### ADVANCE VLSI DESIGNING

**A CERTIFICATE COURSE CONDUCTED BY**

THE SURE TRUST

**Skill Upgradation for Rural-youth Empowerment – TRUST** [**(www.suretrustforruralyouth.com)**](http://www.suretrustforruralyouth.com/)

### COURSE TRAINING ATTENDED

### BY

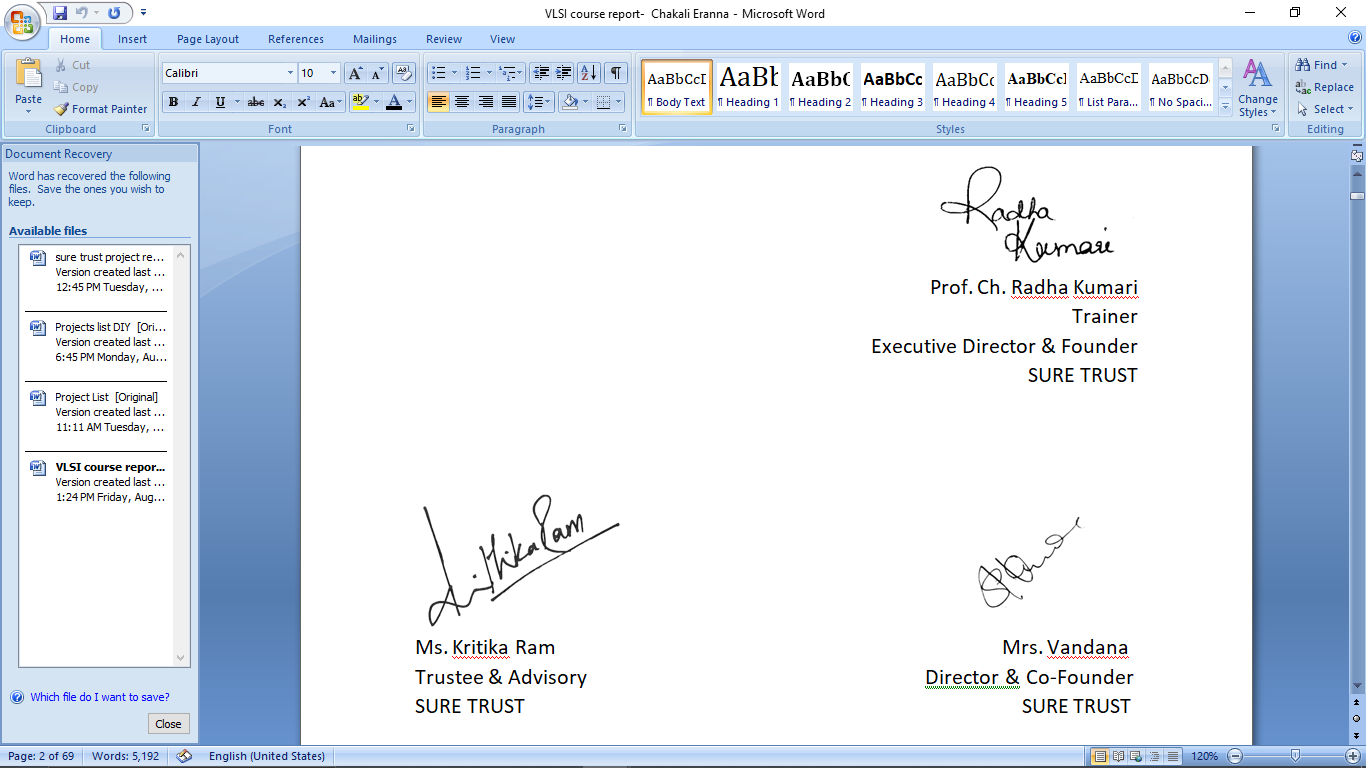
### Soniya Janjuluri

**MAY2023 – SEPTEMBER2023**

****

### Declaration

#### This is to certify that Mrs.Soniya Janjuluri has successfully completed the four months training given in “Advance VLSI Designing conducted by The SURE TRUST” during the period from may 2023 to September 2023.



