G.L. Book House

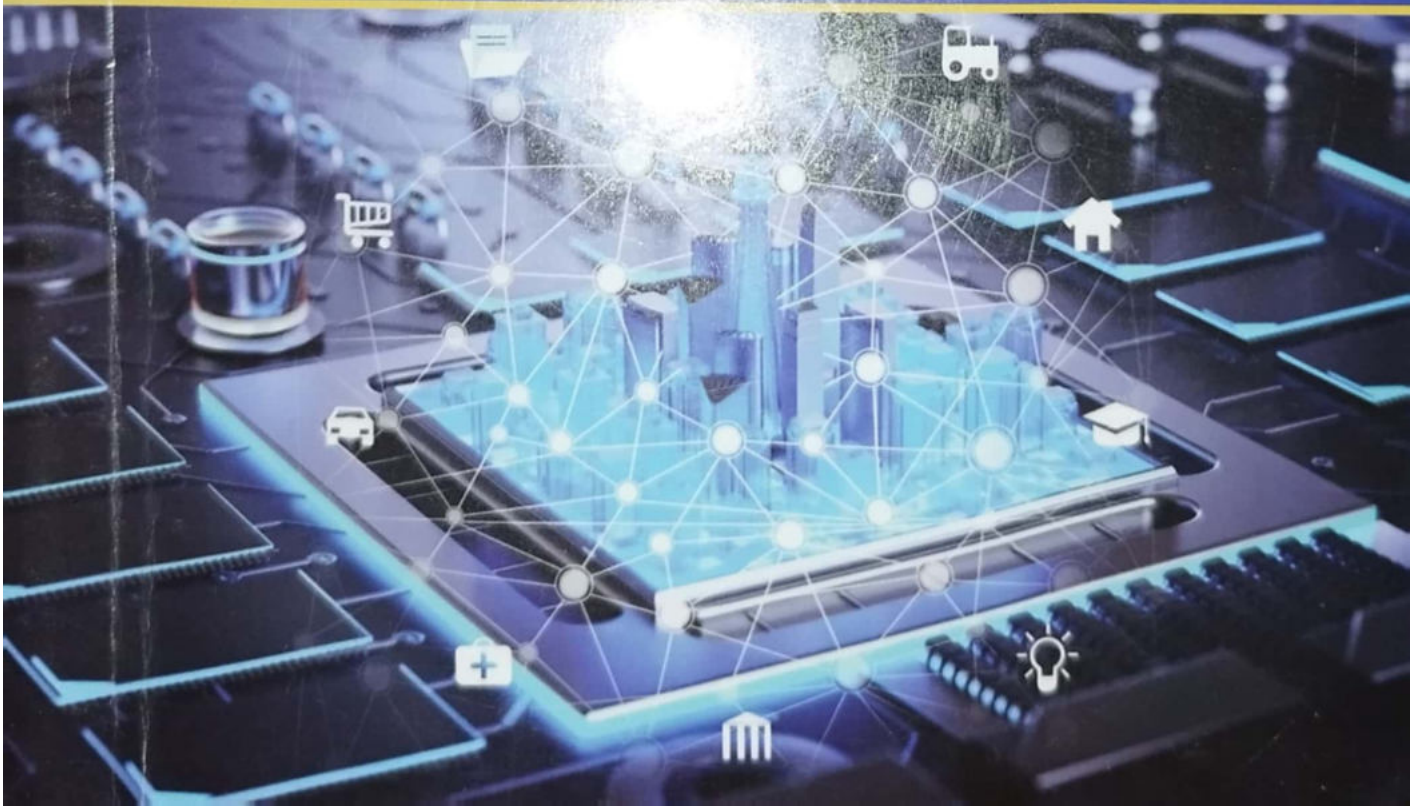
a complete manual

EMBEDDED SYSTEM

BACHELOR OF ENGINEERING

Computer Engineering
Electrical Engineering
Electronics and Communication Engineering

Er. PRAKRITI GYAWALI



Prapthulla Giri
020-326.

G.L. BOOK HOUSE

a complete manual

EMBEDDED SYSTEM

BACHELOR OF ENGINEERING

Computer Engineering

Electrical Engineering

Electronics and Communication Engineering

PRAKRITI GYAWALI

Publisher

G.L. Book House Pvt. Ltd.

Infront of Thapathali Engineering Campus, Maitighar, Kathmandu

Embedded System

Bachelor of Engineering

Author

Prakriti Gyawali

Publisher

G.L. Book House Pvt. Ltd.

Maitighar, Kathmandu

Distributor

Laxmi Pustak Bhandar

Maitighar, Kathmandu

9841-250324

Printed in Nepal

First Edition : 2022/023

Price : Rs. 375.00

Layout : Pratap Acharya

Graphics : Sanyr Prajapati

Preface

One of the key courses enlisted in Bachelors of Computer Engineering, Electrical Engineering and Electronics & Communication Engineering in all the Universities of Nepal is **'Embedded System'** as known in all universities of Nepal.

This book will assist computer, electronics, and electrical engineering students in learning the fundamentals to advanced ideas of embedded systems, hence a thorough knowledge of it is crucial. **"A Complete Manual of Embedded System"** is a joint, humble effort to help the future Engineers for a better and easy comprehension of concepts and ideas.

It is an excellent reference for a brief summary (notes) and constitutes sample of numerical (with their solution) which can eventually lead learners to score better. Furthermore, it incorporates plenty of TU and PoU exam questions with their solution. Every effort has been made to keep the volume accessible and informative.

I want to express my gratitude to my parents for their kind support. This book is dedicated to my parents, who have supported throughout my carrier and to professors, and seniors for encouraging throughout the process. I would like to express heartfelt gratitude to Er. Gaurav Panth for his love and support, as well as G.L. publications and Laxmi Man Shrestha for providing with a wonderful platform to publish material through this book, Sanyr Prajapati and Pratap Acharya for their editing effort. I want to express my gratitude to Prajesh Gyawali, Prajeeta Gyawali for their encouragement and support throughout the process.

Comments are always welcome and will be incorporated in the succeeding editions.

Author
Prakriti Gyawali

Embedded System Syllabus of Tribhuvan University (IOE)

CT 655

Lecture : 3
Tutorial : 1
Practical : 1.5

Year : III
Part : II

Course Objectives:

To introduce students to understand and familiarization on applied computing principles in emerging technologies and applications for embedded systems

Course Content:

1. **Introduction to Embedded System** [3 Hours]
   - 1.1 Embedded Systems overview
   - 1.2 Classification of Embedded Systems
   - 1.3 Hardware and Software in a system
   - 1.4 Purpose and Application of Embedded Systems
2. **Hardware Design Issues** [4 Hours]
   - 2.1 Combination Logic
   - 2.2 Sequential Logic
   - 2.3 Custom Single-Purpose Processor Design
   - 2.4 Optimizing Custom Single-Purpose Processors
3. **Software Design Issues** [6 Hours]
   - 3.1 Basic Architecture
   - 3.2 Operation
   - 3.3 Programmer's View
   - 3.4 Development Environment
   - 3.5 Application-Specific Instruction-Set Processors
   - 3.6 Selecting a Microprocessor
   - 3.7 General-Purpose Processor Design
4. **Memory** [5 Hours]
   - 4.1 Memory Write Ability and Storage Permanence
   - 4.2 Types of Memory
   - 4.3 Composing Memory
   - 4.4 Memory Hierarchy and Cache
5. **Interfacing** [6 Hours]
   - 5.1 Communication Basics
   - 5.2 Microprocessor Interfacing: I/O Addressing, Interrupts, DMA
   - 5.3 Arbitration
   - 5.4 Multilevel Bus Architectures
   - 5.5 Advanced Communication Principles
6. **Real-Time Operating System (RTOS)** [8 Hours]
   - 6.1 Operating System Basics
   - 6.2 Task, Process, and Threads
   - 6.3 Multiprocessing and Multitasking

6.4 Task Scheduling
6.5 Task Synchronization
6.6 Device Drivers

7. Control System [3 Hours]
   7.1 Open-loop and Close-Loop control System overview
   7.2 Control System and PID Controllers
   7.3 Software coding of a PID Controller
   7.4 PID Tuning

8. IC Technology [3 Hours]
   8.1 Full-Custom (VLSI) IC Technology
   8.2 Semi-Custom (ASIC) IC Technology
   8.3 Programming Logic Device (PLD) IC Technology

9. Microcontrollers in Embedded Systems [3 Hours]
   9.1 Intel 8051 microcontroller family, its architecture and instruction sets
   9.2 Programming in Assembly Language
   9.3 A simple interfacing example with 7 segment display

10. VHDL [4 Hours]
   10.1 VHDL overview
   10.2 Finite state machine design with VHDL

Practical:

Student should be complete project work related to this subject.

Reference Books:

1. David E. Simon, "An Embedded Software Primer", Addison-Wesley, 2005
2. Muhammad Ali Mazidi, "8051 Microcontroller and Embedded Systems", Prentice Hall, 2006
3. Frank Vahid, Tony Givargis, "Embedded System Design", John Wiley & Sons, 2008
4. Douglas L. Perry, "VHDL Programming by example", McGraw Hill, 2002

Evaluation Scheme:

The question will cover all the chapters of the syllabus. The evaluation scheme will be as indicated in the table below:

Unit	Hour	Mark Distribution*
1	3	4
2	4	8
3	6	8
4	5	8
5	6	8
6	8	12
7	3	8
8	3	8
9	3	8
10	4	8
Total	45	80

*There may be minor variation in marks distribution.

Embedded System (PoU)

(3-1-2)

Course Objectives:

1. To provide the students with the basic information about embedded systems.
2. To familiarize students to applied computing principles in emerging technologies and applications for embedded systems.

Course Content:

1. Introduction to Embedded Systems [3 Hours]
   1.1 General characteristics of embedded systems
   1.2 Classification of Embedded system
   1.3 Essential components
   1.4 Overview of processors and hardware units in an embedded system
   1.5 Application of embedded systems

2. Hardware and Software Design Issues [10 hrs]
   2.1 Hardware design issue
      2.1.1 Combinational and sequential logic
      2.1.2 Custom single-purpose processor design
      2.1.3 Optimizing custom single-purpose processors
   2.2 Software design issues
      2.2.1 Basic architecture
      2.2.2 Operation
      2.2.3 Programmers view
      2.2.4 Development environment
      2.2.5 Application-specific instruction-set processors
      2.2.6 Selecting a microprocessor
      2.2.7 General-purpose processor design

3. Memory [5 hrs]
   3.1 Memory write ability and storage permanence
   3.2 Types of memory
   3.3 Composing memory
   3.4 Memory hierarchy and cache

4. Interfacing [6 hrs]
   4.1 Communication basics
   4.2 Microprocessor interfacing: (/0 addressing, interrupts, OMA)
   4.3 Arbitration
   4.4 Multilevel bus architecture
   4.5 Advanced communication principles

5. Real Time Operating System (RTOS) [8 hrs]
   5.1. Definitions of process, tasks and threads
   5.2. The real-time kernel
   5.3. OS tasks, task states and task scheduling
   5.4 Interrupt processing
   5.5 Clocking communication and task synchronization

5.6 Control blocks
5.7 Memory requirements and control kernel services

6. Embedded Software Development Tools [2 hrs]
   6.1 Cross assemblers
   6.2 Cross compilers
   6.3 Debuggers
   6.4 Downloader

7. Microcontrollers [3 hrs]
   7.1 Intel 8051 microcontroller family
      7.1.1 Architecture and
      7.1.2 Instruction sets
   7.2 Programming in assembly language
   7.3 A simple interfacing example with 7 segment display

8. VHDL [8 hrs]
   8.1 Background and basic concepts
   8.2 Structural specification of hardware and design organization
   8.3 VHDL realization of basic digital circuits
      8.3.1 Binary adder
      8.3.2 Decoder
      8.3.4 Multiplier
      8.3.5 Counters
      8.3.6 Shift registers
      8.3.7 Sequence detectors

Laboratory:

1. Simulation of various digital circuits using VHDL
2. Student should complete one project work related to this subject

Text Books:

1. David E, Simon, 'An Embedded Software Primer," Addison Wesley, Latest Edition.
2. Muhammad Ali Mazid I, "8051 Microcontroller and Embedded Systems", Prentice Hall, Latest Edition.
3. Frank Vah id, Tony Givargis, "Embedded system Design," John Wiley and Sons, Latest Edition.

Contents

Chapter 1

INTRODUCTION TO EMBEDDED SYSTEM

★★★

★★★

1.1 EMBEDDED SYSTEMS OVERVIEW

As its name suggests, embedded means something that is attached to another thing. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. An embedded system is a system that has embedded software and computer hardware which makes it a system dedicated for an application are specific part of an application or a product of a large system.

Computing systems are everywhere. It is probably no surprise that millions of computing system are built every year, destined for desktop computers like personal computer, laptops, workstations, mainframes, and servers. Embedded systems are found in a variety of common electronic devices, such as consumer electronics (cell phones, pagers, digital cameras), home appliances (microwave ovens, answering machines, thermostats, home security systems, washing machines, and lightning system), office automation (fax machines, copiers, printers, and scanners), business equipment (cash registers, curbside check-in, alarm system, card readers, product scanners, and automated teller machines), and automobiles (transmission control, cruise control, fuel injection, antilock brakes, and active suspension).

1.1.1 Features of an Embedded System

There are several features are common to many embedded system where as it is also not mandatory that all the features will be supported by all the embedded system. Following are some of the important features exhibited by the system:

i) Single functioned system

Most Embedded system performs a single job repeatedly. Example: A washing machine has an embedded controller that can take user input and perform the job of washing continuously. However all the embedded systems are not single functioned.

ii) Interaction with the physical environment

Most embedded system interacts with the physical environment around them. Data are collected from the environment using sensors by taking some of the parameters of the environment.

iii) User Interface

Unlike the common user interfaces like keyboard, mouse touch screen, embedded system contains dedicated user interfaces like specially designed buttons, LEDs, stearing wheel etc.

iv) Dependable system

Embedded system are often used in critical applicant's like nuclear plants, instrumentation etc. hence this dependent demands a high degree of dependability on such system.

v) Tightly constrained system

All computing system have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded system often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

vi) Reactive and real time

Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.

vii) Microprocessors based

It must be microprocessor or microcontroller based.

viii) Memory

It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.

ix) Connected

It must have connected pheripherials to connected input and output devices.

x) Hardware-software system

Software is used for more features and flexibility. Hardware is used for performance and security.

1.2 CLASSIFICATION OF EMBEDDED SYSTEMS

An embedded system is an electric system that has a software and is embedded in computer hardware. Embedded systems can be classified into different types based on performance, functional requirements and performance of the microcontroller.

Embedded systems are classified into four categories based on their performance and functional requirements.

i) Stands alone embedded systems
ii) Real time embedded systems
iii) Networked embedded systems
iv) Mobile embedded systems

Embedded systems are classified into three types based on the performance of the microcontroller such as:

i) Small scale embedded systems
ii) Medium scale embedded systems
iii) Sophisticated embedded systems

a) Small scale embedded systems

Normally small scale embedded system is designed by using an 8 bit microcontroller that may even be activated by a battery. For developing

embedded software for such system, an editor, assembler or cross assembler are used for specific microcontroller or processor used in the system.

b) Medium Scale Embedded Systems

The medium scale embedded system are designed using single or multiple 16 bit or 32 bit microcontroller or digital signal processor (DSP's) or reduced instruction set computer (RISC's). These types of embedded systems have both hardware and software complexities. The development tools like real time operating system (RTOS), source code engineering tools, simulator, debugger and integrated development tools are required for such complex software design system. These software tools also provide the solution for the hardware complexities, so assembler is used very rarely.

c) Sophisticated Embedded Systems

Sophisticated embedded systems consist of large quantity of hardware and software complexities hence they may require scalable processors or configurable processors and programmable logic arrays (PLA's). They are used for cutting-edge applications that need hardware and software co-design and components which have to assemble in the final system.

1.3 HARDWARE AND SOFTWARE IN A SYSTEM

The embedded systems basics include the components of embedded system hardware, embedded system types and several characteristics.

The following are some important components of embedded system.

i) Microprocessor

This is the heart of any embedded system. The microprocessors used here are different from general purpose microprocessor. *Examples*: SPARC, Pentium. They are designed to meet some specific requirements. *Examples*: INTEL 8048 is a special purpose microprocessor which we will find in the keyboards of desktop computers and it is used to scan the key strokes and send them in a synchronous manner to the personal computer.

Similarly mobile phones, digital cameras use special purpose processors for voice and image processing.

II) Memory

The microprocessor and memory must co-exist on the same PCB (Power Circuit Board) or same chip. Compactness, speed and low power consumptions are the characteristics required for the memory to be used in embedded system. The memory should be non- volatile and should be easily programmable.

III) I/O devices and Interfaces

Input and output interfaces are necessary to make the embedded system to interact with the external world. They would be visual display units, touch pad key, antenna, microphones, sensors etc.

The embedded system should also have an open interface to other devices such as desktop computers, LANs and other embedded systems.

IV) Software

The embedded system is just a physical body as long as it is not programmed. Whenever you switch on your mobile phones or whenever you are moving away from your own destination to another you might have marked some activities like no signal sign and some messages etc.

These activities are taken care of by the RTOS stored in the non volatile memory of the embedded system.

1.4 DESIGN ISSUES

The design of a real time embedded system have number of constrains are imposed by external as well as internal specification. Design metrics are introduced to measure the cost function by taking into account the technical as well as economic considerations.

1.4.1 Common Design Metrics

A design metric is a measurable feature of a system implementation. Commonly used metrics include:

i) NRE cost

It is one time monetary cost of designing of the system. Once the system is designed any number of units can be manufactured without having any additional cost, hence it is non-recurring.

ii) Size

Size of the system is very important. It refers to the code size for the software portion of the embedded system. The code size affects the memory space requirement and thereby increasing the overall chip or board size.

iii) Power requirement

This is the other most important design matric particularly because the embedded systems are expected to have light weight and long battery life. Hence the amount of power consumed by the system may determine the lifetime of the battery.

iv) Design flexibility

It refers to the effort needed to modify a system if the specifications change to some extend later. Software are typically considered very flexible and the main problem in the design change is the repetition of NRE cost.

v) **Time to prototype**
Time needed to build a working version of the system should be very less and it can be used to verify the system's usefulness and correctness.

vi) **Design turnaround time**
This is the time needed to complete the design starting from specification up to taking it to the market. Due to the very high rate of absolution of electronic goods it is usual that this time be very small.

vii) **Maintainability**
It is the ability to modify the system after its initial release especially to those who didn't originally design the system.

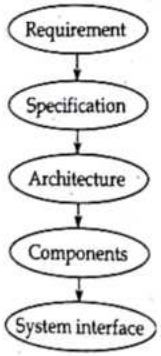
viii) **Performance**
It is the execution time of the system. Normally the specification of the system will have some performance requirements to be met by the design.

ix) **Correctness**
It should be ensure that the embedded system is manufactured correctly so that it can be implemented in critical applications with full confidence.

x) **Safety**
The probability that the system will not cause any harm.

1.4.2 Embedded System Design Process

The measure steps in the design process are explain in the top-down view as follows.



a) **Requirements**
- The requirement phase of the design what to design.
- Informal description gather from customer is known as requirement.
- Requirement can be of two types
  i) Functional requirement
  ii) Non-functional requirement

- Some of the non-functional requirements are:
  i) Power consumption
  ii) Physical size and weight
  iii) Performance
  iv) Cost

Requirement Validation:
- Requirement validation is a measure to examine whether the system is performing as per the specification or not.
- It can be done by using the technical skill and past experience of the customers.
- The requirements can be gathered in terms of a requirement from like a check list.

b) **Specification**
- It serves as the contract between the customer and the developer. It should be carefully written such that it accurately reflects the customer's requirements.

c) **Architectural Design**
- The plan used to design the components of a system is called as a architecture.
- It deals with how the system will perform the specified operations.
- The architecture is the plan for the overall structure of a system; that will be used to develop the components of the final system later.

d) **Hardware and Software Components**
- The architectural description tells us what the components we need. The component designers build the components according to the architecture's specification.
- Some of the components are readymade like CPU and some of are standard components like memory are initially developed. And some of the components are involved rigorous programming.

e) **System Integration**
- System Integration deals with the integration or assembly of all the components created.

1.5 PURPOSE AND APPLICATION OF EMBEDDED SYSTEM

An embedded system is the use of a computer system built into a machine of some sort, usually to provide a means of control. Embedded systems are every-where in our lives, from the TV remote control to the microwave, to control the central heating to the digital alarm clock next to our bed. They are in cars, washing machines, cameras, drones and toys.

An embedded system has a microprocessor in it which is essentially a complete computer system with limited, specific functionality. As far as user goes, they can usually interact with it through a limited interface. This typically will allow the user to input settings and make selections and also to receive output using text, video or audio signals.

1.5.1 Applications

i) Automation system

In modern era, we are all aware of automation systems. Whether you are at home or your office, if things are not automated then it can make your work quite difficult. We don't need to go far, let's take the example of remote control. It has made our life so easy now we can turn things on or off by simply clicking buttons on our remote control. Smart Home automation is a widely diverse application of embedded systems. From curtains to doors, lights or fans, everything is controlled by a simple remote for the ease of use.

ii) Security systems

Now coming towards security systems, these are all the fruits of embedded systems. These days we have sensors installed in our homes and even if you are not at home you can still get complete visual camera feeds on your mobile from anywhere in the world. That becomes possible just because of embedded system.

iii) Appliances

Most of our home appliances, like kitchen appliances which we normally called electronics products are actually embedded products. Microwave ovens, toasters, burners, in fact, even modern irons are also embedded products.

Gaming routers, photocopiers, media players and music system etc, they are all in our hands because of embedded systems. So, there's a lot of such real-life applications of embedded systems and without them, we can't even take a single steps.

SOME SOLVED QUESTIONS

1. What is an embedded system? Explain the classification of embedded system in detail.

Answer:

An embedded system is a microcontroller or microprocessor based system which is designed to performance a specific task.

Second part: See the topic 1.2 of chapter 1.

2. What are the applications of embedded system?

Answer: See the topic 1.5 of chapter 1.

3. Explain the components of embedded system hardware.

Answer:

As we know embedded systems are the combination of hardware and software. There are different hardware components like power supply, processor, memory, timers and counters that make the embedded hardware.

Power supply:

The power supply is an essential part of any embedded systems circuits. An embedded system may need a supply of 5 volts or if it is low power then may be 3.3 or 1.8 V. The supply may be provided with the help of battery or we can use any wall adapter. It depends on the application need.

The power supply circuit can be designed with the help some little knowledge of electronics. For that, we need a bridge rectifier circuit, capacitors as a filter and a voltage regulator that provides constant output supply.

Processor:

A processor is the main brain inside any embedded systems. This is a major factor that affects the performance of the system. There are different processors available in the market. An embedded system may use microprocessor or microcontroller.

The processor comes in differently architecture like 8-bit, 16-bit and 32-bit. The 8-bit processor is generally used in a small application where we need some basic computation like input and output no heavy processing.

Memory:

If we are using a microcontroller like AT89s51, AT89s52 or ATmega. The memory is available on-chip. We generally talk about two types of memory in the embedded systems.

- Read Only Memory (ROM)
- Random Access Memory (RAM)
- Electricity Erasable Programmable Read Only Memory (EEPROM)

RAM memory is volatile memory and used for temporary storage of the data. And the selection of it depends on the user need and the application.

The ROM memory or code memory is used for the storage of the program. Once system powered, the system fetches the code from the ROM memory.

The EEPROM is a unique memory. The content can be erased and reprogrammed by a high voltage pulse input. This is used to store the data by the program itself. Suppose we have a temperature data logger, And it needs to store the data very one hour. It means we need the data at runtime after the system is started. The system will read temperature and store in the EEPROM memory. And it will be permanents and you can retrieve the data later.

So an embedded system developer decide which memory to use for its application.

Timers-counters:

If you are working in embedded systems you must have heard about

- What are timer and counter?
- Why we use timer and counter?

In some application, we need to generate some delay. Like for blinking an LED, we need a delay. For making square pulse we need a delay. But there is some issue when we generate the delay from the normal coding style by making any loop running for a particular time. Definitely, this will give you some delay but the code after this loop remains in waiting for state and delayed.

So it is not the best approach to generate the delay. For such kind of application where we need a delay for a specific time interval without affecting the normal code execution, we use timer and counter.

By setting some register for timer and counter using the programming we get the desired delay. The amount of delay depends on the system frequency and crystal oscillator.

Communication Ports:

Embedded systems hardware has different types of communication ports to communicate with the other embedded devices.

Input and Output:

To interact with the embedded systems we need input. The input may be provided by the user or by some sensor. Sometimes some systems need more input or output. So the selection selection will be based on I/O.

Application specific circuits:

Some hardware components are common while designing the embedded systems. But some are different and depends on the application need.

Like a temperature sensor need a temperature sensor for sensing the temperature. While others hands an alcohol detector has a sensor to detect the alcohol level.

4. **Washing machine is an embedded system. Explain**

Answer:

Washing machine supports three functional modes:

i) **Fully Automatic Mode**

In fully automatic mode, once the system is started it perform independently without user interference and after the completion of work it should notify the use about the completion of work. This mode instantaneously sense cloth quality and requirement of water, water temperature, detergent, load, wash cycle time and perform operation accordingly.

ii) **Semi Automatic Mode**

In this semi automatic mode in which washing conditions are predefined. Once the predefined mode is started completion it forms the user its job and after completion it informs the user about the completion of work.

iii) **Manual Mode**

In this mode, user has to specify which operation he wants to do and has to provide related information to the control system. For examples, if user wants to wash clothes only, he has to choose 'wash' option manually. Then the system asks the user to enter the wash time, amount of water and the load. After these data are entered, the user should start the machine. When the specified operation is completed system should inform the user.

Remember that modes should be a selectable be a keypad.

A washing machine may have a system controller (Brain of the system) which provides the power control for various monitors and pumps and even controls the display that tells us how the wash cycles are proceeding. A washing machine comprises several components as shown in figure below:

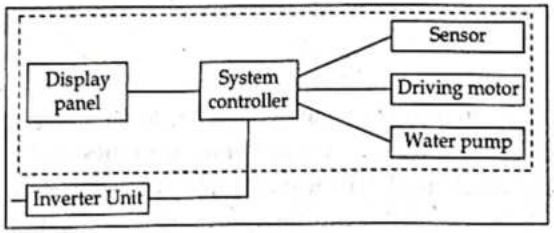


Figure: Block diagram of washing machine

The working of these components is as follows:

i) **Display panel**

It is a touch panel screen to control all the operations of a machine.

ii) **Sensor**

It measures the water level and appropriate amount of soup. Input devices for automatic washing machine are sensors for water flow, water level and temperature; door switch; selector knob or buttons for settings such as spin speed, temperature, load size and types of wash cycle required.

Water level sensor:

It indicates beep sound when water level is low in washing tub.

Door sensor:

It indicates deep sound when all clothes are washed that means now you can open the machine door and also you can move to your next phase. Next phase will be dry phase. This phase also follows same concept for drying the clothes.

iii) **Driving Motor**

Motor can rotate in two directions either "reverse" or "forward". The forward direction drives the current in forward direction and motor rotates forward. The reverse direction driver does the opposite of it. A washing machine can maintain single motor in fully automatic or double motor in semi automatic washing machine.

Sequence of washing the clothes with this can be explained in few steps as follows:

1. Put on your dirty clothes on to the wash tub for washing.
2. Put the detergent soap.
3. Put ON the tap, water rushes inside the tub.
4. If its electronic control, then by the press of the keys, you could program, if its mechanical it shall something like an mechanical switches wherein you are allowed to operate for setting the wash time.
5. Now the wash motor rotates and washes the clothes and gives you a deep sound.
6. Now your clothes are washed ..... removed it from the wash tub and put it on the spin tub and program it accordingly ..... after spinning clothes are dried and you are allowed to hang it for proper drying in sunlight.

The fully automatic also comes in two category front loading as well as top loading.

a) Front loading is the one wherein you are given an opening to put clothes in on the front side.

b) Top loading is on the top.

iv) **System controller**

Such component is used to control the motor speed. Motor can be move in forward direction as well as reverse direction. System controller reads the speed of motor and controls the speed of motor in different phases such as in washing, cleaning, drying etc. All kinds of sensors such as door sensor, pressure sensor and keypad, speed sensor are also maintained by this.

v) **Water pump**

The water pump is used to re-circulate water and drain out the dirty water. This pump actually contains two separate pumps inside one. The bottom half of the pump is hooked up to the drain line, while the top half recirculates the wash water. The motor that drives the pump can reverse direction. It spins one way when the washer is running a wash cycle and recirculates the water; and it spins the other way when the washer is doing a spin cycle and draining the water.

5. Explain the components of embedded system software.

Answer:

There are different software tools for programming and coding. These software tools are referred to as software components.

We need a program written in assembly or in embedded C language. And then we compile it. This complied code converted into HEX code. This hex code is programmed or burned into the ROM of the system using programmer.

These are the tools that are generally used in embedded system development.

- Assembler
- Emulator
- Debugger
- Compiler

Assembler

When you program in assembly language. This assembly language program is converted into the HEX code using this utility. Then using some hardware called as a programmer we write the chip.

Emulator

An emulator is hardware or software tool that has a similar functionality to the target system or guest system. It enables the host system to execute

the functionality and other components. It is a replica of the target system and used for debugging the code and issues.

Once program or code is fixed at the host system. It is transferred to the target system.

Debugger

Sometimes we are not getting expected results or output due to errors or bug. There are certain tools that are specifically used for the debugging process, where we can see the controls flow and register value to identify the issue.

Compiler

A compiler is a software tool that convert one programming language into target code that a machine can understand. The compiler basically used for translating the high level language into the low level language like machine code, assembly language or object code.

EXAM SOLUTION (TU, IOE)

1. What are the common characteristics of embedded system? Explain. [2069 Bhadra]

Answer: See 1.1.1 of chapter 1.

2. What is an embedded system? Describe its various applications. [2070 Magh]

Answer:

An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task.

Second part: See the topic 1.5.1 of chapter 1.

3. What are the major purposes of embedded system? [2071 Bhadra]

Answer: See the topic 1.5 of chapter 1.

4. What are the common characteristics of embedded system? How does a digital camera satisfy those characteristics? [2072 Ashwin]

Answer:

First part: See the topic 1.1.1 of chapter 1.

Second part:

A digital camera is very good example of embedded system. Cameras that we use today are smart and have a lot of features that were not present in early cameras all because of embedded system used in them. A digital camera has basically three functions, to capture image which we call data, to store image data, and to represent this data. Today images are stored and processed in form of digital data in bits. There is no need of film for storing images. This feature has increased the storage capacity and made it easy to transfer images. In digital cameras first image is captured and converted to digital form. This digital image is stored in internal memory. When the camera is attached to your personal computer for uploading images, it transfers the stored data. Smart camera has some extra features than digital cameras. Components of a smart camera include,

- Image sensor that may be a CCD (Charge Coupled Device) or a CMOS (Complementary metal oxide semiconductor).
- Analog to digital converter
- Image processor
- Memory
- Lens
- Led or other illuminating device
- Communication Interface etc.

Smart cameras may consist of some more devices depending on features. So, we can say that camera is one of the important embedded systems examples. It has its own processor, sensors, actuators and also memory for storage purposes.

5. Define embedded system and classify the embedded system based on generations. [2072 Magh]

Answer:

An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task.

Embedded system based on generations:

i) **First Generation (1G):**
- Built around 8 bit microprocessor and microcontroller.
- Simple in hardware circuit and firmware developed.
- *Examples*: Digital telephone keypads.

ii) **Second generation (2G):**
- Built around 16-bit μP and 8-bit μC.
- They are more complex and powerful than 1G μP and μC.
- *Examples*: SCADA systems

iii) **Third generation (3G):**
- Built around 32-bits and 16-bits μC.
- Concepts like Digital Signals Processor (DSPs), Application Specific Integrated Circuits (ASICs) evolved.
- *Examples*: Robotics, media etc.

iv) **Fourth generation:**
- Built around 64-bit μP and 32-bit μC.
- The concept of system on chips (SoC), multi-core processors evolved.
- Highly complex and very powerful.
- *Examples*: Smart phones.

6. What is an embedded system? List out its different types. Why is it so hard to define? [2073 Bhadra]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.2 of chapter 1.

Third part:

Embedded systems are hard to define because they cover such a board range of electronic devices.

7. Define embedded system. What are the typical characteristics of embedded system? [2073 Magh]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.1.1 of chapter 1.

8. Define embedded system. Clarify the statement 'Digital camera is a good example of an embedded system.' [2074 Bhadra]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See question number 4 (Example Solution TU).

9. What is a design metric and explain the purpose of embedded system. [2075 Baisakh]

Answer:

A design metric is a measure of an implementation's features such as cost, size, performance and power.

Second part: See the topic 1.5 of chapter 1.

10. Define embedded system and classify the embedded system based on generations. [2076 Bhadra]

Answer: Same as question of 2072 Magh TU.

11. What is embedded system? List out reasons for sudden interests in embedded system. How general purpose operating system is differ then real time operating system. [2077 Poush]

Answer:

An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task.

Second part: See the topic 1.5 of chapter 1.

Third part: See the chapter 6.

12. Define embedded system. Explain it's purpose and application in today's digital world. [2078 Baisakh]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.5 of chapter 1.

OLD EXAM QUESTIONS SOLUTION (PoU)

1. Define embedded system? Explain the different characteristics of embedded system. Give few application areas. [2016 Fall PoU]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.1.1 of chapter 1.

Third part: See the topic 1.5 of chapter 1.

2. What is embedded system? Explain components of embedded hardware. [2016 Spring]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: Same as question number 3 of some solved question.

3. What is embedded system? Explain essential components of embedded system. [2017 Fall]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.3 of chapter 1.

4. What is an embedded system? List and define the three main characteristics of embedded systems that distinguish embedded system from other computing systems. [2017 Spring]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: See the topic 1.1 of chapter

5. What is an embedded system? Justify, how automatic fuel machine is a good example of embedded system? [2018 Fall]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: Washing machine is a automatic fuel machine. Explained in [some solved questions number 4].

6. Define embedded system. Discuss the various skills required for an embedded system designer. [2018 Spring]

Answer:

First part: See question number 5 (Example Solution TU).

Second part: The various skills required for an embedded system designer are;

i) In-depth knowledge

Whether you are an assembly language programmer or design embedded circuits; you must have in-depth knowledge in your vertical. The more knowledgeable you are the more value you bring and the less dispensable you become. Subject matter experts are invaluable. Build your knowledge base and your reputation as the person who holds the knowledge. You will establish a reputation for yourself, your team and your company while making yourself indispensable.

ii) Be an all-arounder

Engineers who understand the hardware and software parts of any design, including embedded systems, are in high demand these days. You don't have to be able to design the board and create the software at the same time, but you should have at least a basic understanding of one while working with the other.

iii) Network

Online forums, whether it's the internet, a coworker, industry journals, or focus groups learn. Pose inquiries respond to questions that you are able to answer. Take part in activities. You will expand your social and professional network in addition to your knowledge is arguably the finest approach to learn something on your own. As a result, don't limit yourself to using it for social networking and pleasure. Perhaps someone has already solved the issue you are having with your Raspberry Pie board. So, take use of everything that is open-source. Join an online group or a forum connected to your field to gain new skills that will help you advance as an embedded system engineer.

iv) Learn internet-based technologies

Internet of things and ubiquitous computing are going to take the world by storm sooner or later. When this occurs, the devices in our environment that are already based on embedded technologies will become smarter and communicate with one another through internet-based communication. Then knowing the ins and outs of embedded, C won't be enough. You must learn such internet-based technologies today in order to stay relevant.

v) Become familiar with the latest processors

There's nothing wrong with continuing to use the 8051 or ARM processors. You're in severe trouble if you don't know anything about the latest processors and MCUs. You will need some understanding and experience with the latest MCUs to be a good embedded systems engineer. So, as soon as you can your hands on the latest technology, start fiddling with it.

vi) Prefect your project management skills;

You are a professional as well as an embedded systems engineer. Most of the time, you will be working with a group of people to achieve a specific goal. You will have to work together and coordinate with these people as

well as your coworkers. Even if you are not the project manager, your team may be allocated to one. It will only benefit you in the long run if you understand the method, steps, and approach. Skills in project management will assist you in dealing with the numerous aspects of business life.

vii) Troubleshoot

You could be a fantastic engineer who excels in designing, coding and putting things together. Even if you have superhero-like abilities, you will run into situations when your design fails. Whether it's a malfunctioning IC or a faulty component, the bottom line is that the problem must be fixed. Having excellent troubleshooting abilities will pay off handsomely. You will save time, effort, and possibly embarrassment if you can identify, isolate, and resolve the difficulties. Troubleshooting isn't necessarily reserved for specialists; there is a methodology that can assist you in the process. It will take time and patience to develop your trouble-shooting skills, but the work will be well worth it.

viii) Be creative

Your most significant asset is your mind. What distinguishes an excellent engineer from an average engineer creativity. You must first visualize a working system that stands out from the crowd in order to create it. Your imagination will aid you in becoming a good and effective embedded systems engineer.

7. Define embedded system. Explain essential components of embedded system? [2019 Fall]

Solution: See the question number 3 (Old Exam Question PoU 2017 Fall).

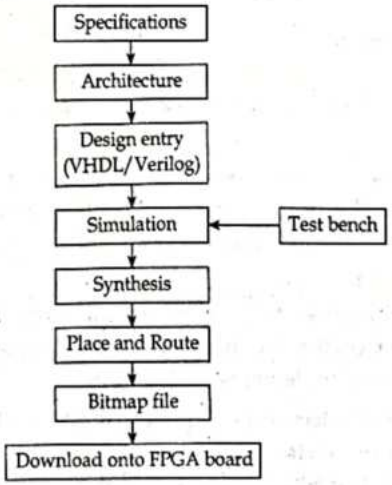
Chapter 2

HARDWARE DESIGN ISSUES

★★★

★★★

Hardware design, of course, is more constrained than software by the physical word. The systems with hardware components considering timing can be designed by using a programmable devices like PLA, PAL, PGA, FPGA or a non-programmable device like ASIC. The systems behavior can be expressed by using a Hardware Description Language like verilog HDL, VHDL etc, instead of a programming language. These HDL languages describe the timing behavior of the hardware elements. The design flow for an FPGA is shown in figure below:

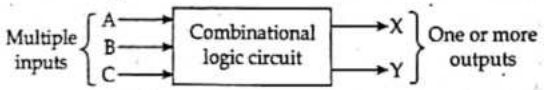


The system design with the refined specifications. The specifications are converted into architecture. The architecture functionality is expressed by using Hardware Description Language (HDL) like VHDL or verilog HDL. The functionality can be verified by using simulation with the help of test bench. After the functional verification the design can be converted into gate level structural interconnected form called as net list with the help of synthesis. The synthesized net list will be placed and routed on the selected target FPGA board. A bit file will be generated to load the designed system on to the target FPGA.

The advantage of hardware based design methodology is the timing information can be specified by using HDL and high speed system can be designed compared to software based methods.

2.1 COMBINATION LOGIC

Combination logic circuits care made up from basic logic NAND, NOR or NOT gates that are "combined" or connected together to produce more complicated switching circuits. These logic gates are the building blocks of combinational logic circuits. An example of a combinational circuit is a decoder, which converts the binary code data present at its input into a number of different output lines, one at a time producing an equivalent decimal code at its output.



Combinational logic circuits can be very simple or very complicated and any combinational circuit can be implemented with only NAND and NOR gates as these are classed as "universal" gates.

The three main ways of specifying the functional of a combinational logic circuit are:

I) Boolean algebra

This forms the algebraic expression showing the operation of the logic circuit for each input variables either true or false that results in a logic '1' output.

II) Truth Table

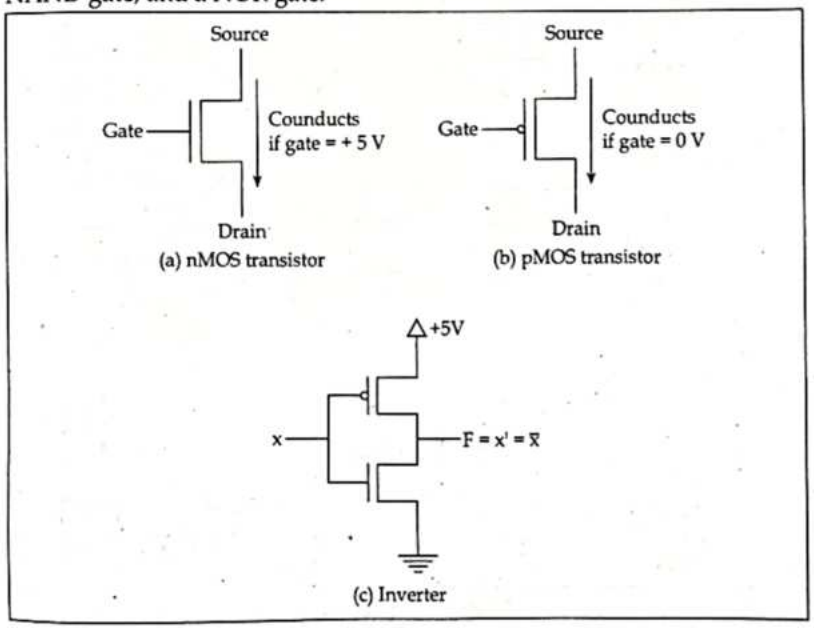
A truth table defines the function of a logic gate by providing a concise list that shows all the output states in tabular form for each possible combination of input variable that the gate could encounter.

III) Logic Diagram

This is a graphical representation of a logic circuit that shows the wiring and connections of each individual logic gate, represented by a specific graphical symbol that implements the logic circuit.

A transistor is a basic electrical component digital system. Combinations of transistors more abstract components called logic gates, which designers primarily use when building digital systems.

A transistor act as a simple on/off switch. One type of transistor (CMOS-Complementary Metal Oxide Semiconductor) is shown in figure 2.1 (a). The gate controls whether or not current flows from the source to the drain. When a high voltage (typically +5 volts, which we will refer to as logic 1) is applied to the gate, the transistor conducts, so current flows. When low voltage (which we will refer to as logic O, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in figure 2.1 (b). When logic O is applied to the gate, the transistor conducts, and when logic 1 is applied, the transistor does not conduct. Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in fig 2.1 (c). When the input x is logic O, the top transistor conducts (and the bottom does not), so logic 1 appears at the output F. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic O, as shown in figure 2.1 (d). When at least one of the inputs x and y is logic O, then at least one of the top transistors conducts (and the bottom transistors do not), so logic 1 appear at F. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones do, so logic O appears at F. Likewise, we can easily build a circuit whose output is logic 1 when both of its inputs are logic O, as illustrated in figure 2.1 (e). The three circuits shown implement three basic gates an inverter, a NAND gate, and a NOR gate.



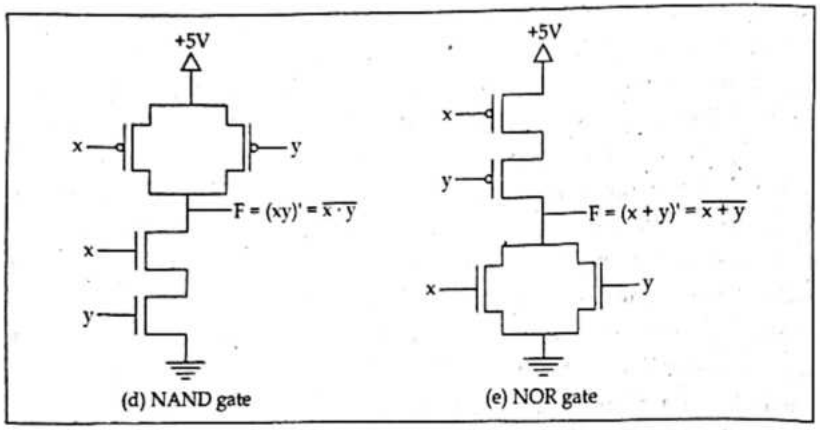


Figure 2.1: CMOS transistor implementations of some basic logic gates

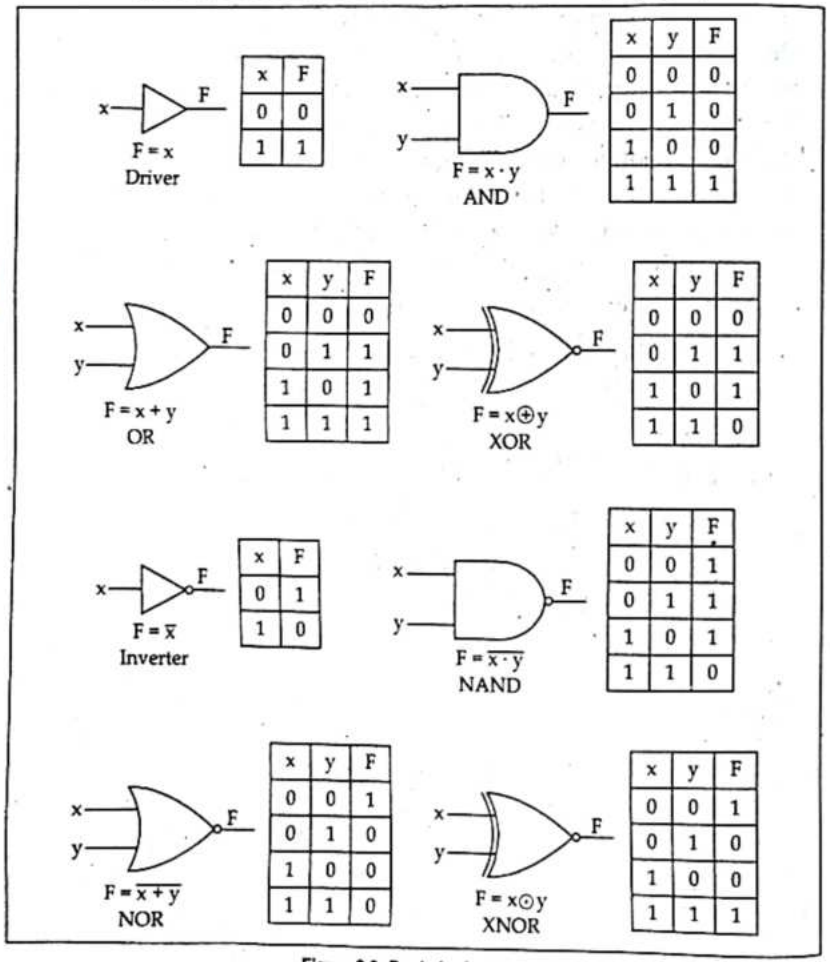


Figure 2.2: Basic logic gates

Digital system designer usually work with logic gatesm not transistors. Figure 2.2 describes 8 basic logic gates. Each gate is represented symbolically, with Boolean equation, and with a truth table. The truth table has inputs on the left, and output on the right. The AND gate outputs 1 if and only if both input are 1. The OR gate outputs 1 if and only if at least one of the input is 1. XOR (exclusive-OR) gate outputs 1 if and only if exactly one of its two inputs is 1. The NAND, NOR and XNOR gates output the complement of AND, OR and XOR respectively.

A combinational circuit is a digital circuit whose output is purely a function of its inputs; such a circuit has no memory of past inputs. We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in figure 2.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often want to minimize the logic gates in the circuit. We can minimum the output equations by algebraically manipulating the equations. Alternatively, we can use karnaugh maps, as shown in figure below.

a) Problem description:

y is 1 if a is equal to 1, or b and c is equal to 1, z is 1 if b or c is equal to 1, but not both.

b) Truth table:

Input			Output	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

c) Output equations

$y = \bar{a}bc + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc$

$z = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c + ab\bar{c} + abc$

d) **Minimized output equations**

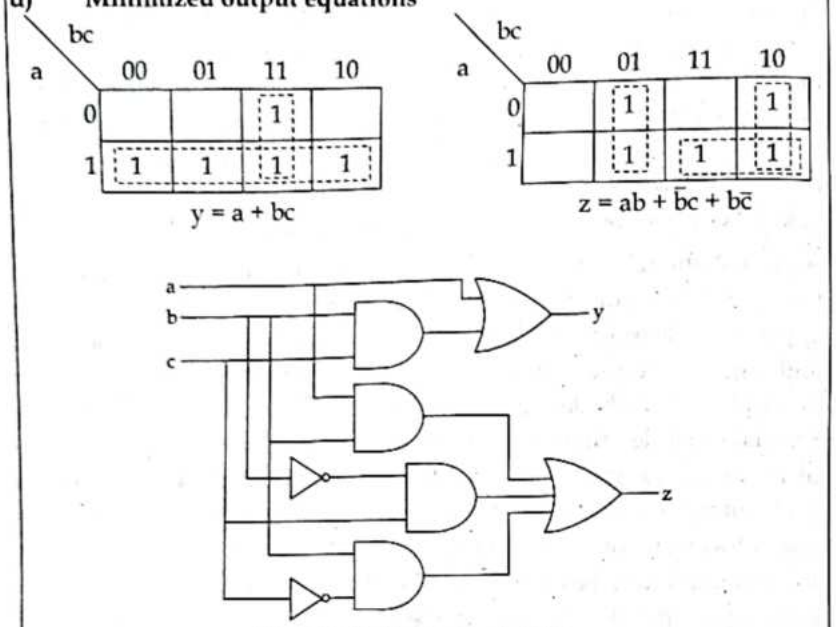
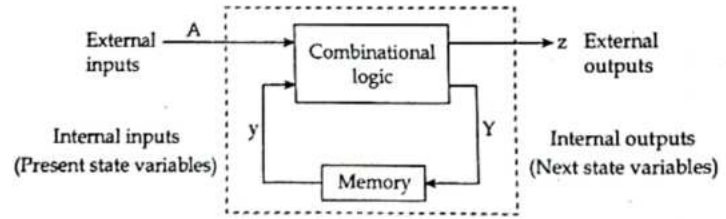


Figure 2.3: Combinational logic design

Once we've obtained the desired output equations (minimized or not), w can draw the circuit diagram.

Although we can design all combinational circuits in the above manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2^{16} or 64 K, rows in its truth table. One way to reduce the complexity is to use components that are more abstract than logic gets. Figure 2.4 shows several such such combinational components.

