# INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

## COURSE CONTENT

| ANALOG AND DIGITAL CIRUITS LABORATORY | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **IV Semester: EEE** | | | | | | | | |
| **Course Code** | **Category** | **Hours / Week** | | | **Credits** | **Maximum Marks** | | |
| | | **L** | **T** | **P** | **C** | **CIA** | **SEE** | **Total** |
| **AECD17** | **Core** | 0 | 0 | 2 | 1 | 40 | 60 | 100 |
| **Contact Classes: Nil** | **Tutorial Classes: Nil** | **Practical Classes: 45** | | | | **Total Classes: 45** | | |
| **Prerequisite: There are no prerequisites to take this course.** | | | | | | | | |

### I. COURSE OVERVIEW:

The objective of this laboratory course is to meet the requirements of practical work meant for com- ponents basics, analysis and design and provides hands-on experience by examining the characteris- tics of various semiconductor devices and measuring instruments. This lab covers the analysis of the characteristics of semiconductor devices and functionality of the digital circuits to use as elementary blocks in analog and digital circuit applications. Students will proficiency with the capability to use simulation tools for performing various analysis of semiconductor devices, combinational and sequential circuit applications.

### II. COURSE OBJECTIVES:

**The students will try to learn:**

I The characteristics and applications of diodes.
II The characteristics of transistor in different configurations.
III The function and applications of gates.
IV The different combinational circuits.

### III. COURSE OUTCOMES:

**After successful completion of the course, students should be able to:**

CO1 Apply the PN junction characteristics for the diode applications such as half wave and full wave rectifier.
CO2 Apply the volt-ampere characteristics of PN junction diode, Zener diode forfinding cut-in voltage, static and dynamic resistance.
CO3 Analyze the input and output characteristicsof transistor configurations for determining the input-output resistances.

determining the input-output resistances.
CO4 Identify the functionality of Boolean expressions using gates such as and,or,not, and, nor, nor and xnor.
CO5 Build combinational circuits such as adder, subtract or,multiplexers and comparators realization using low levelelementary blocks.
CO6 Construct shift registers using the functionality of the flip flops.

## DO's

1. Once the operation is completed pull the plug itself rather chord attached to it.
2. Operate the equipment on supply; see that hands are dry, if that is not possible, hide the hand in the pockets.
3. If a person comes in contact with current unexpectedly don't touch the person with hands but immediately use any insulator material and shut down the power (like leather belts, wood and plastic bars etc).
4. If water is nozzles on the equipment, immediately shutdown the power using circuit breaker or pull out the plug.
5. Use the connecting wires of good continuity, short circuit of connecting wire leads damage of circuit parameters.

## DON'Ts

1. Do not wear loose clothing and do not hold any conducting materials in contact with skin when the power is on.
2. Do not pull out the connections until unless all the currents are dead.
3. Do not over load the circuit by plugging in too many appliances.
4. If you are mentally and physically stressed don't operate the power equipment.
5. Never operate the equipment under wet conditions.
6. Do not inter connect two or more wires, take appropriate length of wire.

## SAFETY NORMS

1. The lab must be equipped with fire extinguisher.
2. See that the connections are made tight.
3. Use single plug for each equipment.
4. Cover the body completely to avoid arc effect.
5. To change the connections during the experiment, switch off the supply and carry on.
6. Used equipment may get heated, so take care handling the equipment after it is used.
7. Do the wiring, all setups and check the circuit connections before the supply is on

I. COURSE CONTE

II. NT:

# EXERCISES FOR ANALOG AND DIGITAL CIRCUITS LABORATORY

**Note:** Students are encouraged to bring their own laptops for laboratory practice session

## Introduction

This laboratory course enables students to get practical experience on analog and digital electronics circuits. Analog components and circuits like p-n junction diode, OP-AMP etc. and digital electronics exercises like logic gates, adders and sub tractors, flip flops, counters etc. Simulation packages preferred are vivado software, Multisim etc.

Analog Electronics is one of the fundamental courses found in all Electrical Engineering and most science programs. Analog circuit's process signals with continuous variation of voltage. The different Components that are normally used in Analog Electronics are:

1. Bi polar Junction Transistors
2. MOSFET's
3. OP-AMP

## 1. Getting Started Exercises

The semi-conductor diode is created by simply joining an n-type and a p-type material together nothing more just the joining of one material with a majority carrier of electrons to one with a majority carrier of holes.

### 1.1 Determine the V-I characteristics of forward Bias and Reverse Bias P-N junction diode as shown in fig 1.1.1 and 1. 1. 2 and draw the V-I Characteristics.

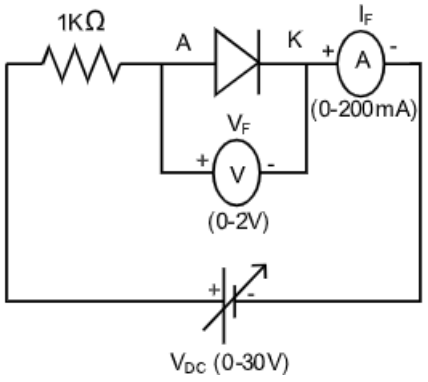


**Fig 1.1.1. Forward Bias P-N Junction diode**

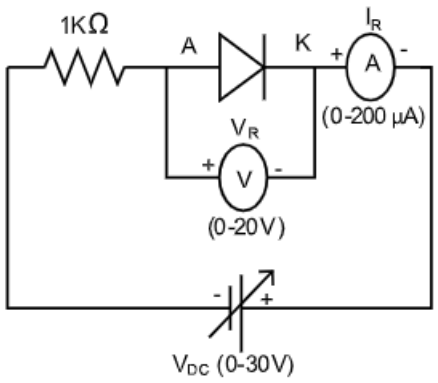


**Fig 1.1.2. Reverse Bias P-N Junction diode**

**1.2 A diode connected to an external resistance and an e.m.f. assuming that the barrier potential developed in diode is 0.5V as shown in the fig1.3. Obtain the value of current in the circuit in milli- ampere as shown in the fig.1.2.**

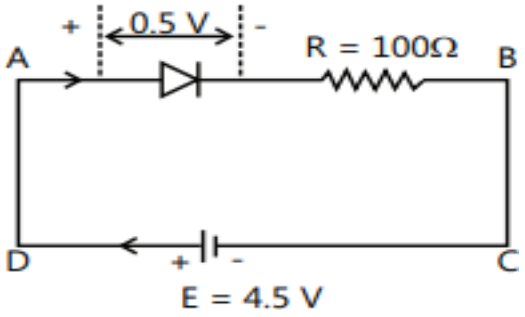


**Fig 1.2. Barrier potential in diode**

**1.3 A diode has a constant voltage drop of 0.5 V at all current and a maximum power rating of 100 mill watts. What should be the value of the resistor R, connected in series with the diode for obtaining maximum current?**

**Try**

1. Plot the V-I Characteristics of germanium diode and find the cut in voltage.
2. Using ua 741 Opamp, design a 1 kHz Relaxation Oscillator with 50% duty cycle. And simulate the same.

## 2. Exercises on Transistor Common Emitter Characteristics

Bipolar junction transistor (BJT) is a 3 terminal (emitter, base, collector) semiconductor device. There are two types of transistors namely NPN and PNP. It consists of two P-N junctions namely emitter junction and collector junction. In Common Emitter configuration the input is applied between base and emitter and the output is taken from collector and emitter. Here emitter is common to both input and output and hence the name common emitter configuration.

## 2.1Verification of input and output characteristics of CE Configuration using hardware shown in fig 2.1 and draw the Waveforms.

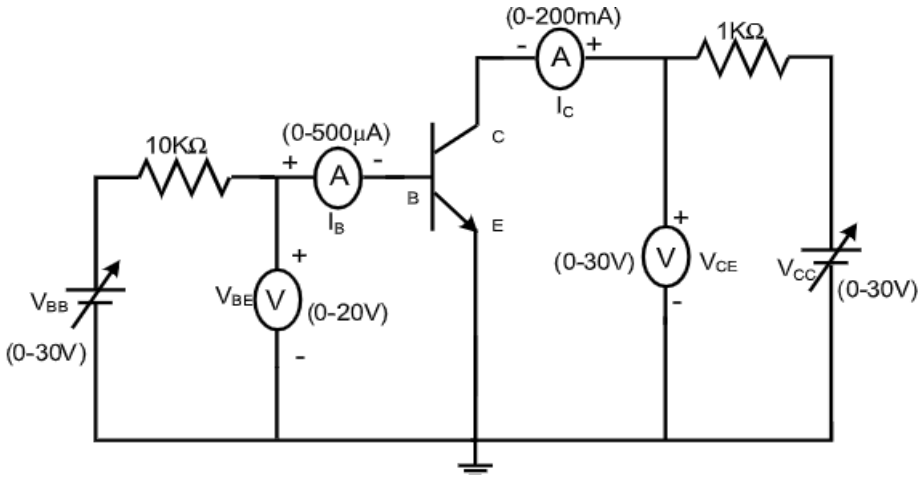


**Fig 2.1. CE configuration**

**Try**

1. Determine the Gain and Bandwidth of CE (Common Emitter) amplifier using hardware.
2. Obtain Drain characteristics and Transfer characteristics of FET.

## 3. Exercises on Transistor Common Base Characteristics

Bipolar junction transistor (BJT) is a 3 terminal (emitter, base, collector) semiconductor device. There are two types of transistors namely NPN and PNP. It consists of two P-N junctions namely emitter junction and collector junction. In Common base configuration the input is applied between base and emitter and the output is taken from collector and base. Here base is common to both input and output and hence the name common base configuration.

## 3.1 Verification of input and output characteristics of CB Configuration using hardware shown in Fig 3.1 and draw the Waveforms

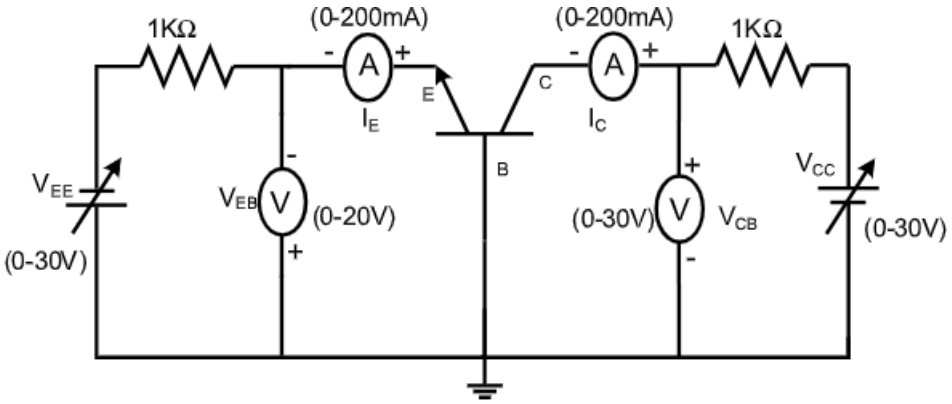


**Fig 3.1. Circuit Diagram for transistor as common base configuration**

**Try**

1. Determine the Gain and Bandwidth of CB (Common Base) amplifier using hardware.
2. Obtain Drain characteristics and Transfer characteristics of FET.

## 4. Exercises on Half wave rectifier and Full wave rectifier

Half-wave rectifier1is an electronic circuit that converts only one-half of the AC cycle into pulsating DC. It uses only half of the AC cycle for the conversion process. Full-wave rectifier is an electronic circuit that converts the entire cycle of AC into pulsating DC. It has two diodes, and its output uses both halves of the AC signal. During the period that one diode blocks the current flow, the other diode conducts and allows the current. Both half-wave and full-wave rectifiers have their merits and demerits.