### **­­­­­­** TABLE OF CONTENTS

#### The SURE TRUST

1. Course Content

#### Conduct of the Course

Student byelaws Written Tests Assignments

#### Student Feedback

1. Uniqueness of the Course according to student
2. Concluding Remarks

## Introduction to the SURE TRUST

### Introduction to The SURE TRUST

The SURE TRUST is born to enhance the employability of educated unemployed rural youth. It is observed that there is a wide gap between the skills acquired by students from the academic institutions and the skills required by the industry to employ them. Employability enhancement is done through giving one on one training in emerging technologies, completely through online mode. The mission of the SURE TRUST is to bridge the gap between the skills acquired and the skills required by training them in the most emerging technologies such as Artificial Intelligence (AI), Python Program, Machine Learning (ML), Deep Learning (DL), Data Science & Data Analytics, Blockchain Technology, Robotic Process Automation (RPA), Project Management, Excel for Business Application, Statistical tools & Applications, Spoken English and Business Communication etc., that will enhance their employability. After completion of four months training in the course, the trainees will get live projects from industries as internship activity to get experience in applying to real time situation what they have learnt during the course. These projects will give them hands on experience which is much sought after by the prospective industry employing them. Currently students from all over India are enrolling for various courses offered by the SURE TRUST. The SURE TRUST offers every course free of cost with no financial burden of any kind to students. This initiative is purely a service oriented one aiming to guide the rural youth who are educated but unemployed due to lack of upgradation in their skill sets. The birth of SURE TRUST is a God given boon to rural youth who could reach great heights either in employment or in entrepreneurship once they receive the training offered followed by the company internship. Many companies are coming forward to join their hands with us by offering internship projects to hand hold and lead the rural youth in their career settlement.

### Vision of the SURE TRUST

The vision of the SURE TRUST is to enhance the employability of educated unemployed youth, particularly living in rural areas, through skill upgradation, with no cost to the students.

### Mission of the SURE TRUST

The mission is to bridge the gap between the skills acquired in the academic institutions and the skills required in industries as a pre-condition for employment.

### Functioning of the SURE TRUST

There are three dedicated, committed, and hard-working women on the board of management of the SURE TRUST who will look into the various administrative and other matters relating to the enrolment of students, organizing trainers, entering into agreements with companies for getting live projects to students as internship programs, and so on. All the three women on the board are all the alumni from Sri Sathya Sai Institute of Higher Learning, Anantapur Campus, deemed to be a University. The women board is supported by five eminent advisories who are from different walks of life and have made outstanding mark in career in their respective fields. For more details about SURE TRUST please visit the website [www.suretrustforruralyouth.com](http://www.suretrustforruralyouth.com/)

### Course Content

The SURE TRUST conducts four months of training for every course on a uniform basis. A session spanning across one to one & half hour is taken by the trainers for every major course. Sessions are conducted to complete the predesigned course structure within a fixed time period. Course content is designed to suit the current requirement of the Industry and validated by industry experts. The course content of all these courses is so dynamic that any changed condition noticed in the industry will automatically get reflected in the content of the respective course. As the course content is dynamic, the Following is the course content of the current course in Advance VLSI Designing:

## Advance VLSI Designing

##### Objective:

* The dynamic curriculum of the Advance VLSI Design and Verification course fits perfectly with the career aim of fresh engineering graduates and helps them to ‘future-proof’ themselves and remain relevant for the rapidly evolving Semiconductor technology space.

##### Course Content:

* Module 1 - Introduction to VLSI design and verification.
* Module 2 - Introduction to RTL design using Verilog. Modeling methods.
* Module 3 - Procedural assignments and Continuous assignments. MUX design.
* Module 4 - Introduction to Testbench design using Verilog. Design and Verification Memory Modules.
* Module 5 - Introduction to Finite State Machines. Design and Verification of FSM.
* Module 6 - Final project - Router 1x3 Design and Verification.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##### Conduct of the Course:

1. Modalities for the conduct of all the courses are fixed by the SURE TRUST which are uniformly followed across the courses.
   * Mode of Training --- Online
   * Period of Training --- Four months
   * Sessions per week --- 3 to 6
   * Length of the session --- 1 to 2 hours
   * Tests to be taken --- 2 per month
   * Assignments --- 2 per month
   * Last 15 days --- Final practice and preparing the course report
2. Students Byelaws:

Students enrolling for the courses under SURE TRUST are strictly required to follow the following Byelaws set for them.

**Byelaws for students to become eligible for certificate at the end of the course** I. Minimum Attendance:

Every student must put in a minimum of 85% attendance in attending the classes for getting the eligibility to receive the certificates.