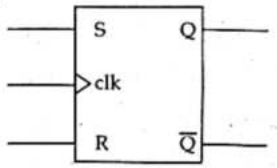
2.2 SEQUENTIAL LOGIC



A sequential circuit is a digital circuit whose outputs are a function of the current as well as previous input values. In other words, sequential logic possesses memory. One of the most basic sequential circuits is flip-flop. A flip-flop stores a single bit. The simplest type of flip-flop is the D flip flop.

a) **SR flip-flop:**

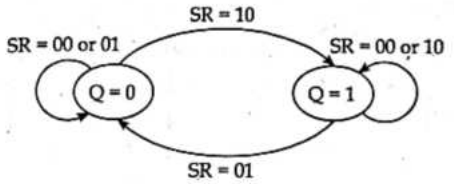
FF symbol:



Characteristic (Truth) Table:

S	R	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	NA

State diagram/Characteristic equations:



Excitation Table:

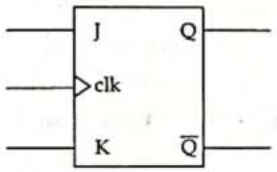
Q	Q_{next}	S	R
0	0	0	×
0	1	1	0
1	0	0	1
1	1	×	0

$Q_{next} = s + R'Q$

$SR = 0$

b) **JK flip-flop:**

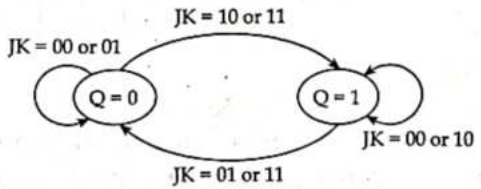
FF symbol:



Characteristic (Truth) Table:

J	K	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

State diagram/Characteristic equations:



Excitation Table:

Q	Q_{next}	J	K
0	0	0	×
0	1	1	×
1	0	×	1
1	1	×	0

$Q_{next} = J'K'Q + JK' + JKQ'$

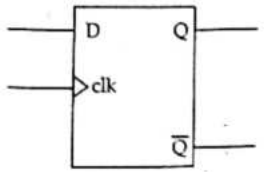
$= J'K'Q + JK'Q + JK'Q' + JKQ'$

$= K'Q(J' + J) + JQ'(K' + K)$

$= K'Q + JQ'$

c) D flip-flop:

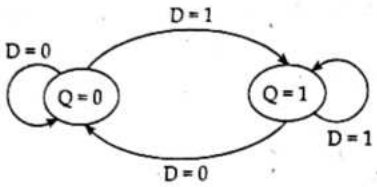
FF symbol:



Characteristic (Truth) Table:

D	Q_{next}
0	0
1	1

State diagram/Characteristic equations:



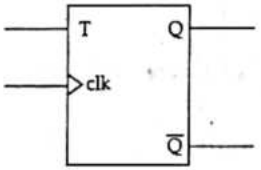
$Q_{next} = D$

Excitation Table:

Q	Q_{next}	D
0	0	0
0	1	1
1	0	0
1	1	1

d) T flip-flop:

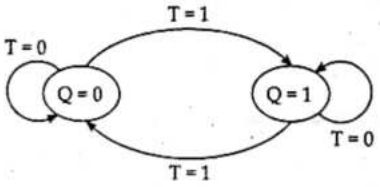
FF symbol:



Characteristic (Truth) Table:

T	Q_{next}
0	Q
1	$\overline{Q}$

State diagram/Characteristic equations:



$Q_{next} = TQ' + T'Q = T \oplus Q$

Excitation Table:

Q	Q_{next}	T
0	0	0
0	1	1
1	0	1
1	1	0

The characteristics table is a shorter version of the truth table that gives for every set of input values and the states of the flip-flop before the rising edge, the corresponding state of the flip-flop after rising edge of the clock. It is used during the analysis of sequential circuits.

The characteristic equation is just the functional expressions derived from the characteristics (truth) table. It formally describes the functional behavior of a latch or flip-flop. They specify the flip-flop's next state as a function of its current state and inputs.

The excitation table gives the value of the flip-flop inputs that are necessary to change the flip-flop's present state to the desired next state after the rising edge of the clock signal. It is obtained from the characteristics table by transposing input and output columns. It is used during the synthesis of sequential circuits.

A register stores n bits from its n-bit data input I with those stored bits appearing at its output Q. A register usually has at least two control inputs, clock and load. For a rising-edge triggered register, the inputs I are only stored when load is 1 and clock is rising from 0 to 1. The clock input is usually drawn as a small triangle. Another common register control input is clear, which resets all bits to 0, regardless of the value of I. Because all n bits of the register can be stored in parallel, we often refer to this type of register as a parallel-load register.

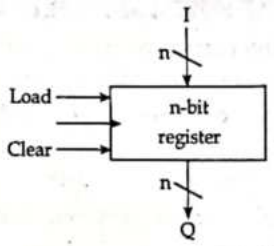
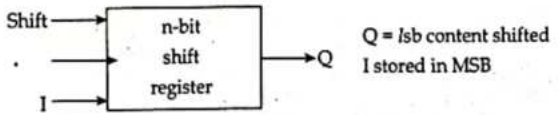


Figure: n-bit register

A shift register stores n bit but they cannot be stored in parallel. Instead they must be shifted into the register *i.e.*, one bit at a single clock. The register have at least a data input 'I' that holds a single bit at at time and control input shift that is used to insert a data. When clock is rising mode the shift equals 1 such that the data bits in 'I' is inserted into 'n' bit position of register while n^{th} bit is inserted into $(n-1)^{th}$ and (n – 1) bit into (n –2) and so on. The first bit is usually shifted to the output end Q.



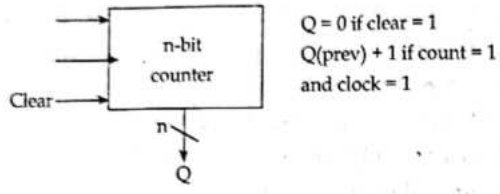
A counter is a register that can be also increment, meaning add the binary value 1, to its stored binary value. A counter has a control input clear that resets all bits of register to value '0' and count input that increments the value of stored number by 1 in each raising edge of clock. A counter has also a load input to load the n-bit data in parallel. Commonly a counter operates in both mode as down and up. The up counter increments the value stored in register by 1 and down counter decrements the contents of register by value 1 upto define level. Mode M counter counts from 0 to M-

1 or M-1 to 0. For this it requires another control input as count UP/DOWN. These control input may be

i) Synchronous

ii) Asynchronous

A synchronous input value only have effect during in a clock edge. An asynchronous value effect the circuit independent of clock.



2.3 CUSTOM SINGLE PURPOSE PROCESSOR DESIGN

A processor is an integrated electronic circuit that performs the calculations that run a computer. The minimum requirement to be a processor is the presence of controller and data path. Processor is the heart of an embedded system. It is the basic unit that takes inputs and produces an output after processing the data.

- Digital circuit that performs a computation tasks.
- Controller and data path.
- General-purpose: Variety of computation tasks.
- Single-purpose: One particular computation task.
- Custom single-purpose: Non-standard task.

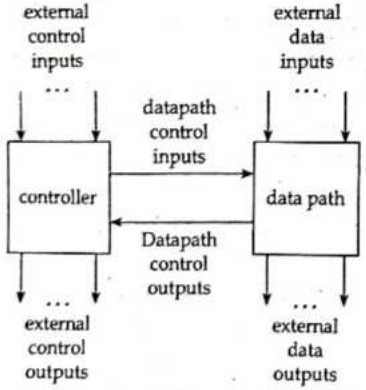


Figure: Controller and data path

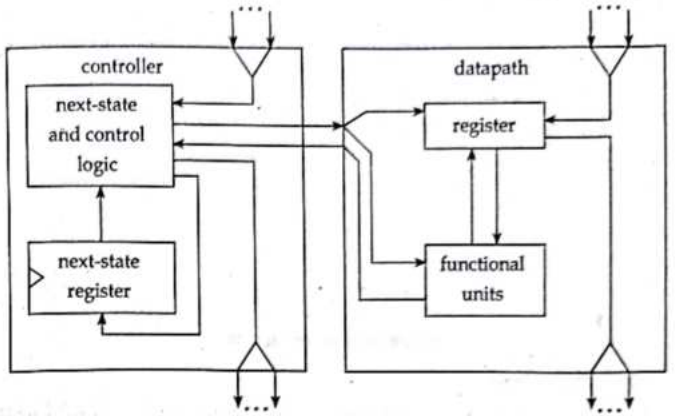


Figure: A view inside the controller and data path

Data Path

The data path stores and manipulates the system data. Examples of data in embedded system includes binary numbers representing external conditions like temperature or speed, the character to be displayed on the screen or digitized photographic image to be stored and compressed. Data path consists of registers, ports, functional unit and and the interconnection between these modules. Thus data path can be configured or designed to read data from particular register, feed them through certain function units and store the result back to a register.

Controller

A controller carries out the configuration of data path. It sets the data path control signals like load input to register, select inputs for selecting the register, operation selection for functional unit and connection unit to obtain the desired configuration at particular instant of time. Controller is capable of monitoring external control inputs as well as data path control outputs.

Both controller and data path can be designed using combinational and sequential logic.

Statements

Statements are used to implement the logical operation, data flow and control flow during the custom single processor design. Some of the useful statements are:

i) Assignment statement

ii) Loop statement

iii) Branch statement

1) Assignment Statement

- For an assignment statement, create a state with that assignment as its action.
- Add an arc from this state to the state for the next statement.

Assignment statement

a = b

Next statement

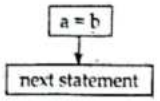


Figure: Assignment update

2. Loop Statement

- For a loop statement, create a condition statement (c) and join state (J) both with no action
- Add arc with the loop's condition from the condition state to the first statement in the loop body.
- Again add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body.
- Don't forget to add an arc from join state back to condition state.

```
While (cond) {
        Loop-body-statement;
    }
Next statement;
```

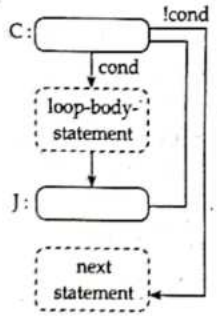


Figure: Loop template

3. *Branch Statement*

- For branch statement, we create a condition state C and a join state J, both with no actions.
- Add an arc with the first branch's condition from the condition state to the branch first statement.

- Similarly add another arc with the complement of the first branch condition ANDed with second branches condition from the condition state to the branch's first statement.
- Same procedure is repeated for each branch
- Finally connect the arc leaving the last statement of each branch to the join state and add an arc from this state to the next state.

```
If (C1)
    C1 statement;
else if (C2)
    C2 statement;
else
    Other statements;
Next statement
```

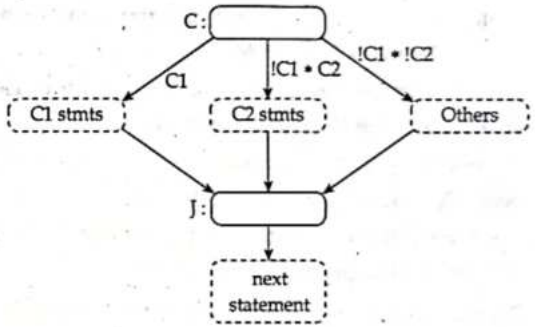


Figure: Branch template

 # Steps to design custom single purpose processor

1. To begin building a custom single purpose processor we need to convert a program's statement into a complex state diagram.
   - In complex state diagram, the states and arc (transitional condition) may include arithmetic expression and those expressions any use external inputs and outputs as well as variables.
   - Complex state diagrams looks like a sequential programs in which statements have been changed into states.
   - We classify statements into assignment, loop and branch.
2. Once complex state diagram is obtained, functionality is divided between controller and data path.
   - The data path should only consist of an interconnection of the combinational and sequential blocks.
   - The controller should consists of a basic state diagram *i.e.*, one containing only Boolean action and conditions.

Constructing the data path

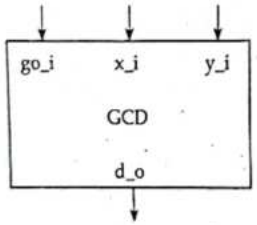
- Create a register, for any declared variable
- Create a functional unit for each arithmetic operation in the state diagram.
- Then connect the ports, register and functional units together.
- For each logic and arithmetic operation connect sources to input of the operations corresponding functional blocks.
- When more than one source is connected to a register, add an approximately sized multiplexer.
- Finally create a unique identifier for each data path components, control inputs and outputs.

Constructing the controller

- The state diagram for the controller has the same characteristics as the complex state diagram (FSMD) but the complex actions have been replaced by the Boolean ones.
- Replace every variable "write" by action that set the select signal of multiplexer in front of the variable register.
- Replace every logical operation in a condition by the corresponding functional unit control output.
- Complete the controller design using a basic state diagram (FSM) using the standard sequential design process.

For example: Design of custom single processor to find the GCD of given two numbers.

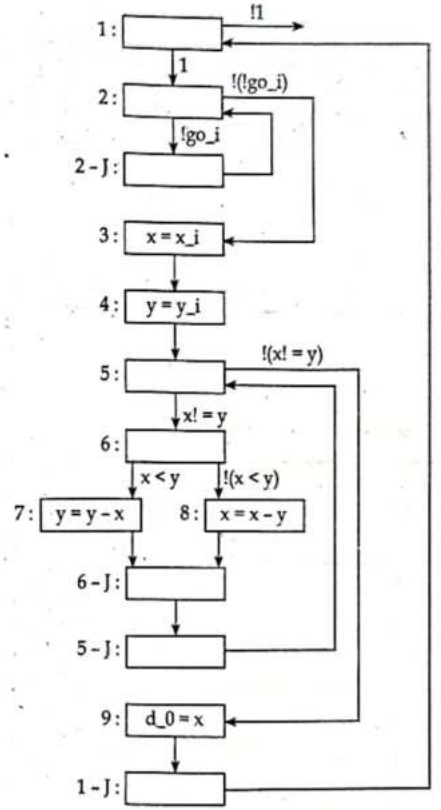
Suppose Xi and Yi be the two input numbers, go be the control input and do is the GCD of Xi and Yi such that the black box, functionality and state diagram be designed as below be represented as:




```
0: int x, y;
1:  while (1) {
2:  while (! go_i);
3:  x = x_i;
4:  y = y_i;
5:  while (x! = y) {
```



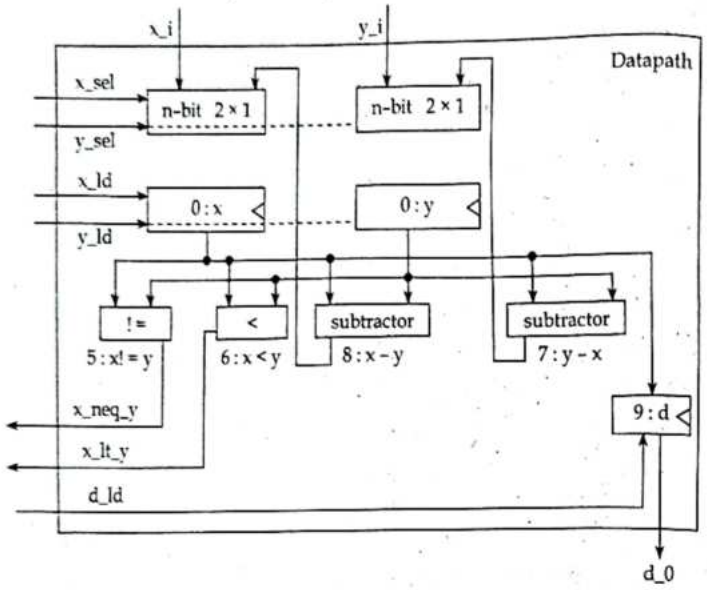
```
6:  if (x < y)
7:  y = y_x;
    else
8:  x = x-y;
    }
9:  d_0 = x;
    }
```



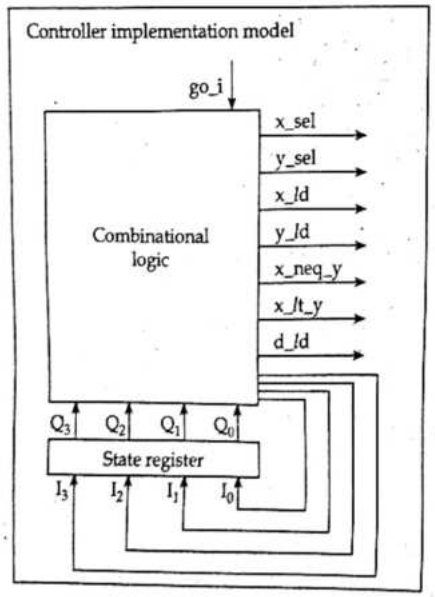
The data path can be designed as below:

- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources

- Create unique identifier
  - For each data path component control input and output.



The controller of the above functionality is



Controller state table for GCD:

Inputs							Outputs								
Q_3	Q_2	Q_1	Q_0	x-neq-y	x-*lt*-y	go-i	I_3	I_2	I_1	I_0	x-sel	y-sel	x-*ld*	y-*ld*	d-*ld*
0	0	0	0	-	-	-	0	0	0	1	x	x	0	0	0
0	0	0	1	-	-	0	0	0	1	0	x	x	0	0	0
0	0	0	1	-	-	1	0	0	1	1	x	x	0	0	0
0	0	1	0	-	-	-	0	0	0	1	x	x	0	0	0
0	0	1	1	-	-	-	0	1	0	0	0	x	1	0	0
0	1	0	0	-	-	-	0	1	0	1	x	0	0	1	0
0	1	0	1	0	-	-	1	0	1	1	x	x	0	0	0
0	1	0	1	1	-	-	0	1	1	0	x	x	0	0	0
0	1	1	0	-	0	-	1	0	0	0	x	x	0	0	0
0	1	1	0	-	1	-	0	1	1	1	x	x	0	0	0
0	1	1	1	-	-	-	1	0	0	1	x	1	0	1	0
1	0	0	0	-	-	-	1	0	0	1	1	x	1	0	0
1	0	0	1	-	-	-	1	0	1	0	x	x	0	0	0
1	0	1	0	-	-	-	0	1	0	1	x	x	0	0	0
1	0	1	1	-	-	-	1	1	0	0	x	x	0	0	1
1	1	0	0	-	-	-	0	0	0	0	x	x	0	0	0
1	1	0	1	-	-	-	0	0	0	0	x	x	0	0	0
1	1	1	0	-	-	-	0	0	0	0	x	x	0	0	0
1	1	1	1	-	-	-	0	0	0	0	x	x	0	0	0

2.4 OPTIMIZING CUSTOM SINGLE PURPOSE PROCESSOR

The finite state machine uses the number of states so that they can be reduced without halting the operation.

Some of the optimization opportunities are:

- Original program
- FSMD
- Data path
- FSM for controller

I) Optimization of Original Program

The program can be optimize by optimizing the number of computation, size of variable, time and space complexity, operation used like multiplication and division may have the higher cost etc. *For example*:

Original Program

```
0:  int x, y;
1:  while (1) {
2:  while (! go_i);
3:  x = x_i;
4:  y = y_i;
5:  while (x! = y) {
6:  if (x < y)
```

```
7:      y = y-x;
        else
8:      x = x-y;
        }
9:      d_0 = x;
        }
```

Let us see, how many iteration does it takes to find GCD of (42, 8)

(42,8) → (34,8) → (26,8) → (18,8) → (10,8) → (2,8) → (2,6) → (2,4) → (2,2)

Thus, we can see it takes 9 iteration to complete. Now, let us refine our algorithm by using the next approach as shown below.

Optimized program

```
0:      int x, y, r;
1:      while (1) {
2:      while (! go_i);
        // x must be the larger number
3:      if (x_i > = y_i) {
4:      x = x_i;
5:      y = y_i;
        }
6:      else {
7:      x = y_i;
8:      y = x_i;
        }
9:      while (y! = 0) {
10:     r = x%y;
11:     x = y;
12:     y = r;
        }
13:     d_0 = x;
        }
```

Again let's see how many iteration does it takes to find GCD of (42, 8).

(42, 8) → (8, 2) → (2, 0)

Thus this algorithm is for more efficient in terms of time it takes for computation of result. Thus the choice of algorithm can have the biggest impact on the efficiency of the design processor.

II) Optimization of FSMD

Some of the area of possible improvement are;

a) Merge state

- States with constants on transitions can be eliminated, transition taken is already known.
- State with independent operations can be merged.

b) Separate states

- States which requires complex operation can be broken into smaller states to reduce hardware. *e.g.*, if we have statement as

  $A = b * c * d * e$

- This statement requires three multipliers as multipliers are expensive. Breaking above operations into smaller operations can reduce number of multipliers.

  $t_1 = b * c$

  $t_2 = d * e$

  $A = t_1 * t_2$

- This statement requires only one multiplier as the same multiplier can be shared.

c) Scheduling

- The timing of output operation could be changed. We can generate. GCD output in fewer clock cycles. However changing the timing is not always acceptable.

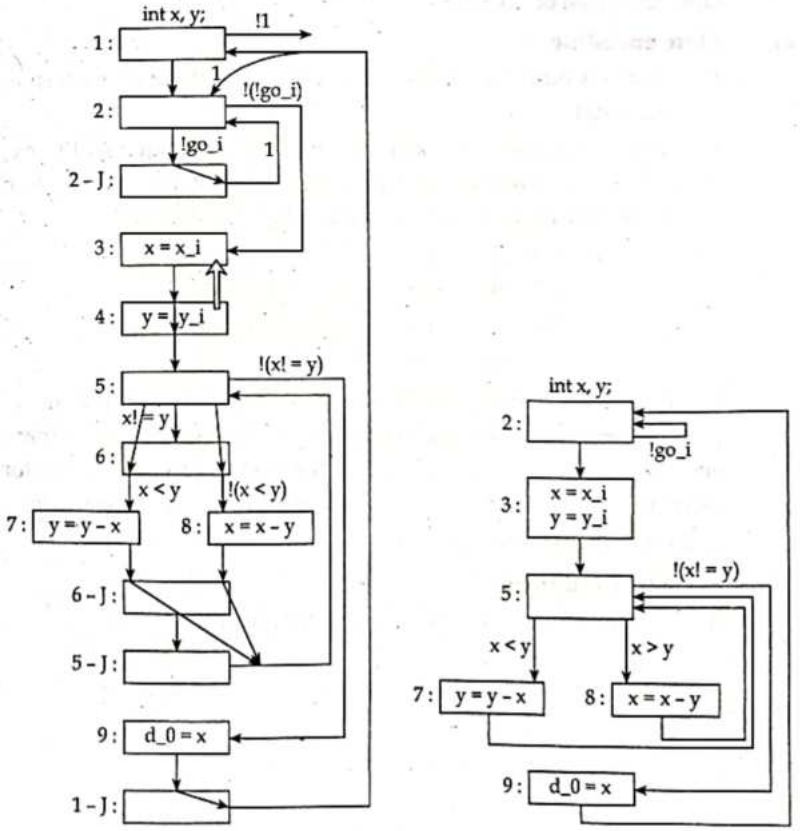


Figure (a) Original FSMD and optimization

(b) Optimized FSMD

- Eliminate the state 1 the consists constant value
- Merge the state 2 and 2-J since there is no loop operation in between them.
- Merge the state 3 and 4 since they are independent to each other.
- Merge the state 5 and 6 since transition from state 6 can be done from 5.
- Eliminate state 6-J and 5-J since it can be done from state 7 and 8.
- Eliminate the state 1-J since it can be done through state 9.

iii) Optimizing data path

To optimize the data path, a sheared functional unit can used such that there is a optimized data path and able perform the variety of operation. For this purpose we can use the sheared ALU circuit that supports variety of state equation as well as state operations. For examples for operation x-y and y-x instead of using two subtractor we can generate the result.

iv) Optimizing FSM of controller

a) State encoding

- State encoding is the task of assigning a unique bit pattern to each state in FSM.
- Any assignment in which the encoding are unique will work, but size of state register as well as size of the combination logic may differ for different encoding. *For example;*

A	00	A	11
B	01	B	10
C	10	C	01
D	11	D	10

For FSM with n states where n is a power of two, there are n! possible encoding. We can order the states and assign binary encoding starting with 00, ... 00 for first state 00, ... 01 for second state and so on. Thus there are n! possible ways for n sates. CAD tools helps in searching for the optimum state encoding.

b) State Minimization

It is the task of merging equivalent states into a single state.

OLD EXAM QUESTION SOLUTION (TU)

1. What is optimization? Explain optimization of single purpose processor in detail with suitable example. [2070 Bhadra]

Answer:

The process of improving the efficiency of a program in terms of time (speed) or space (size) is known as optimization.

Optimization of single purpose processor.

i) Optimizing the original program

a) Number of computations
b) Size of variable
c) Time and space complexity
d) Operation used

Original Program

Let us take example of GCD

```
0:  int x, y;
1:  while (1) {
2:      while (!go_i);
3:          x = x_i;
4:          y = y_i;
5:          while (x! = y) {
6:              if (x < y)
7:                  y = y - x;
            else
8:                  x = x - y;
            }
9:      d_0 = x;
    }
```

GCD of (42, 8)

(42, 8)→(34, 8)→(26, 8)→(18, 8)→(10, 8)→(2, 8)→(2, 6)→(2, 4)→(2, 2)

The process is quite lengthy. Then, optimized program is:

```
0:  int x, y, r;
1:  while (1) {
2:  while (!go_i);
    // x must be the larger number
3:  if (x_i > = y_i) {
4:      x = x_i;
```

```
5:          y = y_i;
            }
6:      else {
7:          x = y_i;
8:          y = x_i;
        }
9:      while (y! = 0) {
10:         r = x% y;
11:         x = y;
12:         y = r;
        }
13:     d_0 = x;
        }
```

GCD of (42, 8)

(42, 8) → (8, 2) → (2, 0)

In optimized program, we are putting extra variable but it reduces the time complexity.

ii) Optimizing the FSMD

FSMD can be optimized by using different areas such as merge state, separate state and scheduling.

Merge states

- States with constants on transitions can be eliminated, transition taken is already known.
- States with independent operations can be merged.

Separate states

- States which require complex operations (a × b × c × d) can be broken into smaller states to reduce hardware size.

Scheduling

- Task of assigning operations from the original program to states in an FSMD.

iii) Optimizing data path

a) Sharing of functional units

- One-to-one mapping, as done previously, is not necessary.
- If same operation occurs in different states, they can share a single functional unit.

b) Multi-functional units

- ALU support a variety of operations, it can be shared among operations occurring in different states.

iv) Optimizing the FSM of controller

a) State encoding

- It is the task of assigning a unique bit pattern to each state in an FSM.
- Size of state register and combinational logic may differ for different encoding.
- Can be treated as an ordering problem.

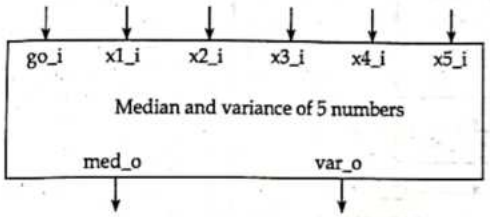
b) State minimization

- It is the task of merging equivalent states into a single state.

2. Design a dual-purpose processor that calculates the median and variance of 5 numbers entered by the user, by showing the algorithm, FSMD, FSM, data-path and controller design. [2070 Magh]

Answer:

A. Black Box

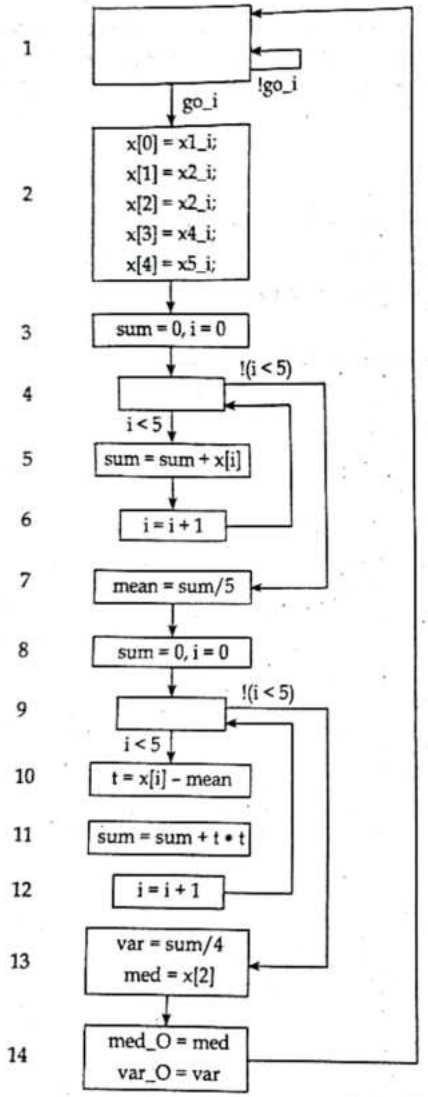


B. System Functionality

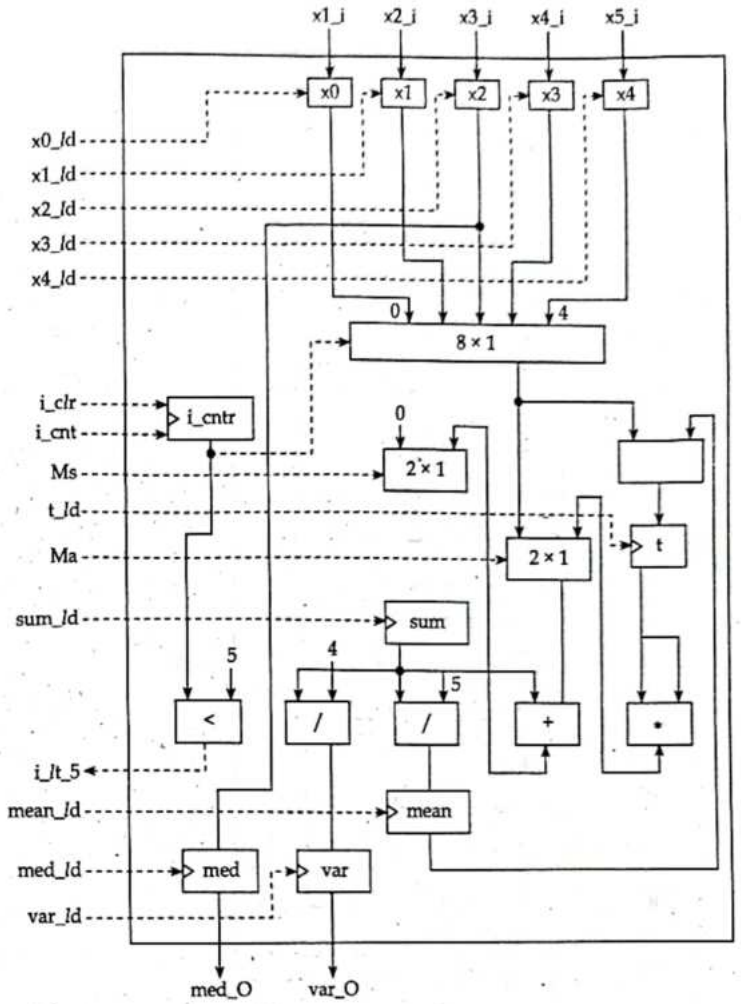
```
int x [5], i, med;
float mean, sum, t, var;
while (1) {
    while (!go_i);
    x[0] = x1_i;
    x[1] = x2_i;
    x[2] = x3_i;
    x[3] = x4_i;
    x[4] = x5_i;
    sum = 0, i = 0;
    while (i < 5) {
        sum = sum + x[i];
        i = i + 1;
    }
    mean = sum/5;
    sum = 0, i = 0;
```

```
while (i < 5) {
        t = x[i] - mean;
sum = sum + t * t;
i = i + 1;
}
var = sum/4, med = x [2];
med_O = med, var_O var;
}
```

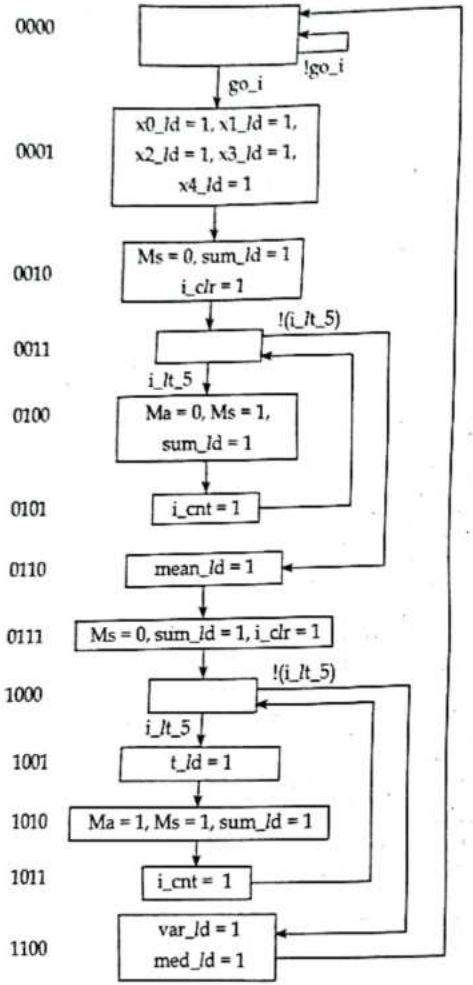
C. **FSMD**



D. **Datapath**



E. FSM



3. What are the optimization opportunities in single purpose processors? Explain with an example. [2071 Bhadra]

Answer:

Optimization opportunities in single purpose processors are;

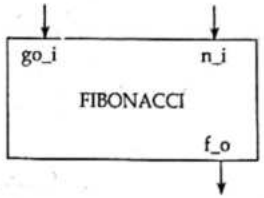
i) Original program
ii) FSMD
iii) Data path
iv) FSM

Same as question number 2 (old exam question solution TU).

4. Design a single-purpose processor that outputs Fibonacci numbers up to 'n' places. Start with a function computing the desired result, translate it into a state diagram and sketch a probable datapath. [2072 Ashwin]

Answer:

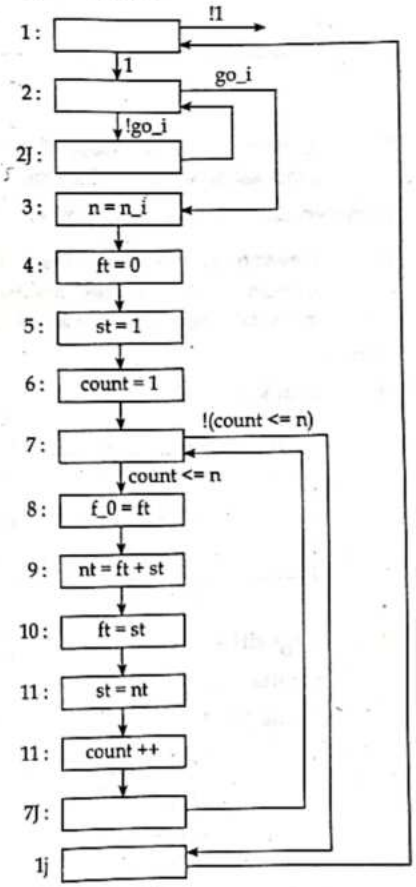
A. Black Box View



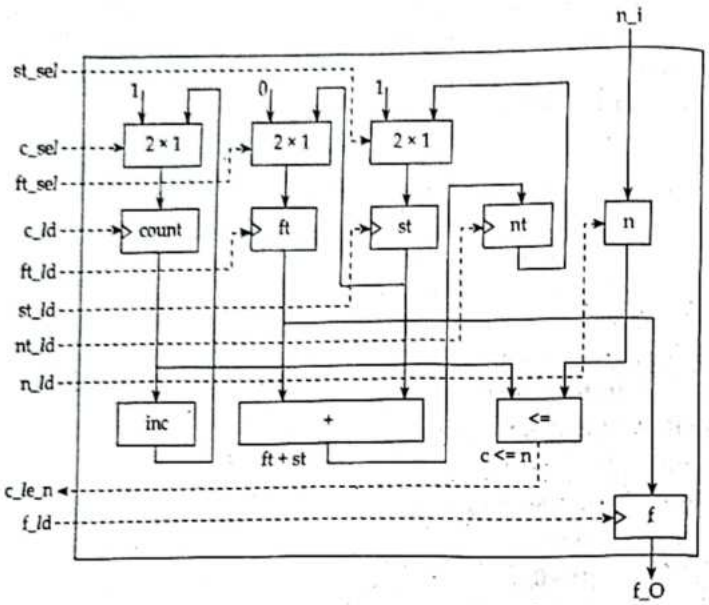
B. Functionality code

```
int ft, st, nt, count, n;
while (1) {
        while (!go_i);
        n = n_i;
        ft = 0;
        st = 1;
        count = 1;
        while (count < = n) {
            f_O = ft;
            nt = ft + st;
            ft = st;
            st = nt;
            count ++;
        }
}
```

C. FSMD



D. Data path



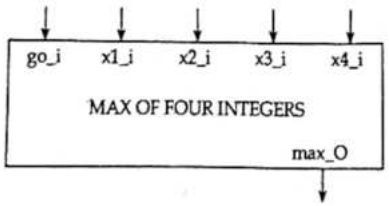
5. Explain in detail about the optimization of custom single purpose processors with a suitable example. [2072 Magh]

Answer: Same as question number 2 (Old Exam Question Solution TU).

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

Answer:

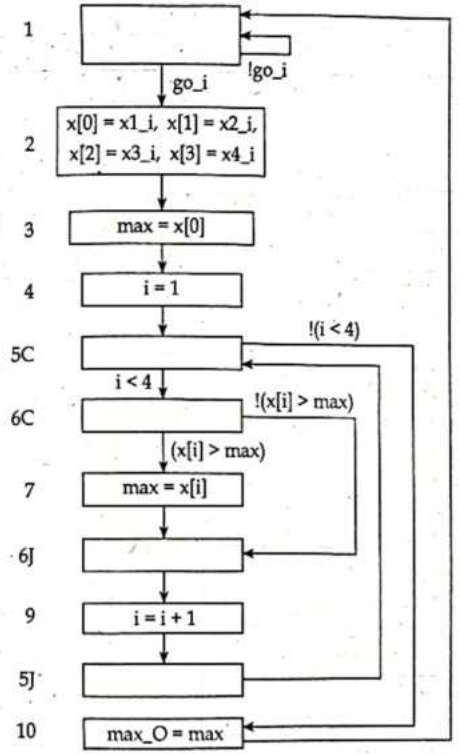
A. Black Box



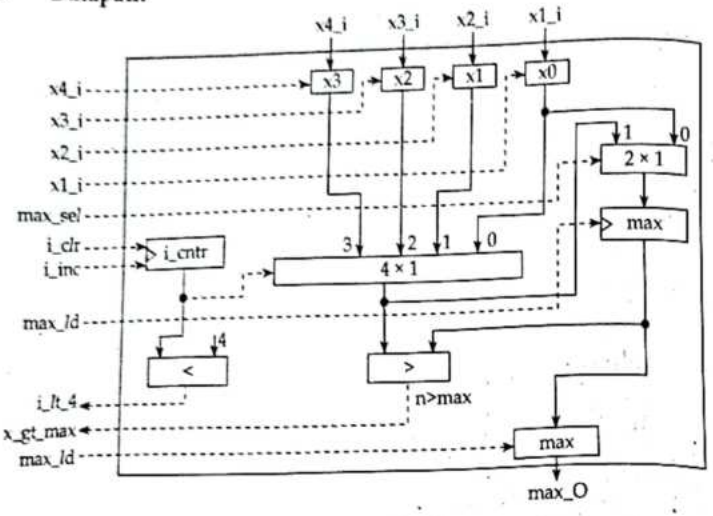
B. Algorithm

```
int max, x[4], i;
while (1) {
        while (! go_i);
        x[0] = x1_i;
        x[1] = x2_i;
        x[2] = x3_i;
        x[3] = x4_i;
        max = x[0];
        i = 1;
        while (i < 4) {
            if (x[i] > max]
                max = x[i]
            i = i + 1;
        }
    max_O = max;
    }
```

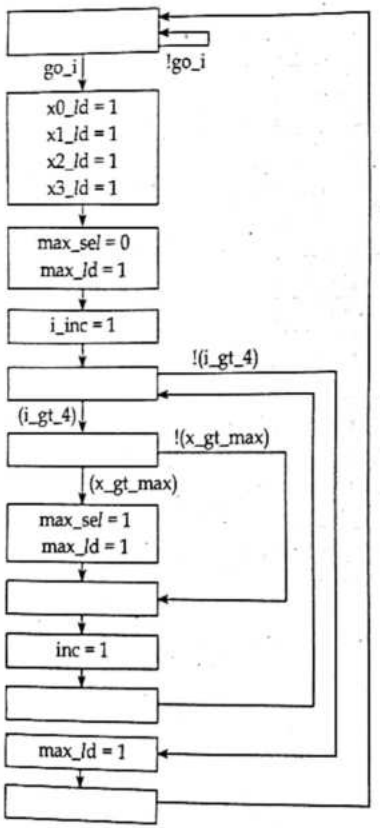

C. FSMD



D. Datapath



E. FSM

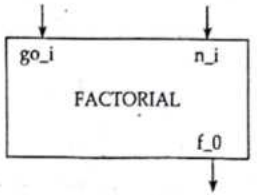


7. Design a single purpose processor that calculates factorial of an integer number 'n'. Start with a function computing the desired result, translate it into a state diagram and sketch a probable data path.

[2073 Magh, 2077 Poush]

Answer:

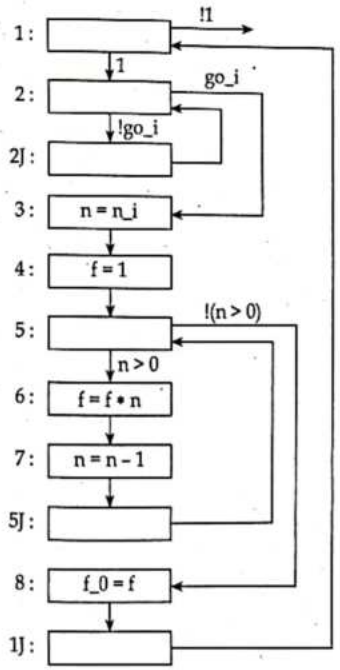
A. Black Box



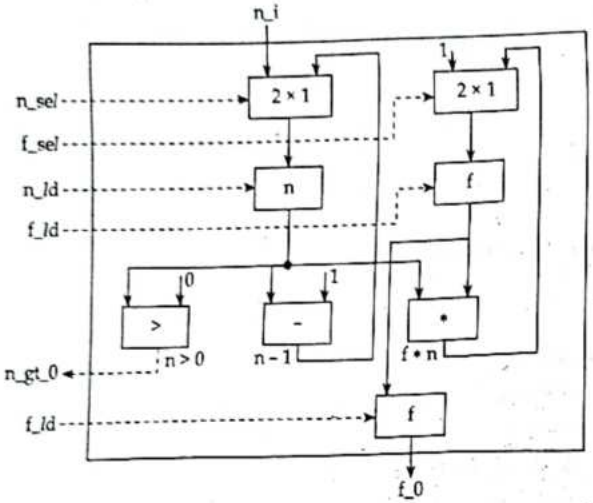
B. Functionality code

```
int n, f
while (1) {
        while (! go_i);
        n = n_i;
        f = 1;
        while (n > 0) {
        f = f * n;
        n = n - 1;
        }
        f_0 = f;
}
```

C. FSMD



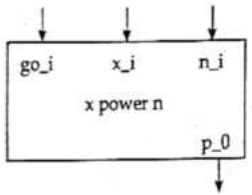
D. Datapath



8. Design a single purpose processor to determine the value of x to the power n. Start the design from the function computing the desired result, FSMD, datapath and controller. [2074 Bhadra]

Answer:

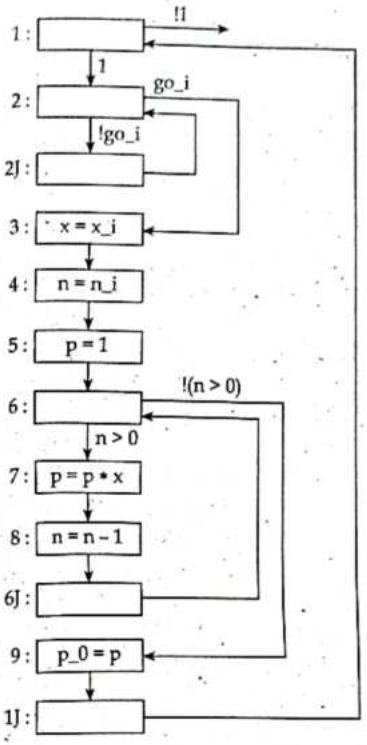
A. Black Box



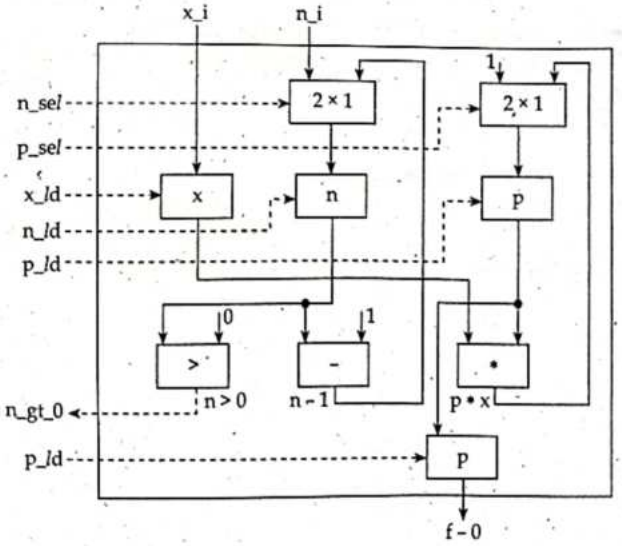
B. Functionality code

```
int x, n, p;
while (1) {
        while (! go_i);
        x = x_i;
        n = n_i;
        p = 1;
        while (n > 0) {
            p = p * x;
            n = n - 1;
        }
        p_0 = p;
}
```

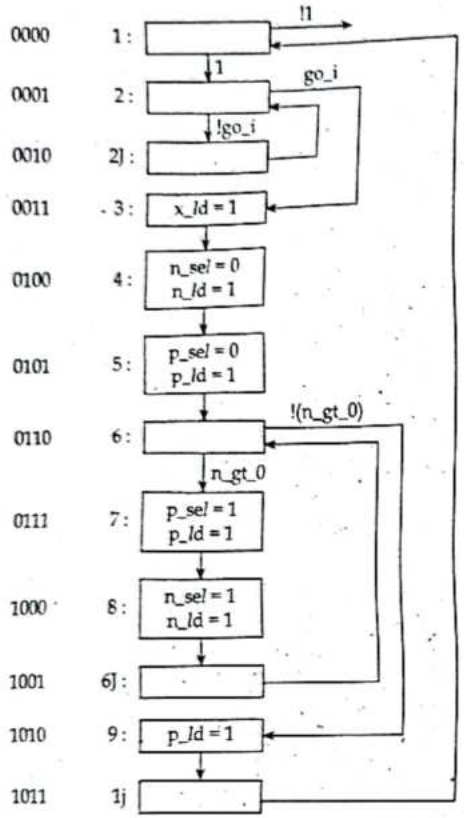

C. FSMD



D. Datapath



E. FSM controller



9. Design a single purpose processor that outputs fibonacci numbers up to 'n' places. Start with a function computing desired result, translate it into a state diagram, and sketch a probable datapath. [2075 Baisakh]

Answer: See question number 6 (Old Exam Question Solution TU).

10. Design a custom single purpose processor to read two integers a and b and to calculate a^b showing all the steps involved in detail. [2078 Baisakh]

Answer: Same as question number 8 (Old Exam Question Solution TU).

OLD EXAM QUESTION SOLUTION (PoU)

1. Design a synchronous sequential machine that produces output 1 when input sequence is 1011 using JK flip-flop. [2016 Fall]

Answer:

Truth Table

A	B	C	D	Output (0)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

2. Design a custom single purpose processor to calculate GCD between two integers. [2016 Fall]

Answer: See example in chapter 2.

3. Explain with an example how to optimize custom single purpose processors. [2016 Spring]

Answer: See question number 2 (Old Exam Question Solution TU).

4. Draw the combinational logic design for three inputs a, b and c and two outputs y and z. The output 'y' is such that y is 1 if a is 1 or b and c is 1 and = 1 if b or c is 1 but not both. [2016 Spring]

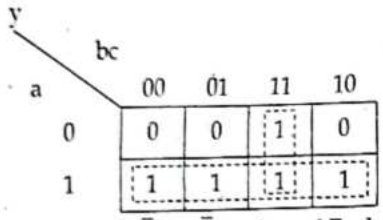
Answer:

Truth Table

Input			Output	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

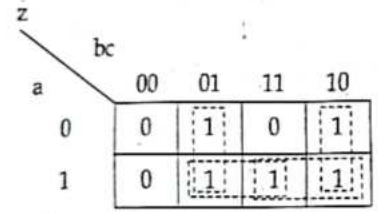
K-map reduction



$y = bc + a\bar{b}\bar{c} + a\bar{b}c + abc + ab\bar{c} = bc + a\bar{b}[\bar{c} + c] + ab[c + \bar{c}]$

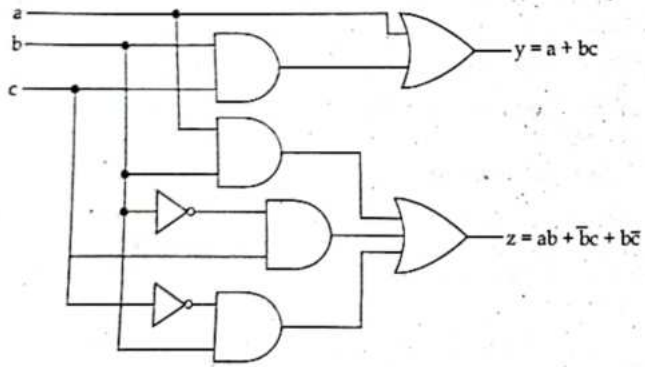
$= bc + a\bar{b} + ab = bc + a[\bar{b} + b] = bc + a$

$y = a + bc$



$z = \bar{b}c + ab + b\bar{c} = ab + \bar{b}c + b\bar{c}$

Logic Diagram



5. What is optimization? Explain optimization of single purpose processor in detail. [2017 Spring]

Answer: Same as question number 2 (Old Exam Question Solution TU).

6. Design a processor that calculates the GCD of two numbers. Show the design of datapath only and construct the diagram of controller.[2017 Spring]

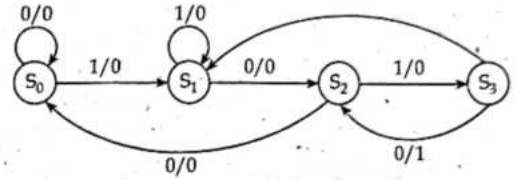
Answer: See example in chapter 2.

7. Design a custom single purpose processor that generates Fibonacci series up to n places. Start with a function computing the desired result, translate it into a state diagram, and sketch a probable datapath. [2018 Spring]

Answer: See question number 6(Old Exam Question Solution TU)

8. Design an overlapping sequence detector for the sequence 1010. [2019 Fall]

Answer:



State table;

Present state	Next State		Output	
	X = 0	X = 1	X = 0	X = 1
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_3	0	0
S_3	S_2	S_1	1	1

State assignments

Let, $S_0S_0 = 00$, $S_1S_1 = 01$, $S_2S_2 = 10$, $S_3S_3 = 11$

Then,

Present state	Next state		Output	
	X = 0	X = 1	X = 0	X = 1
00	00	01	0	0
01	10	01	0	0
10	00	11	0	0
11	10	11	1	0

Four states will require 2 flip flops. Consider 2 D flip flops.

