculate the factorial of an integer and also check whether the number is prime or not.

Scanned with CamScanner

SOFTWARE DESIGN ISSUES

- Basic Architecture
- Operation
- Programmer's View
- Development Environment
- Application-Specific Instruction Set Processors
- Selecting a Microprocessor
- General-Purpose Processor Design

3.1 Basic Architecture

A general-purpose processor is a programmable digital system w consists of a datapath and a controller which are tightly linked wi memory. Figure 3.1 shows the various components in the architecture of general-purpose processor.

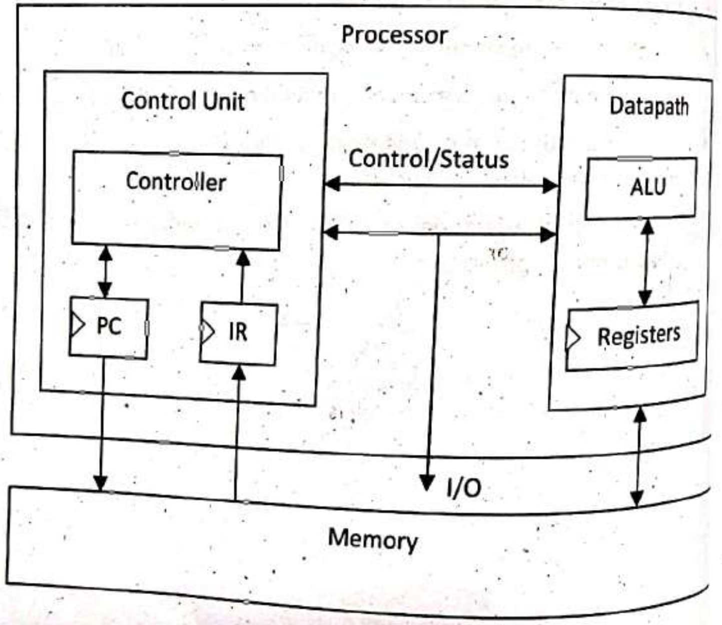


Figure 3.1: Basic architecture of general-purpose processor

instruct^n register (IR) → holds instr. that is currently being executed
decoded

Datapath

Datapath consists of the circuitry for transforming data and for temporary data storage. It contains an arithmetic-logic unit which manipulates data through various operations such as addition, subtraction, logical AND, logical OR, rotating, shifting, etc. ALU also generates status signals to represent various conditions such as carry, zero, sign, parity and so on. Such information is stored in status register.

Data path contains registers to store temporary data and different status generated by operations. The temporary data may be the data from memory for ALU to process, or the data that needs to be moved from one memory to another memory, or the data from ALU that needs further processing by ALU or needed storage. For data transfer within datapath, internal bus is used. But movement of data from and to memory is done by external bus.

2. Control Unit

The control unit consists of circuitry to general control signals to carry out various operations. It consists of controller, program counter (PC), and instruction register (IR).

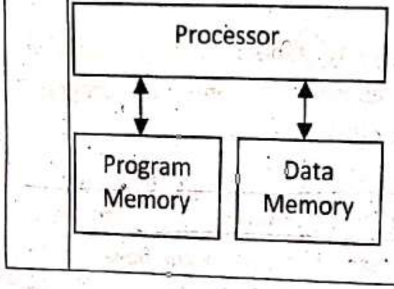
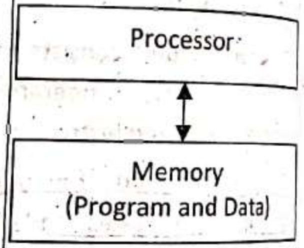
Controller consists of a state register and control logic. It sequences through the states and generates the control signals to read instructions into the instruction register, and control the flow of data between ALU, registers of datapath and memory. Controller also determines the next value of program counter. For non-branch instruction, the value of program counter is incremented. But for branch instruction, status signals from datapath and content of instruction register are evaluated for next address of program counter.

Program counter is used to hold the address of the next program instruction to be fetched, while an instruction register is used to hold the fetched instruction. The bit width of program counter indicates the address size of memory which in turn can be used to determine the number of directly accessible memory locations. For example, a 16-bit PC represents address size of 16 bits and $2^{16} = 65536$ addressable memory locations.

3. Memory

Memory is used to store information for medium, or long Information can be data or program. Program information is th of instructions that is used to carry out desired function. Data a information used by the program for various purposes.

There are two memory architectures based on program and storage.

SN	Harvard Architecture	Princeton Architecture
1.	Distinct data and program memory space	Data and program memory space
2.	Improved performance: Data and instructions can be fetched simultaneously	Data and instructions cannot fetched simultaneously
3.	It requires more connecting wires	It requires less connecting wi
4.	**Block Diagram** 	**Block Diagram** 

3.2 Operation

1. Instruction Execution

Instructions are the sets of code that carry out particular functio For each instruction, the controller sequences through several stage Each stage may consist of one or more clock cycles. The vario stages or sub-operations can be explained as:

i. **Fetch instruction:** The next instruction to be executed is loade into instruction register from memory. The address of th memory where instruction resides is given by program counter.

ii. **Decode instruction:** Instruction in the instruction register ma represent various operations based on op-code and may includ register or memory as operands. In this stage, the operation to be done by the instruction is determined.

iii. **Fetch operand:** For a given operation, operand can be a register or memory. In operations including registers, the required data are loaded into registers as specified by the instruction.

iv. **Execute operation:** The ALU handles the arithmetic and logical operations defined by the instructions. The loaded registers are fed to the inputs of ALU to carry out the operation.

v. **Store results:** The destination to store results may be either register or memory. After the execution of operation, the final data is loaded into register or memory as defined by the instruction.

Pipelining

Pipelining is implemented to increase the throughput of the system. In pipeline, the given task is divided into various stages and multiple stages which are independent of each other are executed simultaneously. For efficient instruction pipeline, different stages must be of almost same length and each instruction must require same number of cycles to complete its execution.

Branching instructions can be an obstacle for efficient pipeline as next instruction to be executed will only be known after execution stage of branch instruction. This problem, however, can be addressed using various techniques. One simple method is to stall the pipeline when there is an occurrence of branching instruction. The pre-fetch of next instructions is not done in this method rather waited for execute stage to complete first. Another popular method is to use branch prediction. In this method, the branch is guessed and the next instruction is fetched correspondingly. If the guess is correct, then it results in efficient pipeline. But, however, if the prediction is not correct then all pre-fetched instructions in the pipeline must be ignored. The following diagram shows an example of an instruction pipeline having five stages.

Scanned with CamScanner

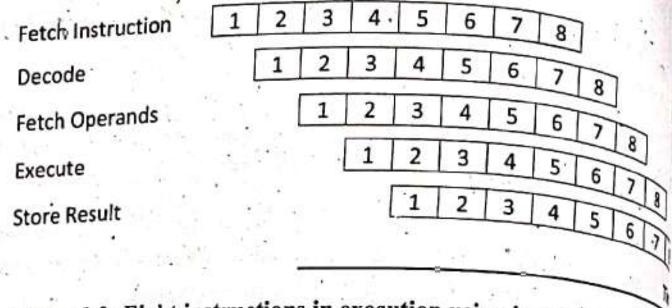


Figure 3.2: Eight instructions in execution using instruction pipeline

In Figure 3.2, there are 8 instructions in the pipeline with instruction divided into five stages and each requiring equal ti complete. In absence of pipeline, the total time required to com eight instructions would be 8x5 = 40 clock cycles, assuming stage. to complete in one cycle. However, with pi implementation, the total completion time required is 12 cycles. In this way, pipeline helps to improve the performance system.

3. Superscalar and Very Long Instruction Word (VL Architectures

Multiple ALU architecture is implemented in supers architectures to improve the performance of the system. systems can execute two or more scalar operations in parallel, w increase the requirement of ALU in the processor. It may req extensive hardware to detect multiple independent instructions can be executed simultaneously. Instructions in such architec systems are ordered statically (at compile time) or dynami (during runtime).

Very long instruction word (VLIW) architecture is a type of s superscalar architecture. It contains multiple independ instructions in a single word. Several operations are encoded single machine instruction. The compiler detects and schedules instructions.

3.3 Programmer's View

In embedded system design, the programmer must be aware of following:

Instruction Set

The instruction set is a list of instructions which represent the bit configurations for operations that can be carried out by the processor. Assembly language programmer must be aware of the available instruction set. Since embedded system design may require some portion of assembly code to be written, programmer of embedded system must know the instruction set available for the processor they are working on.

Every instruction, in general, consists of op-code and operand field. Op-code field specifies the operation to be done. An operand field specifies the location of actual data that takes part in an operation. The number of operands per instructions varies among processors and its instruction type. Addressing modes are used to represent data location and its accessing mechanism. The simple instruction format is shown in the figure below.

Op code	Operand1	Operand2

Figure 3.3: Simple two address instruction format

Commonly used addressing modes are explained in the following paragraph.

- **Immediate Addressing Mode**: The operand field contains the actual data.
- **Register Direct Addressing Mode**: The operand field contains the address of the register. And the register contains the actual data.
- **Register Indirect Addressing Mode**: The operand field contains the address of a register, which in turn contains the effective address of the actual data in memory.
- **Direct Addressing Mode:** The operand field contains the effective address of operand that is used in operation. The actual data is available in the memory.
- **Indirect Addressing Mode:** The operand field contains the address of a memory location, which in turn contains the address of a memory location where actual data is available.

Scanned with CamScanner

- **Implicit or Implied Addressing Mode:** The operand field i used in this mode; the register to be used in operat defined implicitly. In general, accumulator is used as an i register.
- **Displacement Addressing Mode:** The operand is added particular register to obtain the effective address of the da index addressing, index registers are used. While in re addressing, value of operand is added to the current addre determine the actual address.

The operations of few addressing modes can be visualized following figure.

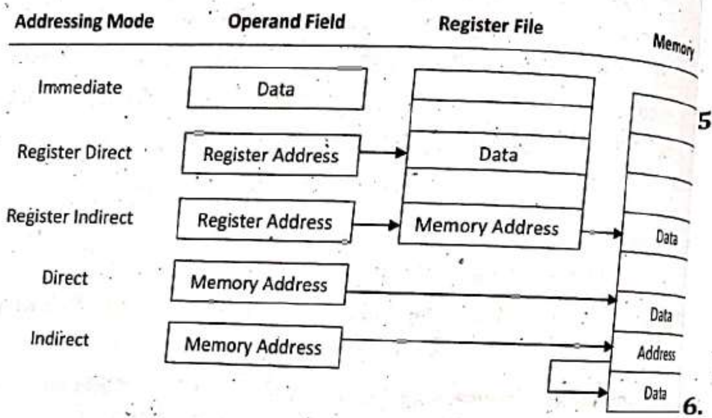


Figure 3.4: Addressing modes

2. Program and Data Memory Space

The programmer in embedded system design must be aware of t size available for program and for data. Programs must be writ within the defined memory space limits. For example, microcontrollers, the on-chip memory for program and data i fixed. So, one should be able to write the code efficiently so as not exceed the memory limit.

3. Available Registers

Programmer of embedded system design must be informed abo the number of registers available for general purpose and speci purpose. For example, multiplication in 8051 microcontroller can't done using accumulator and register B. Information abo accumulator and register B is not required for structured language programmer. However, various special function registers used for configuring timers, serial communication, and interrupts must be known to every programmer.

4. Input Output Facility

Every processor facilitates programmer with input output pins to communicate with external devices. Programmer working with processors must be alert about the number of input output pins available and their functions. In parallel I/O, port can be read or written to using specific function register. Also, communications can be done through system bus in which address and data ports can be activated by certain instructions.

5. Interrupts

Interrupt is a facility provided to the user in which the processor serves the device which requires urgent attention. It causes processor to suspend execution of the current program and starts executing interrupt service routine that does the function required by the device which interrupts the processor. The programmer should be aware of the types of interrupts supported by the processor and must write interrupt service routine when required.

6. Operating System

An operating system is a layer of software that provides low-level services to the application layer. Few services involve loading and executing of programs, sharing and allocating system resources, and synchronization mechanism. Another important service is process scheduling in which the high priority process is executed first. Other services include handling hardware interrupts, and provide device drivers.

High level applications invoke operating system using system call. When a program requires service from operating system, it generates a predefined software interrupt that is served by the operating system. Values required to the services are typically passed as the parameters in the program. CPU registers are involved for information exchange among application programs and operating system.

3.4 Development Environment

Processors along with different development tools are used for t development of software or an embedded system. Processor that is used write and debug the program is commonly referred as "developme processor". Desktop computer can be taken as an example of developme processor. Such processors may not be a part of embedded systen implementations. But the processor in which our program is loaded referred as "target processor". AVR, 8051, PIC microcontrollers or 80 8086 microprocessor can be few examples of target processor. Su processors are always a part of system implementations. Various tools the software development as well as embedded systems development described in the following paragraphs.

3.4.1 Tools for Implementation and Verification Phase

1. **Tools for Implementation Phase**

During the implementation phase, we need tools to convert hum developed code into machine readable code.

i. ***Assembler***

Assembler converts assembly instructions to binary machi instructions. It replaces op-code and operand mnemonics binary equivalent. It also translates symbolic labels into actu addresses. It generates an equivalent binary code for a sing machine instruction, so it follows one to one mapping principle

ii. ***Compiler***

Compiler converts high level programs to machine program Each high-level constructs may be translated to several machi instructions. Hence, it may not follow one to one mappir principle. Cross compilers are those compilers which run on or processor but generate the code for a processor with differe architecture.

iii. ***Linker***

Linker combines object files into a single executable file, another object file. It allows creation of a program in separate assembled or compiled files. It combines machine instruction of user code and instructions from standard library.

2. **Tools for Verification Phase**

During verification phase, we need tools or devices to test whether the code generated for target processor works as per the required functionality.

i. ***Debugger***

Debuggers are programs that are used to test and debug the targeted program. These are programs that run on development processor but execute code designed for target processors. It simulates the function of the target processors and allows evaluation and correction of programs in development processor. These debuggers are also known as instruction set simulators (ISS) or virtual machines (VM). Design cycle for debuggers is fast as compared to other tools, since the program is coded and tested in development processor. But, these tools can, however, lead to inaccuracy as it does not interact with the actual system.

ii. ***Emulator***

Emulator can be a hardware or software that enables one system to behave like another system. It consists of debugger coupled with a board connected to development processor. The board consists of target processor or device similar to target processor and support circuitry. It supports debugging of program while it executes on target processor. It also enables one to control and monitor the program's execution in actual embedded system circuit. Since the code must be downloaded into emulator hardware in each test, the design cycle is little longer compared to debugger. But it leads to accurate testing as it interacts with the rest of the system components as well.

iii. ***Device programmer***

Device programmers are the devices with the help of which binary machine programs are loaded into target processor's memory. Using this tool, the program can be tested in its realistic form which results in high accuracy as program runs on actual system. The design cycle, however, is longest since the target processor is removed from the system, programmed using programmer and returned to the system. If the device

Scanned with CamScanner

programmer can be made in-build within the system, the de cycle will be reduced.

3.4.2 Design Flow

Every software or system development process inclu implementation and verification phase. During implementation ph various implementation tools such as assembler,. compiler are used wi verification tools such as debugger, programmer are used in verifica phase.

1. Software Development Process

For a software development, the development processor as wel the target processor may be common. And the development to are available in a single package which is referred as "Integra Development Environment (IDE)".

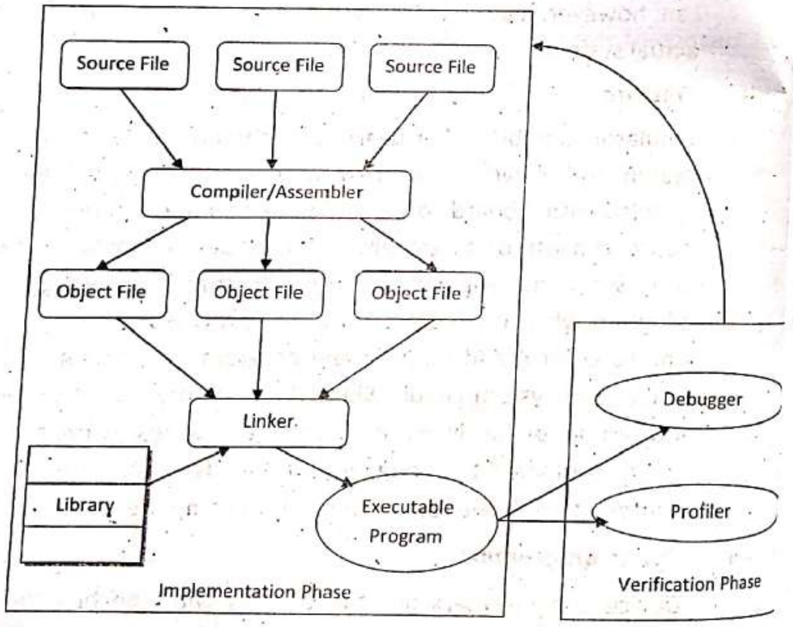


Figure 3.5: Software development process

a. **Implementation Phase:** Source code is written using an editor, a then the code is compiled/assembled using compiler/assembl Finally, with the help of linker all required files are combined into final executable file.

b. **Verification Phase:** The executable file is run under the command of a debugger. All possible inputs, especially boundary cases, are used to check the behavior of program. Profilers can be used for performance analysis of the program. Time and space complexity can be analyzed. Time complexity includes duration of execution of program whereas space complexity includes memory usage.

2. Embedded System Development Process

In case of embedded system design, the target and development processors are different in almost all systems. The Integrated Development Environment (IDE) tools for various processors are available for implementation phase. Though the implementation phase for embedded system is similar to that of software implementation phase, the verification phase differs drastically.

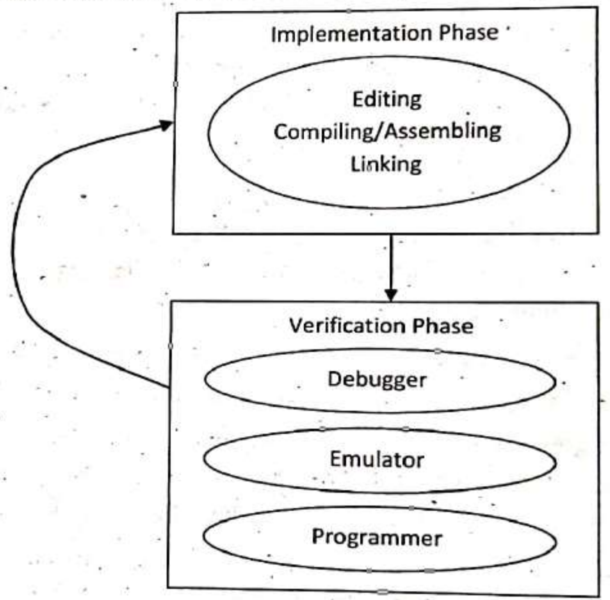


Figure 3.6: Embedded system development process

a. **Implementation Phase:** The process of editing, compiling/ assembling and linking the program is same as that of software development process. However, development processors use cross compilers or cross assembler. As those compilers run on development processor, for example PC, and generate the file for target processor, for example hex file for microcontrollers.

Scanned with CamScanner

b. **Verification Phase:** Embedde njunction other components as well as with real time environmen debugging a program requires control over time and environ Based on requirement and availability, debuggers, emulato device programmers can be used for verification. Code ma simulated on development processor using debuggers or code be checked by loading into emulator hardware. Also, program can be used to load the code directly into the target processor.

3.5 Application-Specific Instruction Set Processors

Application-specific instruction set processors are specific to particular application domain. They can be programmed based requirement of particular arena, which makes it more flexible. Also constraints such as performance, power, cost, and size are efficient en to develop a system. However, instruction set processor and its associ software tools are expensive to develop. It can be categorized microcontrollers, digital signal processors and less general applica specific instruction set processors.

1. Microcontrollers

Microcontrollers are specific to applications that perform a l amount of control oriented tasks. The following are few gen features of microcontrollers.

- It includes several peripheral devices such as timers, analog digital converters, serial communication devices, and so on.
- It generally contains program and data memory on the same Various peripherals along with memory incorporated within same IC result in compact and low-power implementation.
- It provides the programmer direct access to number of pins the IC. Access to pins enable programmer to interface w other devices such as sensor, actuators, LCDs, and ot different devices that may be used in the system.
- Some specialized instructions may be available. Such faci improves the performance of the system.

2. Digital Signal Processors (DSP)

These are processors which are specific to applications that proc large amounts of data. The source of large amount of data inclu

image captured by a camera, voice packet through a network router, audio clip played by an instrument. Few features, out of many, are listed below.

- It may contain numerous register files, memory blocks, multipliers and other arithmetic units.
- It facilitates with instructions that are applicable uniquely to digital signal processing. Filtering and transforming vectors can be two examples.
- Frequently used arithmetic functions are implemented using hardware. It results in faster execution of arithmetic functions compared to software implementation.
- Some special digital signal processors allow concurrent execution of functions which boost the performance of the system.
- It incorporates many peripherals specific to signal processing. It may include ADC, DAC, PWN, DMA controllers, timers, and counters.

Advantages:

- **DSP provides flexibility**: Digital signal processing operations can be changed by changing the program in digital programmable system.
- **DSPs are less susceptible**: The digital circuits are less sensitive to tolerances of component values.
- **DSP improves performance**: It has a better control of accuracy in digital systems compared to analog systems.
- **DSP supports complex operation**: Sophisticated signal processing algorithms can be implemented by DSP method.

3. Less-General ASIP

These are developed to perform some very domain specific processing while allowing some degree of programmability. Processors designed for networking hardware can be taken as an example of less-general ASIP.

Scanned with CamScanner

3.6 Selecting a Microprocessor

In any embedded system, a designer must select the microproce based on technical and nontechnical aspects.

- **Technical aspects:** Selection of processor must be done base required speed within limited power, size, and cost.
- **Non-technical aspects:** Before selecting microprocessor, one mu aware of development environment, prior expertise of proce licensing arrangements and so on.

Comparing Speed

Speed of processors can be measured and compared using va methods.

i. **Clock speed of processor**

Speed can be compared based on clock speeds of processors, but number of instructions per clock cycle may differ. So, it may no an efficient method unless processors to be compared have s number of instructions per cycle.

ii. **Instruction per second**

The speed can be evaluated using number of instructions execu per second. But the complexity of available instruction sets r differ creating some hindrance in speed comparison. For example perform same operation, one processor may require 200 instructi while another may require 300 instructions.

iii. **Dhrystone benchmark**

It is a program that runs on different processors and evaluates th performance based on execution of certain operations. Dhrysto benchmark performs no useful work rather checks the inte arithmetic and string-handling capabilities of the processor on wh the benchmark runs on. Since processors can execute su operations thousands of times in a second, speed of processor m be expressed in terms of Dhrystones per seconds.

iv. **Millions of instruction per second (MIPS)**

It is a general measure of computing performance and the amount work a processor can do. MIPS can be useful when compari performance of processors having similar architecture. The origin

MIPS is based on VAX 11/780 which could execute one million instructions per second or could execute 1757 Dhrystones per second. Hence, 1 MIPS = 1757 Dhrystones/sec. Also, performance of other computers were measured based on VAX 11/780.

3.7 General-Purpose Processor Design

General-purpose processor can be designed using the design technique of single-purpose processor as general-purpose processor is a type of single-purpose processor which process instructions stored in program memory. The design starts with the design or selection of instruction set, followed by creating a FSMD, and then datapath along with its connections with control unit and finally a controller or FSM is developed.

EXAMPLE: Design a general purpose processor with four data transfer instruction, two arithmetic operations and one jump instruction.

The following are the considerations made in the design.

- 16 bit instruction size, which has direct impact on memory and register selections.
- Instruction Register (IR) and Program Counter (PC) of 16 bit,
- Memory of 64K x 16 bit,
- Register file of 16 x 16 bit

1. Instruction Set Selection

Instruction	First Byte		Second Byte		Operation
MOV Rn, direct	0000	Rn	Direct		Rn = M(direct)
MOV direct, Rn	0001	Rn	Direct		M(direct) = Rn
MOV @Rn, Rm	0010	Rn	Rm		M(Rn) = Rm
MOV Rn, #imm	0011	Rn	Immediate		Rn = immediate
ADD Rn, Rm	0100	Rn	Rm		Rn = Rn + Rm
SUB Rn, Rm	0101	Rn	Rm		Rn = Rn – Rm
JZ Rn, relative	0110	Rn	Relative		PC = PC + relative (if Rn is 0)

Figure 3.7: A simple instruction set

From the above instruction set, the various means of data tra and operations can be analyzed which may be useful in devel FSMD and datapath.

- The address of memory location are available from
  - **Instruction register:** In instruction MOV Rn, direct and direct, Rn, the direct address is used which is available as lower bytes.
  - **Register:** In instruction MOV @Rn, Rm, the addre memory is given by value of register.

(Address is also given by PC to load the instruction into IR)

- The value in register can be loaded from:
  - **Memory:** In instruction MOV Rn, direct, register is lo from memory whose address is given by lower eight bi IR.
  - **Instruction register:** In instruction MOV Rn, #imm, immediate value of IR is loaded into register.
  - **ALU:** After execution of ADD Rn, Rm and SUB Rn, Rm, final result is stored in register.
- Three operations are performed by ALU
  - Addition, subtraction, and comparison

2. FSMD for Given Instruction Set

In FSMD, the basic stages of instruction cycle are implemented states. It includes RESET, FETCH, DECODE and EXECUTE state. RESET, FETCH and DECODE states are common to almost ev design. The EXECUTE state, however, differs when the number type of instructions are different.

Aliases:

op – IR[15..12]
rn – IR[11..8]
rm – IR[7..4]
dir – IR[7..0]
imm – IR[7..0]
rel – IR[7..0]

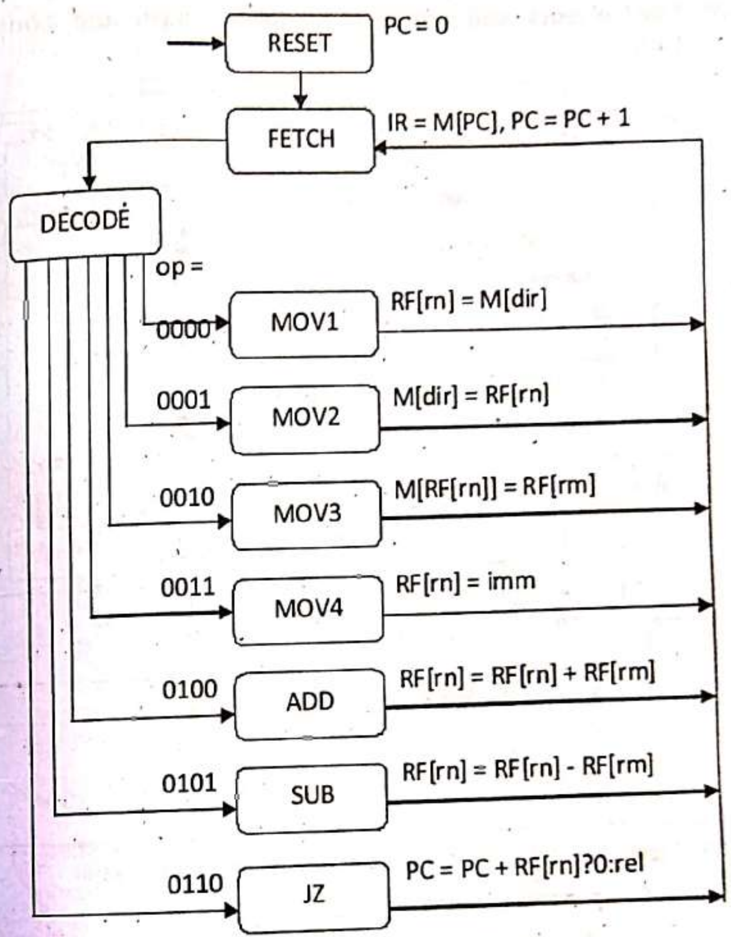


Figure 3.8: Finite state machine with data (FSMD)

3. Components and Connections in Datapath and Con Unit

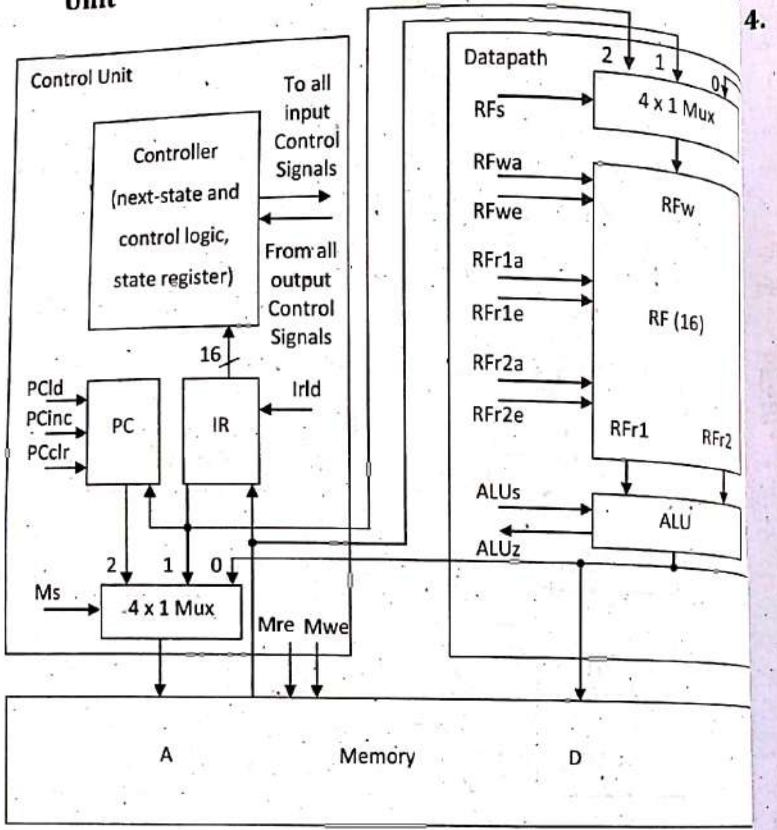


Figure 3.9: Datapath of our simple general purpose processor

i. **Components in datapath**

- Register file of 16x16 and a general purpose ALU.
- Multiplexer of 4x1, since the register in register file ca have three sources; Immediate data from IR, data fro Memory, and data from ALU

ii. **Components in control unit**

- Controller for next-state and control logic, state registe Program Counter, Instruction Register
- Multiplexer of 4x1, since memory address can be selecte from three sources; address from PC, direct address fro IR, and address from register.

4. Finite State Machine (FSM) Design

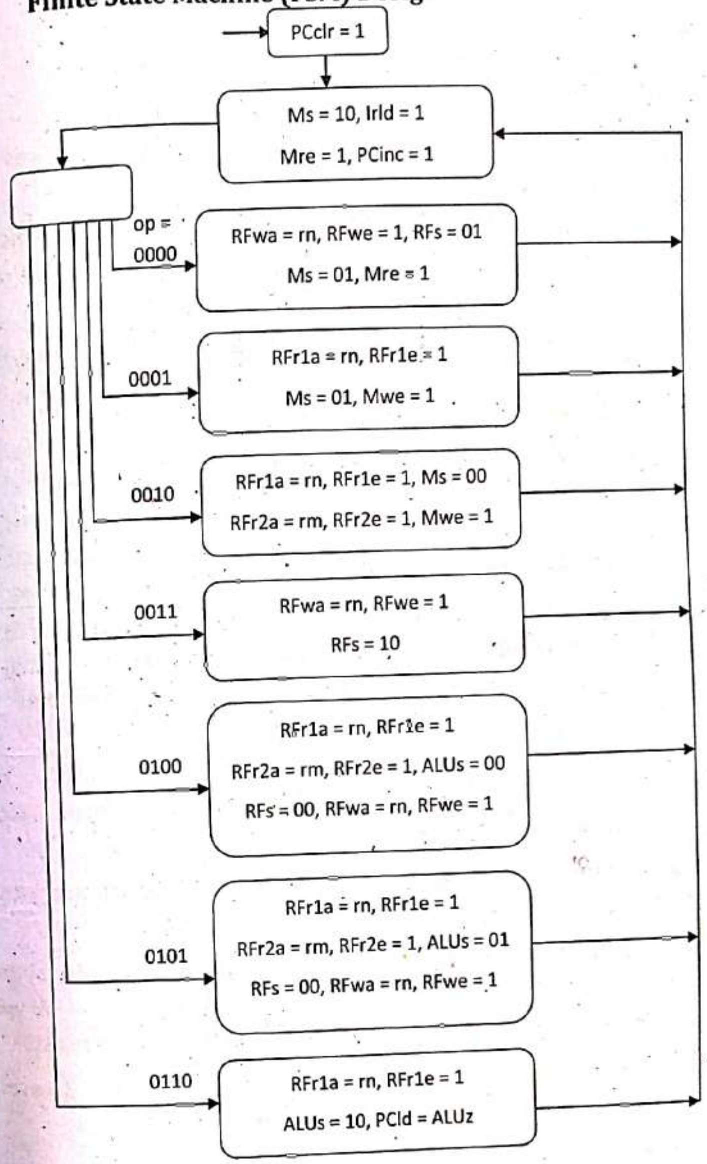


Figure 3.10: Finite state machine (FSM)

Scanned with CamScanner

Converting FSMD operations to FSM operations

Example 1: MOV Rn, direct → RF[rn] = M[dir]

It means to read the content of memory of address dir (8 lower of IR) and write it into one of registers of register file. Value of rn gives address of register in register file.

- Address of memory is directly available in IR, using multipl selection Ms = 01 will select address from IR. For a memory r operation, Mre must be set (Mre = 1).
- The value is to be written into register file, so RFwa = rn selec register from register file and RFwe enables the write operation. RFs = 01, as data is coming from memory.
- Hence, the required Boolean expressions are: Ms = 01, Mre = RFwas = rn, RFs = 01

Example 2: ADD Rn, Rm → RF[rn] = RF[rn] + RF[rm]

Here, values from two registers are read and then added using Al The final result is stored in register. Address of registers to be selected given by rn and rm for read operation while value of rn gives the address register for write operation.

- Selection of registers for read operation: RFr1a = rn and RFr2a = r select two registers while RFr1e = 1 and RFr2e =1 enable bo registers for read operation.
- Adding the value of registers using ALU: ALUs = 00 represent th addition of two registers.
- Selection of register for write operation: RFwa = rn selects th register and RFwe = 1 enables the write operation. And RFs = 00 wi connect the output of ALU through the mux to the selected register.
- Hence, the required Boolean expressions are: RFr1a = rn, RFr2a = rm RFr1e = 1, RFr2e = 1, ALUs = 00, RFs = 00, RFwa = rn, RFwe = 1

Chapter 4

MEMORY

- Introduction
- Memory Write Ability and Storage Permanence
- Common Memory Types
- Composing Memory
- Memory Hierarchy and Cache

4.1 Introduction

The functionality of any embedded system can be basically divided into processing, storage and communication. And memory is required to address the storage aspect of the embedded system's functionality. Furthermore, memory is an electronic device that is used for the retention of information for later use. Here, information can be program instructions or data for computational purposes. And, the instructions represent the group of bits of operation code along with operands, while data indicates the values of operand that will be used in the operation. In general, the instructions are stored in a type of memory which we refer as read only memory. And the data for regular computation are temporarily stored in registers.

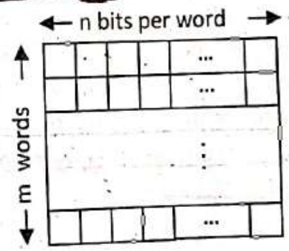


Figure 4.1: m×n memory

A memory stores information in form of large numbers of bits. And bits are combined to form a word. Also, words are stacked together to represent a memory. So, we refer to a memory as m words of n bits each. And it is represented by m x n. For example, if a memory is specified as 4096 x 8 then it holds the following information.

- Number of words in the memory (m): 4096
- Number of bits per word (n): 8

- Total bits (m x n): 32768 bits
- Number of Address lines = $\log_2(m) = \log_2(4096) = \log_2(2^{12}) = 12$
- Total input/output data signals: 8

A memory access may refer to memory read (retrieve the word (particular address) or memory write (store a word in a particular addre Control input signal r/w is used to indicate the type of access. Another con input signal, enable, which when asserted, is used to access the memory.

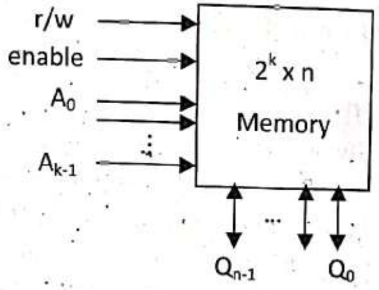


Figure 4.2: External view of memory

4.2 Memory Write Ability and Storage Permanence

1. Write Ability

Write ability refers to the manner and speed that a particu memory can be written. It also can be used to represent the number times a memory can be programmed or written into. In-syste programmable is used to categorize memories into two along the wri ability axis. In-system programmable memory can be programmed by processor whereas non in-system programmable memory must b programmed by some external means.

Range of write ability:

- **High End** – processor can write to memory simply and quickly b setting its address lines, data input bits and control line appropriately. Example: RAM
- **Middle Range** – processor can write to memory a bit slowe compared to high end. Example: EEPROM, FLASH
- **Lower Range** – special device called programmer is used to write into the memory. The device must apply suitable voltage levels to write to the memory. E.g.: EPROM, OTP ROM

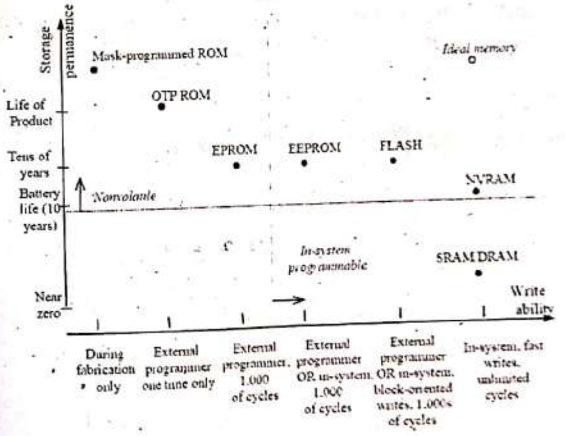
- **Low End** – bits are stored only once during fabrication. Example: Mask-programmed ROM

2. Storage Permanence

Storage permanence refers to the ability of memory to hold its stored bits after those bits have been written. Volatile and Nonvolatile attributes are commonly used to divide memory types into two categories along the storage permanence axis. The nonvolatile memory can hold its bits even after power is no longer supplied. On the contrary, volatile memory requires continual power to retain its data.

Range of storage permanence:

- **Low End** – memory in this range begins to lose its bits almost immediately after those bits are written and therefore it must be refreshed periodically. *Example*: DRAM
- **Lower Range** – memory holds bit as long as power is applied to the memory. *Example*: SRAM'.
- **Middle Range** – memory in this range holds bits for days, months, or even years after the memory power source has been turned off. *Example*: NVRAM
- **High End** – memory in this end will never lose its bits, as long as the memory chip is not damaged. *Example*: Mask Programmed ROM



Write ability and storage permanence of memories,- showing relative degrees along each axis (not to scale).

Figure 4.3: Various memories based on write ability and storage permanence

Scanned with CamScanner

4.3 Common Memory Types

1. Read Only Memory (ROM)

It is a nonvolatile memory, that can be read from but canno written to, by a processor, but it can be programmed by setting bits within the memory. Traditionally, ROM is programmed off when it is not actively involved within an embedded system.

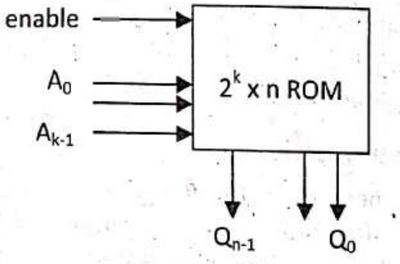


Figure 4.4: External block diagram

i. Uses of ROM

- to store a software program for a general purpose processor
- to store constant data, like large lookup tables of strings numbers
- to implement a combinational circuit.

ii. Internal View of the ROM

It consists of an address lines, word lines, data lines, control li decoder, OR gate and programmable connections which conn word line and data line.

Example: Internal view/sketch/design of an 8x4 ROM

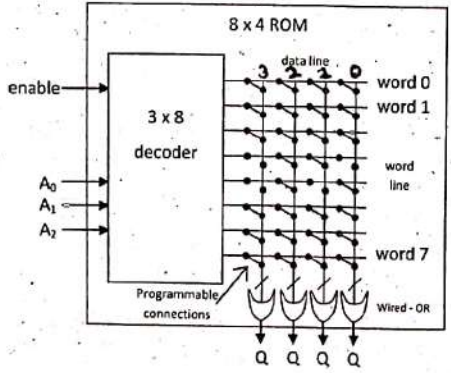


Figure 4.5: Internal view of an 8×4 ROM

- Decoder selection: number of words is 8, so at least a 3x8 decoder is required. Number of output lines of decoder must be equal or greater than number or words of ROM.
- Horizontal lines = words (8), Vertical lines = data (4)
- Word line connected to data line via the programmable connections
- Circles on data and word lines are connected to represent high logic(1)
- Wired-OR represents all word lines are ORed together.
- If word 3 needs to be read then the input of decoder is set to 011 which makes the word 3 line high and other word lines low, since the data lines 0 and 3 are not connected to the high word 3 line, the output of the ROM will be 0110.

iii. Implementation of Combinational Functions using ROM

Example:

Combinational functions: y = a'b'c' + a'bc' + ab'c + abc, z = a'b'c + a'bc' + a'bc + ab'c + abc

Solution:

Three inputs a, b and c is taken as address lines. So, for three inputs the decoder of 3×8 must be used resulting in eight word lines. And there are two outputs, so there must be two data lines. Hence, a ROM of 8×2 is required. Initially, if only combinational function is given, then its truth table should be formed. The programming connections are done based on the output of the functions based on the various combinations of inputs.

Inputs			Outputs	
a	b	c	y	z
0	0	0	1	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

Scanned with CamScanner

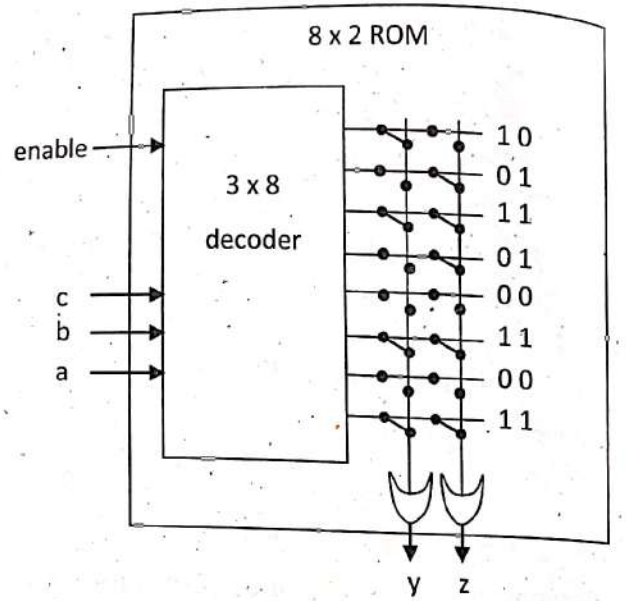


Figure 4.6: Truth table for given function and its implementation using ROM

iv. Types of ROM

a. Mask-Programmed ROM

- Connection is programmed during fabrication, by creat an appropriate set of masks.
- It has extremely low write ability. Once fabricated, content cannot be reprogrammed or changed.
- It has highest storage permanence. Stored bits will nev change unless the chip is damaged.
- It is used in such embedded systems whose design h been finalized and large numbers of unit are needed to b manufactured.

b. One-Time Programmable ROM – OTP ROM

- Connection is programmed using a device calle programmer that configures each programmab connection according to the file provided by use Programmer blows fuses by passing a large curre wherever a connection is not required. The blown ou fuses cannot be reestablished, hence it is referred as on time programmable ROM.