### 4.1 Verification of Half wave rectifier without filters using hardware shown in fig 4.1 and draw the waveforms.

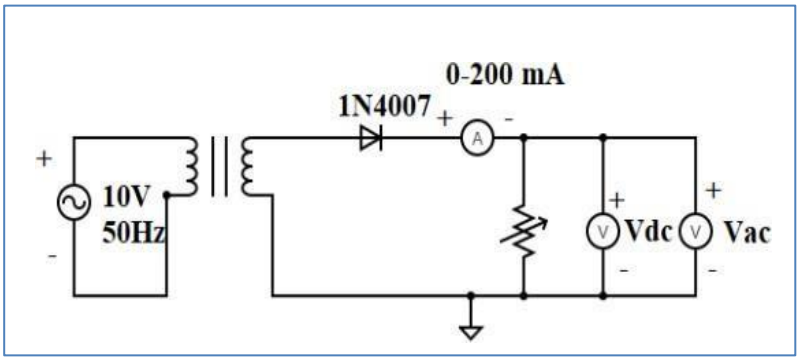


**Fig 4.1. Half wave rectifier without filter**

### 4.2 Verification of Half wave rectifier with filters using hardware shown in fig 4.2 and draw the waveforms.

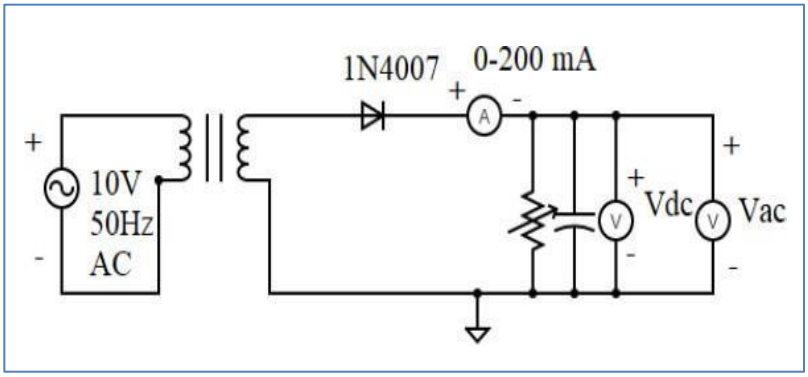


**Fig 4.2. Half wave rectifier with filter**

## 4.3 Verification of Full wave rectifier without filters using hardware shown in fig 4.3 and draw the waveforms.

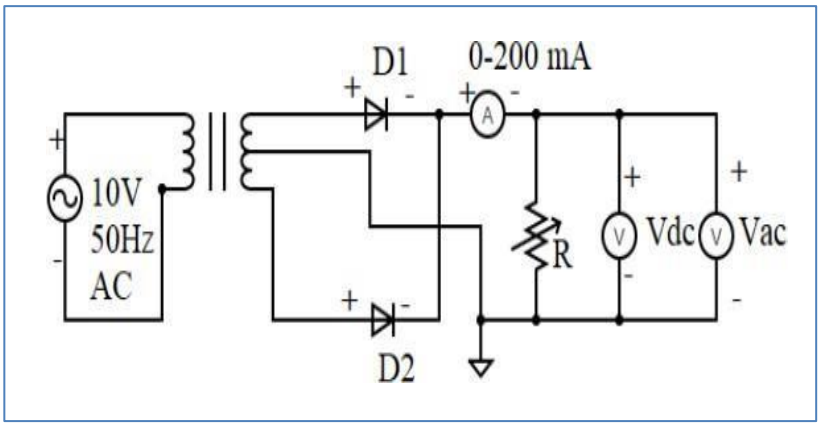


**Fig 4.3 Full wave rectifier without filter**

## 4.4 Verification of Full wave rectifier with filters using hardware shown in fig 4.4 and draw the waveforms.

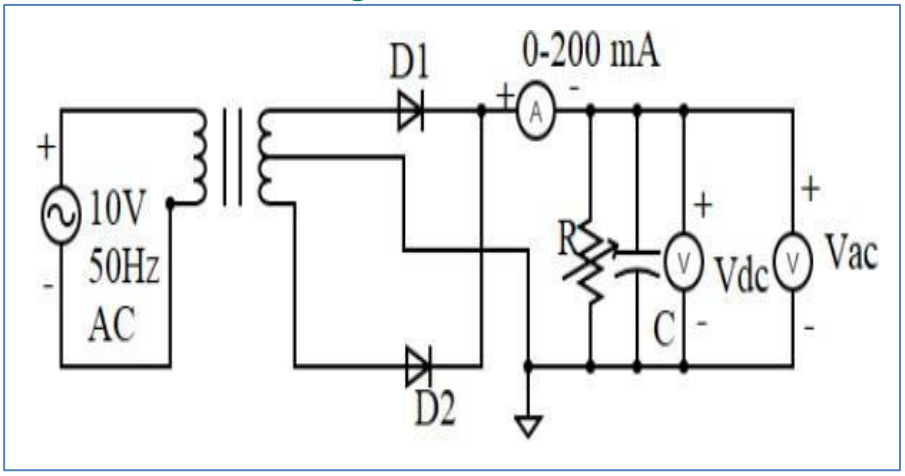


**Fig 4.4 Full wave rectifier with filter**

**Try**

1. Design and verify bridge rectifier with and without filter.
2. Design and verify precision rectifier.

## 5. Exercises on DRAIN AND TRANSFER CHARACTERISCS FET

The functioning of Junction Field Effect Transistor depends upon the flow of majority carriers (electrons or holes) only. Basically, JFETs consist of an n type or p-type silicon bar containing p-n junctions at the sides.

To study the drain and transfer characteristics of FET and find the drain resistance, trans-conductance and amplification factor for the Figure 5.1

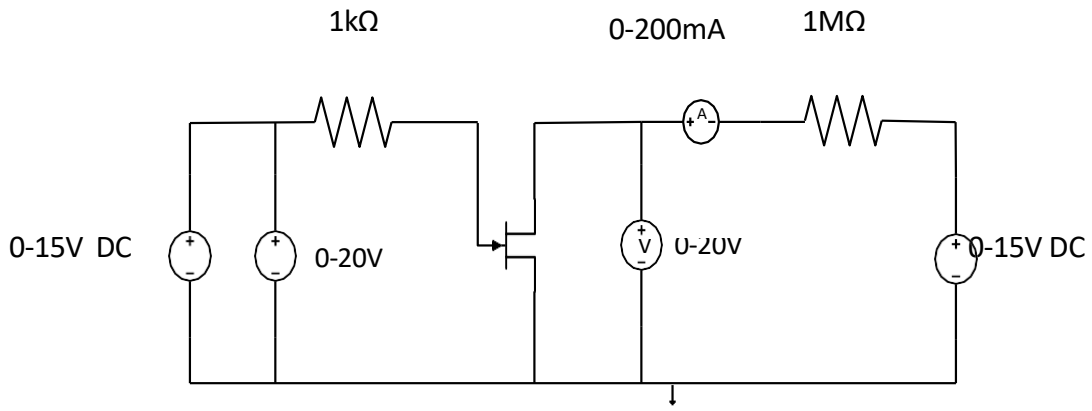


**Figure 5.1: Field effect transistor**

## Try

1. Plot the drain and transfer characteristics of P-channel JFET BFW11/10/BF245A with RL= 50 Ω.
2. A JFET has the following parameters: IDSS = 32 mA; VGS (off) = – 8V; VGS = – 4.5V. Find the value of drain current.
3. An N-channel JFET having Vp=-4V and VDSS = 10mA is used in the circuit of Figure 5.1.1. The parameter values are V DD = 18V, Rs=2k Ω, R1=450k Ω and R2=90kΩ. Determine ID and VDS.

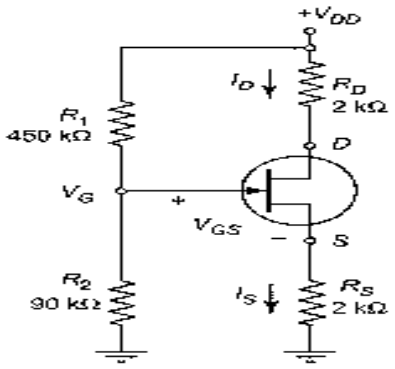


**Figure 5.1.1**

## 6. Exercises on RC Phase Shift Oscillator

RC phase shift oscillator has a CE amplifier followed by three sections of RC phase shift feedback networks. The output of the last stage is return to the input of the amplifier. The values of R and C are chosen such that the phase shift of each RC section is 600. Thus, the RC ladder network produces a total phase shift of 1800 between its input and output voltage for the given frequencies. Since CE amplifier produces 1800 phase shift the total phase shift from the base of the transistor around the circuit and back to the transistor will be exactly 3600. To explore and understand oscillator design and need to perform the following practical exercises.

### 6.1 Design an RC phase oscillator with Resistance R1=Rc=10KΩ, RE=1 KΩ, R=10 KΩ, CE=CC=1μF and C=10μF for the circuit shown in figure.6.1 and determine the frequency of oscillation of the circuit.

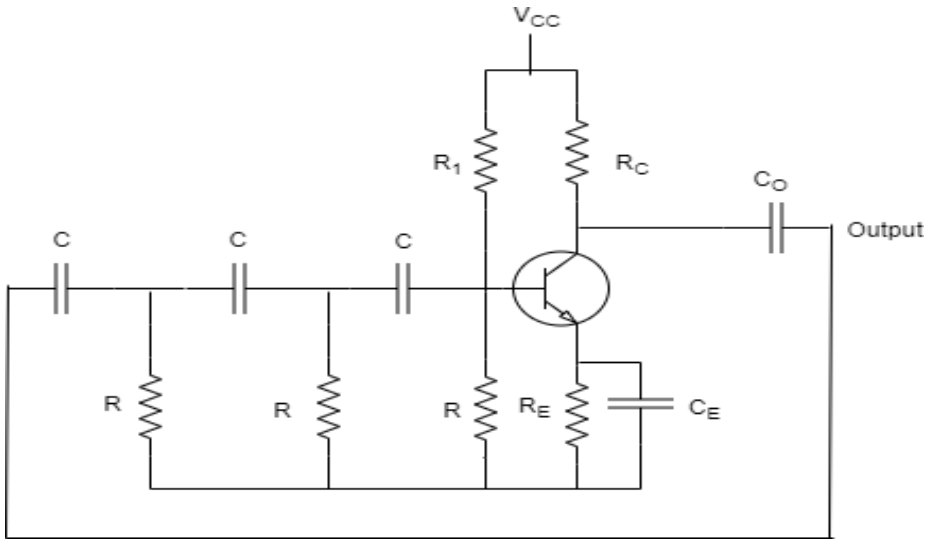


**Figure 6.1: RC Phase shift oscillator**

**Try**

1. Design an RC phase shift oscillator to produce an oscillation frequency of 2 K Hz and compare the theoretical and practical frequencies.
2. Design an RC phase shift oscillator with a target frequency of 1.5 K Hz and an amplitude of 5 Volts peak-to-peak.
3. Determine the individual and total phase shift introduced by each stage for a given RC phase shift oscillator for a frequency of 3.5 K Hz.

## 7. Inverting and Non-Inverting Operational Amplifiers

Inverting amplifiers are used in a number of applications like phase shifter, integration, signal balancing, mixer circuits, etc. Non-inverting amplifiers are used in circuits where high input impedance is required. They are used as voltage followers, isolation of cascaded circuits, to perform mathematical simulations.

### 7.1 Design an inverting amplifier circuit with OP AMP 741C for gain of 10.

Plot the waveforms, observe the phase reversal, measure the gain for the circuit shown in Figure.7.1.1.

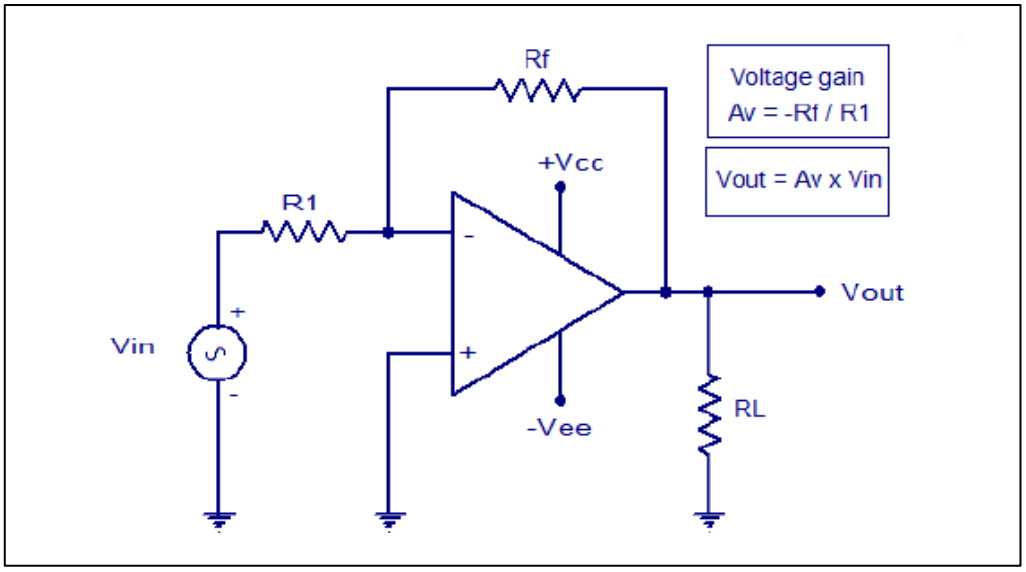


**Figure 7.1.1: Inverting Amplifier**

### 7.2 Design of Non-Inverting amplifier with OP AMP741C for gain of 11.

Plot the waveforms, observe the phase reversal, measure the gain for the circuit shown in Figure.7.2.1.