1. Two written tests are to be taken each month:

Since the objective of the certification program is to turn out well-qualified students from the respective courses, minimum two written tests are to be taken in each month for each course to ensure that the students are pulled along the expected line of standard.

1. Assignment Submission:

Ten exercises constituting one assignment for every two to three new functions/topics taught, resulting in minimum seven such assignments are to be submitted during the four months period.

1. Preparing the final course report in the prescribed format:

During the last fifteen days in the fourth month, students many be asked to consolidate and compile all the assignments submitted in a word document along with the other chapters which will constitute a course report for each student. This report will be the unique contribution a student carries from the trust to show case the rigorous training he/she received during the four months period. Besides the report will stand as a testimony for the detailed learning a student has acquired in the chosen area.

This will facilitate the industry in handpicking the required student for the job.

1. External Viva-voce:

All the students are expected to attend each class for full duration. Some students are observed moving out of classes after logging in which does not go well with the learning objective of students.

1. KYC norms:

Each student wishing to enroll for the course must submit a written letter saying that he/she will not drop from the course until its completion, which will also be signed by father / mother besides the student himself / herself.

1. Attend the full class:

All the students are expected to attend each class for full duration. Some students are observed moving out of classes after logging in which does not go well with the learning objective of students.

1. Ensure discipline in the group:

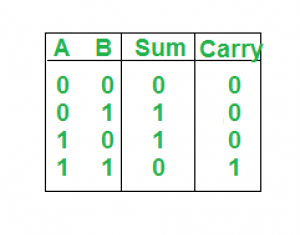
All the students are advised strictly to follow group etiquette and restrain from posting in the group any unethical messages or teasing messages or personal interactive messages. This group is purely created for academic purposes and hence only academic interactions should go on

**Combinational Circuits:**

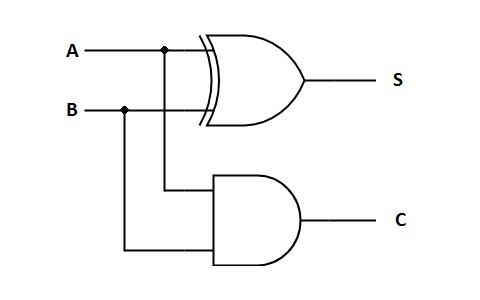
**HALF ADDER :**

**A half adder is a digital logic circuit that performs binary addition of two single-bit binary numbers. It has two inputs, A and B, and two outputs, SUM and CARRY. The SUM output is the least significant bit (LSB) of the result, while the CARRY output is the most significant bit (MSB) of the result, indicating whether there was a carry-over from the addition of the two inputs.**

**Truth Table:**



**Circuit Diagram:**

****

**RTL code:**

module halfadderbehavioral(a,b,sum,cout);

input a;

input b;

output reg sum;

output reg cout;

always @(\*)

begin

case ({a,b})

2'b00:begin sum=0 ; cout=0; end

2'b01: begin sum=1 ; cout=0; end

2'b10: begin sum=1; cout=0; end

2'b11: begin sum=0 ; cout=1; end

default : begin sum=0 ; cout=0; end

endcase

end

endmodule

**Test bench:**

module halfadderbeh\_tb();

reg a,b;

wire sum,cout;

halfadderbehavioral h(a,b,sum,cout);

initial

begin

a=1'b0;

b=1'b0;

#1000 $finish();

