MARRI LAXMAN REDDY
INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(AN AUTONOMOUS INSTITUTION)
(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)
Accredited by NBA and NAAC with 'A' Grade & Recognized Under Section2(f) & 12(B)of the UGC act,1956

Department of Electronics & Communication Engineering

(2415521) Digital System Design with FPGAs Lab

M.TECH - I YEAR- I SEMESTER (ECE)
R24 (MLRS) REGULATION

INDEX

MARRI LAXMAN REDDY
INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(AN AUTONOMOUS INSTITUTION)
(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)
Accredited by NBA and NAAC with 'A' Grade & Recognized Under Section2(f) & 12(B)of the UGC act,1956

CERTIFICATE

This is to certify that this manual is a bonafide record of practical work in the ***Digital Systems Design with FPGAs lab*** in I **Semester of I -year M. Tech Sem I (ECE) Programme** during the academic year **2022-2023.** This book is prepared by **Dr. N Srinivas (Associate Professor), Dr. K. Naveen Kumar (Associate Professor), Mrs. R Babitha (Assistant Professor), Mrs. B Manjula (Assistant Professor),** Department of Electronics and Communication Engineering.

LAB I/C **Head of the Department**

PREFACE

It is one of the core areas of ECE and constitutes the largest applications in use today. Communication has entered into every part of today's world. This laboratory is intended to make students understand the use of different Digital Systems Design with FPGA's and is designed to help students understand the basic principles of design techniques as well as giving them the insight on design, simulation and hardware implementation of circuits. The main aim is to provide hands-on experience to the students so that they are able to put theoretical concepts to practice. The content of this course consists of two parts, 'simulation' and 'hardwired'. Students will carry out design experiments as a part of the experiments list provided in this lab manual. Students will be given a specific design problem, which after completion they will verify using the simulation software or hardwired implementation.

By,

Dr. N Srinivas (Associate Professor)
Mrs. B Manjula (Assistant Professor)
Mrs. R Babitha (Assistant Professor.

ACKNOWLEDGEMENT

It was really a good experience, working with ***Digital Systems Design with FPGAs Laboratory*.** First, we would like to thank Dr. N. Srinivas, Assoc. Professor, HOD of Department of Electronics and Communication Engineering, Marri Laxman Reddy Institute of technology & Management for his concern and giving the technical support in preparing the document.

We are deeply indebted and gratefully acknowledge the constant support and valuable patronage of Dr. Ravi Prasad, Dean, Marri Laxman Reddy Institute of technology & Management for giving us this wonderful opportunity for preparing the *Digital Communications Laboratory* manual.

We express our hearty thanks to Dr. R.Murali prasad, Principal, Marri Laxman Reddy Institute of technology & Management, for timely corrections and scholarly guidance.

At last, but not the least I would like to thanks the entire ECE Department faculty those who had inspired and helped us to achieve our goal.

By,

Dr.NSrinivas(Associate Professor)
Mrs.B .Manjula (Assistant Professor)
Mrs.R.Babitha (Assistant Professor)

GENERAL INSTRUCTIONS

1. Students should report to the concerned labs as per the timetable schedule.

2. Students who turn up late to the labs will in no case be permitted to perform the experiment scheduled for the day.

3. After completion of the experiment, certification of the concerned staff in-charge in the observation book is necessary.

4. Students should bring a notebook of about 100 pages and should enter the readings/observations into the notebook while performing the experiment.

5. The record of observations along with the detailed experimental procedure of the experiment.

6. Performed in the immediate last session should be submitted and certified by the staff member in-charge.

7.. Not more than one student is permitted to perform the experiment on a setup.

8. When the experiment is completed, students should disconnect the setup made by them, and should return all the components/instruments taken for the purpose.

9. Any damage of the equipment or burnout of components will be viewed seriously by putting penalty.

10. Students should be present in the labs for the total scheduled duration.

11. Students are required to prepare thoroughly to perform the experiment before coming to Laboratory.

12. Procedure sheets/data sheets provided to the student's should be maintained neatly and to be returned after the experiment.

SAFETY PRECAUTIONS

1. No horseplay or running is allowed in the labs.

2. No bare feet or open sandals are permitted.

3. Before energizing any equipment, check whether anyone is in a position to be injured by your actions.

4. Read the appropriate equipment instruction manual sections or consult with your instructor.

5. Before applying power or connecting unfamiliar equipment or instruments into any circuits.

6. Position all equipment on benches in a safe and stable manner.

7. Do not make circuit connections by hand while circuits are energized. This is especially.

8. Dangerous with high voltage and current circuits.

INSTITUTION VISION AND MISSION

VISION

To establish as an ideal academic institution in the service of the nation, the world and the humanity by graduating talented engineers to be ethically strong, globally competent by conducting high quality research, developing breakthrough technologies, and disseminating and preserving technical knowledge.

MISSION

To fulfill the promised vision through the following strategic characteristics and aspirations:

A. Contemporary and rigorous educational experiences that develop the engineers and managers;

B. An atmosphere that facilitates personal commitment to the educational success of students in an environment that values diversity and community;

C. Prudent and accountable resource management;

D. Undergraduate programs that integrate global awareness, communication skills and team building;

E. Leadership and service to meet society's needs;

F. Education and research partnerships with colleges, universities, and industries to graduate education and training that prepares students for interdisciplinary engineering research and advanced problem-solving abilities;

G. Highly successful alumni who contribute to the profession in the global society.

DEPARTMENT VISION, MISSION, PROGRAMME EDUCATIONAL OBJECTIVES AND SPECIFIC OUTCOMES

Vision and Mission

Our Vision

Imparting quality technical education through research, innovation and team work for a lasting technology development in the area of Electronics and Communication Engineering.

Our Mission

To develop a strong center of excellence for education and research with excellent infrastructure and well qualified faculties to instill in them a passion for knowledge.

To achieve the Mission the department will:

M1: Establish a unique learning environment to enable the students to face the challenges of the Electronics and Communication Engineering field.

M2: Promote the establishment of center of excellence in niche technology areas to nurture the spirit of innovation and creativity among faculty and students.

M3: Provide ethical and value-based education by promoting activities addressing the societal needs.

M4: Enable students to develop skills to solve complex technological problems of current times and also provide a framework for promoting collaborative and multidisciplinary activities.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO 1: Have successful careers in Industry.

PEO 2: Show excellence in higher studies/ Research.

Program Outcomes (PO)

PO 1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and engg. specialization to the solution of complex engineering problems.

PO 2: Problem analysis: Identify, formulate, research literature, and analyze engineering problems to arrive at substantiated conclusions using first principles of mathematics, natural, and engineering sciences.

PO 3: Design/development of solutions: Design solutions for complex engineering problems and design system components, processes to meet the specifications with consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4: Conduct investigations of complex problems: Use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9: Individual and team work: Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.

PO 10: Communication: Communicate effectively with the engineering community and with society at large. Be able to comprehend and write effective reports documentation. Make effective presentations, and give and receive clear instructions.

PO 11: Project management and finance: Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team. Manage projects in multidisciplinary environments.

PO 12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE STRUCTURE

Level	Credits	Periods/Week	Prerequisites
PG	2	3	Entire subject of Digital System Design

Evaluation Scheme:

MID (Internal Lab) Semester Test	40 marks
End Semester Lab external Examination	60marks

The end semester examination shall be conducted with an external examiner and internal examiner.

The external examiner shall be appointed by the principal / Chief Controller of examinations

Course Objectives:

- The ability to code and simulate any digital function in Verilog HDL.
- Know the difference between synthesizable and non-synthesizable code.
- Understand library modeling, behavioral code and the differences between them.
- Understand the differences between simulator algorithms.
- Learn good coding techniques per current industrial practices.
- Understand logic verification using Verilog simulation.

Course Outcomes:

At the end of the laboratory work, students will be able to

- Describe Verilog hardware description languages (HDL).
- Design Digital Circuits in Verilog HDL.
- Write behavioral models of digital circuits.
- Write Register Transfer Level (RTL) models of digital circuits.

- Verify behavioral and RTL models.
- Describe standard cell libraries and FPGAs.
- Synthesize RTL models to standard cell libraries and FPGAs.
- Implement RTL models on FPGAs and Testing & Verification.Course Outcomes (COs)

At the end of the laboratory work, students will be able to

CO1: To Design Digital Circuits in Verilog HDL.

CO2: Write behavioral models of digital circuits.

CO3: Verify behavioral and RTL models.

CO4: To Synthesize RTL models to standard cell libraries and FPGAs.

CO5: To Implement RTL models on FPGAs and Testing & Verification.

Course Outcomes (CO's)–Program Outcomes (PO's)Mapping

CO's Po's	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	-	-	3	-	-	-	-	-	-	-
CO2	3	3	-	-	3	-	-	-	-	-	-	-
CO3	3	3	-	-	3	-	-	-	-	-	-	-
CO4	3	3	-	-	3	-	-	-	-	-	-	-
CO5	3	3	-	-	3	-	-	-	-	-	-	-

- **Simple-1** **Moderate-2** **High-3**

2215521: DIGITAL SYSTEM DESIGN WITH FPGAs LAB (Lab – I)

I Year M.Tech (ES) I – Sem.

L	T	P	C
0	0	3	2

Programming can be done using any complier. Down load the programs on FPGA/CPLD boards and performance testing may be done using pattern generator (32 channels) and logic analyzer apart fromverification by simulation with any of the front-end tools.

1. HDL code to realize all the logic gates
2. Design and Simulation of Full Adder, Serial Binary Adder, Multi Precession Adder, CarryLook Ahead Adder.
3. Design of Combinational circuit using Decoders.
4. Design of Combinational circuit using encoder (without and with parity).
5. Design of Combinational circuit using multiplexer.
6. Design of 4 bit binary to gray converter using MUX or Decoders.
7. Design of Multiplexer/ Demultiplexer, comparator in all 3 styles.
8. Modelling of an Edge triggered and Level triggered FFs : D, SR, JK
9. Design of 4-bit binary, BCD counters (synchronous/ asynchronous reset) or any sequencecounter
10. Design of a N- bit Register of Serial- in Serial -out, Serial in parallel out, Parallel inSerial out and Parallel in Parallel Out using different FFs.
11. Design of Sequence Detector (Finite State Machine- Mealy and Moore Machines).
12. Design of 4- Bit Multiplier, Divider.
13. Design of ALU to Perform - ADD, SUB, AND-OR, 1's and 2's Compliment,
14. Implementing the above designs on FPGA kits.

HDL CODE TO REALIZE ALL LOGIC GATES

EXPT. NO: 1

DATE:

AIM: To develop the source code for logic gates by using VERILOG and obtain thesimulation, Synthesis, place and route and implement into FPGA.