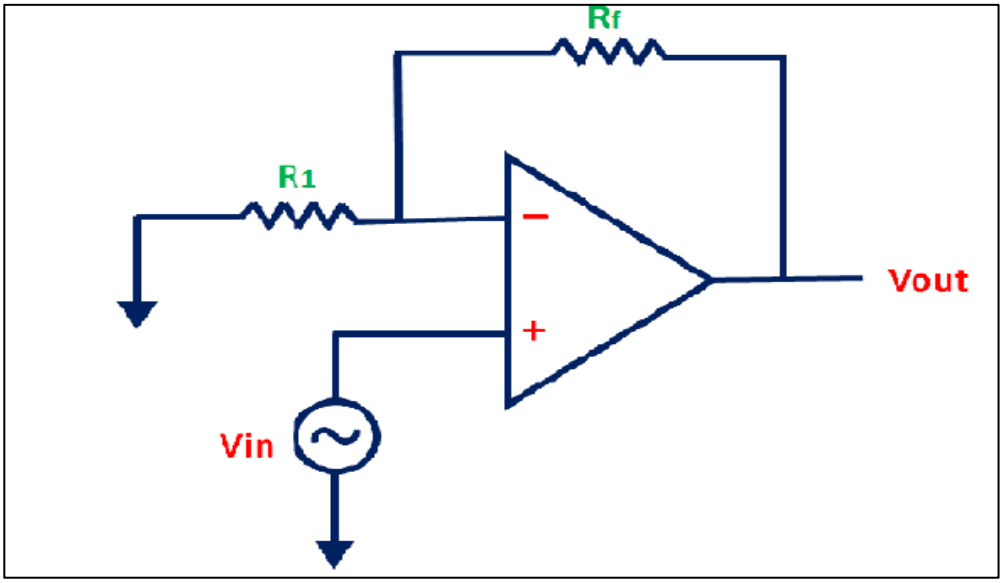
Rf
R1
−
+
Vout
Vin

**Figure 7.2.1: Inverting Amplifier**

### 7.3. What is the output of the summing amplifier in Figure 7.3.1, with the given DC input voltages? Compare the values obtained from Simulation and Practical values.

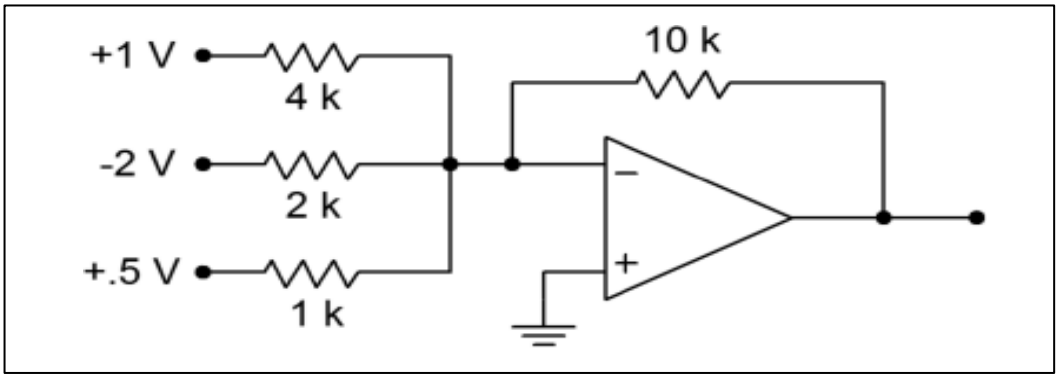


**Figure.7.3.1**

Hint: The easy way to approach this is to just treat the circuit as three inverting voltage amplifiers, and then add the results to get the final output.

**Try**

1. Design a simple difference amplifier with an input impedance of 10 k per leg, and a voltage gain of 26 Db.
2. Design Adder/ Subtractor Circuits.

Hint: If inverting and non- inverting summing amplifiers are combined using the differential amplifier topology, an adder/subtractor results. Normally, all resistors in an adder/subtractor are the same value.

## 8. Integrators and Differentiators

The integrator circuit is mostly used in analog computers, analog-to-digital converters and wave-shaping circuits. A common wave-shaping use is as a charge amplifier and they are usually constructed using an operational amplifier though they can use high gain discrete transistor configurations. Differentiating amplifiers are most commonly designed to operate on triangular and rectangular signals. Differentiators also find application as wave shaping circuits, to detect high frequency components in the input signal.

### 8.1.1. Design of integrator circuit for periodic signal with a frequency of 5 kHz

Draw the input and output waveforms for different time constants for the basic integrator circuit shown in Figure.8.1.1.

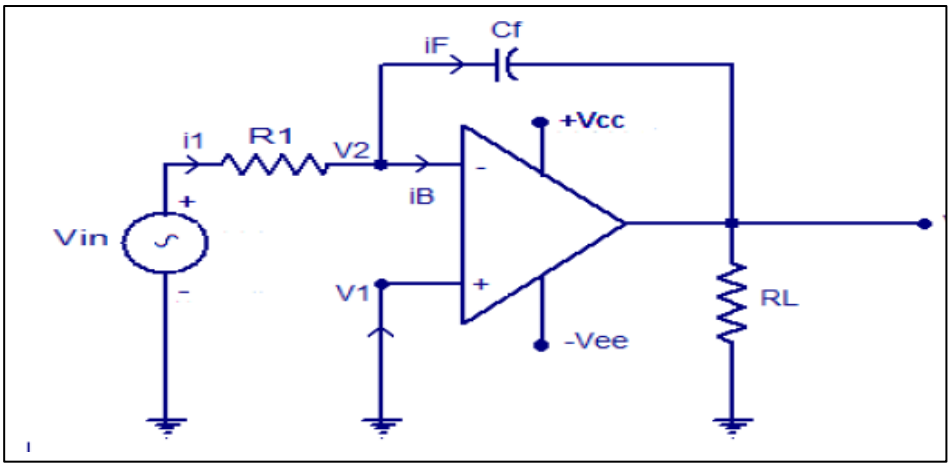


Figure 8.1.1: Basic Integrator

### 8.1.2 Design lossy integrator so that the peak gain is 20dB and the gain is 3dB down from its peak when W= 10,000 rad/sec. use a capacitance of 0.01µF in hardware and multisim.

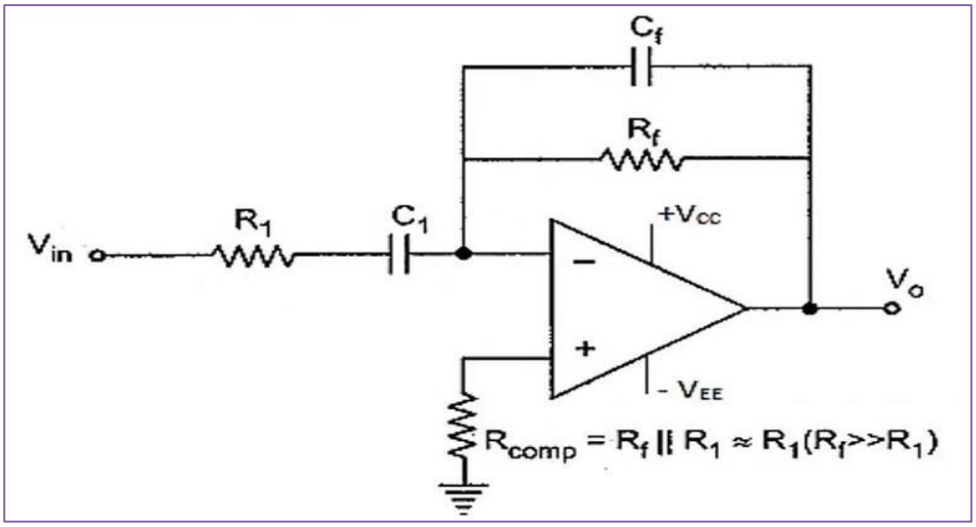


Figure 8.1.2: Practical Integrator

### 8.1.3. Design Summing integrator to give output as per the following equation using Multisim.

$$v_o(t) = -\frac{1}{C_f}\int_0^t \left(\frac{v_1(t)}{R_1} + \frac{v_2(t)}{R_2} + \frac{v_3(t)}{R_3}\right) dt + v_o(0)$$

### 8.2.1. Design of Differentiator circuit for periodic signal with a frequency of 5 kHz

Draw the input and output waveforms for different time constants for the basic Differentiator circuit shown in Figure.8.2.1.

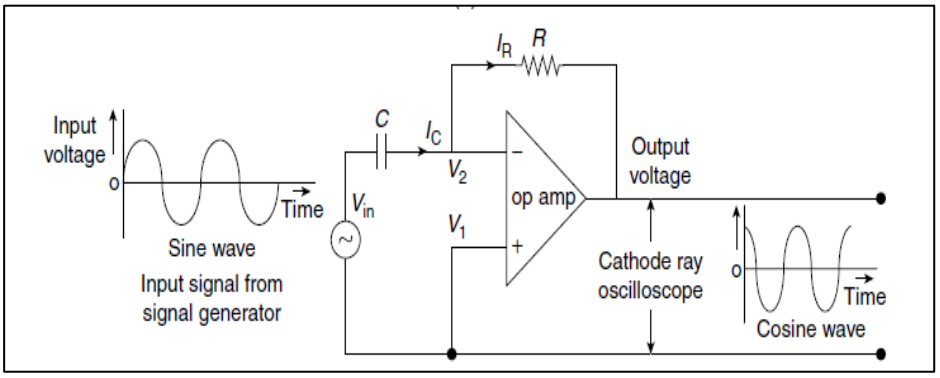


Figure 8.2.1: Basic Differentiator

**8.2.2. Design a differentiator using op-amp to differentiate an input signal with fmax = 200 Hz. Also draw the output waveforms for a sine-wave and a square-wave input of 1V peak at 200 Hz. Compare the values obtained from Simulation and Practical values.**

**8.2.3. Construct a differentiator for a given differential input signal that varies in frequency from 100 Hz to about 5 KHz with a sine wave of 10V peak at 2k Hz is applied to this differentiator. Also draw the required waveforms.**

**Try**

1. The circuit that integrates the input signal twice is called double integrator. The design of double integrator requires two reactive portions for obtaining double integration. Design double integrator in mutisim /pspice to produce output as per the equation as shown below.
2. Elaborate practical integrator circuit and state the advantages of practical integrator over ideal integrator. Using simulation with multisim determine R1 and Rf for a given lossy integrator for a peak gain of 30dB and the gain is 3dB down from its peak when W = 5000 rad/sec with a capacitance of 0.01micro farads.

$$v_o(t) = -\frac{4}{(RC)^2}\iint v_i(t)dt$$

3. Sketch the output waveform of a differentiator for a sinusoidal signal of 3V peak at 250Hz. Compare the values obtained from Simulation and Practical values.
4. Determine the useful range for differentiation in the circuit of Figure 8.2.3. Also determine the output voltage if the input signal is a 2 V peak sine wave at 3 kHz.

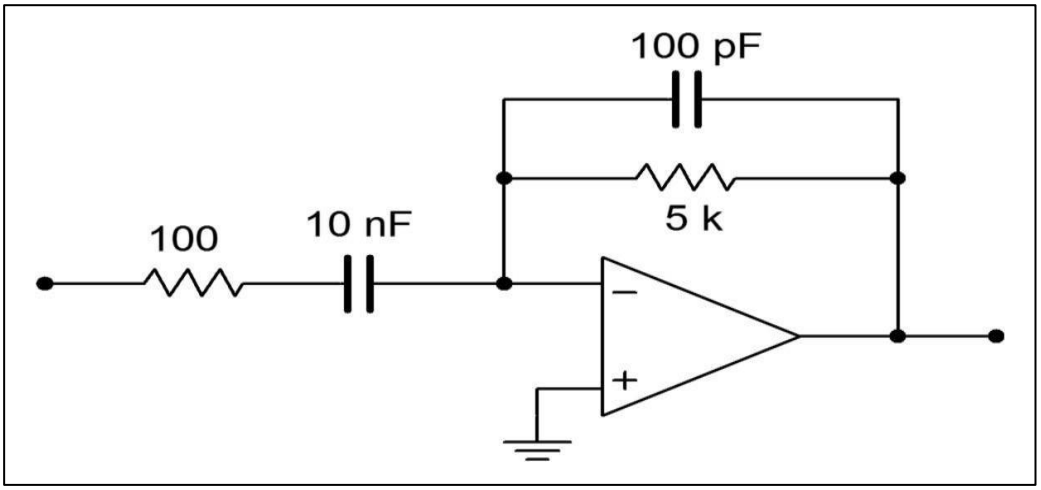


Figure 8.2.3

5. Given the circuit of Figure 8.2.4 carryout the simulation and analyse the output waveforms if the input is a 100 Hz, 1 V peak triangle wave.