Excitation table

Present state		Input	Next state		F/F inputs		Output
A	B	X	A	B	D_1	D_2	Z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	1	0	1	0	1	0	1
1	1	1	1	1	1	1	0

k-maps to determine inputs to D flipflop

A \ B	00	01	11	10
0	0	0	0	1
1	0	1	1	1

$D_1 = AX + B\overline{X}$

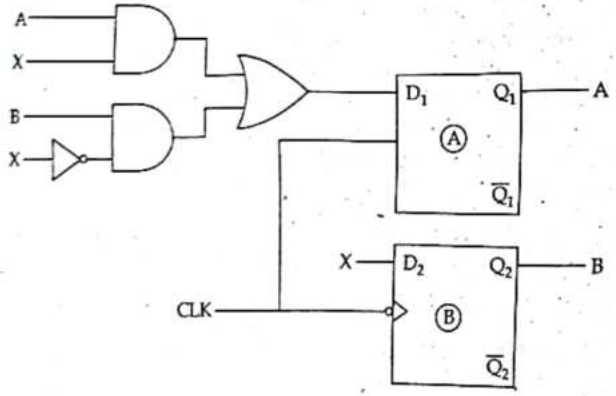
A \ BX	00	01	11	10
0	0	1	1	0
1	0	1	1	0

$D_2 = X$

A \ BX	00	01	11	10
0	0	0	0	0
1	0	0	0	1

$Z = AB\overline{X}$

Circuit diagram for the sequence detector



Chapter 3

SOFTWARE DESIGN ISSUES

★★★

★★★

A general purpose processor is a programmable digital system intended to solve computation problems in a large variety of applications.

The unit cost of the processor may be very low, because the manufacturer spreads NRE over large number of units. Motorola sold half a billion 68 HC 05 microcontroller in 1996 alone.

Since, processor manufacturer can spread NRE cost over large number of units; the manufacturer can afford to invest large NRE cost into the processor design, without significantly increasing the unit cost.

3.1 BASIC ARCHITECTURE

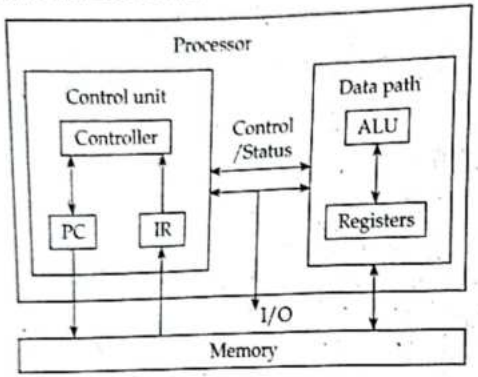


Figure: Geheral purpose processor

The general purpose processor sometimes called as central processing unit (CPU) or a microprocessor. It consists of a data path and a control unit, tightly coupled with memory as shown in figure above. The general purpose basic architecture consists of:

- Data path
- Control unit
- Memory

Data Path

It consists of circuitry for transforming data and for storing temporary data. The data path contains an arithmetic logic unit (ALU) capable of transforming data through operation like addition, subtraction, logical AND, logical OR inverting, shifting etc.

Storing of temporary data is done by using registers. The temporary data includes.

- Data read from memory but not yet send to ALU.
- Result from ALU operation that is used for next operation or going to write on memory.

The capacity of processing measure by bandwidth of data path *i.e.,* data carrying capacity. The n-bit size processor consists:

- N bit registers set
- N bit internal and external system bus
- N bit ALU

Control Unit

Control unit consists of circuitry for retrieving program instructions and for moving data to/from and through the data path according to those instructions. A register called as program counter (pc) that is use to

sequence the program *i.e.,* it points address of next instruction that is going to fetch and execute. Another register called as instruction register (IR) is used to hold the instruction read from memory. The register called as address register (AR) is used to store the memory address during memory read/write operation. The control unit has a controller of a status register plus next state and control logic. It controls the data flow on data path and such flow includes

- Inputting the two particular register for ALU operation
- Storing the ALU results in particular register
- Moving the data between memory and register

For each instruction, the controller typically sequence through the following;

- Fetch stage
- Decode stage
- Fetch operand stage
- Execute stage
- Store stage

Memory

Register acts as temporary or short term storage entities. While memory services as processor medium and long-term storage requirements. Memory can be classified as:

- Program memory
- Data memory

Program information consists of the sequence of instruction that cause the processor to carry out the desired system functionality. Data information represents the value being i/p, o/p and transformed by the program. Both program and data can be stored together or separately. The memory architecture follows the following two model:

- Princeton Architecture (Von-Neumann)
- Harvard Architecture

Princeton Architecture

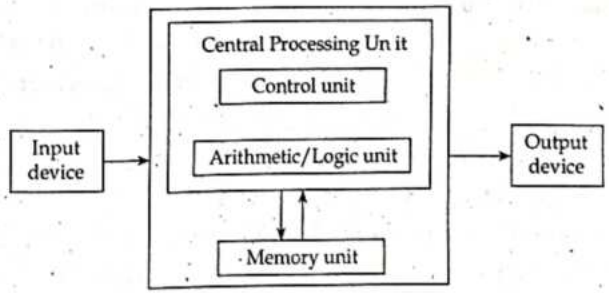


Figure: Von-Neumann architecture

- In Princeton architecture both program and data share the same memory space
- Since single bus for both data and instruction architecture is much simpler.

Harvard Architecture

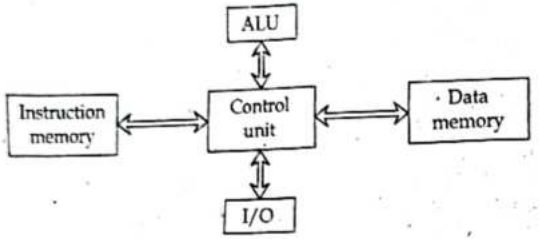


Figure: Harvard architecture

- Program memory space is kept distinct from data memory space.
- Harvard architecture can perform both data instruction fetch simultaneously.
- 8051/52 microcontroller has Harvard architecture.

3.2 OPERATION

Instruction Execution

Microprocessor instruction execution consists of following basic stages:

- **Fetch Instruction (FI):** It is a task that reads the instruction from memory pointed by pc and loaded into the instruction register.
- **Decode Instruction (DI):** In this phase, an instruction be decoded to separate the operand reference as well as operation code to represent the particular operation as ADD, MOV, AND, SUB etc.
- **Fetch Operand (FO):** In this stage, the operand be read from the memory represented by the effective address (EA). EA calculation is needed for indirect address.
- **Execute Instruction (EI):** In this phase, the instruction be executed in accordance with its opcode an operand and generates the result.
- ***Store result* (SR):** The result is stored on the particular destination that may be register or memory.

Pipeline for Instruction Execution

Instruction pipelining is a technique that implements a form of parallelism called instruction level parallelism within a single processor, thus allows faster CPU throughput. Rather than processing each instruction sequentially, each instruction is split up into a sequence of steps, so different instruction can be executed parallel and can be processed concurrently.

The pipelining is easily understands by taking the example of washing and drying eight dishes as below;

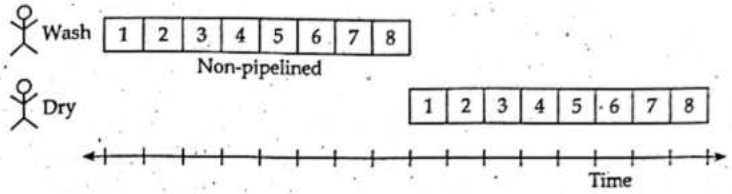


Figure: (a) Non-pipelined dish cleaning

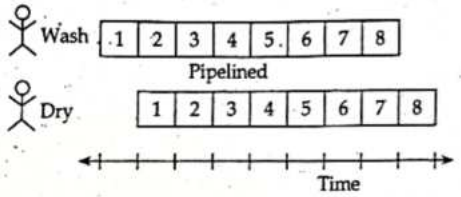


Figure: (b) Pipelined dish cleaning

In the first approach the first person washes all the dishes and second person dries all the dishes. Assuming 1 minute per dish per person, this approach requires 16 minutes. The better approach is that where a person washes and another goes to drying immediately such that this task is finished at 9 minutes.

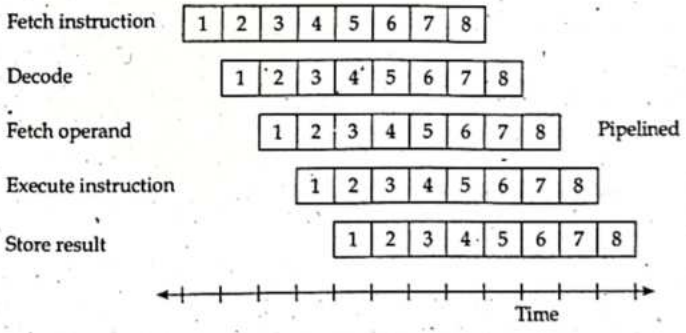


Figure: (c) Pipelined instruction execution

3.3 PROGRAMMER'S VIEW

A programmer writes the program instruction carryout the desired functionality on GPP. A programmer does not need to know the detailed information about the processor architecture, but instead need to know what instructions can be executed.

There are three levels of programming:

i) Assembly level
ii) Structure level
iii) Machine level

I) Assembly level

In assembly language, program is written using processor specific instruction. An assembly language is a low-level programming language for microprocessors and other programmable devices. It is not just a single language, but rather a group of languages. An assembly language implements a symbolic representation of the machine code needed to program a given CPU architecture. Assembly language is also known as assembly code. The term is often also used synonymously with 2GL. The assembler is needed to convert the assembly code into machine code.

II) Structure level

In structured level, programming language like C/C++, java are preferred, which are processor independent instruction. Structured programming is a logical programming method that is considered a precursor to object oriented programming (oop). Structured programming facilitates program understanding and modification and has a top-down design approach, where a system is divided into compositional subsystems. These languages are machine independent and need a compiler to convert them into machine code.

III) Machine level

Machine level is lower than assembly level. Program instructions are written in binary format. Such programs are tedious and are rarely used. Due to invention of assembler (which translates assembly language into machine language) program are not written at machine level.

3.3.1 Programmer's Consideration

i) *Program and data memory space*
ii) *Registers*
iii) *Input output (I/O)*
iv) *Interrupts*
v) Operating system

I) Program and data memory space

- In embedded system design, the programmer must be aware of the program and data size limits.
- Programs must be written to fit inside the memory limitations.

- *For example*, on-chip memory for program and data is fixed in microcontrollers.
- As a result, in order to not exceed the RAM limit, the code must be written effectively.

II) Registers

- The amount of registers available for general and specific purposes must be known to the embedded system design programmer.
- *For example*, multiplication in 8051 microcontroller can be accomplished using the accumulator and register B.
- The structured language programmer does not need to know about the accumulator or register B.
- Every programmer, however, must be familiar with the many special function registers used to configure timers, serial connection, and interrupts.

III) Input output (I/O)

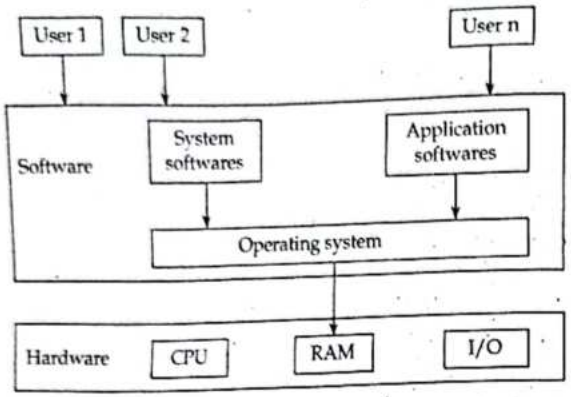
- Every processor has input output pins that allow the programmer to communicate with external devices.
- When working with processors, programmers must be aware of the number of input and output pins available, as well as their purposes.
- A port can be read or written to in parallel I/O using a specified function register.
- Additionally, communications can be accomplished through the system bus, which allows for the activation of address and data ports in response to specific instructions.

IV) Interrupts

- Interrupt is a feature that allows the processor to serve a device that requires immediate attention.
- It causes the processor to halt the current program's execution and launch an interrupt service routine that performs the function required by the device that interrupts the processor.
- The programmer should be aware of the sorts of interrupts that the processor supports and, where necessary, build an interrupt service procedure.

V) Operating System

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.



Function of Operating System

Operating system performs three functions:

i) **Convenience:** As OS makes a computer more convenient to use.

ii) **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.

iii) **Ability to Evolve:** As OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions at the same time without interfering with service.

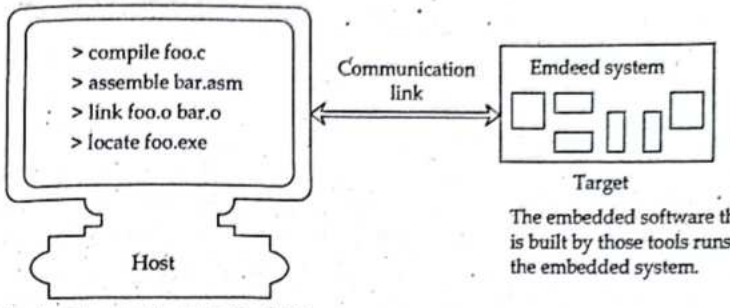
An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling pheripherial devices such as disk drives and printers.

The system call provides an interface to the operating system services. Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system call. By using the API, certain benefits can be gained.

- **Portability:** as long a system supports an API, any program using that API can compile and run.
- **Ease of use:** using the API can be significantly easier than using the actual system call.

3.4 DEVELOPMENT ENVIRONMENT

Development Environment comprises of general software design tools that are used by embedded system designer in system design, system testing, debugging and verification of embedded software. The software be developed in general processor called as development processor then it is burned go the target embedded processor.



Performance of Host Machine

The application program developed runs on the host computer. The host computer is also called as development platform. It is a general purpose computer. It has a higher capability processor and more memory. It has different input and output devices. The compiler, assembler linker, and locator run on a host computer rather than on the embedded system itself. These tools are extremely popular with embedded software developers because they are freely available (even the source code is free) and support many of the output binary image. Once a program has been written, complied, assembled and linked, it is moved to the target platform.

Performance of Target Machine

The output binary image is executed on the target hardware platform. It consists of two entities-the target hardware (processor) and runtime environment (OS). It is needed only for final output. It is different from the development platform and it does not contain any development tools.

 # IDE (Integrated Development Environment)

It is a programming environment that contains a source code editor, a compiler, a linker/locator and usually a debugger. It is actually a software application that provides comprehensive facilities to computer programmers for software development. This type of environment allows an application developer to write code while compiling, debugging and executing it at the same place. But more often the IDEs support multiple languages, processors, etc. Some commonly used IDEs for embedded systems are the GNU compiler collection (gcc), eclipse, Delphi.

Editor

A source code editor is a text editor program designed specifically for editing source code to control embedded systems. It may be a standalone application or it may be built into an integrated development

environment. Source code editors may have features specifically designed to simplify and speed up input of source code, such as syntax highlighting and auto complete functionality. These features ease the development of code.

Compiler

A compiler is a computer program that translates the source code into computer language (object code). Commonly the output has a form suitable for processing by other programs (*e.g.*, a linker), but it may be a human readable text file. A compiler translates source code from a high level language to a lower level language (*e.g.*, assembly language or machine language). The most common reason for wanting to translate source code is to create a program that can be executed on a computer or on an embedded system. The compiler is called a cross complier if the source code is compiled to turn on a platform other than the one one which the cross compiler is run. For embedded systems the compiler always run on another platform, so a cross compiler is needed.

Linker

A linker or link editor is a program that takes one or more objects generated by compilers and assembles them into a single executable program or a library that can later be linked to in itself. All of the object flies resulting from compiling must be combined in a special way before the program locator will produce an output file that contains a binary image that can be loaded into the target ROM. A commonly used linker/locator for embedded system isld (GNU).

Debugger

A debugger is a computer program that is used to test and debug other programs. A debugger is a piece of software running on the PC, which has to be tightly integrated with the emulator that you use to validate your code. A debugger allows you to download your code to the emulator's memory and then control all of the functions of the emulator from a PC.

Simulators

A simulator try to model the behavior of the complete microcontroller in software. Some simulators go even a step further and include the whole system (simulation of pheripherials outside of the microcontroller). No matter how fast your PC, there is no simulator on the market that can actually simulate a microcontroller's behavior in real time. Simulating external events can become a time consuming exercise, as you have to manually create "stimulus" flies that tell the simulator what external waveforms to expect on which microcontroller pin. A simulator can also not talk to your target system, so functions that rely on external components are difficult to verify. For that reason simulators are best suited to test algorithms that run completely within the microcontroller.

Emulator

An emulator is a piece of hardware that ideally behaves exactly like the real microcontroller chip with all its integrated functionality. It is the most powerful debugging tool of all. A microcontroller's Functions are emulated in real time. Because, depending on memory technology, a microcontroller's program memory cannot (ROM) or only once (OTP) be programmed, an emulator uses external static RAM as the emulated micro's program memory. Even some flash based microcontroller can, depdending on manufacturer, only be reprogrammed 100 to 1000 times, which warrants the use of external RAM memory rather than the micro's integrated flash for emulation. RAM memory allows for code to be changed quickly and an "indefinite" number of times during the software debugging process. Bond-out chips have additional pins that allows the emulator electronics to feed the externally stored program information to the microcontroller in place of the on-chip memory contents in real time; control the program execution flow; and access on-chip registers and data memory.

3.5 APPLICATION SPECIFIC INSTRUCTION SET PROCESSORS

Application Specific Instruction Set processors (ASIP). Programmable microprocessor where hardware and instruction set are designed together for one special application. Application specific integrated circuit (ASIC). Algorithm completely implemented in hardware. Application specific systems some of the typical approaches of building an application specific system or an embedded system is to use one or more of the following implementation methods: GPP, ASIC or ASIP. GPP functionality of the system is exclusively build on the software level. Although the biggest advance of such system is the flexibility but it is not optimal in term of performance, power consumption, cost, physical space and heat dissipation. An application-specific instruction set processor (ASIP) is a component used in system-on-a-chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC.

- Customized to perform particularly well in a particular application area.
- Can improve performance for particular problem instances while maintaining the flexibility of the overall system.
  - Motivated by application-specific nature of embedded system.

Problems

- Substantial non-recurring engineering costs
  - Each new ASIP must be verified both from the functionality and timing perspectives.
  - A new mask set must be created to fabricate the chip.
  - Software side: the compiler must be retargeted to each new processor. Any hand-written libraries must be migrated to new platform.
- Automation of some of these tasks may be possible.
  - However, the majority of this work is still a manual process.
- Difficult to adopt a new ASIP despite the potential advantages.

Advantages

- System is post-programmable and can tolerate modest changes to the application (little performance degradation)
  - *For example,* changes in standard
- Computation intensive portions of applications from the same domain (*e.g.*, encryption) are often similar in structure.
  - Customized instructions can often be generalized in small ways to make them more useful across a set of applications.
  - Lowers the cost than ASIC.

3.6 SELECTING A MICROPROCESSOR

With numerous kinds of processors with various design philosophies available at our disposal for using in our design, following considerations need to be factored during processor selection for an embedded system.

I) Performance Considerations

The first and foremost consideration in selecting the processor is its performance. The performance speed of a processor is dependent primarily on its architecture and its silicon design. Presence of cache reduces data/instruction fetch timing. Pipelining and super-scalar architectures further improves the performance of the processor. So, size of cache, processor architecture, instruction set etc has to be taken into account when comparing the performance.

II) Power Consideration

Increasing the logic density and clock speed has adverse impact on power requirement of the processor. A higher clock implies faster charge and discharge cycles leading to more power consumption. More logic leads to higher power density there by making the heat dissipation difficult.

Further with more emphasis on greener technologies and many systems becoming battery operated, it is important the design is for optimal power usage.

III) Peripheral Set

Every system design needs, apart from the processor, many other peripherals for input and output operations. Since in an embedded system, almost all the processors used are SoCs, it is better if the necessary peripherals are available in the chip itself. This offers various benefits compared to peripherals in external IC's such as optimal power architecture, effective data communication using DMA, lower BoM etc. So it is important to have peripheral set in consideration when selecting the processor.

IV) Operating Voltages

Each and every processor will have its own operating voltage condition. The operating voltage maximum and minimum rating will be provided in the respective data sheet or user manual.

V) Specialized processing

Apart from the core, presence of various co-processors and specialized processing units can help achieving necessary processing performance. Co-processors execute the instructions fetched by the primary processor thereby reducing the load on the primary.

VI) Digital Signal processors

DSP is a processor designed specifically for signal processing applications. Its architecture supports processing of multiple data in parallel. It can manipulate real time signal and convert to other domains for processing. DSP's are either available as the part of the SoC or separate in an external package. DSP's are very helpful in multimedia applications. It is possible to use a DSP along with a processor or use the DSP as the main processor itself.

VII) Price

Various consideration discussed above can be taken into account when a processor is being selected for an embedded design. It is better to have some extra buffer in processing capacities to enable enhancements in functionality without going for a major change in the design. While engineers will want to have all the functionalities, price will be the determining factor when designing the system and choosing the right processor.

3.7 GENERAL-PURPOSE PROCESSOR DESIGN

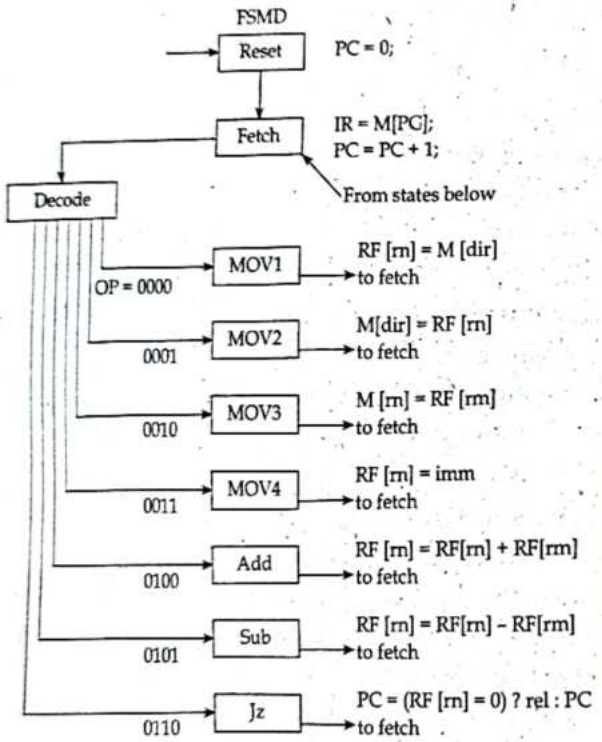
Not something an embedded system designer normally would do

- But instructive to see how simply we can build one top down
- Remember that real processors aren't usually build this way
  - Much more optimized, much more bottom up design

Declarations:

bit PC [16], IR [16];

bit M [64 k] [16], RF [16] [16];

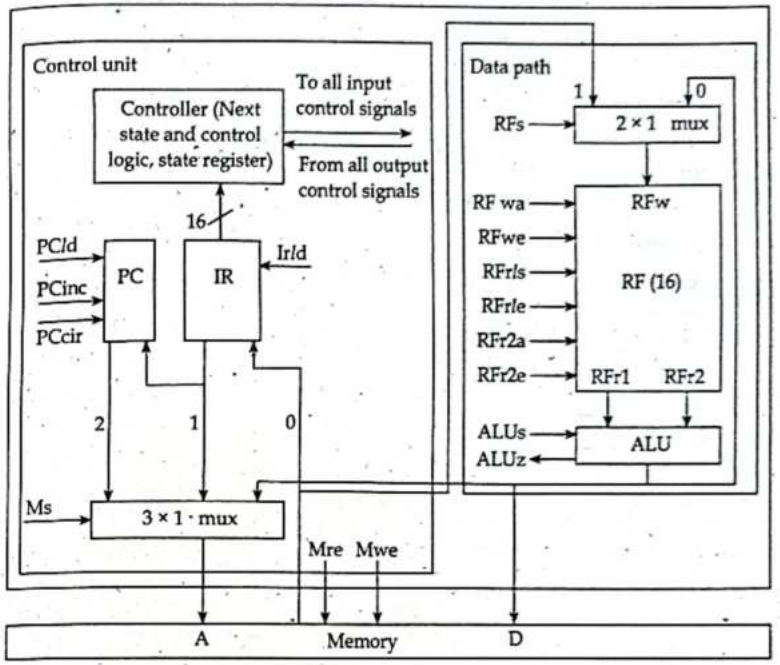


Aliases:
OP IR [15.....12] dir IR[7.....0]
rn IR[11.....8] imm IR[7.....0]
rm IR[7.....4] rel IR [7.....0]

Architecture of a simple microprocessor

- Storage devices for each declared variable
  - Register file holds each of the variables
- Functional units to carry out the FSMD-operations
  - One ALU carries out every required operation

- Connections added among the component's ports corresponding to the operations required by the FSM.
- Unique identifiers created for every control signal.



We now create FSM from FSMD.

Each FSMD operation is replaced by binary operation on control signal.

Finally we design controller using combinational and sequential logic.

PCclr =1; (Program counter clear)

MS = 10; (Memory select)

Irld = 1; (Instruction register load)

Mre = 1; (Memory read enable)

PC inc = 1; (Program counter enable)

OLD EXAM QUESTION SOLUTION (TU)

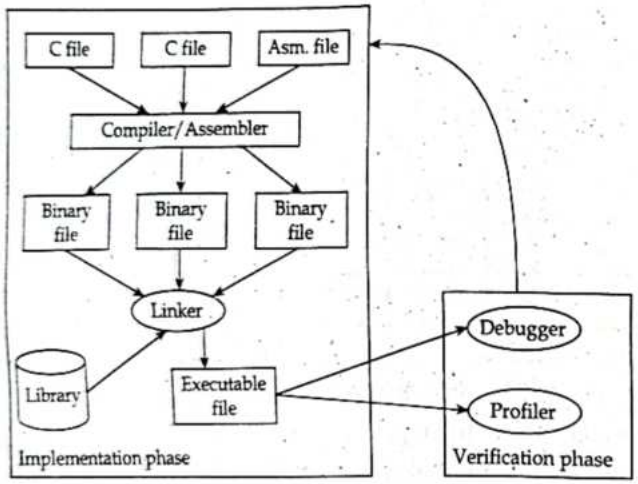
1. What are the programmer considerations? Explain the software development processes according to embedded systems. [2069 Bhadra]

Answer:

Programmer Considerations are;

i) Program and data memory space
ii) Registers
iii) Input output (I/O)
iv) Interrupts
v) Operating system

Software development process



The development processor and the target processor for software development may be the same. The development tools are also available as part of a single package known as the "Integrated Development Environment (IDE).

a) Implementation phase

An editor is used to write source code, which is then compiled/assembled using compiler/assembler. Finally, the linker is used to join all of the essential files into a single executable file.

b) Verification phase

A debugger controls the execution of the executable file. To check the behavior of the software, all possible inputs are used, particularly boundary conditions. The program's performance can be analyzed using profilers. It is possible to study time and space complexity. The duration of a program's execution is measured in time complexity, whereas memory utilization is measured in space complexity.

2. Describe the operation of general-purpose processor in terms of datapath and controller. [2070 Bhadra, 2078 Baisakh]

Answer: See the topic 3.2 of chapter 3.

3. Explain the testing and debugger. [2070 Bhadra]

Answer:

Testing

Testing is the process of verifying that a software or application is bug-free, meets technical criteria as defined by its design and development, and meets user needs effectively and efficiently while dealing with all exceptional and boundary instances.

Debugger

A debugger is a computer program that is used to test and debug other programs. A debugger is a piece of software running on the PC, which has to be tightly integrated with the emulator that you use to validate your code. A debugger allows you to download your code to the emulator's memory and then control all of the functions of the emulator from a PC.

4. Differentiate between application specific instruction set-processor and general purpose processor. Also discuss on issues related to selection of particular processor. [2070 Magh]

Answer:

Application specific instruction set processor	General purpose processor
1. ASIP is application dependent instruction processor, and it is used to process numerous instruction sets within a combinational circuit of an embedded system.	1. A general purpose processor is a programmable digital system intended to solve computation problems in a large variety of applications.
2. Performance is high	2. Performance is low
3. Flexibility is good	3. Flexibility is excellent
4. Efficiency is high	4. Efficiency is less
5. Examples of applical specific processor are embedded microcontroller, network processor and DSP.	5. Example of general purpose processor is pentium.

Second part: See the topic 3.6 of chapter 3.

5. What are the difference between datapath and control unit? Explain the control unit sub-operation. [2071 Bhadra]

Answer:

Datapath

- It consists of circuitry for transforming data and for storing temporary data.

- Datapath is general
- Datapath operations:
  - Load
  - ALU operation
  - Store

Control unit

- Configures the datapath operations.
- Control unit doesn't store the algorithm-the algorithm is "programmed" into the memory.

Control unit consists of circutry for retrieving program instructions and for moving data to/from and through the datapath according to those instructions. Control unit sub operation are;

i) Fetch
ii) Decode
iii) Fetch operand
iv) Execute
v) Store

i) Fetch

It retrieves the next instructions from the PC-pointed place in memory and puts them in IR.

- Get next instruction into IR
- PC: Program counter, always points to next instruction
- IR: holds the fetched information

ii) Decode

This defines what operations the instructions perform at IR

- Determine what the instruction means

iii) Fetch operand

This function fetches the operand data required for an instruction from memory and places it in the proper datapath registers.

- Move data from memory to datapath register

iv) Execute

Moving data through an ALU performs the real arithmetic or logical computations.

- Move data through the ALU
- This particular instruction does nothing during this sub-operation

v) Store

It reads data from the datapath register and writes it to memory

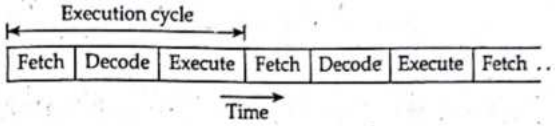
- Write data from register to memory
- This particular instruction does nothing during this sub-operation

6. Describe the control unit sub-operations with its instruction cycles. [2071 Magh]

Answer:

Same as question number 5 second part (Old exam Question Solution TU)

Instruction cycle



Six fundamental phases of the instruction cycle

i) Fetch instruction (pre fetch)
ii) Decode instruction
iii) Evaluate address (address generation)
iv) Fetch operands (read memory data)
v) Execute (ALU access)
vi) Store result (write back memory data)

7. Briefly explain the criterion for selecting processor? Explain the data path operation and its instruction cycles. [2072 Ashwin]

Answer:

Criterias for selecting processor are;

A designer must choose a microprocessor for an embedded system based on both technical and non-technical factors.

i) Technical factors

The processor must be chosen depending on the desired speed within the constraints of power, size, and cost.

ii) Non-technical factors

Before choosing a microprocessor, consider the development environment, the processor's prior expertise, license agreements, and other factors.

Datapath consists of circuitry for transforming data and for storing temporary data. Datapath operations are;

i) Load

- Read memory location into register

ii) ALU operation

- Input certain registers through ALU, store back in register

iii) Store

- Write register to memoty location

8. Define addressing mode and pipelining. Explain data path operation with suitable example. [2072 Magh]

Answer:

The term addressing mode relates to the manner an instruction's operand is expressed. Before the operand is actually performed, the addressing

mode provides a rule for interpreting or altering the address field of the instruction.

Instruction pipelining is a technique that implements a form of parallelism called instruction level parallelism within a single processor, thus allows faster CPU throughout.

Datapath Operation: Same as question number 7 (Old Exam Question Solution TU).

9. How general purpose processors differ from application specific instruction set processor (ASIPs)? What are the common features of ASIPs for digital signal processors? [2073 Bhadra]

Answer:

First part: Same as question number 4 (old exam question solution TU)

Features of ASIPs for digital signal processors are;

i) There could be a lot of register files, memory blocks, multipliers, and other arithmetic units in it.

ii) It helps by providing instruction that are specific to digital signal processing. Filtering and vector transformation are two examples.

iii) Hardware is utilized to implement commonly used mathematical functions. When compared to software implementation, it results in faster arithmetic function execution.

iv) Concurrent execution of functions is possible with some particular digital signal processors, which improves system performance.

v) It includes a number of signal-processing-specific peripherals. ADC, DAC, PWN, DMA controllers, timers, and counters are all possible components.

10. Explain the design flow of embedded software development. Explain in brief about programmer's view for general purpose processor. [2073 Magh]

Answer:

First part: Same as question number 1 (Old Exam Question Solution TU)

Second part: See the topic 3.3 of chapter 3.

11. Explain the design flow of embedded software development. Explain in brief about programmers view for general purpose processor. [2074 Bhadra]

Answer: See question number 10 (Old Exam Question Solution TU)

12. Define pipelining and show 6 stage pipeline concept. Explain DSP with characteristics and advantages. [2075 Baisakh]

Answer:

First part: See the topic 3.2 (pipeline for instruction execution) of chapter 3.

Second part:

These are processors designed for large-scale data processing. Images taken by a camera, voice packets sent over a network router, and audio clips played by an instrument are all examples of data sources that generate vast amounts of data.

Some features are;

i) There could be a lot of register files, memory blocks, multipliers, and other arithmetic units in it.

ii) It helps by providing instructions that are specific to digital signal processing. Filtering and vector transformation are two examples.

iii) Hardware is utilized to implement commonly used mathematical functions. When compared to software implementation, it results in faster arithmetic function execution.

iv) Concurrent execution of functions is possible with some particular digital signal processors, which improves system performance.

v) It includes a number of signal processing specific peripherals. ADC, DAC, PWN, DMA controllers, timers and counters are all possible components.

Advantages:

i) The digital system can be cascaded in DSP without any concerns with loading. In this way, digital circuits can be easily replicated in huge quantities at cheaper cost.

ii) Digital circuits are less affected by component value tolerance. Because the digital signals can be processed offline, they are conveniently transportable.

iii) The program of a digital programmable system can be modified to change the digital signal processing activities.

iv) In comparison to analog systems, digital systems have better accuracy control.

v) The DSP approach can be used to create sophisticated signal processing algorithms.

vi) Digital signals may be easily stored on magnetic media such as magnetic tape without sacrificing signal quality.

13. Explain the design flow of embedded software development. Explain in brief about programmer's view for general purpose processor. [2076 Bhadra]

Answer: Same as Question of 2073 Magh TU.

14. Explain datapath and controller in a general purpose processor. Write software development environment used in embedded system. [2077 Poush]

Answer:

First part: See the topic 3.1 of chapter 3.

Second part: See the topic 3.4 of chapter 3.

OLD EXAM QUESTION SOLUTION (PoU)

1. Explain general purpose processor design with a suitable diagram.
[2016 Fall] [2018 Spring]

Answer: See the topic 3.1 of chapter 3.

2. How does a programmer view a microprocessor based embedded system? What are his/her concern?
[2016 Spring, 2017 Fall, 2017 Spring, 2018 Fall]

Answer: See the topic 3.3 of chapter 3.

3. Explain the firmware development process with necessary block diagram?
[2019 Fall]

Answer: Same as question number 1 (old exam question solution TU).

Chapter 4

MEMORY

★★★

★★★

4.1 EMBEDDED SYSTEM'S FUNCTIONALITY ASPECTS

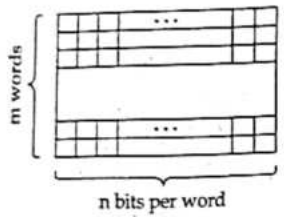
- Processing
  - Processor
  - Transformation of data
- Storage
  - Memory
  - Retention of data
- Communication
  - Buses
  - Transfer of data

4.2 MEMORY

Stores large number of bits

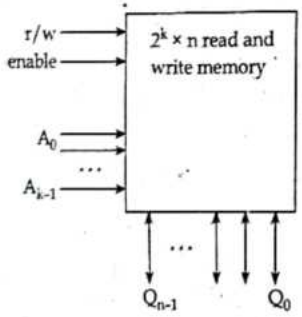
- m × n: m words of n bits each
- $K = \log_2 (m)$ address input signals
- Or m = 2^k words
- *For example,* 4096 × 8 memory
  - 32768 bits
  - 12 address input signals
  - 8 input/output data signals

m × n memory



- Memory access
  - r/w: selects read or write
  - enable: read or write only when asserted
  - multiport: multiple accesses to different locations simultaneously.

Memory external view



4.3 MEMORY WRITE ABILITY AND STORAGE PERMANENCE

Traditional ROM/RAM distinctions

- ROM
  - Read only, bits stored without power
- RAM
  - Read and write, lose stored bits without power

Traditional distinctions blurred

- Advanced ROMs can be written to
  - *For example;* EEPROM
- Advanced RAMs can hold bits without power
  - For example, NVRAM

Write ability

- Manner and speed a memory can be written

Storage permanence

- Ability of memory to hold stored bits after they are written

4.3.1 Write Ability

Write ability refers to the manner and speed that a particular memory can be written. Basically we have four levels of write ability.

i) High end
- Processor writes to memory simply and quickly
- For example, RAM

ii) Middle range
- Processor writes to memory, but slower
- *For example,* FLASH, EEPROM

iii) Lower range
- Special equipment, "programmer", must be used to write to memory
- *For example,* EPROM, OTP ROM

iv) Low end
- Bits stored only during fabrication
- *For example,* Mask-programmed ROM

The term "in system programmable" is used to divide memories into two categories along the write ability axis. They are:

- In system programmable
- Non in system programmable

In system programmable

The ability of some programmable logic devices, microcontroller and other embedded devices to be programmed while installed in a computer system rather than requiring the chip to be programmed prior to installing it into the system.

Non in system programmable

The chip is removed from the target board and placed in a programmable device to be programmed. Memories in the lower range and lower end fall under this category.

4.3.2 Storage Permanence

It is the ability of memory to hold its stored bits after those bits have been written. On the basis of storage permanence, memory devices can be categorized as:

i) **High end**
- Essentially never loses bits
- *For example*, Mask-programmed ROM

ii) **Middle range**
- Holds bits days, months, or years after memory's power source turned off
- *For example*, NVRAM (non-volatile RAM)

iii) **Lower range**
- Holds bits as long as power supplied to memory
- For example, SRAM

iv) **Low end**
- Begins to lose bits almost immediately after written
- For example, DRAM

Non-volatile memory
- Holds bits after power is no longer supplied
- High end and middle range of storage permanence.

Volatile memory
- Holds bits until the power is supplied.

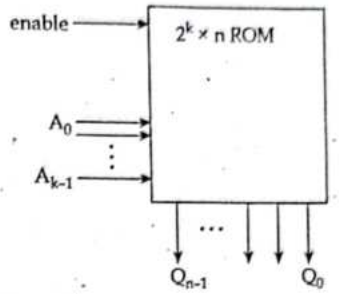
Trade-Offs

Some of the trade-offs are as follows:
- Design metrics compete with each other. (Memory write ability and storage permanence are two such metrices).
- Memory write ability and storage permanence tents to be inversely proportional to each other.
- Highly writeable memory typically requires more area and/or power than less-writable memory

4.4 TYPES OF MEMORY

4.4.1 ROM: Read Only Memory
- Non volatile memory
- Can be read from but not written to, by a processor in an embedded system.
- Traditionally written to, "programmed", before inserting to embedded system.

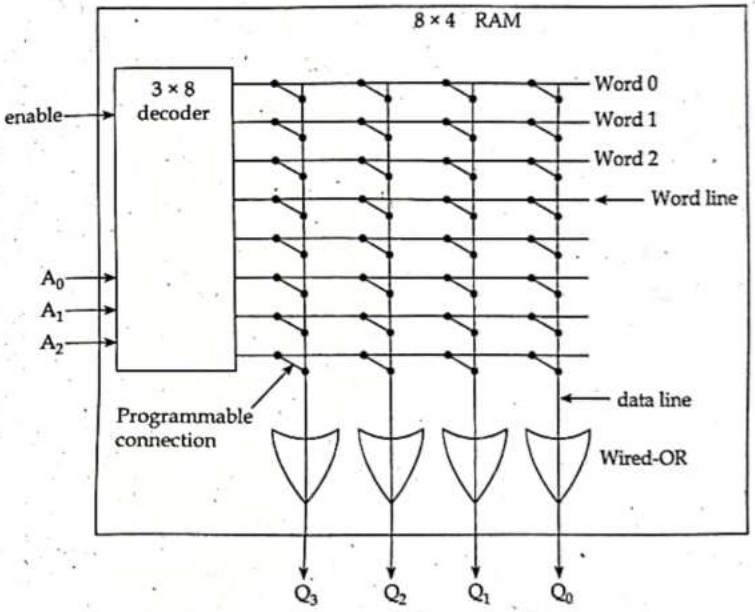
External view



Uses
- Stores software program for general-purpose processor
- Stores constant data needed by system
- Implement combinational circuit.

Example: 8 × 4 ROM

Internal view



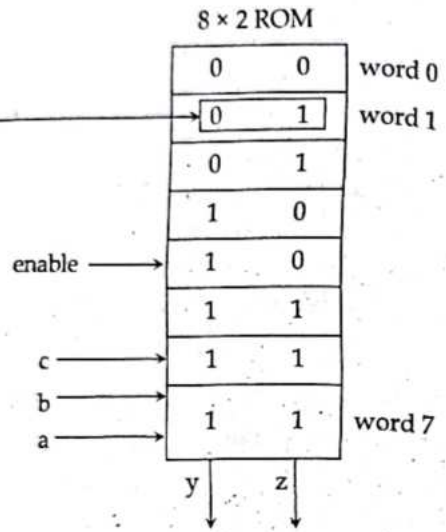
- Horizontal lines = words
- Vertical lines = data
- Lines connected only at circles.
- Decoder sets word 2's line to 1 if address input is 010.
- Data lines Q_3 and Q_1 are set to 1 because there is a "programmed" connection with word 2's line.
- Word 2 is not connected with data lines Q_2 and Q_0.
- Output is 1010.

Implementing Combination Function

Any combinational circuit of n functions of same k variables can be done with $2^k \times n$ ROM

Truth table:

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



4.4.2 Types of ROM

I) **Mask-programmed ROM**

- Connections "programmed" at fabrication
  - Set masks
- Lowest write ability
  - Only once
- High storage performance
  - Bits never change unless damaged
- Typically used for final design of high-volume systems
  - Spread out NRE cost for a low unit cost.

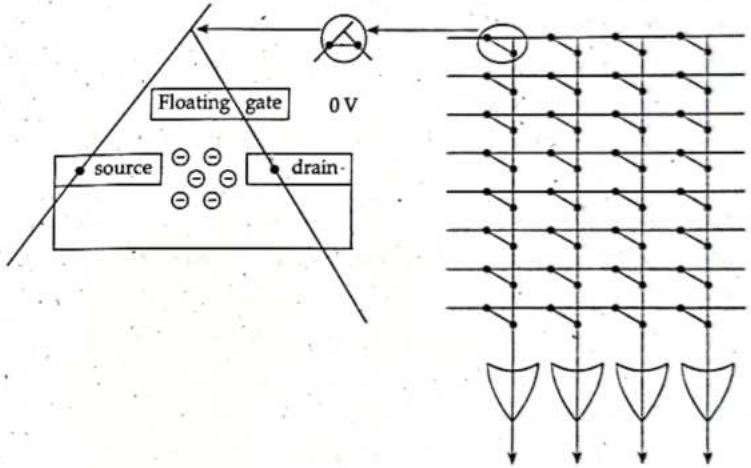
II) **OTP ROM: One-time Programmable ROM**

- Connections "programmed" after manufacture by user.
  - User provides file of desired contents of ROM.
  - File input to machine called ROM programmer.
  - Each programmable connection is a fuse.
  - ROM programmer blows fuses where connections should not exist.

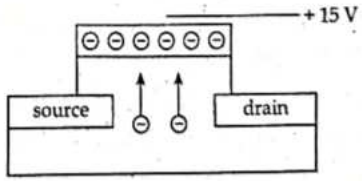
- Very low write ability
  - Typically written only once and requires ROM programmer device.
- Very high storage permanence
  - Bits don't change unless reconnected to programmer and more fuses blown.
- Commonly used in final products
  - Cheaper, harder to inadvertently modify.

III) **EPROM: Erasable Programmable ROM**

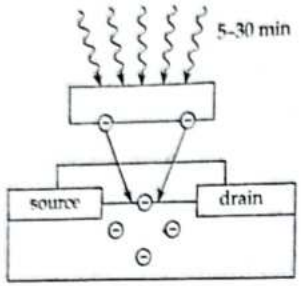
- Programmable component is a MOS transistor
  - Transistor has "floating" gate surrounded by a insulator
  - Negative charges from a channel between source and drain storing a logic 1.



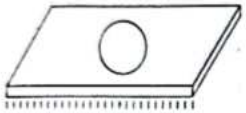
  - Large positive voltage at gate causes negative charges to move out of channel and get trapped in floating gate storing a logic 0.



  - (Erase) shining UV rays on surface of floating-gate causes negative charges to return to channel from floating gate restoring the logic 1.



- An EPROM package showing quartz window through which UV light can pass.



- Better write ability
  - Can be erased and reprogrammed thousands of times
- Reduced storage permanence
  - Program lasts about 10 years but is susceptible to radiation and electric noise
- Typically used during design development

iv) **EEPROM: Electrically Erasable Programmable ROM**

- Programmed and erased electronically
  - Typically by using higher than normal voltage
  - Can program and erase individual words
- Better write ability
  - Can be in-system programmable with built-in circuit to provide higher than normal voltage
    - built in memory controller commonly used to hide details from memory user
  - Writes very slow due to erasing and programming
    - "busy" pin indicates to processor EEPROM still writing
  - Can be erased and programmed tens of thousands of times
- Similar storage permanence to EPROM (about 10 years)
- Far more convenient than EPROMs, but more expensive

v) **Flash Memory**

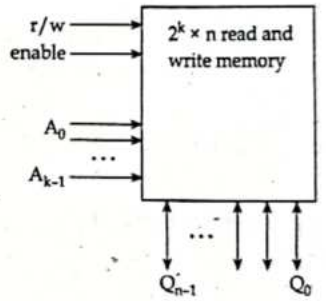
- Extension of EEPROM
  - Same floating gate principle
  - Same write ability and storage permanence

- Fast erase
  - Large blocks of memory erased at once, rather than one word at a time.
  - Blocks typically several thousand bytes large
- Writes to single word may be slower
  - Entire block must be read, word updated, then entire block written back.
- Used with embedded systems storing large data items in non volatile memory.
  - *For example*, Digital cameras, TV set-up boxes, cell phones

4.4.3 RAM: "Random-Access" Memory

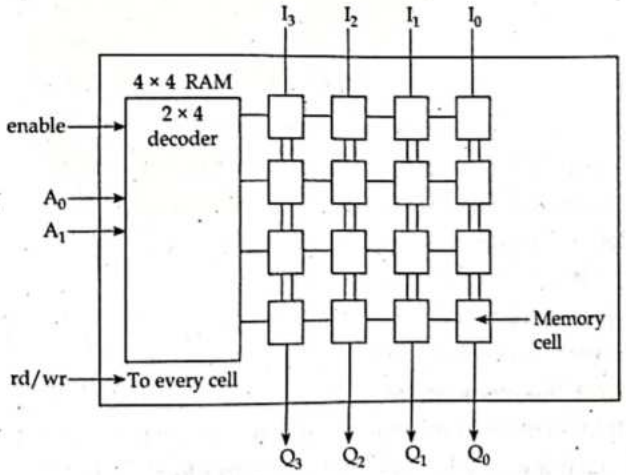
- Typically volatile memory
  - Bits are not held without power supply
- Read and written to easily by embedded system during execution.

External view



- Internal structure more complex than ROM

Internal view



- A word consists of several memory cells, each storing 1 bit.
- Each input and output data line connects to each cell in its column.
- rd/wr connected to every cell.
- When row is enabled by decoder, each cell has logic that stores input data bit when rd/wr indicates write or output stored bit when rd/wr indicates read.

4.4.4 Basic Types of RAM

I) SRAM: Static RAM

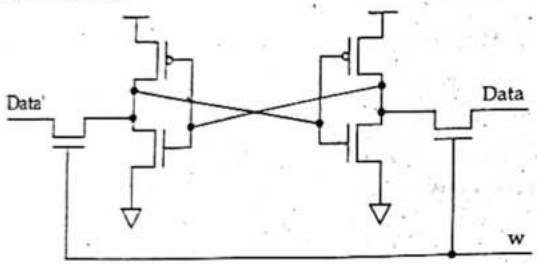


Figure: Memory cell internal of SRAM

- Memory cell uses flip-flop to store bit
- Requires 6 transistors
- Holds data as long as power supplied

II) DRAM: Dynamic RAM

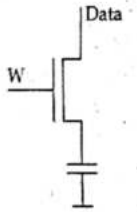


Figure: Memory cell internal of DRAM

- Memory cell uses MOS transistor and capacitor to store bit
- Memory compact than SRAM
- "Refresh" required due to capacitor leak
  - Word's cells refreshed when read
- Typical refresh rate 15.625 micro sec
- Slower to access than SRAM

III) PSRAM: Pseudo-static RAM

- DRAM with built-in memory refresh controller
- Popular low-cost high-density alternative to SRAM

IV) NVRAM: Non-volatile RAM

- Holds data after external power removed
- Battery backed RAM
  - SRAM with own permanently connected battery
  - Writes as fast as reads
  - No limit on number of writes unlike non volatile ROM-based memory
- SRAM with EEPROM or flash
  - Stores complete RAM contents on EEPROM or flash before power turned off.

4.5 COMPOSING MEMORY

- Memory size needed often differs from size from of readily available memories.
- When available memory is larger, simply ignore unneeded high-order address bits and higher data lines.
- When available memory is smaller, compose several smaller memories into one larger memory
  - Connect side-by-side to increase width of word.
  - Connect top to bottom to increase number of words.
  - Combine techniques to increase number and width of words.

I) Increase width of words

The available memory has correct number of words but each word is not wide enough. In such case we connect memories side by side.

For example, We need ROM that is 3-times wider than what is available, here we connect 3-ROMs side by side sharing the same address and enable lines and concatening data lines to form desired word width.

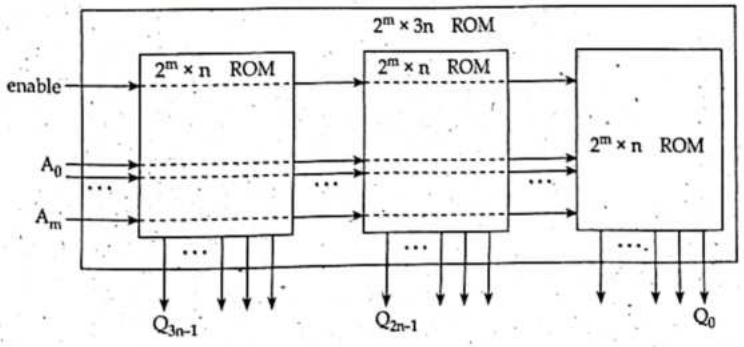


Figure: Increase width of words

ii) Increase number of words

Available memories have correct words width but not enough words.

- In such case we connect available memories top to bottom. *E.g.*, we need a ROM with twice as many words than what is available; here we connect the ROMs, top to bottom, ORing the corresponding data lines of each.
- To select between two ROMs, we need extra address line. 1 × 2 decoder can select between two ROMs.