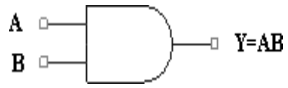
SOFTWARE & HARDWARE:

1. CADENCE
2. FPGA-SPARTAN-3

LOGIC DIAGRAM:

AND GATE:

LOGIC DIAGRAM:

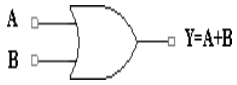


TRUTH TABLE:

A	B	Y=AB
0	0	0
0	1	0
1	0	0
1	1	1

OR GATE

LOGIC DIAGRAM

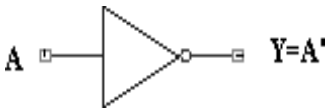


TRUTH TABLE

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOT GATE:

LOGIC DIAGRAM:

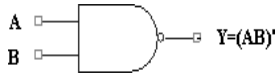


TRUTH TABLE:

A	Y=A’
0	0
1	0

NAND GATE:

LOGIC DIAGRAM:

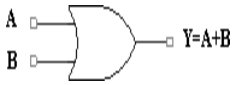


A	B	Y=(AB)’
0	0	1
0	1	1
1	0	1
1	1	0

TRUTH TABLE:

OR GATE

LOGIC DIAGRAM

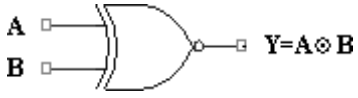


A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

TRUTH TABLE

XOR GATE

LOGIC DIAGRAM

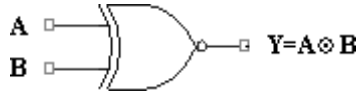


TRUTH TABLE:

A	B	Y=A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR GATE:

LOGIC DIAGRAM:



A	B	Y=A⊕B
0	0	1
0	1	0
1	0	0
1	1	1

TRUTH TABLE:

Description:

A logic gate is an idealized or physical device implementing a Boolean function, that is, it performs a logical operation on one or more logical inputs, and produces a single logical output. Depending on the context, the term may refer to an ideal logic gate, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device.

VERILOG SOURCE CODE:

DATA FLOW MODEL:

```
modulelogicgates1(a,b,c); input a;
input b; output [6:0]c;
assignc[0]=a&b;
assignc[1]=a|b;
assignc[2]=~(a&b);
assign c[3]=~(a|b);
assignc[4]=a^ b;
assignc[5]= ~(a ^ b);
assign c[6]=~ a;
endmodule
```

(OR)

```
module all(a,b,an,o,na,no,nt,xo,xn);
input a,b;
```

```
output (an,na,no,nt,xo,xn);
assign#5 an=a&b;
assign#5 na=~(a&b);
assign#5  o=a|b;
assign#5 no=~(a|b);
assign#5 nt=~a;
assign#5 xo=a^b;
assign#5 xn=~(a^b);
endmodule
```

BEHAVIOURAL MODELLING OF AND GATE:

```
module andg(a,b,an); input a,b;
output an; reg an;
always @(a or b) begin
if(a==1`b0 && b==1`b0) an=1`b0;
else if(a==1`b0 && b==1`b1) an=1`b0;
else if(a==1`b1 && b=1`b0) an=1`b0;
else if(a==1`b1 && b==1`b1) an=1`b1;
end
endmodule
```

BEHAVIOURAL MODEL FOR OR GATE,NOR,NAND,NOR,XOR,X-NOR:

```
module all(a,b,o,na,no,nt,xo,xn);
input a,b;
output(o,na,,o,nt,xo,xn);
reg o,na,no,nt,xo,xn;
always @(a or b)
begin
o=a|b;
na=~(a|b);
nt=~(a);
xo=(a^b);
xn=~(a^b);
endmodule
```

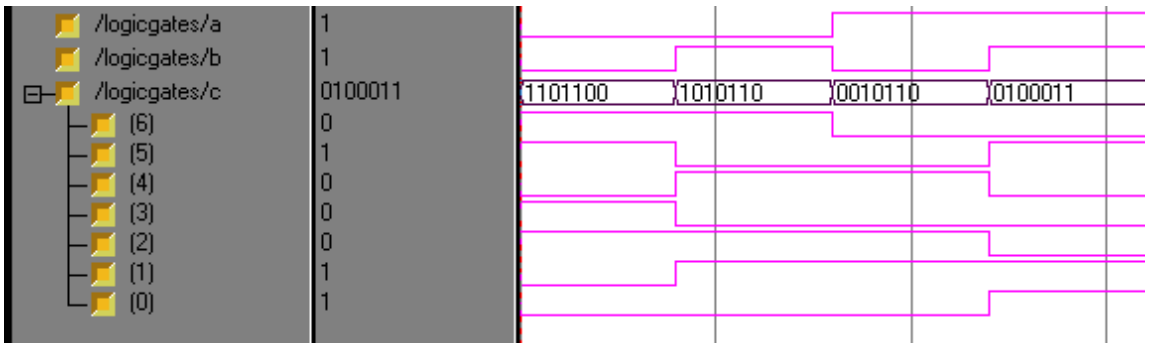
STRUCTURAL MODEL:

```
module all(a,b,an,o,na,no,nt,xo,xn);
input a,b;
output (an,o,na,no,nt,xo,xn);
wire t1,t2,t3,t4;
and a1,(an,a,b);
or a2(o,a,b);
not a3(nt,a);
and a4(t1,a,b);
not a5(na,t1);
or a6(t2,a,b);
not a7(no,t2);
xor a8(xo,a,b);
xor  a9(t3,a,b);
not a10(xn,t3);
endmodule
```

MIXED MODEL:

```
module all(a,b,an,o,na,no,nt,xo,xn);
input a,b;
output (an,o,na,no,nt,xo,xn);
reg na,no,nt,xo,xn;
assign #5 an=a&b;
assign #5 o=a|b: not a1(nt,a);
always @(a or b)
begin na=~(a&b);
no=~(a^b); xo=a^b;
xn=~(a^b);
end
endmodule
```

Simulation output:



Gates	A	B	Y
NOT	0	x	
	1	x	
AND	0	0	
	0	1	
	1	0	
	1	1	
OR	0	0	
	0	1	
	1	0	
	1	1	
NAND	0	0	
	0	1	
	1	0	
	1	1	
NOR	0	0	
	0	1	
	1	0	
	1	1	
XOR	0	0	
	0	1	
	1	0	
	1	1	
XNOR	0	0	
	0	1	
	1	0	
	1	1	

Design and Simulation of Full Adder, Serial Binary Adder, Multi Precession Adder, Carry Look Ahead Adder	EXPT. NO: 2
	DATE:

AIM: To write a HDL code to describe the functions of a full Adder Serial Binary Adder, Multi Precession Adder, Carry Look Ahead Adder.

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

A full adder consists of 3 inputs and 2 outputs. Fig 7.1 shows truth table of full adder. Use "assign" keyword to represent design in data flow style. The output signal expressions can be obtained from the truth table using K-maps.

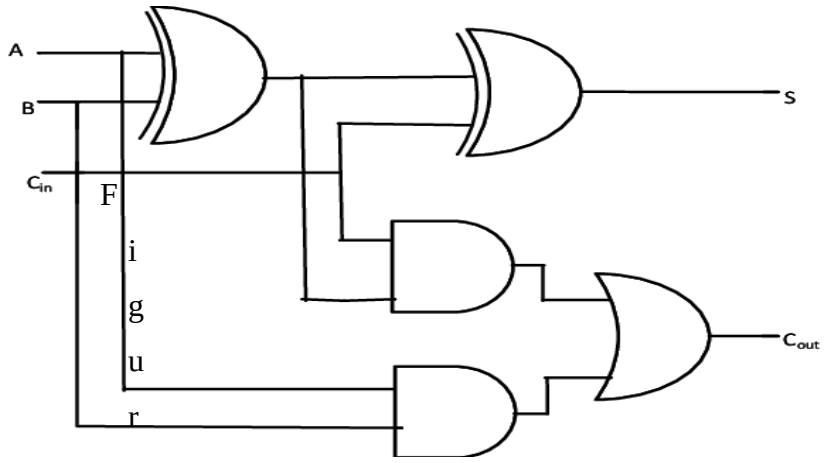


Figure 2.1 Logic diagram for 1-bit full adder

Table 2.1 Truth table for 1-bit full adder

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
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

PROCEDURE

- Create a module with required number of variables and mention it's input/output.
- Write the description of the full adder in 3 styles.
- Create another module referred as test bench to verify the functionality.
- Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
// full adder