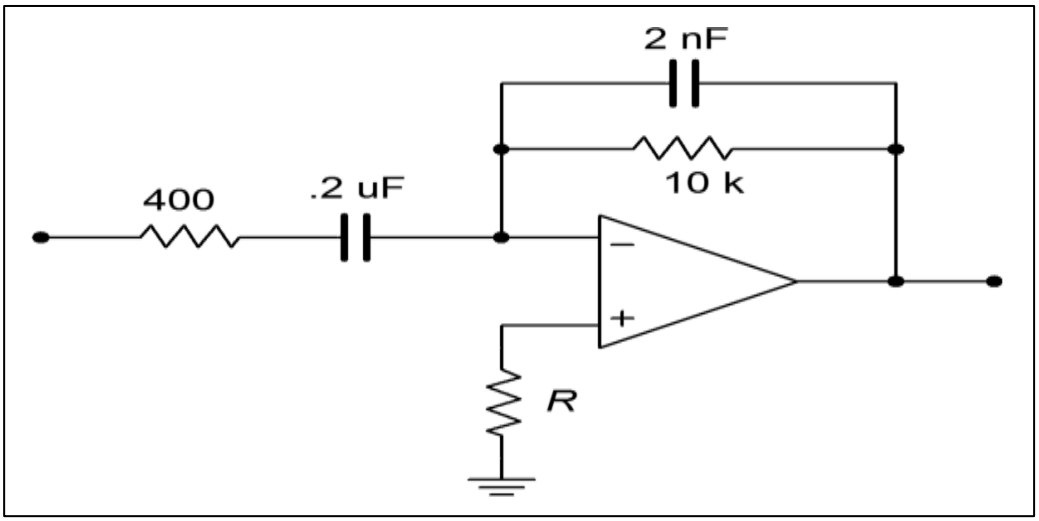


Figure 8.2.4

## 9.Exercises on Basic Logic Gates

AND, OR and NOT gates are basic gates. NAND and NOR are universal gates. Basically logic gates are electronic circuits because they are made up of number of electronic devices and components. Inputs and outputs of logic gates can occur only in two levels. These two levels are term HIGH and LOW, or TRUE and FALSE, or ON AND off, OR SIMPLY 1 AND 0. A table which lists all possible combinations of input variables and the corresponding outputs is called a „truth table". It shows how the logic circuit"s output responds to various combinations of logic levels at the inputs.

### 9.1.1 Basic Gates

9.1.1.1. Install Xilinx vivado on your machine.

9.1.1.2. Write a VHDL program using vivado simulator for:

9.1.1.3. Verifying the functionality of design under test (DUT) by writing test bench to pass the stimulus

9.1.1.4. Synthesize the register transfer logic (RTL) using Xilinx XST synthesis tool

9.1.1.5. Elaborate the design and generate bit file to dump RTL code into the zynq series and Z-board FPGA

9.1.1.6. Verify the functionality of the design under test (DUT) on FPGA board

### 9.1.2. AND-OR-INVERT AND OR-AND-INVERT LOGIC

Write VHDL code to implement the function expressed by the following logic equation

$$Y = \bar{a}_0\bar{b}_0 + \bar{a}_1\bar{b}_1 + \bar{a}_2\bar{b}_2$$

Use only simple signal assignment statements in your VHDL data flow model.

**Hints**

Use and, or and compliment operators for implementation of the logic.

```
Input a0, a1, a2, a3, b0, b1, b2, b3;
Output y;


/**   Data flow model for AOI logic */
entity AOI port (
         a0, a1, a2, a3, b0, b1, b2, b3: in std_logic;
         y: out std_logic);
end AOI;

/**   architecture body */
architecture arch_AOI of AOI is

begin
    Y = not ((a0 & b0) | (a1 & b1) | (a2 & b2) | (a3 & b3));
    . . .

End arch_AOI;

//Write the test bench for providing the stumulus
```

```
entity tb_AOI port
end tb_AOI;

architecture arch_AOI of AOI is
   signal a0, a1, a2, a3, b0, b1, b2, b3: std_logic := '0';
   signal y: std_logic;

   component AOI port (
          a0, a1, a2, a3, b0, b1, b2, b3: in std_logic;
          y: out std_logic);
   end component;

begin
   DUT: AOI port map (a0, a1, a2, a3, b0, b1, b2, b3, y);

   a0 = '0'; b0 = '0';
   a1 = '0'; b1 = '0';
   a2 = '0'; b2 = '0';
   a3 = '0'; b3 = '0';
   wait for 10 ns;

   a0 = '0'; b0 = '1';
   a1 = '1'; b1 = '0';
   a2 = '0'; b2 = '1';
   a3 = '1'; b3 = '0';
   wait for 10 ns;

   . . .
   . . .

   a0 = '1'; b0 = '1';
   a1 = '1'; b1 = '1';
   a2 = '1'; b2 = '1';
   a3 = '1'; b3 = '1';
   wait for 10 ns;

End architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1. Develop a model for a general or- and-invert gate, with two std_logic_vector input ports a and b and a standard-logic output port y.

### 9.1.2.1. Product of Sum Boolean expression

Write a program to perform product of sum which evaluates the given Boolean expressions and write the test-bench to verify the functionality with all possible combinations of the input.

**Hints**

```
/**
 * Consider the POS expression for F = π(0, 1,2,4,8,9,10,12,13)   */
Input a, b, c, d;
Output F;


Minimize the logic for the given F          // POS logic
Implement the logic using basic gates


/**   Declare the port signals */
entity POS port (
         a, b, c, d: in std_logic;
         f: out std_logic);
end POS;

/**   architecture body  */
architecture arch_POS of POS is

Begin
    Y = F(a, b, c, d);
    . . .

End arch_POS;

//Write the test bench for providing the stumulus

entity tb_POS port
end tb_POS;

architecture arch_POS of POS is
   signal a, b, c, d: std_logic := '0';
   signal f: std_logic;

   component POS port (
         a, b, c, d: in std_logic;
         f: out std_logic);
   end component;

begin
   DUT: POS port map (a, b, c, d, f);

   a = '0'; b ='0'; c = '0'; d ='0';
   wait 10 ns;

   a = '0'; b ='0'; c = '0'; d ='1';
   wait 10 ns;

   . . .
   . . .
```

```
   a = '1'; b ='1'; c = '1'; d ='1';
   wait 10 ns;

End architecture

Provide the stimulus for all 16 possible combinations starting from 0000 to 1111

Simulate the DUT with the given stimulus

Verify the output using waveforms

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1. Modify the POS expression including the don't care cases F = π (0, 1,2,4,9,10,12,13) + d (3, 7, 11)

### 9.1.2.2. Sum of Product Boolean expression

Write a program to perform sum of product which evaluates the given Boolean expressions and write the test-bench to verify the functionality with all possible combinations of the input.

**Hints**

```
/**
 * Consider the POS expression for F = Σ (0, 1,2,4,8,9,10,12,13)   */
Input a, b, c, d;
Output F;

Minimize the logic for the given F           // POS logic
Implement the logic using basic gates

/**   Declare the port signals */
entity SOP port (
          a, b, c, d: in std_logic;
          f: out std_logic);
end SOP;
/**   architecture body */
architecture arch_SOP of SOP is

Begin
    Y = F (a, b, c, d);
    . . .

End arch_SOP;

//Write the test bench for providing the stumulus

entity tb_SOP port
end tb_SOP;

architecture arch_SOP of SOP is
   signal a, b, c, d: std_logic := '0';
   signal f: std_logic;

   component SOP port (
```

```
          a, b, c, d: in std_logic;
          f: out std_logic);
   end component;

begin
   DUT: SOP port map (a, b, c, d, f);

   a = ‘0’; b =’0’; c = ‘0’; d =’0’;
   wait 10 ns;

   a = ‘0’; b =’0’; c = ‘0’; d =’1’;
   wait 10 ns;

   . . .
   . . .

   a = ‘1’; b =’1’; c = ‘1’; d =’1’;
   wait 10 ns;

End architecture

Simulate the DUT with the given stimulus

Verify the output using waveforms

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1. Modify the POS expression including the don’t care conditions F = Σ (0, 1,2,4,9,10,12,13) + d (3, 7, 11)

### 9.1.2.3. Code Conversions

To familiarize students with code converters. The student should also become familiar with gray to binary conversion, binary to gray conversion.
Given the sequence of three-bit Gray code as (000, 001, 011, 010, 110, 111, 101, 100)

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.
3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.

**Hints**

```
/**   Declare the port signals */
entity gray2binary port (
         gray_code: in std_logic_vector(3 downto 0);
         binary_code: out std_logic_vector(3 downto 0));
end gray2binary;

/**   architecturebody  */
architecture arch_gray2binary of gray2binary is

Begin
    binary_code[3] = gray_code[3];
    binary_code[2] = gray_code[3] xor gray_code[2];
    . . .

End arch_gray2binary;

//Write the test bench for providing the stumulus

entity tb_gray2binary port
end tb_gray2binary;

architecture arch_gray2binary of gray2binary is
   signal gray_code: std_logic_vector(3 downto 0) := “0000”;
   signal binary_code: std_logic_vector(3 downto 0);

   component gray2binary port (
         gray_code: in std_logic_vector(3 downto 0);
         binary_code: out std_logic_vector(3 downto 0));
   end component;

begin
   DUT: gray2binary port map (gray_code, binary_code);

   Process
   begin
        gray_code = gray_code + 1;
        Wait for 10 ns;

   End process;

End architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1. Design and implement the binary to gray code conversion
2. Design and implement the binary to excess 3-code conversion
3. Design and implement the excess 3-code to binary conversion

## 9.1.3. Introduction

NAND gate is universal gate. It can perform all the basic logic function. NAND means NOT AND that is, AND output is NOTed.so NAND gate is combination of an AND gate and a NOT gate. The output is logic 0 level, only when each of its inputs assumes a logic 1 level. For any other combination of inputs, the output is logic 1 level. NAND gate is equivalent to a bubbled OR gate.

NOR gate is universal gate. It can perform all the basic logic function. NOR means NOT OR that is, OR output is NOTed.so NOR gate is combination of an OR gate and a NOT gate. The output is logic 1 level, only when each of its inputs assumes a logic 0 level. For any other combination of inputs, the output is logic 0 level. NOR gate is equivalent to a bubbled AND gate.

### 9.1.3.1. Basic gates realization using NAND gate

Realize the inverter gate logic using NAND gate.
NAND gate is actually a combination of two logic gates i.e. AND gate followed by NOT gate. So its output is complement of the output of an AND gate. This gate can have minimum two inputs. By using only NAND gates, we can realize all logic functions: AND, OR, NOT, Ex-OR, Ex-NOR, NOR. So this gate is also called as universal gate.

**Hint**

```
/**   Declare the port signals */
entity inv_nand port (
         i: in std_logic;
         y: out std_logic);
end inv_nand;

/**   architecture body  */
architecture arch_inv_nand of inv_nand is

component nand_gate port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: nand_gate port map (i, i, y);
    . . .

End arch_inv_gate;

//Write the test bench for providing the stumulus

entity tb_inv_nand port
end tb_inv_nand;

architecture arch_inv_nand of inv_nand is
   signal i: std_logic := '0';
   signal y: std_logic;

   component inv_nand port (
         i: in std_logic;
         y: out std_logic);
   end component;

begin
```

```
   DUT: inv_nand port map (i, y);

   i =’0’ after 10 ns;
   i =’1’ after 10 ns;

   . . .
   . . .

end architecture
```

**Try**

1. Realize the AND gate logic using NAND gate
2. Realize the OR gate logic using NAND gate

### 9.1.3.2. Gate realization using NOR

Realize the inverter gate logic using NOR gate.

**Hint**

```
/**   Gate level model for */
entity inv_nor port (
         i: in std_logic;
         y: out std_logic);
end inv_nor;

/**   architecture body  */
architecture arch_inv_nor of inv_nand is

component nor_gate port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: nor_gate port map (i, i, y);
    . . .

End arch_inv_nor;

//Write the test bench for providing the stumulus

entity tb_inv_nor port
end tb_inv_nor;

architecture arch_inv_nor of inv_nor is
   signal i: std_logic := ‘0’;
   signal y: std_logic;

   component inv_nor port (
         i: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: inv_nor port map (i, y);

   i =’0’ after 10 ns;
```

```
   i =’1’ after 10 ns;

   . . .
   . . .

End architecture
```

**Try**

1. Realize the AND gate logic using NOR gate
2. Realize the OR gate logic using NOR gate

### 9.1.3.3 OR gate realization using minimum number of NAND gates

Realize XOR gate using minimum number of NAND gates.