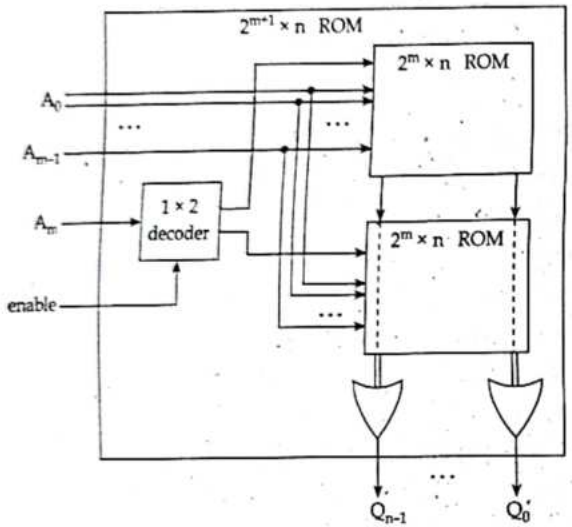


Figure: Increase number of words

iii) Increase number of words and width of words

The available memory has smaller word width and fewer word than needed. So we combine technique 1 and 2.

- First we create number of column of memories necessary to achieve needed word width secondly we create number of rows of memories necessary along a decoder to achieve needed number of words.

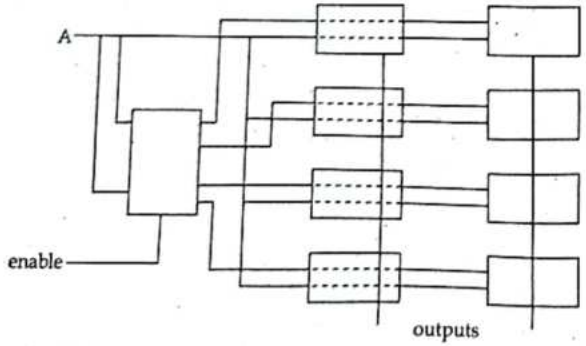
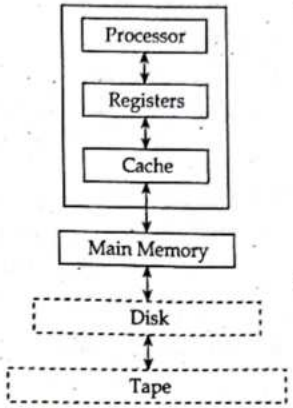


Figure: Increase number of words and width of words

4.6 MEMORY HIERARCHY AND CACHE

4.6.1 Memory Hierarchy



- Want inexpensive, fast memory
- Main memory
  - Large, inexpensive, slow memory stores entire program and data.
- Cache
  - Small, expensive, fast memory stores copy of likely accessed parts of larger memory.
  - Can be multiple levels of cache.

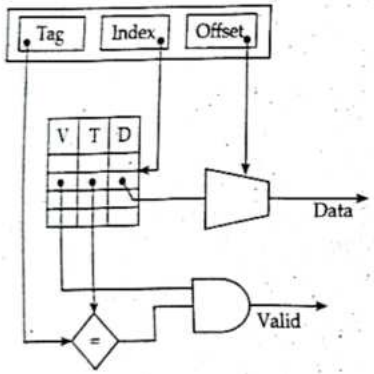
4.6.2 Cache

- Usually designed with SRAM
  - Faster but more expensive than DRAM
- Usually on same chip as processor
  - Space limited, so much smaller than off-chip main memory
  - Faster access (1 cycle vs. several cycles for main memory)
- Cache operation
  - Request for main memory (read or write)
  - First, check cache for copy
    - Cache hit
      - Copy is in cache, quick access
    - Cache miss
      - Copy not in cache, read address and possibly its neighbors into cache.
- Several cache design choices
  - Cache mapping, replacement policies and write techniques.

1. Cache Mapping

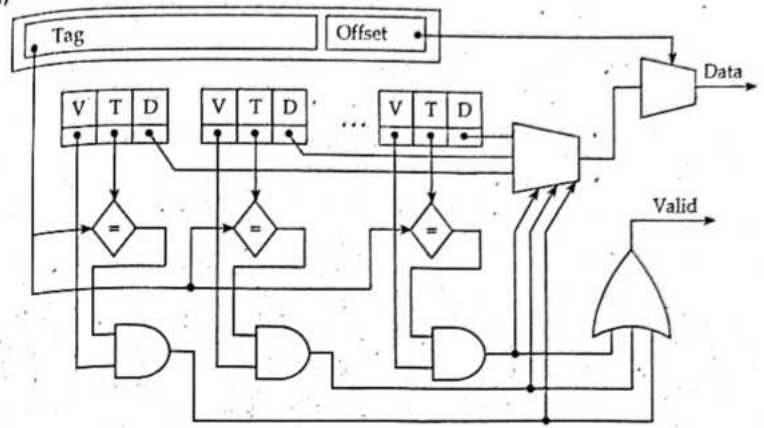
- Far fewer number of available cache address
- Are address' contents in cache?
- Cache mapping used to assign main memory address to cache address and determine hit or miss.
- Three basic techniques:
  - Direct mapping
  - Fully associative mapping
  - Set associative mapping
- Cache partitioned into indivisible blocks or lines of adjacent memory addresses
  - Usually 4 or 8 addresses per line

i) Direct mapping



- Main memory address divided into 2 fields *i.e.,* Index and tag
  - Index
    - Cache address
    - Number of bits determined by cache size
  - Tag
    - Compared with tag stored in cache at address indicated by index
    - If tags match, check valid bit
- Valid bit
  - Indicates whether data in slot has been loaded from memory.
- Offset
  - Used to find particular word in cache line

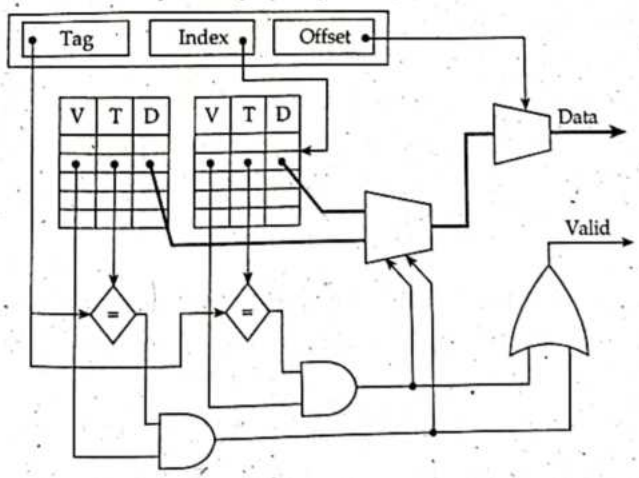
ii) Fully associative mapping



- Complete main memory address stored in each cache address
- All address stored in cache simultaneously compared with desired address
- Valid bit and offset same as direct mapping

iii) Set associative mapping

- Compromise between direct mapping and fully associative mapping
- Index same as in direct mapping
- But, each cache address contains content and tags of 2 or more memory address locations.
- Tags of that set simultaneously compared as in fully associative mapping.
- Cache with set size N called N-way-set-associative
  - 2-way, 4-way, 8-way are common



2. **Cache-replacement Policy**

- Technique for choosing which block to replace
  - When fully associative cache is full
  - When set-associative cache's line is full
- Direct mapped cache has no choice
- Random
  - Replace block chosen at random
- LRU: least recently used
  - Replace block not accessed for longest time
- FIFO: first in first out
  - Push block onto queue when accessed
  - Choose block to replace by popping queue

3. **Cache write Techniques**

- When written, data cache must update main memory
- Write through
  - Write to main memory whenever cache is written to
  - Easiest to implement
  - Processor must wait for slower main memory write
  - Potential for unnecessary writes
- Write back
  - Main memory only written when "dirty" block replaced
  - Extra dirty bit for each block set when cache block written to
  - Reduces number of slow main memory writes

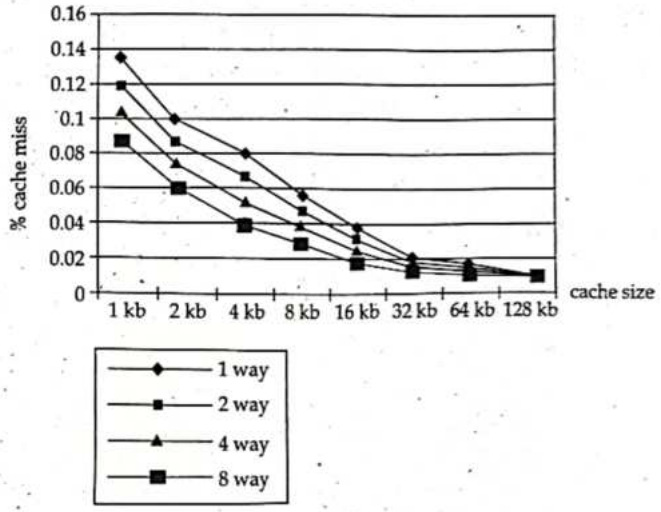
Cache impact on system performance

- Most important parameters in terms of performance
  - Total size of cache
    - Total number of data bytes cache can hold
    - Tag, valid and other housekeeping bits not included in total
  - Degree of associativity
  - Data block size
- Larger caches achieve lower miss rates but higher access cost
  - *For example,* 2 kbyte cache: miss rate = 15% hit cost = 2 cycles, miss cost = 20 cycles
  - Average cost of memory access = $(0.85 * 2) + (0.15 * 20) = 4.7$ cycles

- 4 kbyte cache: miss rate = 6.5%, hit cost = 3 cycles, miss cost will not change
  - Average cost of memory access = $(0.935 * 3) + (0.065 * 20)$ = 4.105 cycles (improvement)
- 8 kbyte cache: miss rate = 5.565%, hit cost = 4 cycles, miss cost will not change
  - Average cost of memory access = $(0.94435 * 4) + (0.05565 * 20)$ = 4.9804 cycles (worse)

Cache performance trade-offs

- Improving cache hit rate without increasing size
  - Increase line size
  - Change set-associativity



OLD EXAM QUESTION SOLUTION (TU)

1. Explain the operations of storing and erasing the data in UV-EPROM. [2069 Bhadra]

Answer: See the topic 4.4.2 (iii) of chapter 4.

2. Describe ROM and introduce its types in detail. Sketch the internal design of a 4 × 3 ROM. [2070 Bhadra]

Answer:

The program needed to boot the computer is stored in the ROM, which contains critical information required to operate the machine. It is non volatile memory. It saves all of the information it collects. It is employed in embedded systems and other situations when the programming does not need to be changed. It is also found in calculators and other electronic gadgets.

Types of ROM

i) Mask-programmed ROM
ii) OTP-ROM: One-time programmable ROM
iii) EPROM: Erasable programmable ROM
iv) EEPROM: Electrically erasable programmable ROM
v) Flash memory

For detail explanation of types of ROM:

See the 4.4.2 of chapter 4.

4 × 3 ROM

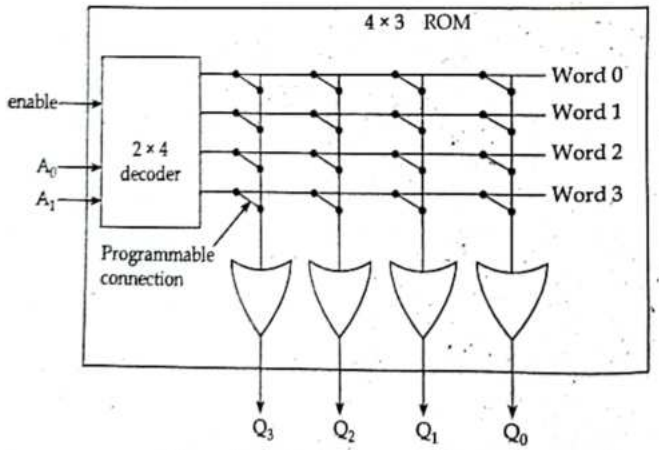


Figure: Internal design of 4 × 3 ROM

3. Design a ROM that will store the following words in the corresponding addresses. [2070 Magh]

X	Y	Z	F_1	F_2
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

Answer:

Three inputs X, Y and Z are address lines. As a result, the decoder 3 × 8 must be used for three inputs, resulting in eight word lines. There must be two data lines because there are two outputs. As a result, 8 × 2 ROM is required. The truth table is given. The programming links are made depending on the functions outputs' based on various input combinations.

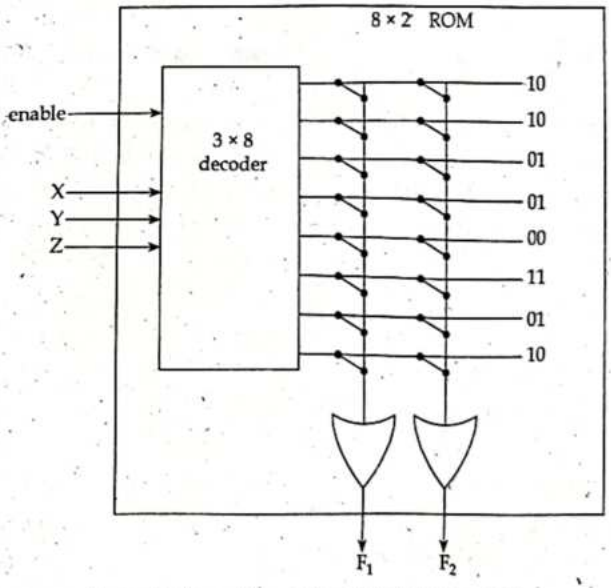


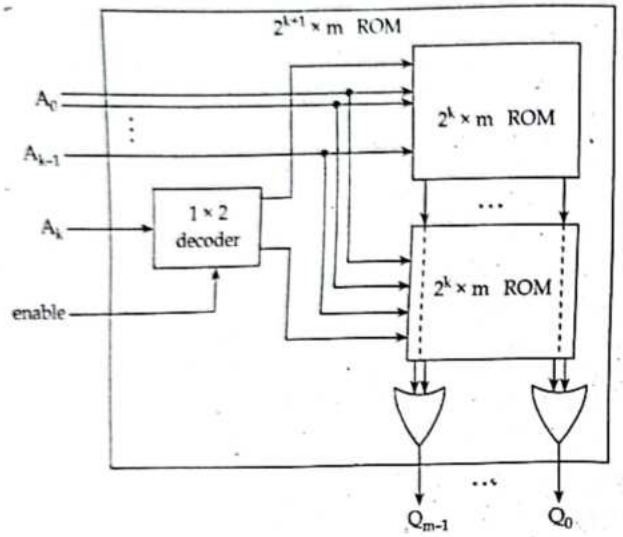
Figure: Implementation of given truth table using ROM.

4. Compose 2^{k+1} × m memory using 2^k × m memories. [2070 Magh]

Answer:

The available ROM 2^k × m and required ROM 2^{k+1} × m differ in number of words.

Increase number of words: $2^{k+1}/2^k = 2$, two ROMS are required with 1×2 decoder.



5. What is cache memory? Write cache mapping techniques. [2071 Bhadra]

Answer:

Cache memory, which is utilized in association with main memory to store copies of frequently accessed areas of main memory, is a quick but expensive memory. It's made out of static RAM. When a request for main memory access is made, the copy in cache is verified first, and if the cache hits memory access becomes faster. In the event of a cache miss, read from main memory.

There are three basic cache mapping techniques;

i) Direct mapping
ii) Fully associative mapping
iii) Set associative mapping

Cache mapping techniques: Explained in 4.4.2 of chapter 4.

6. What is enhanced DRAM built around the conventional DRAM core? Write the cache replacement policies. [2071 Magh]

Answer: Enhanced DRAM is based on a conventional DRAM core. It is a DRAM with a little quantity of static RAM to speed up memory access. Data is first examined in SRAM. If there is a miss, data is retrieved from DRAM.

i) Fast page mode DRAM (FPM DRAM)

The fast page mode DRAM design is an improvement on the basic DRAM architecture. In this design, each row of the memory bit-array is viewed as

a page. A page contains multiple words. Each word is addressed by a different column address. In FPM DRAM, the sense amplifier amplifies the entire page once its address is strobed into the row address latch. Then, each word of that page is read (or written) by strobing the corresponding column address.

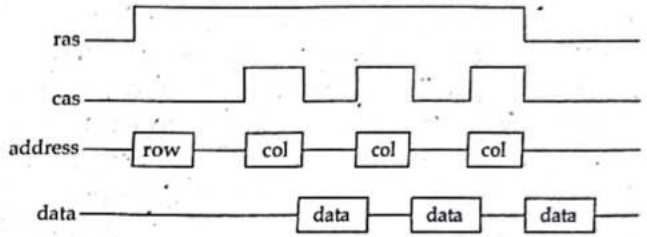


Figure: Timing Diagram of FPM DRAM

After selecting a particular page (row), three data words within that page are read consecutively. The page mode design eliminates an extra cycle on each read/write of words from the same page when compared to the basic DRAM design.

ii) Extended data out DRAM (EDO DRAM)

The extended data out DRAM design is an improvement of the FPM DRAM architecture. In this design an extra output latch is added between the sense-amplifier and the output buffers. This allows overlapping of the column select and data read operations. In other words, while a previously selected word is being read from the output buffer, a new column address can be strobed by asserthing the cas signal.

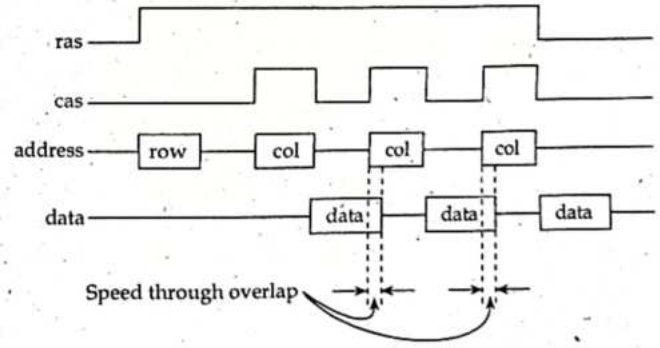


Figure: Timing diagram of EDO DRAM

iii) Synchronous DRAM (SDRAM)

The FPM and EDO DRAM architectures described so far are controlled asynchronously by the processor or the memory controller. This means that a transaction takes place when the ras/cas and rd/wr signals are

asserted in appropriate order. An alternative is to make the interface to the DRAM synchronous such that the DRAM latches information to and from the controller on the active edge of the clock signal. A synchronous interface will eliminate a small amount of time (thus latency) that is needed by the DRAM to detect the ras/cas and rd/wr signals.

In addition to a lower latency I/O, after a proper page and column setup, an SDRAM may store the starting address internally and output new data on each active edge of the clock signal, as long as the requested data are consecutive memory locations. This is accomplished by adding a column address counter to the base DRAM architecture. This counter is seeded with a starting column address strobed in by the processor (or memory controller) and is thereafter incremented internally by the DRAM on each clock cycle.

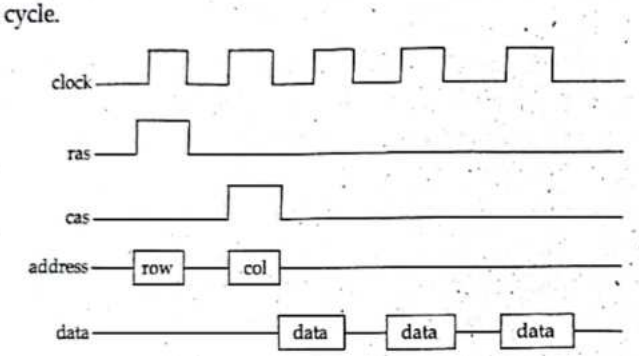


Figure: Timing diagram of SDRAM

iv) Rambus DRAM (RDRAM)

Rambus is really more of a bus interface architecture than DRAM architecture. Rambus uses multiplexed address/data lines to connect the memory controller (or processor) to the RDRAM device. The specification for this interface states that the clock runs at 300 MHz. In addition, data is latched on both rising and falling edge of the clock. Using such a bus, theoretically, a transfer rate of 600 million cycles is possible. In addition, each 64-Mbit RDRAM is broken into four parts each with its own row decoders. So, at any given time, four pages remain open. The RDRAM protocol is packet driven, where address packets are followed by data packets. The smallest transaction requires a minimum of four cycles. Because of its multiple open page schemes and fast bus I/O, RDRAM, when utilized properly, is capable of very high throughput.

v) Double data rate SDRAM (DDR SDRAM)

Higher transfer rates are possible with DDR SDRAM because the data and clock signals are timed more precisely. To double the data bus bandwidth, the interface transfers data on both the rising and falling

edges of the clock signal. DDR SDRAM, also known as DDR1, was succeeded by DDR2, which worked on the same premise as DDR1 but had a higher clock frequency and delivered double the throughput. DDR3 and DDR4 delivered improved performance for increased bus speed and new features, respectively.

Second part:

Cache replacement policies: See the topic 4.6.2 (2) of chapter 4.

7. What do you mean by write ability and storage permanence of memory? Explain associative cache mapping. [2072/Ashwin]

Answer:

Write ability: See the topic 4.3.1 of chapter 4.

Storage permanence: See the topic 4.3.2 of chapter 4.

Associative cache mapping: See the topic 4.6. [1 (ii)] of chapter 4.

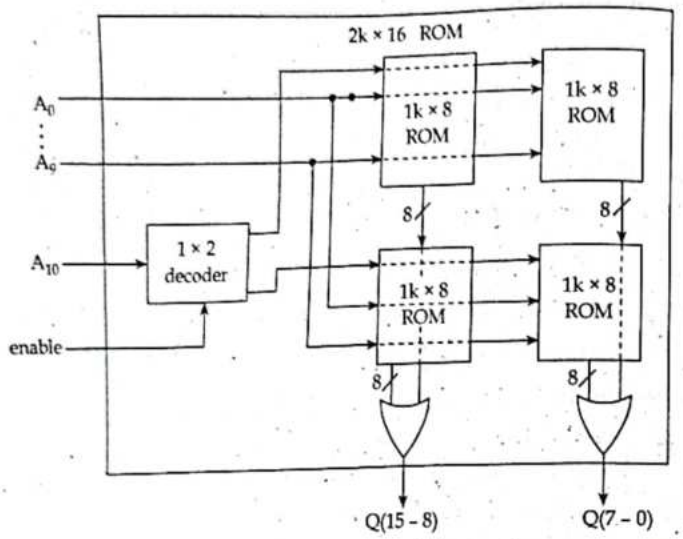
In associative cache mapping, each cache address contains not only the contents of a main memory address, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.

8. Show the differences between SRAM and DRAM. Compose 1k × 8 ROMs into a 2k × 16 ROM. [2072 Magh]

Answer:

SRAM	DRAM
i) It uses a memory cell consisting of a flip flop to store a bit.	i) It uses a memory cell consisting of a MOS transistor and capacitor to store a bit.
ii) It requires about six transistors to represent a single bit.	ii) It requires only one transistor, resulting in more compact memory than SRAM.
iii) SRAM has lower access time, so it is faster compared to DRAM.	iii) DRAM has higher access time, so it is slower than SRAM.
iv) SRAM requires constant power supply, which means this type of memory consumes more power.	iv) DRAM offers reduced power consumption, due to the fact that the information is stored in the capacitor.
v) It is costlier than DRAM.	v) DRAM casts less compared to SRAM.
vi) It has low packaging density.	vi) It has high packaging density.

1k × 8	2k × 16
k = 10	k = 11
n = 8	n = 16



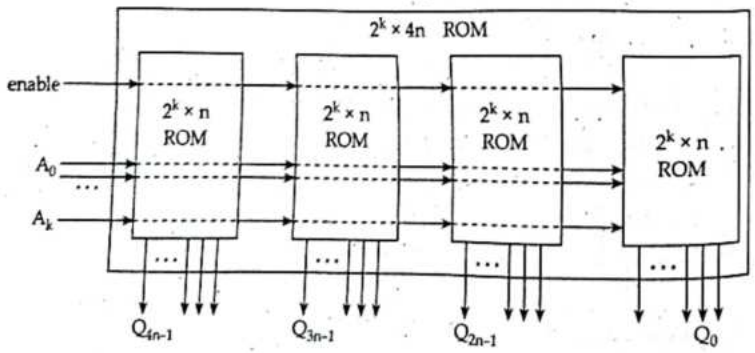
9. Construct $2^{(k+1)} \times n$ and $2^{(k)} \times 4n$ memories using $2^{(k)} \times n$ memory modules: [2073 Bhadra]

Answer:

The available ROM $2^{(k)} \times n$ and required ROM $2^{k+1} \times n$ differ in number of words.

Same as question number 4 (Old Exam Question Solution TU).

The available ROM $2^k \times n$ and required ROM $2^k \times 4n$ differ in width of words. We need ROM that is 4-times wider than what is available ($2^k \times n$); here we connect 4-ROMs side by side sharing the same address and enable lines and concatenating data lines to form desired word width.



10. What are the basic techniques for cache mapping? How direct mappings differ from fully associative mapping? [2073 Magh]

Answer:

There are three basic techniques for cache mapping. They are;

i) Direct mapping
ii) Fully associative mapping
iii) Set associative mapping

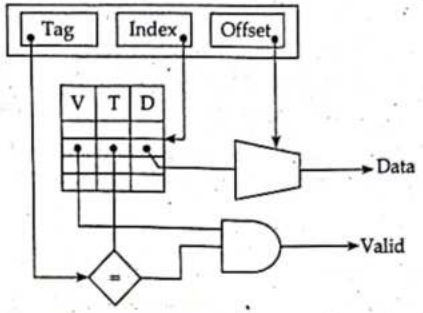


Figure: Direct Mapping

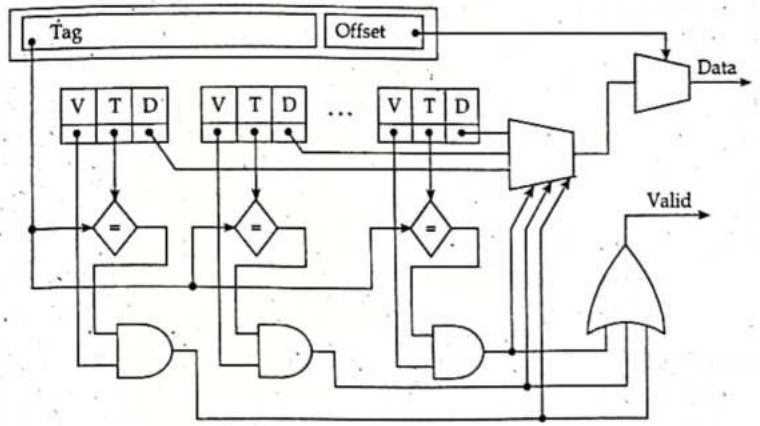


Figure: Fully associative mapping

Each block in main memory can be placed anywhere in the cache in a full associative cache mapping. Assume that the cache block sizes are 2n for some n (typically 4 to 6). The low order n bits of a memory reference's address are removed, and the remainder of the address indicates the tag field. To see if the appropriate physical memory block in the cache, the tag fields of the blocks in the cache are scanned in simultaneously.

Each block in main memory can only move into one block in the cache when using a direct mapped cache mapping. The address's low order n bits are deleted once again. The tag is represented by a portion of the remaining address bits (the one cache block where this main memory block would be stored. To verify if the proper block is already in the

cache, the remaining address bits are compared to the appropriate part of the block's address in the one possible cache location.

Collisions between blocks attempting to occupy the cache are substantially less likely with full associative mapping. With direct mapping, two or more main memory blocks may have to fit into the same cache block, but with complete (or set) associative mapping, they may go into different cache blocks. However, more hardware is required to identify which cache block to employ and to examine all cache blocks in parallel to determine whether a memory access is successful.

11. Define write ability and storage permanence of memory. Explain associative cache mapping technique with its merits and demerits. [2074 Bhadra]

Answer:

Write ability refers to the manner and speed that a particular memory can be written. Basically we have four levels of write ability.

i) High end
ii) Middle range
iii) Lower range
iv) Low end

Storage permanence is the ability of memory to hold its stored bits after those bits have been written. On the basis of storage permanence memory devices can be categorized as,

i) High end
ii) Middle range
iii) Lower range
iv) Low end

Second part: See question number 7 (old exam question solution TU)

Merits of associative cache mapping

- It is fast
- It is easy to implement

Demerits of associative cache mapping

- Cache memory implementing associative mapping is expensive as it requires to store address along with the data.

12. Explain the operation of storing data in one-time programmable ROM. Why it can't be reprogrammed? Compose 1k × 8 ROMs into a 4k × 8 ROM. [2075 Baisakh]

Answer:

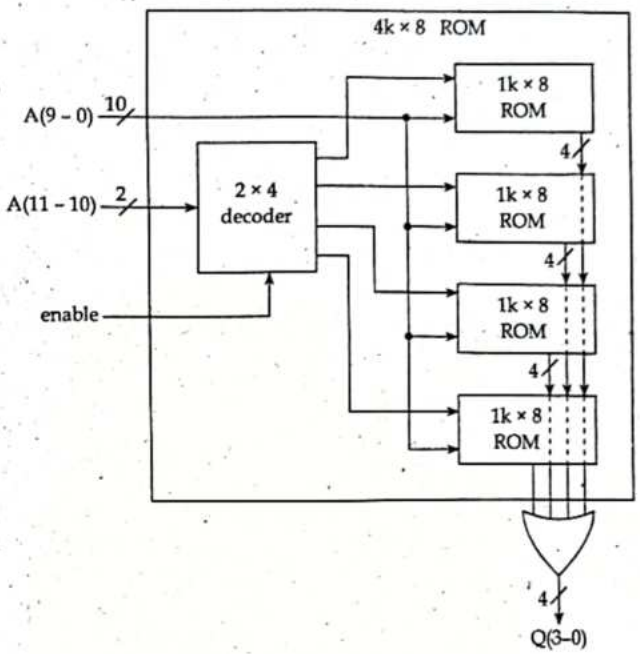
First part: See the topic 4.4.2 (ii) of chapter 4.

Second part:

A device known as a programmer is used to program each programmable connection according to the file provided by the user. By passing a large

current wherever a connection is not required, the programmer blows fuses. Because the blown fuses cannot be replaced, it is referred to as a one-time programmable ROM. Hence, it can't be reprogrammed.

1k × 8 ROMs into 4k × 8 ROM



13. Define two characteristics of memory: write ability and storage permanence with their different levels. Explain replacement algorithms used in cache memory. [2076 Bhadra]

Answer:

First part: See the topic 4.2.1 and 4.2.2 of chapter 4.

Second part: See the topic 4.5.2 (2) of chapter 4.

14. Show the trade-offs between memory write ability and memory storage performance. With neat diagram, explain RAM internal diagram, also explain types of RAM. [2077 Poush]

Answer:

First part: See the topic 4.3 of chapter 4.

Second part: See 4.4.3 and 4.4.4 of chapter 4

OLD EXAM QUESTIONS SOLUTION (PoU)

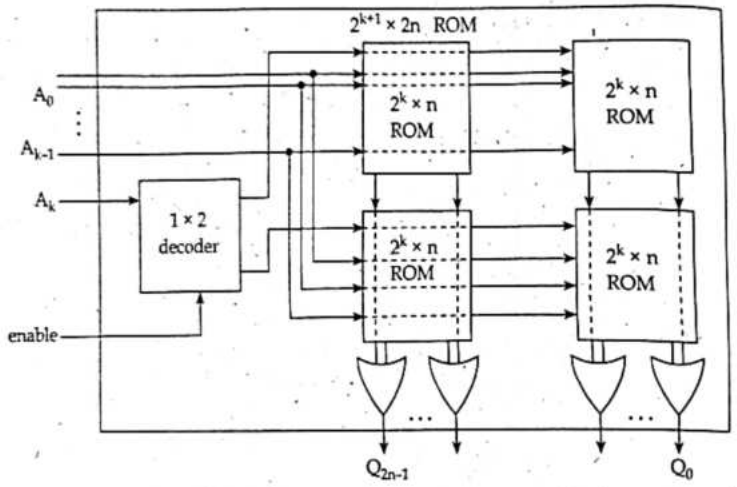
1. Design 4k × 8 ROMs using 1k × 8 ROMs (1k = 1024 words). [2016 Fall]

Answer: Same as question number 12 (second part) (Old Exam Questions Solution TU).

2. Compose $2^{k+1} \times 2n$ memory using $2^k \times n$ memory modules. [2016 Spring]

Answer:

The available ROM $2^k \times n$ and required ROM $2^{k+1} \times 2n$ differ in number of words and width of words.



3. Explain briefly the different cache mapping techniques with suitable diagrams. [2017 Fall]

Answer: See the topic 4.6.2 (1) of chapter 4.

4. Compose 1k × 8 ROM into an 2k × 16 ROM. [2017 Spring/2020 Fall]

Answer: Same as question number 8 (second part) (Old Exam Questions solution TU).

5. What is memory cache mapping? Explain cache mapping techniques. [2018 Spring]

Answer: Same as question number 5 (Old Exam Question Solution TU).

6. How can you compose memory to increase number and width of words? Design 2k × 16 ROMs using 1k × 8 ROMs (1k = 1024 words). [2018 Spring]

Answer:

First part: See the topic 4.5 of chapter 4.

Second part: Same as question number 8 (second part) (Old Exam Question Solution TU).

7. Explain different types of RAM with RAM variation. [2019 Fall]

Answer: See the topic 4.4.3 and 4.4.4 of chapter 4.

Chapter 5
INTERFACING

★★★

★★★

5.1 INTRODUCTION

- Embedded system functionality aspects
  - Processing
    - Transformation of data
    - Implemented using processor
- Storage
  - Retention of data
  - Implemented using memory
- Communication
  - Transfer of data between processors and memories
  - Implemented using buses
  - Called interfacing

5.2 A SIMPLE BUS

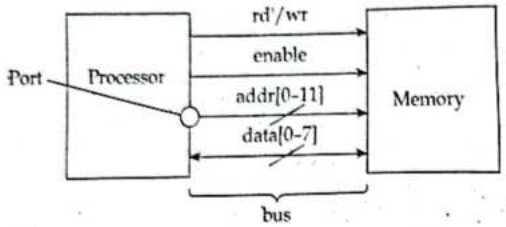
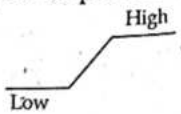
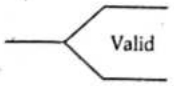


Figure: Bus structure

- Wire
  - Uni-directional or bi-directional
  - One line may represent multiple wires
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication
- Ports
  - Conducting device on periphery
  - Connect bus to processor or memory
  - Often referred to as a pin
    - Actual pins on periphery of IC package that plug into socket on printed-circuit board
    - Sometimes metallic balls instead of pins
    - Today, metal "pads" connecting processors and memories within single IC
  - Single wire or set of wires with single function
    - *For example,* 12-wire address port
- Timing Diagrams
  - Most common method for describing a communication protocol
  - Time proceeds to the right on x-axis
  - Control signal: low or high



    - May be active low (*For example,* go', /go, or go_L)
    - Use terms assert (active) and deassert
    - Asserting go' means go = 0
  - Data signal: not valid or valid



- Protocol may have sub protocols
  - Called bus cycle, *For example,* read and write
  - Each may be several clock cycles
- Read example
  - rd'/wr set low, address paced on addr for at least t_{setup} time before enable asserted, enable triggers memory to place data on data wires by time t_{read}

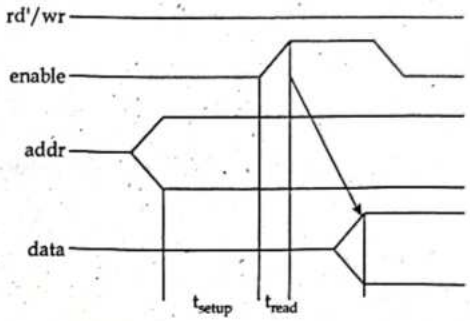


Figure: Read protocol

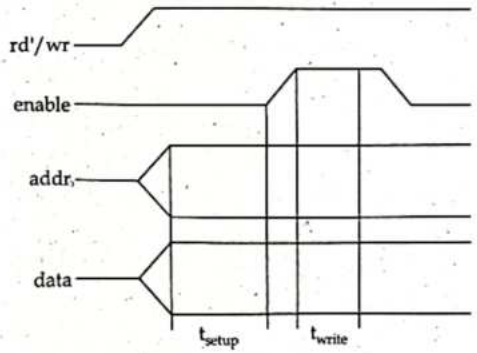


Figure: Write protocol

5.3 BASIC PROTOCOL CONCEPTS

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Address: special kind of data
  - Specifies a location in memory, a pheriperial or a register within a pheriperial
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time
- Time multiplexed data transfer

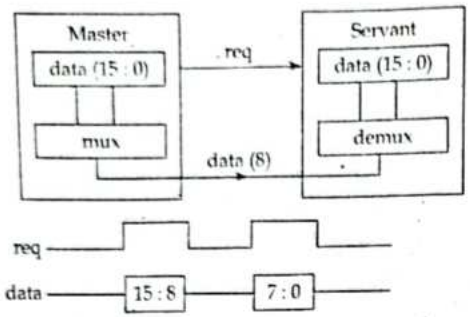


Figure: Data serializing

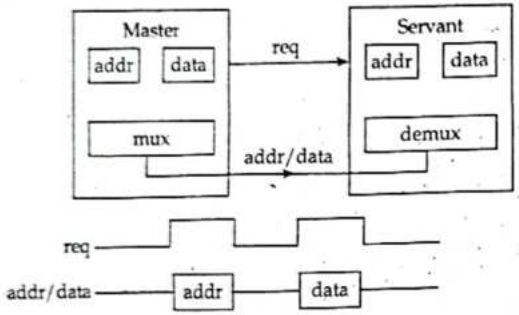
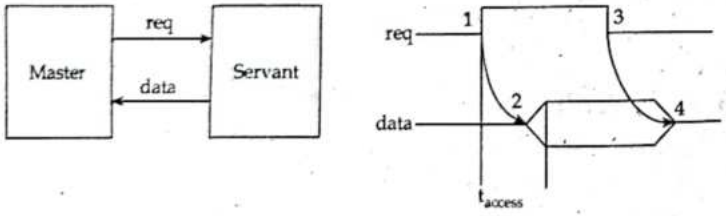


Figure: Address/data muxing

- Control Methods:
  - Control methods are schemes for initiating and ending the transfer
  - Two of the most common method is "strobe" and "handshake"

I) Strobe

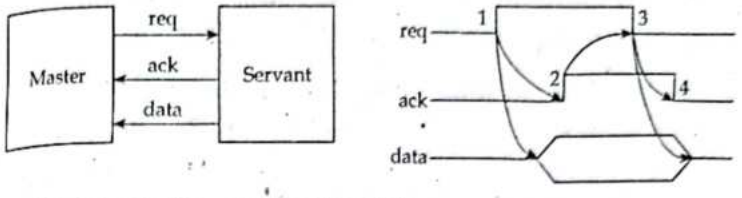
In strobe protocol, the master uses one control line, called the request line to initiate the data transfer and the transfer is considered to be complete after some fixed time interval after initiation.



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

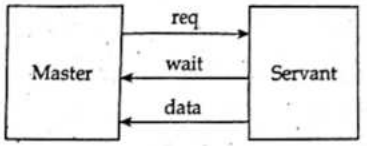
II) Handshake

In this protocol the master uses a request line to initiate the transfer and the servant uses an acknowledge line to inform the master when data is ready.

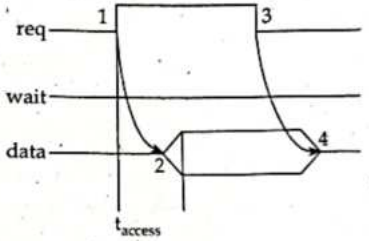


1. Master asserts req to receiver data
2. Servant puts data on bus and asserts ack
3. Master receives data and asserts req
4. Servant ready for next request.

III) Strobe/handshake compromise

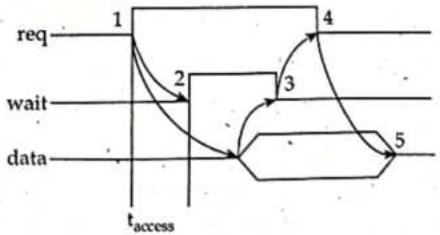


a) Fast response case:



- Mater asserts req to receive data
- Servant puts data on bus within time t_{access} (wait line is unused)
- Master receives data and deasserts req
- Servant ready for next request

b) Slow response case

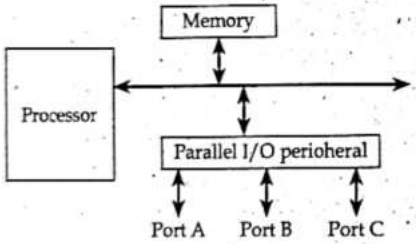


- Master asserts req to receive data
- Servant can't put data within t_{access}, asserts wait ack.
- Servant puts data on bus and deasserts wait
- Master receives data and deasserts req
- Servant ready for next request

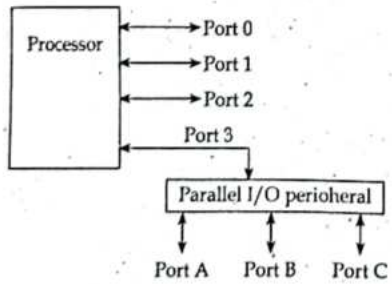
5.4 MICROPROCESSOR INTERFACING: I/O ADDRESSING, INTERRUPTS, DMA

5.4.1 Microprocessor Interfacing: I/O Addressing

- A microprocessor communicates with other devices using some of its pins
  - Port-based I/O (parallel I/O)
    - Processor has one or more N-bit ports
    - Processor's software reads and writes a port just like a register
    - *For example,* P0 = OxFF; V = P1.2; - - P0 and P1 are 8-bits ports
  - Bus-based I/O
    - Processor has address, data and control ports that form a single bus
    - Communication protocol is built into the processor
    - A single instruction carries out the read or write protocol on the bus
- Compromises/extensions
  - Parallel I/O peripheral



    - When processor only supports bus-based I/O but parallel I/O needed
    - Each port on peripheral connected to a register within peripheral that is read/written by the processor
  - Extended parallel I/O



    - When processor supports port-based I/O but more ports needed.
    - One or more processor ports interface with parallel I/O peripherial extending total number of ports available for I/O
    - *For example,* extending 4 ports to 6 ports in figure

Types of bus-based I/O

- Processor talks to both memory and peripherials using same bus- two ways to talk to peripherials:
  a) memory-mapped I/O
  b) Standard I/O (I/O mapped I/O)

a) Memory-mapped I/O

- Peripherial registers occupy addresses in same address space as memory.
- *For example,* Bus has 16-bit address
  - Lower 32 k addresses may correspond to memory
  - Upper 32 k addresses may correspond to peripherials
- Requires no special instructions
  - Assembly instructions involving memory like MOV and ADD work with peripherials as well.
  - Standard I/O requires special instructions (*For example,* IN, OUT) to move data between peripherial registers and memory.

b) Standard I/O (I/O mapped I/O)

- Additional pin (M/IO) on bus indicates whether a memory or peripheral access.
- *For example,* Bus has 16-bit address
  - All 64k address correspond to memory when M/IO set to 0.
  - All 64k address correspond to peripherial when M/IO set to 1.
- No loss of memory addresses to peripherals.
- Simpler address decoding logic in peripherals possible.
  - When number of peripherals much smaller than address space then high-order address bits can be ignored.
  - Smaller and/or faster comparators.

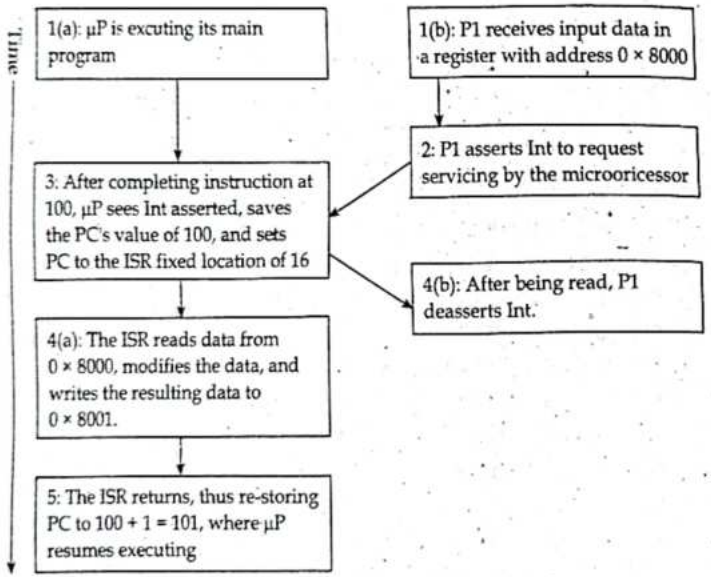
5.4.2 Microprocessor Interfacing: Interrupts

- Suppose a peripherial intermittently receives data, which must be serviced by the processor.
  - The processor can poll the peripherial regularly to see if data has arrived-wasteful.
  - The peripherial can interrupt the processor when it has data.
- Requires an extra pin or pins: Int
  - If Int is 1, processor suspends current program, jumps to an interrupt service routine, or ISR.

- Known as interrupt-driven I/O
- Essentially, "polling" of the interrupt pin is built into the hardware, so no extra time!

- What is the address (interrupt address vector) of the ISR?
  - Fixed interrupt
    - Address built into microprocessor, cannot be changed
    - Either ISR stored at address or a jump to actual ISR stored if not enough bytes available
  - Vectored interrupt
    - Peripherial must provide the address
    - Common when microprocessor has multiple peripherials connected by a system bus.
  - **Compromise:** Interrupt address table.

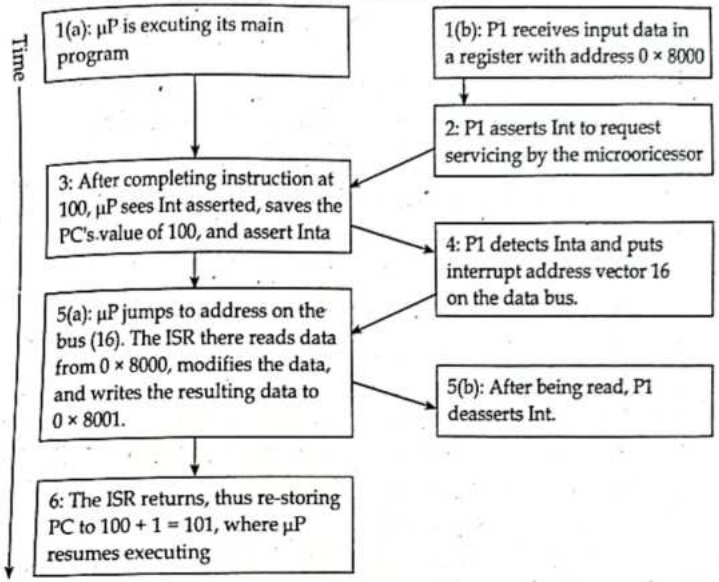
a) **Interrupt driven I/O using fixed ISR location**



- In this example, data received by peripheral 1 must be read, transformed and then written to peripheral 2. Peripheral 1 might represent a senor and peripheral 2 is a display.
- The processor is a running its main program located in program memory starting at address 100.
- When peripheral 1 receives data, its asserts INT to request the microprocessor to service the data.
- After microprocessor completes execution of its current instruction, it stores its state and jumps to the ISR location at the fixed program memory location of 16.

- The ISR reads the data from peripheral 1 transforms it and writes the result to peripheral 2.
- The last ISR instruction is a return from interrupt, causing the microprocessor to restore its state and resume execution of its main program.

b) **Interrupt driven I/O using vectored interrupt**



5.4.2.1 Interrupt Address Table

- Compromise between fixed and vectored interrupts
  - One interrupt pin
  - Table in memory holding ISR addresses (may be 256 words)
  - Peripheral does not provide ISR address, but rather index into table
    - Fewer bits are sent by the peripheral

5.4.2.2 Interrupt Issues

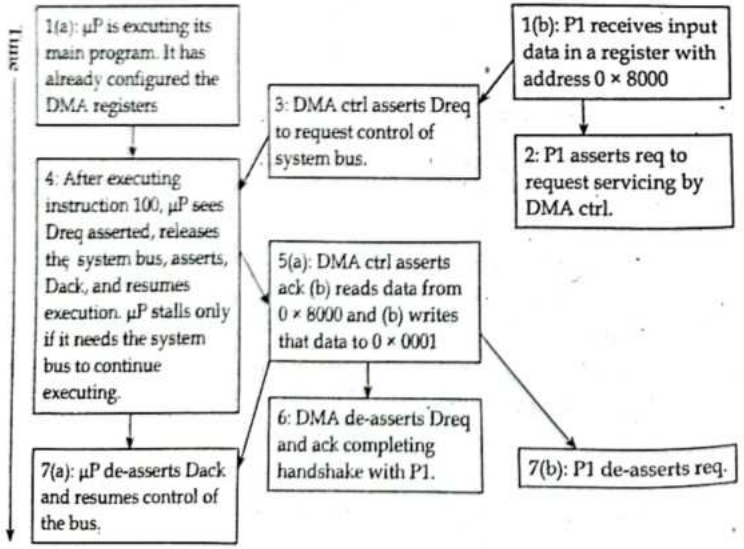
- Maskable VS: Non-maskable interrupts
  - Maskable: programmer can set bit that causes processor to ignore interrupt.
    - Important when in the middle of time-critical code.
- Non-maskable: a separate interrupt pin that can't be masked
  - Typically reserved for drastic situations, like power failure requiring immediate back up of data to non-volatile memory.
- Jump to ISR
  - Some microprocessors treat jump same as call of any subroutine.
    - Complete state saved (pc, register)-may take hundreds of cycles

- Others only save partial state, like pc only
  - Thus, ISR must not modify registers, or else must save them first.
  - Assembly-language programmer must be aware of which register stored

5.4.3 Microprocessor Interfacing: Direct Memory Access

- Buffering
  - Temporarily storing data in memory before processing
  - Data accumulated in peripherals commonly buffered.
- Microprocessor could handle this with ISR
  - Storing and restoring microprocessor state inefficient
  - Regular program must wait.
- DMA controller more efficient
  - Separate single-purpose processor
  - Microprocessor relinquishes control of system bus to DMA controller
  - Microprocessor can meanwhile execute its regular program
    - No inefficient storing and restoring state due to ISR call
    - Regular program need not wait unless it requires the system bus
      - Harvard architecture-processor can fetch and execute instructions as long as they don't access data memory-if they do processor stalls

 # Peripheral to memory transfer with DMA



5.5 ARBITRATION

Multiple peripherals might request service from a single resources as well as multiple peripherals might share single DMA controller that services their DMA request.

In such situations, two or more peripherals may request service simultaneously. Thus, there need to be some-way to arbitrate (decide) the contending request.