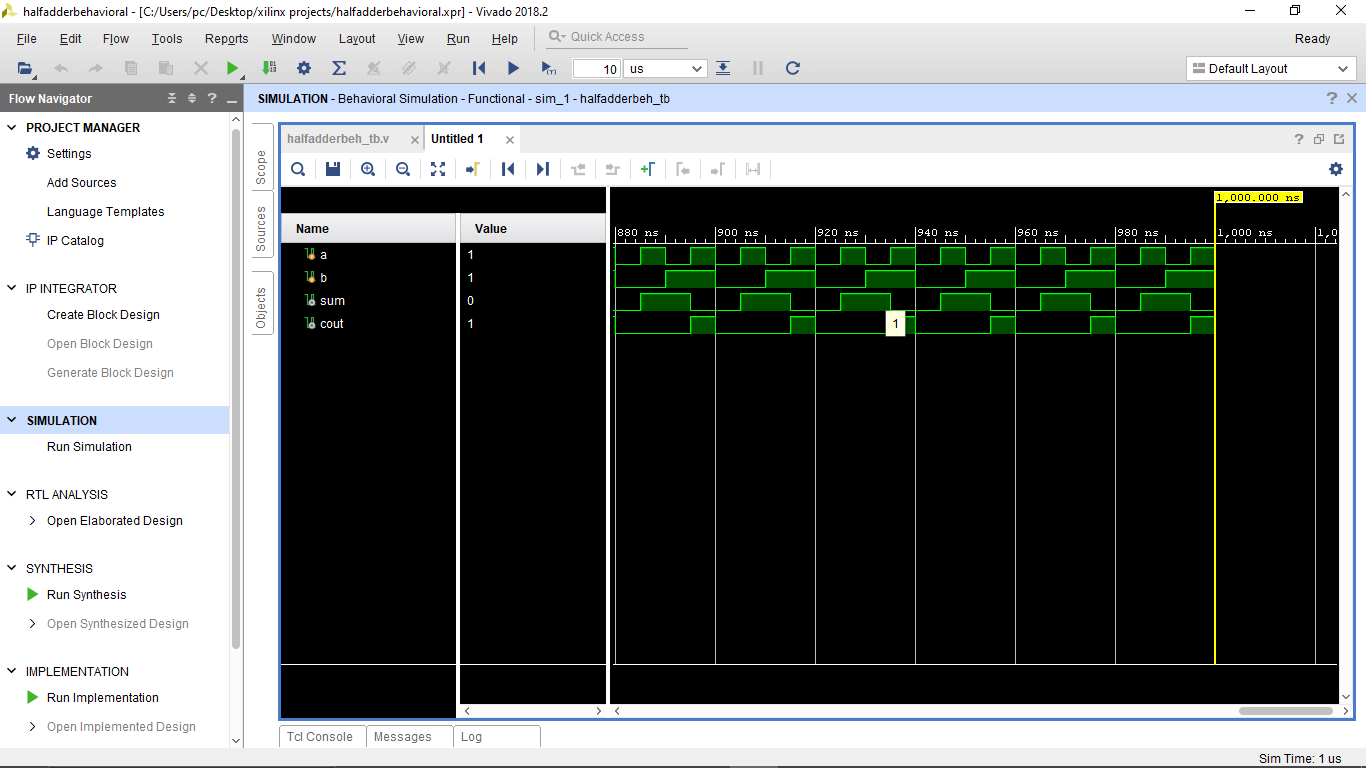
end

always #5 a=~a;

always #10 b=~b;

endmodule

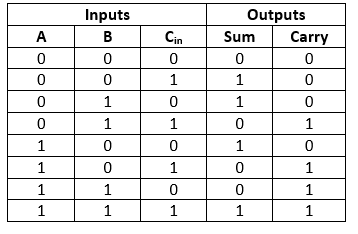
**Waveforms**:



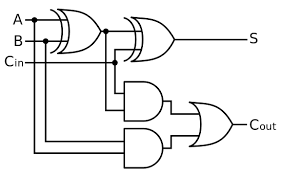
**FULL ADDER:**

The half adder is used to add only two numbers. To overcome this problem, the full adder was developed. The full adder is used to add three 1-bit binary numbers A, B, and carry C. The full adder has three input states and two output states i.e., sum and carry.

**Truth Table:**



**Circuit diagram:**

****

**RTL Code:**

module fulladderbehavioral(a,b,c,sum,cout);

input a,b,c;

output reg sum,cout;

always @(\*)

begin

case({a,b,c})

3'b000: sum=0;

3'b001: sum=1;

3'b010: sum=1;

3'b011: sum=0;

3'b100: sum=1;

3'b101: sum=0;

3'b110: sum=0;

3'b111: sum=1;

default: sum=0;

endcase

case({a,b,c})

3'b000: cout=0;

3'b001: cout=0;

3'b010: cout=0;

3'b011: cout=1;

3'b100: cout=0;

3'b101: cout=1;

3'b110: cout=1;

3'b111: cout=1;

default: cout=0;

endcase

end

endmodule

**Test bench:**

module fulladder\_tb();

reg a,b,c;

wire sum,cout;

fulladderbehavioral f(a,b,c,sum,cout);

initial

begin

a=0;

b=0;

c=0;