**Hint**

```
/**   Declare the port signals */
entity xor_nand port (
         a, b: in std_logic;
         y: out std_logic);
end xor_nand;

/**   architecture body  */
architecture arch_xor_nand of xor_nand is

component nand_gate port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: nand_gate port map (a, b, y);
    . . .

End arch_xor_nand;

//Write the test bench for providing the stumulus

entity tb_xor_nand port
end tb_xor_nand;

architecture arch_xor_nand of xor_nand is
   signal a, b: std_logic := ‘0’;
   signal y: std_logic;

   component xor_nand port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: xor_nand port map (a, b, y);

   a =’0’ after 10 ns;
   b =’0’ after 10 ns;
```

```
   . . .
   . . .

End architecture
```

**Try**

1. Realize XNOR gate using minimum number of NAND gates

### 9.1.3.4 Three input NAND gate using min no of 2 input NAND Gate

Implement 3 input NAND gate realization using minimum number of NAND gates.

a) A and B to the first NAND gate

b) Output of first Nand gate is given to the two inputs of the second NAND gate (this basically realizes the inverter functionality)

c) Output of second NAND gate is given to the input of the third NAND gate, whose otherinput is C ((A NAND B) NAND (A NAND B)) NAND C Thus, can be implemented using '3'2-input NAND gates.

**Hints:**

Assume three inputs of the NAND gate are A, B and C and connect these inputs as

```
/**   Declare the port signals */
entity nand3 port (
          a, b, c: in std_logic;
          y: out std_logic);
end nand3;

/**   architecture body  */
architecture arch_nand3 of nand3 is

component nand2_gate port (
          a, b: in std_logic;
          y: out std_logic);
   end component;

begin
   DUT: nand2_gate port map (a, b, y);
    . . .

End arch_nand3;

//Write the test bench for providing the stumulus

entity tb_nand3 port
end tb_nand3;

architecture arch_tb_nand3 of tb_nand3 is
   signal a, b: std_logic := ‘0’;
   signal y: std_logic;

   component nand2 port (
          a, b: in std_logic;
```

```
          y: out std_logic);
   end component;

begin
   DUT: nand2 port map (a, b, y);

   a =’0’ after 10 ns;
   b =’0’ after 10 ns;

   . . .
   . . .

   a = ‘1’ after 10 ns;
   b = ‘1’ after 10 ns;

End architecture
```

**Try:**

Implement XNOR gate realization using minimum number of NAND gates

### 9.1.3.5 User defined logic gate (Muller-C element cell)

Develop a behavioral model for a two-input Muller-C element cell, with two input ports and one output, all of type bit. The inputs and outputs are initially ‘0’. When both inputs are ‘1’, the output changes to ‘1’. It stays ‘1’ until both inputs are ‘0’, at which time it changes back to ‘0’. Your model should have a propagation delay for rising output transitions of 3.5 ns, and for falling output transitions of 2.5 ns.

**Hints:**

Assume three inputs of the NAND gate are A, B and C and connect these inputs as,

```
Take inputs A and B

Extract the truth table and Boolean expression as per the specifications

Implement the gate using VHDL model

Write the logic for selecting the stimulus for verifying the logic


/**   Declare the port signals */
entity mc_cell port (
          a, b, c: in std_logic;
          y: out std_logic);
end mc_cell;

/**   architecture body  */
architecture arch_mc_cell of mc_cell is

component mc_cell port (
          a, b: in std_logic;
          y: out std_logic);
   end component;

begin
```

```
   DUT: nand2_gate port map (a, b, y);
    . . .

End arch_mc_cell;

//Write the test bench for providing the stumulus

entity tb_mc_cell port
end tb_mc_cell;

architecture arch_tb_mc_cell of tb_mc_cell is
   signal a, b: std_logic := '0';
   signal y: std_logic;

   component mc_cell port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT: mc_cell port map (a, b, y);

   a ='0' after 10 ns;
   b ='0' after 10 ns;

   . . .
   . . .

   a = '1' after 10 ns;
   b = '1' after 10 ns;

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

Develop a behavioral model for a two-input Muller-C element cell, with two input ports and one output, all of type bit. The inputs and outputs are initially '1'. When both inputs are '0', the output changes to '0'. It stays '0' until both inputs are '1', at which time it changes back to '1'.

## 10. Exercises on Adder / Subtractor

In digital electronics, adders and subtractors both are the combinational logic circuits (a combinational logic circuit is one whose output depends only on the present inputs, but not on the past outputs) that can add or subtract numbers, more specifically binary numbers. Adders and subtractors are the crucial parts of arithmetic logic circuits in processing devices like microprocessors or microcontrollers.

A combinational logic circuit which is designed to add two binary digits is called as a half adder. The half adder provides the output along with a carry value (if any). The half adder circuit is designed by connecting an EX-OR gate and one AND gate. It has two input terminals and two output terminals for sum and carry.

## 10.1. Implementation of half adder

Design a gate level circuit for half adder and verify for the following truth table and write a test bench for verifying the functionality of the ha1f adder.

```
Consider the inputs are A, B and outputs are Sum, Cout
A       B       Sum     Cout
0       0       0       0
0       1       1       0
1       0       1       0
1       1       0       1
```

### Hints

The code pattern for implementing half adder:

```
// Declare the port signals
Input a, b;
Output sum, cout


//Write VHDL model for half adder in behavioral flow model
entity ha_df port (
          a, b: in std_logic;
          sum, cout: out std_logic);
end ha_df;

/**   architecture body */
architecture arch_ha_df of ha_df is

begin
   process(a, b)
   begin

      case {a, b} is
        when “00” => sum <= ‘0’; cout <= ‘0’;
        when “01” => sum <= ‘1’; cout <= ‘0’;
        when “10” => sum <= ‘1’; cout <= ‘0’;
        . . .
        . . .
        . . .
      End case
   End process;

End arch_ha_df;

//Write the test bench for providing the stumulus

entity tb_ha_df port
end tb_ha_df;

architecture arch_tb_ha_df of tb_ha_df is
   signal a, b: std_logic := ‘0’;
   signal sum, cout: std_logic;
```

```
   component ha_df port (
          a, b: in std_logic;
          sum, cout: out std_logic);
   end component;

begin
   DUT: ha_df port map (a, b, sum, cout);

   process
   begin
     a = ~a;
     wait for 10ns;
     b = ~b;
     wait for 15ns;

   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

#### 10.1.1.1 Implementation of full adder

Design a gate level circuit for full adder and verify for the following truth table and write a test bench for verifying the functionality of the full adder.

```
Consider the inputs are A, B, C and outputs are Sum, Cout

A       B       C       Sum     Cout
0       0       0       0       0
0       0       1       1       0
0       1       0       1       0
0       1       1       0       1
1       0       0       1       0
1       0       1       0       1
1       1       0       0       1
1       1       1       1       1
```

**Hints**

The pseudo code for printing full adder:

```
// Declare the port signals

Input a, b, c;

Output sum, cout

Work lib should consists of xor, and, or gates modules

//Write VHDL model for half adder in behavioral flow model
entity fa_gl port (
         a, b, cin: in std_logic;
         sum, cout: out std_logic);
```

```
end fa_gl;

/**     architecture     body     */
architecture arch_fa_gl of ga_gl is
   component xor_gate port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

    component declaration for or_gate, and_gate
begin

    u0: xor_gate port map(a, b, x1);
    u1: xor_gate port map(x1, cin, sum);
    . . .
    . . .

End arch_fa_gl;

//Write the test bench for providing the stumulus

entity tb_fa_gl port
end tb_fa_gl;

architecture arch_tb_fa_gl of tb_fa_gl is
   signal a, b, cin: std_logic := '0';
   signal sum, cout: std_logic;

   component fa_gl port (
         a, b, cin: in std_logic;
         sum, cout: out std_logic);
   end component;

begin
   DUT: fa_gl port map (a, b, cin, sum, cout);

   process
   begin
     a = ~a;
     wait for 5ns;
     b = ~b;
     wait for 10ns;
     cin = ~cin;
     wait for 15ns;

   end process

End architecture
```

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA

### 10.1.1.2 Realization of full adder using half adders

Design a gate level circuit for full adder using half adder and verify for the functionality and write a test bench for verifying the functionality of the full adder.

**Hints**

The pseudo code for printing full adder:

```
// Declare the port signals
Input a, b, c;
Output sum, cout


Work lib should consists of half adder, or_gate modules


entity fa_ha port (
         a, b, cin: in std_logic;
         sum, cout: out std_logic);
end fa_ha;

/**   architecture body  */
architecture arch_fa_ha of ga_ha is
   component ha_df port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

    component declaration for or_gate, and_gate
begin

    u0: xor_gate port map(a, b, x1);
    u1: xor_gate port map(x1, cin, sum);
    . . .
    . . .

End arch_fa_gl;

//Write the test bench for providing the stumulus

entity tb_fa_gl port
end tb_fa_gl;

architecture arch_tb_fa_gl of tb_fa_gl is
   signal a, b, cin: std_logic := '0';
   signal sum, cout: std_logic;

   component fa_gl port (
         a, b, cin: in std_logic;
         sum, cout: out std_logic);
   end component;

begin
   DUT: fa_gl port map (a, b, cin, sum, cout);

   process
```

```
   begin
     a = ~a;
     wait for 5ns;
     b = ~b;
     wait for 10ns;
     cin = ~cin;
     wait for 15ns;

   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

#### 10.1.1.3 Design and implement 4-bit ripple carry adder

Design a gate level circuit for ripple carry adder using full adder and verify for the functionality and write a test bench for verifying the functionality of the ripple carry adder.

**Hints**

The pseudo code for printing full adder:

```
// Declare the port signals
Input a, b;      //vector of 4-bit size
Input cin;
Output sum;      //vector of 4-bit size
Output cout;


// Declare xor gate, and gate
Component declaration for full adder


// instance xor gate, and gate
Port map for full adder
```

#### 10.1.1.4 Implementation of half subtractor

Design a gate level circuit for half subtractor and verify for the following truth table and write a test bench for verifying the functionality of the ha1f subtractor.

```
Consider the inputs are A, B and outputs are Diff, Bout

A       B       Diff    Bout
0       0       0       0
0       1       1       1
1       0       1       0
1       1       0       0
```

### Hints

The code pattern for printing half subtractor:

```
// Declare the port signals
Input a, b;
Output diff, bout

// Declare xor gate, and gate
Component declaration for xor gate, and gate

// instance xor gate, and gate
Port map for xor gate
Port map for and gate
```

#### 10.1.1.5 Implementation of full subtractor

Design a gate level circuit for full subtractor and verify for the following truth table and write a test bench for verifying the functionality of the full subtractor.

```
Consider the inputs are A, B, Bin and outputs are Diff, Bout

A       B       Bin     Sum     Bout
0       0       0       0       0
0       0       1       1       1
0       1       0       1       1
0       1       1       0       1
1       0       0       1       0
1       0       1       0       0
1       1       0       0       0
1       1       1       1       1
```

### Hints

The pseudo code for printing full subtractor:

```
// Declare the port signals
Input a, b, bin;
Output diff, bout
// Declare xor gate, and gate
Component declaration for xor gate, and gate
// instance xor gate, and gate
Port map for xor gate
Port map for and gate
```

#### 10.1.1.6 Realization of full subtractor using half subtractor

Design a gate level circuit for full subtractor using half subtractor and verify for the functionality and write a test bench for verifying the functionality of the full subtractor.

**Hints**

The pseudo code for printing full subtractor:

```
// Declare the port signals
Input a, b, bin;
Output diff, bout
// Declare xor gate, and gate
Component declaration for half subtractor, and gate, or gate
// instance xor gate, and gate
Port map for half subtractor
Port map for and gate
Port map for or gate
```

**Try**

Design a gate level circuit for ripple carry adder using full adder and verify for the functionality and write a test bench for verifying the functionality of the ha1f adder.

**Hints**

The pseudo code for printing full adder:

```
// Declare the port signals
Input a, b;      //vector of 4-bit size
Input cin;
Output sum;      //vector of 4-bit size
Output cout;

// Declare xor gate, and gate
Component declaration for full adder

// instance xor gate, and gate
Port map for full adder
```

## 11. Exercises on Multiplexer, Decoders

In digital systems, many times it is necessary to select a single data line from several data-input lines and the data from the selected data input line should be available on the output line. The digital circuit which does this task is a multiplexer.

A multiplexer is a digital circuit that selects one of the n data inputs and forwards it to the output. The selection of one of the n inputs is done by the select inputs. To select one of several inputs, we need m select lines such that 2^m =n.

### 11.1. Implementation of 2x1, 4x1 multiplexers

Develop a behavioral model for a two-input multiplexer, with ports of type bit and a propagation delay from data or select input to data output of 5 ns. You should declare a constant for the propagation delay, rather than writing it as a literal in signal assignments in the model.

The inputs to the MUX are data inputs I1, I0 and a one control input SEL(s) The single output is Y.

**Hints**

```
/**     Implementation of 2x1 multiplexer    **/

Declare the inputs I0, I1 and S
Declare the output Y.