5.5.1 Priority Arbiter

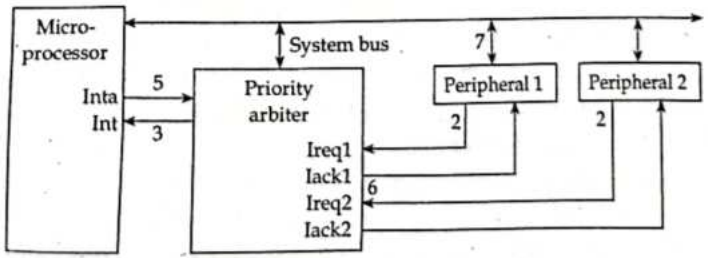
- Consider the situation where multiple peripherals request service from single resource (*For example*, microprocessor, DMA controller) simultaneously-which gets serviced first?
- Priority arbiter
  - Single purpose processor
  - Peripherals make requests to arbiter, arbiter make request to resources
  - Arbiter connected to system bus for configuration only.
- Priority arbiter uses two common schemes to determine priority among peripherals
  - Fixed priority
  - Rotating priority

i) Fixed priority

- Each peripheral has unique rank
- Highest rank chosen first with simultaneous requests
- Preferred when clear difference in rank between peripherals

ii) Rotating priority

- Priority changed based on history of servicing
- Better distribution of servicing especially among peripherals with similar priority demands.

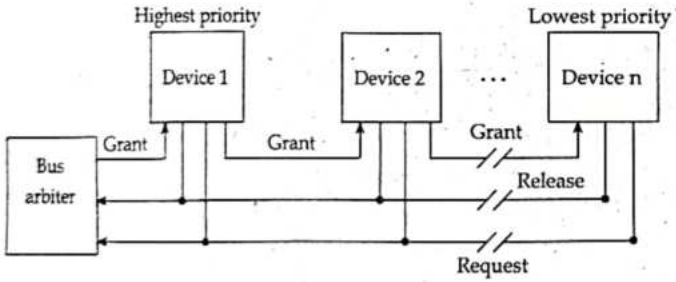


1. Microprocessor is executing its program
2. Peripheral 1 needs servicing so asserts Ireq1. Peripheral 2 also needs servicing so asserts Ireq2.

3. Priority arbiter sees at least one Ireq input asserted, so asserts Int.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts Inta.
6. Priority arbiter asserts lack 1 to acknowledge peripheral 1
7. Peripheral 1 puts its interrupt address vector on the system bus
8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

5.5.2 Daisy-Chain Arbitration

- Arbitration done by peripheral
  - Built into peripheral or external logic added
    - ➢ req input and ack output added to each peripheral.
- Peripherals connected to each other in daisy-chain manner
  - One peripheral connected to resource, all others connected "upstream"
  - Peripheral's req flows "downstream" to resource, resource's ack flows "upstream" to requesting peripheral
  - Closest peripheral has highest priority
- Easy to add/remove peripheral-no system redesign needed
- Does not support rotating priority
- One broken peripheral can cause loss of access to other peripherals.



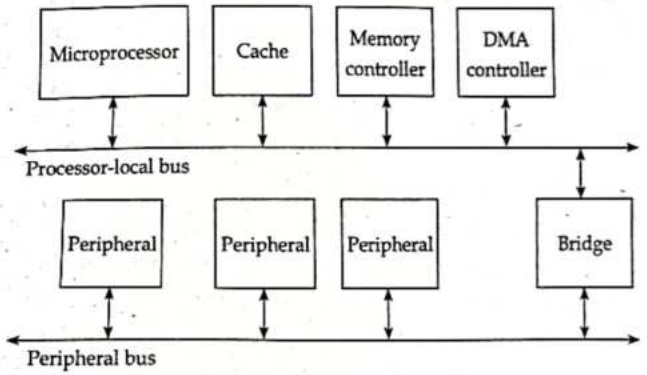
1. Processor is executing its program
2. Any peripheral needing service activities "req-out". This "req-out" goes to the req-in of the subsequent device in the chain.
3. The peripheral nearest to the microprocessor assert "int".
4. The processor stores executing instruction and stores its state.
5. The processor asserts 'inta" to nearest device.
6. The "inta" signal passes through the chain till it finds a flag which is set by the device which has generated the interrupt.
7. The interrupting device sends the interrupt address vector to the processor for its interrupt service routine.

8. The processor jumps to the address of ISR reads from the data bus, ISR executes the return.
9. The flag is reset.
10. The processor checks for the next device that has interrupted.

5.5.3 Network-oriented Arbitration

- When multiple microprocessors share a bus (sometimes called a network)
- Arbitration typically built into bus protocol
- Separate processors may try to write simultaneously causing collisions
  - Data must be resent
  - Don't want to start sending again at the same time
    - ➢ Statistical methods can be used to reduce chances.
- Typically used for connecting multiple distant chips
  - Trend-use to connect multiple on-chip processors.

5.6 MULTIPLE BUS ARCHITECTURES



- Don't want one bus for all communication
  - Peripherals would need high-speed, processor-specific bus interface
    - ➢ Excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus
- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc
- Peripherals bus
  - Lower speed, narrower, less frequent communication
  - Typically industry standard bus (ISA,PCI) for portability
- Bridge
  - Single-purpose processor converts communication between buses

5.7 ADVANCED COMMUNICATION PRINCIPLES

a) Parallel Communication

- Physical layer capable of transporting multiple bits of data.
- Multiple data, control, and possibly power wires.
  - One bit per wire
- High data throughput with short distances
- Typically used when connecting devices on same IC or same circuit board.
  - Bus must be kept short
    - Long parallel wires result in high capacitance values which requires more time to charge/discharge.
    - Data misalignment between wires increases as length increases.
- Higher cost, bulky

b) Serial Communication

- Single data wire, possibly also control and power wires.
- Words transmitted one bit at a time.
- Higher data throughput with long distances.
  - Less average capacitance, so more bits per unit of time.
- Cheaper, less bulky
- More complex interfacing logic and communication protocol.
  - Sender needs to decompose word into bits.
  - Receiver needs to recompose bits into word.
  - Control signals often sent on same wire as data increasing protocol complexity.

c) Wireless Communication

- Infrared (IR)
  - Electronic wave frequencies just below visible light spectrum.
  - Diode emits infrared light to generate signal.
  - Infrared transistor detects signal, conducts when exposed to infrared light
  - Cheap to build
  - Need line of sight, limited range
- Radio frequency (RF)
  - Electromagnetic wave frequencies in radio spectrum.
  - Analog circuitry and antenna needed on both sides of transmission
  - Line of sight not needed, transmitter power determines range.

d) Layering

- Breaking complexity of communication protocol into pieces easier to design and understand.
- Lower level provide services to higher level.
  - Lower level might work with bits while higher level might work with packets of data.
- Physical layer
  - Lowest level in hierarchy
  - Medium to carry data from one actor (device or node) to another.

e) Error Detection and Correction

- Often part of bus protocol
- Error detection: ability of receiver to detect errors during transmission.
- Error correction: ability of receiver and transmitter to cooperate to correct problem.
  - Typically done by acknowledgement/retransmission protocol
- Bit error: single bit is inverted
- Burst of bit error: consecutive bits received incorrectly.
- Parity: extra bit sent with word used for error detection
  - Odd parity: data word plus parity bit contains odd number of 1's.
  - Even parity: data word plus parity bit contains even number of 1's.
  - Always detects single bit errors, but not all burst bit errors.
- Checksum: extra sent with data packet of multiple words.
  - *For example,* extra word contains XOR sum of all data words in packet.

OLD EXAM QUESTIONS SOLUTION (TU)

1. Explain arbitration systems that implemented to communicate with peripheral devices from the microprocessor. Differentiate between memory mapped I/O with standard I/O. [2069 Bhadra]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part:

Differences between memory mapped I/O and standard (isolated) I/O are given below;

Memory mapped I/O	Standard I/O
i) Memory and I/O have same address space.	i) Memory and I/O have separate address space.
ii) Due to addition of I/O addressable memory become less for memory.	ii) All address can be used by the memory.
iii) Same instructions can control both I/O and memory.	iii) Separate instruction control read and write operation in I/O and memory.
iv) Normal memory address are for both.	iv) In this I/O address are called ports.
v) Lesser efficient.	v) More efficient due to separate buses.
vi) Smaller in size.	vi) Larger in size due to more buses.
vii) Simpler logic is used as I/O is also treated as memory only.	vi) It is complex due to separate logic is used to control both.

2. Describe the purpose of the direct-memory-access (DMA) controller. Draw the flow of actions between peripheral and memory using DMA. [2070 Magh]

Answer: See the topic 5.4.3 of chapter 5.

3. Describe the advanced communication principles used in embedded systems. [2070 Magh]

Answer: See the topic 5.7 of chapter 5.

4. Explain different types of arbitration methods used in peripherals devices to gain control of system bus. Describe the significance of I^2C serial communication protocol. [2070 Bhadra]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part:

Inter integrated circuit (I^2C) is a serial bus short distance protocol. I^2C is appropriate for interfacing to devices on a single board, and can be stretched across multiple boards inside a closed system. It is a protocol intended to allow multiple "peripheral" digital integrated circuits (chips) to communicate with one or more "controller" chips.

5. What is the difference between memory-mapped I/O and standard I/O. Explain the operation of peripheral to memory transfer without DMA, using vectored interrupt. [2072 Ashwin]

Answer:

First part: Same as question number 1 second part (Old Exam Questions Solution TU).

Second part: See the topic 5.4.2 (b) of chapter 5.

6. Describe two-level bus architecture in detail. Describe priority arbitration method and compare it with daisy-chain arbitration. [2074 Bhadra]

Answer:

First part: See the topic 5.6 of chapter 5.

Second part: See the topic 5.5.1 and 5.5.2 of chapter 5.

7. What is arbitration? Explain priority arbitration with the help of a block diagram and steps along with its types. [2075 Baisakh]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part: See the topic 5.5.1 of chapter 5.

8. What is interrupt? Explain summary of flow of actions of interrupt driven I/O using fixed ISR location. [2075 Bhadra]

Answer:

First part:

An interrupt is a signal sent to the processor by hardware or software to indicate that an event has occurred that requires rapid attention. When an interrupt occurs, ISR instructs the processor or controller what to do.

Second part: See the topic 5.4.2 (a) of chapter 5.

9. What is interrupts? How interrupts needed in digital devices? Write a summary of flow of actions for interrupt driven I/O using fixed ISR location. [2076 Bhadra]

Answer:

First part: Same as question number 8.

Second part:

In digital devices, an interrupt is a response by the processor to an event that needs attention from the software. An interrupt condition alerts the

processor and serves as a request for the processor to interrupt the currently executing code when permitted, so that the event can be processed in a timely manner.

Third part: See the topic 5.4.2 (a) of chapter 5.

10. Explain memory mapped I/O standard I/O. Describe priority arbitration method and compare it with daisy-chain arbitration. [2077 Poush]

Answer:

First part part: See the topic 5.4.1 (a) and (b) of chapter 5.

Second part: See the topic 5.5 of chapter 5.

11. Explain the operation of interrupt-driven I/O using vectored interrupt. Describe the ISA bus protocols. [2078 Baisakh]

Answer:

First part part: See the topic 5.4.2 (b) of chapter 5.

Second part: See the topic 5.6 of chapter 5.

OLD EXAM QUESTIONS SOLUTION (PoU)

1. What is arbitration? Explain the steps used in Daisy-chain arbitration with a block diagram. [2016 Fall] [2020 Fall]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part: See the topic 5.5.2 of chapter.

2. What is interrupt? Explain the steps used in data transfer using vector interrupt along with its flowchart. [2016 Spring]

Answer:

First part: See question number 8 [Old Exam Question Solution TU].

Second part: See the topic 5.4.2 (b) of chapter 5.

3. Explain any two arbitration techniques that implemented to communicate with peripheral devices from the microprocessor. [2016 Spring]

Answer: See the topic 5.5 of chapter 5 (describe any two).

4. What do you mean by arbitration? Explain the daisy chain and network-oriented arbitration briefly. [2017 Fall] [2018 Spring]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part: See the topic 5.5.2 and 5.5.3 of chapter 5.

5. What is direct memory access? Why such circuitry is needed? Explain with its block diagram. [2017-Spring]

Answer: See the topic 5.4.3 of chapter 5.

6. What is arbitration? Explain daisy-chain arbitration. [2017 Spring]

Answer:

First part: See the topic 5.5 of chapter 5.

Second part: See the topic 5.5.2 of chapter 5.

7. Why we need DMA? Explain the working principle of DMA. [2018 Fall]

Answer: See the topic 5.4.3 of chapter 5.

8. Compare and contrast bus-based I/O and port based I/O. [2018 Fall]

Answer: See the topic 5.4.1 of chapter .5.

9. Explain daisy-chain arbitration in detail. What is the significant of multilevel bus architecture. [2019 Fall]

Answer:

First part: See the topic 5.5.2 of chapter 5.

Second part: See the topic 5.6 of chapter 5.

10. Explain the benefits that an interrupt address table has over the fixed and vectored interrupt methods. [2017 Fall, 2019 Fall]

Answer:

A table of ISR addresses is stored in the processor's memory in the interrupt address table, which is a compromise between fix and vectored interrupt mechanisms. Instead of the address of ISR, a peripheral supplies the number that corresponds to an entry in the table. One significant advantage is that the number of bits required to address the table is significantly fewer than the number of bits required to address the ISR's true address. It also gives you the freedom to assign and adjust the location of ISR.

The interrupt address vector specifies the location of the interrupt service routine (ISR). The CPU acquires the address of ISR in one of two ways. Fixed interrupt or vectored interrupt. The address of the subroutine is integrated into the microprocessor and remains fixed in fixed interrupt. Simply save the ISR at that place, or use jump instruction to leap to the real location of the ISR that the programmer has saved.

The peripheral must provide the CPU with the address in a vectored interrupt. Along with the INT pin, the INTA pin is required in this manner to acknowledge that an interrupt has been detected so that the peripheral can transmit the address of the appropriate ISR through the system bus. The address is provided by the peripheral through the data bus, which is read by the microprocessor.

Chapter 6

REAL TIME OPERATING SYSTEM

★★★

★★★

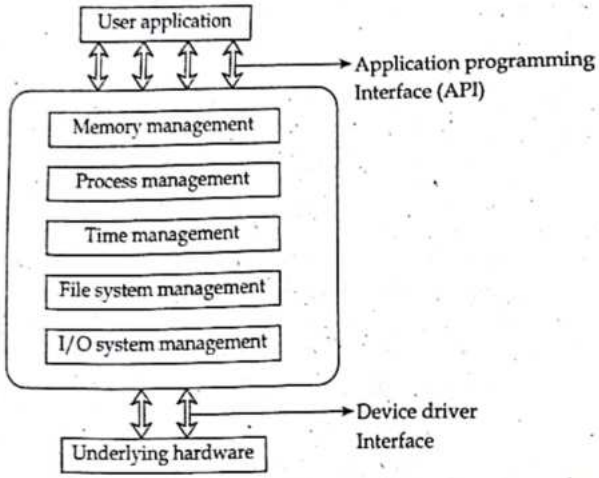
6.1 OPERATING SYSTEM BASICS

An operating system (OS) is large and complex set of system programs that control the various operations of a computer systems and provide a collection of services to other (user) programs.

The purpose of an operating system involves two keys goals:

- Availability of a convenient, easy-to-use, and powerful set of services that are provided to the users and the application programs in the computer system.
- Management of the computer resources in the most efficient manner.

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



6.1.1 Comparison of General Purpose OS (GPOS) with Real Times OS (RTOS)

A general purpose OS (GPOS) is an essential component of any mobile device, server or computer system, and is responsible for running all the applications in an installation. Platforms like linux, windows, and Mac OS are GPOS. GPOS is great for performing multiple tasks at the same time, but issues with latency and synchronization make them less than ideal for time-sensitive applications.

Real time OS (RTOS) are software platforms designed for uses cases in which time is of the essence, for example, in connected cars. Processing time must be far shorter than in a GPOS, and the execution pattern for applications and processes needs to be predictable. Generally, RTOS runs on smaller, more lightweight hardware than GPOS, as larger hardware configurations tend to be less agile.

6.1.2 Differences between GPOS and RTOS

- GPOS cannot perform real time tasks whereas RTOS is suitable for real time applications.
- There is deadline associated with real time kernel but GPOS does not follow timely mechanism.
- The real time kernel follows preemptive scheduling policy whereas GPOS follows non--preemptive scheduling technique.

- Synchronization is a problem with GPOS whereas synchronization is achieved in real time kernel.
- Inter task communication is done using real time OS where GPOS does not.
- Latency is a problem with GPOS but it is overcome using real time OS.
- Priority inversion cannot be done in GPOS, it is done with real time kernel.
- Jitter *i.e.*, timing error in task is not present in real time OS, present in GPOS.
- The mathematical relation cannot be defined for GPOS, for a real time OS mathematical equation can be defined.

A. The Kernel

The kernel in the OS provides the basic level of control on all the computer peripherals. In the operating system, the kernel is an essential component that loads firstly and remains within the main memory. So, that memory accessibility can be managed for the programs within the RAM, it creates the programs to get access from the hardware resources. It resets the operating states of the CPU for the best operation at all times. The Kernel contains different services for handling the following:

i) Process Management

The creation, execution, and termination of processes keep on going inside the system whenever a system is in the ON mode. A process contains all the information about the task that needs to be done. So, for executing any task, a process is created inside the systems. At a time, there are many processes which are in live state inside the system. The management of all these processes is very important to avoid deadlocks and for the proper functioning of the system, and it is handled by the kernel.

ii) Memory Management

Whenever a process is created and executed, it occupies memory, and when it gets terminated, the memory can be used again. But the memory should be handled by someone so that the released memory can be assigned again to the new processes. This task is also done by the kernel. The kernel keeps track about which part of the memory is currently allocated and which part is available for being allocated to the other processes.

iii) Device Management

The kernel also manages all the different devices which are connected to the systems, like the input and output devices, etc.

iv) Interrupt Handling

While executing the processes, there are conditions where tasks with more priority need to be handled first. In these cases, the kernel has to interrupt in-between the execution of the current process and handle tasks with more priority which has arrived in between.

v) I/O Communication

As the kernel manages all the devices connected to it, so it is also responsible for handling all sorts of input and output that is exchanged through these devices. So, all the information that the system receives from the user and all the output that the user is provided with via different applications is handled by the kernel.

B. Kernel Space and User Space

i) Kernel Space

- This is a protected memory space that has full access to the hardware and system state.
- Kernel space contains kernel code, core data structures identical to all process.
- In kernel space, most of the memory is directly mapped to physical memory at the fixed offset.

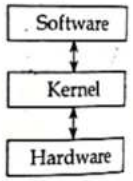
ii) User Space

- The normal application executes in user space and they can see the only subset of the machine's available resources and perform certain system functions.
- User space contains process code, data and memory-mapped files.
- In user space, memory mapping differs from one address space to another.

C. Types of Kernel

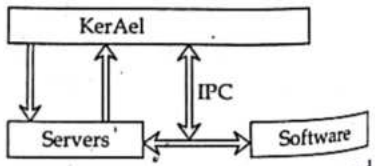
A kernel is classified into two main types:

i) Monolithic Kernel



Monolithic kernels are those kernels where the user services and the kernel services are implemented in the same memory space *i.e.*, different memory for user services and kernel services are not used in this case. By doing so, the size of the kernel is increased and this, in turn, increases the size of the operating system. As there is no separate user space and kernel space, so the execution of the process will be faster in monolithic kernels.

ii) Micro Kernel



A microkernel is different from monolithic kernel because in a microkernel, the user services and kernel services are implemented into different spaces *i.e.*, we use user space and kernel space in case of micro kernels. As we are using user space and kernel space separately, so it reduces the size of the kernel and this, in turn, reduces the size of operating system.

As we are using different spaces for user services and kernel service, so the communication between application and services is done with the help of message parsing and this, in turn, reduces the speed of execution.

6.2 TASK, PROCESS AND THREADS

6.2.1 Task

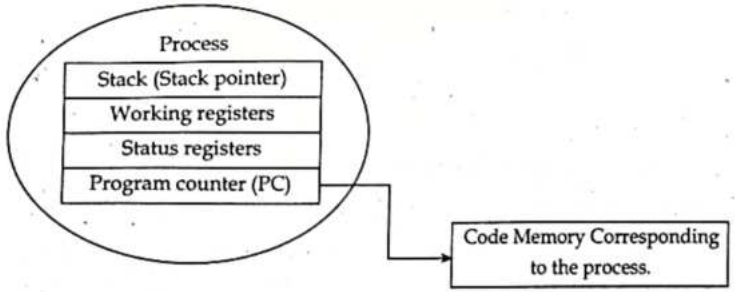
A task is a basic unit of programming that an operating system controls. Depending on how the operating system defines a task in its design, this unit of programming may be an entire program or each successive invocation of a program. Since one program may make requests or other utility programs, the utility programs may also be considered tasks (or sub-tasks). All of today's widely-used operating systems support multitasking, which allows multiple tasks to run concurrently, taking turns using the resources of the computer.

6.2.2 Process

A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an "active" entity as opposed to program which is considered to be a "passive" entity. Attributes held by process include hardware state, memory, CPU etc.

6.2.2.1 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 meaning occupied by the process is separated into 3 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.

6.2.2.2 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 states 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

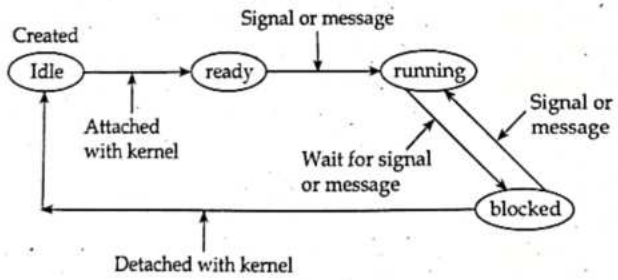
It is the state where the source code instructions corresponding to the process are being executed. The process execution happens in this state.

Blocked/waiting state

Execution of task codes suspends after saving the needed parameters into its context. It needs some IPC (input) or it needs to wait for an event or wait for higher priority task to block to enable running after blocking.

Terminated/finished state

The created task has memory deallotted to its structure *i.e.*, task be deleted such that it frees the memory.



6.2.2.3 Process Control Block (PCB)

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc. It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Structure of the process control block

The process control stores many data items that are needed for efficient process management. Some of these data Items are explained with the help of the given diagram.

Process state
Process number
Program counter
Registers
Memory limits
List of open files
⋮

The following are the data items:

- **Process state:** This specifies the process state *i.e.*, new, ready, running, waiting or terminated.
- **Process number:** This shows the number of the particular process.
- **Program counter:** This contains the address of the next instruction that needs to be executed in the process.
- **Registers:** This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers, etc.
- **List of open files:** These are the different files that are associated with the process.
- **CPU Scheduling Information:** The process priority, pointers to scheduling queues etc is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.
- **Memory management information:** The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.
- **I/O status information:** This information includes the list of I/O devices used by the process, the list of files etc.
- **Accounting Information:** The time limits, accounts numbers, amounts of CPU used, process number etc. are all part of the PCB accounting information.

6.2.3 Threads

A thread is a path of execution within a process. A process can contain multiple threads. A thread is also known as light weight process. The idea is to achieve parallelism by dividing a process into multiple threads. For examples, in a browser, multiple tabs can be different threads. MS word uses multiple threads one thread to format the text, another thread to process inputs, etc.

Process vs Threads?

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a results threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Advantages of threads/Multithread

i) **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

ii) **Faster context switch:** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

iii) **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

iv) **Resources sharing:** Resources like code, data, and files can be shared among all threads within a process.

v) **Communication:** Communication between multiple threads is easier, as the threads shares common address space. While in process we have to follow some specific communication technique for communication between two process.

vi) **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

6.2.4 User Level and Kernel Level Threads

User level threads

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter (PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronization for user-level threads.

Kernel-level threads

Kernel level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Relationship between user level thread and kernel level threads:

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

Many-to-one model:

Many-to-one model maps many user level threads to one kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

One-to-one model:

There is one-to-one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Many-to-many model:

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads. In this model, developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor machine.

6.2.5 Threads Libraries

A thread library provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist 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 operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main threads libraries are in use today:

i) POSIX threads
ii) Win 32 threads
iii) Java threads

i) POSIX threads

The POSIX standard defines the specification for pThreads, not the implementation.

- pThreads are available on Solaris, Linux, Mac OSX, Tru 64, and via public domain shareware for windows
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function.

Thread call	Description
pThread_create ()	Creates a new thread
pThread_exit ()	Terminates the calling threads
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 threads run.
pThread_attr_init ()	Create and initialize a thread's attributes.
pThreads_attr_destroy ()	Releases a thread's attributes

ii) Win 32 threads

Similar to pThreads. Win 32 threads are supported by various flavors of the windows OS. The win 32 API libraries provide a standard set of win 32 thread creation and management function. Win 32 thread library is a kernel level library.

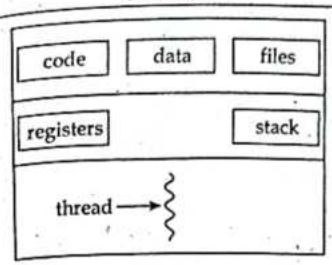
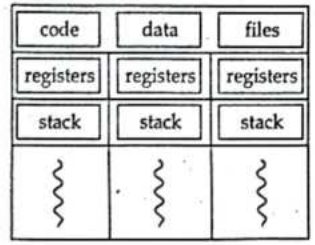
Thread call	Description
Create Thread ()	Creates a new thread.
Suspend Thread ()	Temporarily suspends thread execution.
Resume Thread ()	Wakes up a suspended thread.
Exit Thread ()	It terminates a thread and allocates the thread stack resources along with other resources that were held by it.

iii) Java threads

All Java programs use threads-even "common" single-threaded ones. Because Java does not support global variables, threads must be passed a reference to a shared object in order to share data. Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependent, and may be one to one, many to many or many to one.

Threads 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.

6.2.6 Differences between Process and Thread

Process	Thread
 Single-threaded process	 Multi-threaded process
Program in execution.	Basic unit of CPU utilization.
Unit of allocation ▪ Resources, privileges, etc	Unit of execution ▪ PC, SP, registers ▪ PC-program counter ▪ SP-stack pointer
Heavy weight	Light weight
Inter-process communication is expensive: need to context switch Secure: one process cannot corrupt another process.	Inter-thread communication cheap: can use process memory and may not need to context switch Not secure: a thread can write the memory used by another thread.
Process are typically independent	Thread exist as subsets of a process
Process carry considerable state information.	Multiple thread within a process share state as well as memory and other resources.
Process have separate address space.	Thread share their address space.
Process interact only through system-provided inter-process communication mechanisms.	Context switching between threads in the same process is typically faster than context switching between processes.

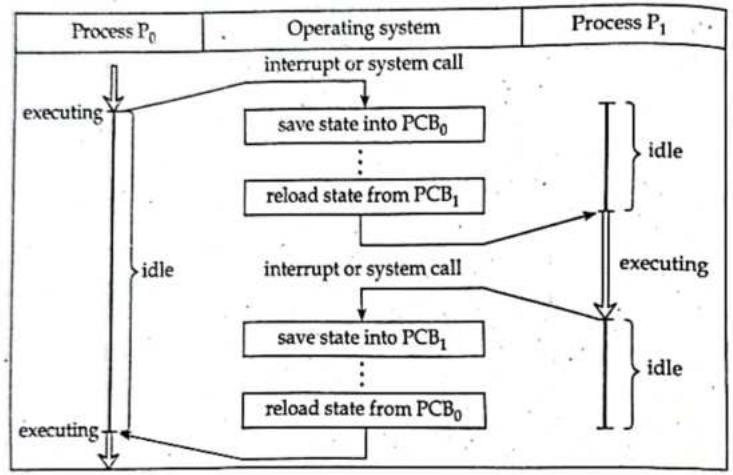
6.3 MULTIPROCESSING AND MULTITASKING

In a uni-processor system only one process executes at a time. Multiprocessing is the use of two or more CPUs within a single computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Now since there are multiple processors available, multiple processes can be executed at a time. These multiprocessors share the computer bus, sometimes the clock, memory and peripheral devices also.

As the name itself suggests, multitasking refers to execution of multiple tasks at a time. Multitasking is a logical extension of multiple

programming. The major way in which multitasking differs from multiprogramming is that multiprogramming work solely on the concept of context switching whereas multitasking is based on time sharing alongside the concept of context switching.

A. Context Switching



Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

B. Types of Multitasking

In this technique, multiple tasks, also known as processes, share common processing resources such as a CPU. Multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types.

Co-operative multitasking

Windows and Mac OS used cooperative multitasking. A windows program would do some small unit of work in response to a message and then relinquish the CPU to the operating system until the program got another message. That worked well, as long as all programs were written with consideration for other programs and had no bugs.

Preemptive multitasking

Desktop operating system use preemptive multitasking. Unix used this form of multitasking from the beginning. Windows starting using

preemptive multitasking with windows NT and windows 95. Macintosh gained preemptive multitasking with OSX.

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

Real time task scheduling essentially refers to determining the order in which the various tasks are to be taken up for execution by the operating system. Every operating system relies on one or more task schedulers to prepare the schedule of execution of various tasks it needs to run. Each task scheduler is characterized by the scheduling algorithm it employs.

When a process transitions from running state to blocked/wait state to running state to ready state from blocked/wait state to completed state the process scheduling choice is made.

The following factors should be considered while choosing a scheduling criterion.

i) CPU utilization

The CPU utilization should always be high as a scheduling requirement. CPU usage is a metric that indicates how much of the CPU is being used.

ii) Throughput

This represents the number of processes completed per unit of time. A good schedule should always have a better throughput.

iii) Turnaround time

It is the time it takes for a procedure to complete its execution. It covers the time spent waiting for main memory, time spent in the ready queue, time spent completing I/O operations, and time spent in execution by the process. For an effective scheduling algorithm, the turnaround time should be kept to bare minimum.

iv) Waiting time

It is the amount of time a process spends in the 'Ready' queue, waiting for CPU time to execute. For a good scheduling algorithm, the waiting time should be as short as possible.

v) Response time

The response time is the time between when a process is submitted and when it receives its first response. For a good scheduling algorithm, the response time should be as short as possible.

In relation with CPU scheduling, the operating system maintains multiple queues, and a process goes through these queues on its way to execution completion. The several queues that the OS maintains in relation with CPU scheduling are;

- **Job queue:** All of the system's processes are stored in the job queue.
- **Ready queue:** Comprises all of the processes that are ready to run and are waiting for their turn of the CPU.
- **Device queue:** Contains a list of processes that are waiting for an I/O device to be available.

The scheduling algorithm can be classified as;

1. Non-Preemptive Scheduling

It is used in systems that use a non-preemptive multitasking approach. The presently running task/process is permitted to run until it terminates or enters the wait state while waiting for an I/O or system resources in the scheduling type. The following are the types of non-preemptive scheduling.

a) First come first served (FCFS)/FIFO scheduling

The FCFS scheduling technique assigns CPU time to processes in the order that they appear in the ready queue. The first process that is entered gets served first. For example, a ticketing reservation system is which customers must queue and first person in line is served first.

b) Last come first served (LCFS)/LIFO scheduling

The LCFS scheduling method also assigns CPU time to processes in the ready queue in the order in which they are entered. The process that was entered last gets handled first.

c) Shortest job first (SJF) scheduling

When a process relinquishes the CPU, the SJF scheduling algorithm sorts the ready queue to select the processes with the shortest anticipated completion time. The process with the shortest estimated run time is scheduled first, then the process with the next shorted projected run time, and so on.

d) Priority based scheduling

When compared to other low priority processes in the ready queue, this scheduling technique assures that a process with a high priority is served first. The SJF algorithm can be thought of as a priority-based scheduling system, with each task being prioritized according to the amount of time it takes to perform it. Another method of giving a priority to a task/process is to do so at the moment of task/process is created. The priority is a number that ranges from 0(zero) to the highest priority supported by the operating system. A priority value of 0(zero) signifies the highest priority for the windows CE operating system.

2. Preemptive Scheduling

In systems that use the preemptive multitasking model, preemptive scheduling is used. Every job in the ready queue has a chance to run during this scheduling. The type of preemptive scheduling algorithm determines when and how often each process is allowed to run. The scheduler can preempt (temporarily halt) the currently running process and choose another task from the ready queue to execute in this scheduling mechanism. The job that the scheduler has preempted is moved to the ready queue. Preemption is the act of the scheduler shifting a running process in to a ready queue without the processes requesting it. The following are the several types of preemptive scheduling used in process scheduling.

a) Preemptive SJF scheduling/Shortest Remaining Time (SRT)

When a new process joins the ready queue, the preemptive SJF scheduling algorithm sorts the ready queue and verifies whether the new process's execution time is less than the remainder of the total expected time for the presently executing process. If the new process's execution time is less than the current one, the current one is preempted and the new one is scheduled to run.

b) Round Robin Scheduling

Each process in the ready queue is executed for a pre-defined time period in this scheduling mechanism. The execution begins with the first process in the ready queue being selected. It runs for a pre-defined time slice, and when that time elapses or the process completes before that time elapses, the next process in the ready queue is chosen for execution. The scheduler selects the first process in the ready queue for execution again when each process in the ready queue has completed its pre-defined time period, and the sequence is repeated. Round robin scheduling is comparable fo FCFS scheduling, with the addition of time slice preemption to switch between processes in the ready queue.

c) Priority Based Scheduling

Except for the switching of execution across processes, the priority based preemptive scheduling algorithm is identical to the non-preemptive priority based scheduling algorithm. Any high priority process that enters the ready queue is scheduled for execution immediately in preemptive scheduling, whereas in non-preemptive scheduling, any high priority process that enters the ready queue is scheduled only after the currently executing process has completed its execution or has voluntarily relinquished the CPU.

6.5 TASK SYNCHRONIZATION

- In general, a task must synchronize its activity with other task to execute a multithreaded program properly.
- Consider a situation where two processor try to access a shared memory area where one process tries to write to memory allocation when other process try to read.
- This scenario leads to conflicts *i.e.*, task synchronization issue.

6.5.1 Task Communication/Synchronization Issues

1. **Deadlock**

It creates a situation where none of the processes are able to make any progress in their execution.

a) Coffman conditions

The different condition favouring a deadlock situation are;

i) **Mutual exclusion:** The condition in which a process can hold one resource at a time.
For example; Hardware in a embedded device

ii) **Hold and wait:** The condition in which a process hold a shared resource by acquiring the lock controlling the shared access and waiting for additional resource held by other processor.

iii) **No resource preemption:** The criteria that operating system cannot take back a resource from a process which is currently holding it and resource can only be released voluntarily by the processor holding it.

iv) **Circular wait:** A process is waiting for a resource which is currently held by another process which in turn waiting for a resource held by the first resource. In general, there exists a set of waiting process P1 and P1 is waiting for a resource held by P0, P1, ..., Pn with P0 is waiting for a resource held by P0, ..., Pn is waiting for a resource held by P0 and P0 is waiting for resource held by Pn and so on. This forms a circular wait queue.

b) Dead lock Handling

The OS may adopt any of the following techniques to detect and prevent deadlock condition.

i) **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 happening a deadlock.

ii) **Detect and recover:** This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arises at a traffic condition. When the vehicles from different direction compete to cross the junction, deadlock condition is resulted.

iii) **Avoid deadlocks:** Deadlock is avoided by the careful resource allocation techniques by operating system. It is similar to the traffic light mechanism at junctions at avoid the traffic jam.

iv) **Prevent deadlocks:** Prevent the deadlock condition by negating the below condition of the deadlock situation.

- Ensure that a process does not hold any other resources, when it request a resource.
- A process must request all its required resource and the resources should be allocated before the process begins its execution.

2. **Live lock**

It is similar to a deadlock, except that the status of the process involved in the live lock constantly change with regard to one another. In this process are not in the waiting state and they are running concurrently.

3. **Starvation**

It occurs when a low priority program is requesting for a system resource, but are not able to execute because a higher priority program utilizing that resource for an extended period.

4. **Racing**

The situation where multiple processes compete with each other to access and manipulate shared data concurrently is called racing or race condition.

6.5.2 Task Synchronization Techniques

Task synchronization is essential for

i) Getting shared access to resources

ii) Communicating

1. **Semaphore**

- A semaphore is a kernel object that one or more threads can acquire or release for synchronization or mutual exclusion purposes.
- The kernel assigns an associated SCB, a unique ID, a value (binary or count) and a task waiting list to a semaphore when it is first established.

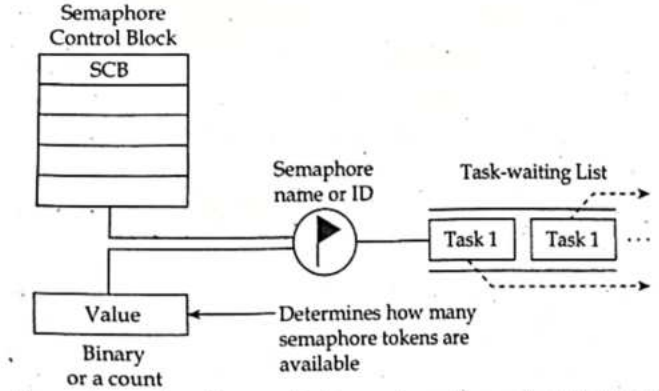


Figure: A semaphore it's associated parameters and supporting data structure

A semaphore is like a key that allows a task to carry out some operation or to access a resource. If the task can acquire the semaphore, it can carry out the intended operation or access the resource. A single semaphore can be acquired a finite number of times.

The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created. The task waiting list tracks all tasks blocked while waiting on an unavailable semaphore. These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order or highest priority first order. When an unavailable semaphore becomes available, the kerned allows the first task in the task-waiting list to acquire it. The kernel moves this unblocked task either to the running state, if it is the highest priority task, or to the ready state, until it becomes the highest priority task and is able to run.

a) Binary Semaphores

A binary semaphore can have a value of either 0 or 1. When a binary semaphore's value is 0, the sema-phore is considered unavailable (or empty); when the value is 1, the binary semaphore is considered available (or full).

Note that when a binary semaphore is first created, it can be initialized to either available or unavailable (1 or 0, respectively).

b) Counting Semaphores

A counting semaphore uses a count to allow it to be acquired or released multiple times. When creating a counting semaphore, assign the semaphore a count that denotes the number of semaphore tokens it has initially. If the initial count is 0, the counting semaphore is created in the unavailable state. If the count is greater than 0, the semaphore is created in the available state, and the number of tokens it has equals its count.

6.6 DEVICE DRIERS

- Device driver is a software program that connects the operating system to the hardware.
- The OS kernel's architecture now allows direct device access from user applications.
- All device access should go through the OS kernel, which will then route it to the appropriate hardware peripherals.
- Drivers for hardware peripherals are in charge of initiating and managing communication with them.
- Device drivers are loaded into RAM by the operating system when the device is booted.

- Installable drivers are device drivers that must be installed in order to access a device.
- The OS loads the appropriate driver into memory whenever the device is connected.
- "dll" files are the most common type of driver file.
- Drivers can operate in either user or kernel space.
- Drivers that operate in user space are known as user mode drivers, whereas those operate in kernel space are known as kernel mode drivers.

A device driver implements the following;

I) Device initialization and interrupt configuration

The device's various registers are configured by the driver. The interrupt configuration section is responsible for configuring the interrupts that must be linked to the hardware. The basic interrupt configuration involves.

- Set the type of interrupt (edge triggered or level triggered), activate the interrupts and prioritize them.
- Bind the interrupt to an interrupt request (IRQ). Through IRQ, the the processor detects an interrupt. The interrupt controller generates these IRQs. The interrupt must be connected to an IRQ in order to be identified and interrupted.
- With an IRQ, create an interrupt service routine (ISR). ISR is an interrupt handler. An ISR must be linked with an IRQ in order to service an interrupt.

II) Interrupt handling and processing

An interrupt is served based on its priority, the related ISR is executed. An interrupt's processing is handled by an interrupt service routine (ISR). The ISR can handle the entire interrupt processing or activate an interrupt service thread (IST). The ISR's interrupt processing is handled by the IST.

III) Client Interfacing

For interacting and synchronizing with user programs and drivers, the client interfacing implementation makes advantage of the embedded OS's interprocess communication features. *For example;* the client interfacing code can signal an event to alert a user program that an interrupt has occurred and that the data received from the device has been placed in a shared buffer.

OLD EXAM QUESTION SOLUTION (TU)

1. Explain the basic functions of real-time kernel. [2069 Bhadra]

Answer:

A real-time kernel is software that manages the time of microprocessor to ensure that time critical events are processed as efficiently as possible. It is very specialized, containing only the minimal set of services required to keep the user up and running application and tasks.

Basic functions of real-time kernel are;

i) Task/process management
ii) Task/process scheduling
iii) Task/process synchronization
iv) Memory management
v) Device management
vi) Interrupt handling
vii) I/O communication
viii) Error/exception handling

For description: See the topic 6.1 (A) of chapter 6.

2. Distinguish between process and thread. Write different states of task with appropriate example. [2070 Magh]

Answer:

First part: See the topic 6.2.6 of chapter 6.

Second part: See the topic 6.2.2.2 of chapter 6.

3. What are the advantages of multi-threading program? Write a simple multithreading program in C. [2070 Magh]

Answer:

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time.

Advantages of multiple threads to execute:

See 6.2.3 of chapter 6.

Second part:

```
#include <windows.h>
#include <stdio.h>
// child thread
void childThread (void)
{
    char i;
    for (i = 0; i< = 10; i ++)
```



```
    {
        printf ("Executing child thread: counter = % d\n", i);
        sleep (500);
    }
}
// primary thread
int main (int argc, char * argv [ ])
{
    HANDLE hThread;
    DWORD dwThreadID;
    char i;
    hThread = Create Thread (NULL, 1000
                                (LPTHREAD_START_ROUTINE))
    childThread, NULL, 0, and dwThread ID);
    if (hThread == NULL)
    {
        printf ("Thread Creation Failed \n Error No: % d\n,"
                                                GetLastError());
        return 1;
    }
    for (i = 0; i < = 10; ++ i)
    {
        printf ("Executing Main Thread: Counter = %d\n", i);
        sleep (500);
    }
    return 0;
}
```

Output:

```
Executing Main Thread: Counter = 0
Executing Child Thread: Counter = 0
Executing Main Thread: Counter = 1
Executing Child Thread: Counter = 1
Executing Main Thread: Counter 2
Executing Child Thread: Counter = 2
Executing Child Thread: Counter = 3
Executing Main Thread: Counter = 3
Executing Child Thread: Counter = 4
Executing Main Thread: Counter = 4
```

Executing Child Thread: Counter = 5
Executing Main Thread: Counter = 5
Executing Child Thread: Counter = 6
Executing Main Thread: Counter = 6
Executing Child Thread: Counter = 7
Executing Main Thread: Counter = 7
Executing Main Thread: Counter = 8
Executing Child Thread: Counter = 8
Executing Main Thread: Counter = 9
Executing Child Thread: Counter = 9
Executing Main Thread: Counter = 10
Executing Child Thread: Counter = 10

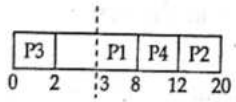
4. Describe the context switching process in detail.
Three processes with process IDs P1, P2, P3 with estimated completion times 6, 8, 2 milliseconds respectively enters the ready queue together. Process P4 with estimated execution completion time 4 milliseconds enters the ready queue after 1 millisecond. Calculate the waiting time and turn-around-time for each process and the average waiting time and turn-around time in the non-preemptive shortest-job-first scheduling. [2070 Bhadra]

Answer:

First part: See the topic 6.3 (A) of chapter 6.

Second part:

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



Waiting time = (Execution starting point-entry point)
Turnaround time = (completion point – Entry point)

Waiting time calculation

P3 = (0 – 0) = 0 ms
P1 = (2 – 0) = 2 ms
P4 = (8 – 3) = 5 ms
P2 = (12 – 0) = 12 ms

Average waiting time = $\frac{0+2+5+12}{4}$ = 4.75 ms

Turnaround time calculation

P3 = (2 – 0) = 2 ms
P1 = (8 – 0) = 8 ms
P4 = (12 – 3) = 9 ms
P2 = (20 – 0) = 20 ms

Average TAT = (2 + 8 + 9 + 20)/4 = 9.75 ms

5. Explain in detail the coffman conditions that favor deadlock. Differentiate between user-level threads and kernel-level threads. [2070 Bhadra]

Answer:

First part: See the topic 6.5.1 [2(i)] of chapter 6.

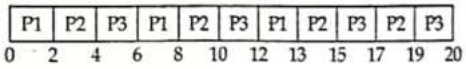
Second part: See the topic 6.2.4 of chapter 6.

6. Differentiate between multiprocessing and multi tasking in RTOS. Three processes with process IDs, P_1, P_2, P_3 with estimated completion time 5, 8, 7 ms respectively, enters the ready queue together in order P_1 P_2 P_3 calculate waiting time and TAT for ach process and average waiting time and TAT. Assume there is no I/O waiting for the process and PR (Round Robin) algorithm with time slice = 2 ms. [2072 Ashwin]

Answer:

First part: See the topic 6.3 of chapter 6.

Second part:



Turnaround time calculation

P1 = 13 – 0 = 13 ms
P2 = 19 – 0 = 19 ms
P3 = 20 – 0 = 20 ms

Average turnaround time

= $\frac{(13+19+20)}{3}$ = 17.33 ms

Waiting time calculation

(Waiting time = Turnaround time – Completion time)
P1 = 13 – 5 = 8 ms
P2 = 19 – 8 = 11 ms
P3 = 20 – 7 = 13 ms

Average waiting time = $\frac{(8+11+13)}{3}$ = 10.66 ms

7. Define thread in RTOS. Explain the concept of multi threading and why is it necessary? Consider three processes with process ID's P1, P2, P3 with estimated completion time 9, 6, 3 ms respectively enter the ready queue in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for process and also calculate the average waiting time and average TAT in Round Robin algorithm with time slice = 3 ms. Assume there is no I/O waiting for the process. [2072 Magh]

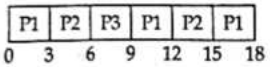
Answer:

First part: A thread is a path of execution within a process.

A thread is also known as light weight process.

Second part: Same as question number 3 (exam solution TU)

Third part:



Turn Around Time Calculation

P1 = 18 – 0 = 18 ms

P2 = 15 – 0 = 15 ms

P3 = 9 – 0 = 9 ms

$$\text{Average Turn Around Time} = \frac{18 + 15 + 9}{3} = 14 \text{ ms}$$

Waiting time calculation

P1 = 18 – 9 = 9 ms

P2 = 15 – 6 = 9 ms

P3 = 9 – 3 = 6 ms

$$\text{Average waiting time} = \frac{9 + 9 + 6}{3} = 8 \text{ ms}$$

8. Define kernel and differentiate between monolithic kernel and microkernel. Consider three processes with process ID's P1, P2, P3 with estimated completion time 10, 5, 7 ms and priorities 1, 3, 2 (0-highest priority, 3-lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Calculate the waiting time and Turn Around Time (TAT) for each process and also calculate the average waiting time and average TAT for priority based preemptive scheduling algorithm. Assume there is no I/O waiting for the processes. [2073 Bhadra]

Answer:

First part: The kernel is the operating system's brain and it's in charge of managing system resources as well as communication between hardware and other system services.

Second part:

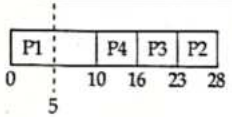
Differences between monolithic kernel and microkernel are;

Monolithic kernel	Microkernel
i) Monolithic kernel is larger than microkernel	i) It is smaller in size
ii) Fast execution	ii) Slow execution
iii) It is hard to extend	iii) It is easily extendible
iv) If a service crashes, the whole system crashes in monolithic kernel.	iv) If a service crashes, it does effect on working on the microkernel.
v) To write a monolithic kernel less code is required.	v) To write a microkernel more code is required.
vi) *For example*; Linux, BSDs (free BSD, open BSD, net BSD) etc.	vi) *For example*; QNX, symbian, L4Linux, etc.

Third part:

Given information from the question are tabulated as shown below;

Process	Entry Time	Completion Time	Priority
P1	0	10	1
P2	0	5	3
P3	0	7	2
P4	5 ms after P1 starts	6	0



Turn Around Time Calculation (Completion point – Entry point)

P1 = 10 – 0 = 10 ms

P2 = 28 – 0 = 28 ms

P3 = 23 – 0 = 23 ms

P4 = 16 – 5 = 11 ms

$$\text{Average Turn Around Time} = \frac{10 + 28 + 23 + 11}{4} = 18 \text{ ms}$$

Waiting Time Calculation (Execution starting point – Entry point)

P1 = 0 – 0 = 0 ms

P2 = 23 – 0 = 23 ms

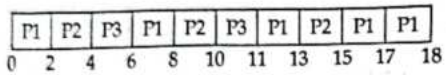
P3 = 16 – 0 = 16 ms

P4 = 10 – 5 = 5 ms

$$\text{Average waiting time} = \frac{0 + 23 + 16 + 5}{4} = 11 \text{ ms}$$

9. Consider three processes with process IDs P1, P2, P3 with estimated completion time 9, 6, 3 ms respectively, enters 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 in RR (Round–Robin) algorithm with time slice 2 ms. Assume there is no I/O waiting for the process. [2073 Magh]

Answer:



Turnaround time calculation

P1 = 18 - 0 = 18 ms

P2 = 15 - 0 = 15 ms

P3 = 11 - 0 = 11 ms

$$\text{Average turnaround time} = \frac{18 + 15 + 11}{3} = 14.67 \text{ ms}$$

Waiting time calculation

P1 = 18 - 9 = 9 ms

P2 = 15 - 6 = 9 ms

P3 = 11 - 3 = 8 ms

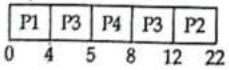
10. Write any four differences between thread and process. Three processes P1, P2 and P3 with estimated completion time 4, 10, 5 ms and priorities 1, 3, 2 respectively enters the ready queue together. A new process P4 with estimated completion time 3 ms and priority 0 enters the ready queue after 5 ms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

Answer:

First part: See the topic 6.2.6 of chapter 6.

Second part:

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



Turn Around Time Calculation

P1 = 4 - 0 = 4 ms

P2 = 22 - 0 = 22 ms

P3 = 12 - 0 = 12 ms

P4 = 8 - 5 = 3 ms

$$\text{Average turnaround time} = \frac{4 + 22 + 12 + 3}{4} = 10.25 \text{ ms}$$

Waiting Time calculation

P1 = 0 - 0 = 0 ms

P2 = 12 - 0 = 12 ms

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

P4 = 5 - 5 = 0 ms

$$\text{Average waiting time} = \frac{0 + 12 + 7 + 0}{4} = 4.75 \text{ ms}$$

11. Define threads and differentiate between user level thread and kernel level thread. Three processes with IDs P1, P2, P3 with estimated completion time 6, 8, 2 ms respectively enters the ready queue together in the order P1, P2, P3. Process P4 with the estimated execution time 4 ms enters the ready queue after 1 ms. Calculate the waiting time and turn- around time for each process and the average waiting time and TAT in the non-preemptive shortest-job-first scheduling.

Answer:

First part: See the topic 6.2.3 and 6.2.4 of chapter 6.

Second part: Same as question number 4 [Old Exam Question Solution TU].