#500 $finish();

end

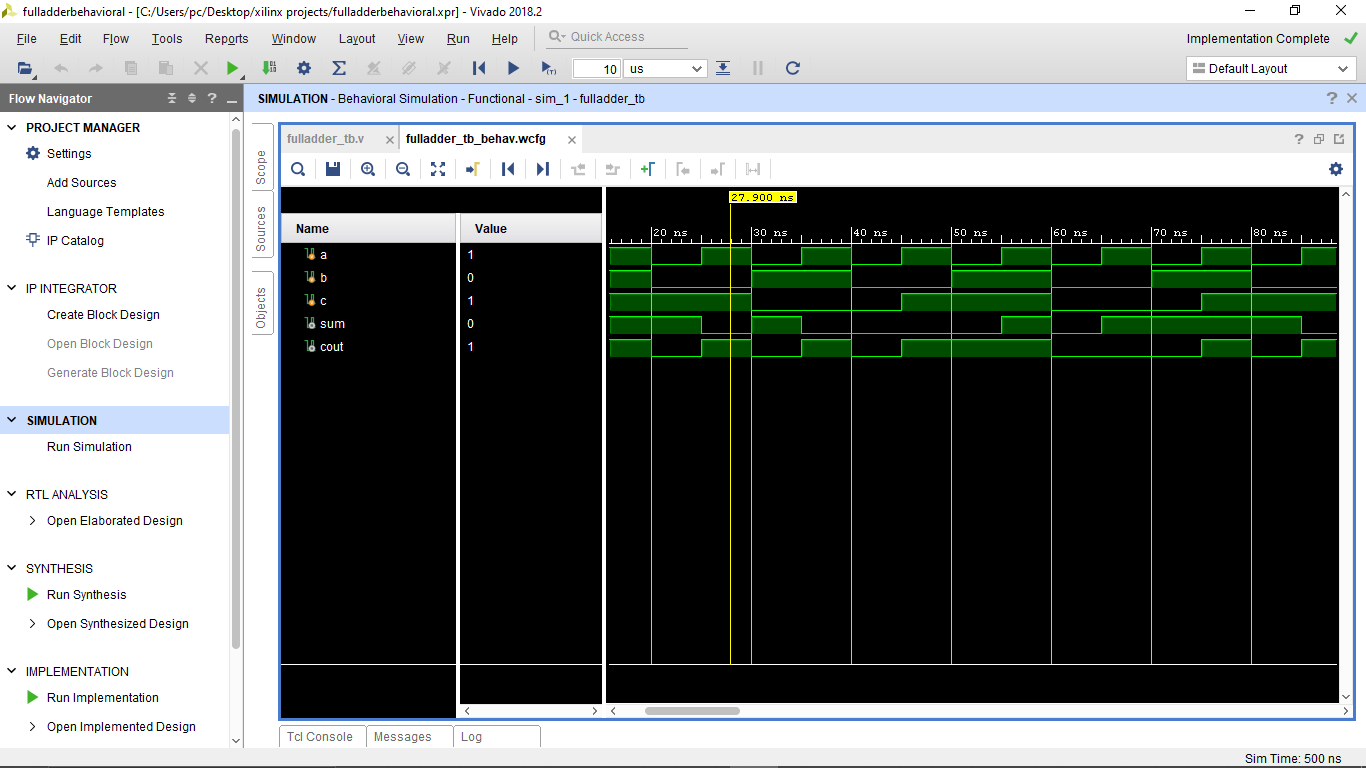
always #5 a=~a;

always #10 b=~b;

always #15 c=~c;

endmodule

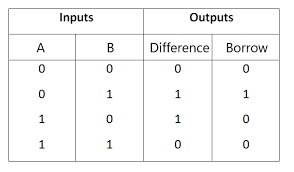
**Wave Forms:**



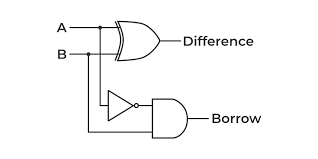
**HALF SUBSTRACTOR:**

A half subtractor is a digital logic circuit that performs binary subtraction of two single-bit binary numbers. It has two inputs, A and B, and two outputs, DIFFERENCE and BORROW. The DIFFERENCE output is the difference between the two input bits, while the BORROW output indicates whether borrowing was necessary during the subtraction.