//Write the logic for selecting the data depends    on the select line and
pass to the output

entity mux_2x1 port (
         i0, i1, s: in std_logic;
         y: out std_logic);
end mux_2x1;

/**   architecture body  */
architecture arch_mux_2x1 of mux_2x1 is

   component nand port (
         a, b: in std_logic;
         y: out std_logic);
   end component;

   component inv port (
         i: in std_logic;
         y: out std_logic);
   end component;

begin
   DUT1: inv port map (s, sb);
   DUT2: nand2_gate port map (i0, sb, s1); . . .

   . . .
   . . .

End arch_mux2x1;

//Write the test bench for providing the stumulus

entity tb_mux2x1 port
end tb_mux2x1;

architecture arch_tb_mux2x1 of tb_mc_mux2x1 is
   signal i0, i1, s: std_logic := '0';
   signal y: std_logic;

   component mux2x1 port (
         i0, i1, s: in std_logic;
         y: out std_logic);
   end component;
```

```
begin
   DUT: mux2x1 port map (i0, i1, s, y);

   s =’0’ after 10 ns, ‘1’ after 10 ns;
   process
   begin
     i0 = ~i0;
     wait for 10ns;
     i1 = ~i1;
     wait for 15ns;

   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA


Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### Try

Develop a behavioral model for a four-input multiplexer, with ports of type bit and a propagation delay from data or select input to data output of 4.5 ns. You should declare a constant for the propagation delay, rather than writing it as a literal in signal assignments in the model.

## 11.2 Implementation of 2x1, 4x1 demultiplexers

Build a behavioral model for a two-input demultiplexers, with ports of type bit and a propagation delay from data or select input to data output of 5 ns.

### Hints

```
/**
  Implementation of 2x1 demultiplexer.
 **/

Declare the inputs I and S
Declare the output Y0, Y1.

//Write the logic for selecting the data depends on the select line and
pass to the output

entity dmux_1x2 port (
          i, s: in std_logic;
          y0, y1: out std_logic);
end dmux_1x2;
```

```
/**   architecture body  */
architecture arch_dmux_1x2 of dmux_1x2 is

begin
   process(I,s)
   begin
      case S is
        when ‘0’: y0 = I; y1 = ‘z’;
        . . .
        . . .

End arch_dmux1x2;

//Write the test bench for providing the stumulus

entity tb_dmux1x2 port
end tb_dmux1x2;

architecture arch_tb_dmux1x2 of tb_dmux1x2 is
   signal i, s: std_logic := ‘0’;
   signal y0, y1: std_logic;

   component dmux1x2 port (
         i, s: in std_logic;
         y0, y1: out std_logic);
   end component;

begin
   DUT: dmux1x2 port map (i, s, y0, y1);

   s =’0’ after 10 ns, ‘1’ after 10 ns;
   process
   begin
     i = ~i;
     wait for 10ns;

   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### Try

1. Modify the program to function as 1x4 demultiplexers
2. Modify the program to function as 1x8 demultiplexers

## 11.3 Realization of higher order multiplexers using lower order multiplexers

Realize the higher order multiplexers using lower order multiplexers. Write the VHDL model for the realized circuits. Simulate the test benches for the corresponding and verify the design under test by plotting the waveforms. Figure 11.3 shows realization of 4x1 mux using 2x1 mux.

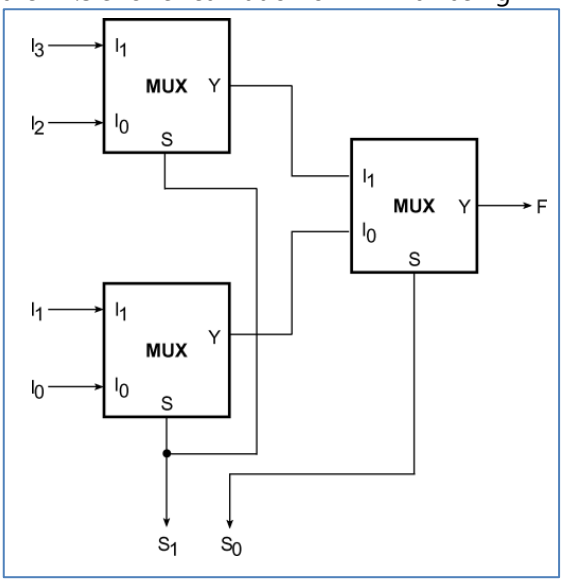


Figure 11.3: Realization of 4x1 mux using 2x1 mux

```
/**
 Realize the lower order multiplexers for design of higher order multiplexers.
 **/

In work library simulate the lower order multiplexer

For the realized higher order multiplexer, instance the component in the
declaration part of the architecture

By using positional or name mapping instance the component with the signals
```

### Hint

```
//Write the logic for selecting the data depends on the select line and
pass to the output

entity mux_4x1 port (
         i0, i1, i2, i3: in std_logic;
         s: in std_logic_vector(1 donwto 0);
         y: out std_logic);
end mux_4x1;

/**   architecture body */
architecture arch_mux_4x1 of mux_4x1 is

   component mux2x1 port (
         i0, i1, s: in std_logic;
         y: out std_logic);
   end component;
   // declare intermediate signals

begin
   DUT1: mux2x1 port map (i0, i1, s(0), s1);
```

```
   DUT2: mux2x1 port map (i2, i3, s(1), s2);
   . . .
   . . .
   . . .

End arch_mux4x1;

//Write the test bench for providing the stumulus

entity tb_mux4x1 port
end tb_mux4x1;

 architecture arch_tb_mux4x1 of tb_mux4x1 is
    signal i0, i1, i2, i3: std_logic := '0';
   signal s: std_logic_vector(1 donwto 0) := "00";
   signal y: std_logic;

   component mux4x1 port (
         i0, i1, i2, i3: in std_logic;
         s: in std_logic_vector(1 donwto 0);
         y: out std_logic);
   end component;

begin
   DUT: mux4x1 port map (i0, i1, i2, i3, s, y);

   s ='00' after 10 ns, '01' after 10 ns . . .;
   process
   begin
     i0 = ~i0;
     wait for 10ns;
     i1 = ~i1;
     wait for 12ns;
     i2 = ~i2;
     wait for 14ns;
     i3 = ~i3;
     wait for 16ns;
   end process

End architecture


Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### 11.4 Realization of basic gates using 2x1 multiplexer

Realize all the basic gates like AND and inverter using 2x1 multiplexer.

```
/**   Realize the multiplexers for function as basic logic gates **/

In work library simulate the 2x1 multiplexer

For the realized basic gates using 2x1 multiplexer, instance the component in the
declaration part of the architecture
```

**Hints**

```
//Write the logic for selecting the data depends on the select line and
pass to the output

entity mux_and port (
         a, b: in std_logic;
         y: out std_logic);
end mux_2x1;

/**   architecture body  */
architecture arch_mux_and of mux_and is

   component mux2x1 port (
         i0, i1, s: in std_logic;
         y: out std_logic);
   end component;
   // declare intermediate signals

begin
   DUT1: mux2x1 port map (‘0’, b, a, y);
   . . .
   . . .
   . . .
End arch_mux_and;

//Write the test bench for providing the stumulus

entity tb_mux_and port
end tb_mux_and;

architecture arch_tb_mux_and of tb_mux_and is
   signal a, b: std_logic := ‘0’;
   signal y: std_logic;

   component mux_and port (
         a, b: in std_logic;
         y: out std_logic);
   end component;
begin
   DUT: mux_and port map (a, b, y);

   process
   begin
     a = ~a;
     wait for 10ns;
     b = ~b;
     wait for 12ns;

   end process

End architecture
Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1 Realize all the OR gate using 2x1 multiplexer

2 Realize all the basic gates like XOR and XNOR using 2x1 multiplexer

3 Realize all the inverter using 2x1 multiplexer

## 11.5 Introduction

A decoder is a multiple input multiple output logic circuit which converts coded input into coded output where input and output codes are different. The input code generally has fewer bits than the output code. Each input code word produces a different output code word i.e there is one to one mapping can be expressed in truth table. In the block diagram of decoder circuit the encoded 2n information is present as n input producing 2 through output 2^n -1.

#### 11.5.1.1 Implementation of 2 to 4 decoder

Write the VHDL code for the circuit contains an input bundle of two input signals and an output bundle of four decoded signals. The input bundle, $i_0$, $i_1$ represents decoder inputs. The output bus, Y0, Y1, Y2 and Y3, are used to indicate the decoded output for the two inputs. The relationship between the input and output is shown in the table below. Use a selected signal assignment statement in the solution.

```
/**
  Implementation of 2 to 4 decoder.
 **/

Declare the inputs I0, I1.
Declare the output Y0, Y1, Y2  and Y3.

I1      I0      Y3      Y2      Y1      Y0
0       0       0       0       0       1
0       1       0       0       1       0
1       0       0       1       0       0
1       1       1       0       0       0
```

**Hints**

```
//Write VHDL model for 2 to 4 decoder gate level model
entity dec2to4 port (
          i0, i1: in std_logic;
          y0, y1, y2, y3: out std_logic);
end dec2to4;

/**   architecture body */
architecture arch_dec2to4 of dec2to4 is

   component nand_gate port (
         a, b: in std_logic;
         y: out std_logic);
   end component;
    // component declaration for inverter
   // declare intermediate signals

begin
   DUT1: nand_gate port map (ni1, ni0, y0);
   . . .
```

```
   . . .
   . . .

End arch_dec2to4;

//Write the test bench for providing the stumulus

entity tb_dec2to4 port
end tb_dec2to4;

architecture arch_tb_dec2to4 of tb_dec2to4 is
   signal i0, i1: std_logic := '0';
   signal y0, y1, y2, y3: std_logic;

   component dec2x4 port (
         i0, i1: in std_logic;
         y0, y1, y2, y3: out std_logic);
   end component;

begin
   DUT: dec2to4 port map (i0,i1, y0, y1, y2, y3);

   process
   begin
     i0 = ~i0;
     wait for 10ns;
     i1 = ~i1;
     wait for 15ns;

   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### Try

1 Write a program to implement gate level model for 2 to 4 decoder. Plot the waveforms

2 Write a program to implement behavioral model for 2 to 4 decoder. Plot the waveforms

## 11.6 Implementation of 3 to 8 decoder

Build behavioral model to function as 3 to 8 decoder.

### Hints

```
/** Behavioral model implementation of 3 to 8 decoder     **/

Declare the inputs I0, I1, I2.

Declare the output Y0, Y1, Y2, Y3, Y4, Y5, Y6 and Y7.

//Write VHDL model for 2 to 4 decoder gate level model
entity dec3to8 port (
         i0, i1, i2: in std_logic;
```

```
          y0, y1, y2, y4, y5, y6, y7: out std_logic);
end dec3to8;

/**   architecture body  */
architecture arch_dec3to8 of dec3to8 is

   // declare intermediate signals

begin
   process(i0, i1, i2)
   begin
      { y0, y1, y2, y4, y5, y6, y7} = “00000000”;

      case {i2, i1, i0} is
        when “000” => Y0 <= ‘1’;
        when “001” => Y0 <= ‘1’;
        when “010” => Y0 <= ‘1’;
        . . .
        . . .
        . . .
      End case
   End process;

End arch_dec3to8;

//Write the test bench for providing the stumulus

entity tb_dec3to8 port
end tb_dec3to8;

architecture arch_tb_dec3to8 of tb_dec3to8 is
   signal i0, i1, i2: std_logic := ‘0’;
   signal y0, y1, y2, y3, y4, y5, y6, y7: std_logic;

   component dec3x8 port (
         i0, i1, i2: in std_logic;
         y0, y1, y2, y3, y4, y5, y6, y7: out std_logic);
   end component;

begin
   DUT: dec3to8 port map (i0, i1, i2, y0, y1, y2, y3, y4, y5, y6, y7);

   process
   begin
     i0 = ~i0;
     wait for 10ns;
     i1 = ~i1;
     wait for 15ns;

   end process

End architecture
```

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA

### Try

1 Construct a 4-to-16 line decoder with two 3-to-8 line decoders having active LOW ENABLE inputs.

2 Implement the three-variable Boolean function $F = \bar{a}c + a\bar{b}\bar{c} + ab\bar{c}$ using (i) an 8-to-1 multiplexer and (ii) a 4-to-1 multiplexer.