module p10(a,b,c,sum,carry); output
sum,carry;
input a,b,c; wire
y0,y1,y2; xor
g1(y0,a,b);
and g2(y1,a,b);
xor g3(sum,y0,c);
and     g4(y2,y0,c);     or
g5(carry,y2,y1);
endmodule
```

To design and simulate the HDL code for carry look ahead adder

PROGRAM LOGIC

Ripple-carry addition suffers from an impractical propagation delay cause by the

sequential generation of arithmetic carries. In other words, c_{i+1} is dependent on c_i, which is further dependent on c_{i-1}, etc. The effect of this carry chain is a propagation delay that has a linear dependency on n, the bit width of the adder. Therefore, methods that compute the arithmetic carries in parallel have potential performance benefits over ripple-carry addition.

As the name implies, carry-look ahead is one such technique for high-speed addition that computes arithmetic carries in a parallel fashion. To understand how exactly a carry-look ahead adder works, consider the addition of two numbers, *X* and *Y*, such that x_i is the i^{th} ith binary digit of *X*, and y is t$_i$he ith binary digit of *Y*. The $(i+1)th$ arithmetic carry is c_{i+1} and is computed as follows:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \qquad (1)$$

$$= x_i y_i + (x_i + y_i) c_i \qquad (2)$$

The effect of simply factoring out ci from the last two terms in expression (1) is shown in expression (2). Now observe that c_{i+1} is logic '1' if either of the two conditions exists:

$x_i y_i$ *is logic '1'*

$x_i + y_i$ is logic '1'1 and there is a previous carry (i.e. $c_i = 1$)

Therefore, $x_i\ y_i$ is referred to as generate function because when '1', a carry is generated, while $x_i + y_i$ is referred to as the propagate function because when '1', it will propagate a carry. In mathematical terms, we see that

$$g_i = x_i y_i \qquad (3)$$

$$p_i = x_i + y_i \qquad (4)$$

$$c_{i+1} = g_i + p_i c_i \qquad (5)$$

Clearly, expressions (3) and (4) do not depend on the carry in the previous bit position and thus, can be generated in parallel. It turns out, we can write expression (5) for the first four carries in such a way that they,

too, do not depend on one another, but rather only depend on the input carry, c0, and the gi and pi. Examine the expressions below to convince yourself of this.

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

Although the expression for ci becomes increasingly complex, the theoretical gate-delay for each of the above expressions, given the gi's, pi's, and $c0$, is $\Delta g =$ 2. However, the increased complexity is reflected in the number of inputs to each gate (i.e. the gate fan-in) and the number of gates required. Figure 1 illustrates this point with the gate-level schematic for each of the sub-modules within a 4-bit carry-lookahead adder. One thing to note is that:

$$p_i = x_i \oplus y_i$$

$$s_i = p_i \oplus c_i$$

In other words, expression (10) is being used in lieu expression (4). It turns out that expression (5) works correctly in either case, and the former allows the Sum to be computed with expression (11). Before moving on, let us try to understand how data flows through the 4-bit carry-lookahead adder. To do so, we enumerate through the steps below:

Data arrives at the Generate/Propagate Unit, and the g_i's and p_i's, are computed in one gate-delay (i.e.$\Delta g = 1$).

The gi's and pi's are forwarded to the Carry-Lookahead Unit, which generates all of the carries in two gate-delays, $\Delta g = 2$.

The carries are then fed into the Summation Unit, which computes the sum bits, the s_i's, in one gate-delay $\Delta g = 1$.

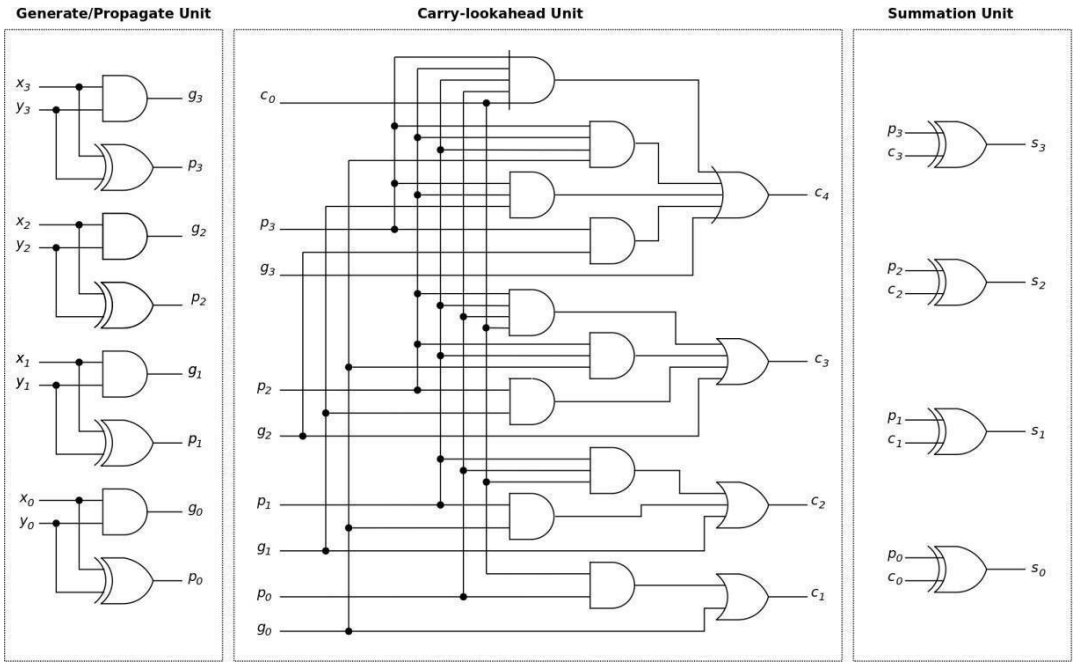


Figure 2.2 Carry-look ahead Adder

For simplicity, we are assuming that all gates have the same delay time. This assumption may or may not be true depending on the target technology that is being used to implement your logic. However, for the sake of comparison with other addition techniques, this model works well. Summarizing the above steps, we can see that the propagation delay for a 4-bit adder is no longer determined by a carry chain and is only four gate-delays, ($\Delta g = 4$). The pre-lab assignment will include an exercise which asks you to look at the gate count of a 4-bit Carry- Look ahead Adder.

PROCEDURE

Create a module with required number of variables and mention it's input/output.

Write the description of the carry look ahead adder using data flow model or gate level model.

Create another module referred as test bench to verify the functionality.

Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
module p21(a,b,cin,sum,cout);
input[3:0] a,b;
```

```
 input cin;
 output [3:0] sum; output cout;
wire p0,p1,p2,p3,g0,g1,g2,g3,c1,c2,c3,c4;
assign p0=(a[0]^b[0]),
        p1=(a[1]^b[1]),
        p2=(a[2]^b[2]),
        p3=(a[3]^b[3]);
        assign g0=(a[0]&b[0]),
        g1=(a[1]&b[1]),
       g2=(a[2]&b[2]),
        g3=(a[3]&b[3]);
                    assign c0=cin, c1=g0|(p0&cin),
                    c2=g1|(p1&g0)|(p1&p0&cin),
                    c3=g2|(p2&g1)|(p2&p1&g0)|(p1&p1&p0&cin),
                    c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&cin); assign
                    sum[0]=p0^c0,
                    sum[1]=p1^c1,
                    sum[2]=p2^c2,
                    sum[3]=p3^c3;
                    assign cout=c4;
          endmodule
```

PRE LAB QUESTIONS

1. What is the functionality of the adder?
2. Design a ripple carry adder and mention its disadvantage.
3. List the various adders and its pros and cons.
4. What is a half adder?
5. Write the sum and carry expression for half adder.
6. What is a full adder?
7. Write the sum and carry expression for 1-bit full adder.
8. Write the difference and barrow out expressions for 1-bit subtractor
9. What is a parallel adder/subtractor?

LAB ASSIGNMENT

1. Design 4-bit ripple carry adder using HDL.
2. Design 4-bit carry look ahead adder using HDL.
3. Observe the RTL schematic of the designed 4-bit look ahead adder.
4. Design a 4-bit ripple carry adder using full adders.
5. Implement full adder using decoder.
6. Implement full subtractor using decoder.
7. Implement a 4-bit adder/subtractor.
8. Design a full adder using minimum number of NAND gates.

POST LAB QUESTIONS

1 Realize a full adder using two half adders.

2 What is the amount of delay involved in ripple carry adder?

3 Compare serial adder and parallel adder with respect to speed and complexity.

4 Implement a single circuit which can perform both addition and subtraction operations on binary input bits.

Design of Combinational circuit using Decoders.	EXPT. NO: 3
	DATE:

AIM:

To design and simulate the HDL code for the following combinational circuits

3 to 8 Decoder

RESOURCES

PC installed with CADENCE tool

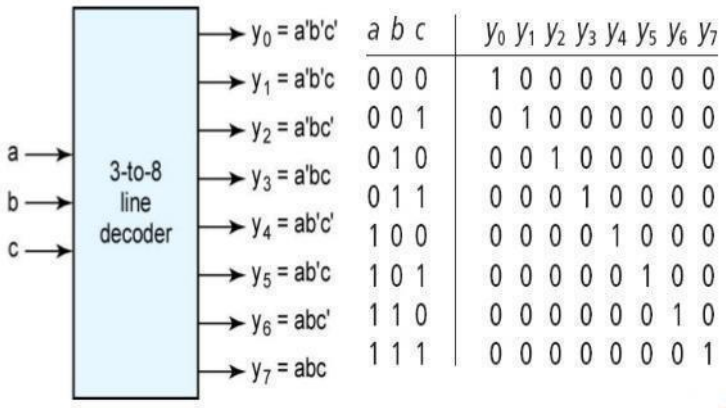
PROGRAM LOGIC

Program logic for Decoder

A decoder is a multiple-input, multiple-output logic circuit which converts coded inputs into coded outputs, where the 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 from input code words into output code words. This one-to-one mapping can be expressed in a truth table.

The most common decoder circuit is an n-to-2^n decoder or binary decoder. Such a decoder has an n-bit binary input code and a 1-out-of-2^n output code. A binary decoder is used when you need to activate exactly one of 2^noutputs based on an n- bit input value.

Figure 3.1 shows the general structure of the 3 to 8 decoder circuit and its truth table.



a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 3.1: General Structure of 3 to 8 Decoder and its truth table

CODE:

```
// 3 to 8 decoder

module
p3(i,d);
input
[2:0]i;
output [7:0]d;
assign d[0]=(~i[2])&(~i[1])&(~i[0]);
assign d[1]=(~i[2])&(~i[1])&(i[0]);
assign d[2]=(~i[2])&(i[1])&(~i[0]);
assign d[3]=(~i[2])&(i[1])&(i[0]);
assign d[4]=(i[2])&(~i[1])&(~i[0]);
assign d[5]=(i[2])&(~i[1])&(i[0]);
assign d[6]=(i[2])&(i[1])&(~i[0]);
assign
d[7]=(i[2])&(i[1])&(i[0]);
endmodule
```

PRE LAB QUESTIONS

1 What is a decoder?

2 What for enable inputs are used in decoder?

LAB ASSIGNMENT

1. Implement full adder circuit using decoder and two OR gates.
2. Implement 3x8 decoder using 2x4 decoder and additional logic.
3. Construct a 4x16 decoder using two 3x8 decoder and additional logic. Show the schematic diagram neatly?
4. Design 2-to-4 decoder using only NOR gates.
5. Construct a 5 x 32 decoder with four 3x 8 decoders with enable and one 2 x 4 decoder.

Design of Combinational circuit using encoder (without and with parity).	EXPT. NO: 4
	DATE:

AIM: To design and simulate the HDL code for the following combinational circuits 8 to 3 Encoder (With priority and without priority).

RESOURCES

PC installed with CADENCE tool.

Program logic for Encoder

An encoder has M input and N output lines. Out of M input lines only one is activated at a time and produces equivalent code on output N lines. If a device output code has fewer bits than the input code has, the device is usually called an encoder. Example Octal-to-Binary take 8 inputs and provides 3 outputs. For an 8- to-3 binary encoder with inputs D0-D7 the logic expressions of the outputs XYZ are obtained by using the Table 4.1.

$$X = D4 + D5 + D6 + D7$$

$$Y = D2 + D3 + D6 + D7$$

$$Z = D1 + D3 + D5 + D7$$

Table 4

.1: Truth Table for 8-3 Encoder wi

t
h

D
7
-
D
0

i
n
p
u
t
s

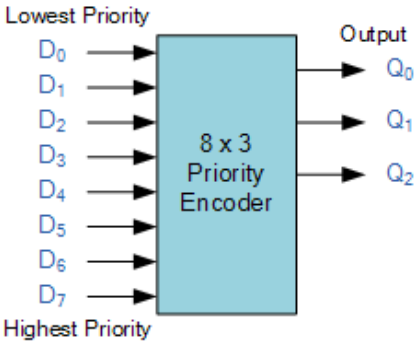
INPUTS								OUTPUTS		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	O 0	0	0	0	0	1	0	0	0	1
0	n 0	0	0	0	1	0	0	0	1	0
0	e 0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	o 0	1	0	0	0	0	0	1	0	1
0	f 1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

t

he main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level "1".

The Priority Encoder solves the problems mentioned above by allocating a priority level to each input. The priority encoders output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. The priority encoder comes in many different forms with an example of an 8- input priority encoder along with its

truth table shown in Figure 4.1



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

PROCEDURE

1. Create a module with required number of variables and mention it's input/output.
2. Implement the logic for decoder or encoder using behavioral or gate level model.
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained output with the corresponding truth table.