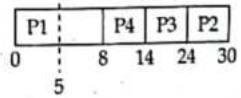
12. Explain process life cycle with process state diagram. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 6, 10 ms and priorities 0, 3, 2 (0-highest, 3-lowest) respectively, enters the ready queue together in order P1, P2, P3 (assume only P1 is present in the ready queue when the scheduler picks it up and P2 and P3 enter ready queue after that). Now the process P4 with estimated completion time 6 ms and priority 1 enters the ready queue after 5 ms 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]

Answer:

First part: See the topic 6.2.2 of chapter 6.

Second part:

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



Now,

Waiting Time Calculation

P1 = 0 - 0 = 0 ms

P2 = 24 - 0 = 24 ms

P3 = 14 - 0 = 14 ms

P4 = 8 - 5 = 3 ms

$$\text{Average Waiting Time} = \frac{0 + 24 + 14 + 3}{4} = 10.25 \text{ ms}$$

Turn Around Time Calculation

P1 = 8 - 0 = 8 ms

P2 = 30 - 0 = 30 ms

P3 = 24 - 0 = 24 ms

P4 = 14 - 5 = 9 ms

$$\text{Average Turn Around Time} = \frac{8 + 30 + 24 + 9}{4} = 17.75 \text{ ms}$$

13. Differentiate between multi processing and multitasking. Suppose three processes with process ID's P1, P2 and P3 with estimated completion time 4, 10, 5 ms and priorities 1, 3, 2 (0-highest priority and 3-lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 3 ms and priority 0 enters the ready queue after 5 ms of start of operation. Calculate the waiting time and average TAT for priority based preemptive scheduling algorithm. Assume there is no I/O waiting for the process. [2077 Poush/2078 Baisakh]

Answer:

First part: See the topic 6.3 of chapter 6.

Second part: Same as question number 10 [Old exam question solution TU]

14. Differentiate between real time OS and general purpose OS. Explain different synchronization issues. [2078 Baisakh]

Answer:

First part: See the topic 6.1.2 of chapter 6.

Second part: See the topic 6.5.1 of chapter 6.

OLD EXAM QUESTIONS SOLUTION (PoU)

1. Describe the major functions of real-time kernel. [2016 Fall]

Answer: Same as question number 1 (Old Exam Question Solution TU).

2. List out the difference between process and thread? Explain various state of process. [2016 Spring]

Answer:

First part: See the topic 6.2.5 of chapter 6.

Second part: See the topic 6.2.2 of chapter 6.

3. What are task states? Describe task scheduling. [2017 Fall]

Answer:

First part: See the topic 6.2.2 of chapter 6.

Second part: See the topic 6.4 of chapter 6.

4. How semaphore can be used for synchronizing execution of multiple task and global resource sharing? [2017 Fall]

Answer:

Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource. The following examples and general discussions illustrate using different types of semaphores to address common synchronization design requirements effectively, as listed:

- Wait-and-signal synchronization,
- Multiple-task wait-and-signal synchronization,
- Single shared-resource-access synchronization,
- Recursive shared-resource-access synchronization, and
- Multiple shared-resource-access synchronization

1. Wait and Signal Synchronization

Two tasks can communicate for the purpose of synchronization without exchanging data. *For example;* a binary semaphore can be used between two tasks to coordinate the transfer of execution control as shown below.

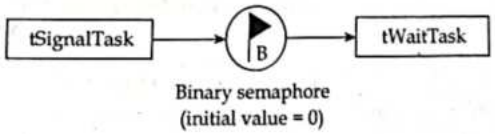


Figure: Wait-and-signal synchronization between two tasks

In this situation, a binary semaphore is initially unavailable (value of 0). tWait task has higher priority and run first. The task makes a request to acquire the semaphore but is blocked because the semaphore is unavailable.

This step gives the lower priority tSignal task a chance to run; at some point tSignal task release the binary semaphore and unblocks the tWaitTask.

```
tWaitTask ( )
{
        :
        Acquire binary semaphore token
        :
}
tSignalTask ( )
{
        :
        Release binary semaphore token
        :
}
```

Because tWait Task's priority is higher than tSinal Task's priority as soon as the semaphore is released, tWait Task pre-empts tSignal Task and starts to execute.

2. Multiple-Task wait and signal synchronization

- When coordinating the synchronization of more than two tasks, use the flush operation on the task-waiting list of a binary semaphore.

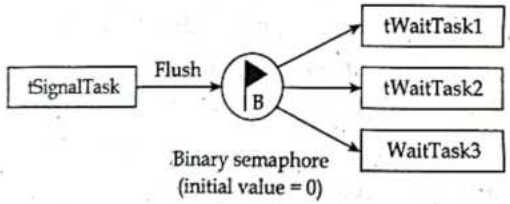


Figure: Wait-and-signal synchronization between multiple tasks

- The binary semaphore is initially unavailable. The higher priority tWait Task 1,2, and 3 all do some processing; when they are done; they may try to acquire the unavailable semaphore an as a result, blocks.

This action gives tSignal Task a chance to complete its processing and execute a flush command on the semaphore, effectively unblocking the three tWait Task.

3. Single shared resource access synchronization

One of the most common uses of semaphore is to provide for mutually exclusive access to a shared resource.

A shared resource might be a memory location, a data-structure or an I/O device essentially anything that might have to be shared between two or more concurrent threads of execution. A semaphore can be used to serialize access to a shared resource as shown below.

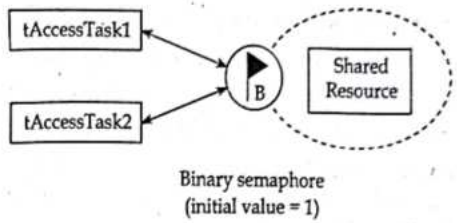


Figure: Single shared-resource-access synchronization

In this scenario, a binary semaphore is initially created in the available state (value = 1) and is used to protect the shared resource.

To access the shared resource, task 1 or 2 need to first successfully acquire the binary semaphore before reading from or writing to the shared resource.

```
tAccessTask ( )
{
        :
        Acquire binary semaphore token
        Read or write to shared resource
        Release binary semaphore token
}
```

- This code serializes the access to the resource. If tAccess Task 1 executes first it makes a request to acquire the semaphore and is successfully become the semaphore is available.
- Having acquired the semaphore; this task is granted access to the shared resource and can read and write to it.

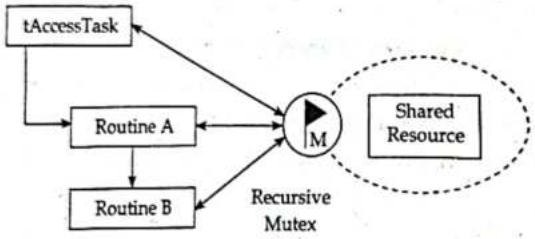
Mean while, the higher priority tAccess Task2 wakes up and runs due to timeout or some external events. It tries to access the same semaphore but is blocked because tAccess Task 1 currently has accessed to it. After tAccess Task 1 release the semaphore, tAccess Task 2 is unblocked and starts to execute.

One of the danger of this design is that any task can accidently release the binary semaphore, even one that never acquire the semaphore in the first place. If this issue were to happen in this scenario, both tAccess Task 1 and tAccess Taks 2 could end up acquiring the semaphore and reading and writing to the shared resource at the same time, which could lead to the incorrect program behavior.

To ensure that this problem does not happen, use a mutez semaphore instead. Because a mutex supports the concept of ownership, it ensures that only the task that successfully acquire (locked) the mutex can release (unlock) it.

4. **Recursive Shared-Resource access synchronization**

- Sometimes a developer might want to access a shared resource recursively. This situation might exist if tAccess Task calls routine A that calls routine B, and all three need access to the shared resource as shown in figure below.



- If a semaphore were used in this scenario, the task would end up blocking; causing a deadlock, when a routine is called from a task the routine effectively becomes a part of the task.
- When routine A runs, therefore it is running as a part of tAccessTask. Routine A trying to acquire the semaphore is effectively the same as tAccess Task trying to acquire the same semaphore. In this case, tAccess Task would end up blocking while waiting for the unavailable semaphore that it already has.
- One solution to this situation is to use a recursive mutex After tAccessTaks locks the mutex, the task own it. Additional attempts from the task itself or from routine that it calls to lock the mutex succeeded. As a result, when routine A and routine B attempt to lock the mutex, they succeed without blocking.

```
tAccessTask ( )
{
        :
        Acquire the mutex
        Access the shared resource
        Call routine A
        Release mutex
        :
}
```



```
Routine A ( )
{
        :
        Acquire the mutex
        Access shared resource
        Call routine B
        Release mutex
        :
}
Routine B ( )
{
        :
        Acquire the mutex
        Access shared resource
        Release mutex
        :
}
```

5. **Multiple shared resource access synchronization**

- For case in which multiple equivalent shared resources are used a counting semaphore comes in handy as shown in figure below.

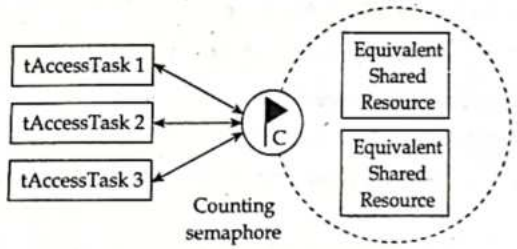


Figure: Single shared-resource-access synchronization

- Note that this scenario does not work if the shared resource are not equivalent. The counting semaphore's count is initially set to number of equivalent shared resources; in this example 2 as a result, the first two tasks requesting a semaphore token are successful. However, the third task ends up blocking until one of the previous two tasks release a semaphore token, as shown in figure.

5. **What is semaphore? How semaphore can be used for global resource sharing?** **[2017 Fall]**

Answer:

First part: See the topic 6.5.2 (1) of chapter 6.

Second part: See question number 4 [Old Exam Question Solution Pokhara University]

6. Define real time operating system? Explain various stages of task. [2017 Spring]

Answer:

First part: See the topic 6.1 of chapter 6.

Second part: See the topic 6.2.2 of chapter 6.

7. Define scheduling? Explain various types of task scheduling techniques in RTOS. [2018 Spring]

Answer: See the topic 6.4 of chapter 6.

8. Explain task and state of tasks in a system with necessary diagram. [2018 Spring]

Answer: See the topic 6.2 of chapter 6.

9. Define real time operating system? Explain Round Robin and preemptive scheduling policies. [2019 Fall]

Answer:

First part: See the topic 6.1 of chapter 6.

Second part: See the topic 6.4 of chapter 6.

10. What do you mean by kernel? Describe the types of RT kernel. [2019 Spring]

Answer:

First part: See the topic 6.1.1 of chapter 6.

Second part: See the topic 6.1.4 of chapter 6.

11. List out the differences between process and thread. [2020 Fall]

Answer: See the topic 6.2.5 of chapter 6.

12. What do you understand by TCB in RTOS? What are the information contents of TCB? [2020 Fall]

Answer: See the topic 6.2.3 of chapter 6.

Chapter 7

CONTROL SYSTEM

★★★

★★★

7.1 INTRODUCTION

- Control physical system's output
  - By setting physical system's input
- Tracking
- Examples:
  - Cruise control
  - Thermostat control
  - Disk drive control
  - Aircraft altitude control
- Difficulty due to
  - Disturbance: wind, road, tire, brake; opening/closing door .....
  - Human interface: feel good, feel right .....

Tracking:

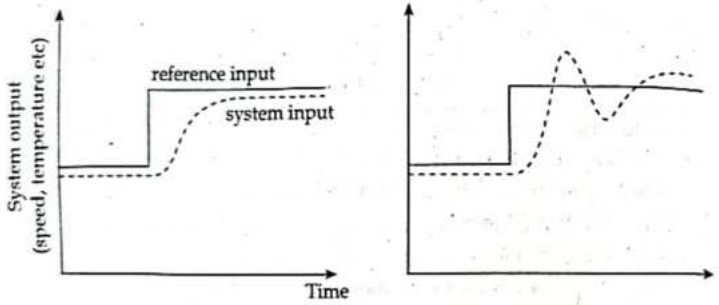


Figure: Good tracking

Figure: Bad tracking

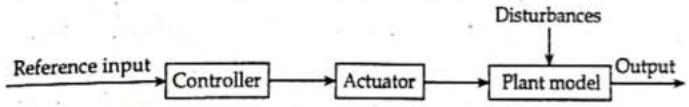
7.2 OPEN-LOOP AND CLOSE-LOOP CONTROL SYSTEM OVERVIEW

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 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.

- Plant
  - Physical system to be controlled
    - Car, plane, disk, heater
- Actuator
  - Device to control the plant
    - Throttle, wing flap, disk motor

- Controller
  - Designed product to control the plant
- Output
  - The aspect of the physical system we are interested in
    - Speed, disk location, temperature
- Reference
  - The value we want to see at output
    - Desired speed, desired location, desired temperature.
- Disturbance
  - Uncontrollable input to the plant imposed by environment
    - Wind, bumping the disk drive, door opening

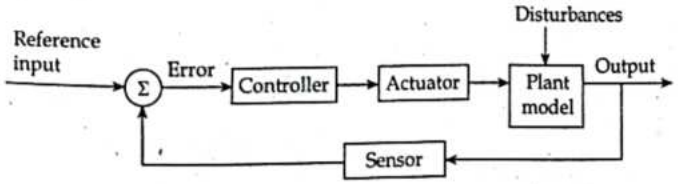
The general block diagram of open-loop control systems is shown in figure below:



Closed-loop control systems are the system, 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
  - Measure the plant output
- Error detector
  - Detect error

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



Comparison of open-loop and close-loop control system:

S.N.	Open-loop control system	Close of loop control system
1.	Feed forward system: output is not fed back.	Feedback 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, increase accuracy and supports optimization
4.	It is slow and unreliable but stable.	It is fast and more reliable but unstable.

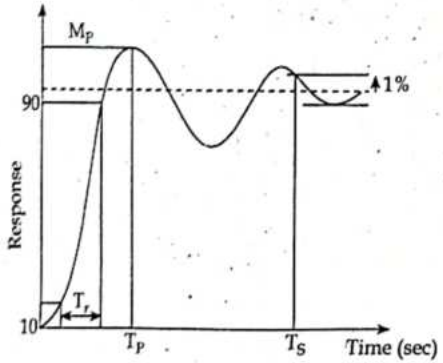
7.3 CONTROL SYSTEM AND PID CONTROLLERS

7.3.1 Control Objectives

The objective of control system design is to make a physical system behave in a useful fashion, in particular, by causing its output to track a desired reference input even in the presence of measurement noise, model error and disturbances. Satisfaction of this objective can be evaluated through several metrics specified relative to a step change in the control system input:

1. Stability: output remains bounded
2. Performance: how well an output tracks the reference
3. Disturbance: rejection
4. Robustness: ability to tolerate modeling error of the plant.

7.3.2 Performance



- Rise time
  - Time it takes from 10% to 90%
- Peak time

- Overshoot
  - Percentage by which peak exceed final value
- Settling time
  - Time it takes to reach 1% of final value

7.3.3 Transient Response and Steady State Response of Control System

As the name suggests, transient response of control system means changing so, this occurs mainly after two conditions and these two conditions are written as follows

- **Condition one:** just after switching 'on' the system that means at the time of application of an input signal to the system.
- **Condition second:** just after any abnormal conditions. Abnormal condition may include sudden change in the load, short circuiting etc.

Steady state occurs after the system becomes settled and at the steady system starts working normally. Steady state response of control system is a function of input signal and it is also called as forced response.

7.3.4 Modeling Real Physical System

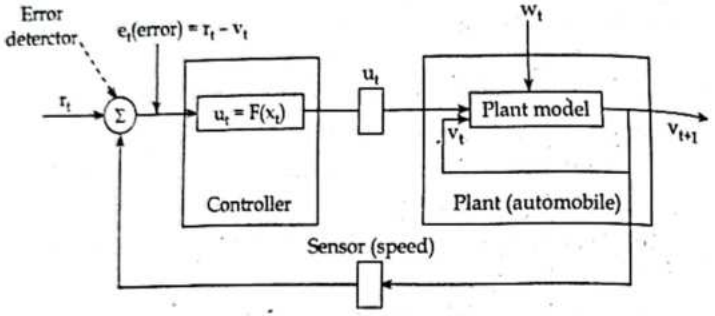
The accurate modeling of the behavior of the plant is an essential factor in control system design.

- May need to be done first
- Plant is usually on continuous time
  - Not discrete time
    - *For example,* car speed continuously react to throttle position, not at discrete interval
  - Sampling period must be chosen carefully
    - To make sure "nothing interesting" happen in between
    - *i.e.,* small enough
- Plant is usually non-linear
  - *For example;* shock absorber response may need to be 8^{th} order differential
- Iterative development of the plant model and controller
  - Have a plant model that is "good enough".

7.3.5 Controller Design: P

- Proportional controller
  - A controller that multiplies the tracking error by a constant
    - $u_t = P * (r_t - v_t)$
  - Close loop model with a linear plant
    - *For example,* $v_{t+1} = (0.7 - 0.5\ P) * v_t + 0.5\ P * r_t - w_t$

- P affects
  - Transient response
    - Stability, oscillation
  - Steady state tacking
    - As large as possible
  - Disturbance rejection
    - As large as possible



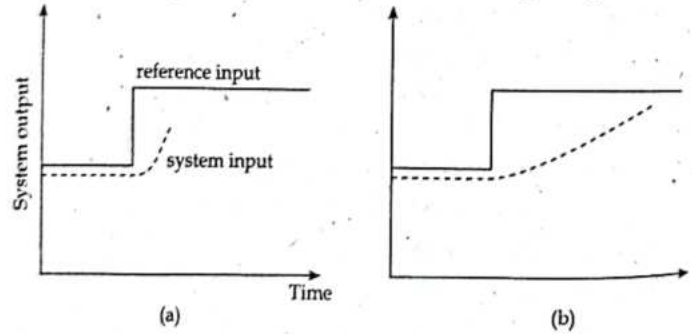
Car model, $v_{t+1} = 0.7\ v_t + 0.5u_t - w_t$

Control law, $u_t = P * (r_t - v_t)$

System model, $v_{t+1} = (0.7 - 0.5\ P)\ v_t + 0.5\ Pr_t - w_t$

7.3.6 Controller Design: PD

- Proportional and Derivative control
  - $u_t = P * (r_t - v_t) + D * ((r_t - v_t) - (r_{t-1} - v_{t-1}))$
    $= P * e_t + D * (e_t - e_{t-1})$
- Consider the size of error over time
- Intuitively
  - Want to "push" more if the error is not reducing fast enough
  - Want to "push" less if the error is reducing really fast.

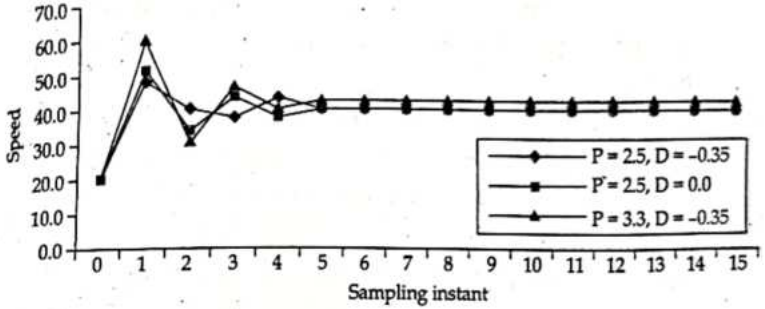


- Need to keep track of error derivative
  - *For example;* Cruise controller example
    - $v_{t+1} = 0.7\ v_t + 0.5\ u_t - w_t$
    - Let $u_t = P * e_t + D * (e_t - e_{t-1}) = r_t - v_t$

- $v_{t+1} = 0.7\ v_t + 0.5$
  $* (P * (r_t - v_t) + D * ((r_t - v_t) - (r_{t-1} - v_{t-1}))) - w_t$
- $v_{t+1} = (0.7\ v_t + 0.5 * (P + D)) * v_t + 0.5\ D * v_{t-1} + 0.5$
  $* (P + D) * r_t - 0.5D * r_{t-1} - w_t$
- Assume reference input and disturbance are constant, the steady - state speed is
  - $V_{ss} = (0.5\ P/(1 - 0.7 + 0.5P)) * r$
  - Does not depend on D

- P can be set for best tracking and disturbance control
- Then D set to control oscillation/overshoot/rate of convergence.

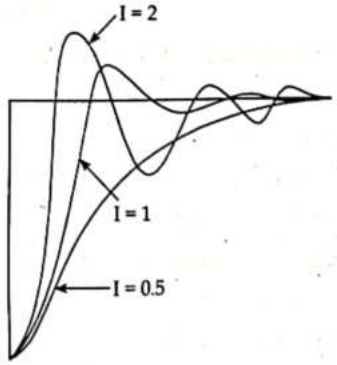
PD control Example



7.3.7 PI Control

- Proportional plus integral control
  - $u_t = P * e_t + I * (e_0 + e_1 + \cdots\cdots + e_t)$
- Sum up error over time
  - Ensure reaching desired output, eventually
  - V_{ss} will not be reached until $e_{ss} = 0$
- Use P to control disturbance
- Use I to ensure steady state convergence and convergence rate.

The following figure shows effect of different values of integral constant in the response of an arbitrary system for PD control action.



7.3.8 PID Controller

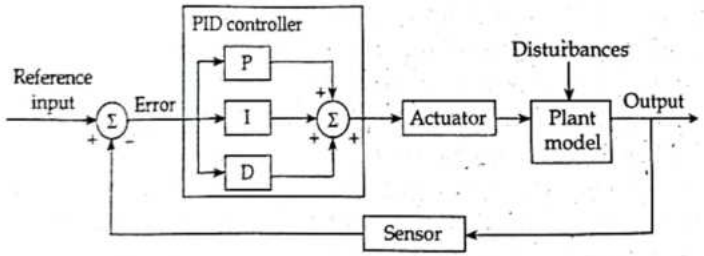
PID controller is a feedback controller that helps to attain a set point irrespective of disturbances or any variation in characteristics of the point of any form. It calculates its output based on the measured error and the three controller gain; proportional gain p, integral gain k, and derivative 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 controller 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

$$u(t) = P * e(t) + I * (e(0) + e(1) + e(2) + \cdots\cdots + e(t)) + D$$
$$* ((e(1) - e(0)) + (e(2) - e(1)) + \cdots\cdots + (e(t) - e(t-1)))$$

The general block diagram of PID controller is shown in figure below:



7.4 SOFTWARE CODING OF PID CONTROLLER

- Main function loops forever, during each iteration
  - Read plant output sensor
    - May require A2D
  - Read current desired reference input
  - Call pid update, to determine actuator value
  - Set actuator value
    - May require D2A


```
Void main ( )
{
        double sensor_value, actuator_value, error_current;
        PID_DATA pid_data;
        pidInitialize (&pid_data);
        while (1)
        {
                sensor_value = sensor Get value ();
                reference_value = reference Get value ();
                actuator_value = pid Update (&pid_data,
                                sensor_value, reference_value);
                Actuator set value (actuator_value);
        }
}
```

- Pgain, dgain, Igain are constants
- Sensor_value_previous
  - for D control
- error_sum
  - for I control

```
typedef struct PID_DATA
{
        double Pgain, Dgain, Igain;
        double sensor_value_previous; // find the deriuvative
        Double error_sum; // cumulative error
}
```

Computation

- $u_t = P * e_t + I * (e_0 + e_1 + \cdots\cdots + e_t) + D * (e_t - e_{t-1})$

```
double pidupdate (PID_DATA * pid_data, double sensor_value,
double reference_value)
{
        double Pterm, Iterm, Dterm;
        double error, difference;
        error = reference_value-sensor_value;
        Pterm = pid_data → Pgain * error; /* proportional term */
        pid_data → error_sum += error; /* current + cumulative */
        // the integral term
        Iterm = pid_data → Igain * pid_data → error_sum;
        difference = pid_data → sensor_value_prevoius = sensor-value;
        //update for next Iteration
```

```
    pid_data → sensor_value_previous = sensor_value;
    // the derivative term
    Dterm = pid_data → Dgain * difference;
    retrun (Pterm + Iterm + Dterm)
}
```

7.5 PID TUNING

- Analytically deriving P, I, D may not be possible
  - *For example;* plant not is not available, or too costly to obtain.
- Adhoc method for getting "reasonable" P, I, D
  - Start with small P, I = D = 0
  - Increase D, until seeing oscillation
    - Reduce D a bit
  - Increase P, until seeing oscillation
    - Reduce P a bit
  - Increase I, until seeing oscillation
- Iterate until can change anything without excessive oscillation.

7.6 PRACTICAL ISSUES WITH COMPUTER BASED CONTROL

- Quantization
- Overflow
- Aliasing
- Computation Delay

I) **Quantization**

- Can't store 0.36 as 4 bit fractional number
- Can only store 0.75, 0.59, 0.25, 0.00, –0.25, – 0.50, –1.00
- Choose 0.25
  - Result in quantization error of 0.11

Sources of Quantization error

- Operations *For example,* 0.50 * 0.25 = 0.125
  - Can use more bits until input/output to the environment/memory
- A2D converts

II) **Overflow**

- Can't store 0.75 + 0.50 = 1.25 as 4-bits fractional number

Solutions:

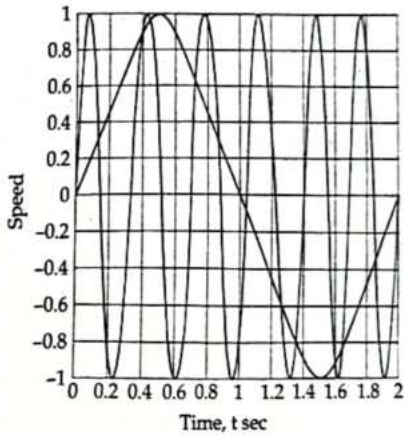
- Use fix-point representation/operations carefully.
  - Time-consuming
- Use floating-point co-processor
  - Costly

III) **Aliasing**

- Quantization/overflow
  - Due to discrete nature of computer data
- Aliasing
  - Due to discrete nature of sampling

Aliasing Example

- Sampling at 2.5 Hz, period of 0.4, the following are indistinguishable
  - $y(t) = 1.0 * \sin(6\pi t)$, frequency 3 Hz
  - $y(t) = 1.0 * \sin(\pi t)$, frequency of 0.5 Hz
- In fact, with sampling frequency of 2.5 Hz
  - Can only correctly sample signal below Nyquist frequency 2.5/2 = 1.25 Hz



IV) **Computation Delay**

- Inherent delay in processing
  - Actuation occurs later than expected
- Need to characterize implementation delay to make sure it is negligible.
- Hardware delay is usually easy to characterize
  - Synchronous design
- Software delay is harder to predict
  - Should organize code carefully so delay is predictable and minimized.
  - Write software with predictable timing behavior (be like hardware)
    - Time trigger Architecture
    - Synchronous software Language

7.7 BENEFIT OF COMPUTER CONTROL

- Cost
  - Expensive to make analog control immune to
    - Age, temperature, manufacturing error
  - Computer control replace complex analog hardware with complex code
- Programmability
  - Computer control can be "upgraded"
    - Change in control mode, gain, are easy to do
  - Computer control can be adaptive to change in plant
    - Due to age, temperature, etc.
  - Future-proof
    - Easily adapt to change in standards etc.

OLD EXAM QUESTIONS SOLUTION [TU]

1. What is PID tuning? Discuss on the practical issues related with computer based control. [2069 Bhadra]

Answer:

First part: See the topic 7.5 of chapter 7.

Second part: See the topic 7.6 of chapter 7.

2. Write the pseudo-code for a PDI controller. What is the purpose of PID tuning, and what are the benefits of computer based control implementations. [2070 Magh]

Answer:

First part: See Exam Solution of 2073 Bhadra (question number 7)

Second part: See the topic 7.5 and 7.7 of chapter 7.

3. Explain the operation of a PID control with a clean block diagram. [2070 Bhadra]

Answer: See the topic 7.3.8 of chapter 7.

4. Define the following terms used in control system controller, plant actuator. [2070 Bhadra]

Answer: See the topic 7.2 of chapter 7.

5. Differentiate between closed loop and open loop control system. With neat diagram write the steps for designing closed loop control system. [2072 Ashwin]

Answer:

First part: See the topic 7.2 of chapter 7.

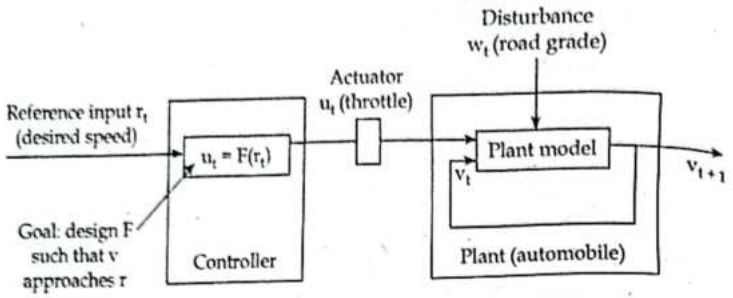
Second part: See the topic 7.2 of chapter 7.

6. Design an open loop automobile cruise controller and derive the conditions for no oscillation and reduction of road disturbance and determine the performance parameters. [2073 Magh]

Answer:

Following steps is to be carried out to design the open loop control system

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



(a)

Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_t$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$

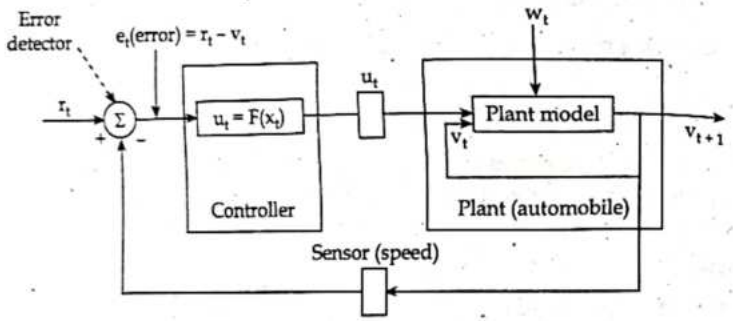


Figure: 1 Open loop automobile cruise controller block diagram

Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * (r_t - v_t)$$

System model

$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

1. Model of the plant

- May not be necessary
  - Can be done through experimenting and tuning
- But,
  - Can make it easier to design
  - May be useful for deriving the controller

- Example; throttle that goes from 0 to 45 degree
  - On flat surface at 50 mph, open the throttle to 40 degree
  - Wait 1 "time unit"
  - Measure the speed, let's say 55 mph
  - Then the following equation satisfy the above scenario

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

$$55 = 0.7 * 50 + 0.5 * 40$$

- If the equation holds for all other scenario
  - Then we have a model of the plant

2. Designing the controller

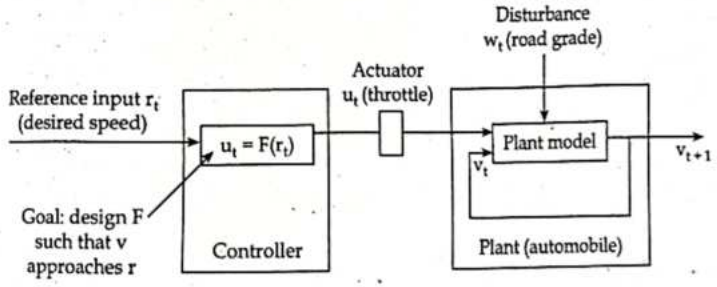
- Assuming we want to use a simple linear function

$$u_t = F(r_t) = P * r_t$$

r_t is the desired speed

- Linear proportional controller
- $v_t + 1 = 0.7 * v_t + 0.5 * u_t = 0.7 * v_t + 0.5P * r_t$
- Let $v_t + 1 = v_t$ at steady state = v_{ss}
- $v_{ss} = 0.7 * v_{ss} + 0.5P * r_t$
- At steady state, we want $v_{ss} = r_t$
- $P = 0.6$

i.e., $u_t = 0.6 * r_t$



Car model

(a)

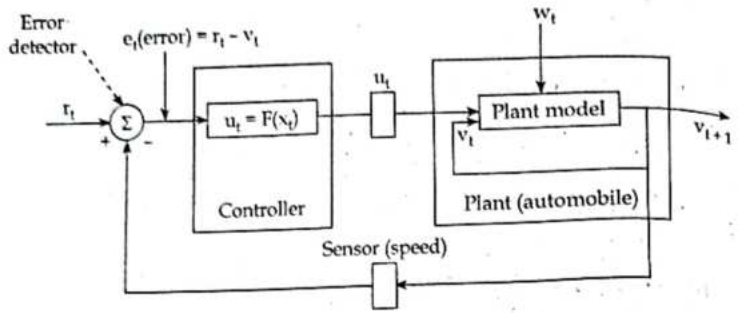
$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_t$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$



(a)

Car model

$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

$u_t = P * (r_t - v_t)$

System model

$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$

3. Analyzing the controller

- Let $v_0 = 20$ mph, $r_0 = 50$ mph
- $v_{t+1} = 0.7 * v_t + 0.5\ (0.6) * r_t = 0.7 * v_t + 0.3 * 50 = 0.7 * v_t + 15$
- Throttle position is $0.6 * 50 = 30$ degree

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

(a) (b) (c)

4. Considering the disturbance

Assume road grade can affect the speed

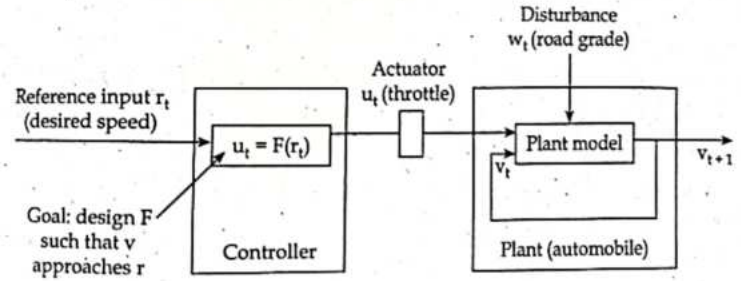
- From -5 mph to +5 mph
- $v_{t+1} = 0.7 * v_t + 10$
- $v_{t+1} = 0.7 * v_t + 20$

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

(a) (b) (c)

5. Determining Performance

- $v_{t+1} = 0.7 * v_t + 0.5P * r_0 - w_0$
- $v_1 = 0.7 * v_0 + 0.5P * r_0 - w_0$
- $v_2 = 0.7 * (0.7 * v_0 + 0.5P * r_0 - w_0) + 0.5P * r_0 - w_0 = 0.7 * 0.7 * v_0 + (0.7 + 1.0) * 0.5P * r_0 - (0.7 + 1.0)\ w_0$
- $v^t = 0.7^t * v_0 + (0.7^{t-1} + 0.7^{t-2} + \cdots\cdots + 0.7 + 1.0)\ (0.5P * r_0 - w_0)$
- Coefficient of v_t determiners rate of decay of v_0
  - > 1 or < -1, v_t will grow without bound
  - < 0, v_t will oscillate



(a)

Car model

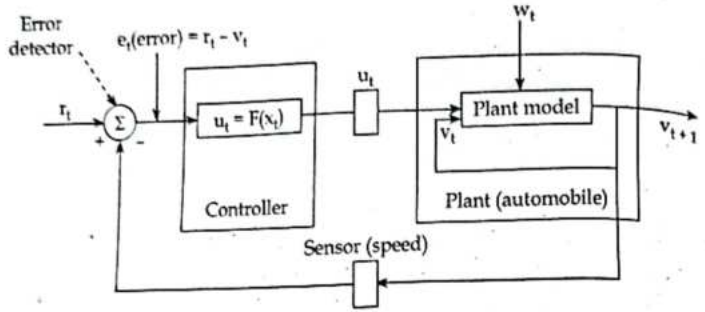
$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

$u_t = P * r_t$

System model

$v_{t+1} = 0.7v_t + 0.5Pr_t$



Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * (r_t - v_t)$$

System model

$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

7. **Explain the metrics used to measure control objectives? Write an algorithm to implement the PID controller in software.** [2073 Bhadra]

Answer:

First part: See the topic 7.3.1 of chapter 7.

Second part:

Algorithm or pseudo code to implement the PID controller in software.

- Set values for Pgain, Igain, Dgain
- Initialize prior_error = 0 and sum_of_errors = 0
- Repeat following steps
  - Read value from sensor, sensor_value = get_value_from_sensor ( )
  - Read the reference_value, refValue = get_Reference_value ( )
  - Calculate error = refValue – sensor_Value
  - Calculate sum_of_errors = Sum_of_errors + error
  - Calculate difference = prior_error – error
  - Output = Pgain * error + Igain * sum_of_errors + Dgain * difference.
  - Set the output of actuator, set Actuator (output).
  - Prior_error = error

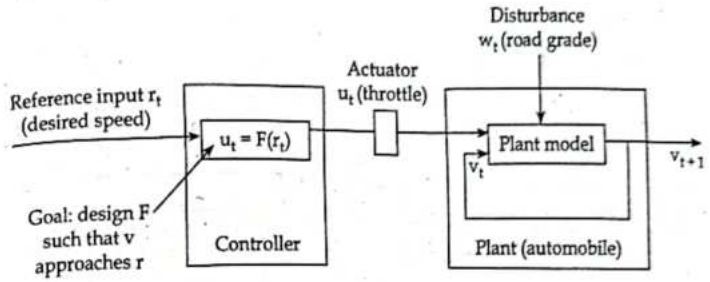
8. **What are the challenges of modeling a real physical system and how can you overcome it? Write an algorithm to implement the PID controller in software.** [2074 Bhadra, 2077 Poush]

Answer: Same as question number 7 [2073 Bhadra TU].

9. **Draw the block diagram of closed-loop control system for speed control of an automobile and explain the conditions for no unbound and no oscillation showing all the design steps.** [2075 Baisakh]

Answer:

Designing Close Loop Control System



(a)

Car model

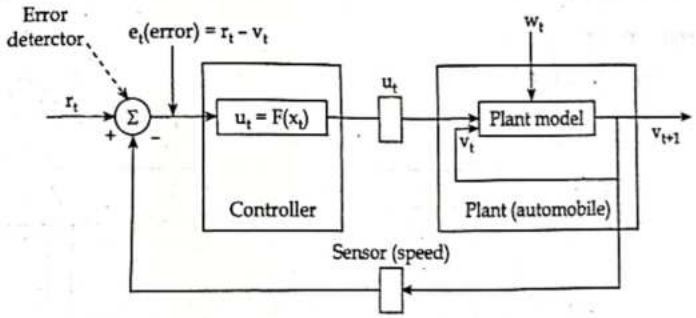
$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_1$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$



(a)

Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * (r_t - v_t)$$

System model

$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

1. **Stability**

$$u_t = P * (r_t - v_t)$$

$$v_{t+1} = 0.7\,v_t + 0.5\,u_t - w_t$$

$$= 0.7\,v_t + 0.5P * (r_t - v_t) - w$$

$$= (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$$

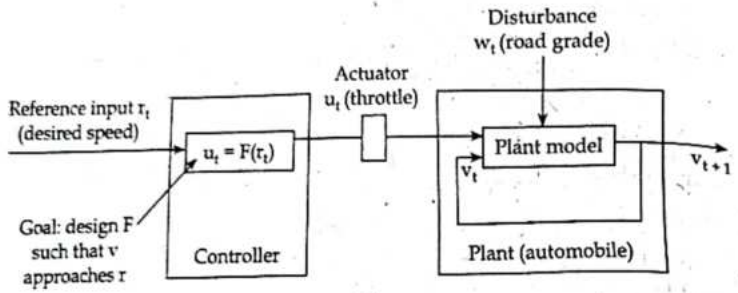
$v_t = (0.7 - 0.5P)\ t * v_0\ ((0.7 - 0.5P)t - 1 + (0.7 - 0.5\ P)t - 2 + \ldots\ldots +$
$0.7 - 0.5P + 1.0)\ (0.5P * r_0 - w_0)$

Stability constraint (*i.e.*, convergence) requires

$|0.7 - 0.5P| < 1$

$-1 < 0.7 - 0.5\ P < 1$

$-0.6 < P < 3.4$



(a)

Car model

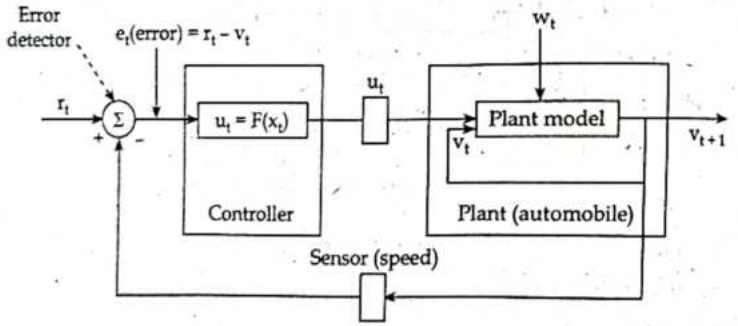
$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

$u_t = P * r_1$

System model

$v_{t+1} = 0.7v_t + 0.5Pr_t$



Car model

$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

$u_t = P * (r_t - v_t)$

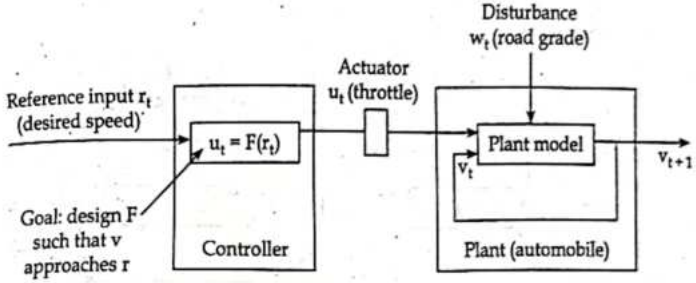
System model

$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$

2. Reducing effect of v_0

- $u_t = P * (r_1 - v_t)$

- $v_{t+1} = 0.7v_t + 0.5\ u_t - w_t$
  $= 0.7\ v_t + 0.5P * (r_t - v_t) - w$
  $= (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$
- $v_t = (0.7 - 0.5P)_t * v_0 + ((0.7 - 0.5P)_{t-1} + (0.7 - 0.5P)_{t-2} + \ldots\ldots + 0.7 - 0.5P + 1.0)\ (0.5P * r_0 - w_0)$
- To reduce the effect of initial condition
  - $0.7 - 0.5P$ as small as possible
  - $P = 1.4$



(a)

Car model

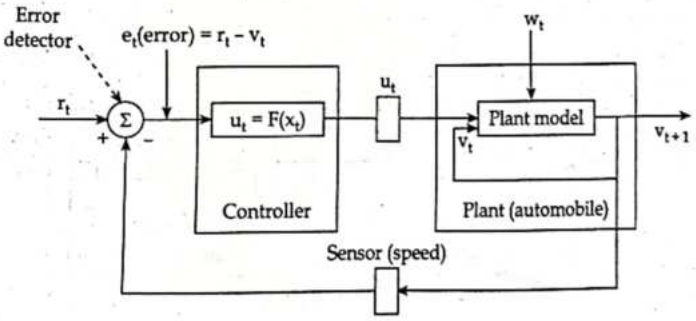
$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

$u_t = P * r_1$

System model

$v_{t+1} = 0.7v_t + 0.5Pr_t$



Car model

$v_{t+1} = 0.7v_t + 0.5u_t - w_t$

Control law

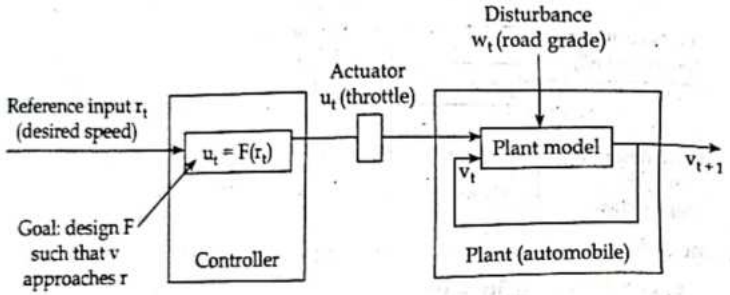
$u_t = P * (r_t - v_t)$

System model

$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$

3. Avoid Oscillation

- $u_t = P * (r_t - v_t)$
- $v_{t+1} = 0.7v_t + 0.5\,u_t - w_t$
  $= 0.7\,v_t + 0.5P * (r_t - v_t) - w$
  $= (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$
- $v^t = (0.7 - 0.5P)_t * v_0 + ((0.7 - 0.5P)_{t-1} + (0.7 - 0.5P)_{t-2} + \cdots\cdots + 0.7 - 0.5P + 1.0)\,(0.5P * r_0 - w_0)$
- To avoid oscillation
  - $0.7 - 0.5\,P >= 0$
  - $P <= 1.4$



(a)

Car model

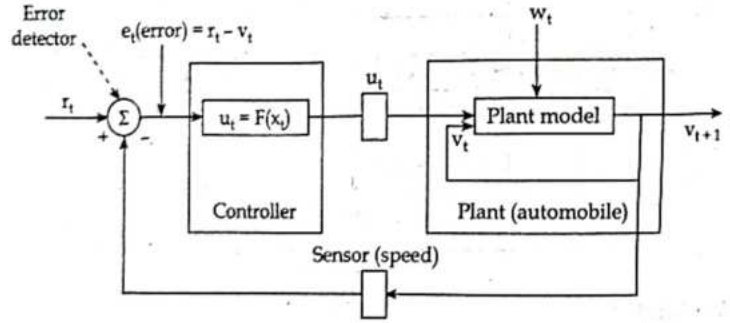
$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_t$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$



Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

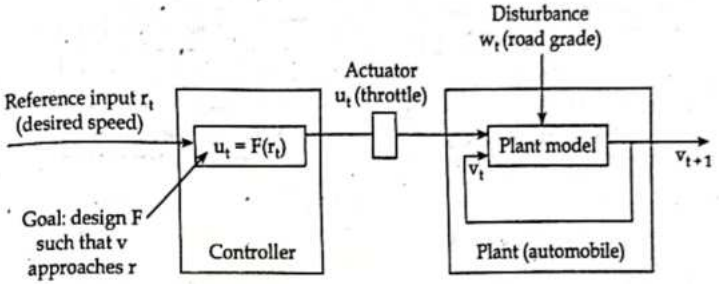
$$u_t = P * (r_t - v_t)$$

System model

$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

4. Perfect Tracking

- $u_t = P * (r_1 - v_t)$
- $v_{t+1} = 0.7v_t + 0.5\,u_t - w_t = 0.7\,v_t + 0.5P * (r_t - v_t) - w$
  $= (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$
- $v_{ss} = (0.7 - 0.5P)^t * v_{ss} + 0.5P * r_0 - w_0\,(1 - 0.7 + 0.5P)\,v_{ss} = 0.5P * r_0 - w_0$
  $v_{ss} = (0.5P/(0.3 + 0.5P)) * r_0 - (1.0\,(0.3 + 0.5P)) * w_0$
- To make v_{ss} as close to r_0 as possible
  - P should be as large as possible



(a)

Car model

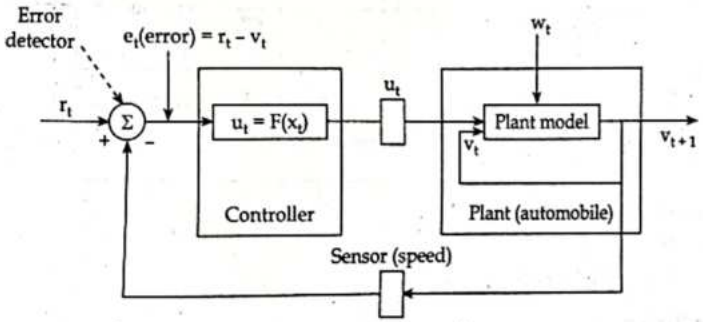
$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_t$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$



Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

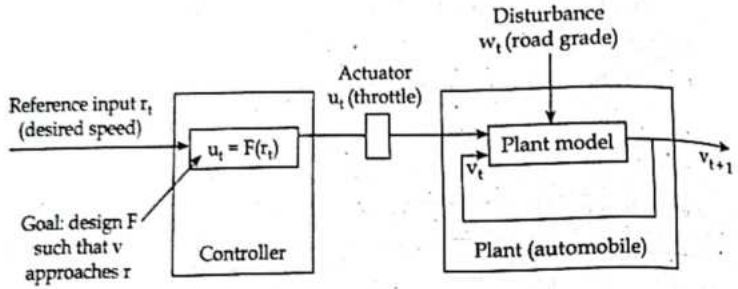
$$u_t = P * (r_t - v_t)$$

System model

$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

5. Close-Loop Design

- $u_1 = P * (r_t - v_t)$
- Finally, setting P = 3.3
  - Stable, track well, some oscillation
  - $u_t = 3.3 * (r_t - v_t)$



(a)

Car model

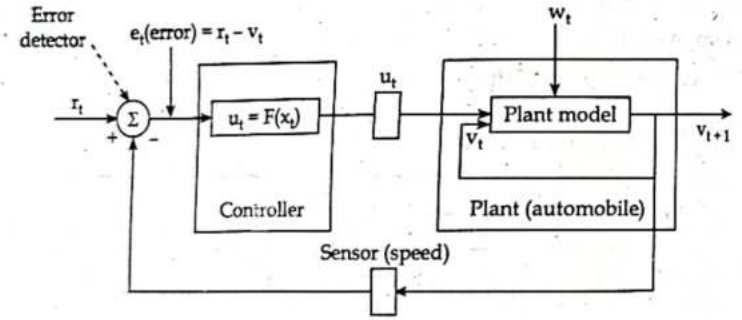
$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * r_1$$

System model

$$v_{t+1} = 0.7v_t + 0.5Pr_t$$



Car model

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Control law

$$u_t = P * (r_t - v_t)$$

System model

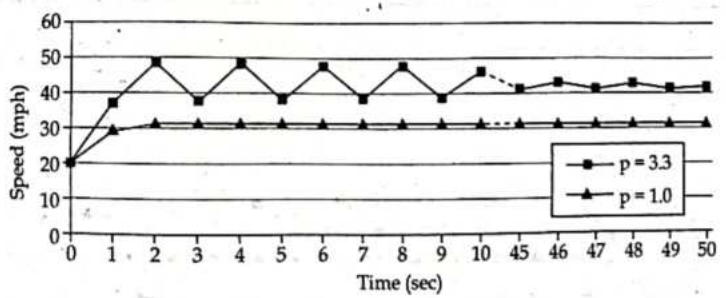
$$v_{t+1} = (0.7 - 0.5P)v_t + 0.5Pr_t - w_t$$

6. Analyze the controller

- $v_0 = 20$ mph, $r_0 = 50$ mph, $w = 0$
- $v_{t+1} = 0.7\ v_t + 0.5P * (r_t - v_t) - w = 0.7\ v_t + 0.5 * 3.3 * (50 - v_t)$

- $u_t = P * (r_t - v_t) = 3.3 * (50 - v_t)$
- But u_t range from 0 - 45
- Controller saturates

Time	v_t	u_t	v_t	u	v_t	u_t
0	20.00	99.00	20.00	45.00	20.00	30.00
1	63.50	-44.55	36.50	44.55	29.00	21.00
2	22.18	91.82	47.83	7.18	30.80	19.20
3	61.43	-37.73	37.07	42.68	31.16	18.84
4	24.14	85.34	47.29	8.95	31.23	18.77
5	59.57	-31.58	37.58	40.99	31.25	18.75
6	25.91	79.50	46.80	10.55	31.25	18.75
7	57.89	-26.02	38.04	39.47	31.25	18.75
8	27.51	74.22	46.36	12.00	31.25	18.75
9	56.37	-21.01	38.45	38.10	31.25	18.75
10	28.95	69.46	45.97	13.31	31.25	18.75
...						
45	44.53	18.06	41.70	27.39	31.25	18.75
46	40.20	32.34	42.89	23.48	31.25	18.75
47	44.31	18.78	41.76	27.20	31.25	18.75
48	40.41	31.66	42.83	23.66	31.25	18.75
49	44.11	19.42	41.81	27.02	31.25	18.75
50	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)	



(d)

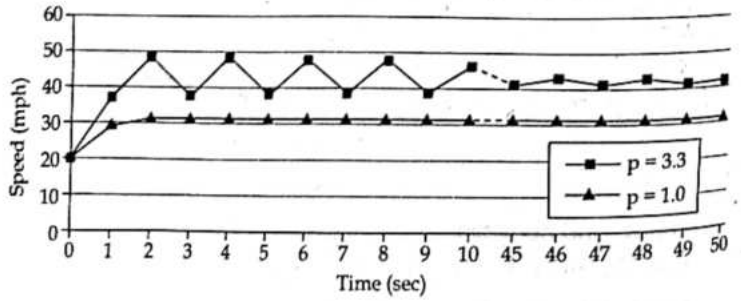
7. Analyze the controller

- $v_0 = 20$ mph, $r_0 = 50$ mph, $w = 0$
- $v_{t+1} = 0.7\ v_t + 0.5 * u_t$

- $u_t = 3.3 * (50 - v_t)$
  - Saturate at 0,45
- Oscillation!
  - "feel bad"

Time	v_t	u_t	v_t	u	v_t	u_t
0	20.00	99.00	20.00	45.00	20.00	30.00
1	63.50	-44.55	36.50	44.55	29.00	21.00
2	22.18	91.82	47.83	7.18	30.80	19.20
3	61.43	-37.73	37.07	42.68	31.16	18.84
4	24.14	85.34	47.29	8.95	31.23	18.77
5	59.57	-31.58	37.58	40.99	31.25	18.75
6	25.91	79.50	46.80	10.55	31.25	18.75
7	57.89	-26.02	38.04	39.47	31.25	18.75
8	27.51	74.22	46.36	12.00	31.25	18.75
9	56.37	-21.01	38.45	38.10	31.25	18.75
10	28.95	69.46	45.97	13.31	31.25	18.75
...						
45	44.53	18.06	41.70	27.39	31.25	18.75
46	40.20	32.34	42.89	23.48	31.25	18.75
47	44.31	18.78	41.76	27.20	31.25	18.75
48	40.41	31.66	42.83	23.66	31.25	18.75
49	44.11	19.42	41.81	27.02	31.25	18.75
50	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)

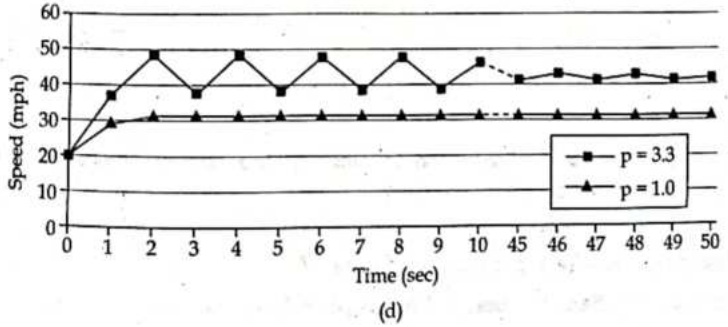


(d)

8. **Analyze the controller**
   - Set P = 1.0 to void oscillation
     - Terrible SS performance

Time	v_t	u_t	v_t	u	v_t	u_t
0	20.00	99.00	20.00	45.00	20.00	30.00
1	63.50	-44.55	36.50	44.55	29.00	21.00
2	22.18	91.82	47.83	7.18	30.80	19.20
3	61.43	-37.73	37.07	42.68	31.16	18.84
4	24.14	85.34	47.29	8.95	31.23	18.77
5	59.57	-31.58	37.58	40.99	31.25	18.75
6	25.91	79.50	46.80	10.55	31.25	18.75
7	57.89	-26.02	38.04	39.47	31.25	18.75
8	27.51	74.22	46.36	12.00	31.25	18.75
9	56.37	-21.01	38.45	38.10	31.25	18.75
10	28.95	69.46	45.97	13.31	31.25	18.75
...						
45	44.53	18.06	41.70	27.39	31.25	18.75
46	40.20	32.34	42.89	23.48	31.25	18.75
47	44.31	18.78	41.76	27.20	31.25	18.75
48	40.41	31.66	42.83	23.66	31.25	18.75
49	44.11	19.42	41.81	27.02	31.25	18.75
50	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

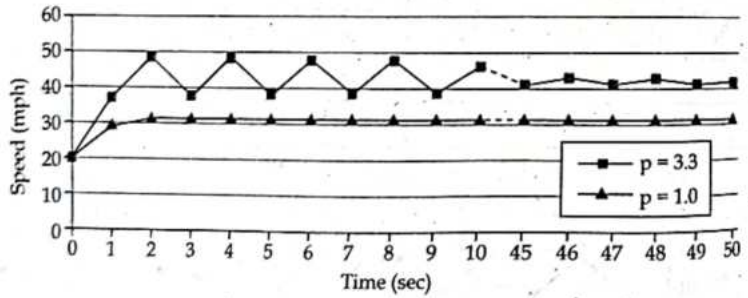
(a) (b) (c)



(d)

9. Analyzing the controller

Time	v_t	u_t	v_t	u_t	v_t	u_t
0	20.00	99.00	20.00	45.00	20.00	30.00
1	63.50	-44.55	36.50	44.55	29.00	21.00
2	22.18	91.82	47.83	7.18	30.80	19.20
3	61.43	-37.73	37.07	42.68	31.16	18.84
4	24.14	85.34	47.29	8.95	31.23	18.77
5	59.57	-31.58	37.58	40.99	31.25	18.75
6	25.91	79.50	46.80	10.55	31.25	18.75
7	57.89	-26.02	38.04	39.47	31.25	18.75
8	27.51	74.22	46.36	12.00	31.25	18.75
9	56.37	-21.01	38.45	38.10	31.25	18.75
10	28.95	69.46	45.97	13.31	31.25	18.75
...						
45	44.53	18.06	41.70	27.39	31.25	18.75
46	40.20	32.34	42.89	23.48	31.25	18.75
47	44.31	18.78	41.76	27.20	31.25	18.75
48	40.41	31.66	42.83	23.66	31.25	18.75
49	44.11	19.42	41.81	27.02	31.25	18.75
50	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)	



(d)

10. Explain the matrices used to measure the control objectives. Explain software coding of PID controller. [2076 Bhadra]

Answer:

First part: See the topic 7.3.1 of chapter 7.