## 11.7 Realization of higher order multiplexers using lower order multiplexers

Construct a 4-to-16 line decoder with two 3-to-8 line decoders having active LOW ENABLE inputs.

### Hints

```
/**
 Realize the lower order multiplexers for design of higher order multiplexers
**/

In work library simulate the lower order multiplexer

For the realized higher order multiplexer, instance the component in the
declaration part of the architecture

By using positional or name mapping instance the component with the signals

Design a test bench

Write the logic for generating stimulus for the input signals

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### Try

Construct a 3-to-8 line decoder with two 2-to-4 line decoders having active LOW ENABLE inputs.

#### 11.7.1.1 Realization of basic gates using 2x1 multiplexer

Realize all the basic gates like AND, OR, XOR, XNOR and inverter using 2x1 multiplexer

```
/**   Realize the multiplexers for function as basic logic gates  **/

In work library simulate the 2x1 multiplexer

For the realized basic gates using 2x1 multiplexer, instance the component in the
declaration part of the architecture

By using positional or name mapping instance the component with the signals

Design a test bench

Write the logic for generating stimulus for the input signals

Plot the wave form for all possible select lines
```

```
Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**
Realization of BCD-to-seven-segment decoder for a light emitting diode (LED) display

Develop a functional model of a BCD-to-seven-segment decoder for a light emitting diode (LED) display. The decoder has a 4-bit input that encodes a numeric digit between 0 and 9. There are seven outputs indexed from 'a' to 'g', corresponding to the seven segments of the LED display as shown in the margin. An output bit being '1' causes the corresponding segment to illuminate. For each input digit, the decoder activates the appropriate combination of segment outputs to form the displayed representation of the digit.

**Hint:**
For example, for the input "0010", which encodes the digit 2, the output is "1101101". Your model should use a selected signal assignment statement to describe the decoder function in truth-table form.

## 11.7.2 Introduction

An encoder is a digital circuit that performs inverse operation of a decoder. An encoder has 2n input lines and n output lines. In encoder the output lines generates the binary code corresponding to the input value. In octal to binary encoder it has eight inputs, one for each octal digit and three output that generate the corresponding binary code. In encoder it is assumed that only one input has a value of one at any given time otherwise the circuit is meaningless. It has an ambiguila that when all inputs are zero the outputs are zero. The zero outputs can also be generated when D0 =1.

### 11.7.2.1 Implementation of 4 to 2 encoder

An encoder is a digital circuit that converts a set of binary inputs into a unique binary code. The binary code represents the position of the input and is used to identify the specific input that is active. Encoders are commonly used in digital systems to convert a parallel set of inputs into a serial code.

The 4 to 2 Encoder consists of four inputs Y3, Y2, Y1 & Y0, and two outputs A1 & A0. At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The figure below shows the logic symbol of the 4 to 2 encoder.

```
/**   Implementation of 4 to 2 decoder  **/

Declare the inputs I0, I1, I2, I3.
Declare the output Y0 and Y1

I3      I2      I1      I0      Y1      Y0
0       0       0       1       0       0
0       0       1       0       0       1
0       1       0       0       1       0
1       0       0       0       1       1
```

### Hints

```
//Write VHDL model for 2 to 4 decoder gate level model
entity enc4to2 port (
    x : in STD_LOGIC_VECTOR(3 downto 0);
    y : out STD_LOGIC_VECTOR(1 downto 0) );
end enc4to2;

architecture arch_enc4to2 of enc4to2 is
begin

    process(i)
    begin
        if (i="1000") then y <= "00";
        elsif (i="0100") then y <= "01";
        elsif (i="0010") then y <= "10";
        elsif (i="0001") then y <= "11";
        else  y <= "ZZ";
        end if;
    end process;

End arch_enc4to2;

//Write the test bench for providing the stumulus

entity tb_enc4to2 port
end tb_enc4to2;

architecture arch_tb_enc4to2 of tb_enc4to2 is
   signal i : in STD_LOGIC_VECTOR(3 downto 0) := “0000”;
   signal y : out STD_LOGIC_VECTOR(1 downto 0);

   component enc4to2 port (
          x : in STD_LOGIC_VECTOR(3 downto 0);
          y : out STD_LOGIC_VECTOR(1 downto 0));
   end component;

begin
   DUT: enc4to2 port map (i, y);

   process
   begin
     i = i + ‘1’;
     wait for 10ns;
   end process

End architecture


Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

### Try

1. Modify VHDL behavioral model with gate level model for 4 to 2 encoder. Plot the waveforms.
2. Decimal to BCD Encoder.

The decimal-to-binary encoder usually consists of 10 input lines and 4 output lines. Each input line corresponds to each decimal digit and 4 outputs correspond to the BCD code. This encoder accepts the decoded decimal data as an input and encodes it to the BCD output which is available on the output lines.

3. Octal to Binary Encoder (8 to 3 Encoder)

The 8 to 3 Encoder or octal to Binary encoder consists of 8 inputs: Y7 to Y0 and 3 outputs: A2, A1 & A0. Each input line corresponds to each octal digit and three outputs generate corresponding binary code.

### 11.7.2.2 Implementation of 8 to 3 priority encoder

A 8 to 3 priority encoder has eight inputs Y7, Y6, Y5, Y4, Y3, Y2, Y1 & Y0 and two outputs A2, A1 and A0. Here, the input, Y7 has the highest priority, whereas the input, Y0 has the lowest priority. In this case, even if more than one input is '1' at the same time, the output will be the binary code corresponding to the input, which is having higher priority.

We considered one more output, V in order to know, whether the code available at outputs is valid or not.

- If at least one input of the encoder is '1', then the code available at outputs is a valid one. In this case, the output, V will be equal to 1.
- If all the inputs of encoder are '0', then the code available at outputs is not a valid one. In this case, the output, V will be equal to 0.

The Truth table of 4 to 2 priority encoder is shown below.

```
/**   Implementation of 4 to 2 decoder **/

Declare the inputs I0, I1, I2,  I3.
Declare the output Y0 and Y1

I3      I2      I1      I0      Y1      Y0
0       0       0       1       0       0
0       0       1       0       0       1
0       1       0       0       1       0
1       0       0       0       1       1
```

**Hints**

```
//Write VHDL model for 2 to 4 decoder gate level model
entity penc8to3 port (
    x : in STD_LOGIC_VECTOR(7 downto 0);
    y : out STD_LOGIC_VECTOR(2 downto 0) );
end enc4to2;

architecture arch_penc8to3 of penc8to3 is
begin

   y <= "111" when i(7)='1' else
        "110" when i(6)='1' else
        "101" when i(5)='1' else
        "100" when i(4)='1' else
        "011" when i(3)='1' else
        "010" when i(2)='1' else
```

```
         "001" when i(1)='1' else
         "000" ;

End arch_penc8to3;

//Write the test bench for providing the stumulus

entity tb_penc8to3 port
end tb_penc8to3;

architecture arch_tb_penc8to3 of tb_penc8to3 is
   signal i : in STD_LOGIC_VECTOR(7 downto 0) := “00000000”;
   signal y : out STD_LOGIC_VECTOR(2 downto 0);

   component penc8to3 port (
          x : in STD_LOGIC_VECTOR(7 downto 0);
          y : out STD_LOGIC_VECTOR(2 downto 0));
   end component;

begin
   DUT: penc8to3 port map (i, y);

   process
   begin
     i = i + ‘1’;
     wait for 10ns;
   end process

End architecture

Plot the wave form for all possible select lines

Synthesize the design

Elaborate the design and dump the bit file into FPGA
```

**Try**

1. Modify VHDL behavioral model with gate level model for 8 to 3 encoder. Plot the waveforms.
2. Realize 8to3 priority encoder using 2x1 mux and implement with VHDL gate level model.

# 12. Exercises on Shift Register

Shift registers, like counters, are a form of sequential logic. Sequential logic, unlike combinational logic, is not only affected by the present inputs but also by the prior history. In other words, sequential logic remembers past events. Shift registers produce a discrete delay of a digital signal or waveform.

## 12.1 4- bit barrel shifter

A barrel shifter is a digital circuit that can shift a data word by a specified number of bits without use of any sequential logic, only pure combinational logic. There are 3 type of bitwise shift operation: logical shift, arithmetic shift, and circular shift (rotate). The data can be shifted to the left as well as to the right circular shift.

**Hints**

Consider the port specification as

Input datain // 8 bit wide

Input sel // 3 bit wide to support 16 operations

Input dataout // 8 bit wide

The circuit allows rotating the input data word right, where the amount of rotating is selected by the control inputs. The circuit can design by three stages of 2:1 multiplexer.

When all multiplexer select inputs are active (low), the input data passes straight through the cascade of the multiplexers and the output data ($q_7$.....$q_0$) is equal to the input data ($d_7$..... $d_0$). When $S_2$control signal is selected, the first stage of multiplexers performs a rotate-right by one bit operation, due to their inter-connection to the next lower input.

The second stage of multiplexers performs a rotate-right by two bits when $S_1$ control signal is selected. Here the corresponding multiplexer inputs are connected to their second next-lower input.

Finally, the third stage of multiplexers performs a rotate-right by four bits, when $S_0$ control signal is selected. The design uses a case statement to exhaustively list all combinations of the amt signal and the corresponding rotated results.

**Try:**

1. Modify the barrel shifter by changing the number of select lines to 4 bit wide to support 16 different functionalities.
2. Modify the barrel shifter by changing the number of data lines to 16 bit wide to support 16 different functionalities.

## 12.2. 8- bit ALU

Arithmetic Logic Unit (ALU) is one of the most important digital logic components in CPUs. It normally executes arithmetic operations such as addition, subtraction, multiplication, division, etc. and logic operations such as and, or, xor, xnor, nand, nor, not, buffer, rotate and shift operations.

**Hints**

Consider the port specification as

Input A, B  // 8 bit wide

Input sel  // 4 bit wide to support 16 operations

Input dataout  // 8 bit wide

// The logic and arithmetic operations being implemented in the ALU are as follows:

1. Arithmetic Addition: ALU_Out = A + B;
2. Arithmetic Subtraction ALU_Out = A - B;
3. Arithmetic Multiplication ALU_Out = A * B;
4. Arithmetic Division ALU_Out = A / B;
5. Logical Shift Left ALU_Out = A logical shifted left by 1;
6. Logical Shift Right ALU_Out = A logical shifted right by 1;
7. Rotate Left ALU_Out = A rotated left by 1;
8. Rotate Right ALU_Out = A rotated right by 1;
9. Logical AND ALU_Out = A AND B;
10. Logical OR ALU_Out = A OR B;
11. Logical XOR ALU_Out = A XOR B;
12. Logical NOR ALU_Out = A NOR B;
13. Logical NAND ALU_Out = A NAND B;
14. Logical XNOR ALU_Out = A XNOR B;
15. Greater comparison ALU_Out = 1 if A > B else 0;
16. Equal comparison

ALU_Out = 1 A = B else 0;

The second stage of multiplexers performs a rotate-right by two bits when $S_1$ control signal is selected. Here the corresponding multiplexer inputs are connected to their second next-lower input.

Finally, the third stage of multiplexers performs a rotate-right by four bits, when $S_0$ control signal is selected. The design uses a case statement to exhaustively list all combinations of the amt signal and the corresponding rotated results.

**Try:**

1. Modify the ALU by changing the number of select lines to 4 bit wide to support 16 different functionalities.
2. Modify the ALU by changing the number of data lines to 16 bit wide to support 16 different functionalities.

# 13 Exercises on Latches and Flip-flops

Computers and calculators use Flip-flop for their memory. A combination of number of flip flops will produce some amount of memory. Flip flop is formed using logic gates, which are in turn made of transistors. Flip flop are basic building blocks in the memory of electronic devices. Each flip flop can store one bit of data. Latches and flip – flops are both 1 – bit binary data storage devices. The main difference between a latch and a flip – flop is the triggering mechanism. Latches are transparent when enabled, whereas flip – flops are dependent on the transition of the clock signal i.e. either positive edge or negative edge.

## 13.1 SR latch, JK latch, D latch and T latch

Construct an SR latch using NOR gates. Verify its operation and demonstrate the circuit. Write an entity declaration for a positive level-triggered SR-latch with asynchronous active-low preset and clear inputs, and Q and outputs. Include concurrent assertion statements and passive processes as necessary in the entity declaration to verify that,

- The preset and clear inputs are not activated simultaneously,
- The setup time of 6 ns from the J and K inputs to the rising clock edge is observed,
- The hold time of 2 ns for the J and K inputs after the rising clock edge is observed and
- The minimum pulse width of 5 ns on each of the clock, preset and clear inputs is observed.

Write a gate level architecture body for the SR latch and a test bench that exercises the statements in the entity declaration.

### Hints