CODE

```
// 8 to 3 Encoder without priority module
p2(d,e);
input [7:0] d;
output [2:0]e;
assign e[2]= d[4] | d[5] | d[6] | d[7];
assign e[1]= d[2] | d[3] | d[6] | d[7];
assign e[0]= d[1] | d[3] | d[5] | d[7];
endmodule

// 8 to 3 Encoder with priority module
p4(din, dout);
input [7:0] din;
output [2:0] dout;
reg [2:0] dout;
```

```
always @(din)
begin
if (din[7]==1'b1) dout=3'b111;
else if (din[6]==1'b1) dout=3'b110; else
if (din[5]==1'b1) dout=3'b101; else if
(din[4]==1'b1) dout=3'b100; else if
(din[3]==1'b1) dout=3'b011; else if
(din[2]==1'b1) dout=3'b010; else if
(din[1]==1'b1) dout=3'b001; else if
(din[0]==1'b1) dout=3'b000; else
dout=3'bXXX;
end
endmodule
```

PRE LAB QUESTIONS

1. What is an encoder?
2. What is a priority encoder?
3. How many input and output lines are there for a 128x7 encoder.

LAB ASSIGNMENT

1. Write a Verilog code to implement Octal-to-Binary Encoder?
2. Write a Verilog code to implement a 8x3 Priority Encoder?
3. Write a Verilog code to implement Decimal-to-BCD Encoder?

POST LAB QUESTIONS

1. Write code for a parallel encoder and a priority encoder.
2. What is the difference between wire and reg data type ?
3. What is the difference between the following two lines of Verilog code? #5 a = b;
   a = #5 b;
4. What is the use of Priority Encoder?

Design of Combinational circuit using multiplexer.	EXPT. NO: 5
	DATE:

AIM: To write HDL codes for an 8X1 multiplexer r and verify its functionality.

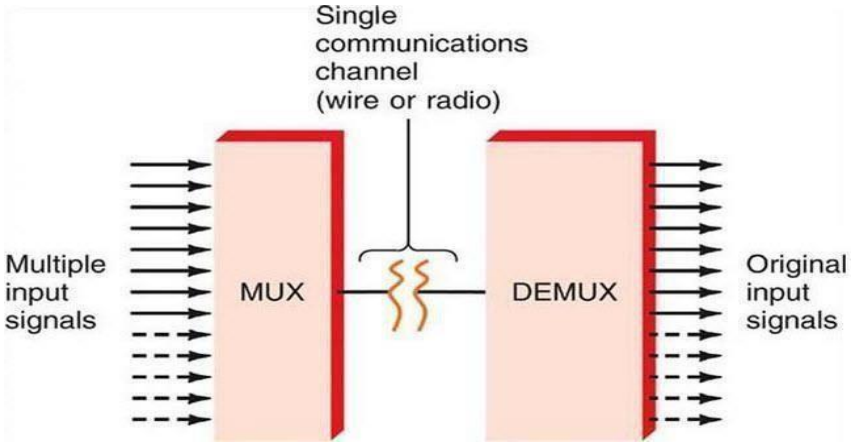
RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

In the large-scale-digital systems, a single line is required to carry on two or more digital signals – and, of course! At a time, one signal can be placed on the one line. But, what is required is a device that will allow us to select; and, the signal we wish to place on a common line, such a circuit is referred to as multiplexer.

The function of a multiplexer is to select the input of any 'n' input lines and feed that to one output line. The function of a de-multiplexer is to inverse the function of the multiplexer and the shortcut forms of the multiplexer. The de-multiplexers are mux and demux. Some multiplexers perform both multiplexing and de-multiplexing operations. The main function of the multiplexer is that it combines input signals, allows data compression,



and shares a single transmission channel.

Figure 5.1 Multiplexer and De-multiplexer

The output value of a 8x1 multiplexer can be represented using the equation (5.1)

$$Y = S_2 S_1 S_0 I_o + S_2 S_1 S_0 I_1 + S_2 S_1 S_0 I_2 + S_2 S_1 S_0 I_3 + S_2 S_1 S_0 I_4 + S_2 S_1 S_0 I_5 + S_2 S_1 S_0 I_6 + S_2 S_1 S_0 I_7$$

… (5.1)

PROCEDURE

1. Create a module with required number of variables and mention it's input/output.
2. Write the description of the multiplexer or demultiplexer using data flow model or gate level model
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained output with the corresponding truth table.

CODE

```
// 8:1 multiplexer module
p5(s,i,y);
input [7:0]i;
input [2:0]s;
output y; wire
[2:0]sb;
not(sb[0],s[0]);
not(sb[1],s[1]);
not(sb[2],s[2]);
assign y = (sb[2]&sb[1]&sb[0]&i[0]) | (sb[2]&sb[1]&s[0]&i[1]) |
(sb[2]&s[1]&sb[0]&i[2]) | (sb[2]&s[1]&s[0]&i[3]) | (s[2]&sb[1]&sb[0]&i[4]) |
(s[2]&sb[1]&s[0]&i[5]) | (s[2]&s[1]&sb[0]&i[6]) | (s[2]&s[1]&s[0]&i[7]);
endmodule
```

PRE LAB QUESTIONS

1. What is a multiplexer?
2. What is the relationship between input lines and select lines?
3. Why a multiplexer is called a data selector?
4. Mention the applications of multiplexer and demultiplexer.

LAB ASSIGNMENT

5 Implement a full adder with two 4x1 multiplexers.

6 Implement 2 to 4 decoder using 1x4 demultiplexer.

7 Implement a full subtractor with two 4x1 multiplexers.

8 Realize 8x1 mux using 4x1 multiplexer.

9 Implement half adder using 2x1 multiplexer.

10 $F(W, X, Y, Z) = \Pi_m(0,1,3,5,7)$ using 8x1 multiplexer.

11 Write code for 1x4 Multiplexer using different coding methods.

POST LAB QUESTIONS

12 Can a multiplexer be used to realize a logic function?

13 Differentiate between decoder and demultiplexer.

14 What are the applications of multiplexers?

15 Design an OR gate from 2:1 MUX.

16 Design a D and T flip flop using 2:1 multiplexer

17Implement the function f(A,B,C)= Σm (0,1,3,5,7) by using multiplexer.

DESIGN OF CODE CONVERTERS	EXPT. NO : 06
	DATE:

AIM: To Design and simulate the HDL code for the following combinational circuits

4 - Bit binary to gray code converter

4 - Bit gray to binary code converter

Comparator

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

Binary to gray code converter logic

This conversion method strongly follows the EX-OR gate operation between binary bits. The steps to perform binary to grey code conversion are given bellow.

To convert binary to grey code, bring down the most significant digit of the given binary number, because, the first digit or most significant digit of the grey code number is same as the binary number.

To obtain the successive grey coded bits to produce the equivalent grey coded number for the given binary, add the first bit or the most significant digit of binary to the second one and write down the result next to the first bit of grey code, add the second binary bit to third one and write down the result next to the second bit of grey code, follow this operation until the last binary bit and write down the results based on EX-OR logic to produce the equivalent grey coded binary.

Gray to binary code converter logic

This conversion method also follows the EX-OR gate operation between grey & binary bits. The steps to perform grey code to binary conversion are given below.

To convert grey code to binary, bring down the most significant digit of the given grey code number, because, the first digit or the most significant digit of the grey code number is same as the binary number.

To obtain the successive second binary bit, perform the EX-OR operation between the first bit or most significant digit of binary to the second bit of the given grey code.

To obtain the successive third binary bit, perform the EX-OR operation between the second bit or most significant digit of binary to the third MSD (most significant digit) of grey code and so on for the next successive binary bits conversion to find the equivalent.

PROCEDURE

Create a module with required number of variables and mention it's input/output.

Write the description of the code converter using data flow model or gate level model.

Create another module referred as test bench to verify the functionality.

Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
// binary to gray code converter module
p7(b,g);
input [3:0] b;
output [3:0] g;
reg [3:0] g;
always@(b)
begin g[3]=b[3];
g[2]=b[3]^b[2];
g[1]=b[2]^b[1];
```

```
g[0]=b[1]^b[0];
end endmodule
//gray to binary converter
module p8(g,b);
input [3:0] g;
output [3:0] b;
reg [3:0] b;
always@(g)
begin b[3]=g[3];
b[2]=b[3]^g[2];
b[1]=b[2]^g[1];
b[0]=b[1]^g[0];
end endmodule