Second part: See the topic 7.4 of chapter 7.

Chapter 8

IC TECHNOLOGY

★★★

★★★

8.1 INTRODUCTION

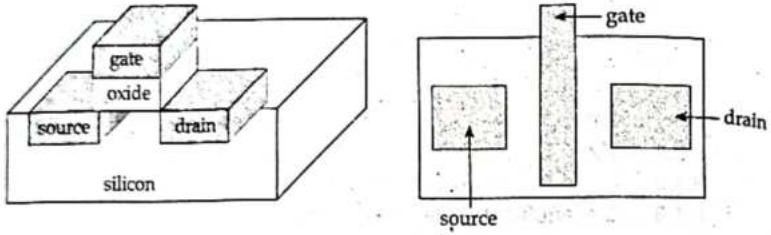
The growth of electronics started with invention of vacuum tubes and associated electronic circuits. This activity termed as vacuum tubes electronics, subsequently the evolution of solid state devices and consequent development of integrated circuits are responsible for the present status of communication, computing and instrumentation. Due to its small dimension, low cost, and very high reliability even the common man is familiar with its applications like smart phones and laptops. The IC's also found its way in military applications, state of the art communication systems, and industrial applications due to its high reliability and compact size. 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.

8.1.1 CMOS Transistor

CMOS transistor consists of three terminal: the source, drain, and gate.

- Source, Drain
  - Diffusion area where electrons can flow
  - Can be connected to metal contacts
- Gate
  - Polysilicon area where control voltage is applied.



8.1.2 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 exist number of oxide layer.

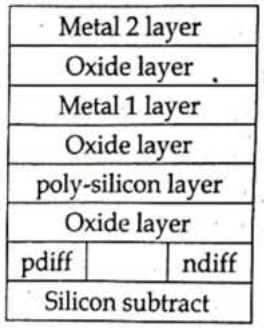
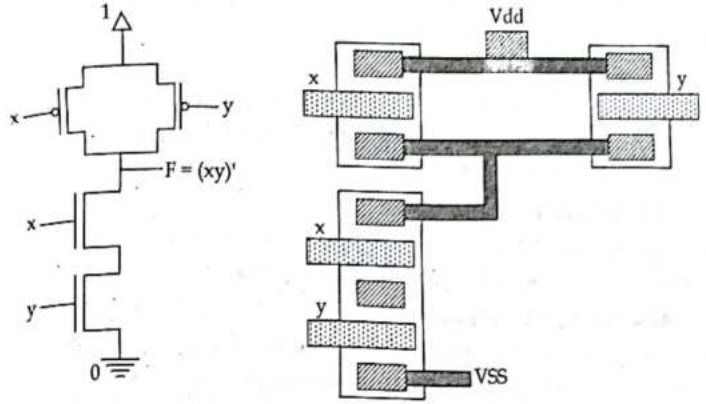


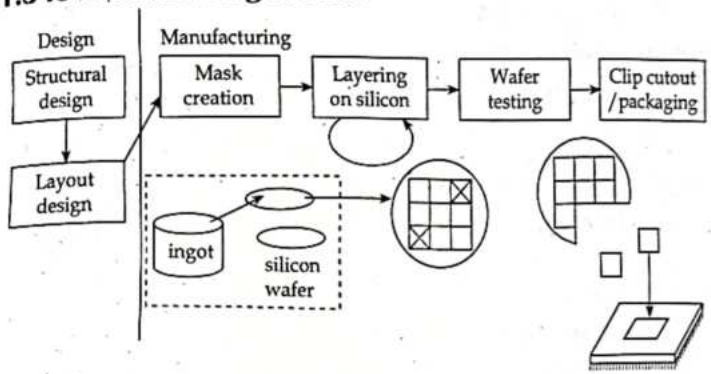
Figure: Basic layers in physical Implementation

NAND gate

- Metal layers for routing (~ 10)
- PMOS don't like 0
- NMOS don't like 1
- A stick diagram from the basis for mask sets.



8.1.3 IC Manufacturing Process



IC manufacturing as a process entails quite a number of specific steps.

1. Wafer production:

Wafer is a round slice of semiconductor material such as silicon. In IC, silicon wafer is used.

- Purified polycrystalline Si is created from sand.
- It is heated to produce molten liquid.
- A small piece of solid Si is dipped into molten liquid
- Solid silicon is slowly pulled from the melt
- The liquid cools to form single crystal.
- A thin round wafer is cut using wafer slicer.
- The wafer surface is smoothened by polishing.
- It is cleaned and dried using purity low particle chemicals.

2. Photolithography

Photolithography is the process of printing a pattern of mask into the silicon wafer. It is carried out using light sensitive photo resist and controlled exposure to light.

Positive photolithography prints a pattern that is same as the pattern on the mask.

Negative photolithography prints a pattern that is opposite of the pattern as that on the mask.

3. Doping

To alter electrical character of silicon, atom with one less electron and atom with one more electron are introduced into area. The impurity atoms in semiconductor material are moved at high temperature.

4. Metallization

It is used to create contact with Si and to make interconnections on chip. A thin layer of aluminium is deposited over the wafer.

5. Assembly and packaging

Each wafer consists of many chips. These chips are separated and packaged by scribing and cleaving. A diamond saw is used to cut wafer into chips. The tested and verified chip is mounted into a package. It is encapsulated for protection.

Wafer condition before

i) Surface includes film composition, bare surface and reflectivity.

ii) It affects:
   a) Photo resist-to-water adhesion.
   b) Alignment accuracy
   c) Line width resolution
   d) Exposure settings
   e) Bake time

Wafer condition after

a) Resist coated wafer
b) Patterned resist layer
c) Withstand etching
d) Withstand ion implanting

Steps of Photo Lithography

i) Surface Preparation

- It increases adhesion of photo resist material to the substrate.
- Dehydration bake removes wafer from substrate by baking at temperature of 200°C to 400°C from 30 to 60 minutes.
- The substrate is then cooled and coated.
- Adhesion promoters like Hexamethyl disilizane (HMDS) are used to reach chemically with surface and replace -OH with organic function group.

ii) Photo resist Coating

- A thin uniform coating of photo resist is accomplished by spin coating.
- The photo resist in liquid form is poured onto the wafer and then spun at a high speed to produce desired film.
- Thickness is affected by spin speed, time and volume of resist.

iii) Soft Bake

- The film contain about 20-40% by weight solvent
- It involves drying or removing of excess solvent
- It helps to stabilize the resist film.
- It improves adhesion and uniformity
- It is baked in oven at 95 °C for 30 minutes.

iv) Alignment and Exposure

- A photo mask with pattern on one side is aligned over the wafer.
- A UV light (12 mw) is exposed to the surface
- It transfers the mask image to the resist coated wafer.
- The exposure time affects critical dimension. With increase in exposure time, CD increase

v) Develop

- Soluble areas of photo resist are dissolved by developer chemical.
- A visible pattern appears on wafer.

vi) Hard Bake

- It hardens the final resist image.
- Hard bake is done at temperature of 110°C for about 30 minutes.
- It improves adhesion.

vii) Develop Inspection

- The resist patterned wafer is inspected for particles, defects, CD(critical dimension), line width resolution and overlay accuracy.

viii) Etching

- It is the selective removal of upper layer of wafer
- It is performed either by using wet chemicals (acids) or in a dry plasma environment.

ix) Strip

- It is the process of removing photo resist.
- Wet acid strip or dry plasma strip.

x) Final Inspection

- Complete removal of photo resist
- Pattern on wafer to be correct
- Defects, particles, step height and CD are checked.

8.2 FULL CUSTOM (VLSI) IC TECHNOLOGY

- Very Large Scale Integration (VLSI)
- Placement
  - Place and orient transistors
- Routing
  - Connect transistors

- Sizing
  - Make fat, fast wires or thin, slow wires
  - May also need to size buffer.
- Design rules
  - "Simple" rules for correct circuit function
    - Metal/metal spacing, min poly width
  - Best size, power, performance
  - Hand design
    - Horrible time-to-market/flexibility/NRE cost
    - Reserve for the most important units in a processor
      - ALU, Instruction fetch
- Physical design tools
  - Less optimal, but faster

8.3 SEMI-CUSTOM (ASIC) IC TECHNOLOGY

- Gate Array
  - Array of prefabricated gates
  - "Place" and route
  - Higher density, faster time-to-market
  - Does not integrate as well with full-custom
- Standard cell
  - A library of pre-designed cell
  - Place and route
  - Lower density, higher complexity
  - Integrate great with full-custom
- Most popular design style
- Jack of all trade
  - Good
    - Power, time-to-market, performance, NRE cost, per unit cost, area
- Master of none
  - Integrate with full customs for critical regions of design

8.4 PROGRAMMABLE LOGIC DEVICE (PLD) IC TECHNOLOGY

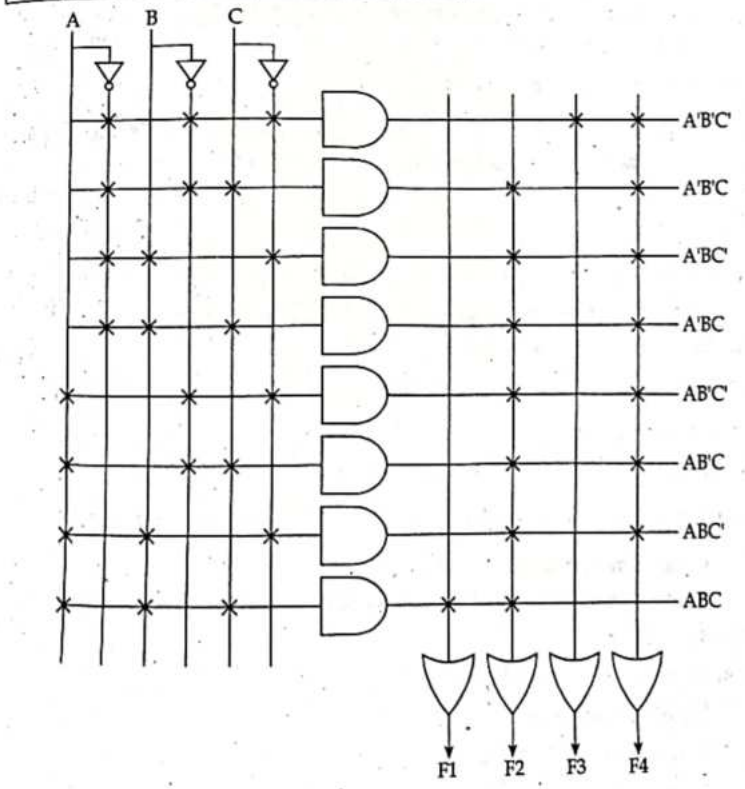
- Programmable Logic Device
  - Programmable Logic Array, programmable Array Logic, field programmable gate array
- All layers already exist
  - Designers can purchase an IC
  - To implement desired functionality
    - Connections on the IC are either created or destroyed to implement

- Benefits
  - Very low NRE costs
  - Great time to market
- Drawback
  - High unit cost, bad for large volume
  - Power
    - Expect special PLA
  - Slower

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'



OLD EXAM QUESTION SOLUTION (TU)

1. Define the photolithography. Explain the various steps involved in photolithography. [2069 Bhadra]

Answer:

First part: See the topic 8.1.3 (2) of chapter 8.

Second part: See the topic 8.1.3 (steps of photolithography) of chapter 8.

2. Explain the IC manufacturing steps with a neat block diagram. [2070 Magh, 2077 Poush]

Answer:

First part: See the topic 8.1.3 of chapter 8.

3. What are IC manufacturing steps in semiconductor industries? Describe the alignment and exposure techniques in photolithographic process in IC fabrication. [2071 Magh]

Answer:

First part: See the topic 8.1.3 of chapter 8.

Second part: See the topic 8.1.3 (steps of photolithography) of chapter 8.

4. Why photolithography process role is important in IC technology? Explain with suitable example. [2071 Bhadra]

Answer: See the topic 8.1.3 of chapter 8.

5. What is photolithography and what are its types? Describe the steps that need to be performed before photolithography process. [2072 Magh, 2078 Baisakh]

Answer:

Photolithography is the process of printing a pattern of mask into the silicon water. It is carried out using light sensitive photo resist and controlled exposure to light. It is of two types.

They are:

i) Positive photo lithography

ii) Negative photolithography

Second part: See the topic 8.1.3 (water production) of chapter 8.

6. Discuss the advantages and disadvantages of full-custom IC technology. Explain the basic steps of photolithograph process. [2072 Ashwin]

Answer:

Advantage of full custom IC technology

- Excellent performance
- Small size
- Low power

Disadvantages of full custom IC technology

- High NRE cost
- Long time-to-market

Second part: See the topic 8.1.3 (steps of photolithography) of chapter 8.

7. Show various steps of photolithography process using appropriate diagrams. Describe briefly about full custom VLSI technology. [2075 Baisakh]

Answer:

First part: See the topic 8.1.3 of chapter 8.

Second part: See the topic 8.2 of chapter 8.

8. Discuss the advantages and disadvantages of full-custom IC technology. Explain the basic steps of photolithography process. [2076 Bhadra]

Answer:

First part: Same as question number 8.

Second part: See the topic 8.1.3 of chapter 8.

9. List the three major IC technologies with brief definitions. [2070 Magh, 2077 Poush]

Answer: See the topic 8.2, 8.3 and 8.4 of chapter 8.

10. Describe the steps involved in manufacturing an IC. Show the top-down view of the circuit F = xz + y on an IC. [2070 Bhadra, 2077 Baisakh]

Answer:

First part: See the topic 8.1.3 of chapter 8.

Second part:

The way of solving this question is by representing the given function using NAND, NOR and NOT gates. So the given function can be written as:

$$F = xz + y$$

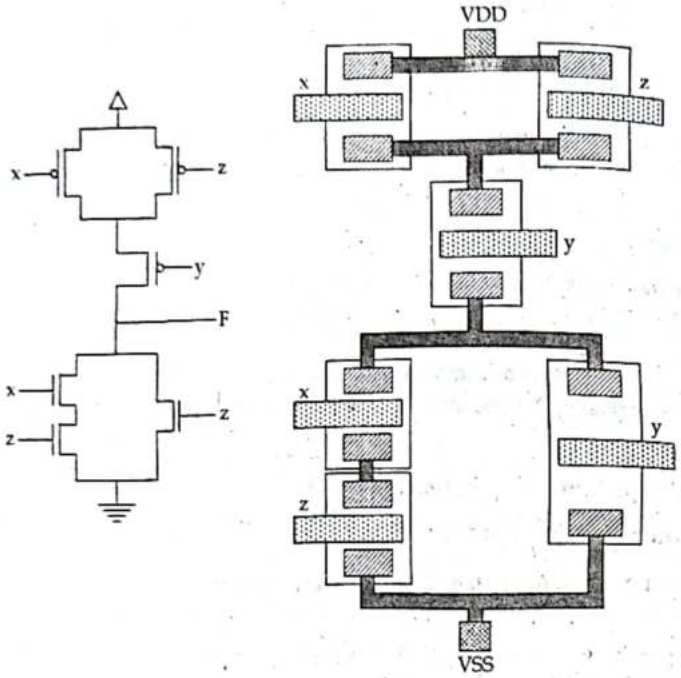
$$F = (\overline{\overline{x \cdot z + y}})$$

$$F = \overline{\overline{x \cdot z} + \overline{y}}$$

$$F = (x \text{ NAND } z) \text{ NAND } (\text{NOT } y)$$

So we need to use top level view of two NAND gate and one NOT gate.

The top level view of the function F = xz + y is shown below:



Chapter 9

MICROCONTROLLERS IN EMBEDDED SYSTEMS

★★★

★★★

9.1 INTEL 8051 MICROCONTROLLER FAMILY, ITS ARCHITECTURE AND INSTRUCTION SETS

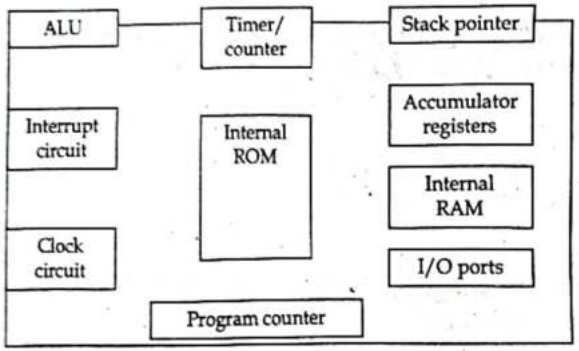
9.1.1 Introduction

Microcontroller is a compact tiny computer that is fabricated inside a chip and is used in automatic control systems including security systems, office machines, power tools, alarming system, traffic light control,

washing machine, and much more. It was specially built for embedded system and consisted of read write memory, read only memory, I/O ports, processor and built in clock.

9.1.2 Block System

The general block diagram of the microcontroller is shown in the figure below:



9.1.3 Comparison with Microprocessor

S.N.	Microprocessor	Microcontrollers
1.	Microprocessor contains ALU, General purpose register, stack pointer, program counter, clock timing circuit, interrupt circuit.	Microcontroller contains the circuitry of microprocessor, and in addition it has built in ROM, RAM I/O devices, timers counter's etc
2.	Few bit handling instruction.	It has many bit handling instructions.
3.	Less number of pins are multifunctional.	More number of pins are multifunctional.
4.	Single memory map for data and code (program).	4. Separate memory map for data and code
5.	Access time for memory and I/O are more.	Less access time for built in memory and I/O.
6.	Microprocessor based system requires additional hardware.	It requires less additional hardware.
7.	More flexible in the design point of view.	Less flexible since the additional circuits which is residing inside the microcontroller is fixed for a particular microcontroller.
8.	Large number of instructions with flexible addressing modes.	Limited number of instructions with few addressing modes.

9.1.4 Criteria for Choosing a Microcontroller

i) Microcontroller should meet the computing requirement like speed of processing, ability of handling multiple tasks efficiently and effectively.

ii) Cost also plays an important role to choose microcontroller. Because cost of per unit of each microcontroller also plays an important role in cost of product.

iii) Software availability should be considered while choosing microcontroller. Software tools like compilers, assemblers and simulators should be easily available.

iv) Help resources should be available from manufacturer.

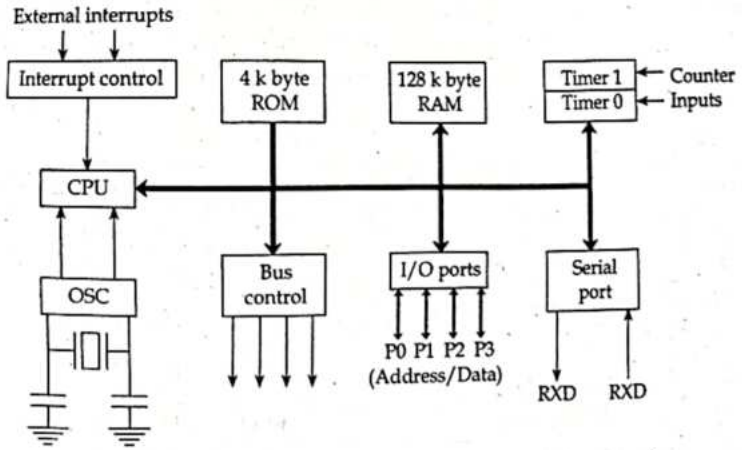
v) Online source code libraries and help formus should also available.

9.1.5 Comparison of 8051 Family Members

The 8051 is the original member of the 8051 family. There are two other members in the 8051 family of microcontrollers. They are 8052 and 8031. The following table shows comparison of 8051, 8052 and 8031 micro-controllers.

Features	8051	8052	8031
ROM(on-chip program space in bytes)	4 K	8 K	0 K
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

9.1.6 8051 Architecture



9.1.6.1 Features of 8051 Architecture

Salient features of 8051 microcontroller are given below.

- Eight bit CPU
- On chip clock oscillator

- 4 kbytes of internal program memory (code memory) [ROM]
- 128 bytes of internal data memory [RAM]
- 64 kbytes of external program memory address space.
- 64 kbytes of external data memory address space.
- 32 bidirectional I/O lines (can be used as four 8 bit ports or 32 individually addressable I/O lines)
- Two 16 bit timer/counter: T0, T1
- Full duplex serial data receiver/transmitter.
- Four register banks with 8 registers in each bank.
- Sixteen bit program counter (PC) and a data pointer (DPTR).
- 8 bit program status word (PSW).
- 8 bit stack pointer.
- Five vector interrupt structure (RESET not considered as an interrupt).
- 8051 CPU consists of 8 bit ALU with associated registers like accumulator 'A', B register, PSW, SP, 16 bit program counter, stack pointer.
- ALU can perform arithmetic and logic functions on 8 bit variables.
- 8051 has 128 bytes of internal RAM which is divided into
  - Working registers [00-1F]
  - Bit addressable memory area [20-2F]
  - General purpose memory area (scratch pad memory) [30-7F]

9.1.6.2 8051 Special Function Registers (SFRs)

The 8051 microcontroller special function register act as a control table that monitor and control the operation of the 8051 microcontroller. All the 21 8051 microcontroller special function registers (SFRs) along with their functions Internal RAM Address and Access Level is given in the following table:

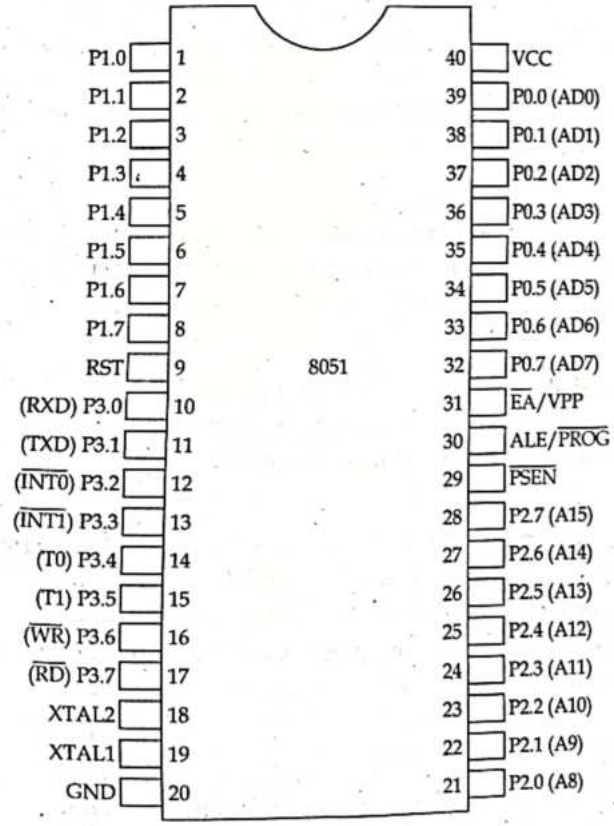
Name	Function	Ram Address	Access Level
ACC	Accumulator	E0H	Bit Addressable
B	B register (for Arithmetic)	F0H	Bit addressable
DPH	Addressing external Memory	83 H	
DPL	Addressing External memory	82 H	
IE	Interrupt Enable Control	A8H	Bit Addressable
IP	Interrupt Priority	B8H	Bit Addressable
P 0	PORT 0 Latch	8 0H	Bit Addressable
P 1	PORT 1 Latch	9 0H	Bit Addressable
P 2	PORT 2 latch	A0H	Bit Addressable
P 3	PORT 3 Latch	B0H	Bit Addressable

PCON	Power control	87 H	Bit Addressable
PSW	Program status Word	DOH	Bit Addressable
SCON	Serial port control	98 H	Bit Addressable
SBUF	Serial port data buffer	99 H	
SP	Stack pointer	81 H	
TMOD	Timer/Counter Mode Control	89 H	
TCON	Timer/Counter Control	88 H	Bit Addressable
TLO	Timer 0 Low byte	8 AH	
THO	Timer 0 HIGH Byte	8 CH	
TL 1	Timer 1 Low Byte	8 BH	
TH 1	Timer 1 HIGH Byte	8 DH	

9.1.7 Pin Descriptions

PINS 1-8

Recognized as port 1 Different from other ports, this port doesn't provide any other purpose. Port 1 is a domestically pulled up, quasi bidirectional input/output port.



Pin 9

As made clear previously RESET pin is utilized to set the microcontroller 8051 to its primary values, where as the microcontroller is functioning or at the early beginning of the application. The RESET pin has to be set elevated for two machine rotations.

Pin 10-17

Recognized as port 3. This port also supplies several other functions such as timer input, interrupts, serial communication indicators TXD and RXD, control indicators for outside memory intefacing WR and RD etc. This is a domestic pull up the port with quasi bidirectional port within.

Pin 18 and 19

These are employed for interfacing an outer crystal to give system clock.

Pin 20:

Titled as VSS-it symbolizes ground (0 V) association.

Pins 21-28

Recognized as port 2 (P 2.0 – P 2.7)–other than serving as input/output port, senior order address bus indicators are multiplexed with this quasi bidirectional port.

Pin 29

Program store Enable or PSEN is employed to interpret signs from outer program memory.

Pin 30

External Access or EA input is employed to permit or prohibit outer memory interfacing. IF there is no outer memory need, this pin is dragged high by linking it to VCC.

Pin 31

Aka Address Latch Enable or ALE is brought into play to de multiplex the address data indication of port 0(for outer memory interfacing). Two ALE throbs are obtainable for every machine rotation.

Pins 32-39

Recognized as port 0 (P 0.0 to P 0.7) other than serving as input/output port, low order data and address but signals are multiplexed with this port (to provide the use of outer memory interfacing). This pin is a bidirectional input/output port (the signal one in microcontroller 8051) and outer pull up resistors are necessary to utilize this port as input/output.

Pin 40

Termed as VCC is the chief power supply. By and large, it is +5 V DC.

9.1.8 Minimum Hardware Configuration

Power Supply

Pin 40 is connected to + 5 VDC, 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.

Oscillator circuit

Ceramic capacitors of value between 20 μF – 40 μF 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 10 K ohm pull-up resistor.

9.1.9 8051 Instruction Sets

The 8051 instruction sets are divided into five functional groups:

1. Arithmetic instructions
2. Logical instructions
3. Data transfer instructions
4. Boolean variable instructions
5. Program branching instructions

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.

1. **Arithmetic Instructions**

Mnemonics	Operation
ADD A, Rn	A ← A + Rn
ADD A, direct	A ← A + [direct]
ADD A, @ Ri	A ← A + [Ri]
ADD A, # data	A ← A + data
ADDC A, Rn	A ← A + Rn + C

ADDC A, direct	A ← A + [direct] + C
ADDC A, @ Ri	A ← A + [Ri] + C
ADDC A, # data	A ← A + data + C
SUBB A, Rn	A ← A – Rn – C
SUBB A, direct	A ← A – [direct] – C
SUBB A, @ Ri	A ← A – [Ri] – C
SUBB A, # data	A ← A – data – C
INC A	A ← A + 1
INC Rn	Rn ← Rn + 1
INC direct	[direct] ← [direct] + 1
INC @ Ri	[Ri] ← [Ri] + 1
DEC A	A ← A – 1
DEC Rn	RN ← Rn – 1
DEC direct	[direct] ← [direct] – 1
DEC @ Ri	[Ri] ← [Ri] – 1
INC DPTR	DPTR ← DPTR + 1
MULAB	A ← Lower Byte
	B ← Higher Byte
DIV AB	A ← Quotient
	B ← Remainder
DA A	Decimal adjust accumulator

2. **Logical Instructions**

Mnemonics	Operation
ANL A, Rn	A ← A AND Rn
ANL A, direct	A ← A AND [direct]
ANL A, @ Ri	A ← A AND [Ri]
ANL A, # data	A ← A AND data
ANL direct, A	[direct] ← [direct] AND A
ANL direct, # data	[direct] ← [direct] AND data
ORL A, Rn	A ← A OR Rn
ORL A, direct	A ← A OR [direct]
ORL A, @ Ri	A ← A OR [Ri]
ORL A, # data	A ← A OR data
ORL direct, A	[direct] ← [direct] OR A
ORL direct, # data	[direct] ← [direct] OR data
XRL A, Rn	A ← A XOR Rn
XRL A, direct	A ← A XOR [direct]
XRL A, @ Ri	A ← A XOR [Ri]
XRL A, # data	A ← A XOR data

XRL direct, A	[direct] ← [direct] XOR A
XRL direct, # data	[direct] ← [direct] XOR data
CRL A	A ← 0
CPL A	A ← A'
RL A	Rotate A left
RLC A	Rotate A left through C
RR A	Rotate A right
RRC A	Rotate A right through C
SWAP A	Swap nibbles of A

3. **Data Transfer Instructions**

Mnemonics	Operation
MOV A, Rn	A ← Rn
MOV A, direct	A ← [direct]
MOV A, @ Ri	A ← [Ri]
MOV A, data	A ← data
MOV Rn, A	Rn ← A
MOV Rn, direct	Rn ← [direct]
MOV Rn, # data	Rn ← data
MOV direct, A	[direct] ← A
MOV direct, Rn	[direct] ← Rn
MOV direct, direct	[direct] ← [direct]
MOV direct, @ Ri	[direct] ← Ri
MOV direct, # data	[direct] ← data
MOV @ Ri, A	[Ri] ← A
MOV @ Ri, direct	[Ri] ← [direct]
MOV @ Ri, # data	[Ri] ← data
MOV DPTR, # data 16	DPTR ← data 16
MOVC A, @ A + DPTR	A ← [A + DPTR]
MOVC A, @ A + PC	A ← [A + PC]
MOVX A, @ Ri	A ← [Ri]
MOVX A, @ DPTR	A ← [DPTR]
MOVX @ Ri, A	[Ri] ← A
MOVX @ DPTR, A	[DPTR] ← A
PUSH direct	Stack ← [direct]
POP direct	[direct] ← stack
XCH A, Rn	A ← Rn, Rn ← A
XCH A, direct	A ← [direct], [direct] ← A
XCH A, @ Ri	A ← [Ri], [Ri] ← A
XCHD A, @ Ri	

4. Boolean Variable Instructions

Mnemonics	Operation
CLR C	Clear C
CLR bit	Clear direct bit
SETB C	Set C
SETB bit	Set direct bit
CPL C	Complement C
CPL bit	Complement direct bit
ANL C, bit	AND bit with C
ANL C,/bit	AND NOT bit with C
ORL C, bit	OR bit with C
ORL C, /bit	OR NOT bit with C
MOV C, bit	MOV bit to C
MOV bit, C	MOV C to bit
JC rel	Jump if C set
JNC rel	Jump if C not set
JB bit, rel	Jump if specified bit set
JNB bit, rel	Jump if specified bit not set
JBC bit, rel	If specified bit set then clear if and jump.

5. Program Branching Instructions

Mnemonics	Operation
ACALL addr11	Absolute jump to routine
LCALL addr16	Long jump to routine
RET	Return from subroutine
RETI	return from interrupt
AJMP addr11	Absolute jump
LiTMP addr16	long jump
SJMP rel	short jump
JMP@ A + DPTR	Jump relative to DPTR
JZ rel	Jump if A is zero
JNZ rel	Jump if A is not zero
CJNE A, direct, rel	
CJNE A, # data, rel	Compare and jump if not equal
CJNE RN, # data, rel	
CJNE @ Ri, # data, rel	
DJNZ Rn, rel	Decrease and jump is not
DJNZ direct, rel	Zero
NOP	No operation

9.1.10 Addressing Modes in 8051

1. **Immediate Addressing Mode**

In this addressing mode, the data is provided as a part of instruction itself. In other words data immediately follows the instruction. *For example;* MOV A, #30H ADD A, #83

2. **Register Direct Addressing Mode**

The operand is a register which holds the data to be manipulated. *For examples,* ADD A, R 5 will add content of A and R 5, and store back in A.

3. **Register Indirect Addressing Mode**

Register is used to point the effective address of the operand. Registers R0, R1 and DPTR are used as point registers which must be preceded by @ sign. *For examples,* MOV A @ R0 represents copying the contents of the address in R0 to the accumulator.

4. **Direct Addressing Mode**

In this addressing mode, the source or destination address is specified by using 8-bit data in the instruction. Only the internal data memory can be used in this mode.

For examples, MOV R2, 45 H
MOV R0, 05 H

5. **Relative Addressing**

Relative addressing is used only with conditional jump instructions. The relative address, (off set), is an 8 bit signed number, which is automatically added to the PC to make the address of the next instruction.

6. **Absolute, Addressing Mode**

Absolute addressing is used only by the AJMP (absolute jump) and ACALL (absolute call) instructions.

7. **Indexed Addressing Mode**

In this addressing mode, either the program counter (PC), or the data pointer (DPTR) is used to hold the base address and the A is used to hold the offset address.

9.2 PROGRAMMING IN ASSEMBLY LANGUAGE

The assembly language is made up of elements which all are used to write program in sequential manner. Follow the given rules to write programming in assembly language.

9.2.1 Rules of Assembly Language

- The assembly code must be written is upper letters.
- The labels must be followed by a colon (label:).
- All symbols and labels must begin with a letter.

- All comments are typed in lower case.
- The last line of the program must be the END directive.

The assembly language mnemonics are in form of op-code, such as MOV, ADD, JMP, and so on, which are used to perform the operations. Directives are used to give directions to the assembler. Generally used directives are DB, ORG, END and EQU. The ORG directive is used to set the register address during assembly. The DB (define byte) is used to allow a string of bytes. The EQU directive is used to equate address of the variable. The END directive is used to indicate the end of the program.

The assembly language program, in general, is written using following format:

[label:] Mnemonics [operands] [; comments]

9.2.2 Assembly Language Program

1. **Write a program to add the values of locations 50 H and 51 H and store the result in locations in 52 H and 53 H.**

```
ORG 0000 H      ; Set program counter 0000 H
MOV A, 50 H     ; Load the content of memory location 50 H into A
ADD A, 51 H     ; Add the contents of memory 51 H with
                  CONTENTS A
MOV 52 H, A     ; Save the LS byte of the result in 52 H
MOV A, # 00     ; Load 00 H into A
ADDC A, # 00    ; Add the immediate data and carry to A
MOV 53 H, A     ; Save the MS byte of the result in location 53 H
END
```

2. **Write a program to store data FFH into RAM memory location 50 to 58 H using direct addressing mode.**

```
ORG 0000 H      ; Set program counter 0000 H
MOV A, # 0FF H  ; Load FFH into A
MOV 50 H, A     ; Store contents of A in location 50 H
MOV 51 H, A     ; Store contents of A in location 51 H
MOV 52 H, A     ; Store content of A in location 52 H
MOV 53 H, A     ; Store content of A in location 53 H
MOV 54 H, A     ; Store content of A in location 54 H
MOV 55 H, A     ; Store content of A in location 55 H
MOV 56 H, A     ; Store content of A in location 56 H
MOV 57 H, A     ; Store content of A in location 57 H
MOV 58 H, A     ; Store content of A in location 58 H
END
```


3. **Write a program to subtract a 16 bit number stored at locations 51 H- 52 H from 55 H-56 H and store the result in locations 40 H and 41 H. Assume that the least significant byte of data or the result is stored in low address. IF the result is positive, then store 00 H, else store 01 H in 42 H.**

```
ORG 0000 H      ; Set program counter 0000 H
MOV A, 55 H     ; Load the contents of memory location 55 H into A.
CLRC            ; Clear the borrow flag.
SUBB A, 51 H    ; Sub the contents of memory 51 H from the
                  contents of A
MOV 40 H, A     ; Save the LS Byte of the result in location 40 H.
MOV A, 56 H     ; Load the content of memory location 56 H into A.
SUBB A, 52 H    ; Subtract the content of memory 52 H from the
                  content A.
MOV 41 H, A     ; Save the MS byte of the result in location 41 H.
MOV A, # 00     ; Load 005 into A.
ADDC A, # 00    ; Add the immediate data and carry flag to A.
MOV 42 H, A     ; If result is positive, store 000 H, else store 01 H in
                  42 H
END
```

4. **Write a program to add two 16 bit numbers stored at locations 51 H-52 H and 55 H-56 H and store the result in locations 40 H, 41 H and 42 H. Assume that the least significant byte of data and the result is stored in low address and the most significant byte of data or the result is stored in high address.**

```
ORG 0000 H      ; Set program counter 0000 H
MOV A, 51 H     ; Load the content of memory location 51 H into A.
ADD A, 55 H     ; Add the content of 55 H with contents of A.
MOV 40 H, A     ; Save the LS byte of the result in location 40 H.
MOV A, 52 H     ; Load the contents of 52 H into A.
ADDC A, 56 H    ; Add the contents of 56 H and CY flag with A.
MOV 41 H, A     ; Save the second byte of the result in 41 H.
MOV A, # 00     ; Load 00 H into A.
ADDC A, # 00    ; Add the immediate data 00 H and CY to A.
MOV 42 H, A     ; Save the MS byte of the result in location 42 H.
END
```

5. **Write a program to store data FFH into RAM memory locations 50 H to 58 H using indirect addressing mode.**

```
ORG 0000 H      ; Set program counter 0000 H.
MOV A, # 0FF H  ; Load FFH into A
MOV R 0, # 50 H ; Load pointer, R0-50 H
MOV R 5, # 08 H ; Load counter, R 5-08 H
```

```
Start MOV @ R 0, A ;  Copy contents of A to RAM pointed by R0.
INC R0             ;  Increment pointer
DJNZ R 5, start    ;  Repeat until R 5 is zero
END
```

6. Write a program to add two binary coded decimal (BCD) numbers stored at locations 60 H and 61 H and store the result in BCD at memory locations 52 H and 53 H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H          ;  Set program counter 0000 H
MOV A, 60 H        ;  load the contents of memory location 60 H into A
ADD A, 61 H        ;  Add the contents of memory location 61 H with
                      contents of A.
DA A               ;  Decimal adjustment of the sum in A
MOV 52 H, A        ;  Save the least significant byte of the result in
                      location 52 H
MOV A # 00         ;  Load 00H into A
ADDC A, # 00H      ;  Add the immediate data and the contents of carry
                      flag to A
MOV 53H, A         ;  Save the most significant byte of the result in
                      location 53
```

7. Write a program to clear 10 RAM locations starting at RAM address 1000 H.

```
ORG 0000H          ;  set program counter 0000 H
MOV DPTR, # 1000 H ; Copy address 1000 H to DPTR
CLR A              ;  Clear A
MOV R6, #0AH       ;  Load 0AH to R6
Again: MOV @ DPTR, A ; Clear Ram location pointed by DPTR
INC DPTR           ;  Increment DPTR
DJNZ R6, again     ;  Loop until counter R6 = 0
END
```

8. Write a program to compute 1 + 2 + 3 + N (say N = 15) and save the sum at 70 H.

```
ORG 0000 H         ;  Set program counter 0000 H
N EQU 15           ;
MOV R0, # 00       ;  Clear R0
CLR A              ;  Clear A
Again: INC R0      ;  Increment R0
ADD A, R0          ;  Add the contents of R0 with A
CJNE R0, # N, again ; Loop until counter, R0, N
MOV 70 H, A        ; Save the result in location 70 H
END
```


9. Write a program to multiply two 8 bit numbers stored at locations 70 H and 71 H and store the result at memory locations 52 H and 53 H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000 H         ;  Set program counter 0000 H
MOV A, 70 H        ;  Load the contents of memory location 70 H into A
MOV B, 71 H        ;  Load the contents of memory location 71 H in to B
MUL AB             ;  Perform multiplication
MOV 52 H, A        ;  Save the least significant byte of the result in
                      location 52 H
MOV 53 H, B        ;  Save the most significant byte of the result in
                      location 53 H
END
```

10. Ten 8 bit numbers are stored in internal data memory from location 50 H. Write a program to increment the data.

Assume that ten 8 bit numbers are stored in internal data memory from location 50 H, hence R0 or R1 must be used as a pointer.

The program is as follows

```
ORG 0000 H
MOV R0, # 50 H
MOV R3, # 0AH
Loopl : INC @ R0
INC R0
DJNZ R3, loopl
END
```

11. Write a program to find the average of five 8 bit numbers. Store the result in H. (Assume that after adding five 8 bit numbers, the result is 8 bit only.

```
ORG 0000 H
MOV 40 H, # 05 H
MOV 41 H, # 55 H
MOV 42 H, # 06 H
MOV 43 H, # 1 AH
MOV 44 H, # 09 H
MOV R0, # 40 H
MOV R5, # 05 H
MOV B, R5
CLR A
Loop: ADD A, @ R0
INC R0
DJNZ R 5, loop
DIV AB
```

```
MOV 55 H, A
END
```

12. Write a program to find the cube of an 8 bit number.

```
ORG 0000 H
MOV R1, # N
MOV A, R1
MOV B, R1
MUL AB
MOV R2, B
MOV B, R1
MUL AB
MOV 50, A
MOV 51, B
MOV A, R2
MOV B, R1
MUL AB
ADD A, 51 H
MOV 51 H, A
MOV 52 H, B
MOV A, # 00H
ADDC A, 52 H
MOV 52 H, A
END
```

13. Write a program to exchange the lower nibble of data present in external memory 6000 H and 6001 H.

```
ORG 0000 H          ;  Set program  counter 00H
MOV DPTR, # 6000 H  Copy address 6000 H to DPTR
MOVX A, @ DPTR  ;   Copy contents of 6000⁸ to A
MOV R0, #45 H       ;   Load pointer, R0 = 45 H
MOV @ R0, A         ;   Copy cont of A to RAM pointed by 80
INC DPL             ;   Increment pointer
MOVX A, @ DPTR  ;   Copy contents of 60018 to A
XCHD A, @ R0        ;   Exchange lower nibble of A with RAM pointed by
                        R0
MOVX @ DPTR, A  ;   Copy contents of A to 60018
DEC DPL             ;   Decrement pointer
MOV A, @ R0         ;   Copy cont of RAM pointed by R0 to A
MOV @ DPTR, A   ;   Copy cont of A to RAM pointed by DPTR
END
```


14. Write a program to count the number of 1's and 0's of 8 bit data stored in location 6000 H.

```
ORG 00008           ;  Set program counter 00008
MOV DPTR, # 60000H ; Copy address 6000 H to DPTR
MOVX A, @ DPTR  ;   Copy number to A
MOV R0, # 08        ;   Copy 08 in R0
MOV R2, # 00        ;   Copy 00 in R2
MOV R3, # 00        ;   Copy 00 in R3
CLR C               ;   Clear carry flag
BACK: RLC A         ;   Rotate A through carry flag
JC NEXT             ;   If CF = 1, branch to next
INC R2              ;   If CF = 0, increment R2 AJMP NEXT 2
NEXT: INC R3        ;   If CF = 1, increment R3
NEXT2: DJNZ R0; BACK; Repeat until R0 is zero
END
```

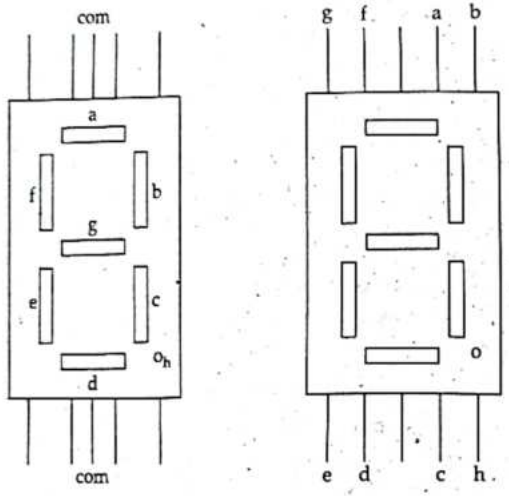
15. Write a program to shift a 24 bit number stored at 57 H – 55 H to the left logically four places. Assume that the least significant byte of data is stored in lower address.

```
ORG 0000H           ;  Set program counter 0000H
MOV R1, # 04        ;   Set up loop count to 4
Again: MOV A, 55H ; Place the least significant byte of data in A
CLR C               ;   Clear the carry flag
RLC A               ;   Rotate contents of A(55 H) left through carry
MOV 55 H, A
MOV A, 56 H
RLC A               ;   Rotate contents of A (56 H) left through carry
MOV 56 H, A
MOV A, 57 H
RLC A               ;   Rotate contents of A(57 H) left through carry
MOV 57 H, A
DJNZ R1, again      ;   Repeat until R1 is zero
END
```

9.3 INTERFACING WITH SEVEN SEGMENT DISPLAY

A note about 7 segment LED display: 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters like A, b, C, H, ·, E, e, F, n, o, t, u, y, etc. A seven segment display consists of seven LEDs arranged in the form of a squarish '8' slightly inclined to

the right and a single LED as the dot character. Different characters can be displayed by selectively glowing the required LED segments. Seven segment displays are of two types, common cathode and common anode. In common cathode type, the cathode of all LEDs are tied together to a single terminal which is usually labeled as 'com' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g and h(or dot). In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.