```
// Declare port signal

Input rst_l, clk;
Input s, r;

Output q, qb;

// Component declaration of NAND gates
Component NAND_gate (input a, b; output y);

// component instance in the architecture body of VHDL program
NAND_gate port map(s, qb, q);
NAND_gate port map(r, q, qb);

// Test bench for SR latch
Design a test bench to provide the stimulus for the inputs s, r
Generate a clk signal with 5MHz frequency
Generate the reset logic to reset the latch at initial
```

### Try

1. Construct SR latch using NAND gates. Verify its operation and demonstrate the circuit
2. Construct JK latch using NAND gates. Verify its operation and demonstrate the circuit
3. Construct D latch using NAND gates. Verify its operation and demonstrate the circuit
4. Construct T latch using NAND gates. Verify its operation and demonstrate the circuit

## 13.2 JK flip-flop, D flip-flop and T flip-flop

Construct flip-flops using latches and verify its operation and demonstrate the circuit.

Write an entity declaration for a positive edge-triggered JK-flipflop with asynchronous active-low preset and clear inputs, and Q and outputs. Include

concurrent assertion statements and passive processes as necessary in the entity declaration to verify that,

- The preset and clear inputs are not activated simultaneously,
- The setup time of 6 ns from the J and K inputs to the rising clock edge is observed,
- The hold time of 2 ns for the J and K inputs after the rising clock edge is observed
- The minimum pulse width of 5 ns on each of the clock, preset and clear inputs is observed.

Write a structural architecture body for the flipflop and a test bench that exercises the statements in the entity declaration.

### Hints

The pseudo code for SR latch

```
// Declare port signal

Input rst_l, clk;
Input s, r;

Output q, qb;

// Component declaration of NAND gates
Component NAND_gate (input a, b; output y);

// component instance in the architecture body of VHDL program
NAND_gate port map(s, qb, q);
NAND_gate port map(r, q, qb);

// Test bench for SR latch
Design a test bench to provide the stimulus for the inputs s, r
Generate a clk signal with 5MHz frequency
Generate the reset logic to reset the latch at initial
```

## Try

## Design 2-bit register

1. Write component instantiation statements to model the structure shown by the schematic diagram in Figure. Assume that the entity ttl_74x74 and the corresponding architecture basic have been analyzed into the library work. Figure 13.2 shows the 2-bit register.

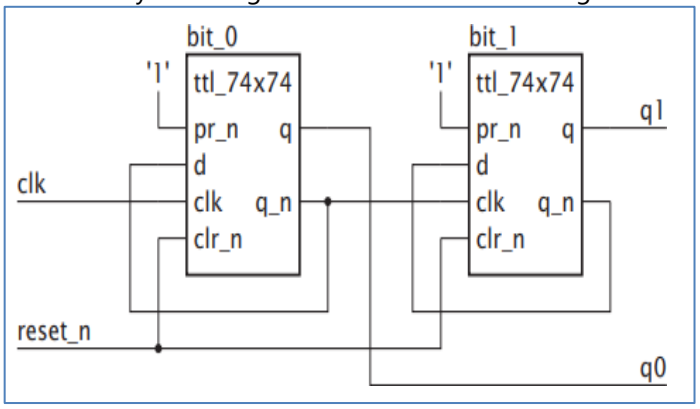

Figure 13.2: 2-bit register.

# 14. Exercises on Synchronous and Asynchronous Counters

A special type of sequential circuit used to count the pulse is known as a counter, or a collection of flip flops where the clock signal is applied is known as counters. The counter is one of the widest applications of the flip flop. Based on the clock pulse, the output of the counter contains a predefined state. The number of the pulse can be counted using the output of the counter.

There are two types of counters: Synchronous counter and Asynchronous counter.

## 14.1: 4-bit synchronous counter with synchronous reset

Build an entity for a 4-bit counter with synchronous reset input. Include a process in the entity declaration that measures the duration of each reset pulse and reports the duration at the end of each pulse.

### Hints

```
/**   Declare the port signals */
entity counter_synrst port (
         clk, rst: in std_logic;
         q: out std_logic_vector(3 downto 0));
end counter_synrst;

/**   architecturebody  */
architecture arch_counter_synrst of counter_synrst is

Begin
    Process (clk, rst)
    Begin

       If clk'event and clk = '1' then
           If rst then
                 q = "0000";
          else
                  q = q + 1;
       end if;
    end process

End arch_counter_synrst;

//Write the test bench for providing the stumulus

entity tb_counter_synrst port
end tb_counter_synrst;

architecture arch_tb_counter_synrst of tb_counter_synrst is

   signal clk, rst: std_logic := '0';
   signal q: std_logic_vector(3 downto 0);

   component counter_synrst port (
         clk, rst: in std_logic;
         q: out std_logic_vector(3 downto 0));
   end component;

begin
```

```
   DUT: counter_synrst port map (gray_code, binary_code);

   Process
   begin
        clk = ~clk;
        Wait for 10 ns;
   end process;

   Process
   begin
        rst = '0';
       Wait for 10 ns;
        rst = '1';
        wait;
   end process;

end architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design
```

**Try**

1. Design 4-bit synchronous counter with asynchronous reset
2. Design 4-bit asynchronous counter with synchronous reset
3. Design 4-bit asynchronous counter with asynchronous reset

## 14.2 Decade counter with asynchronous reset

Construct an entity for a decade counter with asynchronous reset input. Include a process in the entity declaration that measures the duration of each reset pulse and reports the duration at the end of each pulse.

### Hints

```
/**   Declare the port signals */
entity dec_counter_asynrst port (
          clk, rst: in std_logic;
          q: out std_logic_vector(3 downto 0));
end dec_counter_asynrst;

/**   architecturebody  */
architecture arch_dec_counter_asynrst of dec_counter_asynrst is

Begin
    Process (clk, rst)
    Begin

       If rst then
                 q = "0000";
       elif clk'event and clk = '1' then
```

```
           if (q = “1010”) then
                 q = “0000”;
           else
                  q = q + 1;
           end if;

       end if;
    end process

End arch_dec_counter_asynrst;

//Write the test bench for providing the stumulus

entity tb_dec_counter_asynrst port
end tb_dec_counter_asynrst;

architecture arch_tb_dec_counter_asynrst of tb_dec_counter_asynrst is

   signal clk, rst: std_logic := ‘0’;
   signal q: std_logic_vector(3 downto 0);

   component dec_counter_asynrst port (
         clk, rst: in std_logic;
         q: out std_logic_vector(3 downto 0));
   end component;

begin

   DUT: dec_counter_asynrst port map (clk, rst, q);

   Process
   begin
        clk = ~clk;
        Wait for 10 ns;
   end process;

   Process
   begin
        rst = ‘0’;
       Wait for 10 ns;
       rst = ‘1’;
       wait;
   end process;

end architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design
Elaborate the design and create bit file
Dump the bit file in zybo fpga
```

**Try**

1 Design decade synchronous counter with synchronous reset.

2 Design counter to count the events from 3 to 12.

#### 14.2.1.1 4-bit serial in serial out shift register (SISO)

In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a four-bit vector W = w3w2w1w0 one bit position to the right when a control signal Shift is equal to 1. Let the outputs of the circuit be a four-bit vector Y = y3y2y1y0 and a signal k, such that if Shift = 1 then y3 = 0, y2 = w3, y1 = w2, y0 = w1, and k = w0. If Shift = 0 then Y = W and k = 0.

Build an entity for a shift register to drive the resister serially and output the data serially. Write a test bench architecture to simulate and verify the design.

**Hints**

```
/**   Declare the port signals */
entity siso port (
        clk, rst: in std_logic;
        sin : in std_logic;
        q: out std_logic_vector(3 downto 0));
end siso;

/**   architecturebody */
architecture arch_siso of siso is

Begin
    Process (clk, rst)
    Begin

       If rst then
                 q = “0000”;
       elif clk’event and clk = ‘1’ then
          q[3] = sin;
          q[2] = q[3];
          q[1] = q[2];
          q[0] = q[1];

       end if;
    end process

End arch_siso;

//Write the test bench for providing the stumulus

entity tb_siso port
end tb_siso;

architecture arch_tb_siso of tb_siso is

   signal clk, rst: std_logic := ‘0’;
   signal sin: std_logic := ‘0’;
   signal q: std_logic_vector(3 downto 0);

   component siso port (
         clk, rst: in std_logic;
```

```
          sin : in std_logic;
          q: out std_logic_vector(3 downto 0));
   end component;

begin

   DUT: siso port map (clk, rst, sin, q);

   Process
   begin
        clk = ~clk;
        Wait for 10 ns;
   end process;

   Process
   begin
        rst = '0';
       Wait for 10 ns;
        rst = '1';
        wait;
   end process;

   Process
   begin
        sin = ~sin;
        Wait for 25 ns;
   end process;


end architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design
Elaborate the design and create bit file
Dump the bit file in zybo FPGA
```

**Try**

1 Design 4-bit serial in parallel out shift register (SIPO).
2 Design 4-bit parallel in serial out shift register (PISO).
3 Design 4-bit parallel in parallel out shift register (PIPO).

## 14.2.2 Introduction

A carry look-ahead adder (CLA) is an electronic adder used for binary addition. Due to the quick additions performed, it is also known as a fast adder. The CLA logic uses the concepts of generating and propagating carries. We can say that the CLA adder is the successor of the Ripple Carry Adder

### 14.2.2.1 Carry look ahead adder

Build 4- bit carry look ahead adder (CLA) and justify the speed of operation CLA is more than ripple carry adder

Develop a functional model of a 3-bit carry-look-ahead adder. The adder has two 3-bit data inputs, a (2 downto 0) and b(3 downto 0); a 3-bit data output, s(2 downto 0); a carry input, c_in; a carry output, c_out; a carry generate output, g; and a carry propagate output, p. The adder is described by the logic equations and associated propagation delays: where the Gi are the intermediate carry generate signals, the Pi are the intermediate carry propagate signals and the Ci are the intermediate carry signals. C–1 is c_in and C3 is cout. Your model should use the expanded equation to calculate the intermediate carries, which are then used to calculate the sums.

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$
$$g_i = a_i \,.\, b_i$$
$$p_i = a_i + b_i$$
$$c_i = g_i + p_i\, c_{i-1}$$

### Hints

```
/** Declare the port signals */
entity cla port (
        a, b: in std_logic-vector(2 downto 0);
        sum: out std_logic_vector(2 downto 0);
        cout: out std_logic);
end cla;

/**   architecturebody */
architecture arch_cla of cla is

Begin
    g0 = a[0] & b[0];
    p0 = a[0] | b[0];
    c1 = g0 + p0 & c0;
     . . . . . .
     . . . . . .
     . . . . . .

End arch_cla;

//Write the test bench for providing the stumulus
entity tb_cla port
end tb_cla;

architecture arch_tb_cla of tb_cla is

   signal a, b: std_logic_vector(2 downto 0) := “000”;
   signal cin: std_logic := ‘0’;
   signal sum: std_logic_vector(s downto 0);

   component cla port (
         a, b: in std_logic-vector(2 downto 0);
         sum: out std_logic_vector(2 downto 0);
         cout: out std_logic);
   end component;

begin

   DUT: cla port map (clk, rst, sin, q);

   Process
   begin
```

```
        a = a + “0110”
        wait for 10 ns;
        b = b + “1010”
   end process;

end architecture

// After post simulation

Simulate the design using Xilinx software

Plot the wave forms and verify the functionality of the design

Synthesize the design
Elaborate the design and create bit file
Dump the bit file in zybo FPGA
```

**Try:**

1. Design and implement 4-bit carry look ahead adder
2. Design and implement 4-bit ripple carry adder

## III. TEXT BOOKS


1. Douglas Perry, “VHDL”, Tata McGraw Hill, 4th edition, 2002.
2. W.H. Gothmann, “Digital Electronics- An introduction to theory and practice”, PHI, 2nd edition, 2006.


## IV. REFERENCE BOOKS


1. Jacob Millman, Herbert Taub, Mothiki S PrakashRao, “Pulse Digital and Switching Waveforms”, Tata McGraw-Hill, 3rd edition, 2008.
2. David A. Bell, “Solid State Pulse Circuits”, PHI, 4th edition, 2002.
3. D Roy Chowdhury, “Linear Integrated Circuits”, New Age International (p) Ltd, 2nd edition, 2003.
4. Ramakanth A. Gayakwad, “Op-Amps linear ICs”, PHI, 3rd edition, 2003