// 4 bit comparator

module p9(a,b,g,l,e); input
[3:0]a;
input [3:0]b;
output g,l,e; reg
g,l,e; always@(a,b)
begin
if (a<b) begin
e = 0;  l = 1;  g = 0;
end
else if (a==b)
begin
e = 1;  l = 0;  g = 0;
end else
begin
e = 0;  l = 0;  g = 1;
end end
endmodule
```

PRE LAB QUESTIONS

1. What is a code converter? List some of the code converters.
2. What are the typical applications of gray code?
3. Distinguish between the weighted and non-weighted codes. Give examples.
4. Realize the Boolean expressions for binary to gray code conversion
5. Realize the Boolean expressions for gray to binary code conversion

LAB ASSIGNMENT

1. Design BCD to Excess-3 code converter.
2. Design a BCD to seven segment code converter.
3. Design octal to binary code converter.

POST LAB QUESTIONS

1. What is the difference between blocking and non-blocking assignments?

2. What is the difference between case x and case statements?

3. What is this `timescale compiler directive?

4. What is sensitivity list?

DESIGN OF MULTIPLEXER AND DEMULTIPLEXER	EXPT. NO : 07
	DATE:

AIM: To write HDL codes for an 8X1 multiplexer and 1X8 demultiplexer and verify its functionality.

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

In the large-scale-digital systems, a single line is required to carry on two or more digital signals – and, of course! At a time, one signal can be placed on the one line. But, what is required is a device that will allow us to select; and, the signal we wish to place on a common line, such a circuit is referred to as multiplexer.

The function of a multiplexer is to select the input of any 'n' input lines and feed that to one output line. The function of a de-multiplexer is to inverse the function of the multiplexer and the shortcut forms of the multiplexer. The de-multiplexers are mux and demux. Some multiplexers perform both multiplexing and de-multiplexing operations. The main function of the multiplexer is that it combines input signals, allows data compression, and shares a single transmission channel.

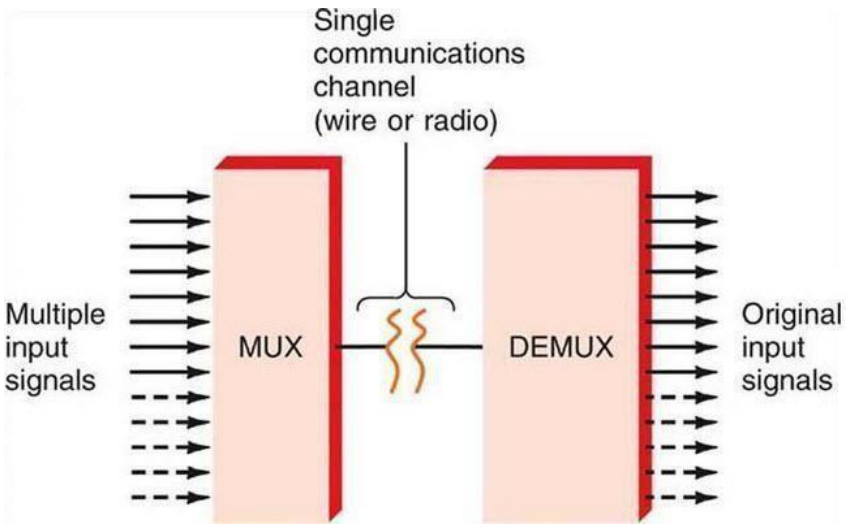


Figure 7.1 Multiplexer and De-multiplexer

The output value of a 8x1 multiplexer can be represented using the equation (7.1)

$$Y = S_2 S_1 S_0 I_o + S_2 S_1 S_0 I_1 + S_2 S_1 S_0 I_2 + S_2 S_1 S_0 I_3 + S_2 S_1 S_0 I_4 + S_2 S_1 S_0 I_5 + S_2 S_1 S_0 I_6 + S_2 S_1 S_0 I_7 \quad \ldots (7.1)$$

For the combination of selection input, the data line is connected to the output line. The 8x1 multiplexer requires 8 AND gates, one OR gate and 3 selection lines. As an input, the combination of selection inputs are giving to the AND gate with the corresponding input data lines.

In a similar fashion, all the AND gates are given connection. In this 8x1 multiplexer, for any selection line input, one AND gate gives a value of 1 and the remaining all AND gates give 0. And, finally, by using OR gate, all the AND gates are added; and, this will be equal to the selected value.

The demultiplexer is also called as data distributors as it requires one input, 3 selected lines and 8 outputs. De-multiplexer takes one single input data line, and then switches it to any one of the output line. 1-to-8 demultiplexer circuit diagram is shown below; it uses 8 AND gates for achieving the operation. The input bit is considered as data D and it is transmitted to the output lines.

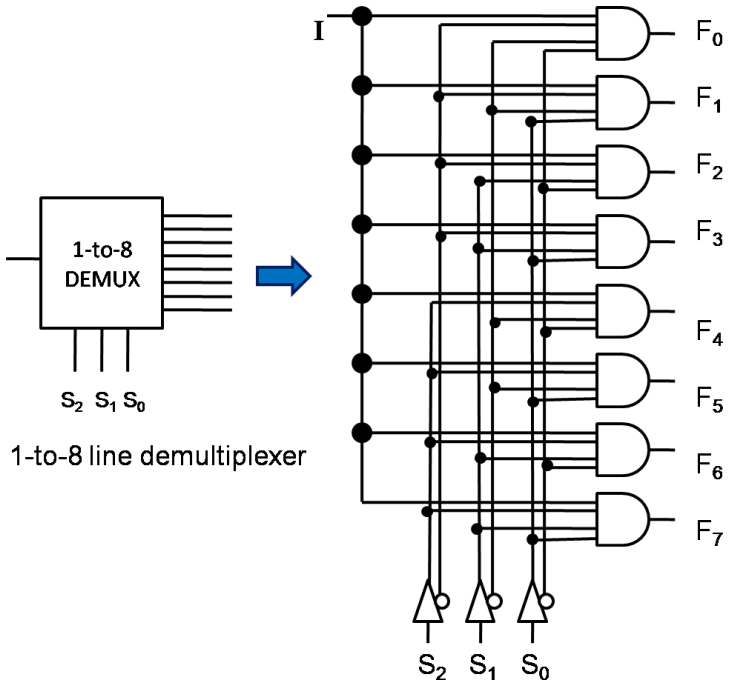


Figure 7.2 Demultiplexer circuit diagram

PROCEDURE

Create a module with required number of variables and mention it's input/output.

Write the description of the multiplexer or demultiplexer using data flow model or gate level model

Create another module referred as test bench to verify the functionality.

Follow the steps required to simulate the design and compare the obtained output with the corresponding truth table.

CODE

```
// 8:1
multiplexer
module
p5(s,i,y);
                                    input [7:0]i;
i
n
p
u
t
```

```
[
2
:
0
]
s
;
o
u
t
p
u
t
y
;
w
i
r
e
[
2
:
0
]
s
b
;
not(sb[0],s[0]);
not(sb[1],s[1]);
not(sb[2],s[2]);
assign y = (sb[2]&sb[1]&sb[0]&i[0]) | (sb[2]&sb[1]&s[0]&i[1]) |
(sb[2]&s[1]&sb[0]&i[2]) | (sb[2]&s[1]&s[0]&i[3]) |
(s[2]&sb[1]&sb[0]&i[4]) |(s[2]&sb[1]&s[0]&i[5]) |
(s[2]&s[1]&sb[0]&i[6]) | (s[2]&s[1]&s[0]&i[7]); endmodule
```

//1:8 Demultiplexer

```
module
p6(din,s,do
ut); output
[7:0]dout ;

input din ;
input [2:0]s ;
assign dout[7] = din & (s[2]) & (s[1]) & (s[0]);
assign dout[6] = din & (s[2]) & (s[1]) & (~s[0]);
assign dout[5] = din & (s[2]) & (~s[1]) & (s[0]);
assign dout[4] = din & (s[2]) & (~s[1]) & (~s[0]);
assign dout[3] = din & (~s[2]) & (s[1]) & (s[0]);
assign dout[2] = din & (~s[2]) & (s[1]) & (~s[0]);
assign dout[1] = din & (~s[2]) & (~s[1]) & (s[0]);
assign dout[0] = din & (~s[2]) & (~s[1]) &
(~s[0]); endmodule
```

PRE LAB QUESTIONS

1. What is a multiplexer?
2. What is the relationship between input lines and select lines?
3. Why a multiplexer is called a data selector?
4. Mention the applications of multiplexer and demultiplexer.

LAB ASSIGNMENT

1. Implement a full adder with two 4x1 multiplexers.
2. Implement 2 to 4 decoder using 1x4 demultiplexer.
3. Implement a full subtractor with two 4x1 multiplexers.
4. Realize 8x1 mux using 4x1 multiplexer.
5. Implement half adder using 2x1 multiplexer.
6. $F(W, X, Y, Z) = \Pi_m(0,1,3,5,7)$ using 8x1 multiplexer.

7. Write code for 1x4 Multiplexer using different coding methods.

POST LAB QUESTIONS

1. Can a multiplexer be used to realize a logic function?
2. Differentiate between decoder and demultiplexer.
3. What are the applications of multiplexers?
4. Design an OR gate from 2:1 MUX.
5. Design a D and T flip flop using 2:1 multiplexer
6. Implement the function $f(A,B,C)= \Sigma m (0,1,3,5,7)$ by using multiplexer.

Modelling of an Edge triggered and Level triggered FFs : D, SR, JK	EXPT. NO : 08
	DATE:

AIM: To write HDL codes for SR, JK, D, T flip flops and verify its functionality.

RESOURCES

PC installed with Xilinx tool

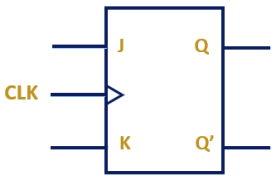
PROGRAM LOGIC

Each flip-flop stores a single bit of data, which is emitted through the Q output on the output section side. Normally, the value can be controlled via the inputs to the input side. In particular, the value changes when the clock input, marked by a triangle on each flip-flop, rises from 0 to 1 (or otherwise as configured); on this rising edge, the value changes according to the tables below.

Table 7.1 Truth tables of D, T, SR, JK flip flops

J K Flip Flop

Symbol



Truth Table

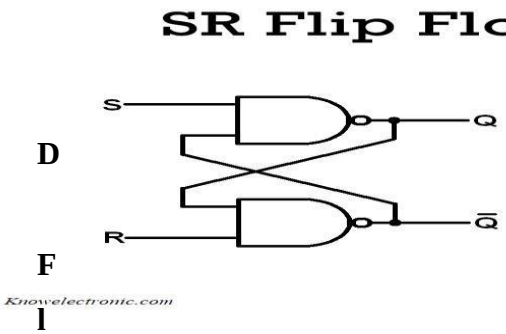
CLK	J	K	Q n+1
↑	0	0	Q n
↑	0	1	0
↑	1	0	1
↑	1	1	Q n'

S R Flip Flop

D

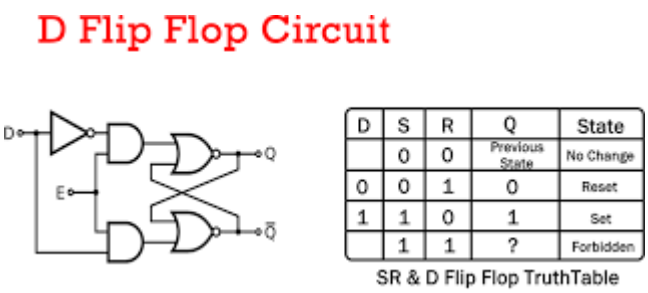
F

l



Sno	S	R	Q	Q'	State
1	1	0	1	0	Q is set to 1
2	1	1	1	0	No change
3	0	1	0	1	Q' is set to 1
4	1	1	0	1	No change
5	0	0	1	1	Invalid

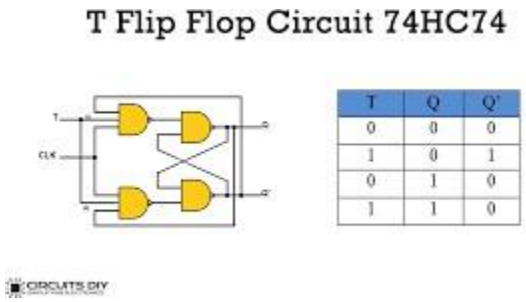
ip Flop.



D	S	R	Q	State
	0	0	Previous State	No Change
0	0	1	0	Reset
1	1	0	1	Set
	1	1	?	Forbidden

SR & D Flip Flop TruthTable

T Flip Flop



T	Q	Q'
0	0	0
1	0	1
0	1	0
1	1	0

Another way of describing the different behavior of the flip-flops is in English text.

D Flip-Flop: When the clock triggers, the value remembered by the flip-flop becomes the value of the D input (Data) at that instant.

T Flip-Flop: When the clock triggers, the value remembered by the flip-flop either toggles or remains the same depending on whether the T input (Toggle) is 1 or 0.

J-K Flip-Flop: When the clock triggers, the value remembered by the flip-flop toggles if the J and K inputs are both 1, remains the same if they are both 0; if they are different, then the value becomes 1 if the J (Jump) input is 1 and 0 if the K (Kill) input is 1.

S-R Flip-Flop: When the clock triggers, the value remembered by the flip-flop remains unchanged if R and S are both 0, becomes 0 if the R input (Reset) is 1, and becomes 1 if the S input (Set) is 1. The behavior in unspecified if both inputs are 1.

PROCEDURE

- Create a module with required number of variables and mention it's input/output.
- Write the description of the flip flops using behavioral model
- Create another module referred as test bench to verify the functionality.
- Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

//SR flipflop