- It has lowest write ability of all PROMs, since it can be programmed only once.
- It has very high storage permanence, since its stored bits won't change unless; some more fuses are blown out using programmer.
- It is cheap which makes it more suitable in final products compared to other types of PROM.

c. Erasable Programmable ROM – EPROM

- EPROM uses a MOS transistor as its programmable component. The transistor has a floating gate surrounded by insulator. When high voltage (12V – 25V) is applied, it causes electrons to tunnel through the insulator into the gate. When the high voltage is removed, the electrons cannot escape and hence the gate has been charged and programming has occurred. To erase the program, the electrons must be excited enough to escape the gate which is done by exposing UV light for 5 – 30 minute. For the UV light to reach the chip, EPROMs are provided with a small quartz window in the package.
- Reading an EPROM is much faster than writing, since reading doesn't require programming.
- EPROMs have improved write ability and can be reprogrammed thousands of times.
- EPROMs have reduced storage permanence. They hold their stored bits for about 10 years.
- Electrical noise or radiations causes stored bits of the chip subject to undesirable changes and hence EPROMs are scarcely used in production. It offers a better choice in the testing phase of the system rather than in production.
- Internal Operation of EPROM:
  a) Negative charges form a channel between source and drain storing a logic 1
  b) Large positive voltage at gate causes negative charges to move out of channel and get trapped in floating gate storing a logic 0

c) Shining UV rays on surface of floating gate ca negative charges to return to channel from floa gate restoring the logic 1

d) An EPROM package showing quartz window thro which UV light can pass

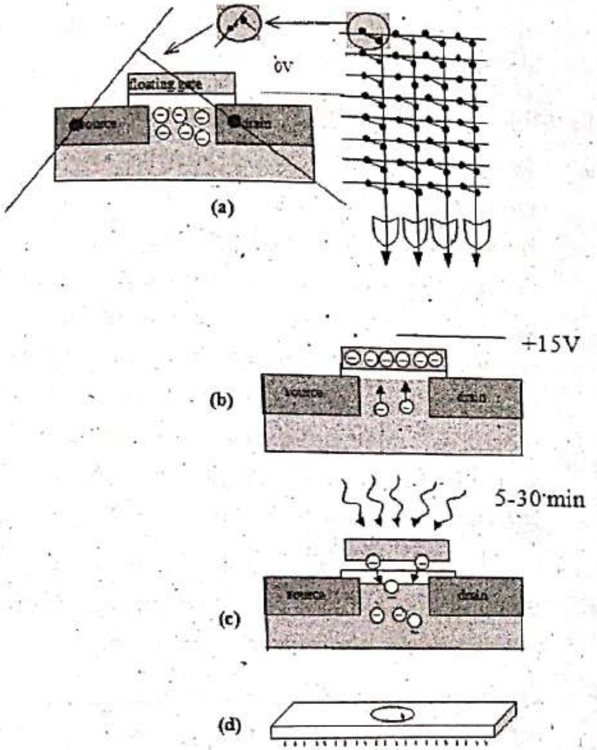


Figure 4.7: Writing and erasing process in EPROM

d. Electrically Erasable Programmable ROM – EEPROM

- EEPROM is programmed and erased electronically, usi higher than normal voltage. Electronic erasing requir seconds, rather than many minutes required for EPROM Moreover, individual words can be erased ar reprogrammed in case of EEPROM, whereas EPROM ca only be erased in their entirety.
- It is in-system programmable since circuit providing highe than normal voltage levels for erasing and programming built into the embedded system. EEPROM is built with built in memory controller which hides internal memor access details for the memory user and provides a simpl memory interface to the user. The memory controlle contains the circuitry and single purpose processor required to erase and program the word at the user specified address.
- EEPROM provides better write ability compared to EPROM, it can be reprogrammed tens of thousands of times.
- EEPROM has storage permanence on a par with EPROM, about 10 years.
- Writing is slower, since it involves the process of erasing and programming. An extra busy pin is available to indicate that the EEPROM is busy in writing.
- EEPROM can be used to serve as the program memory for a microprocessor. It can also be used to store data than an embedded system should save after the system is off.

e. Flash Memory

- It is an extension of EEPROM which uses the same floating-gate principle along with same write ability and storage permanence.
- It improves the performance of a system with its fast erase ability, in which large blocks of memory can be erased all at once.
- Writing to a single word in flash may be slower than writing to a single word in EEPROM, since an entire block will need to be read, updated and written back.

2. Random Access Memory – RAM

It is a memory that can be both read and written easily. Typically RAM is volatile, since it loses its content after the power is removed. The internal structure of RAM is comparatively complex than of ROMs.

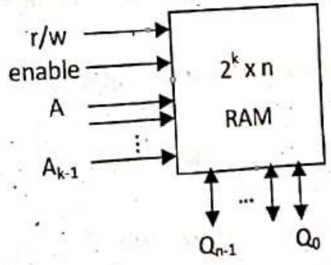


Figure 4.8: External view of RAM

Scanned with CamScanner

I. Internal Structure of RAM

It consists of address lines (A), word lines (output of decoder), lines for reading (Q) and writing (I), control lines (enable & decoder, wired-OR, and cell at each intersection of word lines, data lines.

Example: Sketch the internal structure of a 6 x 6 RAM

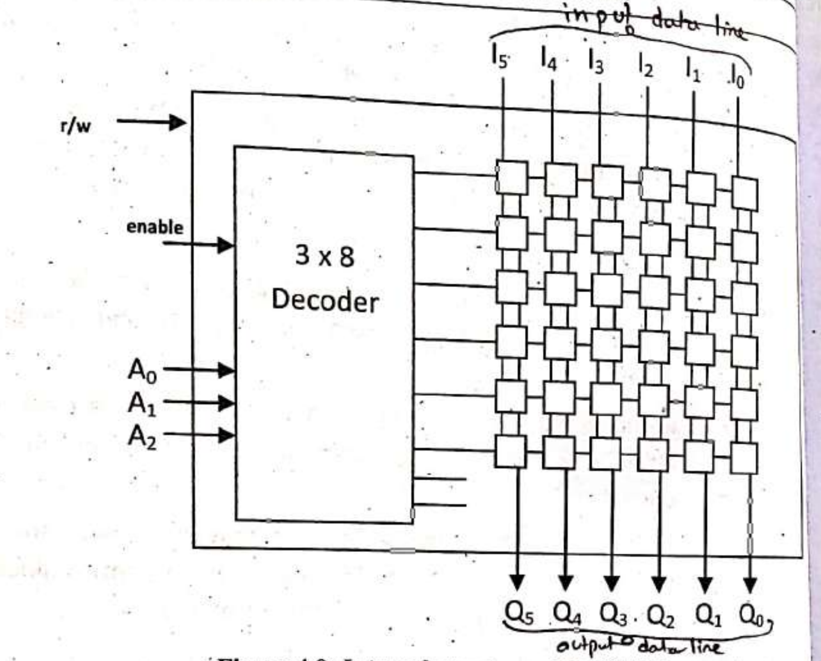


Figure 4.9: Internal structure of 6×6 RAM

- Decoder selection is done based on number of words of RA (3x8 decoder for 6 words)
- Each word consists of a number of memory cells, each stori one bit.
- Each input data line and output data line is connected to eve cell in its column.
- Output of a memory cell being ORed with the output data li of each column.
- The read/write input is connected to every cell
- Wired-OR is not shown in the figure 4.8.

ii. Types of RAM

a. Static RAM – SRAM

- It uses a memory cell consisting of a flip flop to store a bit
- It requires about six transistors to represent a single bit
- It holds data as long as power is supplied hence called static RAM.
- Generally used for high-performance section of a system. E.g., Cache memory

b. Dynamic RAM

- It uses a memory cell consisting of a MOS transistor and capacitor to store a bit
- It requires only one transistor, resulting in more compact memory than SRAM
- Each cell must be charged (refreshed) regularly, since the charge stored in capacitor leaks gradually causing the loss of data.
- DRAM access tends to be slower than SRAM, since accessing a DRAM word results in the word's being stored in a buffer and then being written back to the word's cell.

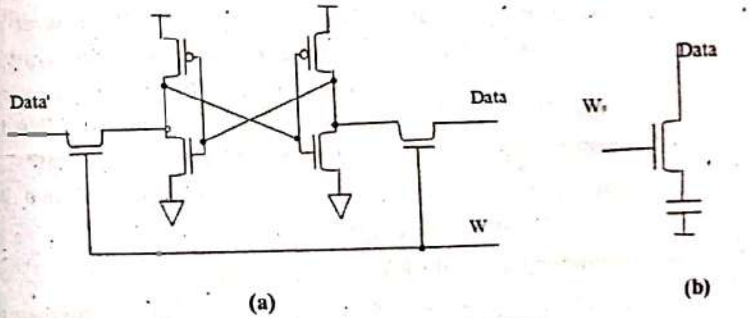


Figure 4.10: (a) SRAM (b) DRAM

c. Pseudo-Static RAM – PSRAM

- These are DRAMs with a memory refresh controller built in.
- PSRAM may be busy refreshing itself when accessed, which could increase access time and add some system complexity.

Scanned with CamScanner

- It is a popular low-cost high-density memory alternative SRAM.

d. **Nonvolatile RAM – NVRAM**

It holds data even after external power is removed.

- **Battery-Backed RAM:** Contains a static RAM w permanent battery connected. When power is removed drops below a certain threshold, the internal batte maintains power and the memory continues to store bits. There is no limit on the number of times the Batter Backed RAM can be written to.
- **Static RAM with EEPROM or FLASH:** This type of NVRAM stores its complete RAM contents into the EEPROM ju before the power is turned off. The data is reloaded in RAM after the power is turned back in.

iii. Advanced RAM

a. **Fast Page Mode Dram (FPM DRAM)**

FPM DRAM is asynchronously controlled which is designed wi some improvements on the basic DRAM architecture. In th design, each row of the memory bit-array is viewed as a pag which contains multiple words. Each word is addressed by unique column address. In its operation, first the row or pag address is sent and then the corresponding column addres must be sent to read a particular word. In each memory cycl three data words can be read consecutively by providing the corresponding column address. Hence, it eliminates th requirement of extra cycle as three cycles would have bee required to read three words.

b. **Extended Data Out DRAM (EDO DRAM)**

EDO DRAM is similar to FPM DRAM with additional feature tha reduces the read/write latency. Here, new access cycle can b started while keeping the data output of previous cycle active In simple words, new column address can be sent while readin previously selected word from the memory. This results i overlapping of the operation which reduces the latency o memory access. However, extra output latch must be introduced in the architecture.

c. **Synchronous DRAM (SDRAM)**

In SDRAM, the information is latched to and from the controller on the active edge of the clock signal. The time required to detect the strobe signals in asynchronous DRAM is eliminated by SDRAM. This DRAM architecture can have additional column address counter which holds the starting address of the data to be accessed. This counter is incremented internally to provide new data in each clock cycle as long as the data required are consecutive memory locations. The enhanced synchronous DRAM (ESDRAM), is the improved version of the SDRAM. ESDRAM provides faster clocking and lower latency in reading and writing data.

d. **Rambus DRAM (RDRAM)**

Rambus represents the bus interface architecture which uses multiplexed address/data lines to connect the processor to the RDRAM device. RDRAM may be further divided into number of banks with each remain open for access. Multiple open page scheme and fast bus I/O can result in high throughput. However, as compared to other standards, Rambus showed increase in latency, heat output, complexity, and cost. Requirement of heat-spreaders along with packet de-multiplexors makes it more complex while manufacturing. More complex interface circuitry and more number of memory banks increased the size and resulted to become expensive.

e. **Double Data Rate SDRAM (DDR SDRAM)**

The DDR SDRAM is capable of making higher transfer rates with more strict control of the timing of the data and clock signals. The interface transfers data on both the rising and falling edges of the clock signal to double the data bus bandwidth. DDR SDRAM also known as DDR1 was replaced by DDR2 which operated on same principle but for higher clock frequency and produced double throughput as compared to DDR1. Similarly, DDR3 and DDR4 offered better performance for increased bus speed and new features.

3. Memory Management Unit (MMU)

Memory management unit is a processor which translates the logical address to physical memory address. MMU has important role in

Scanned with CamScanner

handling DRAM refresh, bus interface and arbitration used memory. In addition, it takes care of memory sharing among multi processors. Contemporary CPUs have built-in MMU as a part processor.

4.4 Composing Memory

Composing memory is needed when there is a need of particul sized memory, which is not readily available. If the available memory larger than required one, then we simply use the needed lower words of t memory and ignore the higher words which are not required. However the available memory is smaller than needed, some more design procedu are needed to be followed. The various cases for composing memory ha been discussed in the following paragraphs.

1. Case 1: To increase the width of words

When the number of words in the available memory is same to th of required one but the number of bits or width of word is n enough then the width must be increased. To do that, the availab memories are connected side by side as shown in the given exampl

Example 1: Compose 1K×8 ROMs into a 1K×32 ROM

Analysis: The available ROM 1K×8 and required ROM of 1K×32 ha same number of words but width is different. The number of ROM t be placed side by side is given by n.

- n = width of required ROM/width of available ROM = 32/8 = 4
- Address line = 1K = 1024 bytes = 2^{10} = 10 address lines
- Data line = 8 lines (4 8-bits of data will result to 32 bits of total d

Hence, four 1K x 8 ROMs are placed side by side to compose 1K x 32

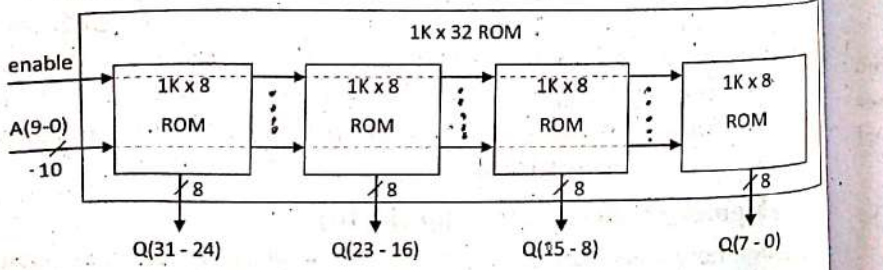


Figure 4.11: Composing 1K×32 ROM from 1K×8 ROM

2. Case 2: To increase the number of words

When the width of the word in the available memory and required memory is same but the number of words are different then the words must be increased. We connect the ROMs top to bottom and data line of each ROM is ORed. Since the number of words has to be increased, extra high-order address is required to select the particular ROM which can be implemented by using appropriate decoder.

Example 1: Compose 1K×8 ROMs into a 4K×8 ROM

Analysis: The available ROM 1K×8 and required ROM 4K×8 have same width of 8 bits but the number of words is different. Number of ROMs and the size of decoder can be determined as

- N = number of words in required ROM/number of words in available ROM = 4K/1K = 4
- Decoder: It must be able to select 4 ROM, so 2×4 decoder must be used.
- Higher address bits = $\overbrace{\log_2(4K)}^{\text{total address bits}} - \overbrace{\log_2(1K)}^{\text{lower address bits}} = \log_2(2^{12}) - \log_2(2^{10}) = 12 - 10 = 2$ bits or lines
- Total address line: 4K = 2^{12} = 12 address lines, and 10 lines (A_9 to A_0) are connected to each ROM. 2 higher address is represented by inputs of decoder.

Hence, four ROM must be connected top to bottom and data line of each ROM is ORed. Decoder of 2×4 is used to select a particular ROM.

Scanned with CamScanner

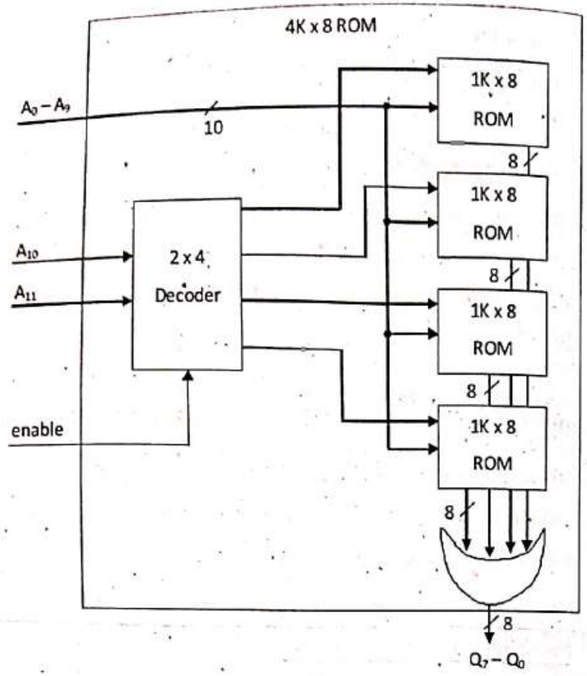


Figure 4.12: Composing 4K×8 ROM using 1K×8 ROM

3. **Case 3: To increase both, number of words and word widt**

When the width of the word as well as the number of words in th available memory and required memory are different then th technique used in case 1 and case 2 must be combined. Initially, th number of words is increased and then the top-bottom set of ROM with ORed data lines are placed side by side to increase the wor width.

Example 1: Compose 1K×8 ROMs into a 4K×16 ROM

Analysis: The available ROM 1K×8 and required ROM 4K×16 differ i number of words as well as word width.

- Increase number of words: 4K/1K = 4, four ROMs are require with a 2 x 4 decoder. 4K represents 12 address lines, 10 line connected to every ROM and 2 lines represented by inputs o decoder.
- Increase word width: 16/8 = 2, four set of ROMs are repeate two times and placed side by side.

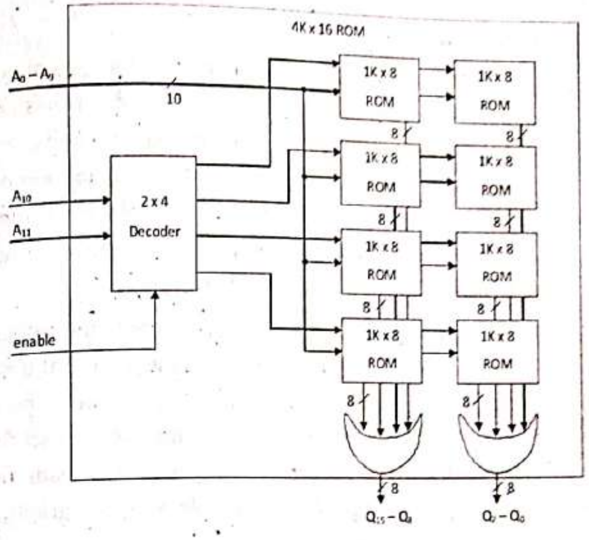


Figure 4.13: Composing 4K×16 using 1K×8 ROM

4.5 Memory Hierarchy and Cache

4.5.1. Memory Hierarchy

A system cannot be implemented with only fast memory as it makes the system very expensive. Also the use of only slow and low cost memory will make system very inefficient. So, the concept of memory hierarchy comes into action in which a system is more likely to implement slow but high capacity memory for storage along with fast but small memory for high speed processing. Memory hierarchy defines the level of memory based on cost per bit, capacity and access time. As we move down the hierarchy, capacity increases, access time increases and cost per bit decreases.

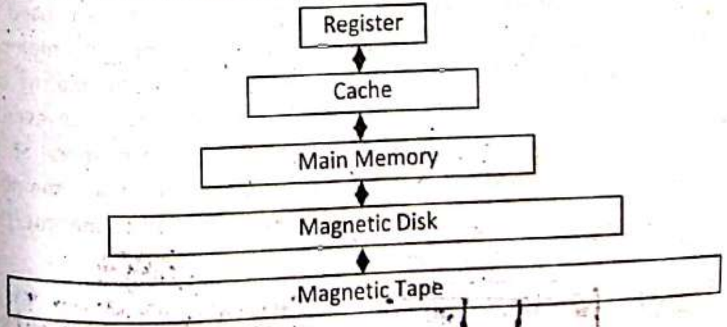


Figure 4.14: Memory hierarchy

4.5.2. Cache Memory

Cache is a small but fast memory which contains a copy of portion main memory to expedite operations of the system. Cache is designed u static RAM which makes it faster as compared to main memory. The acc time for cache can get as low as one clock cycle while main memory acc requires several cycles. So, the instructions and data which are supposed get accessed frequently are placed in cache memory. Hence, the avera access time is reduced resulting in improved performance.

During cache operation, the processor first checks the required w in cache. If it is available (cache hit), the word is delivered to the proces But, however, if the word is not available (cache miss) in cache then t corresponding block of main memory is read into cache. And finally t word is made available to the processor. This operation leads to vario cache design issues which are discussed in the following paragraph.

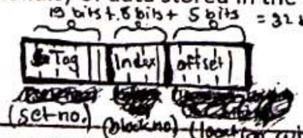
1. Cache Mapping Techniques

Cache memory is very small as compared to main memory. And blocks of main memory cannot be assigned to cache memory once. So, cache mapping techniques are required to assign particu block of main memory to the appropriate line in cache memo There are basically three types of mapping techniques which a discussed below.

a. Direct Mapping (one-way set associative) • fast • worst hit

In this technique, main memory block is assigned to a fixe cache line. The cache stores the content of main memory, th tag and the valid bit. Here, the memory address is divided in the tag, the index and the offset. The index, which is defined b the cache size, represents the cache address. Index is used t select the particular cache line. The tag from main memo address is compared to the tag stored in cache. In case the ta matches, the data from the cache line is accessed. However, single cache line can store few blocks of main memory. So t select a particular block, the offset part of main memo address is used. The valid bit in cache is used to indicate th validity of data stored in the cache slot.

Format:


19 bits + 8 bits + 5 bits = 32 bits
Tag
Index
offset
(set-no.)
(blockno)
(location within the block)

cache can only hold one set at once

{Memory is divided into sets, which is divided further into blo

At 1st CPU looks for set no In the cache directory, if it's not a match then it fetches the set that gives us match from main memory. After that it matches the block no. in

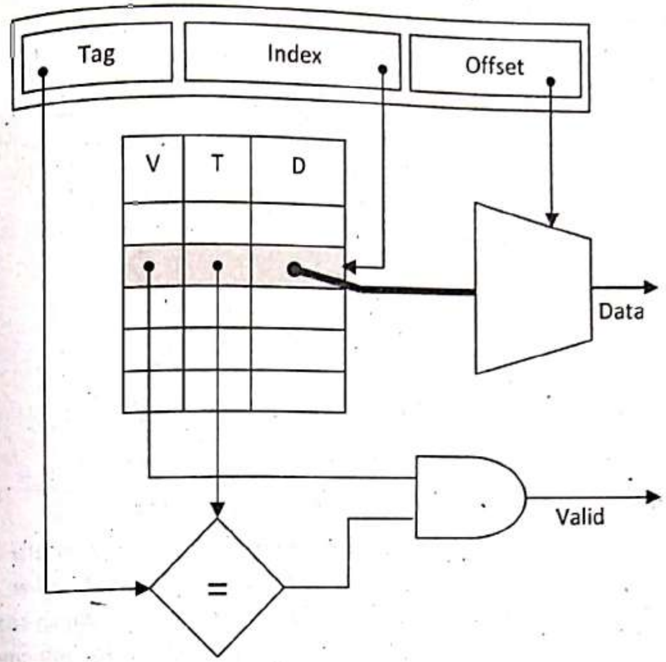


Figure 4.15: Direct cache mapping

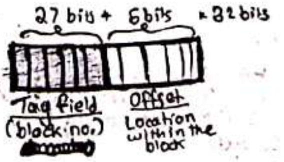
Example: If we have to access block-0 of set 1 & block-0 of set 25 at once; it's not possible, because after accessing 1st block whole cache will need to be replaced

Direct cache mapping is easy and simple in implementation. However, when two blocks of main memory which are assigned to a particular cache line are to be accessed frequently, then cache miss occurs repeatedly. This problem is commonly referred as thrashing. Also, replacement algorithm cannot be used, since main memory blocks are mapped to a fixed cache line.

b. Fully Associative Mapping • very slow • best hit ratio

In this mapping, main memory block can be assigned to any slot of cache line. The main memory address is divided into tag field and offset field. The tag from main memory is compared to each tag in the cache line. After the tag matches the offset is used to select a particular word in cache line.

Format:



↳ no sets

↳ given block no is matched with tag no. in cache directory until a match is found. In case there isn't any matching tag, the CPU moves towards the main memory.

Scanned with CamScanner

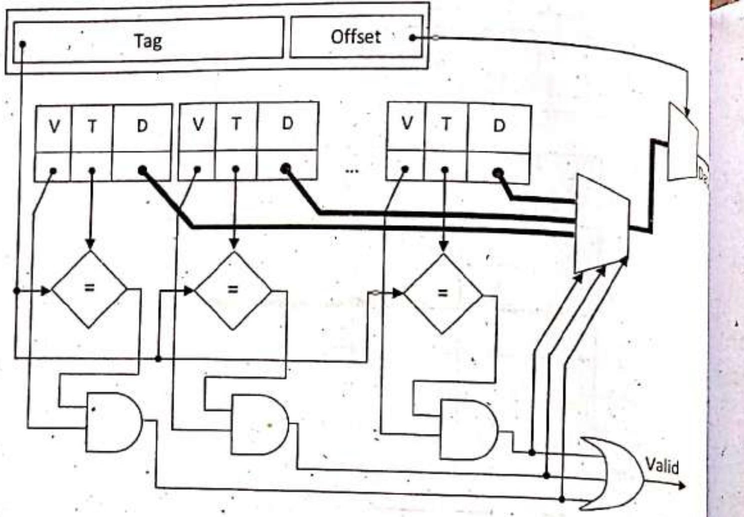


Figure 4.16: Fully associative cache mapping

Fully associative mapping provides high flexibility as block main memory can be assigned to any cache line. However, comparison logic is required for each cache line which mak this mapping method complex and expensive to impleme Miss rate can increase if frequently required block is replace so appropriate replacement algorithm must be utilized efficient cache implementation.

c. Set Associative Mapping • more practical than previous

Maddr(32)
Tag | Index | Offset
20 7 5

It is a compromise mapping which, somehow, follows bo direct and fully associative mapping. The cache is divided in sets, each with number of cache lines. A cache with a set of s N is called an N-way set associative cache. Each block of ma memory can be mapped to particular line of any sets (fixed li but varying sets) or any lines of particular set (fixed set b varying lines). Taking former case into consideration, the ma memory address is divided into tag, index and offset. The ind field is used to select the fixed cache line, and the tag field main memory is compared to tag of each sets. When th particular set is selected, the offset is used to select th particular word from the set in which the tag matches.

• In this method cache can hold more than one sets at once, because of this the drawback of direct mapping (where it couldn't simultaneously access ~~[illegible]~~ two blocks of same no.) is solved.

↳ This method fails if you want (n+1) blocks of same no @ same time of ~~[illegible]~~ (n+1) different sets, but it is a very unlikely scenario

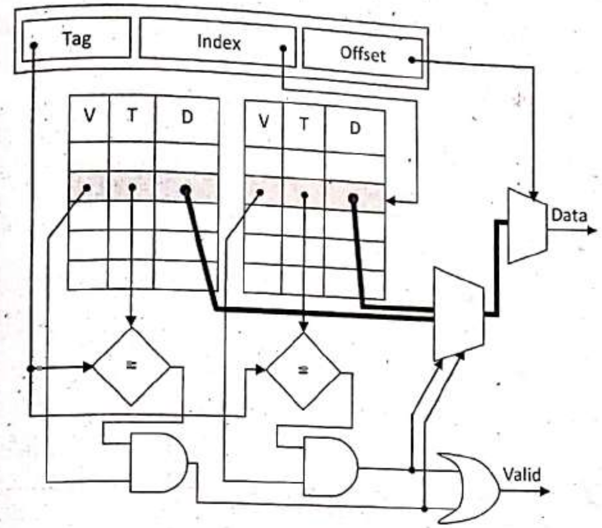


Figure 4.17: Two-way set associative

Set associative cache mapping is more flexible and can reduce cache misses as compared to direct mapping. Though the block of main memory is assigned to fixed cache line, block can be assigned to any sets of cache line. And proper implementation of cache replacement can be used to increase cache hit rate. Also the comparison logic is not required for every cache line rather is required for only available sets, which reduce the complexity and expense for implementing comparison logic.

2. Cache-Replacement Policy

When cache is full and new main memory block is to be assigned to the cache then certain technique must be used to choose which cache line should be replaced. This mechanism of replacing the existing block by new set of blocks is referred as cache-replacement policy. In direct mapping the main memory block always maps to the fixed cache line, so replacement is fixed. But fully associative and set associative can follow various replacement algorithms. Least Recently Used (LRU), First In First Out (FIFO), Least-Frequently Used (LFU) and Random are few commonly used replacement techniques.

a. **Random replacement** replaces the block randomly without following any specific algorithm.

Scanned with CamScanner

for longest period of time

b. **Least Recently Used (LRU)** algorithm is based on time in wh the block not accessed for longest time is replaced by the n block.

c. **First In First Out (FIFO)** method uses queue mechanism replace the first entered block. Each block is pushed into t queue when accessed. And when replacement is required t blocks are popped out from the queue.

the block that has been in cache the longest would be selected & removed

d. **Least Frequently Used (LFU)** technique is based on number time the block is accessed. The block which is accessed le number of times is replaced.

3. Cache Write Techniques

A mechanism is required when content of cache is changed by th processor and the change must be updated to the correspondi main memory block. This technique of updating the main memo after change in cache is referred as cache write policy. There are tw common cache write policy; write-through and write-back.

a. **Write-through** is a technique in which the main memory i updated immediately after the content in cache is changed. Th technique is easier to implement but the processor has to wa for slower main memory frequent access. Also there are chance of unnecessary writes resulting in substantial memory traffic. Fo example when a particular value is changed four times, the la updated value must only be updated in the main memory. Bu the memory is updated four times for every change causin unnecessary memory access.

b. **Write-back** policy allows main memory to be updated only whe cache line is to be replaced. Extra bit is associated with eac cache line to represent whether the content of cache line i changed or not. Based on that extra bit the corresponding mai memory block is updated when cache line is about to b replaced. Extra bit and update checking increase syste complexity; however, it reduces number of slow main memor access and avoids memory congestion.

4. Cache Impact on System Performance

The performance of system is directly related to design and configuration of caches. The total size of cache, degree of associativity, and the data block size are important parameters that have direct impact on performance.

Cache size is the total number of bytes that the cache can store. The tags and extra bits, which do not contribute to the size of the cache, are also stored in cache along with the data of main memory block. Increasing the size of cache results in lower miss rates, however the access of data from the cache will be slower. So, larger cache size does not necessarily mean better performance.

Degree of associativity is related to number of sets used in set associative cache implementation. Increasing the number of sets will improve the hit rate. However, additional logic requirement will increase the access time latency.

Cache line size represents the size of each block in cache that holds the block of data of main memory. When line size is increased, the main memory access time is, obviously, reduced but only at the expense of more complex multiplexing circuitry which increases the access latency.

Example: Effect of Cache Size on System Performance

Case I: Cache size = 2Kbytes, miss rate = 15%, hit cost = 2 cycles, miss cost = 20 cycles

Average cost of memory access = (0.85 x 2) + (0.15 x 20) = 4.7 cycles

Case II: Cache size = 4Kbytes, miss rate = 6.5%, hit cost = 3 cycles, miss cost = 20 cycles

Average cost of memory access = (0.935 x 3) + (0.65 x 20) = 4.105 cycles

Case III: Cache size = 8Kbytes, miss rate = 5.565%, hit cost = 5 cycles, miss cost = 20 cycles

Average cost of memory access = (0.94435 x 2) + (0.05565 x 20) = 4.8904 cycles

In case II, increase in cache size, certainly, improved the performance as average cost of memory access is decreased. However, in case III, increase in cache size added more cycles for memory access in average.

Scanned with CamScanner

Chapter 5

INTERFACING

- Communication Basics
- Microprocessor Interfacing
  - I/O Addressing
  - Interrupts
  - Direct Memory Access
- Arbitration
- Multilevel Bus Architectures
- Advanced Communication Principles
- Serial, Parallel and Wireless Protocols

5.1 Communication Basics

1. Basic Terminology

- **Wires**

  Wires are the connecting lines of two terminals communication system. It may be uni-directional or directional. A single line can be used to represent multipl wires with the help of small angled line drawn through it.

- **Bus**

  Bus refers to the set of wires with a single function. Address b for address, data bus for data are two examples of sing functioned buses. Bus can also be the entire collection of wire System bus, for instance, consists of address, data and contr lines.

- **Port**

  Port is the actual conducting device on periphery whi connects bus to processor or memory or other devices. Port i medium through which a signal is input to output from t processor. Port is also referred as pin which extends from the package and that can be plugged into a socket (IC base) or printed circuit board. Metallic balls instead of pins may present. However, metal pads are more common these days.

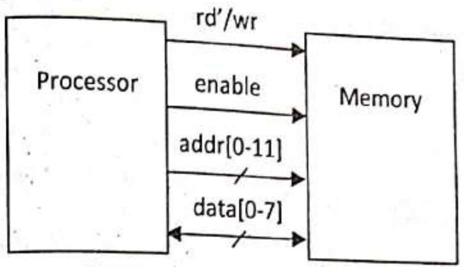


Figure 5.1: A simple bus example

- **Timing diagram**

  Timing diagram is a diagrammatic representation for describing hardware protocol. In the diagram, time proceeds to the right along x-axis. It represents state of control lines or data lines. The control lines may be either low or high, whereas the data lines – address or data -- can be valid or not valid. Active high means that a one on the line makes it active while active low means that a zero on the line makes it active. Asserting a line means making it active and de-asserting the line deactivates the line. A protocol may have several sub-protocols which are also called bus cycle or transaction. A bus cycle may consist of several clock cycles.

  Example: Timing Diagram for Read Protocol

  The timing diagram of memory read protocol gives the following information to the designer

- The processor must set the *rd'/wr* line low for a read operation
- Address of memory must be placed on *addr* line for atleast t_{setup} time before setting the *enable* line high.
- Setting *enable* line high will cause memory to place the data on the *data* line after at time t_{read}.

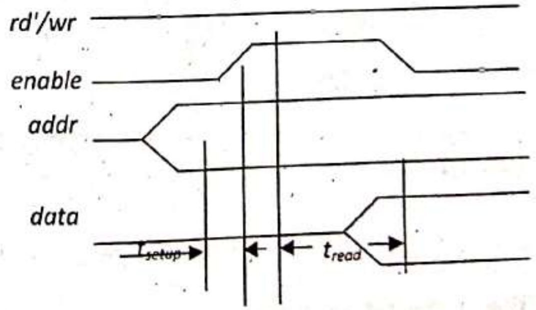


Figure 5.2: Timing diagram: memory read protocol

Scanned with CamScanner

2. Basic Protocol Concepts

- **Actor**

  An actor is a device that can be processor or memory wh takes part in data transfer. Actor can be a master or a slave master initiates the data transfer whereas a slave responds the initiation request.

- **Data direction**

  Data direction represents movement of data among actors. T direction of data is independent of type of actor. Either mast or slave can send or/and receive data.

- **Addresses**

  Addresses represent a special kind of data which specify location in memory, a peripheral, or a register within peripheral. A protocol often includes both an address and dat In every memory access protocol, the address specifies th location where the data should be read from or written to in th memory.

- **Time multiplexing**

  Time multiplexing represents a technique in which the multip sets of data are sent one at a time over the shared line. Numbe of wires requirement can be reduced to a single line at th expense of time. The following figures show the examples o time multiplexing. In both cases, single bus is used to sen multiple data at different time instant.

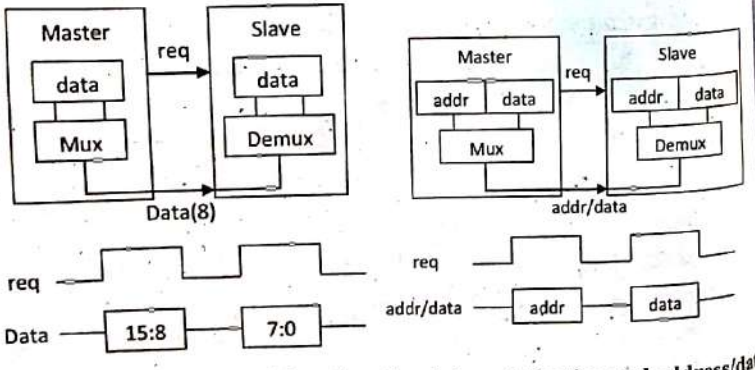


Figure 5.3: Time-multiplexed data transfer: data serialization and address/data muxing

- **Control methods**

  Control methods are schemes for initiating and ending the data transfer. Strobe and handshake are two common control methods.

i. Strobe Protocol

In strobe protocol, master uses a control line and activates it to initiate the data transfer. Then the slave has certain time to put data on data bus. Assuming data to be valid, master reads the data from data bus and deactivates its control line. And both actors are ready for next data transfer. The main disadvantage of strobe protocol is that the master that initiates the transfer has no way of knowing whether the slave has received the data or not.

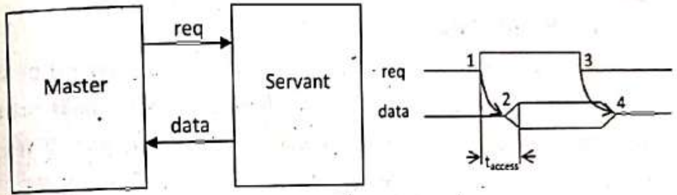


Figure 5.4: Strobe protocol

The flow in timing diagram can be explained as:

1. Master asserts *req* to receive data
2. Servant puts data on *data* line within time t_{access}
3. Master receives data and deasserts *req*
4. Servant ready for next request

ii. Handshake Protocol

In this protocol, servant uses extra line to acknowledge that the data is ready. Initially, master asserts request line to start the transfer. Then the servant, taking its time to put data on data line, asserts acknowledge line to inform the master that the data is ready. Next, the master reads the data from the data line and deasserts the request line which is followed by slave deasserting acknowledge line. Finally the transfer is complete and both actors are ready for next transfer. Though the protocol is somewhat complex, it is more reliable compared to strobe protocol as data availability is confirmed by the sending device.

Scanned with CamScanner

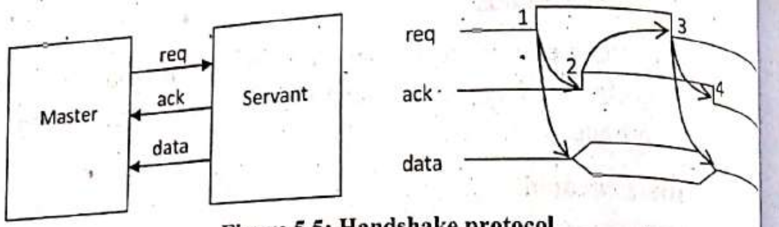


Figure 5.5: Handshake protocol

The flow, as indicated by numbers, in timing diagram can summarized as:

1. Master asserts *req* line to receive data
2. Servant puts data on *data* line and asserts *qck*
3. Master receives data from data bus and deasserts *req*
4. Slave deasserts acknowledgant line.
5. Servant ready for next transfer

iii. Strobe/Handshake Compromise

A compromise protocol can be used to achieve the speed of stro protocol and varying response time tolerance of handshake proto As represented in Figure 5.6, servant can use wait line, if it is n ready to put data on data line.

- If the servant can put data within time t_{access} then it follo strobe protocol representing fast response. And wait li remains unused in this protocol.
- If the servant can't put the data within t_{access} time then it as master to wait longer by asserting wait line. After the data ready, the wait line is deasserted by servant and mast receives the data. And it represents slow response as mast has to wait for certain time for the data transfer

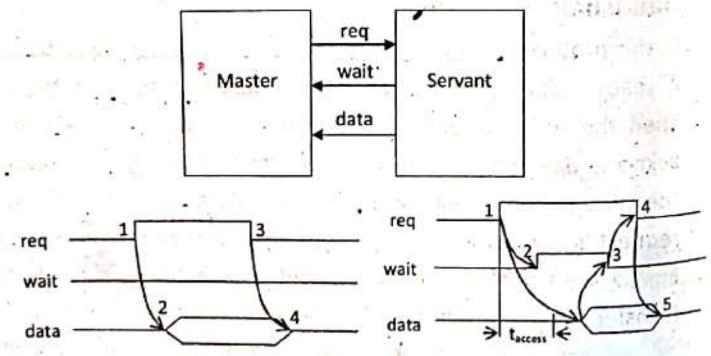


Figure 5.6: A strobe/handshake compromise: fast and slow response

The flow as indicated by numbers in timing diagram can be summarized as:

a. For fast response

1. Master asserts *req* line to receive data
2. Servant puts data on data bus within time *taccess*, wait line remains unused
3. Master receives data and deasserts *req*
4. Servant ready for next request

b. For slow response

1. Master asserts *req* to receive data
2. Servant can't put data within t_{access}, asserts *wait* line
3. Servant puts data on bus and deasserts *wait*
4. Master receives data and deasserts *req*
5. Servant ready for next request

Example: The ISA Bus Protocol – Memory Access

The industry standard architecture bus protocol is common in systems using an 80×86 microprocessors. The processor uses 20-bit memory address and follows compromise strobe/handshake protocol. If the memory is not ready then the processor inserts wait cycles. Four cycles is default for the operation to complete. For the read operation, in the first clock cycle the processor puts address on the address line and asserts address latch enable signal. During second and third clock cycle, the processor asserts memory read signal. After third clock cycle, the data is available on data lines. Finally are signals are deasserted at fourth clock cycle. The timing diagram for memory read operation and memory write operation is shown in the figure below.