9.3.1 Digit Drive Pattern

Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals 'a' to 'h' in order to display different digits and characters. The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Interfacing seven segment display to 8051.

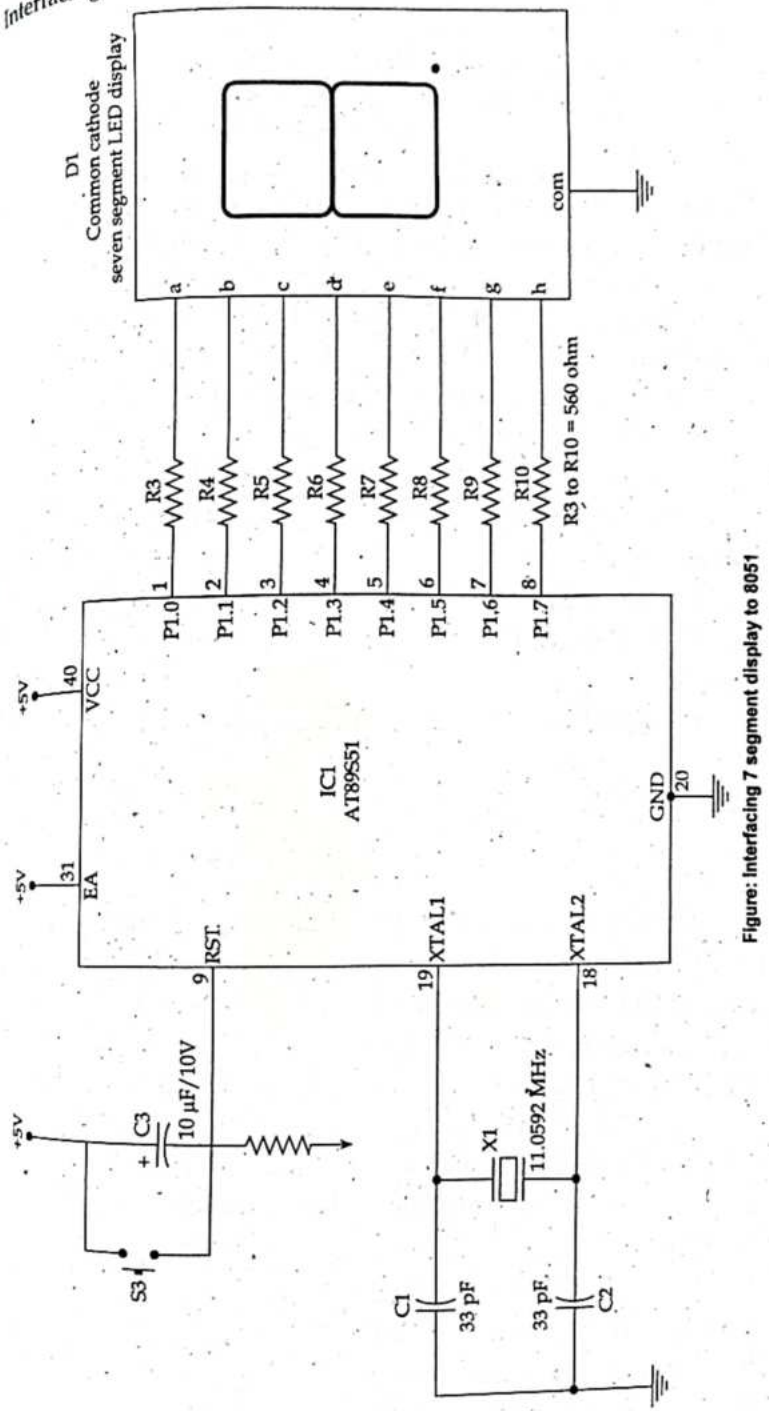


Figure: Interfacing 7 segment display to 8051

The circuit diagram shown above is of an AT89S51 microcontroller based 0 to 9 counter which has a 7 segment LED display interfaced to it in order to display the count. This simple circuit illustrates two things. How to setup simple 0 to 9 up counter using 8051 and more importantly how to interface a seven segment LED display to 8051 in order to display a particular result. The common cathode seven segment display D1 is connected to the port 1 of the microcontroller (AT 89S51) as shown in the circuit diagram. R3 to R10 are current limiting resistors. S3 is the reset switch and R2, C3 forms a debouncing circuitry. C1, C2 and X1 are related to the clock circuit. The software part of the project has to do the following tasks.

- Form a 0 to 9 counter with a predetermined delay (around 1/2 second here).
- Convert the current count into digit drive pattern.
- Put the current digit drive pattern into a port for displaying.

Program

```
ORG 000 H // initial starting address
START: MOV A, # 00001001 B //initial value of accumulator
MOV B, A
MOV R0, # 0AH // Register R0 initialized as counter which counts from 10
                to 0
LABEL: MOV A, B
INC A
MOV B, A
MOVC A, @ A + PC // adds the byte in A to the program counters address
MOV P1, A
ACALL DELAY // calls the delay of the timer
DEC R0 // counter R0 decremented by 1
MOV A, R0 // R0 moved to accumulator to check if it is zero in next
            instruction
JZ START // Checks accumulator for zero and jumps to START. Done to
           check if counting has been finished
SJMP LABEL
DB   3 FH  // digit drive pattern for 0
DB   06 H  // digit drive pattern for 1
DB   5 BH  // digit drive pattern for 2
DB   4 FH  // digit drive pattern for 3
```



```
DB   66 H  // digit drive pattern for 4
DB   6 DH  // digit drive pattern for 5
DB   7 DH  // digit drive pattern for 6
DB   07 H  // digit drive pattern for 7
DB   7 FH  // digit drive pattern for 8
DB   6 FH  // digit drive pattern for 9
DELAY : MOV R4, # 05H // subroutine for delay
WAIT 1 : MOV R3, # 00H
WAIT 2 : MOV R2, # 00H
WAIT 3 : DJNZ R2, WAIT 3
DJNZ R3, WAIT 2
DJNZ R4, WAIT 1
RET
END
```

OLD EXAM QUESTIONS SOLUTION [TU]

1. Why 8051 microcontroller is used? Write an assembly program to get data from P0 and send it to P1 and compare with corresponding C program. [2069 Bhadra]

Answer:

First part:

Microcontroller is a compact tiny computer that is fabricated inside a chip and is used in automatic control systems including security systems, office machines, power tools, alarming system, traffic light control, washing machine, and much more. It was specially built for embedded system and consisted of read write memory, read only memory, I/O ports, processor and built in clock.

Second part:

```
ORG 00H
MOV P0, # 0FFH
MOV P1 # 00H
BACK: MOV A, P0    ; get data from P0
MOV P1, A          ; send it to part 1 (P1)
SJMP BACK          ; keep doing it
END
```

2. Show the internal structure of the 8051 micro-controller. Provide a comparison chart of the 8051 family members. [2070 Bhadra]

Answer:

First part: See the topic 9.1.6 of chapter 9.

Second part: See the topic 9.1.5 of chapter 9.

3. Describe in detail the addressing modes of the 8051 microcontroller. Write a short program in assembly language that computes a precise 2.5 ms delay using the 8051 microcontroller. [2071 Magh]

Answer:

First part: See the topic 9.1.10 of chapter 9.

Second part:

```
ORG 00H
        MOV R4, # 64 H; MC = 1 executes only once
OUTER: MOV R3, # 0AH; MC = 1, executes 100 times
INNER: DJNZ R3, INNER; MC = 2, executes 100 × 10 = 1000 times
        DJNZ R4, OUTER; MC = 2 executes 100 times
        NOP; MC = 1, executes 1 time
        NOP; MC = 1 executes 1 time
        NOP; MC = 1 executes 1 time
        NOP; MC = 1, executes 1 time
END
```


In 8051, crystel frequency is 11.0592 MHz and one machine cycle (MC) is equal to 12 clock cycles.

so, 1 MC = 12/11.0592 MHz = 1.085 μs

Total machine cycles required to execute above code is

$= 1 + 1 \times 100 + 2 \times 1000 + 2 \times 100 + 1 + 1 + 1 + 1$

$= 2305$ MC

Total time duration = 2305 × 1.085 μs = 2.5 ms

However, to generate the specified delay, we need to determine total repetition, initial value of R_4 and R_3 along with number of NOP required. In this case, we have to follow following steps.

- List the given values

  Time required (T) = 2.5 ms

  1 MC (t) = 1.085 μs (for 8051 microcontroller)
- Calculate total required machine cycles

  $\text{Total MC} = \frac{T}{t} = 2305$

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 initialize as 100 (64 H) or 200 (C 8), 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.

4. Write an assembly and C program to display 0 to 9 in seven segment display. [2071 Bhadra]

Answer: See the topic 9.3 of chapter 9.

5. Explain the addressing modes used in 8051 micro-controller with example. Write an assembly language program to blink the 8 LEDs connected at port 2 of the 8051 microcontroller. [2072 Magh]

Answer:

First part: See the topic 9.1.10 of chapter 9.

Second part:

```
        ORG 00H
        MOV P2 # 00H
MAIN: MOV P2 # 0FFH
        LCALL DELAY
        MOV P2 # 00H
        LCALL DELAY
        SJMP MAIN
ORG 300H
OELAY:
        MOV R3, # 0FOH
DEL1 : MOV R2, # 0FAH
DEL2 : DJNZ R2, DEL2
        DJNZ R3 DEL1
RET
END
```

6. Draw the pin diagram of 8051 microcontroller and explain ports 1 and 2 only. Write a program using c-programming language to find the sum between two 8-bit BCD data stored in RAM locations 50 H and 51 H and store the BCD sum at RAM locations 52 H and 53 H. [2072 Ashwin]

Answer:

First part: See the topic 9.1.7 of chapter 9.

Second part: Solve as above questions.

7. Describe the different purpose of port 3 and port 2 of 8051 microcontroller. Write an assembly language programming for 8051 microcontroller to read the data from switches connected at port 1 and send it to port 2 for display in LED. [2073 Magh]

Answer:

First part:

Port 3 [pins 10-17]: See 9.1.7 of chapter 9 [Each of these pins can serve as general input or output]

Port 2 [pins 21-28]: See 9.1.7 of chapter 9. [If there is no intention to use external memory is used, the higher address byte, *i.e.*, address A8-A15 will appear on this port. Even though memory with capacity of 64 kb is not used, which means that not all eight port bits are used for its addressing, the rest of them are not available as inputs/outputs.]

Second part:

```
ORG 00H
MOV P1, # 0FFH
MOV P2, # 00H
MAIN
        MOV A, P1
        MOV P2, A
        SJMP Main
        END
```

8. What are the important operational features of 8051 microcontroller? Write an assembly program to display 0 to 9 in seven segments display. [2073 Bhadra]

Answer:

First part: See the topic 9.1.6.1 of chapter 9.

Second part: See the topic 9.3 of chapter 9.

9. Draw the circuit diagram of the minimum configuration for 8051 microcontroller to operate. Also show the connection of LED at P1.7 and switch at P1.1 in the same circuit. Using an assembly language, generate a pulse of 75% duty cycle at pin P1.7 when the switch at 1.1 is ON. [2074 Bhadra]

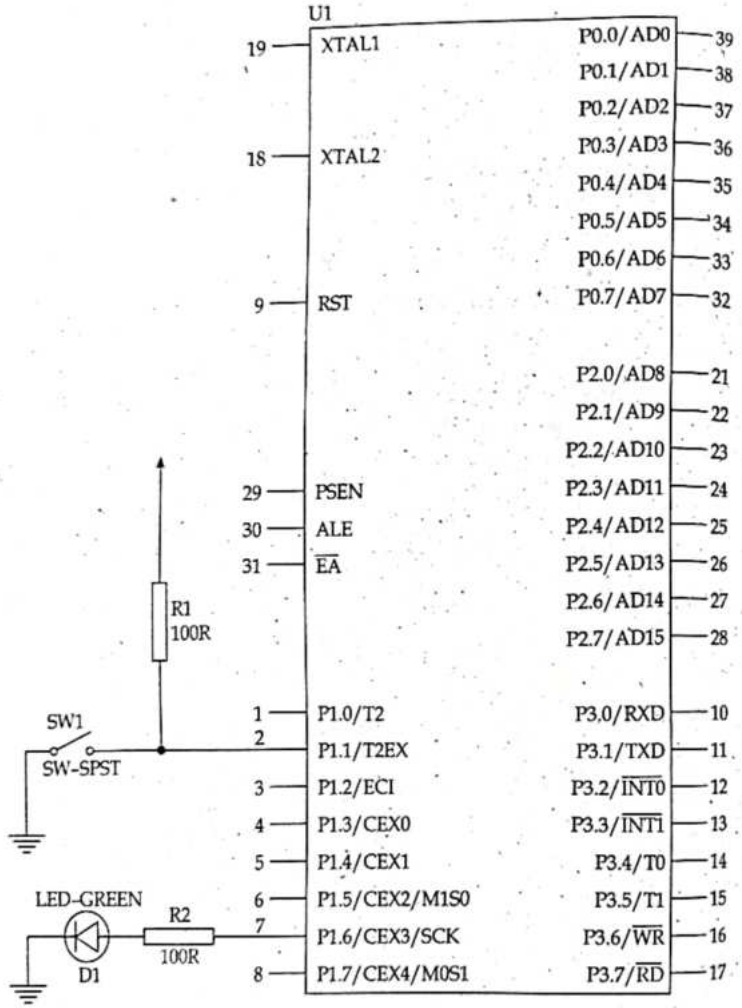
Answer:

First part: See the topic 9.1.8 of chapter 9.

Second part:

```
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
DELAY:
        MOV R3, # 0F0H
```

```
DEL1 : MOV R2, # 0FAH
DEL2 : DJNZ R2, DEL2
       DJNZ R3, DEL1
       RET
END
```



10. Explain the addressing modes used in 8051 micro-controller with example. Write an assembly language programming to blink the 8 LED connected at port 2 of the 8051 microcontroller. [2075 Baisakh]

Answer:

First part: See the topic 9.1.10 of chapter 9.

Second part:

```
        ORG 00H
        MOV P2, # 00H
MAIN: MOV P2, # 0FFH
        LCALL DELAY
        MOV P2, # 00H
        LCALL DELAY
        SJMP MAIN
ORG 300 H
DELAY:
        MOV R3 # 0F0H
DEL1: MOV R2 # 0FAH
DEL2 : DJNZ R2, DEL2
        DJNZ R3, DEL1
RET
END
```

11. Generate a periodic square wave having a period of 15 ms and a duty cycle of 20% in 8051 using assembly programming. The waveform should be produced at pin zero of port two (P2.0) the XTAL frequency is 11.0592 MHz and use Timer 1 in mode 0 (13-bit timer mode). [2076 Bhadra]

Answer:

```
ORG 00H
CLR P2.0
MAIN:
        SETB P2.0
        LCALL DELAY
        CLR P2.0
        LCALL DELAY
        LCALL DELAY
        LCALL DELAY
        LCALL DELAY
        SJMP MAIN
DELAY:
        MOV TMOD, # 00H; to configure timer in mode 0
        MOV TH1, # 15 H
        MOV TLO, # 33 H; 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
```

Period = 15 ms

Duty cycle = 3/15 = 20%

On time = 3ms, off – time = 12 ms

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) * 1.085 μs = 3 ms

x = 5427 in decimal (1533 in hex)

Hence, TH1 = 15H and TL1 = 33H must to be loaded to generate 3 ms.

12. Describe the addressing mode of 8051 microcontroller. Write C programming language to display 0 to 9 in seven segment connected to port 1 of 8051 microcontroller. [2077 Poush]

Answer:

First part: See the topic 9.1.10 of chapter 9.

Second part: See the topic 9.3 of chapter 9.

OLD EXAM QUESTIONS SOLUTION (PoU)

1. Explain briefly the architecture of 8051 microcontroller with the aid of block diagram. [2016 Spring]

Answer: See the topic 9.1.6 of chapter 9.

2. Draw the block diagram of 8051 microcontroller also write an assembly level program to interface a seven segment display with 8051 microcontroller. [2017 Fall]

Answer:

First part: See the topic 9.1.2 of chapter 9.

Second part: See the topic 9.3 of chapter 9.

3. Explain the architecture of 8051 microcontroller with the aid of its block diagram. Also explain the addressing modes of 8051. [2017 Spring]

Answer:

First part: See the topic 9.1.6 and 9.1.2 of chapter 9.

Second part: See the topic 9.1.10 of chapter 9.

4. Write an ALP in 8051 to implement seven segment and implement counter that counts two digit hexadecimal number. [2018 Fall]

Answer:

```
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 DPRT, # LABEL1
MAIN:
      MOV A, R6
      SETB P3.0; activates 2nd display to display lower byte
      CLR P3.1; deactivates the 1st display
      LCALL DISPLAY
      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
```

```
        MOV R5, # 07H
        DEC R6
        CJNE R6, # -1 MAIN
        MOV R6, #.09H
        DEC R7
        CJNE R7, #-1 MAIN
        MOV R7 # 09H
        SJMP MAIN
DISPLAY:
        MOVC A, @ A + DPTR
        MOV P2, A
RET
DELAY:
        MOV R3, # 0F0H
        DEL1 : MOV R2, # 0FAH
        DEL2 : DJNz R2, DEL2
        DJNZ R3 DEL1
RET
LABEL1
        DB 3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H, 7FH, 6FH
END
```

5. **Write an assembly language program for 8051 microcontroller to display the number from 0 to 9 in a seven segment display.**

[2019 Spring-2020 Fall]

Answer: See the topic 9.3 of chapter 9.

Chapter 10

VHDL

★★★

★★★

10.1 INTRODUCTION

VHDL is a description language for digital electronic circuits that is used in different levels of abstraction. The VHDL acronym stands for VHSIC (very high speed integrated circuits) Hardware description language. This means that VHDL can be used to accelerate the design process.

It is very important to point out that VHDL is NOT a programming language. Therefore, knowing its syntax does not necessarily mean being able to designing digital circuits with it.

Why to use an DHL?

- To discover problems and faults in the design before actually implementing it in hardware.

- The complexity of an electronic system grows exponentially. For this reason, it is very convenient to build a prototype of the circuit previously to its manufacturing process.
- It makes easy for a team of developers to work together.

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

Each VHDL design unit comprises an "entity" declaration and one or more "architectures". Each architecture defines a different implementation or model of a given design unit. The entity definition defines the inputs to, and outputs from the module, and any "generic" parameters used by the different implementations of the module.

Entity Declaration format

```
Entity name is
        Port (port definition list); – input/output signal ports
        generic (generic list); – optional generic list
        end name;
```

Port declaration format: port-name: mode data-type;

The mode of a port defines the directions of the signals on that port, and is one of: in, out, buffer, or input.

Port Modes:

An in port: Can be read but not updated within the module, carrying information in to the module. (An in port cannot appear on the left hand side of a signal assignment)

An out port: Can be updated but not read within the module, carrying information out of the module. (An out port cannot appear on the right hand side of a signal assignment).

A buffer port: Likewise carries information out of a module, but can be both updated and read within the module.

An in out port: Is bidirectional and can be both read and updated, with multiple update sources possible.

NOTE: A buffer is strictly an output port, *i.e.*, can only be driven from within the module, while in out is truly bidirectional with drivers both within and external to the module.

Example:

```
Entity counter is
Port (Incr, load, clock: in bit;
        Carry: out bit;
        Data-Out: buffer bit-vector (7 down to 0);
        Data-In: in bit-vector (7 down to 0);
        end counter;
```

Generics allow static information to be communicated to a block from its environment for all architectures of a design unit. These include timing information (set up, hold, delay times), part sizes, and other parameters.

Example:

```
Entity and-gate is
        port (a, b : in bit);
        c : out bit);
        generic (gate-dealy : time : = 5ns);
        end and-gate;
```

Architecture:

An architecture defines one particular implementation of a design unit, at some desired level of abstraction.

```
Architecture arch-name of entity-name is
        .... declarations...
begin
        .... concurrent statements....
end
```

Declarations include data types, constants, signals, files, components, attributes, subprograms, and other information to be used in the implementation description. Concurrent statements describe a design unit at one or more levels of modeling abstraction, including dataflow, structure, and/or behavior.

- Behavioral model: No structure or technology implied. Usually written in sequential, procedural style.
- Dataflow model: All data paths shown, plus all control signals.
- Structural model: Interconnection of components

10.3 DATA TYPES, DATA OBJECTS AND OPERATORS

10.3.1 Data Types

VHDL has a set of standard data types (pre defined/built-in). It is also possible to have user defined data types and subtypes.

Some of the predefined data types in VHDL are: BIT, BOOLEAN and INTEGER.

The STD_LOGIC and STD_LOGIC_VECTOR data types are not built-in VHDL data types, but are defined in the standard logic 1164 package of the IEEE library. We therefore need to include this library in our VHDL code and specify that the STD_LOGIC_1164 package must be used in order to use the STD_LOGIC data type.

```
Library IEEE;
Use IEEE. STD_LOGIC_1164. ALL;
```

BIT:

The BIT data type can only have the value 0 or 1. When assigning a value 0 or 1 to a BIT in VHDL code, the 0 or 1 must be enclosed in single quotes: '0' or '1'.

BIT_VECTOR:

The BIT_VECTOR data type is the vector version of the BIT type consisting of two or more bits. Each bit in a BIT_VECTOR can only have the value 0 or 1.

When assigning a value to BIT_VECTOR, the value must be enclosed in double quotes, *for example*: "1011" and the number of bits in the value must match the size of the BIT_VECTOR.

STD_LOGIC:

The STD_LOGIC data type can have the value X, 0, 1 or Z. There are other values that this data type can have, but the other values are not synthesizable-*i.e.*, they cannot be used in VHDL code that will be implemented on a CPLD or FPGA.

These values have the following meanings

X-Unknown
0-Logic 0
1-Logic 1
Z-High impedance (open circuit)/tristate buffer

when assigning a value to a STD_LOGIC data type, the value must be enclosed in single quotes 'X' '0' '1' or 'Z'.

STD_LOGIC_VECTOR

The vector version of STD_LOGIC data type. Each bit in the set of bits that make up the vector can have the value X, 0, 1 or Z. When assigning a value to a STD_LOGIC_VECTOR type, the value must be enclosed in double quotes. *For example;* "1010", "ZZZZ" or "ZZ001". The number of bits in the value must match the size of the STD_LOGIC_VECTOR.

10.3.2 Data Objects

A data object holds a value of a specified type, created by using object declaration. Every data object belongs to one of the following three classes.

I) Constant

An object of constant class can hold a single value of a given type. This value is assigned to the object before simulation starts and the value cannot be changed during the course of the simulation.

For example:

```
constant RISE_TIME: TIME : = 10 ns;
constant BUS_WIDTH: INTEGER : = 8 ;
```

The first declaration declares the object RISE_TIME that can hold a value of type TIME, with the value assigned at the start of simulation is 10 ns. The second constant declaration declares a constant BUS_WIDTH of type INTEGER with a value of 8. The value of the constant has not been specified, called as a deferred constant and it can appear only inside a package declaration. The complete constant declaration with the associated value must appear in the corresponding package body.

```
constant NO_OF_INPUTS: INTEGAR;
```

II) Variable

An object of variable class can also hold a single value of a given type. However in this case, different values can be assigned to the object at different times using a variable assignment statement.

For example,

```
variable COUNT : INTEFER;
```

This results in the creation of a data object called COUNT which can hold integer values and the object COUNT is also declared to be variable class.

```
variable CTRL_STATUS : BIT_VECTOR (10 down to 0);
variable SUM : INTEGER range 0 to 100 : = 10;
variable FOUND, DONE : BOOLEAN;
```

The first declaration specifies a variable object CTRL_STATUS as an array of 11 elements, with each array element of type BIT.

In the second declaration, an explicit initial value has been assigned to the variable SUM, with an initial value of 10 at the start of simulation.

If no initial value is specified for a variable object, a default value is used as an initial value. This default value is TLEFT, where T is the object type and LEFT is a predefined attribute of a type that gives the leftmost value in the set of values belonging to type T.

In the third declaration, the initial values assigned to FOUND and DONE at start of simulation is FALSE. The initial value for all the array elements of CTRL_STATUS is 'O'.

III) Signal

An object belonging to the signal class has a past history of values, a current values and a set of future values. Future values can be assigned to a signal object using a signal assignment statement.

The signal objects can be regarded as wires in a circuit while variable and constant objects are analogous to their counterparts in a high-level

programming language like C or Pascal. Signal objects are typically used to model wires and flip-flops while variable and constant objects are typically used to model the behaviour of the circuit.

```
signal CLOCK : BIT;
signal DATA_BUS : BIT_VECTOR (0 to 7);
signal GATE_DELAY : TIME : = 10ns;
```

The first signal declaration declares the signal object CLOCK of type BIT and given it an initial values of 'O'.

The second signal declaration declares the signal object DATA_BUS of type BIT_VECTOR with 1-dimensional array of 8 bits with an initial value of "00000000". The third signal declaration declares a signal object GATE_DELAY of type TIME that has an initial value of 10 ns.

10.3.3 Operators

VHDL includes the following kinds of operators:

- Logical
- Relational
- Arithmetic
- Assignment
- Shift

i) Logical Operators

Logical operators, when combined with signals and/or variables, are used to create combinational logic. VHDL provides the following logical operators: and, or, nand, nor, xor, not, xnor.

Operators	Description	Supported data types
NOT	Inverts the signal, high precedence	BIT STD_LOGIC STD_ULOGIC BIT_VECTOR STD_LOGIC_VECTOR STD_ULOGIC_VECTOR
AND	Result high when both inputs is high	
OR	Result high when one of the input is high	
NAND	X < = a NAND b	
NOR	Z < = NOT a NOR b	
XOR	Complements the bit when XORed with 1	
XNOR	Complements the bit when XNORed with 0	

ii) Relational Operators

Relational operators are used to create equality or magnitude comparison functions. VHDL provided the following relational operators.

Operators	Description
=	Equal to
/=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

iii) Arithmetic Operators:

Arithmetic operators are used to create arithmetic functions. VHDL provides the following arithmetic operators.

Operator	Meaning	Description
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Limited to powers of two
mod	Modulus	XMODY results value with sign of Y
rem	Remainder	XREMY results value with sign of X
abs	Absolute value	
**	Exponentiation	Limited to powers of two

iv) Assignment operators

Operators	Assign Value To	Examples
<=	Signal	X<= '1'; Y<="101";
:=	Variable, constant, generic, and for initialization	Z:= "1001"; Z is a variable
=>	Individual elements or with OTHERS	W<=(0=>'1', OTHERS =>'0') LSB assigned 1 and others as 0

v) Shift operators

Operators	Meaning
SLL	Shift left logic
SRL	Shift right logic
SLA	Shift left arithmetic
SRA	Shift right arithmetic
ROL	Rotate left
ROR	Rotate right

10.4 STATEMENTS IN VHDL

10.4.1 Concurrent Statements

Concurrent Signal Assignment

Syntax:

```
Target <= expression;
```

Examples:

```
A <= B NAND C;
X <= (D OR E) AND (F AND F);
```

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';
```

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";
```

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;
```

10.4.2 Behavior Style Architecture

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;
ARCHITERCTURE HALF_ADDER_ARCH OF HALF_ADDER IS BEGIN
    PROCESS_ADDER: PROCESS (A,B)
BEGIN
    S< = XOR B;
    C< = A AND B;
END PROCESS PROCESS_ADDER;
END HALF_ADDER_ARCH;
```

10.4.3 Structural Style Architecture

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.

- 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 of 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 implified mapping, only internal signals and signals of entity of the system are listed. Though it use less space, but it requires the signals be placed in the proper order.

10.5 FSM (FINITE STATE MACHINE) DESIGN

A FSM is specified by five entities: symbolic states, input 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.

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 flops 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.

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;
ENDIF;
END PROCESS;
```


OLD EXAM QUESTION SOLUTION (TU)

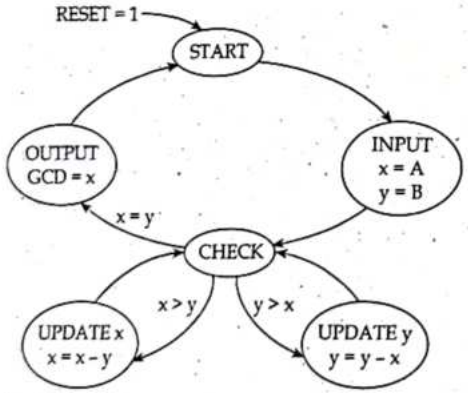
1. Write the VHDL code for processor (GCD) that calculates greatest common divisor of two integer data with its state diagram. [2069 Bhadra]

Answer:

VDHL Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FSM_GCD is
port (
        RESET, CLK; in std_logic;
        GO: in std_logic;
        A, B: 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 process seq_proc;
comb_proc: process (A, B, PS)
variable x, y: integer;
begin
    case PS is
    when START ⇒
        GCD < = 0;
        NS < = INPUT;
    when INPUT ⇒
        x: = A;
        y: = B;
        NS < = CHECK;
    when CHECK ⇒
        If (x, y) then
```

```
            NS < = UPDATEx;
        elsif (x, y) then
            NS < = UPDATEy;
        else
            NS < = OUTPUT;
        end if;
    when UPDATE x ⇒
            x: = x-y;
            NS < = CHECK;
    when UPDATE y ⇒
            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;
```

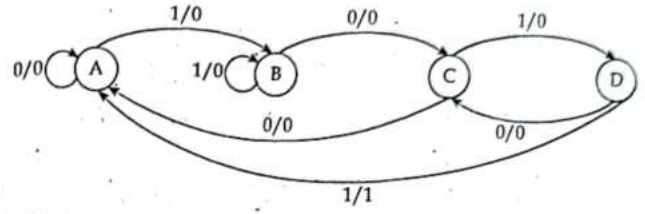


2. **How does FPGA differ from a microcontroller? Design a sequence detector for the string '1011' that output a 1 when the input matches this string, show the FSM and its VHDL implementation.** [2070 Magh]

Answer:

One of the main differences between a microcontroller and FPGA is that FPGA doesn't have a fixed hardware structure, while a microcontroller does. While FPGAs included fixed logic cells, these, along with the interconnects, can be programmed in parallel by using HDL coding language. FPGAs are not predefined and can be altered based on the user's applications.

Second part:




```
library IEEE;
use IEEE. STD_LOGIC_1164.ALL;
entity seq_det is
port (clk: in std_logic;
        reset: in std_logic;
        seq: in std_logic;
        det_vld: out std_logic
    );
end seq_det;
architecture Behavioural of seq_det is
type state_type is (A, B, C,D);
signal state: state_type = A;
begin
            process (clk)
            begin
            if (reset = '1') then
            det_vld < = '0';
            state < = A;
        elsif (rising_edge (clk)) then
            case state is
            when A =>
            det_vld <= '0';
            if (seq = '0') then
            state < = A;
        else
            state < = B;
        endif;
        when B =>
        if (seq = '0') then
            state < = C;
        else
            state < = B;
        end if;
        when C ⇒
```

```
if (seq = '0') then
    State < =A;
else
    state < = D;
end if;
when D ⇒
if (seq = '0') then
    state < = C;
else
    state < = A;
    det_ vld <= '1';
end if;
when others ⇒
    NULL;
end case;
end if;
end process;
end Behavioural;
```

3. Write an algorithm and VHDL code for a custom processor that calculates Greatest common Divisor (GCD). [2071 Bhadra]

Answer:

Algorithm:

```
int x, y;
while (1)
{
    While (! Go);
    x =A:
    y = B;
    while (x ! = y)
    {
        if (x < y)
            y = y - x;
        else
            x = x - y;
    }
    GCD = x;
}
```

VHDL code

Same as question number 1 (Old Exam Question Solution TU).

4. What is behavioral modeling? Write VHDL code for 2-input multiplexer. [2072 Magh]

Answer:

First part: See the topic 10.4.2 of chapter 10.

Second part:

```
Library IEEE;
Use IEEE. STD_LOGIC_1164. ALL;
Entity mux_4 * 1 is
Port (
    in_a: in std_logic_vector (1 down to 0);
    in_b: in std_logic_vector (1 down to 0);
    in_c: in std_logic_vector (1 down to 0);
    in_d: in std_logic_vector 1 down to 0);
    sel: in std_logic_vector (1 down to 0);
    z_out: out std_logic_vector (1 down to 0);
    );
end mux_4 × 1;
architecture mux_arch of mux_ 4 × 1 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 = "10") then
        z_out < = in_c;
    else
        z_out < = in_d;
    end if;
end process proc;
end mux_arch;
```

5. Write an algorithm and VHDL code for a custom processor that calculates Least Common Multiple (LCM) of two numbers. [2072 Ashwin]

Answer:

Algorithm

```
int a, b, c, GCD;
while (1)
    {
    while (reset);
        a = G;
        b = P;
        c = a*b;
```

```
        while (a! = b)
        {
            If (a < b)
                b = b - a;
            else
                a = a - b;
        }
        GCD = a;
        LCM = c/GCD;
    }
```

VHDL code:

```
library IEEE.
use IEEE. STD_LOGIC_1164. ALL;
use IEEE. STD_LOGIC_ARITH. ALL;
entity fsm_lcm is
port (
    reset: in std_logic;
    clk: in std_logic;
    G,P: in integer;
    LCM: out integer;
    );
end fsm_lcm;
architecture behavioral of fsm_lcm is
type state is (START, INPUT, CHECK, UPDATEa, UPDATEb, 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 (G, P, ps)
variable a, b, c, GCD: integer;
begin
    case ps is
        when start ⇒
            LCM < = 0;
            ns < = INPUT;
```



```
        when INPUT ⇒
            a: = G;
            b: = P;
            c: = a*b;
            ns < = CHECK;
        when CHECK ⇒
            if (a> b) then
                ns < = UPDATEa;
            elsif (a<b) then
                ns < = UPDATEb;
            else
                ns < = OUTPUT;
            end if;
        when UPDATEa ⇒
            a: = a - b;
            ns < = CHECK;
        when UPDATEb ⇒
            b: = b - a;
            ns < = CHECK;
        when OUTPUT ⇒
            GCD: = a;
            LCM < = c/GCD;
            ns < = INPUT;
        when OTHERS ⇒
            LCM < = 0;
            ns < = INPUT;
    end case;
end process comb_proc;
end behavioral;
```

6. Explain PROCESS in VHDL. Write a VHDL code for a full adder using 2 half adder as component. [2073 Magh]

Answer:

First part:

A PROCESS is a VHDL code section that runs in a specific sequence. The presence of IF, WAIT, CASE, LOOP, and a sensitivity list distinguishes it. When a signal in the sensitivity list changes, the process is executed.

VHDL code:

```
Library IEEE;
use IEEE.STD_LOGIC_1164. ALL;
entity fulladder is
port (
    x : in std_logic;
```

```
            y : in std_logic;
            cin : in std_logic;
            s : out std_logic;
            c : out std_logic
        );
end fulladder;
architecture structural of fulladder is
component or_gate is
port (
            x, y: in std_logic
            z: out std_logic
        );
end component or_gate;
component half_adder is
port (
            x, y: 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 (x ⇒ x, y ⇒ y,
                sum ⇒ S1, carry_out ⇒ C1);
            HA2: half_adder port map (x ⇒ S1, y ⇒ cin,
                sum ⇒ sum, carry_out ⇒ C2);
            OR1: or_gate port map (x ⇒ C1, y ⇒ C2, z ⇒ C);
end architecture structural;
```

7. Write an algorithm and Finite State Machine VHDL code for a custom processor that calculates Greatest Common Divisor of two numbers. [2073 Bhadra]

Answer: Same as question number 1 [Old Exam Question Solution TU].

8. Write an algorithm and VHDL code for a custom processor that calculates Least Common Multiple (LCM) of two numbers as a finite state machine. [2074 Bhadra]

Answer: Same as question number 6 [Old Exam Question Solution TU].

9. Explain COMPONENT with its declaration, Write a VHDL code for a JK flip-flop using PROCESS. [2075 Balshakh]

Answer:

First part: See the topic 10.4.3 of chapter 10.

Syntax of component Declaration:

COMPONENT COMPONENT_NAME [IS]


```
PORT (
            PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
            PORT_NAME: SIGNAL_MODE SIGNAL_TYPE;
            ........
        );
```

Second part:

```
library IEEE;
use IEEE. STD_LOGIC
entity jk is
port (
            j: in STD_LOGIC;
            k: in STD_LOGIC;
            clk: in STD_LOGIC;
            reset: in STD_LOGIC;
            q: out STD_LOGIC;
            qb: out STD_LOGIC
        );
        end jk;
architecture virat of jk is
begin
        jkff: process (j, k, clk, reset) is
        variable m: std_logic: = '0';
        begin
            if (reset = '1') then
                m: = '0';
            elsif (rising_edge (clk)) then
            if (j / = k) then
                m: = j;
            elsif (j = '1' and k = '1') then
            m: = not m;
        end if;
end if;
            q < = m;
            qb < = not m;
        end process jkff;
end virat;
```

10. Write an algorithm and VHDL code for custom single purpose processor that calculate the Greatest Common Divisor (GCD) of two numbers as finite State Machine. [2076 Bhadra]

Answer: Same as question number 4 [Old Exam Question Solution TU].

OLD QUESTION SOLUTION (PoU)

1. Explain different modeling styles in VHDL. [2016 Fall]

Answer: See the topic 10.4 of chapter 10.

2. Write a VHDL code which results an output '1' when a sequence '1011' is detected else an output results a '0'. [2016 Fall] [2019 Fall] [2019 Spring]

Answer: Same as question number 2 [Old Exam Question Solution TU].

3. Give a introduction to VHDL and explain the basic structure of a VHDL file with examples. [2016 Spring]

Answer: See the topic 10.1 and 10.2 of chapter 10.

4. Discuss the various style of modeling used in architecture body of hardware a description in VHDL with suitable example. [2017 Fall] [2020 Fall]

Answer: See the topic 10.4 of chapter 10.

5. Give the VHDL implementation of full adder using both behavioral and structure architecture (component). And then point out the difference between the two architecture. [2017 Spring]

Answer:

Full adder using structure architecture:

Same as question number 6 (second part) [Old Exam Question Solution TU]

Full adder using behavioral architecture:

```
library IEEE;
use IEEE. STD_LOGIC_1164. ALL;
use IEEE. STD_LOGIC_ARITH. ALL;
use IEEE. STD_LOGIC_UNSIGNED. ALL;

entity FULLADDER_BEHAVIORAL_SOURCE IS
port (A: in STD_LOGIC_VECTOR (2 down to 0);
      O: out STD_LOGIC_VECTOR (1 down to 0);
end FULLADDER_BEHAVIORAL_SOURCE;
architecture Behavioral of FULLADDER_BEHAVIORAL_SOURCE is
begin
process (A)
begin
--for SUM
if (A = "001" or A = "010" or A = "100" or A = "111") then
      O (1) < = '1';
      else
      O (1) < = '0';
```



```
end if;
-- for CARRY
if (A = "011" or A = "101" or A = "110" or A = "111") then
      O(0) < = '1';
      else
      O(0) < = '0';
      end if;
end process;
end Behavioural;
```

Differences between behavioral and structural modeling are:

S.N.	Behavioral Modeling	Structural Modeling
1.	It consists of sequential program statement.	It is set of interconnect component.
2.	It requires truth table for design.	It requires logical diagrammed for design.
3.	It represents behavior.	It represents structure.
4.	It consist gate level abstraction.	It consist RTL abstraction.
5.	It is expressed in a sequential VHDL process.	The view is closest to hardware.

6. Write a VHDL code to simulate 8*1 MUX. [2017 Spring] [2019 Spring]

Answer:

VHDL code to simulate 8*1 MUX

```
library IEEE;
use IEEE. STD_LOGIC_1164.ALL;
      entity mux_8*1 is
      port (d: in STD_LOGIC_VECTOR (0 to 7);
            s: in STD_LOGIC_VECTOR (0 to 2);
            o: out STD_LOGIC);
      end mux _8*1;
architecture mux_arch of mux_8*1 is
begin
      process (d, s)
      begin
            case s is
                  when "000" ⇒ O < = d (0);
                  when "001" ⇒ O < = d (1);
                  when "010" ⇒ O < = d (2);
                  when "011" ⇒ O < = d (3);
```

```
            when "100" ⇒ O < = d (4);
            when "101" ⇒ O < = d (5);
            when "110" ⇒ O < = d (6);
            when "111" ⇒ O < = d (7);
            when others ⇒ O < = '0';
        end case;
    end process;
end mux_arch;
```

7. Write a VHDL program for a 4-bit full adder [2018 Fall] [2019 Fall]

Answer:

```
library IEEE;
use IEEE. STD_LOGIC_1164. ALL;
entity full_adder is
port (
        A: in STD_LOGIC_VECTOR (3 down to 0);
        B: in STD_LOGIC_VECTOR (3 down to 0);
        Cin: in STD_LOGIC;
        S: out STD_LOGIC_VECTOR (3 down to 0);
        Cout: out STD_LOGIC
    );
end full_adder;
architecture Behavioral of full_adder is
component full_adder_vhdl_code
port (
        A: in STD_LOGIC;
        B: in STD_LOGIC;
        Cin: in STD_LOGIC;
        S: out STD_LOGIC;
        Cout: out STD_LOGIC;
end component;
signal C1, C2, C3: STD_LOGIC;
begin
    FA1: full_adder_vhdl_code port map (A (0), B (0), Cin, S (0), C1);
    FA2: full_adder_vhdl_code port map (A (1), B (1), C1, S (1), C2);
    FA3: full_adder_vhdl_code port map (A (2), B (2), C2, S (2), C3);
    FA4: full_adder_vhdl_code port map (A (3), B (3), C3, S(3), Cout);
end Behavioural;
```


8. Write a VHDL program for 4 –to- 1 mux. [2018 spring]

Answer:

```
library IEEE;
use IEEE. STD_LOGIC_1164. ALL;
entity mux_4 to 1 is
port (
        A, B, C, D: in STD_LOGIC;
        So, S1: in STD_LOGIC;
        Z: out STD_LOGIC
    );
    end mux_4 to 1;
    architecture bhv of mux_4 to 1 is
    begin
    process (A, B, C, D, S0, S1) is
    begin
        if (S0 = '0' and S1 = '0') then
            Z < = A;
        elsif (S0 ='1' and S1 = '0') then
            Z < = B;
        elsif (S0 = '0' and S1 = '1') then
            Z < = C;
        else
            Z < = D;
        end if;
    end process;
    end bhv;
```

9. Write a VHDL code 4 : 1 MUX using structural Modeling style. [2020 Fall]

Answer:

```
library IEEE;
use IEEE. STD_LOGIC_1164. ALL;
entity MUX 4_1 is
port (sel0, sel1: in std_logic;
        A, B, C, D: in std_logic;
        Y: out std_logic
    );
    end Mux 4_1;
architecture structural of Mux4_1 is
```

```
component inv
port (pin: in std_logic;
            pout: out std_logic;
      end component;
component and3
port (a0, a1, a2: in std_logic;
aout: out std_logic);
end component
component or4
port (r0, r1, r2, r3,: in std_logic;
            rout: out std_logic);
end component;
signal selbar0, selbar1, t1, t2, t3, t4: std_logic;
begin
INV0: inv port map (sel0, selbar1);
INV1: inv port map (sel1, selbar1);
A1: and3 port map (A, selbar 0, selbar 1, t1);
A2: and3 port map (B, sel0, selbar1, t2);
A3: and3 port map (C, selbar0, sel1, t2);
A4: and3 port map (D, sel0, sel1, t4);
O1: or4 port map (t1, t2, t3, t4, y);
end structural;
```


Chapter 11

EMBEDDED SYSTEM DEVELOPMENT TOOL

OLD EXAM QUESTION SOLUTION [PoU]

1. **Define Debugger, Downloader and Cross-Assembler.**
[2016 Fall] [2017 Spring] [2019 Spring]

Answer:

Debugger

A Debugger or debugging tool is a computer program that used to test and debug other programs. The code to be examined should be running on an instruction set simulator to identify the fault in the code because the software problem cannot be identified when we are running it on the original hardware. The debugger can be used to identify if the program is running correctly, and identify the cause of failure when it fails. The debugger may be a source-level debugger, or a low-level debugger. If it is source-level debugger, the debugger can show the actual position in the original code, when the program crashes. If it is a low-level debugger or a machine-language debugger, it shows that line in the program. Catching run-time errors is not as obvious. Most embedded systems do not have a "screen". Hence we cannot find the run time errors as in general software development.

Downloader

Once a program has been successfully complied, linked, and located it must be moved to the target platform. Downloader will download the binary image to the embedded system. Executable binary image is

transferred and loaded into a memory device on target board. It can be loaded into ROM via a device programmer, which "bums" a chip that is then re-inserted into the embedded system. Your program will then execute when you reset the processor, or apply power to the embedded system.

Cross-Assembler:

A cross-assembler is an assembler that runs on a computer with one type of processor but generates machine code for a different type of processor. *For example*, if we use a PC with the 8086 compatible machine language to generate a machine code for the 8085 processor, we need a cross-assembler program that runs on the PC compatible machine but generates the machine code for 8085 mnemonics. It takes assembly language as input and give machine language language as output.

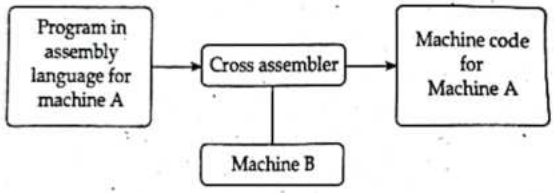


Figure: Cross-assembler

In the above block diagram, we can see that there is an assembler which is running on machine B but converting assembly code of Machine A to machine code, this assembler is Cross-assembler.

Features of Cross-Assembler:

- Cross-assembler is used to convert assembly language into binary machine code.
- Cross-assemblers are also used to develop program which will run on game console and other small electronic system which are not able to run development environment on their own.
- Cross-assembler can be used to give speed development on low powered system.
- C 64 is the best example of cross-assembler.

2. **Write short notes on:**
Cross Copiler and cross assembler
[2016 Spring] [2017 Fall] [2018 Spring] [2019 Fall]

Answer:

A cross compiler runs on one computer platform and produce codes for another computer platform. In other words, a Cross compiler is a compiler that generates executable code for a platform other than one on which the compiler is running. *For example* a complier that running on Linux/x86 box is building a program which will run on a separate Arduino/ARM.

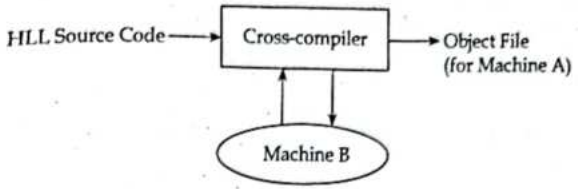


Figure: Cross-compiler

Feature of cross compiler

- Translates program for different hardware/platform/machine other than the platform which it is running.
- It is used to build programs for other system/machine like AVR/ARM.
- It is independent of System/machine and OS
- It can generate raw code.hex
- Keil is a cross compiler.

Cross Assembler: *See in equation number 1.*

3. **Write short notes on:**
Cross compiler and Native compiler. **[2019 Spring]**

Answer:

1. **Native Compiler:**

Native compiler are compilers that generates code for the same Platform on which it runs. It converts high language into computer's native language. *For example*; Turbo C or GCC compiler

2. **Cross Compiler:**

A cross Compiler is a complier that generates executable code for a platform other than one on which the compiler is running. *For example* a compiler that running on Linux/x86 box is building a program which will run or a separate Arduino/ARM.

Difference between Native Compiler and Cross Compiler:

Native Compiler	Cross Compiler
Translates program for same hardware/platform/machine on it is running.	Translate program for different hardware/platform/machine other than the platform which it is running.
It is used to build programs for same system/machine and OS it is installed.	It is used to build programs for other system/machine like AVR/ARM.
It is dependent on system/machine and OS.	It is independent of system/machine and OS.
It can generate executable file like.exe	It can generate raw code.hex
TurboC or GCC is native Compiler.	Keil is a cross compiler.

4. Write short notes on:
Debugger, Emulator and Profiler. [2020 Fall]

Answer:

Debugger:

See in question number 1

Emulator:

Emulator or in-Circuit Emulators (ICE) take the place of (*i.e.*, emulates) target processor. It contains copy of target processor, plus RAM, ROM, and its own embedded software. It allows you to examine the state of processor, while program is running. It uses Remote debugger for human interface and has more capability than target processor. It support software and hardware breakpoints (stop execution on memory and I/O read/write, interrupts) "Stop on write to variable num". It supports real-time tracing stores information about each processor cycle that is executed and allows you to see what order things happened.

Disadvantage: It is expensive.

REFERENCES

1. Vahid, F. and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction.*New Jersey: John Wiley & Sons, Inc., 2002
2. *Embedded systems - shape the world.* (n.d.). Retrieved 2021, from https://users.ece.utexas.edu/~valvano/Volume1/E-Book/
3. *Embedded system design: A unified hardware/software introduction.* Embedded System Design: A Unified Hardware/Software Introduction. (n.d.). Retrieved 2021, from http://esd.cs.ucr.edu/
4. Singh, A. (2008). *Embedded system.* New Delhi, India: Maxford Books.
5. Vahid, F., & Givargis, T. (2010). *Embedded system design.* New Jersey: John Wiley & Sons.
6. *Embedded Systems Task Scheduling Algorithms & Deterministic Behavior Retrieved* 2019, from https://www.just.edu.jo/~tawalbeh/cpe746/slides/Scheduling_Algorithms.pdf
7. *Handwritten notes of Embedded System.* (2019). Retrieved 2020, from https://lecturenotes.in/s/225-embedded-system/notes
8. The Embedded Software Development Process | SoftwareTestPro. (2021). Retrieved 2021, from https://www.softwaretestpro.com/the-embedded-software-development-process/
9. Gautam. K. & Luitel S.R., *Insights on Embedded System.* 1st Edition. System Inspection (Nepal), 2021 A.D.
10. *Embedded Systems Development Life cycle Process.* (2019). Retrieved 2021, from https://www.watelectronics.com/embedded-systems-development-process/
11. *Preemptive and Non-Preemptive Scheduling.* (2019). Retrieved 2022, from https://www.tutorialspoint.com/preemptive-and-non-preemptive-scheduling
12. *Real-Time Operating Systems: Introduction and Process Management.* (2019). Retrieved, 2019 https://nptel.ac.in/content/storage2/courses/108105063/pdf/L37(SM)%20(IA&C)%20((EE)NPTEL).pdf
13. Perry, D. (2002). *Very High-Speed Integrated Circuit Hardware Description Language (VHDL).* New York: McGraw-Hill.

14. *VLSI Design - VHDL Introduction. (2018).*Retrieved 2021, from http://shorturl.at/cmrM8

15. *IOE Note | Embedded System (2020).* Retrieved 17 May 2020, from https://ioesolutions.esign.com.np/notes/text-notes-show/Interfacing

16. Forrai, A. (2013). *Embedded control system design.* Berlin: Springer.