```
module p14(s,r,clk,q,qb);
input s,r,clk;
output q,qb; reg
q,qb; reg [1:0]sr;
wire qp=1'b0;
always@(posedge clk) begin
sr={s,r};
begin case
(sr)
2'd0:q=qp;
2'd1:q=1'b0;
2'd2:q=1'b1;
2'd3:q=1'bX;
endcase end
qb=~q;  end
endmodule
```

//JK flipflop

```
module p15(j,k,clk,q,qb);
input j,k,clk;
output q,qb; reg
q,qb; reg [1:0]jk;
wire qp=1'b0;
always@(posedge clk) begin
```

```
jk={j,k};
begin case (jk)
2'd0:q=qp;
2'd1:q=1'b0;
2'd2:q=1'b1;
2'd3:q=~q; endcase
end  qb=~q;  end
endmodule
```

//D flipflop

```
module p16(q,din,clk);
output q;
reg q; input din
; wire din ;
input clk ;
always @ (posedge (clk))
begin q = din ;
end endmodule
```

//T flipflop

```
module p17(q,t,clk);
output q;
reg q; input t ;
input clk ;
always @ (posedge (clk))
begin
q = ~t; end
endmodule
```

PRE LAB QUESTIONS

1 Distinguish between latch and edge triggered flip-flop?

2 What is the cause for the race around phenomenon in a J - K flip-flop?

3 What is meant by triggering of a flip-flop?

4 What do you mean by clock skew?

5 What is master-slave flip-flop?

LAB ASSIGNMENT

6 Convert a given J-K flip-flop in to a D flip-flop using additional logic if necessary?

7 Convert a given J-K flip-flop in to a T flip-flop using additional logic if necessary?

8 Convert a given D flip-flop in to a T flip-flop using additional logic if necessary?

9 Implement an asynchronous reset JK FF.

POST LAB QUESTIONS

10 What is use of characteristic and excitation table?

11 How is a JK flip flop made to toggle?

12 Differentiate between combinational and sequential circuits.

Design of 4-bit binary, BCD counters (synchronous/ asynchronous reset) or any sequence counter	EXPT. NO : 9
	DATE:

AIM:To write HDL codes for the following counters.

Binary counter

BCD counter (Synchronous reset and asynchronous reset)

RESOURCES

PC installed with CAEDENCE tool

PROGRAM LOGIC

Counter is a sequential circuit. A digital circuit which is used for counting pulses is known as counter. Counter is the widest application of flip-flops. It is a group of flip- flops with a clock signal applied. Counters are of two types.

Asynchronous or ripple counters.

Synchronous counters.

Asynchronous counters are called as ripple counters, the first flip-flop is clocked by the external clock pulse and then each successive flip-flop is clocked by the output of the preceding flip-flop. The term asynchronous refers to events that do not have a fixed time relationship with each other. An asynchronous counter is one in which the flip-flops within the counter do not change states at exactly the same time because they do not have a common clock pulse

In synchronous counters, the clock inputs of all the flip-flops are connected together and are triggered by the input pulses. Thus, all the flip-flops change state simultaneously (in parallel).

A counter is a register capable of counting the number of clock pulses arriving at its clock input. Count represents the number clock pulses arrived. A specified sequence of states appears as the counter output. The name counter is generally used for clocked sequential circuit whose state diagram contains a single cycle. The modulus of a counter is the number of states in the cycle. A counter with m states is called a modulo-m counter or divide-by-m counter. A counter with a non-power-of-2 modulus has extra states that are not used in normal

operation. There are two types of counters, synchronous and asynchronous. In synchronous counter, the common clock is connected to all the flip-flops and thus they are clocked simultaneously.

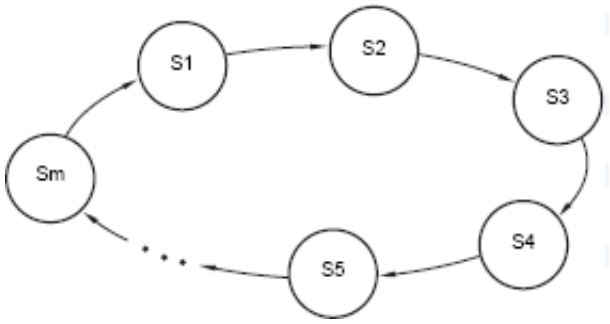


Fig. 9.1 General structure of a counter's state diagram – a single cycle

Asynchronous Decade Counters

The *modulus* is the number of unique states through which the counter will sequence. The maximum possible number of states of a counter is $2^n$where n is the number of flip-flops. Counters can be designed to have a number of states in their sequence that is less than the maximum of 2^n. This type of sequence is called a truncated sequence. One common modulus for counters with truncated sequences is 10 (Modules10). A decade counter with a count sequence of zero (0000) through 9 (1001) is a BCD decade counter because its 10-state sequence produces the BCD code. To obtain a truncated sequence, it is necessary to force the counter to recycle before going through all of its possible states. A decade counter requires 4flip-flops. One way to make the counter recycle after the count of 9 (1001) is to decode count 10 (1010) with a NAND gate and connect the output of the NAND gate to the clear (CLR) inputs of the flip-flops, as shown in Figure 9.1

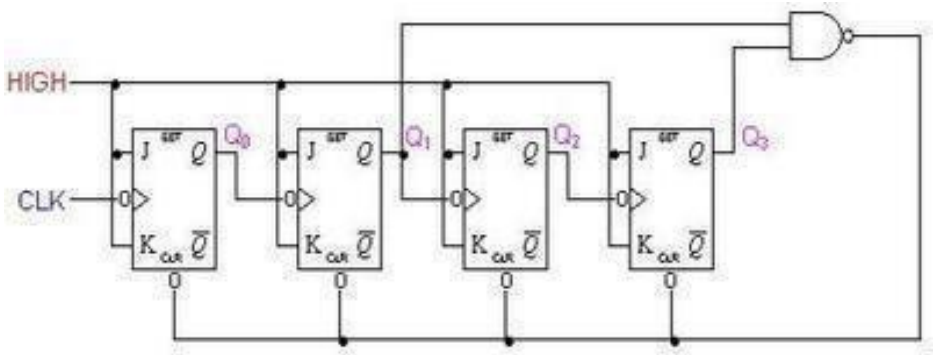


Figure 9.2 Asynchronous Decade Counter

Synchronous Decade Counters

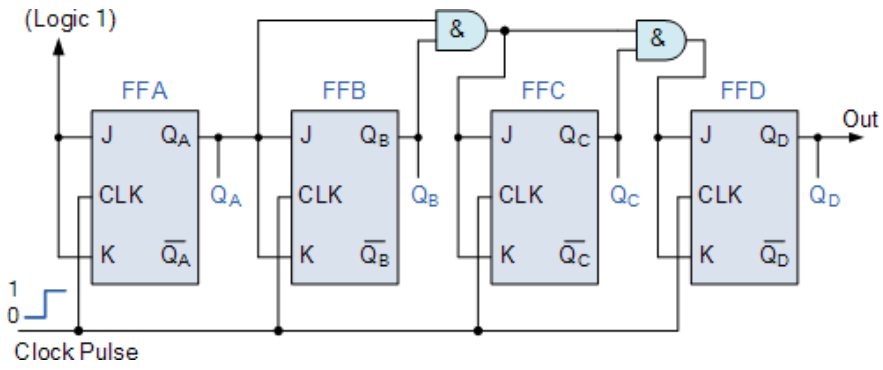


Figure 9.3 Asynchronous Decade Counter

It can be seen from Figure 8.2, that the external clock pulses (pulses to be counted) are fed directly to each of the J-K flip-flops in the counter chain and that both the J and K inputs are all tied together in toggle mode, but only in the first flip-flop, flip- flop FFA (LSB) are they connected HIGH, logic "1" allowing the flip-flop to toggle on every clock pulse. Then the synchronous counter follows a predetermined sequence of states in response to the common clock signal, advancing one state for each pulse.

The J and K inputs of flip-flop FFB are connected directly to the output QA of flip- flop FFA, but the J and K inputs of flip-flops FFC and FFD are driven from separate AND gates which are also supplied with signals from the input and output of the previous stage. These additional AND gates generate the required logic for the JK inputs of the next stage.

If we enable each JK flip-flop to toggle based on whether or not all preceding flip- flop outputs (Q) are "HIGH" we can obtain the same counting sequence as with the asynchronous circuit but without the ripple effect, since each flip-flop in this circuit will be clocked at exactly the same time.

PROCEDURE

1. Create a module with required number of variables and mention it's input/output.
2. Write the description of the counter to count required number of states and to satisfy its conditions.
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
// binary counter

module p18(clk,count );
output [3:0] count ;
reg [3:0] count ; input
clk ;
wire clk ;
initial count = 0;
always @ (posedge (clk))
begin
count <= count + 1; end
endmodule

//BCD counter

module p19(clk ,reset ,dout ); output
[3:0] dout ;
 reg [3:0] dout ; input
clk ;
wire clk ; input
reset ; wire reset ;
initial dout = 0 ;
always @ (posedge (clk))
begin
if (reset) dout <= 0; else if
(dout<=9) begin
dout <= dout + 1; end
else if (dout==9) begin
dout <= 0; end
end endmodule
```

PRE LAB QUESTIONS

1. How many number of flip-flops required in a decade counter?
2. How many number of flip-flops required in a Mod – N Counter?
3. What is the difference between synchronous and asynchronous counters?
4. An n stage ripple counter can count up to_____.

LAB ASSIGNMENT

Design and implement a synchronous 3 – bit up/down counter using J-K flip- flops.

Implement a ring counter.

Implement a Johnson counter.

Design a 4-bit ripple counter and verify its functionality.

POST LAB QUESTIONS

What is an asynchronous counter?

How is it different from a synchronous counter?

What are the advantages of synchronous counters?

Design mod-5 synchronous counter using T FF.

What is a decade counter?

For how many clock pulses the final output of a modulus 8 counter occur?

How the up counter can be made to work as down counter?

Design of a N- bit Register of Serial- in Serial –out, Serial in parallel out, Parallel in Serial out and Parallel in Parallel Out using different FFs.	EXPT. NO : 11
	DATE:

AIM: Design and simulate the HDL code for universal shift register.

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

Universal Shift Register is a register which can be configured to load and/or retrieve the data in any mode (either serial or parallel) by shifting it either towards right or towards left. In other words, a combined design of unidirectional (either right- or left- shift of data bits as in case of SISO, SIPO, PISO, PIPO) and bidirectional shift register along with parallel load provision is referred to as universal shift register.