Scanned with CamScanner

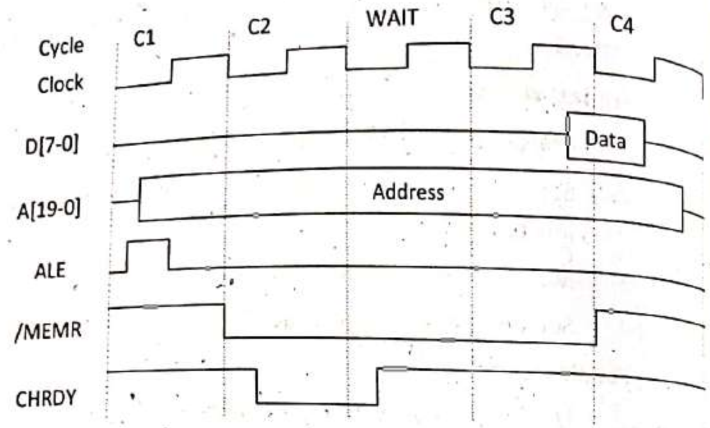


Figure 5.7: ISA bus protocol – read bus cycle

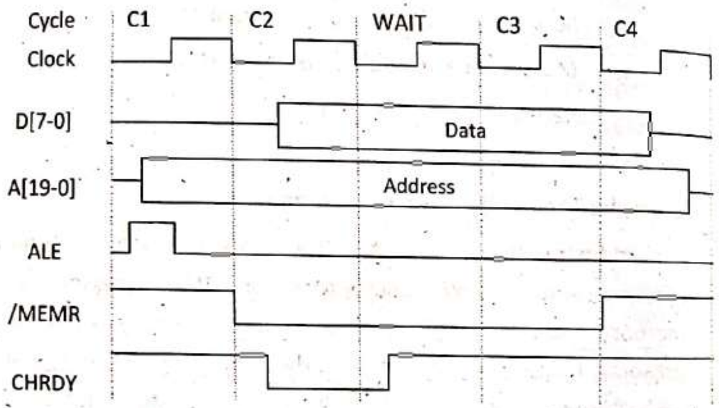


Figure 5.8: ISA bus protocol – write bus cycle

For Write Operation

- In C1, processor puts 20-bit address memory address on the addres line and asserts ALE signal.
- During C2 and C3, the processor puts the data on the data line and asserts MEMW signal to enable write operation. However, if the memory, when not ready, deasserts CHRDY signal in C2 then processor inserts wait cycles until CHRDY is reasserted.
- In cycle C4, all signals are deasserted.

5.2 Microprocessor Interfacing

5.2.1. I/O Addressing

In this section, we discuss on how microprocessor communicates with other different Input output peripherals. I/O addressing basically means how they are connected to the bus structure, how address spaces are assigned to I/O devices and how communication is performed between processor and peripherals. There are two types of I/O addressing: Port based I/O and bus based I/O.

1. Port based I/O

In port based I/O, a port can be directly read from or written into with the help of processor instructions. It is also referred as parallel I/O. Generally the devices may be provided with one or more N-bit ports to facilitate port based I/O and each port is bit addressable. For example, 8051, AVR microcontrollers have 8 bit I/O ports. In 8051, P1 = 0xF7 statement will write into Port 1 of microcontroller. Also, for bit addressing, P2.4 = 1 will set the pin 4 of Port 2.

The port based I/O can be extended using appropriate peripheral which extends the number of available ports from four to six. Each port on peripheral is associated with a register that can be read or written into by the processor.

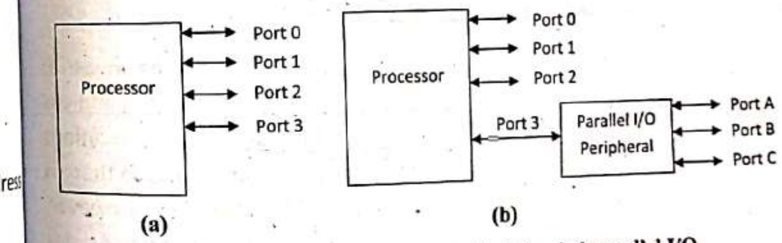


Figure 5.9: (a) Port based I/O (b) Extended parallel I/O

2. Bus based I/O

In bus based I/O, the processor has address, data and control lines for I/O addressing. The communication protocol is built into the processor. A single instruction is available which causes the hardware to write or read data. If a system with bus based I/O requires parallel I/O then parallel I/O peripheral can be connected to the system bus. Bus based I/O can be categorized into memory-mapped I/O and standard I/O.

Scanned with CamScanner

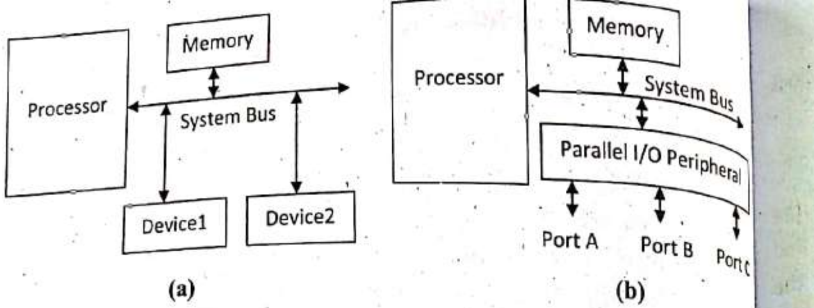


Figure 5.10: (a) Bus-based I/O (b) Extended bus-based I/O with parallel I/O peripheral

a. Memory-Mapped I/O

Memory-mapped I/O is a type of bus-based I/O addressing fo processor to communicate with peripherals in which periphe are addressed using the specific existing address space. T total address space is divided into memory address a peripheral address. Hence, there is loss of memory addresses peripherals. Also, no special instructions for peripherals a required for data transfer, since instructions like MOV used f memory will also work with peripherals.

Example: A bus with 16 bit have total of 65536 addresses. S lower 32768 addresses may correspond to memory addre while upper 32768 correspond to I/O addresses.

b. Standard I/O

Standard I/O is a type of bus-based I/O addressing in whi extra control line (M/IO) is used to indicate whether the addre represents memory location or peripheral. Memory locatio and peripherals use all sets of address for addressing, so there no loss of memory addresses. This addressing, howeve requires special instructions. MOV, LOAD instructions f memory while IN, OUT for peripherals. Also the addre decoding logic for peripherals is simple as the high ord address bits can be ignore when the number of peripherals less.

Example: A bus with 16 bit have total of 65536 addresses. A 65536 addresses can be used to address memory a peripheral. The M/IO line is used to select either memory o peripheral. If M/IO is zero then the address on the address bus corresponds to a memory address.

Example: The ISA Bus Protocol – Standard I/O

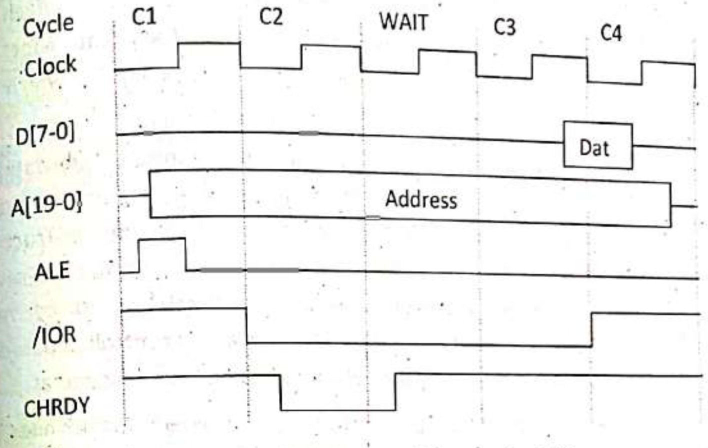


Figure 5.11: ISA bus protocol for standard I/O

5.2.2. Interrupts – Interrupt Driven I/O

The peripherals may require service from the processor which is very much unpredictable. So there is an issue on how to serve the peripherals by the processor as it remains busy on its own task. Polling and interrupts are two basic methods to address that issue.

1. **Polling**

   Polling is a method in which the processor checks for service requirement of every peripheral. This method, though, is easier and simple to implement, the repeated checking, however, wastes many clock cycles which could have been used to do certain useful work.

2. **Interrupt**

   Interrupt is a feature of the processor through which the peripherals can request for service even when processor is busy in its own task. For external interrupt, there is always a pin available to implement the interrupt feature. Whenever the interrupt pin is asserted, the processor jump to a particular address at which the routine for the interrupt is stored. Interrupt overcomes the limitations of polling, but interrupt, in itself, is the type of polling. The pin is checked after the

Scanned with CamScanner

execution of every instruction, so it does not requires extra c cycles.

i. **Interrupt address vector**

Interrupt address vector represents the address in which t interrupt service routine (ISR) resides. Fixed interrupt a vectored interrupt are two common methods by which t processor obtains the address of ISR.

a. **Fixed interrupt:** In fixed interrupt, the address of subroutine built into microprocessor and remains fixed. Programmer simp has to store the ISR at that location or can put jump instructio to move to actual location of ISR where programmer has save Suppose a data from sensor (peripheral1) is to be rea processed and then a motor (peripheral2) is controlled based o calculated data. The flow of actions can be summarized as:

- Peripheral1 has data in its register; meanwhile th processor is executing its main program.
- Peripheral1 asserts INT to request service from th processor.
- After execution of each instruction, the processor chec INT pin. So processor detects the service requirement saves its present context and sets the PC to the fixed IS location.
- The ISR is executed which reads data from peripheral1 modifies it and sends the resulting data to peripheral2. A the same time, peripheral1 deasserts INT after data is rea from it.
- The processor retrieves its state and resumes its work.

b. **Vectored interrupt:** In vectored interrupt, peripheral mus provide the address to the processor. In this method, along wit INT pin, INTA pin is also required to acknowledge that th interrupt has been detected and the peripheral can provide th address of relevant ISR using system bus. The periphera provides the address through the data bus which is read b microprocessor.

The flow of actions can be summarized as:

- Peripheral1 has data in its register; at the same time the processor is executing its main program.
- Peripheral1 asserts INT to request service from the processor.
- After execution of each instruction, the processor checks INT pin. So processor detects the service requirement and it asserts INTA.
- Peripheral1 detects INTA and puts interrupt address vector on the data bus.
- Processor jumps to the address read from data bus and executes its corresponding ISR. It reads data from peripheral1 processes it and sends the result to the peripheral2. Meanwhile peripheral1 deasserts INT after data is read from it.
- The processor retrieves its state and resumes its work.

c. **Interrupt address table:** In interrupt address table, which is a compromise between fixed and vectored interrupt methods, a table with ISR addresses is stored in memory of processor. A peripheral instead provides the number, rather than the address of ISR, corresponding to an entry in the table. One major advantage is that the bit requirement to address the table is very less compared to number of bits of real address of ISR. Also it provides the flexibility to assign and change the location of ISR.

ii. **Additional Interrupt Issues**

External interrupts may be maskable or nonmaskable. In maskable interrupt, the programmer can use specific instruction to disable the interrupt by configuring certain bits of interrupt register. It is important when more critical works need to be executed first. Nonmaskable interrupt cannot be disabled by the programmer. It requires a separate pin for drastic situations. For instance, if power fails, the nonmaskable interrupt can cause a jump to a subroutine that stores critical data in non-volatile memory, before power is completely gone.

Scanned with CamScanner

Another issue regarding the interrupt is jump to ISR in whic microprocessor either saves complete context or partial before jumping to ISR. Some processors save PC, registers w consumes many cycles, while others save the content of only. The ISR, however, must not modify registers if its con is not saved.

5.2.3. Direct Memory Access – DMA controller

1. Introduction

When the communication between memory and peripherals invol microprocessor then there will, somehow, always be waste processor's time. Since the speed of the processor and periphe may not match, data must be stored temporarily before proces which is referred as buffering. Buffering will, certainly, impact system performance. Also while using interrupt feature, the stor and restoring of state of processor is an inefficient process, since process requires many clock cycles. And, the regular program st during transfer of data causing more problems in the performance the system. So, a separate single-purpose processor called a DM controller is required which relieves processor from all data transfe involving memory and peripherals.

DMA controller is specifically used to transfer data betwe memories and peripherals. The peripherals request the service fro DMA controller which then requests control of the system bus fro processor. After that, processor relinquishes the system bus. Finall the data transfer between memory and peripheral is initiated b DMA controller without the involvement of processor. Hence, th overhead required for storing and restoring the state is eliminated Also, the processor can continue its regular task unless it requires th system bus or the particular data being transferred.

2. Block Diagram

The simple block diagram of system involving DMA controller i shown in the figure below:

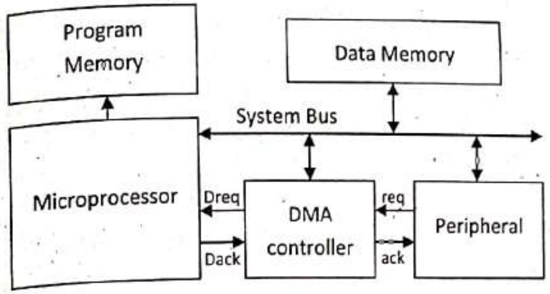


Figure 5.13: Simple system with DMA controller

3. Operation

The flow of action for the transfer of data between peripheral and memory using DMA can be summarized as:

- Initially, processor is busy executing its main program.
- After peripheral has data within its register it asserts request line for service from DMA.
- DMA asserts request signal to request the system bus from processor.
- Processor releases the system bus after seeing the request from DMA, and acknowledges about it to DMA.
- DMA asserts acknowledge signal to peripheral, and starts transfer of data as requested.
- After the completion of transfer, all control lines are deasserted and processor retakes the control of the system bus.

5.3 Arbitration

Arbitration is the mechanism through which a service or shared resource is provided to particular requesting device, out of many contenting devices for service.

1. Priority Arbiter

i. Introduction

Priority arbiter is a single purpose processor which is used to arbitrate among various requests from peripherals. Each of the peripherals, which are connected to the arbiter, can make request for the service. Using certain priority mechanism, the arbiter selects a peripheral to permit the required service. The

Scanned with CamScanner

figure 5.14 shows the priority arbiter connected wi peripherals which use vectored interrupt to request service a processor which provide service to the peripherals. Arbiter connected to system bus for configurations only. Th configurations may include setting priorities of the peripherals

The main advantage of this arbitration is that it can suppo advanced priority schemes. Also, failure of single periphe does not have any impact on the operation of whole system The system, however, must be redesigned if new peripheral are to be added. So, this method is less flexible if ne peripherals are required to be added or removed.

ii. Block Diagram

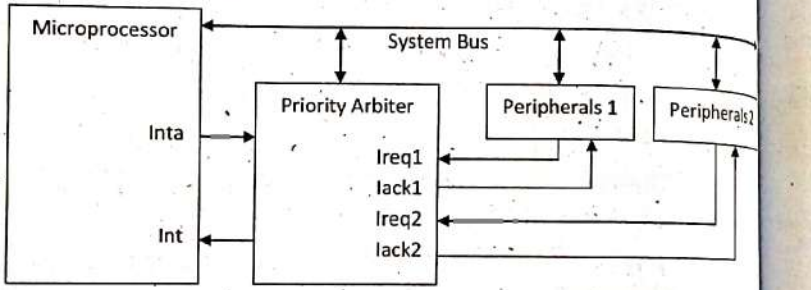


Figure 5.14: Arbitration using a priority arbiter

iii. Operation

The stepwise operation of arbitration using priority arbiter i listed below:

- Initially, microprocessor is busy in its own operation.
- Both peripherals can assert request to priority arbiter which interrupts processor when at least one request i available from peripherals.
- Processor stops its current operation, stores its state and asserts interrupt acknowledgement signal.
- After acknowledged by processor, priority arbiter asserts acknowledge signal to any one peripheral based on priority.
- The selected peripheral puts its interrupt address vecto on the system bus.

- Microprocessor reads ISR address from data bus and jumps into its, executes the ISR.
- After execution of requested ISR, processor retrieves its state and resumes its operation.

iv. Types of Priority Arbiter

The priority among peripherals can be determined based on, basically, two schemes; fixed priority or rotating priority.

a. Fixed Priority

Each peripheral is assigned a unique rank. If two peripherals simultaneously request for service then the arbiter chooses the one with the higher rank. Such method is efficient when there is a clear distinction in priority among peripherals. But it can cause high-ranked peripherals to get much more servicing than other peripherals.

b. Rotating priority or Round – robin priority

In this method, each peripheral gets almost equal time for service from the arbiter. This priority method is efficient when there is not much difference in priority among peripherals. The priority of peripherals changes based on the history of servicing of those peripherals, so the arbiter can get more complex in rotating priority.

2. Daisy-Chain Arbitration

i. Introduction

In daisy-chain arbitration, peripherals are connected to each other in daisy-chain manner. The arbitration is build within the peripherals with each having a request and acknowledge signals as shown in the figure 5.15. The request signal and acknowledge signals flow through the peripherals: peripheral's request signal flows downstream to processor and processor's acknowledge signal flows upstream to requesting peripheral. The peripheral connected first to the processor has the highest priority while the peripheral at the end of chain has lowest priority.

The main advantage of this arbitration method is that one can easily add or remove peripherals from the system without the requirement of system redesign. This method, however, does not support rotating priority. Also, if one peripheral is damaged

Scanned with CamScanner

in the chain, other peripherals beyond that broken point w remain inaccessible as signal cannot pass through the chain.

ii. **Block Diagram**

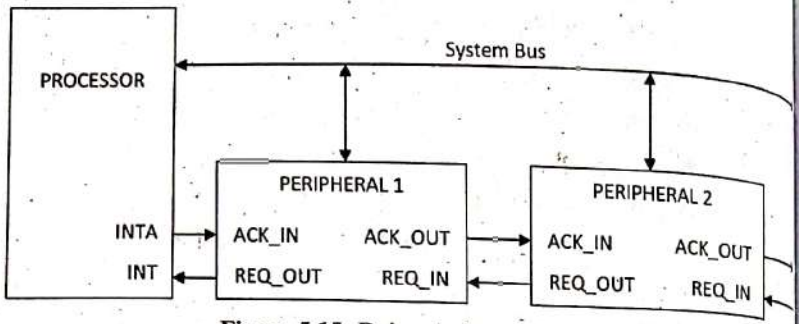


Figure 5.15: Daisy chain configuration

iii. **Operation**

Suppose peripheral 2 requires service from the processor th the operation can be summarizes as:

- Microprocessor is busy in executing its own program.
- The request signal from peripheral 2 is send to process through the peripheral 1 and interrupt pin is asserted.
- Processor stops its current work, stores its state, a asserts acknowledgement signal.
- The acknowledgement signal reaches to peripheral through peripheral 1. Since the request is not generated b peripheral 1, it passes the acknowledge signal t peripheral 2.
- Peripheral 2 puts its interrupt address vector on t system bus.
- Microprocessor reads ISR address from data bus and jum into its, executes the ISR.
- After execution of requested ISR, processor retrieves t state and resumes its operation.

iv. **Daisy Chain Aware Peripherals**

Generally, peripherals have acknowledge input and request o lines but daisy chain aware peripherals must have additio acknowledge output and request input lines. However, if t peripherals do not contain acknowledge output and requ input lines then they will not be daisy chain aware peripherals. But they can be made daisy chain aware by certain logic whose complexity may increase based on complexity of system. One simple example for making a peripheral daisy chain aware is shown in the figure below.

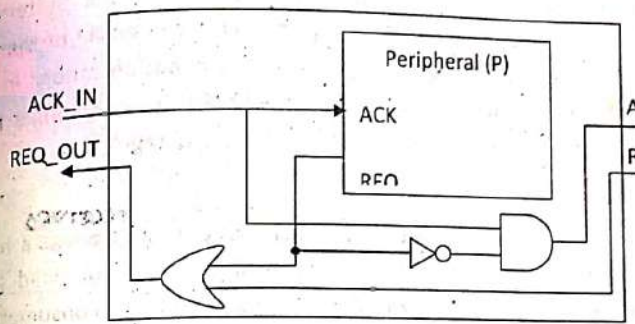


Figure 5.16: Simple logic to make daisy chain aware

Case 1: When request is from downstream peripherals

- Peripheral (P) does not participate in the flow of signal

Case 2: When request is from upstream peripherals beyond peripheral (P)

- REQ_IN = 1 but REQ = 0, resulting in REQ_OUT = 1
- ACK_IN = 1 and REQ = 0, resulting in ACK_OUT = 1

Case 3: When request is from peripheral (P)

- REQ = 1, REQ_IN = X (don't care), resulting in REQ_OUT = 1
- ACK_IN = 1 and REQ = 1 resulting in ACK = 1 and ACK_OUT = 0

3. Network-Oriented Arbitration

In network-oriented arbitration, arbitration is done for multiple microprocessors sharing a common to form a network. Arbitration is build into the bus protocol, as bus is the only the medium that connects multiple processors. However, multiple processors may try to access the bus simultaneously resulting in data collision. The protocol must be designed in such a way that the contending processors don't start sending the data at the same time. Also some statistical methods can be used so as to make chances of data collision very rare, if not eliminate it. Some protocols use efficient

Scanned with CamScanner

address encoding schemes in which higher priority address override the lower-priority one.

5.4 Multilevel Bus Architectures

Multilevel bus architectures are implemented in the system improve the overall performance of the system. One can easily presume single high-speed bus would be enough for all the communications in t system. But, however, there are various drawbacks of using single hi speed bus. Few of them are discussed in the following paragraph.

- **Inefficient interface**

  For a single high speed bus, each peripheral ~~requires~~ receives a hig speed bus interface. But the peripherals may not need su high-speed transfer resulting in extra power consumptio increase in number of gates, and high cost. Also, the high-spee bus can be very processor specific which can lead the interfa of a peripheral to be non-portable.

- **Slower bus**

  When many peripherals are connected to a single bus, a peripherals may not get the access to the bus when require This condition results to slow down the speed of transfer, hence it can create a performance lag.

1. Two Level Bus Systems

Generally, two level bus systems consist of a high-speed process local bus, a lower-speed peripheral bus and a bridge to connect tw buses.

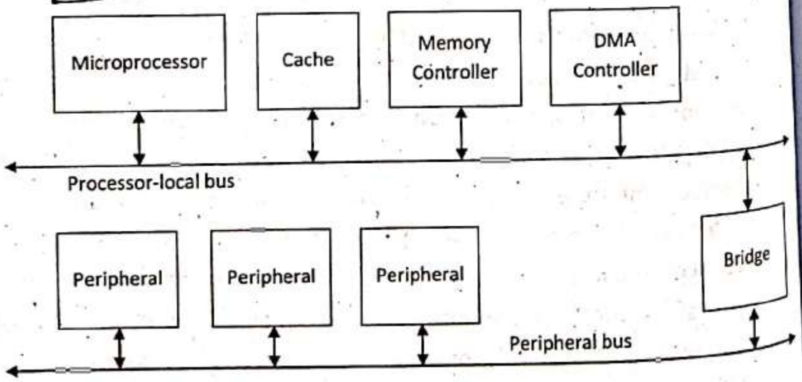


Figure 5.17: A two-level bus architecture

- The processor local bus connects very high speed devices such as microprocessor, cache, memory controllers, and certain high-speed coprocessors. These buses are wide, as wide as a memory word and frequent communication takes place through it.
- On the other hand, the peripheral bus connects to those peripherals which do not have access to processor-local bus. It emphasizes on portability, low power, or low gate count. It is often narrower and slower than a processor local bus. The frequency of communication through peripheral bus is also less as compared to that of processor local bus. So the interface for peripheral bus is comparatively efficient one in terms of number of pins, gates and power consumption.
- A **bridge** is a single purpose processor that connects the two buses of the system and also makes the various conversions required. Speed synchronization is another important function of bridge. Data speed and data formats of processor-local bus is different to that of peripheral bus, such problem is resolved by bridge using various mechanisms.

2. Three level bus hierarchy

Three level bus systems consist of processor local bus, system bus and peripheral bus. A local bus connects the processor to a cache and may support one or more local devices. The system bus, acting as high-speed bus, offloads much of the traffic from the processor local bus. And the peripheral bus is used to connect various peripherals in the system.

5.5 Advanced Communication Principles

1. Parallel Communication

In parallel communication, the physical layer carries multiple bits of data at a time. With each wire carrying a single bit, the bus consists of data wires along with control and power lines.

Advantages

- **High data throughput:** Many bits are transferred at a time.
- **Less complexity:** Easily implemented in hardware requiring only a latch to copy data onto a data bus.

Disadvantages

- Long parallel wires can result in Ferranti effect. And according this effect, there is a voltage build up due to capacitance a voltage at receiving end becomes more than that of sendi end.
- Little variation in wire length can cause data misalignment the bits at the receiving ends with reach at different time.
- It is more costly to construct and can make system bulky. Th cost further increases if insulation of wires to prevent th interference is considered.

Usage

- It is used to connect devices which reside on the same circui board or same IC.

2. Serial Communication

In serial communication, the physical layer carries one bit of data at time. With all bits of data passing through the single wire, the bus composed of single data wire along with control and power lines.

Advantages

- Significant reduction in the size, the complexity of the connectors and the associated costs.
- Throughput can be better for two distant devices as compared to parallel communication of two distant devices.
- It does not exhibit Ferranti effect and data misalignments.

Disadvantages

- Complex interfacing logic and communication protocols; the data are decomposed into bits at sending end, which must be assembled properly at receiving end.
- For short distance communication, its throughput is very less as compared to that of parallel communication.

Usage

- It is used to connect distant devices. But it doesn't mean that it cannot be used to connect devices at short distance. However it is more efficient for distant communications

3. Wireless Communication

In wireless communication, the devices do not need to be connected physically for data transfer. Infrared and radio frequency channels are used as a physical layer.

i. Infrared wave

Infrared wave, which cannot be seen by naked eye, uses electromagnetic wave frequencies that are below the visible light spectrum. Infrared waves are generated using infrared diode whereas infrared transistors are used to detect the infrared emitted by infrared diode. Such infrared transistors conduct when exposed to infrared wave. One advantage of infrared communication is that it is cheap to build transmitters and receivers. But the main disadvantage of this sort of communication is that it requires line of sight between the two devices participating in communication. Also the range of communication is low, which makes it an inefficient method of communication for distant devices.

ii. Radio frequency

Radio frequency uses electromagnetic wave frequencies in the radio spectrum. For such communication, analog circuitry as well as an antenna is required at communicating devices. The main advantage of this type of communication is that the line of sight is not required. Also the longer distance communication is possible. The range of communication is dependent on the transmission power. But building transmitters and receivers can be complex and costly in Radio Frequency communication.

4. Layering

Layering the communications process means breaking down the communication process into smaller and easier to handle interdependent categories, with each solving an important and somehow distinct aspect of data exchange process. Layering can also be viewed as a hierarchical organization of a communication protocol where lower levels of the protocol provide services to the higher levels. The main objective is to break the complexity of a communication protocol into simple levels which ensures easier handling and simplified design. The physical layer provides the lower

Scanned with CamScanner

level services of sending and receiving bits or words of data, where the application layer provides the high level service to the user.

5. Error Detection and Correction

Error detection is the process of detecting errors that may occur during the transmission of data in any communication process. Errors can be bit error or burst of bit errors. In bit error, single bit in the transmitted data is invalid. But in case of burst of bit errors, more than one bit gets changed.

Error correction is the process of correcting the bits that were detected during communication process. Parity and checksum are two basic error correction methods.

In **parity** check, extra bit is send along with data to provide additional information about the data. If extra bit makes an odd number of 1s in data word bits plus parity bit then it is referred as odd parity otherwise it will be even parity. The parity of data must be check at both ends of communication and the parity of the data sent must be same to that of parity of data received. This type of checking method is efficient for single bit error but can create problems for burst of bit errors as it is not able to detect change in even number of bits.

Example:

Data of 7 bits – 0011010

Transmitted data with even parity – 00110101

Received data with parity – 10110101, it indicates error as it should have an even parity.

Received data with parity = 10010101, change in two bits and error not detected.

In **checksum** error checking, multiple words of data in packets are checked for error. The extra word which represents the XOR sum of all data words in a packet is transmitted along with packet. Though it can be implemented for burst of bits error, it does not account for all error combinations. The transmitter sends the packet of data along with the checksum word which is checked at receiving end on reception. If the checksum word is correct then it represents successful transmission. However, few error combinations can generate the checksum word same as received, in such case the error checking fails.

Example:

Data words of packet to be transmitted: 010101, 011101, 110011, 101100

Checksum word at transmitter: 010111 (XOR or all data words)

Received checksum word: 010111

Received data words of packet: 110101, 010101, 110011, 101100, error exists

Calculate checksum word at receiver: 111111, checksum does not match, error check success.

Received data words of packet: 010101, 011101, 110011, 101100, error exists

Calculate checksum word at receiver: 010111, checksum match, error check fails.

5.6 Serial, Parallel and Wireless Protocols

1. Serial Protocol

i. Inter-IC or I2C or I^{2}C

I2C is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems. The I2C has 7-bit or 10-bit address space. Seven bit addressing allows a total of 128 devices to communication over a shared I2C bus. The common speed of I2C bus is 100 Kbit/s in standard mode and 10 Kbit/s in low-speed mode. Recent revisions of I2C can host more nodes and run at faster speeds: 400 Kbit/s in fast mode, 1 Mbit/s in fast mode plus and 3.4 Mbit/s in high speed mode. I2C uses only two wires: SCL (serial clock) and SDA (serial data). Length of the wires is not limited as long as the total bus capacitance is less than 400pf.

Scanned with CamScanner

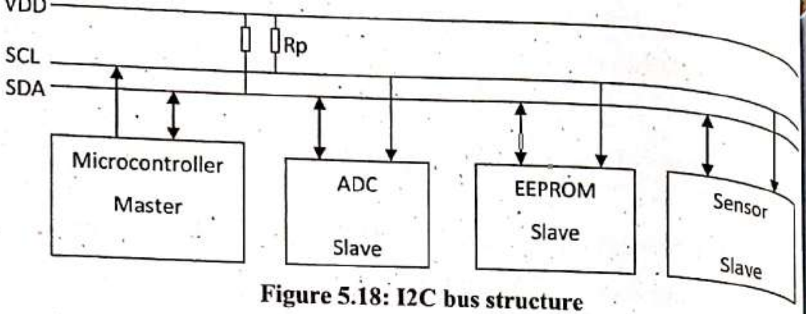


Figure 5.18: I2C bus structure

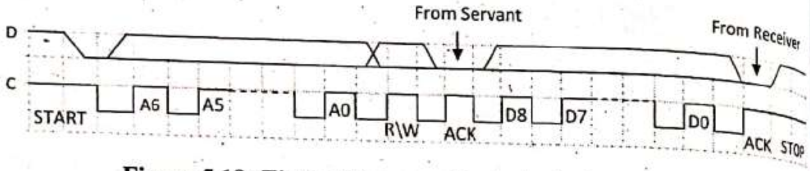


Figure 5.19: Timing diagram of a typical read/write cycle

A typical I2C byte write cycle operates as follows:

- The master initiates the transfer with a start condition. Start condition is represented by a high to low transition of SDA line while the SCL is held high.
- Then, the address of the device to which the data is to be written is sent with most significant bit down to the least significant bit.
- For write operation, the master sends a zero after sending the address. And the slave acknowledges the transmission by holding the SDA line low during first ACK clock cycle.
- Next, the master transmits a byte of data with most significant bit first.
- The slave acknowledges the reception of data by holding the SDA line low during second ACK clock cycle.
- Finally, master terminates the transfer by generating a stop condition. Stop condition is represented by a low to high transition of SDA line while the SCL is held high.

ii. Serial Peripheral Interface (SPI)

The serial peripheral Interface bus is a synchronous serial communication interface specification used for short distance communication. It is used to send data between processors, controllers and small peripherals. It uses separate clock and data lines along with a select line to choose the device that should be communicated with. The SPI bus consists of four logic signals: SCLK – serial clock, MOSI – Master Output Slave Input, MISO – Master Input-Slave Output, and SS – Slave select. The SPI bus can operate with a single master device and with one or more slave devices. Full duplex communication, higher throughput, simple software and hardware implementation, etc are few characteristics of SPI protocol.

iii. Control Area Network (CAN)

CAN is an international standardization organization (ISO) defined serial communications bus originally developed for the automotive industry to replace the complex wiring harness with a two wire bus. The specification calls for high immunity to electrical interference and the ability to self-diagnose and repair data errors. These features have led to CAN's popularity in a variety of industries including building automation, medical, and manufacturing.

Some of the characteristics of the CAN protocol includes high-integrity serial data communications, real-time support, data rates up to 1 Mbit/s, error detection and confinement capabilities. Balanced differential signaling in CAN protocol not only reduces noise coupling but also allows high signaling rates over twister pair cable. The CAN protocol incorporates five methods of error checking which forces transmitting node to resent the message until it is received correctly. But if the error limit is reached then the faulty node is deprived of transmit capability. Faulty nodes are automatically dropped from the bus, which prevents any single node from bringing a network down. This error containment also allows nodes to be added to a bus while the system is in operation, otherwise known as hot-plugging. It implements a non-destructive, bit-wise arbitration in which the node winning arbitration continues with the message without being corrupted by another node. The high speed ISO 11898 standard specifications are given for a maximum signaling rate of 1 Mbps with a bus length of 40m with a maximum of 30 nodes.

Scanned with CamScanner

To summarize, the protocol defines data packet format and transmission rules to prioritize messages, guarantee latency times, allow for multiple masters, handles transmission errors, retransmit corrupted messages, and distinguish between a permanent failures of a node versus temporary errors .

iv. FireWire

FireWire is a serial bus protocol for high-speed data transfer. It was initiated by Apple and developed by IEEE P1394 group so may refer this protocol as IEEE 1394. It supports mass information transfer and allows peer to peer device communication without the involvement of system memory or CPU. Some of the characteristics of FireWire protocol include transfer rates up to 400 Mbit/s, plug and play and hot swapping, packet based layered design structure and provision of power through the cable. Also, the 64 bit addressing allows a local-area network to consist of 1023 sub-networks, each consisting of 63 nodes. FireWire devices are organized at the bus in a tree or daisy chain topology. In arbitration, the closest node requesting for the data transfer gets the high priority. It provides two types of data transfer: asynchronous and isochronous. In asynchronous, data transfer can be initiated as a given length of data arrives in a buffer. But, in isochronous data transfer, data flows at a pre-set rate.

v. Universal Serial Bus (USB)

The universal serial bus (USB) protocol was designed to connect a wide range of peripherals to a computer, including pointing devices, displays, data storage, communication devices and other devices. It standardized the connection of computer peripherals to personal computers, both to communicate and to supply electric power. The original USB 1.0 specification defined data transfer rates of 1.5 Mbit/s for low data rate devices and 12 Mbit/s for high speed devices. The transfer rate for USB 2.0 is 480 Mbit/s while for USB 3.0 it can go up to 5 Gbit/s. USB On-The-Go is the special feature of USB in which two USB devices communicate with each other without requiring a separate USB host. USB uses a tiered star topology, which means some USB devices can serve as connection ports for other USB peripherals. USB hubs and standalone hubs can be used to provide handful of convenient USB ports. USB host controllers manage and control the driver software and bandwidth required by each peripheral connected to the bus.

2. Parallel Protocols

i. PCI Bus

The peripheral component interconnect (PCI) bus is a high-performance bus for attaching hardware devices in a computer. It is synchronous bus architecture with all data transfers being performed relative to a system clock. The maximum clock rate can go up to 66MHz; however, use of 33MHz is very common in personal computers. So the transfer rate can vary from 132 to 512 MB/s. PCI implements a 32-bit multiplexed address and Data bus which allows reduced pin count on the PCI connecter resulting in lower cost and smaller package size. It supports rigorous auto configuration mechanisms which allow identification of the type of device and the company that produced it. In PCI, any device has the potential to take control of the bus and initiate transactions with any other device making multiple master implementations easier which otherwise had been difficult.

ii. ARM Bus

ARM bus was designed to connect and manage different function blocks in a system on a chip (SoC) designs. It supports 32-bit data transfer and 32-bit addressing and is implemented using synchronous data transfer architecture. The transfer rate is the function of the clock speed used in a particular application. The ARM Advanced Microcontroller Bus Architecture is an open-standard for on chip interconnection.

3. Wireless Protocols

i. Infrared Data Association (IrDA)

The infrared data association (IrDA) is an international organization that creates and promotes infrared data interconnection standards. It provides specifications for a complete set of protocols for wireless infrared communications. IrDA has been implemented in portable devices like smart

Scanned with CamScanner

phones, laptops, cameras, etc. It is designed to support communication between two devices over point to point infrared at speeds between 9.6 kbps and 4 Mbps. Simplicity and low cost of IrDA hardware makes it an attractive option. Also line of sight, very low bit error rate and physically secure data transfer are few important features of IrDA. Other wireless technologies with no requirement of direct line of sight have displaced IrDA. However, it is still applicable where interference makes radio based wireless technologies unusable.

ii. Bluetooth

Bluetooth is a wireless technology standard for exchanging data over short distances from fixed and mobile devices. It operates at frequencies between 2402 and 2480 MHz which is the globally unlicensed Industrial, Scientific and Medical (ISM) 2.4 GHz short-range frequent band. Since Bluetooth uses a radio-based link, it does not require line of sight for communication. Bluetooth 4.0 may provide the transfer rate of up to 25Mbps. Bluetooth is a packet based protocol with a master-slave structure and one master may communicates up to maximum of seven slave devices. Low power consumption and short range based communication is the typical feature of Bluetooth. Permitted transmission power and range of communication depend on the radios class. For class 3 radio, range is up to 1m with max permitted power of about 1mW. The range is about 10m and 2.5mW power is permitted in case of class 2 radios, and class 1 radios have a range of about 100m and 100mW of transmission power. Handsfree headset and wireless speakers are two, out of many, examples using Bluetooth.

iii. IEEE 802.11

IEEE 802.11 is a set of media access control (MAC) and physical layer (PHY) specifications for implementing wireless local area network. IEEE 802.11, often termed as Wi-Fi, has the data transfer rate of around 1-2Mbps. IEEE 802.11 has a variety of standards, each with a letter suffix; 802.11a, 802.11b, 802.11g, 802.11n standards are quite common. All these 802.11 Wi-Fi standards operate within the ISM frequency bands. Generally, 2.4 GHz band is common which also makes the chips easier and cheaper to manufacture. The data rate can go up to 54 Mbps with some standard, while few latest standards may support up to 6.75 Gbit/s.

The PHY layer defines the means of transmitting bits over a physical link connecting network nodes. It provides an electrical, mechanical and procedural interface to the transmission medium. Modulation, line coding, synchronization are few functions, out of many, performed by the physical layer. The MAC layer provides the addressing and channel access control mechanism that ensures the communication of several nodes within a shared medium. Each device is assigned a unique serial number which is also known as MAC address. Unique MAC address makes it possible for data packets to be delivered to a destination within a sub-network. Multiple access protocol allows several stations connected to the same physical medium to share it. The most common multiple access protocol is the contention based Carrier Sense Multiple Access with Collision Avoidance (CSMA/CD) protocol.

Scanned with CamScanner

Chapter 6

REAL TIME OPERATING SYSTEM

- Operating System Basics
- Task Process and Threads
- Multiprocessing and Multitasking
- Task Scheduling
- Task Synchronization
- Device Drivers

6.1 Operating System Basics

The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services. The primary function of an operating system is

- Make the system convenient to use
- Organize and manage the system resources efficiently and correctly

The following figure shows the basic components of an operating system and their interfaces with rest of the world.

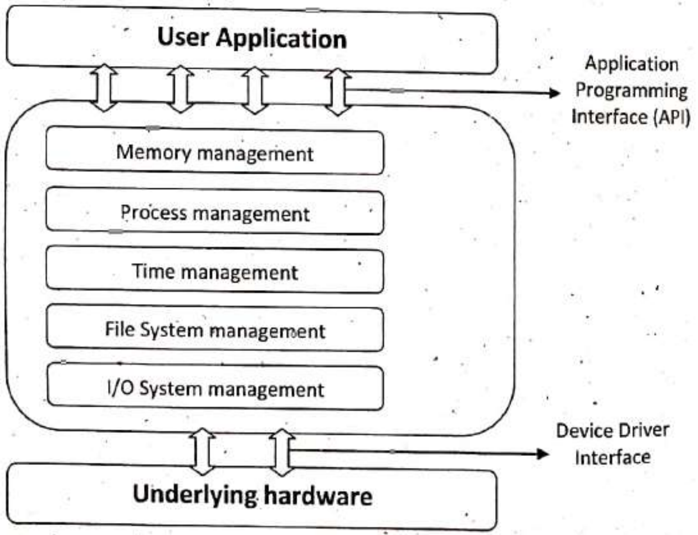


Figure 6.1: Operating system architecture

Comparison of General Purpose OS (GPOS) with Real Time OS (RTOS)

General purpose operating system is software that manages all the system resources and provides common service to all programs running in the system. In case of real time operating system, along with management and services it performs certain function within a specified time constraint. However, both operating systems provide a number of services to application programs and users. Application programming interfaces (API) or system calls are the medium through which the services are accessed by the applications.

The differences between GPOS and RTOS can be clarified using following parameters.

i. **Deterministic nature**

RTOS are deterministic in nature; the time required to execute the services is fixed. However, there may not be fixed time defined for any service in case of GPOS.

ii. **Task scheduling**

RTOS uses priority based preemptive scheduling, while scheduling in GPOS is defined so as to achieve high throughput. In RTOS, high priority process execution will override the low priority ones. In GPOS, high priority process may be delayed to perform several low priority tasks.

iii. **Time critical systems**

RTOS is used in time critical systems in which delay in processing can result in undesirable consequences. However, GPOS are implemented in non-time critical systems.

iv. **Preemptive kernel**

The kernel of an RTOS is preemptive where as a GPOS kernel is non-preemptive. In preemptive kernel, the high priority user process can preempt a kernel call. In other words, the execution of low priority system process can be stopped by high priority user process.

v. **Priority inversion problem**

Priority inversion problem is seen in RTOS in which the high priority task has to wait for the shared resource occupied by low priority task.

Scanned with CamScanner

This results in execution of low priority task first rather than high priority task.

6.1.1. The Kernel

The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services. It acts as the abstraction layer between system resources and user applications. The kernel contains different services for handling the following.

1. **Process Management**

   It includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the process control block (PCB), Inter Process Communication and Synchronization, process termination/deletion, etc.

2. **Primary Memory Management**

   The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The memory management unit (MMU) of the kernel is responsible for

   - Keeping track of which part of the memory area is currently used by which process
   - Allocating and de-allocating memory space on a need basis (dynamic memory allocation)

3. **File System Management**

   File is a collection of related information. A file could be a program, text files, word documents, audio/video files, etc. Each of these files differs in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The file system management service of Kernel is responsible for

   - The creation, deletion and alteration of files and directories
   - Saving of files in the secondary storage memory

   - Providing automatic allocation of file space based on the amount of free space available
   - Providing flexible naming convention for the files

4. **I/O System (Device) Management**

   Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well-structured OS, the direct assessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel. The kernel maintains a list of all I/O devices of the system. The list may be available in advance and recent kernel dynamically updates the list of available devices. The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low level system calls, which are implemented in a service, called device drivers. The device manager is responsible for

   - Loading and unloading of device drivers
   - Exchanging information and the system specific control signals to and from the device

5. **Secondary Storage Management**

   The secondary storage management deals with managing the secondary storage memory devices that are connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most systems, the secondary storage is kept in disks (hard disk). The secondary storage management service of kernel deals with

   - Disk storage allocation
   - Disk scheduling (time interval at which the disk is activated to backup data)
   - Free disk space management

6. **Protection Systems**

   Modern operating systems are designed in such a way to support multiple users with different levels of access permissions (For example: Administrator, Standard, Restricted, Guest, etc).

Scanned with CamScanner

Implementing security policies to restrict the access to both user and system resources by different applications or processes or users, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. Some application may not be granted with permission to make use of some of the system resources.

7. **Interrupt Handler**

Kernel provides a mechanism to handle all external/internal interrupts generated by the system. Based upon the priority of the interrupt the process either runs in the foreground or background. Depending on the type of operating system, a kernel may contain lesser number of services of more number of services which may include network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc.