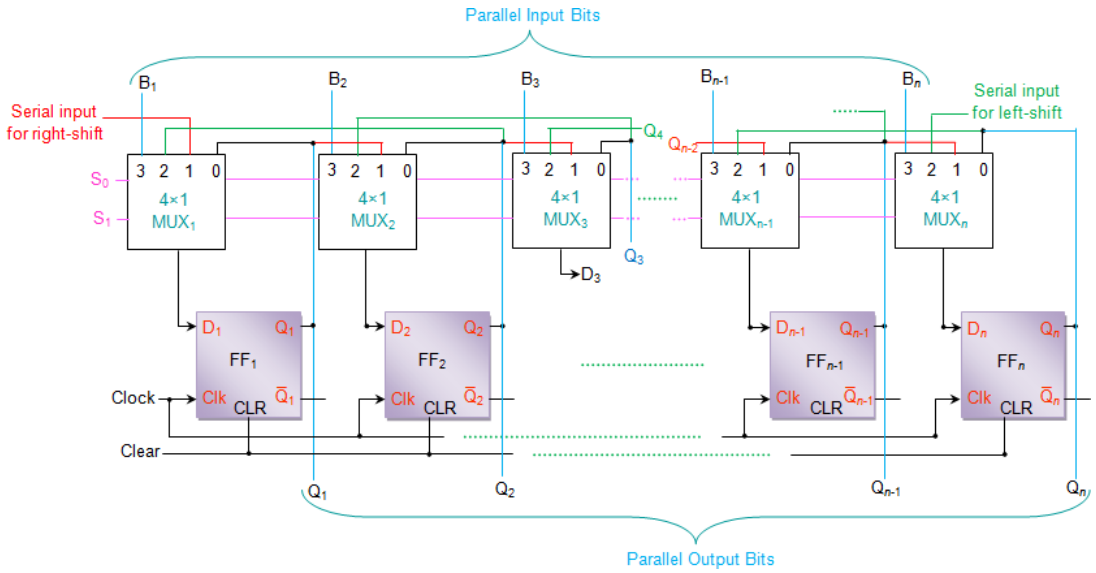


Figure 10.1 N-Bit Universal Shift register

The working of this shift register is explained by the Table 10.1. The corresponding truth table and the wave forms are given by Table 10.2.

Table 9.1 Functional table for n-bit universal shift register

Select lines		Functionality
S_0	S_1	
0	0	No change for any number of clock cycles as the outputs of the flip-flops are back-fed to themselves
0	1	Data bits within the register shift right for each clock tick with the serial input bits being provided at D_1 via MUX_1
1	0	Data bits within the register shift left for each clock tick with the serial input bits being provided at D_n via MUX_n
1	1	Bits of the data word to be stored are fed in parallel format through pin number 3 of each MUX at the rising edge of the clock

Table 9.2 Truth table for n-bit universal shift register

Serial Input for Left Shift							$L_1L_2...L_n$	
Serial Input for Right Shift							$R_1R_2...R_n$	
Parallel Input							$B_1B_2...B_n$	
Clk	**CLR**	**Mux Output**	**Outputs**					
			Q_1	Q_2	---	Q_{n-1}	Q_n	
1	1	X	0	0	---	0	0	Register is Cleared
2	0	1	R_1	0	---	0	0	Right-shift of Data Bits
3	0	1	R_2	R_1	---	0	0	
. . .	. . .	. . .	. . .	. . .	---	. . .	. . .	
n+1	0	1	R_n	R_{n-1}	---	R_2	R_1	
n+2	1	X	0	0	---	0	0	Register is Cleared
n+3	0	2	0	0	---	0	L_1	Left-shift of Data Bits
n+4	0	2	0	0	---	L_1	L_2	
. . .	. . .	. . .	. . .	. . .	---	. . .	. . .	
2n+2	0	2	L_1	L_2	---	L_{n-1}	L_n	
2n+3	0	0	L_1	L_2	---	L_{n-1}	L_n	No Change
2n+4	0	0	L_1	L_2	---	L_{n-1}	L_n	
2n+5	0	3	B_1	B_2	---	B_{n-1}	B_n	Parallel Data Loading
. . .	. . .	. . .	. . .	. . .	---	. . .	. . .	

PROCEDURE

1. Create a module with required number of variables and mention it's input/output.
2. Write the description of the universal shift register.
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained output with

the required one.

CODE

```
//universal shift register

module p20(op,in,s,MSB_in,LSB_in,clk); input
[3:0]in;
input [1:0]s;
input MSB_in, LSB_in,clk; output
[3:0]op;
reg [3:0]op;
always @( posedge clk) case
(s)
2'b00: op <= op;
2'b01: op <= {MSB_in, op[3:1]};
2'b10: op <= {op[2:0], LSB_in};
2'b11: op <= in;
endcase endmodule
```

PRE LAB QUESTIONS

- What is a register
- What is a shift register?
- Mention the various shift operations.
- What is the difference between logical shift and arithmetic shift?

LAB ASSIGNMENT

9. Design a shift right register.
10. Design a shift left register.
11. Design a circular shift right register using JK flip flop.
12. Design a circular left right register using JK flip flop.

POST LAB QUESTIONS

13. Write a HDL code to load the data parallel in universal shift register.
14. Write a HDL code to load the data serial in universal shift register.
15. Write a HDL code to perform serial in parallel out (SIPO) operation in universal shift register.
16. Write a HDL code to perform serial in serial out (SISO) operation in universal shift register.
17. Write a HDL code to perform parallel in serial out (PISO) operation in universal shift register.
18. Write a HDL code to perform parallel in parallel out (SISO) operation in universal shift register.

Design of Sequence Detector (Finite State Machine- Mealy and Moore Machines).	EXPT. NO : 11
	DATE:

AIM: To perform the design flow to generate state machines in Verilog code to detect the given sequence of bits.

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC

As an illustrative example a sequence detector for bit sequence '1011' is described. Every clock-cycle a value will be sampled, if the sequence '1011' is detected a '1' will be produced at the output for 1 clock-cycle. There are two methods to design state machines, first is Mealy and second is Moore style. We will give you an example for both styles.

Following is the behavior description of the sequencer for a Mealy style implementation and the state diagram is shown in figure 11.1:

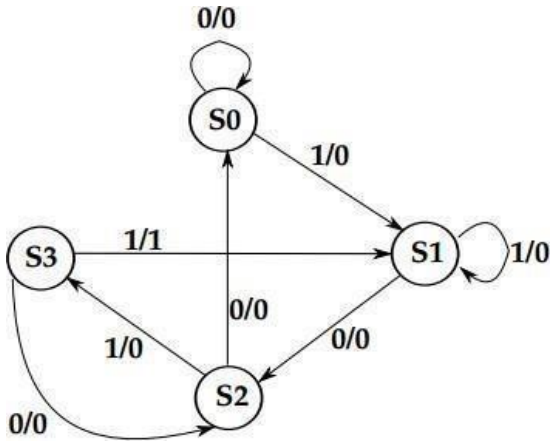


Figure 11.1: Mealy State Machine for Detecting a Sequence of '1011'

- When in initial state (S0) the machine gets the input of '1' it jumps to the next state with the output equal to '0'. If the input is '0' it stays in the same state.
- When in 2nd state (S1) the machine gets an input of '0' it jumps to the 3rd state with the output equal to '0'. If it gets an input of '1' it stays in the same state.

When in the 3rd state (S2) the machine gets an input of '1' it jumps to the 4th state with the output equal to '0'. If the input received is '0' it goes back to the initial state.

When in the 4th state (S3) the machine gets an input of '1' it jumps back to the 2nd state, with the output equal to

'1'. If the input received is '0' it goes back to the 3rd state.

Following is the behavior description of the sequencer for a Moore style implementation and the state diagram is shown in figure 11.2:

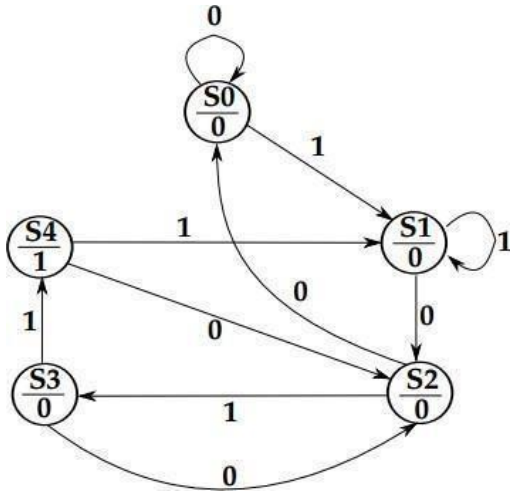


Figure 11.2: Moore State Machine for Detecting a Sequence of '1011'

- In initial state (S0) the output of the detector is '0'. When machine gets the input of'1' it jumps to the next state. If the input is '0' it stays in the same state.
- In 2nd state (S1) the output of the detector is '0'. When machine gets an input of '0'it jumps to the 3rd state. If it gets an input of '1' it stays in the same state.
- In the 3rd state (S2) the output of the detector is '0'. When machine gets an input of'1' it jumps to the 4th state. If the input received is '0' it goes back to the initial state.
- In the 4th state (S3) the output of the detector is '0'. When machine gets an input of'1' it jumps to the 5th state. If the input received is '0' it goes back to the 3rd state.
- In the 5th state the output of the detector is '1'. When machine gets an input of '0'it jumps to the 3rd state, otherwise it jumps to the 2nd state.

After designing the state machines the models have to be transformed into Verilog code describing the architecture. Therefore, it is helpful to get an understanding about the building blocks. Figure 11.3 shows the entity for the sequence detector to be developed. The two blocks inside, i.e., the Combinational and the register block is building out of the two processes used within the architecture in Verilog. The combinational block decides the next state of the FSM according to the current state and the input as well as drives the output according to the state (and input for Mealy implementation). The register block saves the current state of the FSM. This structure can be used to write the Verilog code.