6.1.2. Real Time Kernel

A real-time kernel is software that manages the time of microprocessor to ensure that time-critical events are processed as efficiently as possible. The use of a kernel simplifies the design of embedded systems because it allows the system to be divided into multiple independent elements called tasks. A task is a simple program that thinks it has the microprocessor all to itself. Each task has its own stack space and each task is dedicated to a specific function in your product. The kernel is responsible for keeping track of the top-of-stack for each of the different tasks. Most real-time kernels are preemptive. This means that the kernel will always try to execute the highest priority task that is ready to run. Scheduling is a function performed by the real-time kernel to determine whether a more important task needs to run. Real-time kernels provide a mechanism called a semaphore that tasks need to use in order to access a shared variable, array, data structure, and I/O device.

6.1.3. Kernel Space and User Space

The program code corresponding to the kernel applications/services are kept in a contiguous area of primary memory and are protected from un-authorized access by user programs/applications. The memory space at which the kernel code is located is known as 'Kernel Space'. Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as 'User Space'. User space is the memory area where user applications are loaded and executed. The partitioning of memory into kernel and user space is purely OS dependent.

6.1.4. Types of Kernel

Based on the kernel architecture/design, kernels can be classified into Monolithic and Micro.

1. **Monolithic Kernel**

In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. It runs all basic system services and provides powerful abstraction of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel. The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. The inclusion of all basic services in kernel space leads to different drawbacks such as requirement of large kernel size, lacking extensibility, poor maintainability. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel.

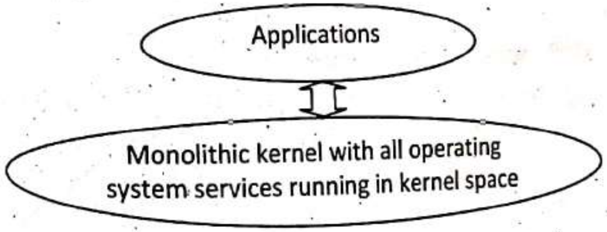

Applications
Monolithic kernel with all operating system services running in kernel space

Figure 6.2: The monolithic kernel model

2. **Microkernel**

The microkernel design incorporates only the essential set of operating system services such as communication and I/O control into the kernel. The rest of the operating system services are implemented in programs known as 'Servers' which runs in user space. It is more stable than monolithic as the kernel is unaffected even if the server fails; Memory Management, process Management, timer systems and interrupt handlers are the essential services which forms the part of microkernel. Microkernel based design approach offers the following benefits.

- **Robustness:**

  If a problem is encountered in any of the service, which runs as 'Server' application, the same can be reconfigured and re-started without the need for re-starting the entire OS.

- **Configurability:**

  Services can be changed, updated without corrupting the essential services residing within the microkernel.

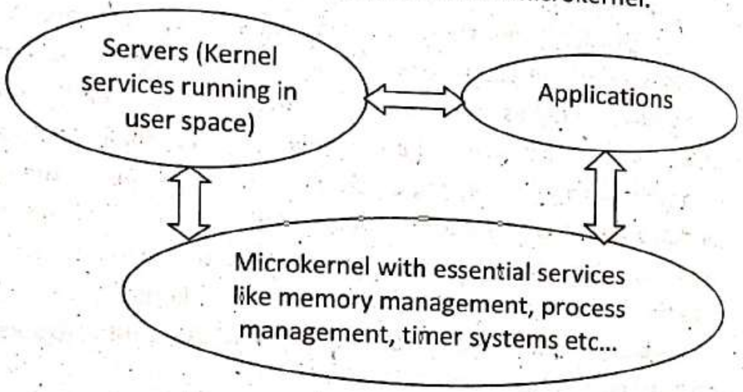

Servers (Kernel services running in user space)
Applications
Microkernel with essential services like memory management, process management, timer systems etc...

Figure 6.3: The microkernel model

6.2 Task Process and Threads

A task is defined as a program in execution and related information maintained by OS for that program. Task is also known as 'job' in the operating system context. A program or part of it in execution is also called a 'Process'. The terms 'task', 'job' and 'process' refer to the same entity in the operating system context and most often they are used interchangeably.

6.2.1 Process

A process is an instance of a program or part of program in execution. A process requires various system resources such as the CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc. A program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on the disk (executable file). A process is an active entity. A program becomes a process when an executable file is loaded into memory.

1. **Structure of a Process**

A process holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. From a memory perspective, the memory occupied by the process is separated into three regions, stack memory, data memory and code memory.

The stack memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process.

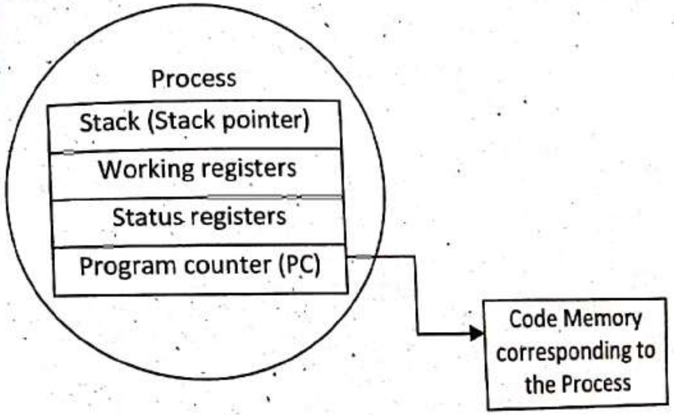


Figure 6.4: Structure of a process

2. **Process States and State Transition**

The process traverses through a series of states during its transition from the newly created state to the terminated state. The cycle through which a process changes its state from 'newly created' to 'execution completed' is known as 'process life cycle'.

- **Created state:** it is the state at which a process is being created. The operating system recognizes a process in the created state but no resources are allocated to the process.
- **Ready state:** It is the state, where a process is loaded into the memory and awaiting the processor time for execution. The process is placed in the ready list queue maintained by the OS.

- **Running state:** It is the state where the source code instructions corresponding to the process are being executed. The process execution happens in this state.
- **Blocked/waiting state:** it refers to a state at where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur or waiting for getting access to a shared resource.
- **Terminated/completed state:** It is a state where the process completes its execution.

Different OS kernel can have different name for the state associated with a task. Created state may be stated as dormant state, waiting state may be restated as Pending state and so on.

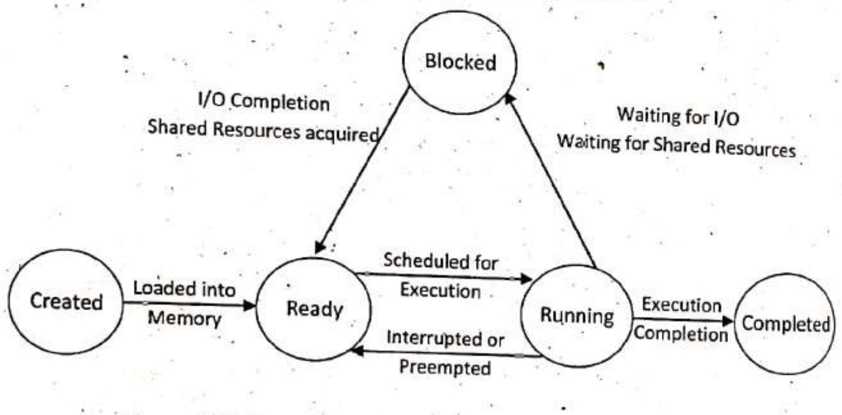


Figure 6.5: Process states and state transition representation

3. Process Control Block (PCB)

Each process is represented in the OS by a process control block. A PCB serves as a repository for any information that may vary form process to process. A PCB contains many pieces of information associated with a specific process.

- **Process state:** The state may be new, ready, running, waiting/blocked/pending or completed.
- **Program counter:** It indicates the address of next instruction to be executed for current process.

- **CPU registers:** They include accumulators index registers, stack pointers, general purpose registers along with any status registers. The content of PC along with the state information of a process must be saved when an interrupt occurs.
- **CPU scheduling information:** This information includes the process priority and the pointers to the scheduling queues
- **Memory management information:** This information includes the value of the base registers, page tables depending upon the memory system used by the OS.
- **Accounting information:** This information includes the amount of CPU time, time limits and process numbers.
- **I/O status information:** It includes the list of I/O devices allocated to a process.

6.2.2. Threads

A thread, basic unit of CPU utilization, is a single sequential flow of control within a process. A process can have many threads of execution. Different threads which are part of process share the data memory, code memory and the heap memory.

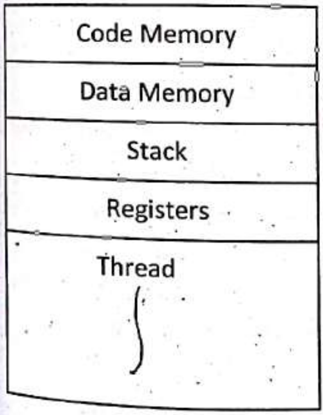


Figure 6.6: Single-threaded process

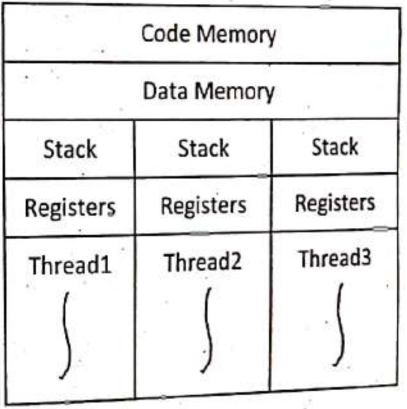


Figure 6.7: Multi-threaded process

However, the threads maintain their own thread status (CPU register value), program counter (PC) and stack. If a process has multiple threads of control, it can perform more than one task at a time. It is called a multithreaded process. If a process has a single thread of

Scanned with CamScanner

control it can perform a single task and is called single threaded process.

1. Concept of Multithreading

A process contain various sub-operations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the sub-functions of a task are executed in sequence, the CPU utilization may not be efficient. For example, if the process I waiting for a user input, the CPU enters the wait state for the event, and for the process execution also enters a wait state. If a process is split into different threads carrying out the different sub-functionalities of the process, the CPU can be effectively utilized and when the thread corresponding to the I/O operation enters the wait state, another thread which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilization of the processor time and resources.

The benefits of multi-threaded can be broken down into the following major categories:

- **Responsiveness:** Multi-threading on interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Economical:** Process creation is costly in terms of allocating memory and resources. Multiple thread creation within a process is economical because threads share the resources of the process to which they belong (code, data, heap memory). Creation of threads and context-switch of threads is economical.
- **Utilization of multiprocessor architecture:** The benefits of multithreading can be greatly increased in a multi-processor architecture, where threads may be running in parallel in different processors. A single threaded process can only run on one processor, no matter how many processors are available. Multi-threading on a multiprocessor machine increases concurrency.

- **Efficient CPU utilization:** CPU is engaged all the time. Since a process is split into different threads, when a thread enters a wait/block state, the CPU can be utilized by other threads of the process. This speeds up the execution of a process.

2. User level & Kernel level threads

i. User Level Threads

The user level threads don't have kernel/OS support and they exists only in a running process. Even if a process contains multiple user level threads, the OS treats it as a single thread. It is the responsibility of the process to schedule each thread as and when ever required. User level threads of a process are non-preemptive at the thread level from the OS perspective.

ii. Kernel Level Threads

These are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution of another kernel thread based on the scheduling policies implemented by the OS. Kernel level threads are pre-emptive.

3. Relationship Between User Level Thread and Kernel Level Thread

There are many ways for binding/connecting user level threads with kernel level threads.

- **Many to one model:** Many user level threads are mapped to a single kernel thread. The kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU.
- **One to one model:** Each user level thread is bonded to a kernel/system level thread. It provides more concurrency than the many to one model by allowing another thread to run when a thread makes a blocking system call. It allows multiple threads to run in parallel on multiprocessor. Creating a user level thread requires creating a corresponding kernel level thread.
- **Many to many model:** It multiplexes many user level threads to a smaller or equal no of kernel level threads. Developers can create as many user level threads as necessary and the

Scanned with CamScanner

corresponding kernel level threads can run in parallel on a multiprocessor. When a thread performs a blocking system call the kernel can schedule another thread for execution.

4. Thread Libraries

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing thread library.

The first approach is to provide a library entirely by the user space with no kernel/OS support. All code and data structure for library exists in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel level library supported directly by the OS. In this case, code and data structure for the library exists in the kernel space. Invoking a function in the API for the library, results in a system call to the kernel.

There are three main thread libraries that are used today.

i. **POSIX threads:** POSIX stands for Portable OS Interface. The POSIX standard for defining API, for thread creation and management, is pthreads. Pthreads library defines the set of POSIX thread creation and management functions in C language. Pthread may be provided as either a user level or a kernel level library.

Thread Call	Description
pthread_create()	Creates a new thread
pthread_exit()	Terminates the calling thread
pthread_join()	Blocks the current thread and waits until the completion of the thread pointed by it.
pthread_yield()	Releases the CPU to let another thread run
pthread_attr_init()	Create and initialize a thread's attributes
pthread_attr_destroy()	Releases a thread's attributes

ii. **Win32 threads:** Win32 threads are supported by various flavors of the windows OS. The win32 API libraries provide a standard set of win32 thread creation and management function. Win32 thread library is a kernel level library.

Thread Call	Description
CreateThread()	Creates a new thread
SuspendThread()	Temporarily suspends thread execution
ResumeThread()	Wakes up a suspended thread
ExitThread()	It terminates a thread and allocates the thread stack resources along with other resources that were held by it.

iii. **Java threads:** Java threads are the threads supported by Java programming language. The java thread class 'Thread' is defined in the package 'java.lang'. The java thread API allows thread creation and management directly in the java programs. Since a java virtual machine runs on the top of host operating system, the JAVA thread API on the top of a host OS, the JAVA thread API typically implemented using a thread library available on the host system. This means that on windows system, java threads are typically implemented using the win32 API. UNIX and LINUS systems user pthreads.

Thread Call	Description
Start()	Allocates memory and initializes a new thread in JAVA
Yield()	A running thread enters the ready state
Sleep()	A thread enters the suspend state
Wait()	A thread enters a blocked state
Stop()	Terminates a thread and de-allocates resources

Scanned with CamScanner

5. **Difference between Thread and Process**

Thread	Process
It is a single unit of execution and is a part of the process	A process is a program in execution and combines one or more threads
A thread shares the code, data, heap memory with other threads of the same process	A process has its own code, data and stack memory
A thread cannot live independently	A process contains at least one thread
Threads are very inexpensive to create	Processes are expensive to create. Involves many OS overhead
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

6.3 Multiprocessing and Multitasking

Multiprocessing describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing are called multiprocessor system. Multiprocessor systems possess multiple CPUs/processors and can execute multiple processes simultaneously. The ability of an OS to have multiple programs in memory, which are ready for execution, is referred as multiprogramming.

In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes. The ability of an operating system to hold multiple processes in memory and switch the processor from executing one process to another process is known as multitasking. Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves 'Context switching', 'Context saving' and 'Content retrieval'.

6.3.1. Context Switching

Each task may exist in any one of the different states (running, ready, blocked, etc). During the execution of an application program, individual tasks are continuously changing from one state to another. At any point of the execution, only one task is in running mode. During the process of state change, CPU control changes from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved.

The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

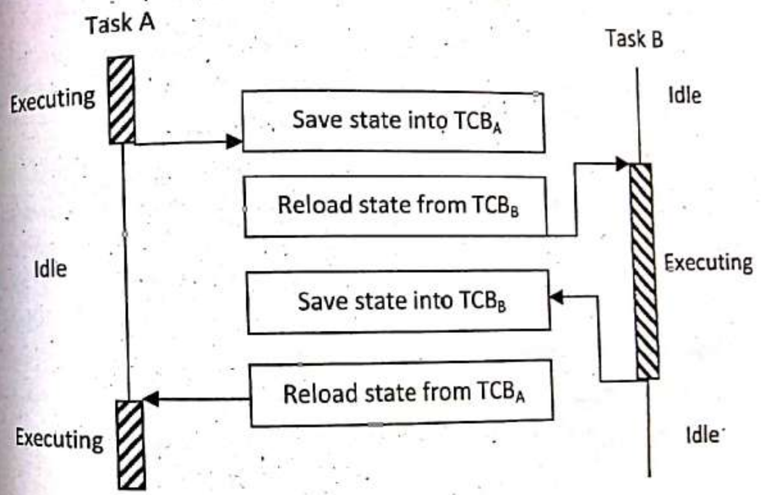


Figure 6.8: Simple context switching diagram

Context saving is the act of saving the current contents which contains the context details (register details, memory details, system resource usage details, etc. for the currently running process at the time of CPU switching. Context retrieval is the process of retrieving the saved context details for a process which is going to be executed due to CPU switching. Context switch time is pure overhead because the system does no useful work while switching.

Scanned with CamScanner

6.3.2. Types of Multitasking

Multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types.

1. **Co-operative Multitasking:** It is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. Any task/process can hold the CPU as much time as it wants. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

2. **Preemptive Multitasking:** It ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. The currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority.

3. **Non-preemptive Multitasking:** In non-preemptive multitasking, the process/task, which is currently given the CPU time, is allowed to execute until it terminates or enters the 'Blocked/Wait' state, waiting for an I/O or system resource. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' state, whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

6.4 Task Scheduling

Multitasking involves the execution switching among the different tasks. Determining which task/process is to be executed at a given point of time is known as task/process scheduling. Scheduling policies forms the guidelines for determining which task is to be executed when. The scheduling policies are implemented in an algorithm and it is run by the kernel as a service. The process scheduling decision may take place when a process switches its state to

- Ready state from running state
- Blocked/wait state from running state

- Ready state from blocked/wait state
- Completed state
- The selection of a scheduling criterion should consider the following factors
- **CPU utilization:** the scheduling criterion should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.
- **Throughput:** This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.
- **Turnaround time:** It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spend in the ready queue, time spent on completing the I/O operations, and the time spend in execution. The turnaround time should be a minimal for a good scheduling algorithm.
- **Waiting time:** It is the amount of time spent by a process in the 'ready' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.
- **Response time:** It is the time elapsed between the submission of a process and the first response, for a good scheduling algorithm, the response time should be as least as possible.

The operating system maintains various queues in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion. The various queues maintained by OS in association with CPU scheduling are:

- **Job queue:** Job queue contains all the processes of the system.
- **Ready queue:** contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution
- **Device queue:** contains the set of processes, which are waiting for an I/O device.

The scheduling algorithm can be classified as:

1. **Non-Preemptive Scheduling**

It is employed in systems which implements non-preemptive multitasking model. In this scheduling type, the currently executing

Scanned with CamScanner

task/process is allowed to run until it terminates or enters the wait state waiting for an I/O or system resources. Various types of non-preemptive scheduling are listed below.

i. **First Come First Served (FCFS)/FIFO Scheduling:** The FCFS scheduling algorithm allocates CPU time to the processes based on the order in which they enter the ready queue. The first entered process is serviced first. E.g. ticketing reservation system where people need to stand to a queue and the first person standing in the queue is serviced first.

ii. **Last Come First Served (LCFS)/LIFO Scheduling:** The LCFS scheduling algorithm also allocates CPU time to the Processes based on the order in which they are entered in the ready queue. The last entered process is services first.

iii. **Shortest Job First (SJF) Scheduling:** SJF scheduling algorithm sorts the ready queue each time a process relinquishes the CPU to pick the process with shortest estimated completion time. The process with the shortest estimated run time is scheduled first, followed by the nest shortest process, and so on.

iv. **Priority Based Scheduling:** This scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the ready queue. The SJF algorithm can be viewed as a priority based scheduling where each task is prioritized in the order of the time required to complete the task. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is the number ranging from 0 to the maximum priority supported by the OS. For windows CE operating system a priority number 0 indicates the highest priority.

2. Preemptive Scheduling

Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In this scheduling, every task in the ready queue gets a chance to execute. When and how often each process gets a chance to execute is dependent on the type of preemptive scheduling algorithm. In this scheduling method, the scheduler can preempt (stop temporarily) the currently executing process and select another task from the ready queue for execution. The task which is preempted by the scheduler is moved to the ready queue. The act of moving a running process into a ready queue by the scheduler, without the processes requesting for it is known as preemption. The different types of preemptive scheduling adopted in process scheduling are explained below.

i. **Preemptive SJF Scheduling / Shortest Remaining Time (SRT):** The preemptive SJF scheduling algorithm sorts the ready queue when a new process enters the ready queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process. If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution. Preemptive SJF scheduling is also known as "shortest remaining time (SRT) scheduling".

ii. **Round Robin Scheduling:** In this scheduling method, each process in the ready queue is executed for a pre-defined time slot. The execution starts with picking up the first process in the ready queue. It is executed for a pre-defined time slice and when the pre-defined time elapses or the process completes before the pre-defined time slice, the next process in the ready queue is selected for execution. Once each process in the ready queue is executed for the pre-defined time period, the scheduler picks the first process in the ready queue again for execution and the sequence is repeated. So, the round robin scheduling is similar to the FCFS scheduling but time slice preemption is added to switch the execution between the processes in the ready queue.

iii. **Priority Based Scheduling:** Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between processes. In preemptive scheduling, any high priority process entering the ready queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the ready queue is scheduled only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU.

Scanned with CamScanner

6.5 Task Synchronization

In a multitasking environment, multiple processes run concurrently and share the system resources. When two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location while the other process is trying to read from this. Then, an issue will arise and hence each process must be made aware of the access of the shared resources. The act of making processes aware of the access of the shared resources by each process to avoid conflicts is known as task/process synchronization. Various synchronization issues may arise if processes are not synchronized properly.

6.5.1. Task Communication/Synchronization Issues

1. **Racing**

Racing or race condition is the situation in which multiple processes compete each other to access and manipulate shared data concurrently. In a race condition the final value of the shared data depends on the process which acted on the data finally.

Suppose that two processes A and B have access to a shared variable Count:

Process A: Count = Count + 5

Process B: Count = Count + 10

Assume that process A and process B are executing concurrently in a time-shared, multi-programmed system.

Each statement requires several machine level instructions such as

For Count = Count + 5

A1: Load Ra, Count

A2: Add Ra, 05

A3: Store Count, Ra

For Count = Count + 10

B1: Load Rb, Count

B2: Add Rb, 10

B3: Store Count, Rb

In a time-shared or multi-processing system the exact instruction execution order cannot be predicted.

Scenario 1	**Scenario 2**
A1: Load Ra, Count A2: Add Ra, 05 A3: Store Count, Ra **Context Switch** B1: Load Rb, Count B2: Add Rb, 10 B3: Store Count, Rb	A1: Load Ra, Count A2: Add Ra, 05 **Context Switch** B1: Load Rb, Count B2: Add Rb, 10 B3: Store Count, Rb **Context Switch** A3: Store Count, Ra
Count is increased by 15	Count is increased by 5

2. **Deadlock**

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. In its simplest form, 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process. For instance, process A holds a resource x and it wants a resource y held by process B. Process B is currently holding resource y and it wants the resources x which is currently held by process A. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes.

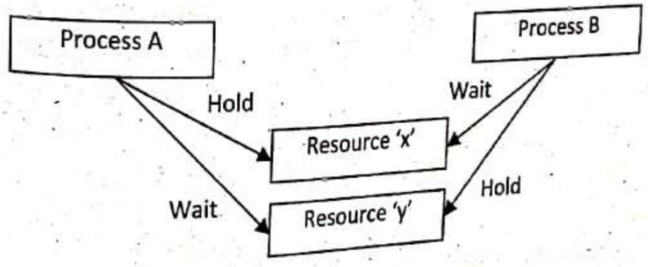


Figure 6.9: Scenarios leading to deadlock

Scanned with CamScanner

i. **Coffman Conditions:** The different conditions favoring a deadlock situation are listed below:

- **Mutual exclusion:** The criteria that only one process can hold a resource at a time. Processes should access shared resources with mutual exclusion.
- **Hold and wait:** The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
- **No resource preemptive:** The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.
- **Circular wait:** A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 ... Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, ..., Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on... This forms a circular wait queue.

ii. **Deadlock Handling**

A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

- **Ignore deadlocks:** Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of deadlock to occur.
- **Detect and recover:** This approach suggests the detection of a deadlock situation and recovery from it. OS keeps a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analyzing the resource graph by graph analyzer algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.
- **Avoid deadlocks:** Deadlock is avoided by the careful resource allocation techniques by the operating system.
- **Prevent deadlocks:** Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.
  - o Ensure that a process does not hold any other resources when it requires a resource.
  - o Ensure resources preemption.

3. Livelock

In a livelock condition, a process changes its state with time but is unable to make any progress in the execution completion. While in deadlock a process enters a wait state for a response and continues in that state forever without making any progress in the execution. For example, two people attempting to cross each other in a narrow corridor. Both the person moves towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target.

4. Starvation

In the multitasking context, starvation is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favoring high priority tasks and tasks with shortest execution time, etc.

6.5.2. Task Synchronization techniques

Task synchronization is essential for:

- Avoiding conflicts in resource access in a multitasking environment.
- Ensuring proper sequence of operation across processes.
- Communicating between processes.

Scanned with CamScanner

The code memory area which holds the program instructions for accessing a shared resource, shared variables is known as critical section. In order to synchronize the access to shared resources, the access to the critical section should be exclusive.

Consider two processes Process A and Process B running on a multitasking system. Process A is currently running and it enters its critical section. Before Process A completes its operation in the critical section, the scheduler preempts process A and schedules Process B for execution. Process B also contains the access to the critical section which is already in use by Process A. if process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

1. **Mutual Exclusion through Busy Waiting/Spin Lock**

The Busy Waiting technique uses a lock variable for implementing mutual exclusion. Each process/ thread checks this lock variable before entering the critical section. The lock is set to 1 by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to 0.

The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes always busy and forces the processes to wait for the availability of the lock for proceeding further. Hence this synchronization mechanism is known as Busy Waiting. This method is useful in handling scenarios where the processes are likely to be blocked for a shorter period of time on waiting the lock, as they avoid OS overheads on context saving and process re-scheduling. The drawback of Spin Lock based synchronization is that if the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting it. The busy waiting mechanism keeps the process always active, performing a task which is not useful and leads to the wastage of processor time and high power consumption.

2. **Mutual Exclusion through Sleep & Wakeup**

The Busy waiting mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery. In sleep and wakeup mechanism, when a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes Sleep and enters the blocked state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The sleep and wakeup mechanism for mutual exclusion can be implemented in different ways. We will discuss one method of sleep and wakeup mechanism using Semaphore.

Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the process which wants to access the share resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it. The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes in a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process. The hard disk of a system is a typical example for sharing the resource among a limited number of multiple processes. So, basically, semaphore can be categorized into two types.

i. **Binary Semaphore (Mutex):** The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. Mutex is a synchronization object provided by OS for process synchronization. Any process can create a mutex object and other processes of the system can use this mutex object at a time. The state of a mutex object is set to signaled when it is not owned by any process, and set to non-signaled when it is owned by any process.

Scanned with CamScanner

ii. **Counting Semaphore:** The counting semaphore limit the access of resources to fixed number of processes or threads. It maintains a count between zero and a value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to signaled when the count of the object is greater than zero. The count associated with a semaphore object is decremented by one when a process acquires it and the count is incremented by one when a process releases the semaphore object. The state of the semaphore object is set to non-signaled when the semaphore is acquired by the maximum number of processes that the semaphore can support.

6.6 Device Drivers

It is a piece of software that acts as a bridge between the OS and the hardware. The architecture of OS kernel will now allow direct device access from the user application. All devices related access should flow through OS kernel and the OS kernel routes it to the concerned hardware peripherals. Device drivers are responsible for initiating and managing the communication with hardware peripherals. They are responsible for establishing connectivity, initializing hardware (setting up various CPU registers) and transferring data.

Device drives which are part of OS are called built in drivers or on-board drivers. These drivers are loaded by OS at the time of booting the device and are kept in RAM. Device drivers which need to be installed for accessing a device are called installable drivers. Whenever the device is connected, the OS loads the corresponding driver into memory. Driver files are usually in the form of '.dll' files. Drivers can run either in user space or in kernel space. Device drivers which run in user space are called user mode driver and the driver which run in kernel space are called kernel mode drivers.

A device driver implements the following:

i. Device initialization and interrupt configuration:

The driver configures the different registers of the device. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. The basic interrupt configuration involves:

- Set the interrupt type (Edge triggered or Level triggered), enable the interrupts and set the interrupt priorities.
- Bind the interrupt with an interrupt request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify and interrupt the interrupt needs to be bonded to an IRQ.
- Register an Interrupt Service Routine (ISR) with an IRQ. ISR is the handler for an interrupt. In order to service an interrupt, an ISR should be associated with an IRQ.

ii. Interrupt handling and processing:

An interrupt is served based on its priority, and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. Since interrupt processing happens at kernel level, user application may not have direct access to the drivers to pass and receive data.

iii. Client interfacing:

The client interfacing implementation makes use of the interprocess communication mechanisms supported by the embedded OS for communicating and synchronizing with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in a shared buffer, the client interfacing code can signal an event.

SOLUTION TO IMPORTANT QUESTIONS

Problem 1:

Three processes with process IDs P1, P2, P3 with priorities 2, 3, 0 and estimated completion time 10, 5, 7 milliseconds respectively enter the ready queue together in the order P1, P2, P3. Calculate the Waiting Time and Turn Around Time for each process and also the Average Waiting Time and Average Turn Around Time. Assume there is no I/O waiting for the process. Use the following non-preemptive scheduling algorithms.

Scanned with CamScanner

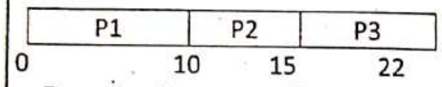
- **First Come First Serve Scheduling**
- **Priority Based Scheduling**
- **Shortest Job First Scheduling**

Solution:

Given information from the question are tabulated as shown below:

Process	Entry Time	Completion Time	Priority	Entered
P1	0	10	2	1st
P2	0	5	3	2nd
P3	0	7	0	3rd

A. First Come First Served Scheduling



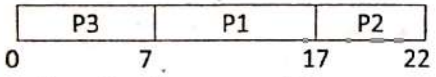
Execution Sequence of Processes

Waiting Time
= Execution Start Point – Entry Point
Turn Around Time
= Completion Point – Entry Point

Waiting Time calculation
P1 = (0-0) = 0ms
P2 = (10-0) = 10ms
P3 = (15-0) = 15ms
Average Waiting Time
= (0+10+15)/3
= 8.33ms

Turn Around Time calculation
P1 = (10-0) = 10ms
P2 = (15-0) = 15ms
P3 = (22-0) = 22ms
Average Turn Around Time
= (10 + 15 + 22)/3
= 15.67ms

B. Priority Based Scheduling



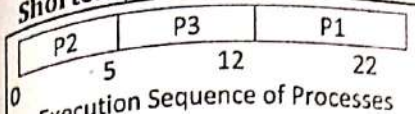
Execution Sequence of Processes

Waiting Time
= Execution Start Point – Entry Point
Turn Around Time
= Completion Point – Entry Point

Waiting Time calculation
P3 = (0 - 0) = 0ms
P1 = (7 - 0) = 7ms
P2 = (17 - 0) = 17ms
Average Waiting Time
= (0 + 7 + 17)/3
= 8ms

Turn Around Time calculation
P3 = (7 - 0) = 7ms
P1 = (17 - 0) = 17ms
P2 = (22 - 0) = 22ms
Average Turn Around Time
= (7 + 17 + 22)/3
= 15.33ms

C. Shortest Job First



Execution Sequence of Processes

Waiting Time
= Execution Start Point – Entry Point
Turn Around Time
= Completion Point – Entry Point

Waiting Time calculation
P2 = (0 - 0) = 0ms
P3 = (5 - 0) = 5ms
P1 = (12 - 0) = 12ms
Average Waiting Time
= (0 + 5 + 12)/3
= 5.67ms

Turn Around Time calculation
P2 = (5 - 0) = 5ms
P3 = (12 - 0) = 12ms
P1 = (22 - 0) = 22ms
Average Turn Around Time
= (5 + 12 + 22)/3
= 13ms

Problem 2:

Three processes with process IDs P1, P2, P3 with priorities 0, 1, 3 and estimated completion time 6, 9, 3 milliseconds respectively enter the ready queue together. If a new process P4 (priority 2) with estimated completion time 2ms enters the ready queue after 3ms of execution of P1. Calculate the Waiting Time and Turn around Time for each process and also the Average Waiting Time and Average Turn Around Time. Make use of following non-preemptive scheduling algorithm to solve the problem.

- **Shortest Job First (SJF) Scheduling**
- **Priority Based Scheduling**

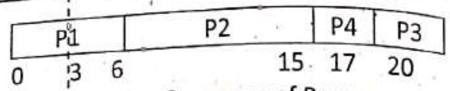
Solution:

A. Non - Preemptive SJF Scheduling

Given information from the question are tabulated as shown below:

Process	Entry Time	Completion Time	Priority
P1	0	6	0
P2	0	9	1
P3	0	3	3
P4	3ms after P1 starts	2	2

Scanned with CamScanner



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)
Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P1 = (0 - 0) = 0ms
P2 = (6 - 0) = 6ms
P4 = (15 - 3) = 12ms
P3 = (17 - 0) = 17ms
Average Waiting Time
= (0 + 6 + 12 + 17)/4
= 8.75ms

Turn Around Time calculation

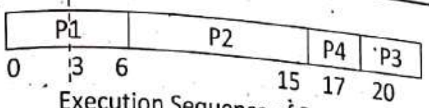
P1 = (6 - 0) = 6ms
P2 = (15 - 0) = 15ms
P4 = (17 - 3) = 14ms
P3 = (20 - 0) = 20ms
Average Turn Around Time
= (6 + 15 + 14 + 20)/4
= 13.75ms

Explanation: Entry point for three processes P1, P2 and P3 is same at 0ms but the process P4 enters only after 3ms of execution of P1. So, the entry point for P4 will be at 6ms. Regardless of the shortest completion time of P4, P4 will not halt the execution of P1 as the algorithm is non pre-emptive. However, after the execution of P1, there remain two processes P2 and P4 with completion time 9ms and 2ms respectively. Hence, P4 will start to execute after completion of P1 according to shortest job first scheduling.

B. Non - Preemptive Priority Based Scheduling

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	6	0
P2	0	9	1
P3	0	3	3
P4	3ms after P1 starts	2	2



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)
Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P1 = (0 - 0) = 0ms
P2 = (6 - 0) = 6ms
P4 = (15 - 3) = 12ms
P3 = (17 - 0) = 17ms
Average Waiting Time
= (0 + 6 + 12 + 17)/4
= 8.75ms

Turn Around Time calculation

P1 = (6 - 0) = 6ms
P2 = (15 - 0) = 15ms
P4 = (17 - 3) = 14ms
P3 = (20 - 0) = 20ms
Average Turn Around Time
= (6 + 15 + 14 + 20)/4
= 13.75ms

Problem 3:

Three processes P1, P2, P3 with estimated completion time 9, 4, 6 ms and priorities 1, 3, 2 respectively enters the ready queue together. A new process P4 with estimated completion time 4ms and priority 0 enters the ready queue after 2 ms of start of execution of P1. Calculate the Waiting Time and Turn Around Time for each process. Also Calculate the Average Waiting Time and Average Turn Around Time, using the Preemptive Shortest Job First Scheduling and Priority Based Scheduling.

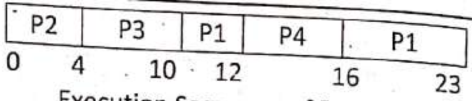
Solution:

A. Preemptive SJF Scheduling

Given information from the question are tabulated as shown below.

Process	Entry Time	Completion Time	Priority
P1	0	9	1
P2	0	4	3
P3	0	6	2
P4	2ms after P1 starts	4	0

Scanned with CamScanner



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point) + Halted time

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P2 = (0 - 0) = 0ms

P3 = (4 - 0) = 4ms

P4 = (12 - 12) = 0ms

P1 = (10 - 0) + (16 - 12) = 14ms

Average Waiting Time

= (0 + 4 + 0 + 14)/4

= 4.5ms

Turn Around Time calculation

P2 = (4 - 0) = 4ms

P3 = (10 - 0) = 10ms

P4 = (16 - 12) = 4ms

P1 = (23 - 0) = 23ms

Average Turn Around Time

= (4 + 10 + 4+ 23)/4

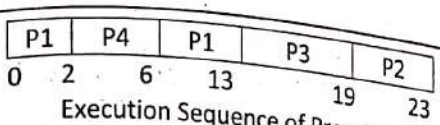
= 10.25ms

Explanation: Entry point for three processes P1, P2 and P3 is same at 0ms but the process P4 enters only after 2ms of execution of P1. So, the entry point for P4 will be at 12ms. At 12ms, there are two processes remaining; P1 with 7ms left to execute and P4 with 4ms. Since P4 is shorter compared to remaining part of P1, P4 will halt the execution of P1 at 12ms and starts its own execution. After P4 completes its execution at 16ms, P1 resumes:

B. Preemptive Priority Based Scheduling

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	9	1
P2	0	4	3
P3	0	6	2
P4	2ms after P1 starts	4	0



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point) + Halted time

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P1 = (0 - 0) + (6 - 2) = 4ms

P4 = (2 - 2) = 0ms

P3 = (13 - 0) = 13ms

P2 = (19 - 0) = 19ms

Average Waiting Time

= (4 + 0 + 13 + 19)/4

= 9ms

Turn Around Time calculation

P1 = (13 - 0) = 13ms

P4 = (6 - 2) = 4ms

P3 = (19 - 0) = 19ms

P2 = (23 - 0) = 23ms

Average Turn Around Time

= (13 + 4 + 19 + 23)/4

= 14.75ms

Problem 4:

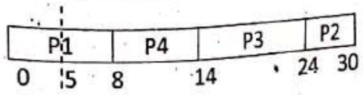
Explain process life cycle with process state diagram. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 6, 10ns and priorities 0, 3, 2 (0-highest priority and 3- lowest priority) respectively, enters a ready queue together in order P1, P2, P3 (assume P1 is present in the ready queue when scheduler picks it up and P2 and P3 enter the queue after that). Now the process P4 with estimated completion time 6ms and priority 1 enters the ready queue after 5ms of execution of P1. Calculate waiting time and TAT for each process and average waiting time and TAT. Assume there is no I/O waiting for the processes and priority-based scheduling.

[2076 Bhadra]

Solution:

Given information from the question are tabulated as shown below:

Process	Entry Time	Completion Time	Priority
P1	0	8	0
P2	0	6	3
P3	0	10	2
P4	5ms after P1 starts	6	1



Execution Sequence of Processes

Scanned with CamScanner

Waiting Time = (Execution Starting Point – Entry Point)

Turn Around Time = (Completion Point – Entry Point

Waiting Time calculation

P1 = (0 - 0) = 0ms

P2 = (24- 0) = 24ms

P3 = (14- 0) = 14ms

P4 = (8- 5) = 3ms

Average Waiting Time = (0 + 24 + 14 + 3)/4

= 10.25ms

Turn Around Time calculation

P1 = (8 - 0) = 8ms

P2 = (30 - 0) = 30ms

P3 = (24- 0) = 24ms

P4 = (14 - 5) = 9ms

Average Turn Around Time= (8 + 30 + 24 + 9)/4

=17.75ms

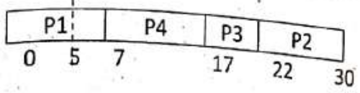
Problem 5:

Three processes with process IDs P1, P2, P3 with estimated completion time 7, 8, 5 ms and priorities 0, 3, 2 (0-highest priority and 3- lowest priority) respectively, enters a ready queue together in order P1, P2, P3 (assume P1 is present in the ready queue when scheduler picks it up and P2 and P3 enter the queue after that). Now the process P4 with estimated completion time 10ms and priority 1 enters the ready queue after 5ms of execution of P1. Calculate waiting time and TAT for each process and average waiting time and TAT. Assume there is no I/O waiting for the processes and priority-based scheduling. [2076 Baishakh]

Solution:

Given information from the question are tabulated as shown below:

Process	Entry Time	Completion Time	Priority
P1	0	7	0
P2	0	8	3
P3	0	5	2
P4	5ms after P1 starts	10	1



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)

Turn Around Time = (Completion Point – Entry Point

Waiting Time calculation

P1 = (0 - 0) = 0ms

P2 = (22- 0) = 22ms

P3 = (17- 0) = 17ms

P4 = (7- 5) = 2ms

Average Waiting Time = (0 + 22 + 17+ 2)/4

= 10.25ms

Turn Around Time calculation

P1 = (7 - 0) = 7ms

P2 = (30 - 0) = 30ms

P3 = (22- 0) = 22ms

P4 = (17 - 5) = 12ms

Average Turn Around Time= (7 + 30 + 22 + 12)/4

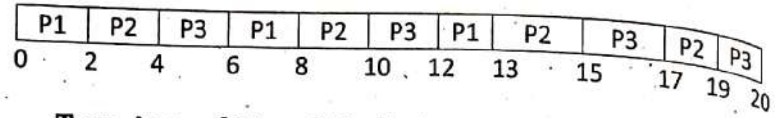
=17.75ms

Problem 6:

Three processes with process IDs P1, P2, P3 with estimated completion time 5, 8, 7 ms respectively, enter the ready queue together in order P1, P2, P3. Calculate waiting time and turnaround time for each process and average waiting time and average turnaround time using Round Robin algorithm with time slice 2ms. [2075 Bhadra]

Scanned with CamScanner

Solution:



Turn Around Time Calculation

P1 = 13 -- 0 = 13ms

P2 = 19 – 0 = 19ms

P3 = 20 – 0 = 20ms

Average Turn Around Time = (13 + 19 + 20)/3

= 17.33ms

Waiting Time Calculation

(Waiting Time = Turn Around Time – Completion Time)

P1 = 13 – 5 = 8ms

P2 = 19 – 8 = 11ms

P3 = 20 – 7 = 13ms

Average Waiting Time = (8 + 11 + 13)/3

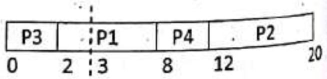
= 10.66ms

Problem 7:

Three processes with process IDs P1, P2, P3 with estimated completion time 6, 8, 2ms respectively enters the ready queue together in the order P1, P2, P3. Process P4 with the estimated time 4ms enters the ready queue after 1ms after the start of execution of P1. Calculate the waiting time and TAT for each process and the average waiting time and TAT in non-preemptive Shortest Job First Scheduling. **[2075 Baishakh]**

Solution:

Process	Entry Time	Completion Time
P1	0	6
P2	0	8
P3	0	2
P4	1ms after P1 starts	4



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)
Turn Around Time = (Completion Point – Entry Point)

Waiting Time Calculation

P3 = (0 - 0) = 0ms

P1 = (2 - 0) = 2ms

P4 = (8 - 3) =5ms

P2 = (12 - 0) = 12ms

Average WT= (0 + 2 + 5 + 12)/4

= 4.75ms

Turn Around Time Calculation

P3 = (2 - 0) = 2ms

P1 = (8 - 0) = 8ms

P4 = (12 - 3) = 9ms

P2 = (20 - 0) = 20ms

Average TAT = (2 + 8 + 9+ 20)/4

= 9.75ms

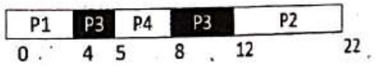
Problem 8:

Three processes with process ID P1, P2, P3 with estimated completion time 4, 10, 5ms and priorities 1, 3, 2 respectively enters the ready queue together. A new process P4 with estimated completion time 3ms and priority 0 enters the ready queue after 5ms of start of operation. Calculate WT, TAT for each process and calculate AWT and ATAT using preemptive priority-based scheduling algorithms. **[2074 Bhadra]**

Solution:

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	4	1
P2	0	10	3
P3	0	5	2
P4	5ms after start of operation	3	0



Execution Sequence of Processes

Turn Around Time calculation

P1 = (4- 0) =4ms

P2 = (22-0) = 22ms

P3 = (12- 0) = 12ms

P4 = (8-5) = 3ms

Average Turn Around Time

= (4+ 22 + 12 + 3)/4

= 10.25ms

Waiting Time calculation

P1 = (0 - 0) = 0ms

P2 = (12 - 0) = 12ms

P3 = (4- 0) + (8-5) = 7ms

P4 = (5- 5) = 0ms

Average Waiting Time

= (0 + 12 + 7 + 0)/4

=4.75ms

Scanned with CamScanner

Chapter 7

CONTROL SYSTEMS

- Introduction
- Open-Loop and Closed-Loop Control Systems Overview
- General Control Systems and PID Controllers
- Software Coding of PID Controller
- PID Tuning
- Practical Issues Related to Computer-Based Control
- Benefits of Computer-Based Control Implementation

7.1 Introduction

Control systems, a class of embedded systems, focus on tracking the reference input that is provided to the system. Initially the reference input is set and the output is more likely to track the same input regardless of the different external factors involved. The tracking can get difficult with the presence of disturbances. However, the system must be able to adjust to external factors for optimum performance. The objective of a control system is to track the reference output. The following figures represent good tracking and bad tracking respectively.