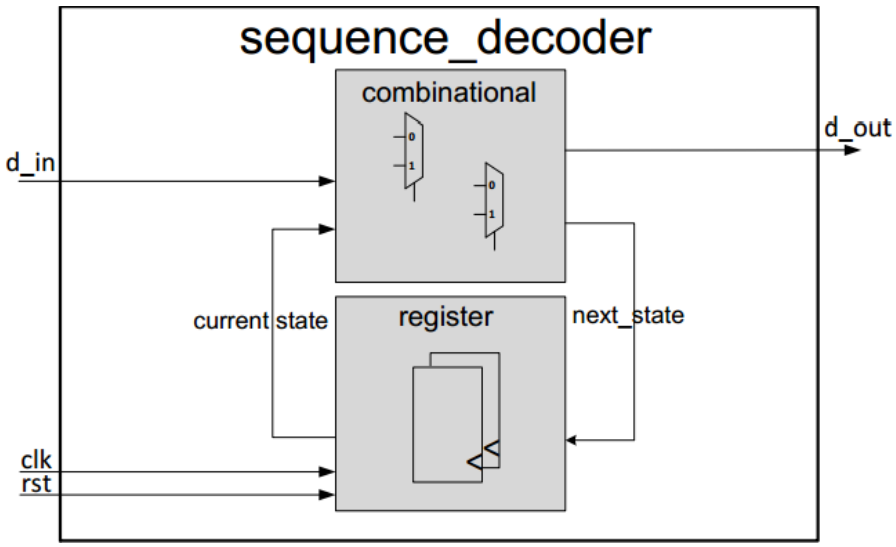


Figure 11.3: Block diagram clarifying the basic building blocks of an FSM

PROCEDURE

1. Create a module with required number of variables and mention it's input/output.
2. Write the description of the sequence detector FSM in behavioral model.
3. Create another module referred as test bench to verify the functionality.

Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
// sequence detector

module p22(clk, rst, inp, outp); input
clk, rst, inp;
output outp;

reg [1:0] state;reg outp; always
@( posedge clk, rst ) begin
if( rst )
state <= 2'b00; else
begin
case( {state,inp} )
3'b000: begin state <=
2'b00; end
3'b001: begin state
<= 2'b01; end
3'b010: begin state
```

```
<= 2'b10; end
3'b011: begin state
<= 2'b01; end
3'b100: begin state
<= 2'b10; end
3'b101: begin state
<= 2'b11; end
3'b110: begin state
<= 2'b10; end
3'b111: begin state
<= 2'b01; end
endcase end
assign outp = (({state,inp})==3'b111)? 1'b1 : 1'b0; end
endmodule
```

PRE LAB QUESTIONS

1. Design a FSM to detect the sequence '1010'.
2. Design a state flow diagram for the sequence detector FSM '10010'.
3. Design the state table for the sequence detector FSM '10010'
4. What is a sequential circuit?

LAB ASSIGNMENT

5. Design a FSM to detect the sequence '1011'.
6. Design a state flow diagram for the sequence detector FSM '1011.
7. Design the state table for the sequence detector FSM '1011'.
8. Obtain the Boolean logic expressions for the next states from the obtained state table.
9. Observe the RTL schematic of the designed FSM.

POST LAB QUESTIONS

10. Design a state flow diagram for the sequence detector FSM '1010101.
11. Design a '1010101' sequence detector using Verilog HDL coding.

Design of 4- Bit Multiplier, Divider	EXPT. NO : 12
	DATE:

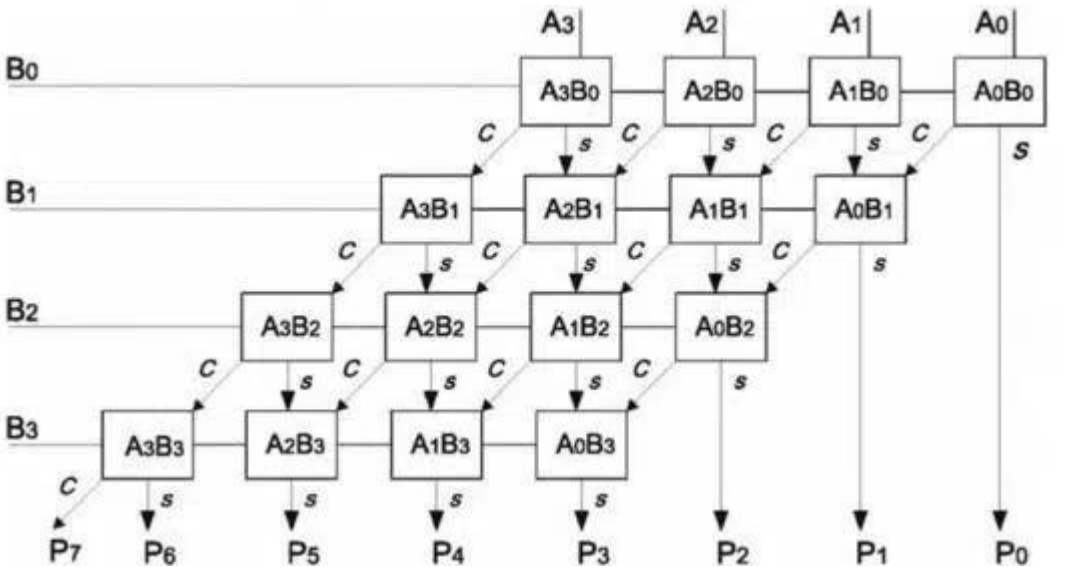
AIM: 4 X 4 Multiplier Combinational Multiplier

A 4 X 4 Array Multiplier is multiplier which takes two array of 4 bit each (a binary 4 bit number) and multiplies them to generate a 8 bit output.

RESOURCES

PC installed with CADENCE tool

PROGRAM LOGIC:



4 X 4 array Multiplier Verilog Code (Using Full Adder)

CODE:

```
module multiplier_4_x_4(product,inp1,inp2);


 output [7:0]product;
 input [3:0]inp1;
 input [3:0]inp2;

 assign product[0]=(inp1[0]&inp2[0]);

 wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17;

 HA HA1(product[1],x1,(inp1[1]&inp2[0]),(inp1[0]&inp2[1]));
 FA FA1(x2,x3,inp1[1]&inp2[1],(inp1[0]&inp2[2]),x1);
 FA FA2(x4,x5,(inp1[1]&inp2[2]),(inp1[0]&inp2[3]),x3);
 HA HA2(x6,x7,(inp1[1]&inp2[3]),x5);

 HA HA3(product[2],x15,x2,(inp1[2]&inp2[0]));
 FA FA5(x14,x16,x4,(inp1[2]&inp2[1]),x15);
```

```
  FA FA4(x13,x17,x6,(inp1[2]&inp2[2]),x16);
  FA FA3(x9,x8,x7,(inp1[2]&inp2[3]),x17);

  HA HA4(product[3],x12,x14,(inp1[3]&inp2[0]));
  FA FA8(product[4],x11,x13,(inp1[3]&inp2[1]),x12);
  FA FA7(product[5],x10,x9,(inp1[3]&inp2[2]),x11);
  FA FA6(product[6],product[7],x8,(inp1[3]&inp2[3]),x10);

endmodule

module HA(sout,cout,a,b);
  output sout,cout;
  input a,b;
  assign sout=a^b;
assign cout=(a&b);
endmodule

module FA(sout,cout,a,b,cin);
  output sout,cout;
  input a,b,cin;
  assign sout=(a^b^cin);
  assign cout=((a&b)|(a&cin)|(b&cin));
endmodule
```

Design of ALU to Perform – ADD, SUB, AND-OR, 1's and 2's Compliment	EXPT. NO : 13
	DATE:

AIM: To design a model to implement 8-bit ALU functionality

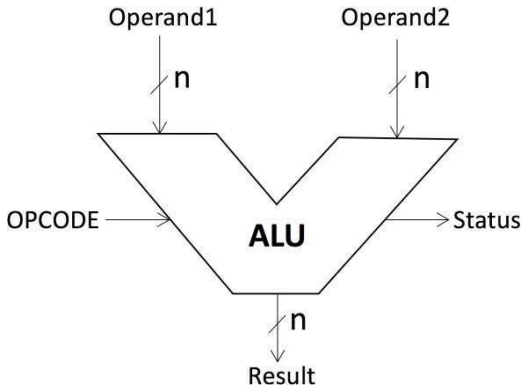
RESOURCES

PC installed with Xilinx tool

PROGRAM LOGIC

An arithmetic logic unit (ALU) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. An ALU is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs.

Figure 13.1 Arithmetic logic unit block diagram



The inputs to an ALU are the data to be operated on, called operands, and a code (opcode) indicating the operation to be performed and, optionally, status information from a previous operation; the ALU's output is the result of the performed operation. In many designs, the ALU also exchanges additional information with a status register, which relates to the result of the current or previous operations

A number of basic arithmetic and bitwise logic functions are commonly supported by ALUs. Basic, general purpose ALUs typically includes these operations in their repertoires:

1. Arithmetic operations
2. Bitwise logical operations
3. Bit shift operations

In this lab, students have to design an 8-bit ALU to implement the following operations:

Table 1: ALU Instructions

Control	Instruction	Operation
000	Add	Ouput<= A+B+Cin (Cout is carry)
001	Sub	Output <= A-B-C (Cou is barrow)
010	Or	Output <= A or B
011	And	Output <= A and B
100	Shl	Output <= A[7:0] & '0'
101	Shr	Output <= '0' & A[7:1]
110	Rol	Output <= A[2:0] & A[7]
111	Ror	Output <= A[0] & A[7:1]

Table 1 also illustrates the encoding of the control input The 4

- bit ALU has the following inputs:

A: 8-bit input
B: 8-bit input
Cin: 1-bit input
Output: 8-bit output
Cout: 1-bit output
Control: 3-bit control input

The following points should be taken care of:

Use a case statement (or a similar 'combinational' statement) that checks the input combination of "Code" and acts on A, B, and Cin as described in Table1.
The above circuit is completely combinational. The output should change as soon as the code combination or any of the input changes.
You can use arithmetic and logical operators to realize your design.

PROCEDURE

Create a module with required number of variables and mention it's input/output.
Write the description of the ALU by using case statements.
Create another module referred as test bench to verify the functionality.
Follow the steps required to simulate the design and compare the obtained output with the required one.

CODE

```
//8 bit ALU

module p13(z,a,b,sel); input
[7:0]a,b;
input [3:0]sel;
output [7:0]z;
reg [7:0]z;
```

```
always@(sel,a,b)
begin
case(sel) 4'b0000:
z=a+b;
4'b0001: z=a-b;
4'b0010: z=b-1;
4'b0011: z=a*b;
4'b0100: z=a&&b;
4'b0101: z=a||b;
4'b0110: z=!a;
4'b0111: z=~a;
4'b1000: z=a&b;
4'b1001: z=a|b;
4'b1010: z=a^b;
4'b1011: z=a<<1;
4'b1100: z=a>>1;
4'b1101: z=a+1;
4'b1110: z=a-1;
endcase end
endmodule
```

PRE LAB QUESTIONS

1. State the basic units of the computer. Name the subunits that make up the CPU, and give the function of each of the units.
2. Give the description of computer architecture.
3. What are arithmetic operations
4. What are bitwise logical operations
5. What are bit shift operations

LAB ASSIGNMENT

1. Design the 4-bit ALU
2. Write a HDL code to implement basic arithmetic operations using ALU.

POST LAB QUESTIONS

1. Write a HDL code to implement bitwise logical operations using ALU.
2. Write a HDL code to implement bit shift operations using ALU.

Implementing the above designs on FPGA kits	EXPT. NO : 14
	DATE:

AIM: Implementing the above designs on FPGA kits

RESOURCES:

PC Installed with CADENCE tool.