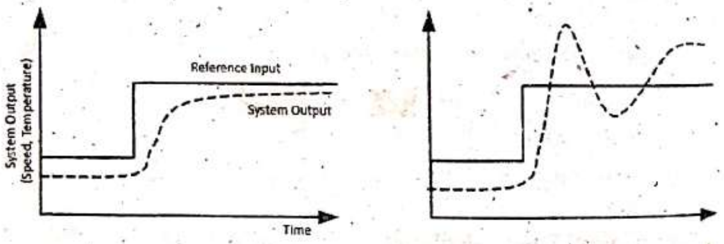


Figure 7.1: Good tracking and bad tracking

7.2 Open-Loop and Closed-Loop Control Systems Overview

7.2.1 Open-Loop Control Systems

Open-loop control systems are those systems in which the output has no influence on the control action of the input signal. It is also referred as feed-forward system or non-feedback system since the output is not fed back for comparison with the reference input. Also the controller is not aware about the tracking of reference input, so optimization is not possible. These systems are best utilized in case of predictable systems whose model is accurate and disturbance effect is minimal. In general, the open-loop control systems consist of following:

- **Plant**, which is also referred as a process, is the physical system to be controlled. Automobiles, fan, heater, disk etc. are few examples.
- **Output** is the aspect or attribute of the physical system that we are about to control. Speed, temperature can be taken as examples.
- **Reference** input is the desired value that is required to be observed as an output of the physical system. Desired speed, temperature set by the user represents a reference input.
- **Actuator** is the device that is used to control the input to the plant. Motor can be taken as an example of an actuator.
- **Controller** is the main processing part of the system which computes the input to the plant such that desired output is achieved based on given reference input.
- **Disturbance** is an undesirable input to the system that may cause the output to deviate from the desired reference input.

The general block diagram of open-loop control system is shown in the figure below.

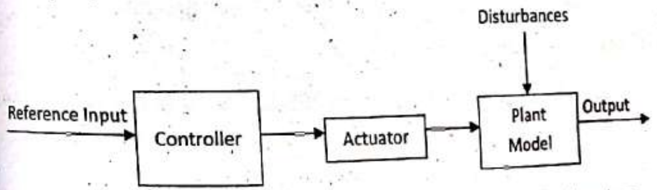


Figure 7.2: Block diagram of an open-loop control system

7.2.2 Closed-Loop Control Systems

Closed-loop control systems are the systems operating on feedback principle. In such system the output is fed back, compared with the reference input and error signal is produced. The controller processes the error signal and reduces the error to obtain the desired output. Since the controller is aware about the output variations, optimization can be done and optimum performance can be obtained by minimizing the error. Apart from the plant, output, reference, controller, actuator, and disturbances, closed-loop control system contains additional components as sensor and error detector.

- **Sensor** is used to sense the output of the system and is fed to the input where error is calculated.
- **Error detector** determines the error being produced in the system. Error is calculated by determining the difference between the output of the system and the reference input.

The general block diagram of closed-loop control system is shown in the figure below.

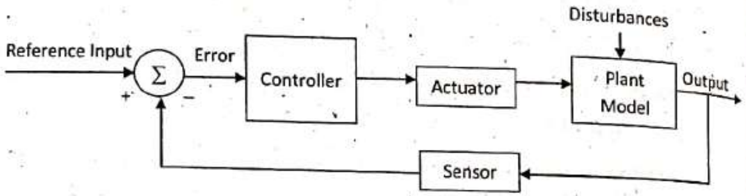


Figure 7.3: Block diagram of closed-loop control system

7.2.3 Comparison of Open-Loop and Closed Loop Control Systems

SN	Open-Loop Control System	Closed-Loop Control System
1.	Feed Forward System: Output is not fed back.	Feed Back System: Output is fed back and compared with input.
2.	It is simple and economical.	It is complex and expensive.
3.	Good calibration can lead to good accuracy but optimization is not possible	Feedback principle reduces error, increases accuracy and supports optimization
4.	It is slow and unreliable but stable.	It is fast and more reliable but unstable,

7.2.4 Open and Closed Loop Control System Design Example

Design of an Open-Loop Automobile Cruise Controller

The following steps should be considered while designing an open-loop control system:

- Develop a model of the plant
- Develop a controller
- Analyze the controller
- Consider disturbance
- Determine performance

For an open-loop automobile cruise controller, the block diagram will be as shown below:

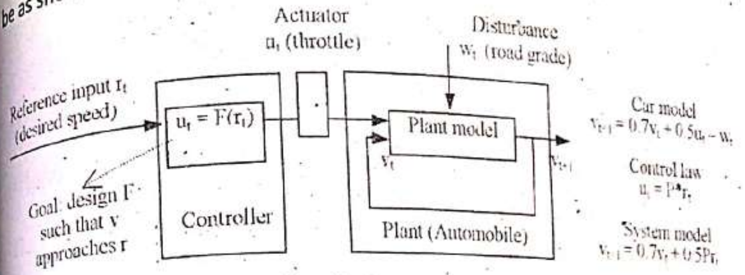

Actuator
u_t (throttle)
Disturbance
w_t (road grade)
Reference input r_t
(desired speed)
u_t = F(r_t)
Plant model
v_t
v_{t+1}
Goal: design F such that v approaches r
Controller
Plant (Automobile)
Car model
v_{t+1} = 0.7v_t + 0.5u_t − w_t
Control law
u_t = P*r_t
System model
v_{t+1} = 0.7v_t + 0.5Pr_t

Figure 7.4: Block diagram of open-loop automobile cruise controller

1. **Model of a plant**

It describes how the plant reacts to the input and current state. Here, the speed of the plant is changed based on reference desired input and throttle position. The modeling of a plant can be done by experiment and observations.

Let us assume a model which is valid for different combination of values of speed (v_t) and throttle position (u_t). And the model is represented as:

$v_{t+1} = 0.7 \times v_t + 0.5 \times u_t$

2. **Developing a controller**

Let us assume that the equation for controller will be:

$u_t = F(r_t) = P \times r_t$.

Where r_t is the desired speed, P is the proportional constant, u_t is input to actuator

Then,

$v_{t+1} = 0.7 \times v_t + 0.5 \times u_t = 0.7 \times v_t + 0.5P \times r_t$

At steady state, let us assume there is no variation at all, then we have $v_{t+1} = v_t = v_{ss}$. So, we get

$v_{ss} = 0.7 \times v_{ss} + 0.5P \times r_t$

$0.3\,v_{ss} = 0.5P \times r_t$.

Scanned with CamScanner

At steady state, we want steady state output and reference input to be equal i.e., $v_{ss} = r_t$.

$P = 0.3/0.5 = 0.6$

Hence, we have a simple proportional controller model i.e., $u_t = 0.6 \times r_t$

3. **Analyzing the controller**

Let $v_0 = 20$ mph, $r_0 = 50$ mph

$v_{t+1} = 0.7v_t + 0.5(0.6) \times r_t = 0.7v_t + 0.3 \times 50 = 0.7v_t + 15$ ...... (i)

Throttle position is 0.6×50 = 30 degree. So solving the equation(i) the speed of car for various time are shown in Table 7.1 (a).

4. **Considering disturbance**

Since, road grade can affect the speed, so considering the range of road grade from –5 mph to +5 mph. The equation will be as shown below:

$v_{t+1} = 0.7v_t + 0.5P \times r_0 - w_0$ ....(ii)

So substituting value of $w_0 = 5$ and –5 on equation (ii). The value will be:

$v_{t+1} = 0.7v_t + 10$

$v_{t+1} = 0.7v_t + 20$

Table 7.1: Open loop cruise controller response when disturbance is a) 0 b) +5 and c) -5

Time (t)	v_t	v_t for w = +5	v_t for w = -5
0	20.00	20.00	20.00
1	29.00	34.00	24.00
2	35.30	43.80	26.80
3	39.71	50.66	28.76
4	42.80	55.46	30.13
5	44.96	58.82	31.09
6	46.47	61.18	31.76
7	47.53	62.82	32.24
8	48.27	63.98	32.56
9	48.79	64.78	32.80
10	49.15	65.35	32.96
11	49.41	65.74	33.07
12	49.58	66.02	33.15

5. **Determining performance**

For determining the performance, we have

$v_{t+1} = 0.7v_t + 0.5P \times r_0 - w_0$

Let t = 0, then

$v_1 = 0.7v_0 + 0.5P \times r_0 - w_0$

Let t=1 then,

$v_2 = 0.7(0.7v_0 + 0.5P \times r_0 - w_0) + 0.5P \times r_0 - w_0$

$= (0.7)^2 v_0 + (0.7 + 1.0) \times 0.5P \times r_0 - (0.7 + 1.0)\, w_0$

Similarly,

$v_t = (0.7)^t v_0 + (0.7^{t-1} + 0.7^{t-2} + ... + 0.7 + 1.0)\,(0.5P \times r_0 - w_0)$

Here the rate of decay is given by the coefficient of v_0 i.e., $\alpha = 0.7$

In $v_{t+1} = 0.7v_t + 0.5P \times r_0 - w_0$, coefficient of v_t determines rate of decay of v_0. So,

- If $\alpha > 1$ or $\alpha < -1$, v_t will grow without bound as time increases.
- If $\alpha < 0$, v_t will oscillate.

Design of a Closed-Loop Automobile Cruise Controller

The speed of the automobile cruise controller in open loop control system may be degraded due to presence of various disturbances like grade of road resulting friction or direction of wind. So, to reduce the error in speed due to disturbance, the closed loop controller may be beneficial as it includes the speed sensor which detects the output speed and based on this value, the error detector detects the error so that the error in speed can be detected and corrected. The block diagram for closed loop automobile is as shown in Figure 7.5.

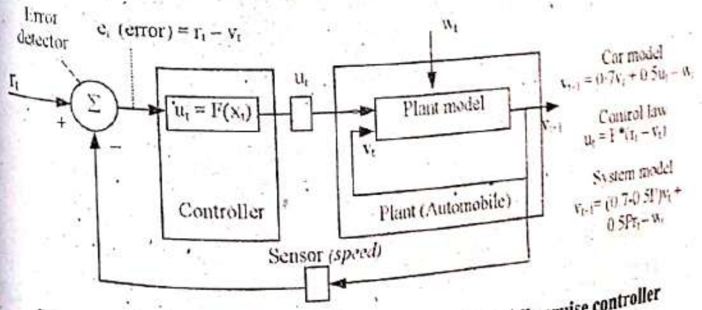


Figure 7.5: Block diagram of closed loop automobile cruise controller

The equation for controller favoring the above car model is:

$u_t = P \times u_t$ where $u_t = (r_t - v_t)$

$\therefore u_t = P \times (r_t - v_t)$ ....(1)

Then the equation for car model as mentioned is:

$v_{t+1} = 0.7v_t + 0.5P \times u_t - w$..................................(2)

Substituting the value of u_t,

$v_{t+1} = 0.7v_t + 0.5P \times (r_t - v_t) - w$

or, $v_{t+1} = (0.7 + 0.5P)v_t + 0.5P(r_t) - w$ ....(3)

Then, for **stability** of the control system,

$v_1 = (0.7 + 0.5P)v_0 + 0.5P \times r_0 - w$

$v_2 = (0.7 + 0.5P)v_1 + 0.5P \times r_0 - w$

$= (0.7 + 0.5P)\{(0.7 + 0.5P)v_0 + 0.5P \times r_0\} + 0.5P \times r_0 - w$

If we generation above equation then for v_t, we get

$v_t = (0.7 - 0.5P)^t v_0 + ((0.7 - 0.5P)^{t-1} + (0.7 - 0.5P)^{t-2} +$..............

$+ 0.7 - 0.5P + 1.0)(0.5P \times r_0 - w_0)$ .... (4)

Here, the value of α is $(0.7 - 0.5P)$. So, for the system to be stable, stability constraint (i.e., convergence) requires

$|0.7 - 0.5P| < 1$

$-1 < 0.7 - 0.5P < 1$

Computing inequalities, we obtain

$-0.6 < P < 3.4$

This means, to make the system stable the value of P should lie between -0.6 to 3.4.

To **reduce the effect of initial condition**, the coefficient of v_0 should be made as small as possible in equation(4). We have coefficient of v_0 is $(0.7 - 0.5P)^t$. Let us make it zero then, we have

$0.7 - 0.5P = 0$

or, $0.7 = 0.5P$

$\therefore P = 1.4$

Similarly, to avoid oscillation,

$0.7 - 0.5P \geq 0$.

or, $P \leq 1.4$

So, the fastest convergence without oscillation and effect of initial condition will occur when P = 1.4. However, the objective of control system is to have perfect tracking.

So, for **perfect tracking**, $v_{t+1} = v_t = v_{ss}$. Now equation(3) becomes

$v_{ss} = (0.7 - 0.5P)v_{ss} + 0.5P \times r_0 - w_0$

or, $(1 - 0.7 + 0.5P)v_{ss} = 0.5P \times r_0 - w_0$

or, $v_{ss} = (0.5P/(0.3 + 0.5P)) \times r_0 - (1.0/(0.3 + 0.5P)) \times w_o$

Since for perfect tracking v_{ss} should be close enough to r_0. So, the value of P should be chosen as large as possible. However, for stability the range of P should be less than 3.4 and for avoiding oscillation and reducing effect of initial condition, P should be 1.4. Hence, tradeoff between these constraints is to be made. So, setting the value of P = 3.3 (stable, track well, some oscillation),

$\mathbf{u_t = 3.3(r_t - v_t)}$

Analyzing the controller:

Let us consider, $v_0 = 20$ mph, $r_0 = 50$ mph, $w = 0$, then

$v_{t+1} = 0.7v_t + 0.5P(r_t - v_t) - w$

or, $v_{t+1} = 0.7v_t + 0.5 \times 3.3 \times (50 - v_t)$

Similarly, $u_t = P(r_t - v_t)$

$u_t = 3.3(50 - v_t)$

The range of u_t is 0 to 45 degree. Now, performing iterative calculation for value of v_t and u_t based on this assumption, Table 7.2(a) can be obtained where speed and throttle position from time 0 to 5 seconds are obtained. Here, the range of throttle is defined between 0 to 45 but from the table 7.2(a), the controller generates the throttle angle outside the range i.e. (0 – 45) which is not valid. The throttle saturates at angle 0 to 45 degrees.

Now, the speed and throttle position where we include this saturation in model in Table 7.2(b). For P = 3.3, $\alpha = 0.7 - 0.5P = -0.95$; the speed oscillates for many second until it reaches the steady state of 42.31. A negative value of α causes such oscillation which means the automobile is accelerating too hard when the current speed is less than the desired speed, thus overshoot the desired speed. Similarly, steady state speed is not 50 mph rather it is 42.31 mph with error of about 7.69 mph.

Scanned with CamScanner

Since, oscillation of the car speed could be uncomfortable to the passenger; the oscillation can be reduced or removed by decreasing the value of P. So let us take the value of P = 1.0 and obtain the data as in Table 7.2(c). Here, though the oscillation is eliminated and convergence time is reduced, but the steady state speed is only 31.25 mph which represent the error range of 18.75 mph.

Table 7.2: Closed loop automobile controller speeding up from 20 mph to 50 mph

Time	v_t	u_t	v_t	u_t	v_t	u_t
0.00	20.00	99.00	20.00	45.00	20.00	30.00
1.00	63.50	-44.55	36.50	44.55	29.00	21.00
2.00	22.18	91.82	47.83	7.18	30.80	19.20
3.00	61.43	-37.73	37.07	42.68	31.16	18.84
4.00	24.14	85.34	47.29	8.95	31.23	18.77
5.00	59.57	-31.58	37.58	40.99	31.25	18.75
6.00	25.91	79.50	46.80	10.55	31.25	18.75
7.00	57.89	-26.02	38.04	39.47	31.25	18.75
8.00	27.51	74.22	46.36	12.00	31.25	18.75
9.00	56.37	-21.01	38.45	38.10	31.25	18.75
10.00	28.95	69.46	45.97	13.31	31.25	18.75
11.00	55.00	-16.49	38.83	36.86	31.25	18.75
12.00	30.25	65.16	45.61	14.48	31.25	18.75
...						
47.00	44.31	18.78	41.76	27.20	31.25	18.75
48.00	40.41	31.66	42.83	23.66	31.25	18.75
49.00	44.11	19.42	41.81	27.02	31.25	18.75
50.00	40.59	31.05	42.78	23.83	31.25	18.75
...						
ss	42.31	25.38	42.31	25.38	31.25	18.75
	(a)		(b)		(c)	

The graph of speed for P = 3.3 and P = 1 is as shown below:

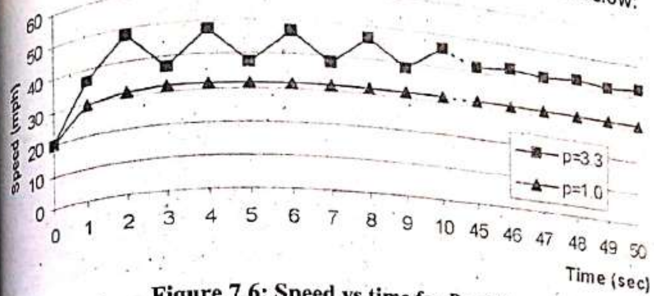


Figure 7.6: Speed vs time for P = 3.3 and P = 1

Considering the disturbances like road grade:

Assuming, road grade of +5% and –5% for P = 3.3 then we get the output at steady state v_{ss} = 39.74 mph and 44.87 mph respectively (value table not shown). In the response of open loop automobile cruise system in presence of disturbances, we observed output values at steady state v_{ss} = 33.15 mph and 66.05 mph respectively (refer to Table 7.1). If we compare output values from two different cases then we can say that the response of closed loop automobile cruise controller has less impact of external disturbances in comparison to the open loop automobile cruise controller.

7.3 General Control Systems and PID Controllers

7.3.1 Control Objectives

The main objective of control system design is to make output track the reference input even in the presence of measurement noise, model error and disturbances. The objective fulfillment can be analyzed and assessed through various metrics.

1. **Stability:** For the system to be stable, all variables in the system remain bounded
2. **Performance:** It describes how well the output is tracking the change in the reference input. The various aspects of performance is shown in the figure below:

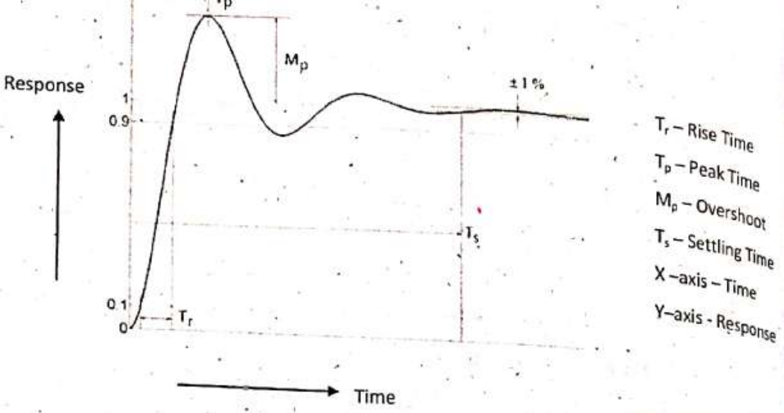


Figure 7.7: Aspect of performance metrics in control system response.

i. **Performance parameters:** The different aspects of performance are discussed below:

- **Rise time (T_r)** is the time required to change from 10% to 90% of its final value. It is a measure of the ability of a system to fast input signals.
- **Peak time (T_p)** is the time required to reach the first peak of the response.
- **Overshoot (M_p)** refers to an output exceeding its final, steady-state value. It is the percentage amount by which the peak of the response exceeds the final value.
- **Settling time (T_s)** is the time required for the system to settle down to within 1% of its final value.

ii. **Transient Response and Steady State Response of Control System**

Transient response occurs just after the system starts and when any undesired conditions occur. The system's response during the settling time is transient response. Whereas the Steady state occurs after the system becomes settled. Steady State Error is defined as the difference between the actual output and the desired output when system reaches steady state.

3. **Disturbance rejection**: Disturbances are the undesired effects which cannot be eliminated but its impact can be minimized.

4. **Robustness**: The system to be designed must be able to tolerate the modeling error of the plant. The stability and performance of the system should not be significantly affected by the presence of model errors.

7.3.2. Modeling Real Physical Systems

The accurate modeling of the behavior of the plant is an essential factor in control system design. Since the controller will be designed based on the plant model, the plant model must be accurate as far as possible. The key features of real systems are:

- **Continuous in nature**: It responds as continuous variables and as continuous function of time. Since real physical systems are continuously reacting, the plant model is represented by differential equations. Though continuous in nature, equivalent discrete time model can be determined. But the sampling period, however, must be selected much smaller than the reaction time of the system. Such sampling ensures system does not change much between sampling instants.
- **Complexity**: It is much more complex than any model we generally assume in our design. Our model may not include nonlinear effects, all system states, or all system interactions. Generally assumed model is a linear model which is sufficient when the variables of the model have a small operating range.

7.3.3. Controller Design

1. Proportional Control

A controller that multiplies the tracking error by a constant is referred as proportional control. The form of proportional control is:

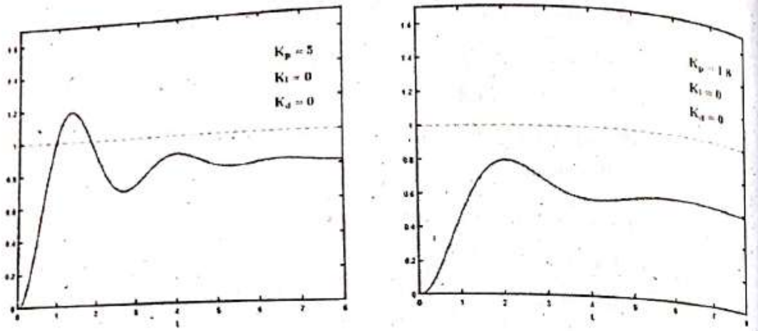
$$u(t) = P \times e(t)$$

Where, u(t) is the output of the controller, P is the proportional constant, e(t) is the measured error and is the difference between reference input and output of the system.

Proportional constant affects transient response, steady state tracking error and disturbance rejection. High value of proportional constant can cause system to become unstable by resulting in high overshoot and oscillation, whereas low value of P will cause the system to be less response or less sensitive, since rise time will be high for low value of P. Also the steady state error will be high for low

Scanned with CamScanner

value of P. The following figure shows the response of an arbitrary control system for different values of P.



(a) (b)

Figure 7.8: Response of system for (a) high value of proportional constant (b) low value of proportional constant

2. Proportional and Derivative (PD) Control

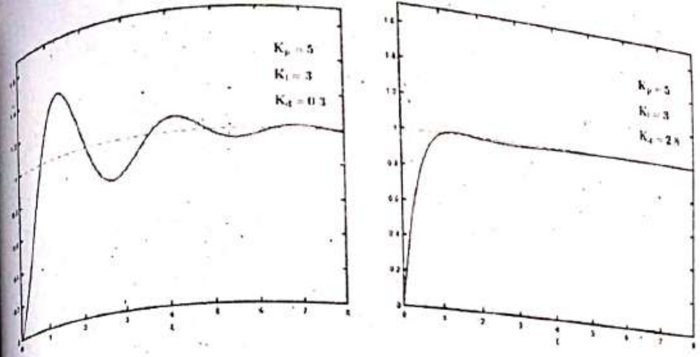
Derivative action predicts system behavior and improves settling time and stability of the system. Derivative term allows the transient response to be optimized without affecting the steady state response and disturbance rejection characteristic. Hence, transient response and the steady state error independently can be adjusted by using appropriate values of P and D in PD controller. The form of PD control is:

u(t) = P × e(t) + D × (e(t) – e(t – 1))

Characteristics of PD control:

- Rise time reduces, improves damping, overshoot reduces, response is stable

The following figure shows the response of an arbitrary system for derivative control action.



(a) (b)

Figure 7.9: Response of system for (a) low value of derivative constant (b) high value of derivative constant

3. Proportional and Integral (PI) Control

A PI controller is a special case of the PID controller in which the derivative of the error is not used. The integral term in PI control is the sum of the instantaneous error over time and the accumulated error is multiplied by integral constant. Its output is given by

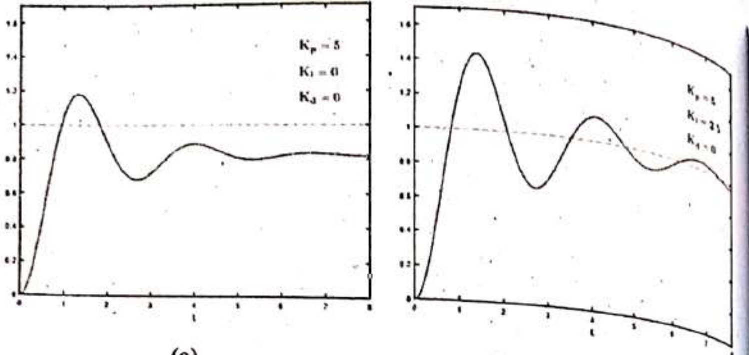
u(t) = P × e(t) + I × (e(0) + e(1) + e(2) + ... + e(t))

PI controller is used to eliminate the steady state error resulting from P controller. However, it has undesirable impact on speed and stability of the system.

Characteristics of PI control:

- Steady state accuracy improves, slight variation in rise time, response is oscillatory

The following figure shows effect of different values of integral constant in the response of an arbitrary system for PI control action.



(a)

(b)

Figure 7.10: Effect of integral term in system's response (a) No Integral gain, (b) Integral gain = 2.5

4. Proportional Integral and Derivative (PID) Control

PID controller is a feedback controller that helps to attain a set point irrespective of disturbances or any variation in characteristics of the plant of any form. It calculates its output based on the measured error and the three controller gains; proportional gain P, integral gain K, and derivative gain D.

- The proportional gain simply multiplies the error by a factor P. It reduces steady state errors while minimizes the effect of external disturbances.
- The integral term is a multiplication of the integral gain and the sum of the recent errors. The integral term helps in getting rid of the steady state error and causes the system to catch up with the desired set point.
- The derivative term determines the reaction to the rate of which the error has been changing and it increases damping and improves stability but has almost no effect on steady state error.

  Its output is given by

  u(t) = P × e(t) + I × (e(0) + e(1) + e(2) + ... + e(t)) + D × ((e(1) – e(0)) + (e(2) – e(1)) +... + (e(t) – e(t–1)))

The general block diagram of PID controller is shown in the figure below:

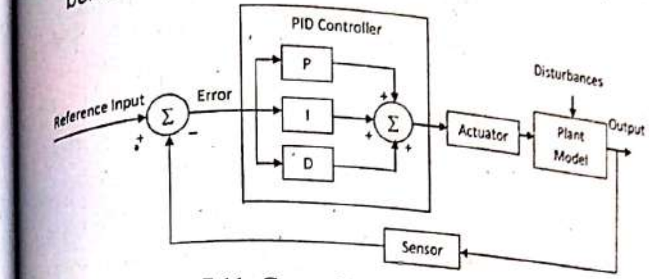


Figure 7.11: General block diagram of PID controller

The following figure shows effect of different values of P, I, D in the response of an arbitrary system for PID control action.

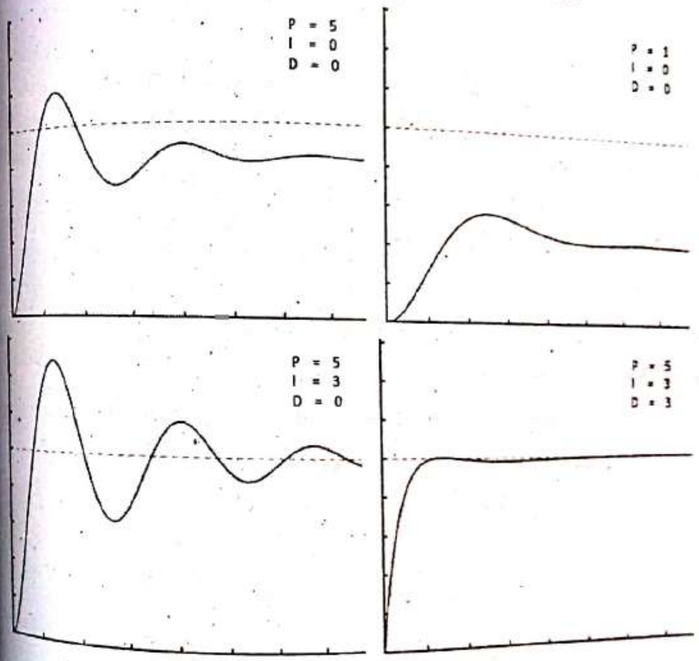


Figure 7.12: Effects of different values of P, I, D in the response of an arbitrary system

Scanned with CamScanner

5. **Summary of PID Control Action**

Type	Rise time	Maximum overshoot	Settling time	Steady-state error	Stability
P	Decrease	Increase	Small change	Decrease	Degrade
I	Decrease	Increase	Increase	Eliminate	Degrade
D	Small Change	Decrease	Decrease	No/small change	Improve

(***Note:** In above table, the effect is considered based on optimal value rather than increasing or decreasing the value of proportional, integral and derivative constant)

7.4 Software Coding of PID Controller

A PID controller can be implemented using software. At first, required initialization is done which is followed by reading reference value and sensor value. Then, after that error can be calculated which further is used to compute the output of PID controller. And, the refined output is fed to the actuator which in turn controls the plant based on the value of proportional, integral and derivative constant defined in the program. The pseudo code for the PID controller can be written as:

- Set values for Pgain, Igain, Dgain
- Initialize prior_error = 0 and sumoferrors = 0
- Repeat following steps
  - Read value from sensor, sensorValue = getValueFromSensor()
  - Read the reference value, refValue = getReferenceValue()
  - Calculate error = refValue – sensorValue
  - Calculate sumoferrors = sumoferrors + error
  - Calculate difference = prior_error – error
  - Output = Pgain × error + Igain × sumoferrors + Dgain × difference
  - Set the output of Actuator, setActuator(output)
  - prior_error = error

A program in C language to implement PID controller

```
typedef struct
{
        double Pgain, Dgain, Igain, errorPreviousValue, errorSum;
}PIDdata;
double PIDupdate(PIDdata *pidData, double sensorValue, double refValue)
{
        double Pterm, Iterm, Dterm;
        double error, difference;
        error = refValue – sensorValue;
        Pterm = pidData -> Pgain * error;
        pidData -> errorSum = pidData -> errorSum + error;
        Iterm = pidData -> Igain * pidData -> errorSum;
        difference = pidData -> errorPreviousValue – error;
        Dterm = pidData -> Dgain * difference;
        pidData -> errorPreviousValue = error;
        return (Pterm + Iterm + Dterm);
}
void main()
{
        double sensorValue, refValue, actuatorValue;
        PIDdata pidData;
        PIDinitialize(&pidData);
        while(1)
        {
             sensorValue = getValueFromSensor();
             refValue = getReferenceValue();
             actuatorValue = PIDupdate(&pidData, sensorValue, refValue);
             setActuator(actuatorValue);
        }
}
```

Scanned with CamScanner

7.5 PID Tuning

PID tuning is the adjustment of its control parameters to the optimum values for the desired control response. Quantitative analysis can be used to determine the values of P, I, and D. However, quantitative analysis is not necessary when safety and cost of using plant is not a concern. There are various methods for PID tuning, one of which is ad hoc tuning process. The steps for ad hoc tuning process are

- Start with small value of P gain, D and I gains as 0
- Increase value of D gain until oscillation is seen, and then D gain is decremented by a factor of 2 to 4.
- Then, increase value of P gain until oscillation or excessive overshoot is observed, and then P gain is reduced by a factor of 2 to 4.
- Next, increase the value of I gain and reduce it slightly when oscillation or excessive overshoot is seen.
- Above steps are repeated until satisfactory performance is achieved.

7.6 Practical Issues Related to Computer-Based Control

The various practical issues related to computer-based control are explained in the following paragraphs.

1. Quantization and Overflow Error

Quantization error occurs when machine number is altered to fit the constraints of the computer memory.

- **Case I:** when arithmetic results require more precisions than original values. For example, in operation 0.50×0.25 = 0.125, the final result requires more precision.
- **Case II:** when analog signals from sensors are quantized by analog to digital converter it can create quantization error. In quantization process limited set of discrete values are defined and if the signal or value from the sensors doesn't match the defined quantized discrete values then rounding or truncation will occur which results in quantization error. For example: When 4 levels are defined between −1.5 and 1.5 as −1.5, −0.75, 0, 0.75 and 1.5 then the value 1.3 will be taken as 1.5.

Overflow error occurs when the system attempts to operate on or results a number that does not lie within the defined range of the system. For example, let us consider a case of signed binary numbers where five bits are used to represent the magnitude while sixth or MSB is used to represent sign. Using such representation, when two binary numbers 010010 (+18) and 010101 (+21) are added then it results in 100111 which is (−25) rather than (+39). Since the first bit is used for sign representation, the undesirable output resulted due to overflow error. The situation can get more complex if we consider multiplication operation and floating point numbers.

2. Aliasing

Aliasing is the consequence of improper sampling process. It arises when a signal is discretely sampled at a rate that is insufficient to capture the changes in the signal. In simple term, aliasing causes the reconstructed signal to be different from original signal. It causes different signals to become indistinguishable. Let us consider an example in which the sampling is done at a period of 0.4 second which results a sampling frequency of 2.5 Hz. Then the following signals will be indistinguishable

y(t) = 1.0 × sin(6πt), frequency 3 Hz

y(t) = 1.0 × sin(πt), frequency 0.5 Hz

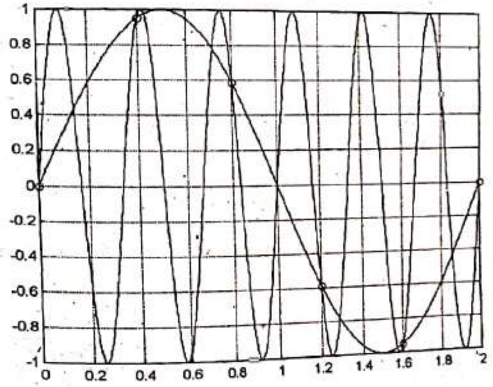


Figure 7.13: Aliasing illustration

For a sampling rate of 2.5 Hz, sine wave with frequency of 0.5 Hz is indistinguishable from sine waves at 3 Hz, 5.5 Hz, 8 Hz and so on. Also, it can only correctly sample signal below Nyquist frequency, which is half the value of sampling rate.

Scanned with CamScanner

Delay results in control signal being applied later than desired time. Computation delay is the attribute of many digital systems but too much delay results in performance degradation. The effect of delay can be accurately analyzed and we need to characterize implementation delay to ensure its effect is negligible in system's performance. Synchronous design makes hardware delay to be characterized easily. Software delay, however, is harder to predict. So, code should be organized carefully to make delay predictable. Also code can be written with predictable timing behavior, such that the effect of delay can be minimized to acceptable level.

7.7 Benefits of Computer-Based Control Implementation

The following are the benefits of computer-based control implementation.

1. **Repeatability**

   Analog systems are more prone to aging, temperature and manufacturing tolerance effects which cause results to vary with time. However, the digital systems can produce identical results for longer time.

2. **Stability**

   Since digital systems are less prone to different sorts of degradations and optimizations can be implemented efficiently, systems can become more stable.

3. **Programmability**

   Advanced features can be easily implemented in digital systems but that would be very complex in analog implementations. Few features include: control mode and gain switching, on-line performance evaluation, data storage, performance parameter estimation, and adaptive behavior.

4. **Flexibility**

   Computer based control can be easily re-configured based on requirement which allows periodic upgrade and enhancement of the system. It permits modification of the sequencing and control procedures for different products and for frequent change in product specifications.

Chapter 8

IC TECHNOLOGY

- Introduction
- Full-Custom (VLSI) IC Technology
- Semi-Custom (ASIC) IC Technology
- Programmable Logic Device (PLD) IC Technology

8.1 Introduction

A structural representation of the system generally deals with the various components and their interconnections to implement system's functionality. IC technology is more about mapping the structural representation to a physical implementation. The physical implementation can be done using various methods, out of which full-custom, semi-custom and programmable technologies are few common methods. As CMOS transistor is the core of every component, let us take a look at CMOS transistor and different layers required for its physical implementations.

CMOS Transistor

CMOS transistor consists of three terminals: the source, drain, and gate. Source and drain are created by implanting ions on the surface of silicon. Gate is formed using poly-silicon, and lies between source and drain. Gate is placed on top of silicon and is isolated from silicon with the help of silicon dioxide insulating layer. Gate voltage controls the current flowing from source to the drain. In case of nMOS transistor, a high voltage at gate will attract electrons from silicon substrate towards it resulting in formation of conducting channel between source and drain. For a low voltage at gate, the conducting channel is not formed.

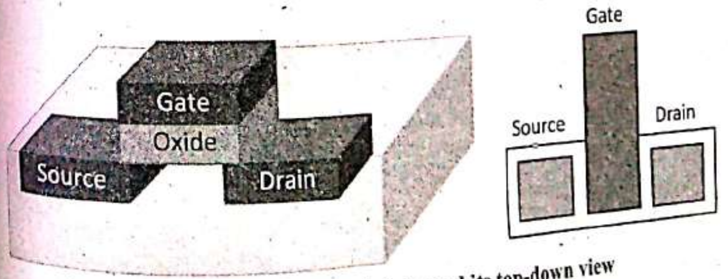


Figure 8.1: CMOS transistor and its top-down view

3. **Computation Delay**

Delay results in control signal being applied later than desired time. Computation delay is the attribute of many digital systems but too much delay results in performance degradation. The effect of delay can be accurately analyzed and we need to characterize implementation delay to ensure its effect is negligible in system's performance. Synchronous design makes hardware delay to be characterized easily. Software delay, however, is harder to predict. So, code should be organized carefully to make delay predictable. Also code can be written with predictable timing behavior, such that the effect of delay can be minimized to acceptable level.

7.7 Benefits of Computer-Based Control Implementation

The following are the benefits of computer-based control implementation.

1. **Repeatability**

Analog systems are more prone to aging, temperature and manufacturing tolerance effects which cause results to vary with time. However, the digital systems can produce identical results for longer time.

2. **Stability**

Since digital systems are less prone to different sorts of degradations and optimizations can be implemented efficiently, systems can become more stable.

3. **Programmability**

Advanced features can be easily implemented in digital systems but that would be very complex in analog implementations. Few features include: control mode and gain switching, on-line performance evaluation, data storage, performance parameter estimation, and adaptive behavior.

4. **Flexibility**

Computer based control can be easily re-configured based on requirement which allows periodic upgrade and enhancement of the system. It permits modification of the sequencing and control procedures for different products and for frequent change in product specifications.

Chapter 8

IC TECHNOLOGY

- Introduction
- Full-Custom (VLSI) IC Technology
- Semi-Custom (ASIC) IC Technology
- Programmable Logic Device (PLD) IC Technology

8.1 Introduction

A structural representation of the system generally deals with the various components and their interconnections to implement system's functionality. IC technology is more about mapping the structural representation to a physical implementation. The physical implementation can be done using various methods, out of which full-custom, semi-custom and programmable technologies are few common methods. As CMOS transistor is the core of every component, let us take a look at CMOS transistor and different layers required for its physical implementations.

CMOS Transistor

CMOS transistor consists of three terminals: the source, drain, and gate. Source and drain are created by implanting ions on the surface of silicon. Gate is formed using poly-silicon, and lies between source and drain. Gate is placed on top of silicon and is isolated from silicon with the help of silicon dioxide insulating layer. Gate voltage controls the current flowing from source to the drain. In case of nMOS transistor, a high voltage at gate will attract electrons from silicon substrate towards it resulting in formation of conducting channel between source and drain. For a low voltage at gate, the conducting channel is not formed.

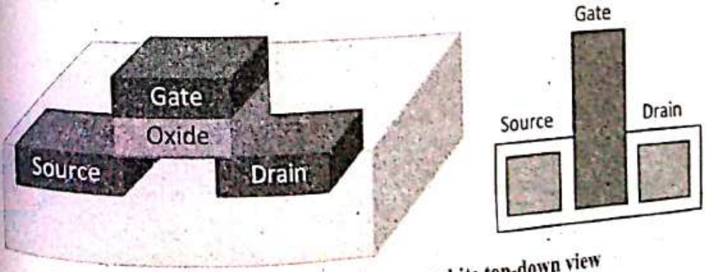
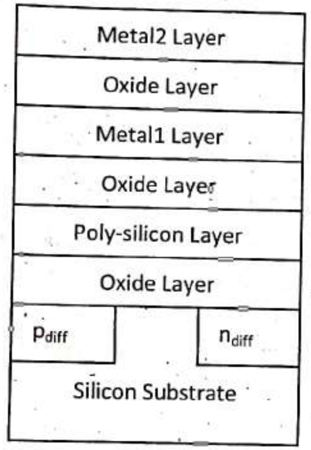


Figure 8.1: CMOS transistor and its top-down view

Layers in Physical Implementation

The transistor basically has three layers: diffusion layer for source and drain, oxide layer for insulation, and poly-silicon layer for gate. For circuits, there will be number of transistors connected together to represent particular functionality. These connections are represented by metal layers. There can be number of metal layers based on complexity of circuit implemented. Each metal layer is insulated from another layer using oxide layer. Hence, there exists number of oxide layer.



p_{diff} = diffusion of p-type material, n_{diff} = diffusion of n-type material

Figure 8.2: Basic layers in physical implementation

Example 1:

Draw the transistor level circuit schematic and top-down view for a NAND gate

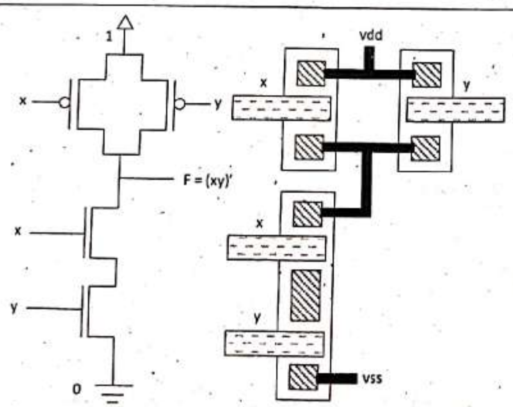


Figure 8.3: Circuit schematic and top-down view of NAND gate

8.1.1 IC Manufacturing Process

Basically, IC manufacturing process can be divided into two phases: design phase and manufacturing phase. In design phase, structural design and layout design is done, whereas manufacturing phase includes various steps from mask creation to final IC packaging.

1. Design Phase

In design phase, the structural description along with the layout of the system is developed. Initially, the behavioral description of the system is implemented using hardware description language. The high-level HDL describes the circuit at the Register Transfer Level. The first step in the synthesis process is compilation which converts high-level VHDL language into a netlist at the gate level. The second process is speed and area optimization which is performed on gate-level netlist. Finally, the physical layout of the system is generated with the help of place-and-route software. The layout specifies the placement of every transistor and every wire connecting those transistors. Several EDA (Electronic Design Automation) tools are available for circuit synthesis, implementation, and simulation.

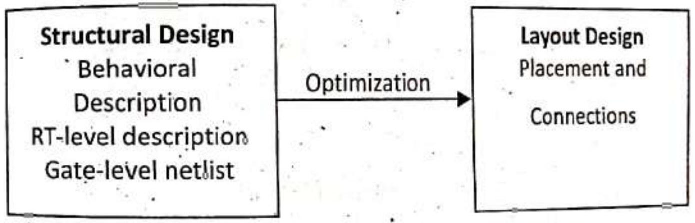


Figure 8.4: Design phase in IC manufacturing process

2. Manufacturing Phase

Manufacturing phase consists of several steps which are shown in the figure below and later each step is explained briefly.

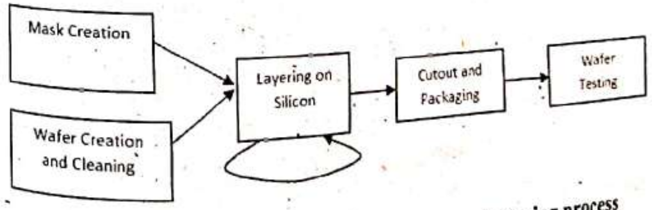


Figure 8.5: Manufacturing phase in IC manufacturing process

Scanned with CamScanner

i. **Mask creation**: The layout design of the system is translated into masks. The number of masks requirement may vary based on number of layers defined by the systems complexity. Masks for different layers – such as oxide layer, metal layers, etc – are generated. Generally, masks contain number of identical regions, so that number of IC's can be produced at once.

ii. **Silicon wafer creation and its cleaning**: In a crucible, high purity silicon is melted. Donor impurity atoms can be added to dope the crystal. A seed crystal is dipped into molten silicon and pulled upwards rotating it. And cylindrical ingot is extracted by controlling temperature gradients, rate of pulling and speed of rotation. Finally, the ingot is sliced with a wafer saw and polished to form wafers.

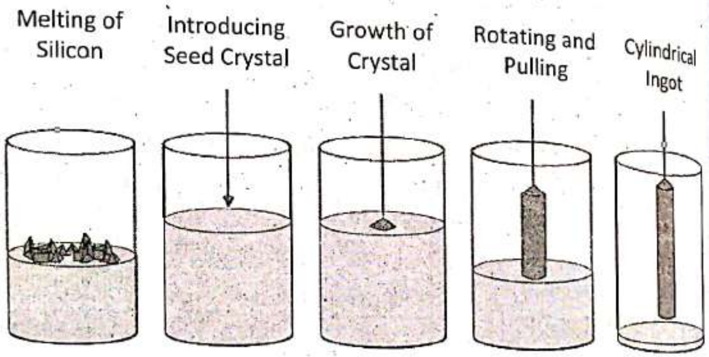


Figure 8.6: Silicon wafer creation

Wafer must be cleaned before any layer is deposited on it. Various cleaning methods can be used. Chemical cleaning methods are commonly used. First method of chemical cleaning is by using piranha solution in which wafer is immersed in hot mixture of hydrogen peroxide and sulfuric acid. Another method is using sonic waves in cleaning solution which is known as megasonic cleaning process. After the water is cleaned with chemical, it must be rinsed with De-Ionized (DI) water. Finally, the wafer is dried using either nitrogen gun or by baking. Also spun dry method can be used to make the wafer dry after cleansing process.

iii. **Layering on silicon:** Various layers are developed on the silicon surface. Layer for masks can be created using different layering techniques. Photolithography, which uses optical radiation to create patterns, is very common method in layering process. In this process, the layer required, for example silicon dioxide, is built onto the silicon surface which is overlapped by photoresist. Positive photoresist becomes soluble when UV rays are exposed on it. Using proper alignment, the UV rays are passed through the masks which cast a shadow on the photoresist wherever the layer of silicon dioxide is required. Then the soluble photoresist is washed using appropriate solvent. Finally, the exposed silicon dioxide is etched away using chemicals and the remaining photoresist is removed to expose the regions of silicon dioxide that we required in our layer. The whole process is repeated for each layer.

iv. **Wafer testing:** Number of ICs is produced in a single silicon wafer, which are subjected to test for errors or faulty ones. Testers or wafer probes are equipments used to test the correctness of the IC's by inspecting the output response for the streams of input.

v. **Chip cutout/packaging:** Individual IC from wafer is cut out using a diamond scribe. Verified ones are mounted in an IC package which encapsulates the IC. Packaging prevents physical damage and corrosion, also supports electrical contact. Through hole package and surface mount package are examples of IC packaging. Single In-line Packaging and Dual In-line Packaging are types of through hole packaging.

8.1.2 Photolithography

Photolithography is the process which transfers a pattern from a mask to a light-sensitive chemical photoresist on the substrate. The word photolithography is from the Greek origin: photo means light, litho means stone and graphy means writing. It uses optical radiation to create patterns, of complex circuit on a wafer. The various steps involved in photolithographic process are deposit barrier layer, photoresist coating, soft bake, mask alignment and exposure, develop photoresist, hard bake, etch window in barrier layer and remove photoresist.

Scanned with CamScanner

The various steps of photolithography are explained below:

i. Deposit Barrier Layer

Barrier layers are the materials which are required to be laid on the substrate. It may be silicon dioxide, silicon nitride, poly-silicon, metals, etc. Different methods can be used for barrier formation: thermal oxidation, chemical vapor deposition, sputtering and vacuum evaporation. Silicon dioxide as a barrier layer is used to isolate one layer from another. For instance, it is used in electrical isolation of multilevel metallization. Silicon Dioxide can be grown using dry oxidation which uses O_2 gas in a chamber or wet oxidation in which the wafer is submerged in water. When heat is applied to the oxidation process, it increases the rate of SiO_2 growth.

ii. Photoresist Coating

Photoresist is a substance which changes its characteristics when exposed to UV light. Before photoresist is coated, hexamethyldisilazane (HMDS) is used on the surface to improve adhesion. After that, liquid photoresist is coated over barrier layer using spin coating method. In this method, the wafer is held on vacuum chuck which is spun at about 3000-6000 rpm for about 15-30 seconds. Appropriate spinner rotational speed and viscosity of resist are essential factors to define photoresist's thickness which is about few micro-meters.

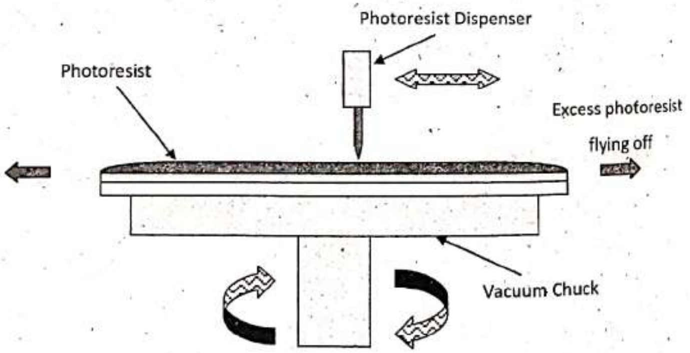


Figure 8.8: Photoresist coating – spin coating method

Types of photoresist:

a. Positive photoresist

b. Negative photoresist

Positive photoresist is insoluble in normal state but becomes soluble when exposed to UV light. Negative photoresist is soluble in normal state but becomes insoluble when exposed to UV light.

iii. Soft Bake or Pre Bake

Soft bake is simply the process of heating the wafer which removes the solvent from the photoresist. Baking time and temperature depend on the type of photoresist used and baking method. Different baking methods include hotplate, oven baking and microwave baking.

iv. Mask Alignment and Exposure

Mask is simply an opaque plate with holes to pass UV rays. It contains pattern to be formed on wafer. Mask is aligned with the wafer accurately with the help of special device: steppers use automatic pattern recognition and alignment systems. Alignment masks are available on the mask and on wafer so as to make alignment more precise.

Once the mask has been precisely aligned, the photoresist is exposed through the pattern on the mask with a controlled amount of UV light. Exposure will cause exposed positive photoresist to become soluble whereas if negative photoresist is used then exposed part of it becomes insoluble. There are three primary exposure methods: contact, proximity, and projection.

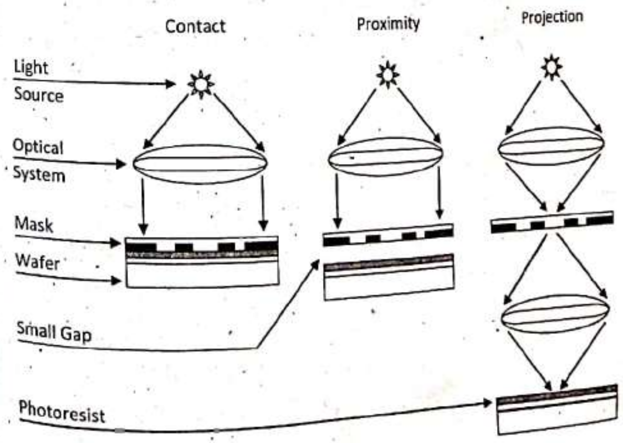


Figure 8.9: Different exposure methods

Scanned with CamScanner

a. **Contact Printing**: In contact printing, the resist-coated silicon wafer and mask are brought into physical contact when exposed to UV light. This method results in very high resolution but the debris, trapped between the resist and the mask, can damage the mask and cause defects in the pattern.

b. **Proximity Printing**: In this method, small gap is maintained between wafer and the mask during exposure. The gap minimizes the risk of mask damage at the expense of resolution.

c. **Projection Printing**: in this printing method, an image of the patterns on the mask is projected onto the resist-coated wafer. High gap eliminates the risk of mask damage and high resolution is possible. For high resolution, only a small portion of the mask is imaged and stepped over the surface of the wafer.

v. Develop Photoresist

Barrier layer is exposed when the soluble photoresist is chemically washed away using a developer solution. In immersion develop method the photoresist-coated wafer is immersed in a developer solution. Then, it is rinsed with DI water and dried using spin dry method.

vi. Hard Bake or Post Bake

Hard bake is used to stabilize and harden developed photoresist. It not only improves adhesion of the photoresist but also removes traces of solvent or developer solution. But, however, improper post bake can cause resist removal more difficult. Baking time and temperature can vary based on type of photoresist and baking method.

vii. Etch Window in Barrier Layer

As hardened photoresist does not shield all part of barrier layer, etching method is implemented to remove the barrier layer which was left uncovered. Two methods of etching can be implemented: wet etch, also known as chemical etching, and dry etch, also known as plasma etching. In Wet etching method, wafer is submerged in HF acid and unprotected barrier layer is removed. Dry etch method uses plasma which collides with the surface and removes the layers of target material.

viii. Remove Photoresist

Finally, the remaining photoresist is stripped from the surface exposing the required barrier layer. Photoresist can be removed by using solvent strippers, which cause the resist to swell and lose adhesion from the substrate. Another method of photoresist removal is by burning the resist in an oxygen plasma system and this process is called resist ashing.

The photolithography process can be summarized diagrammatically as:

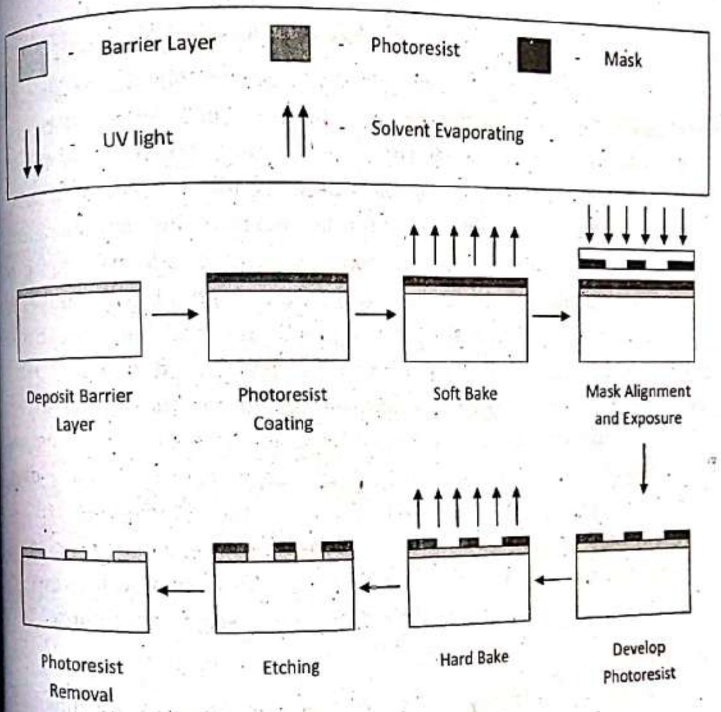


Figure 8.10: Various steps of photolithography process

8.2 Full-Custom (VLSI) IC Technology

Full-Custom IC technology includes VLSI (Very Large Scale Integrated Circuit) design in which the designer designs the complete transistor-level circuit for every processor, memory and other components used in the design. In this technology, first the designer creates layouts for basic components. And then, components are placed and connected, which are

Scanned with CamScanner

later translated to masks. Finally, the masks are given to the manufacturer for fabrication of IC of final design. The design steps are shown in the figure 8.11.

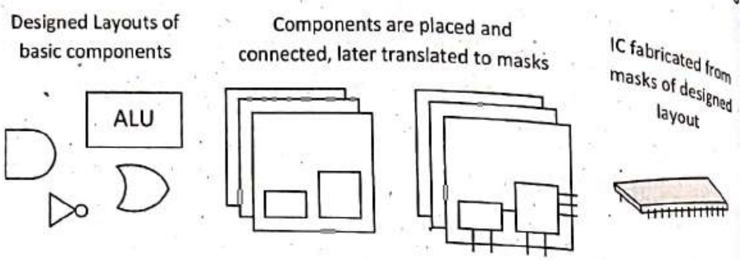

Designed Layouts of basic components
Components are placed and connected, later translated to masks
IC fabricated from masks of designed layout
ALU

Figure 8.11: Full-custom IC technology

Placement, routing and sizing are few important physical design tasks that should be done carefully for an efficient layout design. Placement represents the task of placing and orienting the transistors on the IC. Routing is the task of connecting wires between the transistors. In sizing, width of each wire along with size of transistor is taken into considerations. Placement and routing should be done so as to avoid overlapping of transistors and wires. Placement also defines the length of wire required to connect transistors. Large size of wires and transistors provide better performances, but it increases power consumption and demands more silicon area in the IC. Compact layout can lead to an efficient design. For instance, transistor placed at closer distance requires shorter connecting wires, which further decreases the silicon size in the IC. In early days, compact designs were implemented using hand layout technique which is generally used for small and critical components. Today, however, physical design tools are used for automatic layout of the design which runs for hours or days to generate the optimized layout for better performance.

Advantages:

Excellent efficiency: With respect to power consumption, performance and size, full-custom IC technology can be highly efficient. Since layout design is done by the designer, the components can be placed closer to each other which can be connected using short wires. Such layout yields optimum performance, size and power.

No wasted area and no unused transistors: In full-custom design, the required transistors for the circuit are placed on the IC. But there are no unused transistors which prevents wasted area.

Disadvantages:

High NRE cost and long time-to-market: Designing a complete layout, even with the help of CAD tools, can be time-consuming and prone to error. In addition to that, creating masks for every layer of IC adds more time in design process. Also, manufactured IC may contain errors leading to requirement of several re-spins. All these factors cause full-custom IC technology to have high NRE cost and long time-to-market.

8.3 Semi-Custom (ASIC) IC Technology (Applicat^{n} Specific Int. Ckt)

In semi-custom IC technology, designer does not require to create full-custom layout rather connects the pre-positioned building blocks. The use of chip with pre-existing gates will lessen the design work of layout and mask creation. So, the NRE cost is reduced while the time-to-market is relatively fast as compared to full custom IC technology. But, however, there will be a reduction in performance in terms of power, size and speed. Two types of semi-custom IC technologies are described in the following paragraphs.

8.3.1 Gate Array Semi-Custom IC Technology

In a gate array IC technology, a chip with arrays of pre-designed logic gates is utilized to implement the desired circuit. Here, the masks for transistor and gate levels are already designed, so the designer has the task of connecting pre-designed gates to achieve the desired implementation. In this technology, a set of masks of predefined gates are provided to the designer who then provides the connections among gates to implement require circuit. Masks of connections are generated and all masks are used to fabricate the IC.

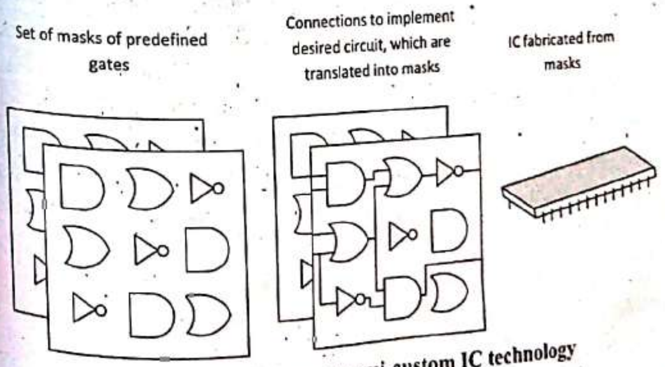

Set of masks of predefined gates
Connections to implement desired circuit, which are translated into masks
IC fabricated from masks

Figure 8.12: Gate array semi-custom IC technology

This technology results in fast and relatively inexpensive design cycles. But, gates are placed in advance which may result in many unused gates, since all instances of each type of gate may not be required in our desired circuit. Also, the fixed placement of gates can result in long routing wires between gates as the connection is not known while gates are already placed.

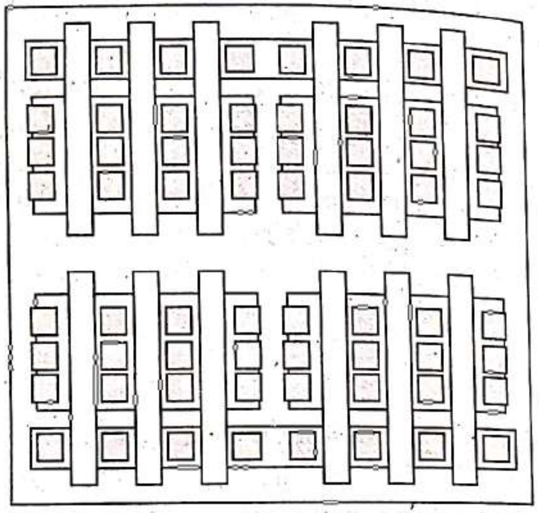


Figure 8.13: A simplified gate array layout

8.3.2 Standard Cell Semi-Custom IC Technology

In standard cell semi-custom IC technology, functional blocks, which are also called cells, with known electrical characteristics are utilized in the design to achieve very high gate density and good electrical performance. Cells may include logic gates such as NAND, NOR, etc. and other function blocks like multiplexor, flip-flop etc. In this technology, designers are facilitated with a library of predesigned cells from which the designer selects the required cells that are needed in the desired circuit. Masks of cells are created after the cells are placed and connected. Also the masks of connections among cells are generated. Using those masks, the IC is fabricated.

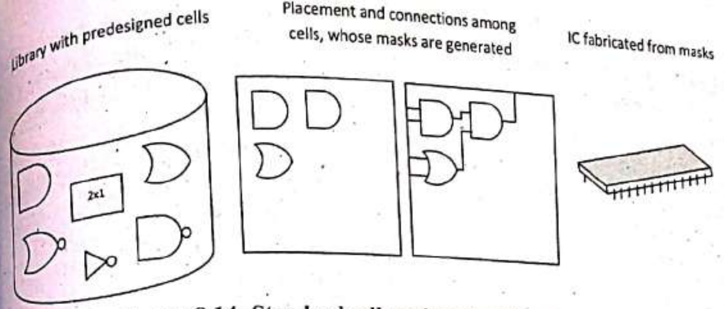

Library with predesigned cells
Placement and connections among cells, whose masks are generated
IC fabricated from masks
2x1

Figure 8.14: Standard cell semi-custom IC technology

The designer selects the cell, its position and its routing mechanism. So, it requires more NRE cost and longer time-to-market as compared to gate-array technology but still requires less than that of full-custom. However, the efficiency is better compared to gate array but less efficient than full-custom design. Hence, standard cell design lies between gate array and full custom design in terms of NRE cost, time-to-marker and performance.

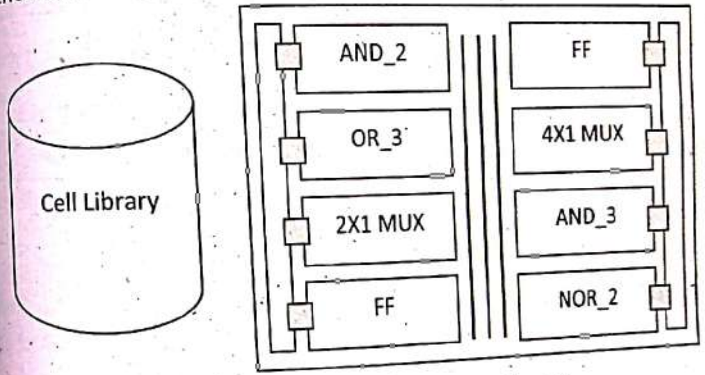


Figure 8.15: A simple standard cell layout

8.4 Programmable Logic Device (PLD) IC Technology

In programmable logic device IC technology, there exist programmable circuits which are programmed by the designer to implement the required design. Programming, in this case, means creating or breaking connections between wires that connect gates, either by blowing a fuse with high current, or setting a bit in a programmable switch. In this technology, a pre-fabricated chip with no logic function programmed is made available to the designer who then programs the required portions of the chip to implement the desired functionality.

It offers the designer the facility of changing design functions even after it has been programmed. PLD can be programmed, erased, and reprogrammed number of times, allowing easier prototyping and design modification.

There is a wide variety of PLD types, including Simple PLD, Complex PLD, GAL (Generic Array Logic), FPGA (Field-Programmable Gate Array) as well as many others left unmentioned. Programmable Logic Array (PLA) and Programmable Array Logic (PAL) are two examples of Simple Programmable Logic Devices (SPLD). Programmable Logic Array (PLA) consists of two planes of logic arrays: a programmable array of AND gates and a programmable array of OR gates. The AND plane and the OR plane give the possibility to computer any function expressed as a sum of products. Every AND gate in AND plane is associated with inputs and complement of inputs to generate any product term. And, OR gate generates the sum of AND gate outputs. The example of PLA is shown in the figure below.

Example 2: Implement the following truth table using PLA

A	B	C	F1	F2	F3	F4
0	0	0	0	0	1	1
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	0	1	0	1
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	1	0	0	1	0	1
1	1	1	1	1	0	0

F1 = ABC
F2 = A + B + C
F3 = A'B'C'
F4 = A' + B' + C'

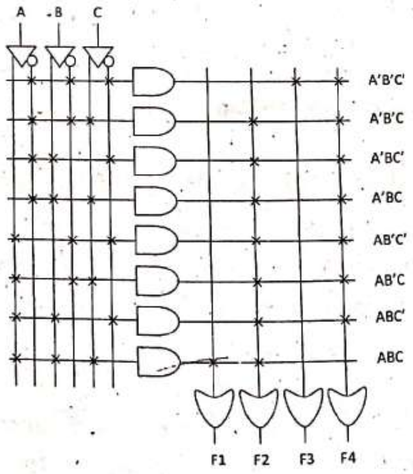


Figure 8.16: PLA implementation for given truth table

Programmable Array Logic uses just one programmable array: fixed OR matrix and programmable AND matrix. It decreases number of expensive programmable components which further reduces size and delay. PLA and PAL are generally used for low-complexity problems which require fairly high speed. As the complexity grows, Complex Programmable Logic Devices (CPLD) must be used. CPLDs are the integration of numerous SPLDs with added programmable interconnect between them. CPLD is a combination of fully programmable AND/OR array which perform a multitude of logic functions and microcells which perform combination or sequential logic. CPLDs may use analog sense amplifiers to boost the performance but at the cost of very high current requirements.

SOLUTION TO IMPORTANT QUESTIONS

Problem 1.

Draw the top-down view of the circuit on an IC for the given function F = XY + Z

Solution:

One way of solving this question is by representing the given function using NAND, NOR & NOT gates. So the given function can be written as:

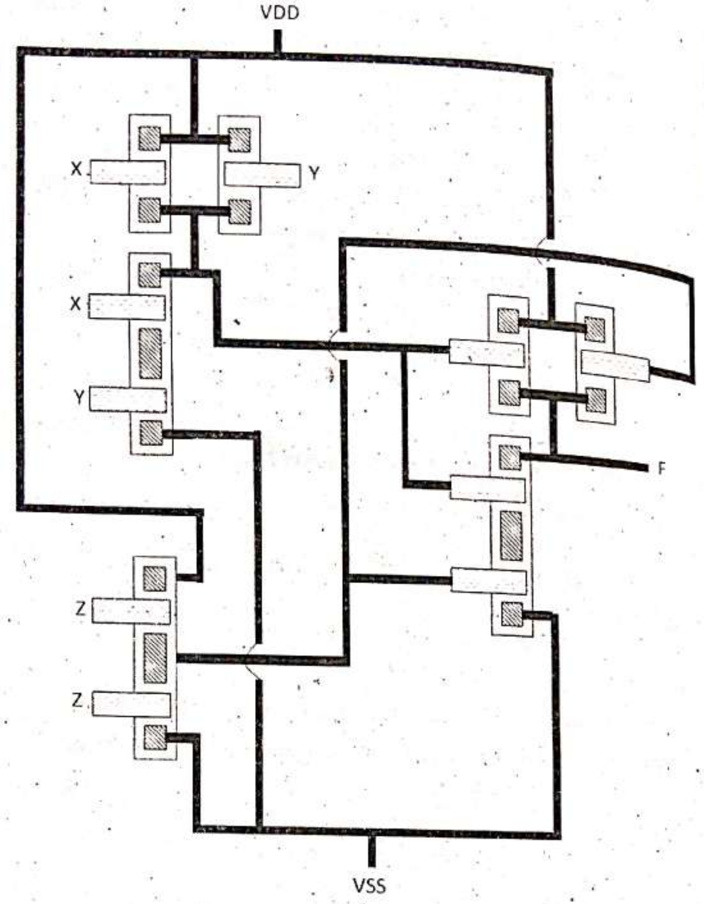
$$F = XY + Z$$

$$F = \overline{\overline{(X.Y + Z)}}$$

$$F = \overline{\overline{X.Y}\,.\,\overline{Z}}$$

F = (X NAND Y) NAND (NOT Z)

So we need to use top level view of two NAND gate and one NOT gate. The top level view of the function F = XY + Z is as shown below.



Chapter 9

MICROCONTROLLERS IN EMBEDDED SYSTEMS

- Intel 8051 Microcontroller Family, its Architecture and Instruction Sets
- Assembly Language Programming
- Interfacing with Seven Segment Display

9.1 Intel 8051 Microcontroller Family, its Architecture and Instruction Sets

1. Introduction

Microcontroller is a small computer on a single IC which contains processor core along with memory, I/O ports and other features. Microcontrollers are used in embedded applications in which systems are controlled automatically to carry out certain application. Almost every system using microcontroller performs control-oriented tasks. Several peripheral devices are inbuilt within the microcontroller to carry out the specified function. Timers, ADC and serial communication devices are few examples of peripheral devices.

2. Block Diagram

The general block diagram of the microcontroller is shown in the figure below:

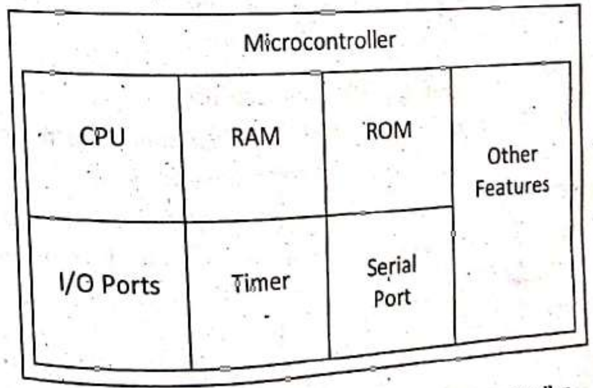


Figure 9.1: General block diagram of microcontroller

3. Comparison with Microprocessor

Many would easily presume that microcontroller and microprocessor to be similar. However, the following table will make a clear distinction between microcontroller and microprocessor.

SN	Microprocessor	Microcontroller
1.	General purpose processors	Special purpose processors
2.	It contains complete functional CPU only	In addition to functional CPU, it has timers, I/O ports, internal RAM and ROM, and other features
3.	Designer can select the size of memory, number of I/O ports, timers, etc. to be used	Size of memory, number of I/O ports, timers etc are fixed for a particular microcontroller
4.	Clock speed in very high in GHz range	Clock speed is low in MHz range
5.	Powerful addressing modes and many instructions are available to move data between memory and CPU	It focuses on bit handling instructions along with byte processing instructions.
6.	Access time for external memory and I/O devices is more	Access time for on-chip memory and I/O devices is less
7.	Microprocessor based systems are expensive and consumes more power	Microcontroller based systems are cheap and consumes less power

4. Criteria for Choosing a Microcontroller

- It must meet the computational needs of the task efficiently and cost effectively. Other considerations includes
  - o Speed, packaging (DIP(dual line package), QFP(quad flat package)), power consumption, amount of RAM and ROM, number of I/O pins and the timer on the chip
  - o Ease to amendments, cost per units
- It must provide flexibility to develop products around it. Some of the considerations include availability of an assembler, debugger, C compiler, emulator, technical support.

- It along with other reliable resources must be readily available in required quantities at any instant of time.

5. Comparison of 8051 Family Members

Each member of 8051, somehow, differs from each other. Though the instruction sets are almost common, the features provided can vary. The 8031 microcontroller is also referred as ROMless 8051 as all features are common except ROM space. The following table shows comparison of 8051, 8052 and 8031 microcontrollers.

Table 9.1: Comparison of three microcontrollers

Feature	8051	8052	8031
ROM	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O Pins	32	32	32
Serial Port	1	1	1
Interrupt Sources	6	8	6

6. 8051 Architecture

i. Internal Block Diagram of 8051

Timer is used to generate delay or count external events on T0 & T1's of μC

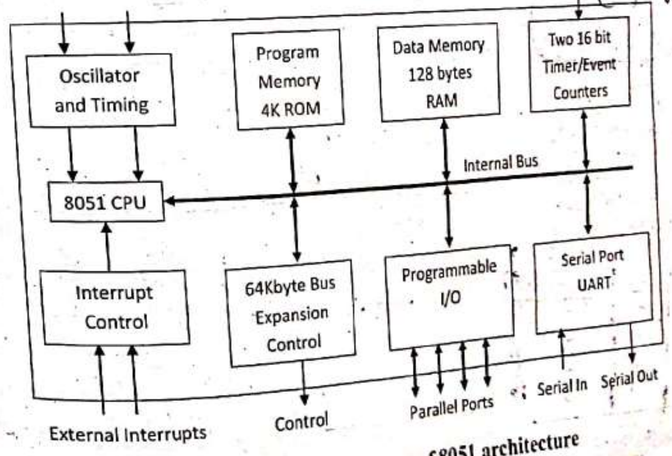


Figure 9.2: Internal block diagram of 8051 architecture

Scanned with CamScanner

ii. Features of 8051 Architecture

a. **Eight bit CPU with registers A and B:** Register A or Accumulator is used for mathematical and data transfer operations. Register B is used for multiplication and division purpose.

b. **Sixteen bit program counter (PC) and data pointer (DPTR):** PC points to the address of next instruction to be executed from ROM while DPTR is used to point to the memory addresses for internal and external code access and external data access. DPTR is made up of two 8 bit registers, DPH and DPL.

c. **Eight bit program status word (PSW):** Four flags in PSW are used to represent the outcomes of mathematical operations; Carry (CY), Auxiliary Carry (AC), Overflow (OV), and Parity (P) flags. It also consists two register select bits which select the particular register bank; RS1 and RS0 determines which register bank is being used out of four register banks.

PSW

CY	AC	F0	RS1	RS0	OV		P
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

Carry flag (CY) : set whenever there is carry out from D7 bit

Auxiliary carry flag (AC) : set whenever there is carry from D3 to D4 during ADD, SUB.

F0 flag : used for general purpose

RS1, RS0 : register bank selector

OV (overflow flag) : Set when result of arithmatical operatn is larger than 255 & can't be stored in single register

Parity flag (P) : • P=1 if A contains odd no. of 1s
• P=0 if A contains even no. of 1s

d. **Eight bit stack pointer (SP):** SP points to the stack which is the area to store and retrieve data quickly for some operations. It follows last in first out technique.

e. **Internal ROM:** It consists of 4 Kbytes or memory space as program memory. Look up tables can also be stored which can be accessed using appropriate instruction.

f. **Internal RAM:** It consists of 128 bytes of memory space as data memory.

- Four Register Banks, each with eight registers (R0–R7): Bank 0 occupies address from 00H to 07H and consecutive addresses are used by bank 1, bank 2 and bank 3. Total 32 registers are available from address 00H to 1FH. Bank 0 is selected as default. RS1 = 0 and RS0 = 1 in PSW register will select the register bank 1.

- Sixteen bytes of bit addressable memory: The address from 20H to 2FH of RAM is bit addressable. It is useful in bit manipulating operations. Each bit can be addressed using direct address from 00H to 7FH.

• Register Bank 0 is the deafault but..
• We can select any of the 4 register bank using the bit4 (RS1) and bit3 (RS0) of the 8-bit register bank:

RS1	RS0	Register Bank
0	0	0
0	1	1
1	0	2
1	1	3

Ex. code to move data to accumulator from R0 of register bank 2.

```
SETB PSW.4
CLR  PSW.3
MOV  A,R0
```

```
• MOV P0, #0FFH ; BYTE addressable instructn
• SETB P0.0     ; BIT addressable instructn
```

(Stack) • 8051 has a 8-bit Stack Pointer (SP) register : the 1st locatn used for stack
• The default value in SP register is 07H i.e. Ram location 08H is the 1st locatn used for stack
Register Bank 1 = Default Stack memory = locatn 08H to 1FH in RAM can be used as stack

- Eighty bytes of general purpose data memory: The memory space from address 30H to 7FH can be used for various operations when required.

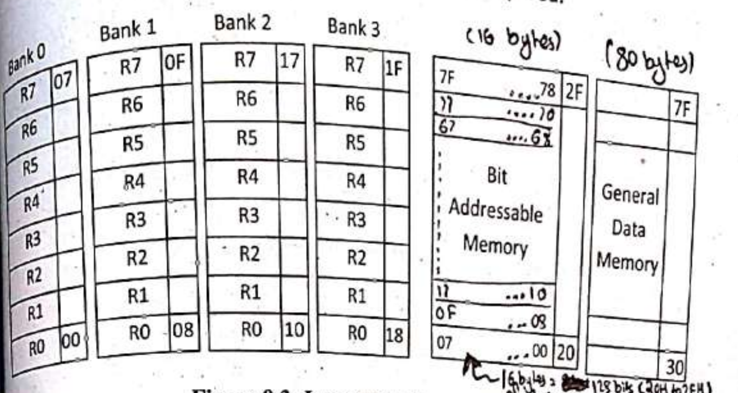


Figure 9.3: Internal RAM organization

g. **Thirty two I/O pins arranged as four 8-bit ports:** Four ports are bidirectional and can be used for input and output. Some of the pins are multifunctional which provide other functions along with input and output.

h. **Two 16-bit timer/counters (T0 and T1):** Each counter can be programmed to count internal clock pulses, acting as a timer, or programmed to count external pulses as a counter. This selection as well as mode of operation of counter can be set by using timer mode control (TMOD) register.

i. **Full duplex serial data receiver/transmitter:** Register serial control (SCON) controls serial data communication, and pins RXD and TXD are used to connect to other devices supporting serial communication. The serial buffer (SBUF) register is used to hold data in serial communication process.

j. **Two external interrupts and three internal interrupt sources:** Interrupt enable (IE) register selects which interrupt is to be selected and enabled. INT0 and INT1 pins are used by external circuitry to interrupt processor. Timer overflow (TF), receive interrupt (RI), and transmit interrupt (TI) are internal interrupts.

k. **The 8051 oscillator and clock:** It is the circuitry that generates the clock pulses by which all internal operations are synchronized. Time for particular instruction execution can be calculated based on number of machine cycles required by the instruction. For AT89S51, operating frequency is 11.0592 and its machine cycle consists of 12 clocks. So, time period for one machine cycle is 12 times the time period of single pulse which is equivalent to 1.085μs.

iii. **8051 Special Function Registers (SFRs)**

They are a group of specific internal registers that use internal RAM and their address lie between 80H and FFH. Some SFRs are bit addressable which allows programmer to access each bit of the register. The list of SFRs is given in the table below:

Table 9.2: List of special function registers

Name	Description	RAM Address	Access Level
A	Accumulator	0E0H	Bit Addressable
B	Register B, for multiplication and division	0F0H	Bit Addressable
PSW	Program Status Word	0D0H	Bit Addressable
SP	Stack Pointer	81H	
DPH	Data Pointer Higher Byte	83H	
DPL	Data Pointer Lower Byte	82H	
IE	Interrupt Enable	0A8H	Bit Addressable
IP	Interrupt Priority	0B8H	Bit Addressable
P0	Port 0	80H	Bit Addressable
P1	Port 1	90H	Bit Addressable
P2	Port 2	0A0H	Bit Addressable
P3	Port 3	0B0H	Bit Addressable
PCON	Power Control	87H	
SCON	Serial Port Control	98H	Bit Addressable
SBUF	Serial Port Data Buffer	99H	
TMOD	Timer/Counter Mode Control	89H	
TCON	Timer/Counter Control	88H	Bit Addressable
TL0	Timer 0 Low Byte	8AH	
TH0	Timer 0 High Byte	8CH	
TL1	Timer 1 Low Byte	8BH	
TH1	Timer 1 High Byte	8DH	

Collectively DPTR (16-bit)

Pin Descriptions

Pins 1 – 8 (PORT 1)

These eight pins represent PORT 1 which can be used as an input/output port. Since it is internally pulled up, it can be used without any external pull up registers configuration.

Pin – 9 (RESET)

RESET pin is used to set different registers to its initial values. The RESET pin must be set high for 2 machine cycles.

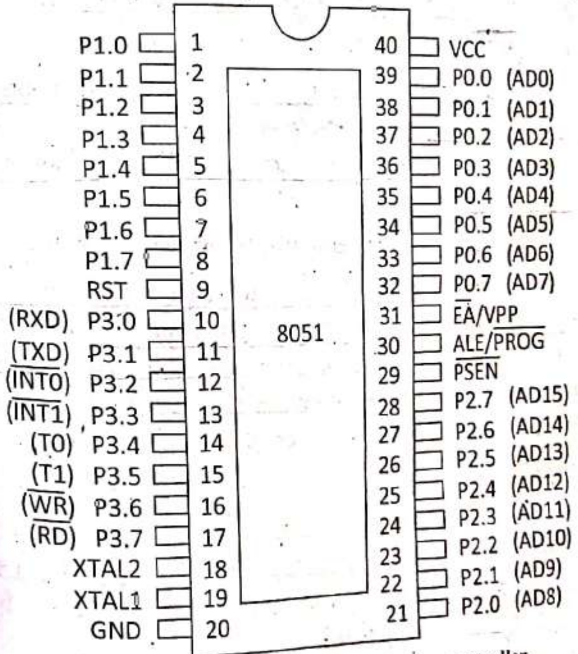


Figure 9.4: Pin descriptions of 8051 microcontroller

Scanned with CamScanner

RXD → Receive data
TXD → Transmit data from µc to other device

Pins 10 – 17 (PORT 3)

These pins together called Port 3 are bi directional and multifunctional in nature. Similar to port 1, it can be used as input or output without any external pull up registers configuration. Besides I/O, it supports serial communication (RXD and TXD), external interrupts ($\overline{INT0}$ and $\overline{INT1}$), timers (T0 and T1), and control signals ($\overline{WR}$ and $\overline{RD}$) for external memory.

Pins 18 – 19 (XTAL)

These pins are used to connect an external crystal to provide system clock.

Pin – 20 (GND, 0V)

Pins 21 – 28 (Port 2) if there is no intention to use ext. memory then use it can be used as general i/o pin

These pins are bidirectional and multifunctional in nature. PORT 2 may be used as an input or output port similar to port 1. The alternate use of port 2 is to provide a high-order address in conjunction with the port 0 to address external memory.

Pin 29 ($\overline{PSEN}$)

Program store enable (PSEN) is connected to output enable (OE) pin of external memory being interfaced. It is an active low output signal. When this pin is reset, microcontroller can read content of external memory location.

Pin 30 (ALE) used to demultiplex address & data

Address latch enable (ALE) is used to select address or data signal that are required while interfacing external memory. It is active high output signal and when it goes high, the lower address provided by port 0 is latched into the external address latch. This pin is also the program pulse input during flash programming

Pin 31 (EA)

External access enables or disables access of program from external memory. It must be connected to GND to fetch code from external program memory locations. It should be strapped to VCC for program executions of internal memory.

memory is not used, P0's pin can be used as general i/o pins. Otherwise, P0 is configured as addr. output (A0-A7) when ALE pin is high, or as data output (Data Bus) when ALE pin is low.

Pins 32-39 (Port 0)

PORT 0 is a collection of open drain bidirectional I/O pins. It can be configured as low-order address or data bus while accessing external memory.

Pin 40 (V_{CC}, +5V)

Minimum Hardware Configuration

Power supply: Pin 40 is connected to +5VDC, Pin 20 is grounded. Pin 31 is connected to VCC, representing the code is accessed from internal memory.

Reset circuit: Charging of capacitor makes RST high, which ensures two machine cycles on RST pin. After completion of charge, capacitor blocks DC causing RST low.

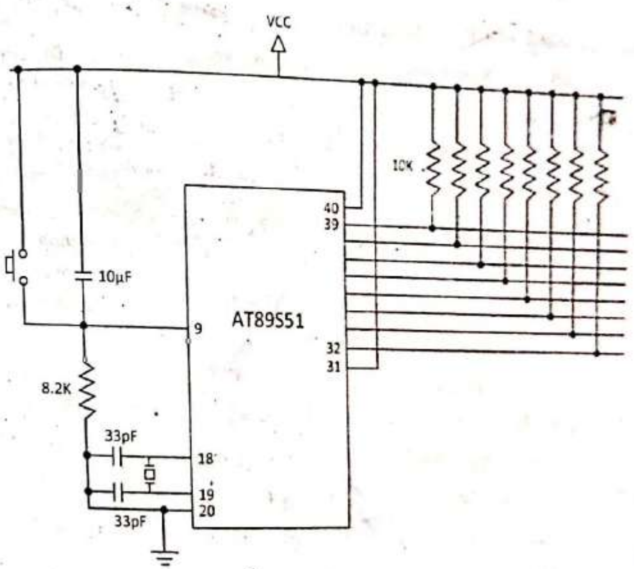


Figure 9.5: Minimum configuration for microcontroller to operate

Oscillator circuit: Ceramic capacitors of value between 20µF – 40µF are used as stabilizing capacitors. They act as loading capacitor and adjust the crystal frequency by shifting the frequency to a lower value.

Pull up circuit: Pins of PORT 0 are open drain so require pull up circuit. Each pin must be connected externally to a 10K ohm pull-up resistor.

↳ pull up resistors maintain default input state as 1 instead of high impedance state
↳ pull down resistors maintain default input state as 0

All other ports come built in with pull up resistors except port 0' because
↳ This allows port 0 to be used as high impedance input when needed

Scanned with CamScanner

9. 8051 Instruction Sets

Different symbols are used in the instruction whose meaning is clarified below:

#data – represents 8 bit data

Rn – represents one of eight registers (R0, R1, R2, R3, R4, R5, R6, R7)

@Ri – represents address pointed by value of Ri. Ri can be either R0 or R1.

direct – represents direct byte addressable memory

bit – direct bit addressable memory.

C – Carry, A – Accumulator, B – Register B

addr11 – 11 bit address, addr16 – 16 bit address and rel – 8 bit relative address

i. Data Transfer Instructions

square brockets means look at the contents of address given by it

no square braket means look at the value contained in it

Eg: MOV A, [redacted]R5

Eg: MOV A, 47H

Eg: MOV A, @R1

Eg: MOV A, #30H

SN	Mnemonics	Operation	Description
1	MOV A, Rn	A ← Rn	Content of Rn is moved to A
2	MOV A, direct	A ← [direct]	data in address 'direct' is moved to A
3	MOV A, @Ri	A ← [Ri] data in address pointed by value of Ri is moved to A	MOV instruction is used to transfer data involving registers, memory and immediate data
4	MOV A, #data	A ← data Hex 'data' is stored in A	
5	MOV Rn, A	Rn ← A	
6	MOV Rn, direct	Rn ← [direct]	
7	MOV Rn, #data	Rn ← data	
8	MOV direct, A	[direct] ← A	
9	MOV direct, Rn	[direct] ← Rn	
10	MOV direct, direct	[direct] ← [direct]	
11	MOV direct, @Ri	[direct] ← [Ri]	
12	MOV direct, #data	[direct] ← data	
13	MOV @Ri, A	[Ri] ← A	
14	MOV @Ri, direct	[Ri] ← [direct]	
15	MOV @Ri, #data	[Ri] ← data	
16	MOV DPTR, #data16	DPTR ← data16	
17	MOVC A, @A + DPTR	A ← [A + DPTR]	
18	MOVC A, @A + PC	A ← [A + PC]	MOVC is used to read data from code memory (ROM)
19	MOVX A, @Ri	A ← [Ri]	
20	MOVX A, @DPTR	A ← [DPTR]	

DPTR is the only user accessible 16-bit (2 byte) register

PC is also a 2-byte register with address of next instr.

SN	Mnemonics	Operation	Description
21	MOVX @Ri, A	[Ri] ← A	
22	MOVX @DPTR, A	[DPTR] ← A	MOVX is used to move data to and from external RAM. R0, R1 and DPTR are used to hold address of RAM
23	PUSH direct	Stack ← [direct]	
24	POP direct ← the popped value is stored at speified addr	[direct] ← Stack	
25	XCH A, Rn	A ← Rn, Rn ← A	
26	XCH A, direct	A ← [direct], [direct] ← A	
27	XCH A, @Ri		
28	XCHD A, @Ri	A ← [Ri], [Ri] ← A	

Arithmetic Instructions

SN	Mnemonics	Operation	Description
1	ADD A, Rn	A ← A + Rn	
2	ADD A, direct	A ← A + [direct]	
3	ADD A, @Ri	A ← A + [Ri]	
4	ADD A, #data	A ← A + data	
5	ADDC A, Rn	A ← A + Rn + C	Accumulator is one of the sources as well as destination for every ADD and SUB instructions
6	ADDC A, direct	A ← A + [direct] + C	
7	ADDC A, @Ri	A ← A + [Ri] + C	
8	ADDC A, #data	A ← A + data + C	
9	SUBB A, Rn	A ← A - Rn - C	
10	SUBB A, direct	A ← A - [direct] - C	
11	SUBB A, @Ri	A ← A - [Ri] - C	
12	SUBB A, #data	A ← A - data - C	
13	INC A	A ← A + 1	
14	INC Rn	Rn ← Rn + 1	
15	INC direct	[direct] ← [direct] + 1	
16	INC @Ri	[Ri] ← [Ri] + 1	
17	DEC A	A ← A – 1	
18	DEC Rn	Rn ← Rn – 1	
19	DEC direct	[direct] ← [direct] – 1	
20	DEC @Ri	[Ri] ← [Ri] – 1	
21	INC DPTR	DPTR ← DPTR + 1	

22	MUL AB	A ← Lower Byte
		B ← Higher byte	

Scanned with CamScanner

SN	Mnemonics	Operation	Description

23	DIV AB	A ← Quotient
		B ← Remainder	

24	DA A	Decimal Adjust accumulator	Performs correctn by translating the additn ~~of BCD bits~~ to ~~a~~ valid BCD number

iii. Logical Instructions

SN	Mnemonics	Operation	Description
1	ANL A, Rn	A ← A AND Rn	Logical AND, OR and XOR allows direct operation on memory address as well.
2	ANL A, direct	A ← A AND [direct]	
3	ANL A, @Ri	A ← A AND [Ri]	
4	ANL A, #data	A ← A AND data	
5	ANL direct, A	[direct] ← [direct] AND A	
6	ANL direct, #data	[direct] ← [direct] AND data	
7	ORL A, Rn	A ← A OR Rn	
8	ORL A, direct	A ← A OR [direct]	
9	ORL A, @Ri	A ← A OR [Ri]	
10	ORL A, #data	A ← A OR data	
11	ORL direct, A	[direct] ← [direct] OR A	
12	ORL direct, #data	[direct] ← [direct] OR data	
13	XRL A, Rn	A ← A XOR Rn	
14	XRL A, direct	A ← A XOR [direct]	
15	XRL A, @Ri	A ← A XOR [Ri]	
16	XRL A, #data	A ← A XOR data	
17	XRL direct, A	[direct] ← [direct] XOR A	
18	XRL direct, #data	[direct] ← [direct] XOR data	
19	CLR A	A ← 0	
20	CPL A	A ← A'	
21	RL A	Rotate A left	
22	RLC A	Rotate A left through C	
23	RR A	Rotate A right	
24	RRC A	Rotate A right through C	
25	SWAP A	Swap nibbles of A	

4 bit

Bit Manipulation and Program Branching Instructions

SN	Mnemonics	Operation	Description
1	CLR C	C ← 0	
2	CLR bit	bit ← 0	
3	SETB C	C ← 1	
4	SETB bit	bit ← 1	
5	CPL C	C ← C'	
6	CPL bit	bit ← bit'	
7	ANL C, bit	C ← C AND bit	
8	ANL C, /bit	C ← C AND bit'	
9	ORL C, bit	C ← C OR bit	
10	ORL C, /bit	C ← C OR bit'	
11	MOV C, bit	C ← bit	
12	MOV bit, C	bit ← C	Short jumps must be within 128 to +127 bytes of the contents of PC
13	JC rel	Jump if C ← 1	
14	JNC rel	Jump if C ← 0	
15	JB bit, rel	Jump if bit ← 1	
16	JNB bit, rel	Jump if bit ← 0	Long Jumps and calls can be used for any location within 64 Kbyte address space
17	JBC bit, rel	Jump if bit ← 1, and b ← 0	
18	ACALL addr11	Absolute jump to routine	
19	LCALL addr16	Long jump to routine	
20	RET	Return from subroutine	Absolute jumps and calls can be used for address within 2Kbyte range
21	RETI	Return from interrupt	
22	AJUMP addr11	Absolute jump	
23	LJUMP addr16	Long jump	
24	SJMP rel	Short jump	
25	JMP @ A + DPTR	Jump relative to DPTR	
26	JZ rel	Jump if A is zero	
27	JNZ rel	Jump if A is not zero	
28	CJNE A, direct, rel	Compare and jump if not equal	
29	CJNE A, #data, rel		
30	CJNE Rn, #data, rel		
31	CJNE @Ri, #data, rel		
32	DJNZ Rn, rel	Decrease and jump if not zero	
33	DJNZ direct, rel		

To call subroutine within 2048 addr. space (2 byte instruction)

To call subroutine within 2^{16} = 65536 address space (3 byte instr: 1st byte opcode, 2nd & 3rd byte address)

Scanned with CamScanner

10. Addressing Modes in 8051

i. Immediate Addressing Mode

The source operand is a constant value which must be preceded by # sign. It is used to load direct values into registers. For example, MOV A, #25H will assign 25 to register A.

ii. Register Direct Addressing Mode

The operand is a register which holds the data to be manipulated. For example, ADD A, R5 will add content of A and R5, and store back in A.

iii. Register Indirect Addressing Mode

Register is used to point the effective address of the operand. Registers R0, R1 and DPTR are used as pointer registers which must be preceded by @ sign. For example, MOV A, @R0 represents copying the contents of the address in R0 to the accumulator.

iv. Direct Addressing Mode

The operand represents the actual address of RAM in the instruction. For instance, MOV A, 80H moves the data of 80H into accumulator.

v. Relative Addressing

A relative address or offset is added to the PC to form the actual address. Generally used in jump instructions.

vi. Absolute Addressing Mode

In instructions, 11-bit or 16-bit absolute address is specified as the operand. ACALL and AJMP instructions use 11-bit address while LCALL and LJMP use 16-bit address.

vii. Indexed Addressing Mode

Index value or displacement is added to the base address to generate the effective address of the operand. For instance, MOVC A, @A + DPTR uses indexed addressing mode. The content pointed by (A + DPTR) address of ROM is copied to accumulator.

2 Assembly Language Programming

An assembly language program consists of series of statements which clude assembly language instructions and directives. Assembly language struction represents the operation to be carried out by the processor.

.2.1 Assembly Language Programming Format

Every instruction is composed of mnemonic followed by one, two or o operand. Mnemonic represents the actual operation to be done which perands are data items being manipulated. Directives are used to give irections to the assembler. Generally used directives are DB, ORG, END, nd EQU. The DB directive is used to define 8-bit data. The ORG represents he beginning of the program address while END represents the end of rogram. The EQU directive is used to define constant within a program. The umbers used must be followed by H to represent hex value otherwise the alue will be taken as decimal.

The assembly language program, in general, is written using following ormat:

[label:]	Mnemonic	[operands]	[; comments]
HERE:	MOV	A, #6AH	; mov operation

xample 1:

Read the content of port1 and port2, OR those contents and store the result in external RAM location 0310.

Problem Analysis: Port1 (with address 90H) and Port2 (with address A0H) provide eight bit data, so after OR operation the final result will also be of eight bit. Hence, single byte memory location is enough to store the result. However, to store the result in external RAM, the address of external RAM must be loaded into DPTR register and MOVX instruction should be used for data transfer.

Source Code:

```
ORG 00H
MOV A, 90H              ; copy the data of port1 to A
ORL A, 0A0H             ; OR the contents of A with port2, and stored in A
MOV DPTR, #0310H        ; DPTR used to point the external RAM address
MOVX @DPTR, A           ; move the content of A to RAM location
                        ; pointed by DPTR

END
```

Scanned with CamScanner

Example 2:

Read the content of internal RAM locations 27H and 28H, add them and store the result in RAM locations 30H and 31H.

Problem Analysis: The largest possible value at memory locations can be FFH, so the maximum value of final result will be FFH + FFH = 01FEH. Hence, two bytes of memory is required to store the result. To solve this, ADD instruction must be used to add two data while ADDC or JNC/JC can be used to add the carry.

Source Code:

```
ORG 00H
MOV 30H, #00H     ; [30] ← 00, assigns zero to memory location 30H
MOV A, 27H        ; A ← [27], assigns content of location 27H to
                  ;accumulator
ADD A, 28H        ; A ← A + [28], adds content of A and memory
                  ;location 28H
MOV 31H, A        ; [31] ← A, moves the lower byte or result to 31H
                  ;address of RAM .
MOV A, #00H       ; A ← 00, reset accumulator
ADDC A, 30H       ; A ← A + [30] + C, to extract the value of carry
MOV 30H, A        ; [30] ← A, move upper byte to location 30.
END
```

Example 3:

Add the content of internal RAM location 29H and port1, and store the result in RAM locations 30H and 31H in BCD form.

Problem Analysis: In BCD, the largest possible value can be 99, so the maximum value of final result will be 99 + 99 = 198. Hence, two bytes of memory is required to store the result. To solve this, ADD instruction must be used to add two BCD data and DA instruction after addition. However, DA instruction is not required for upper byte of result as it is less than 10. But, had there been more numbers causing upper byte to exceed more than 9, DA would have been required for upper byte as well.

Source Code:

```
ORG 00H
MOV 30H, #00H     ; [30] ← 00, assigns zero to memory location 30H
MOV A, 90H        ; A ← [90], assigns content of Port1 to
                  ;accumulator
ADD A, 29H        ; A ← A + [29], adds content of accumulator and
                  ;location 29H
DA A              ; Adjust the content of accumulator to BCD form
MOV 31H, A        ; [31] ← A, moves the lower byte or result to 31H
                  ;address of RAM
MOV A, #00H       ; A ← 00, reset accumulator
ADDC A, 30H       ; A ← A + [30] + C, to extract the value of carry
MOV 30H, A        ; [30] ← A, move upper byte to location 30.
END
```

Example 4:

Add 10 bytes of data of RAM location starting from address 20H. Store lower byte at 30H and upper byte at 31H.

Problem Analysis: Use of direct RAM address can make program complex, so R0 or R1 can be used to point the one byte address of RAM location. The register (R0 or R1) then can be incremented to access data of consecutive locations. The final sum after addition of 10 bytes of data can result in two bytes of data. So, carry must be checked after addition and value in register must be incremented accordingly when carry is resulted after addition.

Source Code:

```
ORG 00H
MOV R0, #20H      ; R0 ← 20H, assigning starting address of RAM
                  ;to R0
MOV R5, #0AH      ; R5 ← 0AH, counter for 10 bytes of data
MOV R6, #00H      ; R6 ← 00H, used for lower byte result
MOV R7, #00H      ; R7 ← 00H, used for upper byte result
```

```
HOME:
MOV A, @R0          ; A ← [R0], move content of RAM location
                    ;pointed by R0 to A
ADD A, R6           ; A ← A + R6, adds data of RAM location in each
                    ;iteration
JNC NEXT            ; checking if carry is generated after addition
INR R7              ; R7 ← R7 + 1, carry after each addition is added
                    ;to form upper byte
NEXT:
MOV R6, A           ; R6 ← A, move partial sum to R6
INR R0              ; R0 ← R0 + 1, increment the address of RAM to
                    ;access next byte
DJNZ R5, HOME       ; decrease R5 by 1 and jump to HOME if R5 is
                    ;not equal to zero
MOV 30H, R6         ; [30] ← R6, move the lower byte to memory
                    ;address 30H
MOV 31H, R7         ; [31] ← R7, move the upper byte to memory
                    ;address 31H
END
```

Example 5:

ASCII character string is stored in the program memory starting at 300H. Send each character to port 2.

Problem Analysis: Since the data is stored in the program memory (ROM), DPTR must be used to point the address of ROM. Using MOVC instruction, the data can be retrieved and further manipulated as required.

Source Code:

```
ORG 00H
MOV DPTR, #300H     ; load address of data into DPTR
HOME:
CLR A               ; A ← 0, clear the content of A
```

```
MOVC A, @A + DPTR   ; A ← [A + DPTR], load ROM content of
                    ; address (A + DPTR) to A
JZ EXIT             ; jump out of loop if last character is detected
                    ; which is 0
MOV P2, A           ; P2 ← A, move content of accumulator to
                    ; port 2
INC DPTR            ; DPTR ← DPTR + 1, to point to next character
                    ; of ROM
SJMP HOME           ; Continue the loop
ORG 300H
DB                  "ASSEMBLY PROGRAM", 0
EXIT:
NOP
END
```

9.2.2 Delay Calculation in Assembly Program

Actual time of the execution of instructions can be determined by making use of the operating frequency of the microcontroller and machine cycles required by the instructions. The total machine cycles required by the instruction is multiplied by time duration of one machine cycle to calculate the total time.

```
DELAY:
                 (m)
          MOV R4, #64H   ; MC = 1, executes only once
                 (n)                        (m)
AGAN:     MOV R3, #0AH   ; MC = 1, executes 100 times
                                            (m)   (n)
AGA:      DJNZ R3, AGA   ; MC = 2, executes 100 x 10 = 1000 times
                                            (m)
          DJNZ R4, AGAN  ; MC = 2, executes 100 times
RET                      ; MC = 2, executes 1 time
```

In 8051, crystal frequency is 11.0592MHz and one machine cycle (MC) is equal to 12 clock cycles.

So, 1 MC = 1.085μs.

Total machine cycles in DELAY subroutine = $1 + 1\text{x}\overset{m}{100} + 2\text{x}\overset{mn}{1000} + 2\text{x}\overset{m}{100} + 2 = 2303$

Total time duration = 2303 x 1.085μs = 2.49ms

Scanned with CamScanner

anode: current enters
cathode: current leaves

9.3 Interfacing with Seven Segment Display

1. Seven Segment Configurations

i. Pin Configurations

The figure shows the pin configuration of seven segment display which consists of 10 pins; eight pins to control the LEDs and two common pins which are grounded or connected to VCC based on common cathode or common anode configuration.

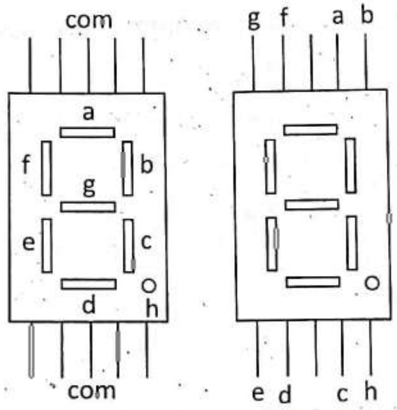


Figure 9.6: Pin configurations of seven segment display

ii. Modes of Configurations

There are two modes of configurations: common anode configuration and common cathode configuration. In common anode configuration, anodes of all LEDs are connected together to form a common pin which must be connected to high logic voltage. In common cathode configuration, cathodes of all LEDs are connected together to form a common pin which must be connected to low logic voltage.

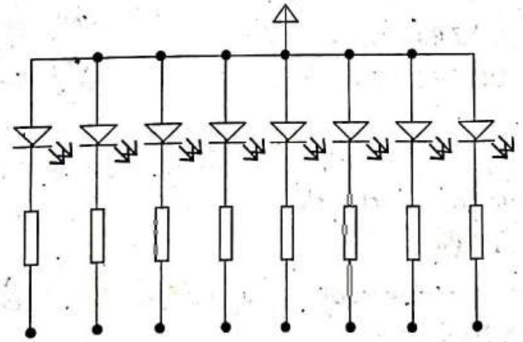


Figure 9.7: Common anode configuration

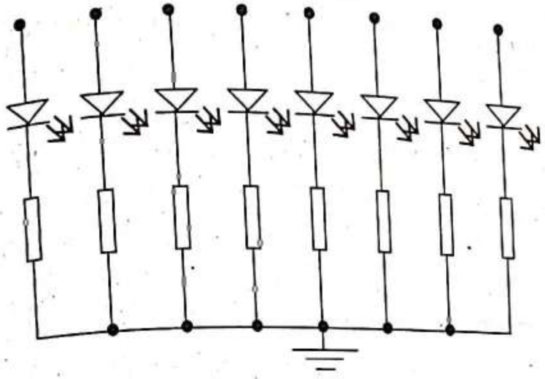


Figure 9.8: Common cathode configuration

iii. Lookup Table of HEX Equivalent

For common anode configurations, low logic must be provided to the pins of the seven segment display to glow a particular led. The equivalent hex values are sent through the port of microcontroller. However, designer must be aware of the driving circuit which should provide the equivalent hex to the pins of seven segment display.

Common Anode Configurations										For Common Cathode mode
Digits	Individual LEDs Illuminated								HEX	HEX
	h	g	f	e	d	C	b	a		
0	1	1	0	0	0	0	0	0	0xC0	0x3F
1	1	1	1	1	1	0	0	1	0xF9	0x06
2	1	0	1	0	0	1	0	0	0xA4	0x5B
3	1	0	1	1	0	0	0	0	0xB0	0x4F
4	1	0	0	1	1	0	0	1	0x99	0x66
5	1	0	0	1	0	0	1	0	0x92	0x6D
6	1	0	0	0	0	0	1	0	0x82	0x7D
7	1	1	1	1	1	0	0	0	0xF8	0x07
8	1	0	0	0	0	0	0	0	0x80	0x7F
9	1	0	0	1	0	0	0	0	0x90	0x6F

Scanned with CamScanner

2. Interfacing Seven Segment

Before connecting the seven segment display to the port of microcontroller, the current requirement of the seven segment display along with the source current and sink current capacity of the microcontroller must be examined. However, it is always better to use the driving circuit rather to connect seven segment display directly to the port of microcontroller.

i. Hardware Connections

Any port of the microcontroller can be used to connect to the seven segment display through the driving circuit. The driving circuit may be in the form of an IC which can sink or source high current. The circuit configurations can vary depending on the designer. IC ULN2003 is an example of a current sinker while IC L293D can be a good source of current for driving circuits.

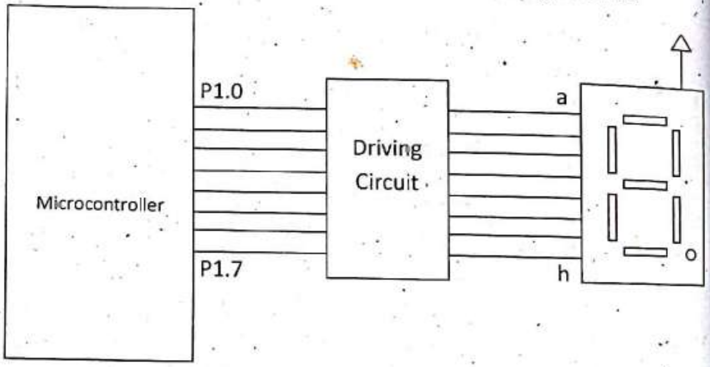


Figure 9.9: Connection of common anode seven segment with microcontroller

ii. Coding Implementations

In assembly language, simple MOV instructions can be used to transfer HEX value to the seven segment display. For example, MOV P2, #92 will display the digit 5 for common anode configurations. Using C programming language, simple assignment can be done. For example, P2 = 0x92; will display digit 5. However, appropriate delay or repetition mechanism must be used to ensure that the digit displays for certain duration and becomes observable to the designer or user.

oblem 1:

Write an assembly and C language program to generate a pulse of 50% duty cycle at pin P2.3 of 8051 microcontroller.

lution:

Problem Analysis: Duty cycle of 50% represents equal ON and OFF time at pin P2.3, so an arbitrary delay is required after setting P2.3 and after resetting P2.3.

Source Code:

```
ORG 00H
        CLR P2.3
Back:
        CLR P2.3
        LCALL DELAY
        SETB P2.3
        LCALL DELAY
        SJMP Back
ORG 300H
DELAY:
        MOV R5, #64H
AGAIN:  MOV R4, #0FFH
AGAN:   MOV R3, #08H
AGA:    DJNZ R3, AGA
        DJNZ R4, AGAN
        DJNZ R5, AGAIN
        RET
END
```

global voribles are stored in RAM

In C programming language

```
#include<at89x52.h>
void delay(unsigned char x)
{
        int i, j;   local variables are stored in stack or registers
        for(i=0;i<x;i++)
            for(j=0;j<1275;j++);
}
void main()
{
        P2_3 = 0;
        while(1)
        {
                P2_3 = 1;
                delay(50);
                P2_3 = 0;
                delay(50);
        }
}
```

Problem 2:

Control the LED connected at 2.1 by a SWITCH which is connected to P1.3. ON/OFF status of LED is defined by ON/OFF status of SWITCH.

Solution:

Problem Analysis: This is a simple data movement problem in which a bit from one pin P1.3 of microcontroller is assigned to another pin P2.1. The code can vary based of LED configuration and SWITCH configuration used. Two cases are given below:

CASE I: The SWITCH will generate high logic when pressed and LED will glow when high logic is assigned to pin 2.1.

Source Code:

```
ORG 00H

CLR P2.1

SETB P1.3

Back:

        MOV C, P1.3
        MOV P2.1, C
        SJMP Back

END
```

```
#include<at89x52.h>
void main()
{
        P2_3 = 0;
        P1_3 = 1;
        while(1)
        {
                P2_1 = P1_3;
        }
}
```

CASE II: The SWITCH will generate low logic when pressed and LED will glow when high logic is assigned to pin 2.1.

Source Code:

```
ORG 00H
CLR P2.1
SETB P1.3
Back:
        MOV C, P1.3
        CPL C
        MOV P2.1, C
        SJMP Back
END
```

```
#include<at89x52.h>
void main()
{
        P2_3 = 0;
        P1_3 = 1;
        while(1)
        {
                P2_3 = !P1_3;
        }
}
```

Problem 3:

Using an assembly and C language program, generate a pulse of 75% duty cycle at pin P1.7 when the switch connected to P1.1 is ON.

Solution:

Problem Analysis: Duty cycle of 75% represents ON time three times more than OFF time at pin P1.7. The status of P1.1 must be checked continuously. Based on the logic level at P1.1 after button is pressed, delay in case of ON time must be three times more than that of OFF time. In the code below, we assume the switch connects P1.1 to ground when pressed and P1.1 is connected to VCC when not pressed. So, a low logic at P1.1 will generate the required pulse using appropriate delay.

Source Code:

```
ORG 00H
        CLR P1.7
        SETB P1.1
BACK:
        MOV C, P1.1
        JC BACK
        SETB P1.7
        LCALL DELAY
        LCALL DELAY
        LCALL DELAY
        CLR P1.7
        LCALL DELAY
        SJMP BACK

ORG 300H
DELAY:
        MOV R5, #64H
AGAIN:  MOV R4, #0FFH
AGAN:   MOV R3, #08H
AGA:    DJNZ R3, AGA
        DJNZ R4, AGAN
        DJNZ R5, AGAIN
        RET
END
```

3 time delay cuz 75% duty cycle.

```
# include<at89x52.h>
# define OUT P1_7
# define SW P1_1
void delay(unsigned int x)
{
        int i,j;
        for(i=0;i<x;i++);
        for(j=0;j<1275;j++);
}
void main()
{
        OUT = 0;
        SW = 1;
        while(1)
        {
                if(SW == 0)
                {
                  OUT = 1;
                  delay(300);
                  OUT = 0;
                  delay(100);
                }
        }
}
```

Problem 4:

Write an assembly and C language program to generate a count from 0 to 9 using a seven segment display. Use Common Cathode configurations.

Solution:

Problem Analysis: The equivalent HEX values are directly assigned one by one to the required port. Certain delay after each digit display can be placed to control the speed of count.

Scanned with CamScanner

Source Code:

```
ORG 00H
    MOV P2, #00H
BACK:
    MOV P2, #3FH
    LCALL DELAY
    MOV P2, #06H
    LCALL DELAY
    MOV P2, #5BH
    LCALL DELAY
    MOV P2, #4FH
    LCALL DELAY
    MOV P2, #66H
    LCALL DELAY
    MOV P2, #6DH
    LCALL DELAY
    MOV P2, #7DH
    LCALL DELAY
    MOV P2, #07H
    LCALL DELAY

    MOV P2, #7FH
    LCALL DELAY
    MOV P2, #6FH
    LCALL DELAY
    SJMP BACK

ORG 300H

DELAY:
        MOV R5, #64H
AGAIN:  MOV R4, #0FFH
AGAN:   MOV R3, #08H
AGA:    DJNZ R3, AGA
```

```
        DJNZ R4, AGAN
        DJNZ R5, AGAIN
        RET

END
```

TERNATIVE CODE:

Problem Analysis: The HEX values are not directly assigned rather ed in memory and accessed using data pointer (DPTR). DPTR is used to nt to the HEX values and MOVC instruction must be used to access the X values. MOVC uses accumulator to represent offset as well as data. The ue of accumulator is added to DPTR to represent the address of memory d finally the data is stored into accumulator.

urce Code:

```
ORG 00H
    MOV P2, #00H
    MOV R6, #00H
    MOV DPTR, #DIGITS
MAIN:
    MOV A, R6
    MOVC A, @A+DPTR
    MOV P2, A
    LCALL DELAY
    INC R6
    CJNE R6, #0AH, MAIN

MOV R6, #00H
    SJMP MAIN

DELAY:
        MOV R3, #0F0H
DEL1:   MOV R2, #0FAH
DEL2:   DJNZ R2, DEL2
        DJNZ R3, DEL1
RET
```

```
# include<at89x52.h>
# define DISPLAY P2
void delay(unsigned int x)
{
        unsigned int i,j;
        for(i=0;i<x;i++)

        for(j=0;j<1275;j++);
}
char digits[] = {0x3F, 0x06, 0x5B,
0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F,
0x6F};
void main()
{
        unsigned char i;
        DISPLAY = 0x00;
        while(1)
        {
          for(i=0;i<10;i++)
          {
DISPLAY = digits[i];
                delay(100);
          }
        }
```

Scanned with CamScanner

```
DIGITS:
        DB 3FH, 06H, 5BH, 4FH, 66H
        DB 6DH, 7DH, 07H, 7FH, 6FH
END
```

Problem 5:

A PUSH BUTTON is connected to P1.1, increase the count in SEVEN SEGMENT when the button is pressed.

Solution:

Source Code:

```
ORG 00H
    SETB P1.1
    MOV P2, #00H
    MOV R6, #00H
    MOV DPTR, #DIGITS
MAIN:
    MOV A, R6
    MOVC A, @A+DPTR
    MOV P2, A
    MOV C, P1.1
    JC MAIN
    LCALL DELAY
    INC R6
    CJNE R6, #0AH, MAIN
    MOV R6, #00H
    SJMP MAIN
DELAY:
        MOV R3, #0F0H
DEL1:   MOV R2, #0FAH
DEL2:   DJNZ R2, DEL2
        DJNZ R3, DEL1
RET
```

```
# include<at89x52.h>
# define DISPLAY P2
# define SW P1_1
void delay(unsigned int x)
{
        unsigned int i,j;
        for(i=0;i<x;i++)

        for(j=0;j<1275;j++);
}
char digits[] = {0x3F, 0x06, 0x5B,
0x4F, 0x66, 0x6D, 0x7D, 0x07,
0x7F, 0x6F};
void main()
{
        unsigned char i = 0;
        DISPLAY = 0x00;
        while(1)
        {
            DISPLAY = digits[i];
            if(SW == 0)
            {
                i++;
                if (i>9)
                        i = 0;
                delay(20);
            }
        }
}
```

```
DIGITS:
        DB 3FH, 06H, 5BH, 4FH, 66H
        DB 6DH, 7DH, 07H, 7FH, 6FH

END
```

oblem 6:

Using two seven segment displays, build a down counter which counts from 99 to 00. Make appropriate assumptions wherever necessary. (Time Multiplexing is used)

olution:

Description: Two ports can be used to display two digits. However, it is better to use a single port for both seven segment displays and control their display through control lines. Two digits separately are sent at different instant of time. Corresponding seven segment display must be enabled while the digits are sent at different instant. However, the time duration between sending of lower digit and upper digit must not be high. High delay can lead to flickering effect which causes both digits to be displayed one by one rather than simultaneously. Also the speed of count can be controlled by using appropriate repetition of each display of both digits.

Source Code:

```
ORG 00H
    MOV P2, #00H
    MOV R6, #09H          ; counter for lower byte
    MOV R7, #09H          ; counter for upper byte
    MOV R5, #07h          ; to control speed of counter
    MOV P3, #00H
    MOV DPTR, #LABEL1     ; loads the starting address of hex code
                          ; list to DPTR

MAIN:
    MOV A, R6
    SETB P3.0             ; activates 2nd display to display lower
                          ; byte
                          ; deactivates the 1st display
    CLR P3.1
    LCALL DISPLAY
```

Scanned with CamScanner

```
        LCALL DELAY
        MOV A, R7
        SETB P3.1               ; activates 1st display
        CLR P3.0                ; deactivates the 2nd display
        LCALL DISPLAY
        LCALL DELAY

        DJNZ R5, MAIN           ; repetition of same display to control
                                ; speed
        MOV R5, #07H
        DEC R6                  ; decrease value of R6
        CJNE R6, #-1, MAIN      ; compare unless R6 becomes less than
                                ; zero
        MOV R6, #09H
        DEC R7
        CJNE R7, #-1, MAIN
        MOV R7, #09H
        SJMP MAIN
DISPLAY:
        MOVC A, @A+DPTR         ; load HEX value to accumulator from
                                ; memory
        MOV P2, A               ; sending HEX value to SEVEN SEGMENT
                                ; through P2
RET

DELAY:
        MOV R3, #0F0H
DEL1: MOV R2, #0FAH
DEL2: DJNZ R2, DEL2
        DJNZ R3, DEL1
RET

LABEL1:                         ; represents starting address of HEX
                                ; value list
        DB 3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H, 7FH, 6FH
END
```

quivalent Code in C Language

Description: Within infinite loop, the first loop is used to select the per byte, the second loop is used to select the lower byte and the third op is used to control the speed of count.

urce Code:

```
# include<at89x52.h>
# define DISPLAY P2
# define SEL0 P3.1
# define SEL1 P3.0
void delay(unsigned int x)
{
        unsigned int i, j;
        for(i=0;i<x;i++)
                for(j=0;j<1275;j++);
}
char digits[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F,
0x6F};
void main()
{
        char i, j, k;
        DISPLAY = 0x00;
        while(1)
        {
        for(i = 9; i >= 0; i--)             // loop for upper byte
        {
                for(j = 9; j >= 0; j--)     // loop for lower byte
                {
                        // loop to control the speed of count
                        for(k = 0; k<7; k++)
                        {
                                DISPLAY = digits[j];
                                SEL0 = 0;
```

Scanned with CamScanner

```
                SEL1 =1;
                delay(5);
                DISPLAY = digits[i];
                SEL0 = 1;
                SEL1 = 0;
                delay(5);
            }
        }
}}}
```

Problem 7:

Generate a periodic wave having period of 15ms and duty cycle 20% in 8051 using assembly programming. The wave should be produced at pin zero of port 2 (P2.0), the XTAL frequency is 11.0592MHZ and use Timer 1 in mode 0. (13-bit timer mode).

[2076 Bhadra]

Solution:

```
    ORG 00H
    CLR P2.0
MAIN:
    SETB P2.0
    LCALL DELAY
    CLR P2.0
    LCALL DELAY
    LCALL DELAY
    LCALL DELAY
    LCALL DELAY
    SJMP MAIN
```

Period = 15ms
Duty cycle = 3/15 = 20%
On time = 3ms, Off-time = 12ms
For Mode 0 timer (13 bit timer): It counts from 0000 to 1FFF. We need to set the values of TH1 and TL1 to generate delay of 3 ms.
Here, 1FFFH = 8191 in decimal.
To calculate starting point of count
(8192 – x)*1.085us = 3ms
x = 5427 in decimal (1533 in Hex)
Hence, TH1 = 15H and TL1 = 33H must be loaded to generate 3ms.

```
DELAY:
    MOV TMOD, #00H        ; to configure timer in mode 0
    MOV TH1, #15H
    MOV TL0, #33H         ; counting starting point set to 1533
    SETB TR1              ; starts the timer (TF1 is set after 1FFF)
BACK: JNB TF1, BACK       ; jump to BACK if TF1 bit is not set
    CLR TR1               ; stops the timer
    CLR TF1               ; clear the timer overflow flag (TF1 bit)
    RET
END
```

Problem 8:

Using 8051 instructions, control rate of blink of LED at pin 1.1 by two switches at P2.1 and P2.2 (one switch to increase the rate of blink, another to decrease the rate of blink). [2076 Baishakh]

Solution:

Assembly coding for given problem statement is as below:

```
ORG 00H
    CLR P1.1          ; making P1,1 as output pin
    SETB P2.1         ; P2.1 as input
    SETB P2.2         ; P2.2 as input
    MOV R2, #80H      ; defines initial value for delay (or blink rate)

FAST:
    LCALL BLINK
    MOV C, P2.1
    JNC SLOW
    LCALL SPEEDUP
SLOW:
    MOV C, P2.2
    JNC FAST
    LCALL SPEEDDOWN
    SJMP FAST
DELAY:
```

Scanned with CamScanner

```
        MOV A, R2
        MOV R5, A
L3:     MOV R4, #3FH
L2:     MOV R3, #08H
L1:     DJNZ R3, L1
        DJNZ R4, L2
        DJNZ R5, L3
        RET
SPEEDUP:
        MOV A, R2
        SUBB A, #0AH
        MOV R2, A
RET
SPEEDDOWN:
        MOV A, R2
        ADD A, #0AH
        MOV R2, A
RET
BLINK:
        SETB P1.1
        LCALL DELAY
        CLR P1.1
        LCALL DELAY
RET
END
```

Problem 9:

Design a circuit with 7 segments display which is used as a counter watch which display second and minute. [2076 Baishakh]

Solution:

To solve the given problem, we need four seven segment displays; two for minutes and two for seconds. And it can be solved using single port along with time multiplexing concept, or another method is by using individual port for each seven segment displays. We have already implemented time-multiplexing concept in Problem 6, so the same concept can be used to implement this problem as well. However, we will be using separate port for each seven segment displays in this problem.

The circuit configuration and respective code for 7 segment display that is used as a counter watch is as shown below. Minute is represented by port P0 (upper byte) and P1 (lower byte) while second is represented by P2 (upper byte) and P3 (lower byte). Port 0 should be used with pull up registers which is shown in the diagram as well. Also, pull up registers are required for all used pins of PORT 0 which is not shown.

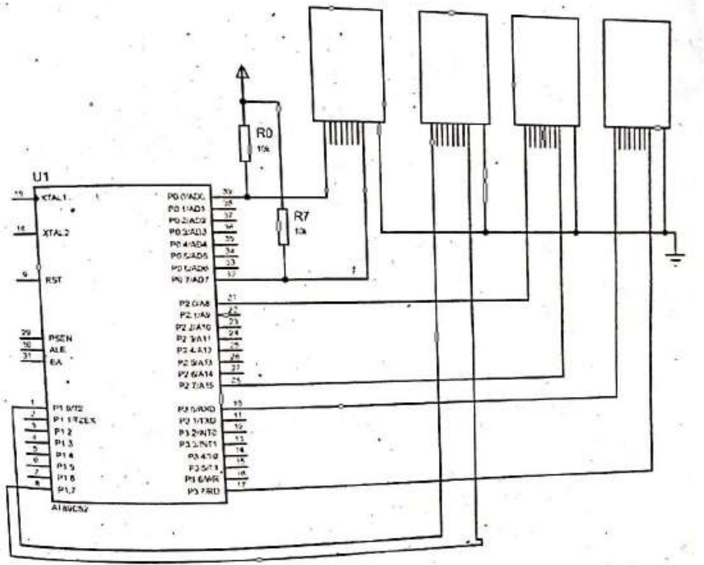


Figure: Counter watch using seven segment for minute and second

```
ORG 00H
        MOV P0, #00H
        MOV P1, #00H
        MOV P2, #00H
        MOV P3, #00H
        MOV R1, #00H
```

Scanned with CamScanner

```
        MOV R2, #00H
        MOV R3, #00H
        MOV R4, #00H
        MOV DPTR, #DIGITS
MSBMIN:
        MOV A, R1
        MOVC A, @A+DPTR
        MOV P0, A
LSBMIN: MOV A, R2
        MOVC A, @A+DPTR
        MOV P1, A
MSBSEC: MOV A, R3
        MOVC A, @A+DPTR
        MOV P2, A
LSBSEC: MOV A, R4
        MOVC A, @A+DPTR
        MOV P3, A
        LCALL DELAY
        INC R4
        CJNE R4, #0AH, LSBSEC
        MOV R4, #00H
        INC R3
        CJNE R3, #06H, MSBSEC
        MOV R3, #00H
        INC R2
        CJNE R2, #0AH, LSBMIN
        MOV R2, #00H
        INC R1
        CJNE R1, #06H, MSBMIN
        MOV R1, #00H
        SJMP MSBMIN
```

```
DELAY:
        MOV R5, #0F0H
DEL1:   MOV R6, #0FAH
DEL2:   DJNZ R6, DEL2
        DJNZ R5, DEL1
RET

DIGITS:
        DB 3FH, 06H, 5BH, 4FH, 66H
        DB 6DH, 7DH, 07H, 7FH, 6FH
END
```

Use registers R0, R5, R6 or R7 in delay routine as R1, R2, R3, and R4 are already in use in main code.

Problem 10:

Write an assembly code to blink the 8 led connected at port 2 of 8051 microcontroller. **[2075 Baishakh]**

Solution:

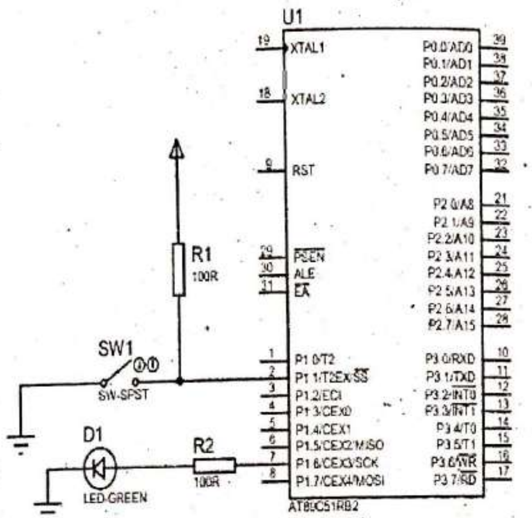
```
            ORG 00H
            MOV P2, #00H
    MAIN:   MOV P2, #0FFH
            LCALL DELAY
            MOV P2, #00H
            LCALL DELAY
            SJMP MAIN
    ORG 300H
    DELAY:
            MOV R3, #0F0H
    DEL1:   MOV R2, #0FAH
    DEL2:   DJNZ R2, DEL2
            DJNZ R3, DEL1
    RET
    END
```

Scanned with CamScanner

Problem 11:

Show the connection of LED at P1.7 and switch at P1.1 of 8051 microcontroller. Using an assembly language, generate a pulse of 75% duty cycle at pin P1.7 when the switch at P1.1 is ON. **[2074 Bhadra]**

Solution:



Assembly coding for given problem is:

```
ORG 00H
        SETB P1.1
        CLR P1.7
LOOP:   MOV C, P1.1
        JNC LOOP
        SETB P1.7
        LCALL DELAY
        LCALL DELAY
        LCALL DELAY
        CLR P1.7
        LCALL DELAY
        SJMP LOOP
ORG 300H
```

3 delays are needed 75% duty cycle.

```
DELAY:
        MOV R3, #0F0H
DEL1:   MOV R2, #0FAH
DEL2:   DJNZ R2, DEL2
        DJNZ R3, DEL1
        RET
END
```

Problem 12:

Write an assembly language programming for 8051 microcontroller to read the data from switches connected to port 1 and send it to port 2 for display in LED. **[2073 Magh]**

Solution:

```
        ORG 00H
        MOV P1, #0FFH
        MOV P2, #00H
Main:
        MOV A, P1
        MOV P2, A
        SJMP Main
        END
```

Problem 13:

Write a short program in assembly language that computes a precise 2.5ms delay using 8051 microcontroller. **[2071 Magh]**

Solution:

```
        ORG 00H
                MOV R4, #64H    ; MC = 1, executes only once
OUTER:          MOV R3, #0AH    ; MC = 1, executes 100 times
INNER:          DJNZ R3, INNER  ; MC = 2, executes 100 x 10 = 1000 times
                DJNZ R4, OUTER  ; MC = 2, executes 100 times
                                ; MC = 1, executes 1 time
                NOP             ; MC = 1, executes 1 time
                NOP
```

Scanned with CamScanner

```
        NOP             ; MC = 1, executes 1 time
        NOP             ; MC = 1, executes 1 time
    END
```

In 8051, crystal frequency is 11.0592MHz and one machine cycle (MC) is equal to 12 clock cycles.

So, 1 MC = 12/11.0592MHz = 1.085μs.

Total machine cycles required to execute above code is

= 1 + 1x100 + 2x1000 + 2x100 + 1 + 1 + 1 + 1

= 2305 MC

Total time duration = 2305 x 1.085μs = 2.5ms

However, to generate the specified delay, we need to determine total repetition, initial value of R4 and R3 along with number of NOP required. In this case, we have to follow following steps:

- List the given values

  Time required (T) = 2.5ms

  1 MC (t) = 1.085 μs (for 8051 microcontroller)

- Calculate total required machine cycles

  Total MC = T/t = 2305

  If total required MC is greater than 255 then one way to generate more than 255 MC is by using nested loop.

- Values and iterations to generate required MC

  We need to know the MC required by branching instruction and others instructions used in the code, in above case it is 2 MC for DJNZ, 1 MC for MOV and 1 MC for NOP.

  Set outer loop initializer as 100 (64H) or 200 (C8), and assume inner loop initializer as 10 (0AH). Then, adjust the value of initializer of inner loop based on requirement. For few missing MC, add number of NOPs if required. Also we can add another loop in the nested structure, to generate higher values of delay.

VHDL

Introduction

VHDL Code Structure

Data types, Data Objects and Operators

Statements in VHDL

Standard Architectures

FSM Design

10.1 Introduction

VHDL is a hardware description language. It is used to describe the behavior of an electronic system, which further enables designer to implement the physical system. VHDL stands for VHSIC Hardware Description Language, where VHSIC is an acronym for Very High Speed Integrated Circuits.

The main purpose of VHDL is to model and synthesize digital circuits. Simulation and testing of the design for the optimum operation can be done using VHDL model of the system. Also, digital integrated circuits for particular operations can be created using VHDL or other hardware description languages. Finally, VHDL code can be used to create actual functional system. Hence, VHDL code can be used either to implement the circuit in a programmable device or can be forwarded for fabrication.

VHDL Invariants

- It is not case sensitive.
- It is not sensitive to white space.
- Comments begin with two consecutive dashes ("--").
- Parenthesis usage is optional in many cases.
- Every statement in VHDL is terminated with a semicolon.
- Statements are inherently concurrent. Only statements placed inside a PROCESS, FUNCTION, or PROCEDURES are executed sequentially.

10.2 VHDL Code Structure

VHDL code comprises of at least the three fundamental sections: LIBRARY Declaration, ENTITY and ARCHITECTURE. LIBRARY is a collection of

Scanned with CamScanner

pre-defined set of codes that can be re-used or shared by various designs. ENTITY specifies the I/O connections of the system. ARCHITECTURE contains the code that describes how the circuit should function.

i. LIBRARY DECLARATION

The general from is:

```
LIBRARY LIBRARY_NAME;
USE LIBRARY_NAME.PACKAGE_NAME.PACKAGE_PARTS;
```

Example:

```
LIBRARY IEEE;
USE Ieee.std_logic_1164.all;
```

The libraries *STD* and *WORK* are made visible by default, so they are not required to declare. However, *STD_LOGIC_1164* package of *IEEE* library must be declared when STD_LOGIC data type is used in the design. Similarly, for SIGNED and UNSIGNED data types and its related arithmetic and comparison operations, package *STD_LOGIC_ARITH* of LIBRARY *IEEE* must be declared.

ii. ENTITY

The VHDL ENTITY declaration describes the interface or the external representation of the circuit. An ENTITY is a list of all input and output pins with its specification such as data type and data direction mode.

Its syntax is:

```
ENTITY ENTITY_NAME IS
PORT(
        PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
        PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
        ...
    );
END ENTITY_NAME;
```

Here, ENTITY_NAME and PORT_NAME are identifiers. The SIGNAL_MODE which defines the direction of signal can be IN, OUT, INOUT, or BUFFER. IN and OUT are unidirectional pins, while INOUT is bidirectional. The data type or SIGNAL_TYPE can be BIT, STD_LOGIC, INTEGER, etc.

Example 1: Entity of AND gate with two inputs each of one bit.

```
ENTITY AND_GATE IS
 PORT(
         IN_A  : IN STD_LOGIC;
         IN_B  : IN STD_LOGIC;
         OUT_Z : OUT STD_LOGIC
     );
END AND_GATE;
```

Example 2: Entity of 4x1 MUX with each input of three bits

```
ENTITY MUX IS
 PORT(
         A, B, C, D : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
         SEL : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         Z : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
     );
END MUX;
```

iii. ARCHITECTURE

The ARCHITECTURE describes how the circuit should function. It describes the internal implementation of the associated entity. There are several models that are followed by architecture to describe the operation of the circuit.

The general form of ARCHITECTURE is:

```
ARCHITECUTRE architecture_name OF entity_name IS
     [declarations]
BEGIN
     [code]
END architecture_name;
```

Here, declarative part is optional and includes signal and constant declarations. Code part includes different VHDL statements describing the system to be designed.

Example 1: ARCHITECTURE of AND gate with two inputs each of one bit.

```
Architecture and_arch of and_gate is
Begin
    OUT_Z <= IN_A AND IN_B;
END and_arch;
```

10.3 Data types, Data Objects and Operators

1. Data Types

Data types represent the type of information stored by the variable or constant. It can be of two types which are discussed in the following section.

i. Pre-Defined Data Types

VHDL contains a series of pre-defined data types. Such data type definitions can be found in various packages or libraries.

- Package STANDARD of library STD includes BIT, BOOLEAN, INTEGER, and REAL.
- Package STD_LOGIC_1164 of library IEEE includes STD_LOGIC and STD_ULOGIC.

Various pre-defined data types are listed in the table below:

SN	TYPE	LEVEL/RANGE	DESCRIPTION
1.	BIT	2 LOGIC LEVEL	0, 1
2.	BIT_VECTOR	2 LOGIC LEVEL	0, 1
3.	STD_LOGIC	8 VALUED LOGIC	X, 0, 1, Z, W, L, H, -
4.	STD_LOGIC_VECTOR	8 VALUED LOGIC	X, 0, 1, Z, W, L, H, -
5.	STD_ULOGIC	9 VALUED LOGIC	U, X, 0, 1, Z, W, L, H, -
6.	STD_ULOGIC_VECTOR	9 VALUED LOGIC	U, X, 0, 1, Z, W, L, H, -
7.	BOOLEAN	2 VALUES	TRUE, FALSE
8.	INTEGER	-2147483647 TO +2147483647	32 BIT NUMBER constant count :integer:=50
9.	NATURAL	0 TO +2147483647	
10.	REAL	-1.0E38 TO +1.0E38	
11.	1st bit of array is sign bit • +ve no. have 0 on their 1st bit SIGNED • -ve no. have 1 on their 1st bit and are stored in 2's complement form	POSITIVE AND NEGATIVE	USED IN ARITHMETIC OPERATIONS
12.	UNSIGNED	POSITIVE	↳ signal A : unsigned (3 downto 0) := '1101'
13.	PHYSICAL	TIME, VOLTAGE	USED IN SIMULATION

ray of ...logic n be d to form arithmetic ration ke logic_vector

Here, various logic levels represent: 'U' – Unresolved, 'X' – Forcing Unknown, '0' – Forcing Low, '1' – Forcing High, 'Z' – High Impedance, 'W' – Weak unknown, 'L' – Weak Low, 'H' – Weak High, '-' – Don't care. STD_LOGIC levels are intended for simulation only. When a node has two STD_LOGIC signals connected, then conflicting logic levels are resolved automatically in case of STD_LOGIC whereas such conflict is not resolved in STD_ULOGIC. For arithmetic operations using STD_LOGIC, packages STD_LOGIC_SIGNED and STD_LOGIC_UNSIGNED must be used.

ii. User Defined Data Types

VHDL allows users to define their own data types. There are two categories of user- defined data types.

a. User-Defined Integer Type

General form:

TYPE TYPE_NAME IS RANGE LOW_VALUE TO HIGH_VALUE;

Example:

```
TYPE TEMPERATURE IS RANGE -125 TO 125;
TYPE MARKS IS RANGE 0 TO 100;
```

b. User-Defined Enumerated Type

General Form:

TYPE TYPE_NAME IS (VALUE1, VALUE2... VALUEN);

Example:

```
TYPE COLOR IS (RED, GREEN, BLUE, WHITE);
```

Based on bits requirement encoding of enumerated type is done sequentially and automatically, unless specified.

Scanned with CamScanner

2. Data Objects

An object is an item in VHDL that has both name and a specific type. Commonly used data objects are signals, variables and constants.

i. Constants

Constants are used to assign default values in the code. It can be declared in PACKAGE, ENTITY or ARCHITECTURE. Declaring CONSTANTS in PACKAGE makes if global, since PACKAGE can be used by several entities. If it is declared in an ENTITY, it can be shared by all ARCHITECTURE that follows that ENTITY. When defined within ARCHITECTURE the scope of CONSTANTS are limited to that ARCHITECTURE only.

Declaration:

```
CONSTANT name : TYPE := value;
```

Examples:

```
CONSTANT high : STD_LOGIC :='1';
CONSTANT count : INTEGER :=10;
```

ii. Signal

ntax for declaring signal:

gnal

d <name>: <type>;

onally signal can declare with an ial value:

d <name>: <type> := <initial value>;

Signal is used to pass value in and out of the circuit and within internal units. It simply represents interconnection of circuit. All ports of ENTITY are signals by default. The change in the SIGNAL may not be updated immediately, since the value is more likely to get updated after the completion of its corresponding PROCESS, FUNCTION or PROCEDURE. Similar to CONSTANT, it can be declared in PACKAGE, ENTITY or ARCHITECTURE.

Declaration:

```
SIGNAL name: TYPE [range] [:= initial_value];
```

The part inside the square bracket may or may not be present depending upon data types used and requirement of initialization.

Examples:

```
SIGNAL start : STD_LOGIC := '0';
SIGNAL count : INTEGER RANGE 0 TO 100;
```

iii. Variable

Variable represents the local information. Its value cannot be passed out directly. The change in value is immediately updated; new value can be promptly used in next line of code. It can be declared and used inside a PROCESS, FUCNTION or PROCEDURE.

Declaration:

```
VARIABLE name: type [range] [:= initial value];
```

Examples:

```
VARIABLE count: INTEGER :=0;
VARIABLE a : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

3. Operators

The various operators supported by VHDL are tabulated below:

i. Assignment Operators

initializatn of both signal & variable is done with ':='

SN	Operator	Assign Value To	Examples
1.	<=	Signal	X <= '1'; Y<="101;
2.	:=	Variable, constant, generic, and for initialization	Z := "1001"; Z is a variable
3.	=>	Individual Elements or with OTHERS	W <= (0=> '1', OTHERS => '0') LSB assigned 1 and others as 0

used in case Statement

ii. Logical Operators

SN	Operators	Description/Example	Supported Data Type
1.	NOT	Inverts the signal, High Precedence	BIT STD_LOGIC STD_ULOGIC BIT_VECTOR STD_LOGIC_VECTOR STD_ULOGIC_VECTOR
2.	AND	Result high when both inputs is high	
3.	OR	Result high when one of the inputs is high	
4.	NAND	X <= a NAND b	
5.	NOR	Z <= NOT a NOR B	
6.	XOR	Complements the bit when XORed with 1	
7.	XNOR	Complements the bit when XNORed with 0	

Scanned with CamScanner

iii. **Relational Operators**

SN	Operators	Description
1.	=	Equal to
2.	/=	Not equal to
3.	<	Less than
4.	>	Greater than
5.	<=	Less than or equal to
6.	>=	Greater than or equal to

iv. **Arithmetic Operators**

SN	Operator	Meaning	Description
1.	+	Addition	
2.	-	Subtraction	
3.	*	Multiplication	
4.	/	Division	Limited to powers of two
5.	**	Exponentiation	Limited to powers of two
6.	MOD	Modulus	X MOD Y results value with sign of Y
7.	REM	Remainder	X REM Y results value with sign of X
8.	ABS	Absolute Value	
Avoid Using MOD operator when dealing with negative numbers			

v. **Shift Operators**

SN	Operator	Meaning	Description
1.	SLL	Shift Left Logic	Zeros are fed from one end and bits are lost from other end. Sign bit never changes in arithmetic shift.
2.	SRL	Shift Right Logic	
3.	SLA	Shift Left Arithmetic	"10101" SLL 3 results in "01000"
4.	SRA	Shift Right Arithmetic	
5.	ROL	Rotate Left	"1001" ROL 2 results in "0110"
6.	ROR	Rotate Right	

vi. **Concatenation Operator**

The concatenation operator (&) is used to combine values of similar data type. The following example will illustrate the use of concatenation operator.

Example:

```
signal A, B : std_logic_vector (3 downto 0); -- Signal A and B of 4 bits
signal C : std_logic_vector (5 downto 0);    -- Signal C of 6 bits
signal D : std_logic_vector (7 downto 0);    -- Signal D of 8 bits
C <= A & "00" ;                              -- 4 bits of A and two bits
                                             "00" assigned to C
D <= B & A ;                                 -- 4 bit of A and B
                                             combined and assigned to
                                             D
```

10.4 Statements in VHDL

1. Concurrent Statements

i. Concurrent Signal Assignment

Syntax:

```
Target <= expression;
```

Examples:

```
A <= B NAND C;
X <= (D OR E) AND (F AND G);
```

ii. Conditional Signal Assignment

Syntax:

```
Target < = expression when condition else
          expression when condition else
          expression;
```

Example:

```
Z  <= '1' when (L = '0' AND M = '0') else
          '1' when (L = '1' AND M = '1') else
          '0';
```

iii. **Selective Signal Assignment**

Syntax:

```
with choose_expression select
        target  <=  expression when choices,
                    expression when choices;
```

Example:

```
with SEL select
     M_OUT <= A3 when "11",
              A2 when "10",
              A1 when "01",
              A0 when "00",
              '0' when others;
```

iv. **Process Statement**

Syntax:

```
label: process(sensitivity list)
begin
        sequential statements
end process label;
```

2. Sequential Statements

i. **Signal Assignment**

Syntax:

```
target <= expression;
```

Example:

```
A <= B NAND C;
X <= (D OR E) AND (F AND G);
```

ii. **IF statement** (only works inside process)

Syntax:

```
if (condition) then
        { sequence of statements }
elsif (condition) then
        { sequence of statements }
else
        { sequence of statements }
end if;
```

Example:

```
if (SEL = "111") then
        F_OUT <= D(7);
elsif (SEL = "110") then
        F_OUT <= D(6);
elsif (SEL = "101") then
        F_OUT <= D(1);
else
        F_OUT <= '0';
end if;
```

> Here D is of 8 bits. And D(7) represents the most significant bit. So if we use D only then it represents 8 bit data whereas if we use D(n) then it represents any specific bit (n^th^) out of many bits.

iii. **CASE statement**

Syntax:

```
case (expression) is
        when choices =>
                sequential statements
        when choices =>
                sequential statements
        when others =>  -- (optional)
                sequential statements

end case;
```

Example:

```
case (ABC) is
        when "100" =>
```

```
                    F_OUT <= '1';
            when "011" =>
                    F_OUT <= '1';
            when "111" =>
                    F_OUT <= '1';
            when others =>
                    F_OUT <= '0';
    end case;
```

10.5 Standard Architectures

1. Dataflow Style Architecture (uses boolean logic and concurrent signal assignments)

Dataflow style architecture specifies a circuit as a concurrent representation of the flow of data through the circuit. In this modeling, the internal working of a system is implemented using concurrent statements. It can be used for small and primitive circuits but not for complex designs. In this style of architecture, whenever there is a change in signal of right hand side, the expression is evaluated and assigned to left hand side.

Example:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY HALF_ADDER IS
PORT(
        A, B: IN STD_LOGIC;
        S, C: OUT STD_LOGIC
    );
END HALF_ADDER;

ARCHITECTURE HALF_ADDER_ARCH OF HALF_ADDER IS
BEGIN
    S <= A XOR B;
    C <= A AND B;
END HALF_ADDER_ARCH;
```

2. Behavior Style Architecture (uses process statements)

The behavioral style architecture models how the circuit outputs will behave to the circuit inputs. This model may not reflect how the circuit is implemented when it is synthesized. Process statement is the core part of behavioral style architecture. In this style, the internal working is implemented using sequential statements within process statements.

Example:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY HALF_ADDER IS
PORT(
        A, B: IN STD_LOGIC;
        S, C: OUT STD_LOGIC
    );
END HALF_ADDER;

ARCHITECTURE HALF_ADDER_ARCH OF HALF_ADDER IS
BEGIN
    PROCESS_ADDER: PROCESS (A, B)
    BEGIN
        S <= A XOR B;
        C <= A AND B;
    END PROCESS PROCESS_ADDER;
END HALF_ADDER_ARCH;
```

A,B is the sensiti list of the process whenever there is change in any of thei value : the process gets executed

Process. (order of execution is sequential inside a process)

3. Structural Style Architecture (uses component instantiation)

The structural style architecture is a modular approach to coding which supports hierarchical design which is essential to understand complex digital designs. Modular designs enhance understandability by combining low-level functionality into modules. These modules can be reused in different designs resulting in save of design time. VHDL structural model may not be efficient for simple designs. However, the following are the general steps for writing structural model code.

- Initially the entity and architecture implementations for the individual gates or modules which are within our system must be defined.
- The entity declaration of our system is done, similar to other models.
- Different components used in our design are declared within the declarative part of architecture. Component declaration is similar to entity declaration, only keyword entity must be replaced by keyword component.

  Syntax of Component Declaration

  ```
  COMPONENT COMPONENT_NAME [IS]
  PORT(
              PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
              PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
              ...
      );
  END COMPONENT [COMPONENT_NAME];
  ```

- Internal signals, which are the intermediate output signals of one module fed into another module as input signals, are declared.
- Finally, instances of all modules are created and mapped in the architecture body. Mapping can be done using direct mapping or implied mapping. In direct mapping, each of the internal signals and signals of entity of the system are directly associated with the signals of corresponding components. Whereas in implied mapping, only internal signals and signals of entity of the system are listed. Though it uses less space, but it requires the signals be placed in the proper order.

Example: To implement Z = (A AND B) OR (C AND D) using structural model

(and-gate.vhd)

```
library ieee;
use ieee.std_logic_1164.all;

entity and_gate is
  port(
        x, y: in std_logic;
        w: out std_logic
      );
end and_gate;
architecture and_ah of and_gate is
begin
      process(x, y)
      begin
        w <= x and y;
      end process;
end and_ah;
```

(behavioral architecture)

(or-gate.vhd)

```
library ieee;
use ieee.std_logic_1164.all;

entity or_gate is
  port(
        x, y: in std_logic;
        w: out std_logic
      );
end or_gate;
architecture or_arch of or_gate is
begin
      process(x, y)
      begin
        w <= x or y;
      end process;
end or_arch;
```

(behavioral architecture)

(structural architecture)

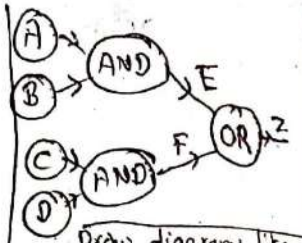
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY TEST IS
  PORT(
          A, B, C, D: IN STD_LOGIC;
          Z: OUT STD_LOGIC
      );
END TEST;

ARCHITECTURE TEST_ARCH OF TEST IS
  COMPONENT AND_GATE IS
    PORT (
                  X, Y: IN STD_LOGIC;
```

Scanned with CamScanner

```
                W: OUT STD_LOGIC
        );
    END COMPONENT;
    COMPONENT OR_GATE
        PORT (
                X, Y: IN STD_LOGIC;
                W: OUT STD_LOGIC
        );
    END COMPONENT;
Don't forget to declare this → SIGNAL E, F: STD_LOGIC;
    BEGIN
        U1: AND_GATE PORT MAP (X => A, Y => B, W => E);
        U2: AND_GATE PORT MAP (X => C, Y => D, W => F);
        U3: OR_GATE PORT MAP (X => E, Y => F, W => Z);
    END TEST;
```



Draw diagrams like those before implementing structural model. May not seem necessary here, but complex ckts will be easier if implemented this way.

10.6 FSM Design

Finite State Machines (FSM) constitute a special modeling technique for sequential logic circuits. The digital systems, in general, can be expressed as a sequence of actions which can be realized using FSM.

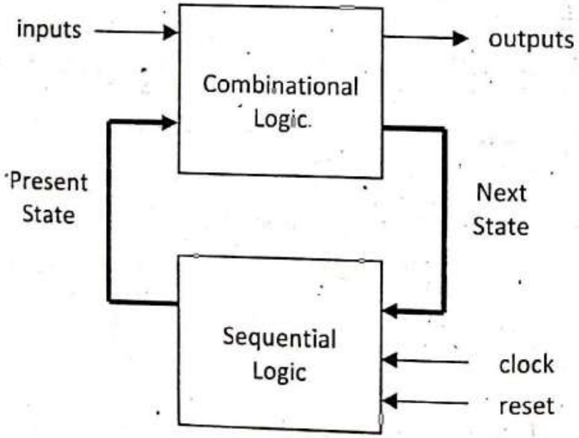


Figure 10.1: General block diagram of Finite State Machine

A FSM is specified by five entities: symbolic states, input signals, output signals, next-state function and output function. A state specifies a unique internal condition of a system and the FSM transits from one state to another with time. The next-state function is used to determine the next state of the system. The output function specifies the value of the output signals. The general block diagram of FSM is shown in the figure 10.1.

FSM consists of two sections; combinational and sequential logic. The combinational part has two inputs – external input and present state – and two outputs; next state and external output. Whereas, the sequential section has three inputs – clock, reset, and next state – and one output in a form of present state. Since the flip flips are implemented in sequential logic, clock and reset are part of this section.

If the output of the machine depends not only on the present state but also on the current input, then it is called a Mealy machine. Otherwise, if it depends only on the current state, it is called a Moore machine.

1. Design of Sequential Section

PROCESS statement is required for sequential section. The clock and reset signals appear in the sensitivity list of PROCESS statement. When reset is asserted, present state will be set to initial state of the system. In other cases, present state will change to next state at the proper clock edge. A typical design template for the sequential section is given as:

```
PROCESS (reset, clock)
BEGIN
    IF(reset = '1') THEN
        present_state <= Initial_state;
    ELSIF (clock'event and clock = '1') THEN
        present_state <= Next_state;
    END IF;
END PROCESS;
```

2. Design of Combinational Section

In this section, the code does not need to be sequential; concurrent code can be used. If sequential is implemented then the input and present state will be the part of sensitivity list of PROCESS statement. Within the PROCESS statement, CASE statement is used to implement the actions and conditions of each state. A basic template for the combinational section is shown as:

```
PROCESS (input, present_state)
BEGIN
    CASE present_STATE IS
        WHEN STATE0 =>-- within when structure of case,
              ...       -- may contain actions and conditions
        WHEN STATE1 =>
              ...       -- number of when structure is
        WHEN STATE2 =>-- defined by number of states in FSM
              ...
        When OTHERS =>
              ...
    END CASE;
END PROCESS;
```

SOLUTION TO IMPORTANT QUESTIONS

Problem 1: Simple NAND Gate with two inputs, each input of single bit

Solution:

```
LIBRARY IEEE
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY NAND_GATE IS
PORT (
        IN_A, IN_B: IN STD_LOGIC;
        X_OUT: OUT STD_LOGIC
    );
END NAND_GATE;
ARCHITECTURE nand_arch OF nand_gate IS
BEGIN
    proc: PROCESS (in_a, in_b)
    BEGIN
        x_out <= in_a NAND in_b;
    END PROCESS proc;
END nand_arch;
```

VHDL is case insensitive language.

The symbol <= is used as assignment operator.

Problem 2:

Write a VHDL code to implement 4 X 1 MUX with each input of 3 bits.

Solution:

```
LIBRARY IEEE
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux_4x1 IS
  PORT(
        in_a, in_b : IN STD_LOGIC_VECTOR (2 downto 0);
        in_c, in_d : IN STD_LOGIC_VECTOR (2 downto 0);
        SEL :  IN STD_LOGIC _VECTOR(1 downto 0);
        z_out : OUT STD_LOGIC_VECTOR (2 downto 0)
    );
END mux_4x1;
ARCHITECTURE mux_arch OF mux_4x1 IS
BEGIN
    proc: PROCESS (in_a, in_b, in_c, in_d, SEL)
    BEGIN
        IF (SEL = "00") THEN
                z_out <= in_a;
        ELSIF (SEL = "01") THEN
                z_out <= in_b;
        ELSIF (SEL = "01") THEN
                z_out <= in_c;
        ELSE
                z_out <= in_d;
        END IF;
    END PROCESS proc;
  END mux_arch;
```

For signals with more than one bit, we need to use std_logic_vector(n-1 downto 0) where n is number of bits.

For signals requiring more than single bit, we need to use double quotation mark for bit values. For example: "01", "1010"

Problem 3: Write a VHDL code to implement D flip-flop.

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Scanned with CamScanner

```
entity DFlipflop is
Port (
            D, clk  :  in  STD_LOGIC;
            Q  :  out  STD_LOGIC
   );
end DFlipflop;
```

CLK'event and CLK = '1' represents rising edge of the clock pulse.

```
architecture Behavioral of DFlipflop is
begin
  process (D, CLK)
  begin
            if (CLK'event and CLK = '1') then
                    Q <= D;
            end if;
  end process;
end Behavioral;
```

Problem 4: Implement a counter that counts from 0 to 9 using VHDL code

Solution:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Counter_Code is
  port(
            CLR : in  std_logic;
            CLK : in  std_logic;
            Q : out  std_logic_vector (3 downto 0)
      );
end Counter_Code;

architecture Behavioral of Counter_Code is
      signal tmp: std_logic_vector (3 downto 0);
```

```
begin
     process (CLK, CLR)
     begin
          if (CLK'event and CLK = '0') then
               if (CLR = '1') then
                    tmp <= "0000";
               else
                    tmp <= tmp + 1;
               end if;
          end if;
     end process;
     Q <= tmp;
end Behavioral;
```

CLK'event and CLK = '0' represents falling edge of the clock pulse.

Problem 5: Write a VHDL code to detect a sequence of "1001"

Solution:

The FSM for the detection of the sequence is given by following diagram.

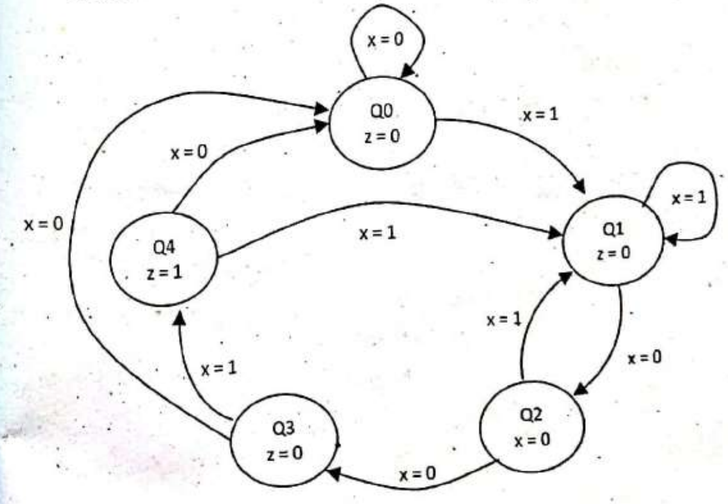


Figure 10.2: FSM for detection of sequence "1001"

VHDL Code

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
entity Sequence_State is
  Port (
          x : in  STD_LOGIC;
          clk, RESET : in  STD_LOGIC;
          z : out  STD_LOGIC
        );
end Sequence_State;

architecture Behavioral of Sequence_State is
  type state is (Q0, Q1, Q2, Q3, Q4);
  signal PS, NS: state;
begin
  sync_proc: process (clk, reset)
  begin
    if (reset = '1') then
       PS <= Q0;
    elsif (rising_edge(clk)) then
       PS <= NS;
    end if;
  end process sync_proc;

  comb_proc: process(PS, x)
  begin
  case PS is
       when Q0 => z <= '0';
            if(x = '1') then
                 NS <= Q1;
            else
                 NS <= Q0;
            end if;
       when Q1 => z <= '0';
            if(x = '0') then
                 NS <= Q2;
            else
                 NS <= Q1;
            end if;
       when Q2 => z <= '0';
            if(x = '0') then
                 NS <= Q3;
            else
                 NS <= Q1;
            end if;
       when Q3 => z <= '0';
            if(x = '1') then
                 NS <= Q4;
            else
                 NS <= Q0;
            end if;
       when Q4 => z <= '1';
            if(x = '1') then
                 NS <= Q1;
            else
                 NS <= Q0;
            end if;
       when others =>
            NS <= Q0;
   end case;
 end process comb_proc;
end behavioral;
```

Problem 6: Calculate the GCD of two numbers using VHDL

Solution:

Functionality code to calculate the GCD of two numbers is given as

```
int X, Y;
while(1)
{
    while(!GO);
```

Scanned with CamScanner

```
    X = NUM1;
    Y = NUM2;
    while(X != Y)
    {
        if(X<Y)
                Y = Y – X;
        else
                X = X – Y;
    }
    GCD = X;
}
```

The FSM for the above code can be represented by following diagram

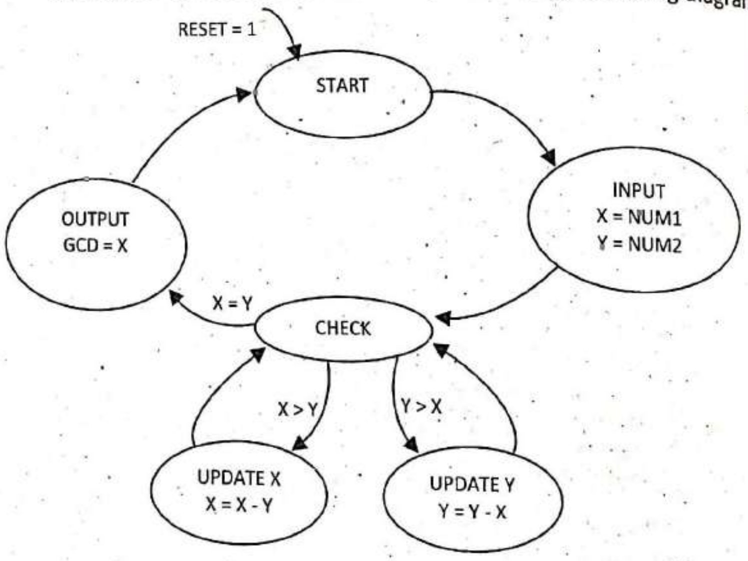


Figure 10.2: FSM for GCD processor

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FSM_GCD is
    port (
                RESET, CLK: in std_logic;
                GO: in std_logic;
```

```
                NUM1, NUM2: in integer;
                GCD: out integer
        );
end FSM_GCD;

architecture Behavioral of FSM_GCD is
type state is (start, input, check, updatex, updatey, output);
  signal PS, NS: state;
begin
  seq_proc: process (CLK, GO, RESET)
  begin
      if (GO = '1') then
            if (RESET = '1') then
                      PS <= start;
            elsif (rising_edge(CLK)) then
                      PS <= NS;
            end if;
      end if;
   end process seq_proc;

   comb_proc: process (NUM1, NUM2, PS)
       variable X, Y: integer;
   begin
       case PS is
            when START =>
                      GCD <= 0;
                      NS <= INPUT;
            when INPUT =>
                      X := NUM1;
                      Y := NUM2;
                      NS <= CHECK;
            when CHECK =>
                      if (X > Y) then
                            NS <= UPDATEX;
```

For variables, the assignment operator is colon followed by equal symbol. (:=)

For signals, the assignment operator is less than equal symbol. (<=)

Scanned with CamScanner

```
            elsif(X < Y) then
                    NS <= UPDATEY;
            else
                    NS <= OUTPUT;
            end if;
        when UPDATEX =>
                X := X – Y;
                ns <= CHECK;
        when UPDATEY =>
                Y := Y – X;
                ns <= CHECK;
        when OUTPUT =>
                GCD <= X;
                NS <= INPUT;
        when OTHERS =>
                GCD <= 0;
                NS <= INPUT;
        end case;
    end process comb_proc;
end behavioral;
```

Problem 7: Design and write the code for Decoder using VHDL. **[2076 Baisakh]**

Solution:

VHDL code for 2 to 4 decoder

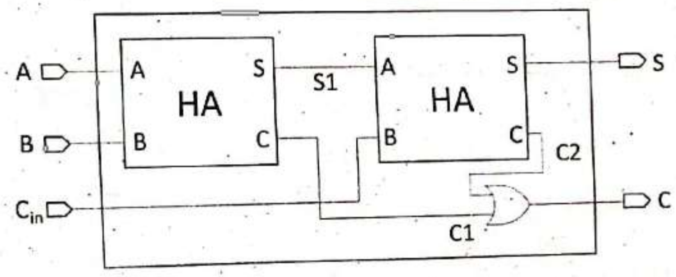
```
library ieee;
use ieee.std_logic_1164.all;
entity decoder is
    port(
            a : in std_logic_vector (1 downto 0);
            b : out std_logic_vector (3 downto 0)
        );
end decoder;
```

```
architecture decoder_arch of decoder is
begin
    process(a)
    begin
        case a is
                when "00" => b <= "0001";
                when "01" => b <= "0010";
                when "10" => b <= "0100";
                when "11" => b <= "1000";
                when others => b <= "0000";
        end case;
    end process;
end decoder_arch;
```

Problem 8:

Write VHDL code for a full adder using two half adder and one or gates. **[2075 Bhadra]**

Solution:



VHDL coding for or gates:

```
Library ieee;
Use ieee.std`logic_1164.all;
Entity or_gate is
Port (
            a, b : in std_logic;
            z : out std_logic
    );
End or_gate;
```

Scanned with CamScanner

```
Architecture or_arch is
Begin
            z <= a or b;
End or_arch;
```

VHDL code for half adder:

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity half_adder is
  port(
            a, b : in std_logic;
            sum, carry_out : out std_logic
    );
End half_adder;

Architecture half_adder_arch of half_adder is
Begin
   process_adder: process (a, b);
   begin
            sum <= a xor b;
            carry_out <= a and b;
   end process process_adder;
End half_adder_arch;
```

VHDL code for full adder using two half adder and or gate:

```
Library ieee;
use ieee.std_logic_1164.all;

entity fulladd is
   port(
            a : in  std_logic;
            b : in  std_logic;
            cin : in  std_logic;
            s : out  std_logic;
            c : out  std_logic
         );
end fulladd;

Architecture structural of fulladd is
   Component or_gate is
   port(
            a, b : in std_logic;
            z : out std_logic
         );
   End component or_gate;

   Component half_adder is
   Port (
            a, b : in std_logic;
            sum, carry_out : out std_logic
         );
   end component half_adder;
   Signal s1, c2, c1: std_logic;
Begin
            HA1: half_adder port map(a => a, b => b,
                              sum => s1, carry_out => c1);
            HA2: half_adder port map(a => s1, b => cin,
                              sum => sum, carry_out => c2);
            OR1: or_gate port map(a => c1, b => c2, z => c);
End architecture structural;
```

Problem 9: Write the VHDL coding for a JK flip-flop using process. **[2075 Baisakh]**

Solution:

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;
```

Scanned with CamScanner

```
Use ieee.std_logic_unsigned.all;

Entity jk_ff is
    port (
                j, k, clock : in std_logic;
                q, qb : out std_logic
    );
End jk_ff;

Architecture behavioral of jk_ff is
Begin
    process (clock, j, k)
    variable tmp: std_logic;
    begin
        if (clock = '1' and clock'event) then
            if (j = '0' and k = '0') then
                    tmp := tmp;
            elsif(j = '1' and k = '1') then
                    tmp := not tmp;
            elsif (j = '0' and k = '1') then
                    tmp := '0';
            else
                    tmp := '1';
            end if;
        end if;
        q <= tmp;
        qb <= not tmp;
    end process;
End behavioral;
```

Problem 10:

Write an algorithm and VHDL for a custom processor that can calculates Least Common Divisor (LCM) of two numbers as a finite state machine. **[2074 Bhadra]**

Solution:

Functionality code to calculate the LCM of two numbers is given as

```
int x, y, z, GCD;
while(1)
{
while(reset);
x = num1;
y = num2;
z = x * y;
while(x != y)
{
        if(x < y)
                y = y – x;
        else
                x = x – y;
}
GCD = x;
LCM = z/GCD;
}
```

The FSM for the above code can be represented by following diagram

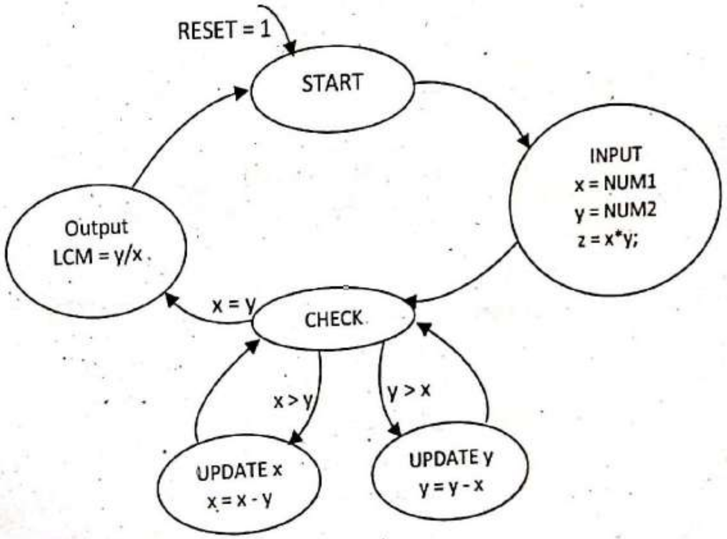


Figure: FSM for LCM processor

VHDL CODE

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;
```

```
Entity fsm_lcm is
Port (
        reset, clk: in std_logic;
        num1, num2: in integer;
        LCM: out integer
    );
End fsm_lcm;
Architecture behavioral of fsm_lcm is
  Type state is (START, INPUT, CHECK, UPDATEX, UPDATEY, OUTPUT);
  Signal ps, ns: state;
Begin
        Seq_proc: process (clk, reset)
        Begin
                if (reset = '1') then
                        ps <= start;
                elsif (rising_edge(clk)) then
                        ps <= ns;
                end if;
        End process seq_proc;

        Comb_proc: process (num1, num2, ps)
            Variable x, y, z, GCD: integer;
        Begin
            case ps is
                when START =>
                        LCM <= 0;
                        ns <= INPUT;
                when INPUT =>
                        x := num1;
                        y := num2;
                        z := x*y;
                        ns <= CHECK;
                when CHECK =>
                        if(x > y) then
                                ns <= UPDATEX;
                        elsif(x < y) then
```

```
                                ns <= UPDATEY;
                        else
                                ns <= OUTPUT;
                        end if;
                when UPDATEX =>
                        x := x – y;
                        ns <= CHECK;
                when UPDATEY =>
                        y := y – x;
                        ns <= CHECK;
                when OUTPUT =>
                        GCD := x;
                        LCM <= z/GCD;
                        ns <= INPUT;
                when OTHERS =>
                        LCM <= 0;
                        ns <= INPUT;
            end case;
        end process comb_proc;
End behavioral;
```

Problem 11: Write a VHDL code for 2 - bit input multiplexer. [2072 Magh]

Solution:

```
    library ieee
    use ieee.std_logic_1164.all;
    entity mux_4x1 is
        port (
                in_a, in_b : in std_logic_vector (1 downto 0);
                in_c, in_d : in std_logic_vector (1 downto 0);
                sel :  in std_logic _vector(1 downto 0);
                z_out : out std_logic_vector (1 downto 0)
            );
    end mux_4x1;
    architecture mux_arch of mux_4x1 is
    begin
      proc: process (in_a, in_b, in_c, in_d, sel)
```

Scanned with CamScanner

```
    begin
        if (sel = "00") then
                Z_out <= in_a;
        elsif (sel = "01") then
                Z_out <= in_b;
        elsif (sel = "01") then
                Z_out <= in_c;
        else
                Z_out <= in_d;
        end if;
  end process proc;
end mux_arch;
```

REFERENCES

Frank Vahid, Tony Givargis. *Embedded System Design: A Unified Hardware/Software Approach*. New Jersey: John Wiley & Sons, Inc., 2002.

Shibu K V. *Introduction to Embedded Systems*. 2nd ed. McGraw Hill Education (India) Private Limited, 2009.

Muhammad Ali Mazidi, Janice Gillispie Mazidi, Rolin D. McKinley. *The 8051 Microcontroller and Embedded Systems*. 2nd ed. New Jersey: Pearson Education, Inc., 2011.

Peter J. Ashenden. *The VHDL Cookbook*. 1st ed. Peter J. Ashenden, 1998.

Douglas L. Perry. *VHDL Programming by Examples*. 4th ed. New York: The McGraw-Hill Companies, Inc., 2002.

M. Morris Mano. *Computer System Architecture*. 3rd ed. New Jersey: Pearson Education, Inc., 2007.

William Stallings. *Computer Organization and Architecture*. 10th ed. New Jersey: Pearson Education, Inc., 2016.

William Stallings. *Operating Systems*. 6th ed. New Jersey: Pearson Education, Inc., 2008.

Scanned with CamScanner