**Truth Table:**

****

Circuit Diagram:



**RTL Code:**

module halfsubbehavioral(a,b,dif,bor);

input a,b;

output reg dif,bor;

always @(\*)

begin

case ({a,b})

2'b00: dif=0;

2'b01: dif=1;

2'b10: dif=1;

2'b11: dif=0;

default: dif=0;

endcase

case({a,b})

2'b00: bor=0;

2'b01: bor=1;

2'b10: bor=0;

2'b11: bor=0;

default: bor=0;

endcase

end

endmodule

**Test Bench:**

**module halfsubstractor\_tb();**

**reg a,b;**

**wire dif,bor;**

**halfsubbehavioral s(a,b,dif,bor);**

**initial**

**begin**

**a=0;**

**b=0;**

**#500 $finish();**

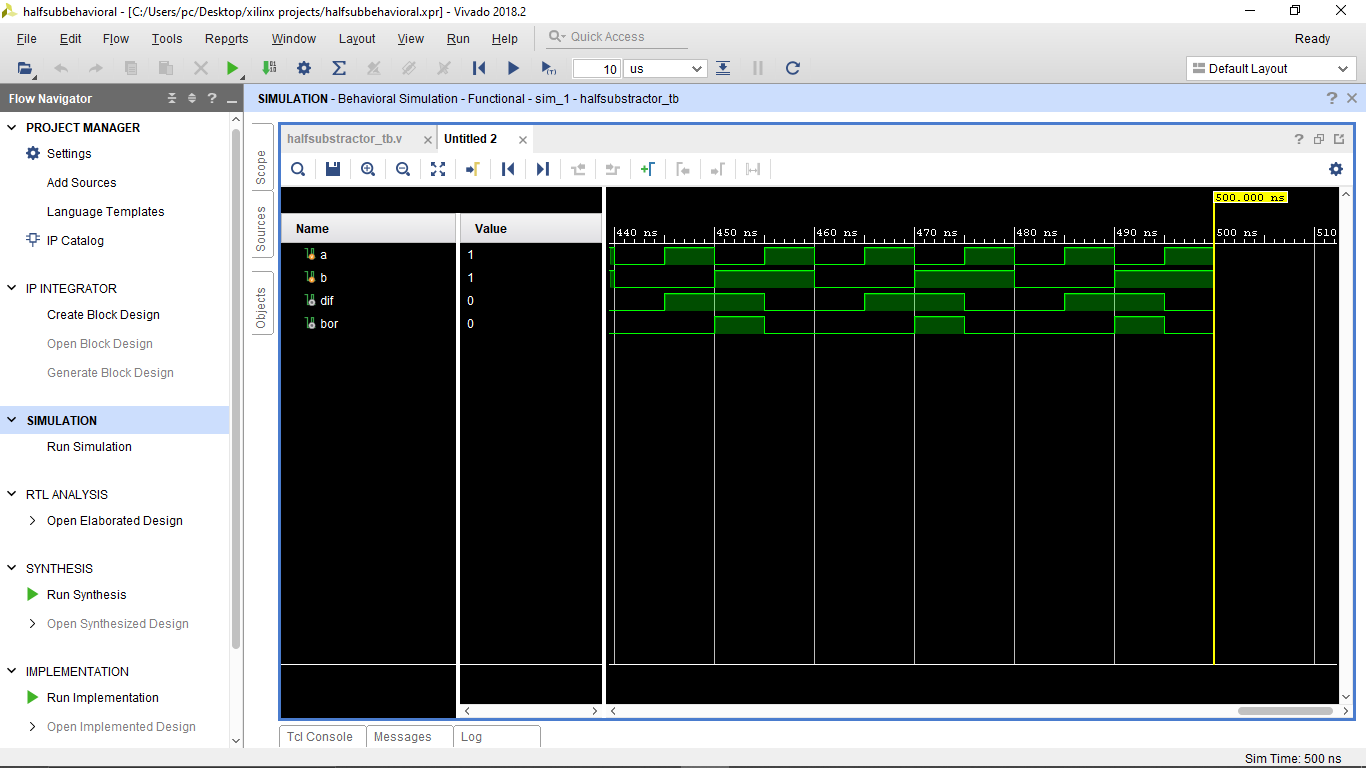
**end**

**always #5 a=~a;**

**always #10 b=~b;**

**endmodule**

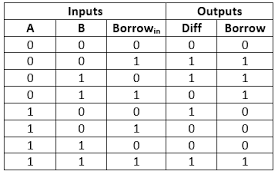
**Wave forms:**

****

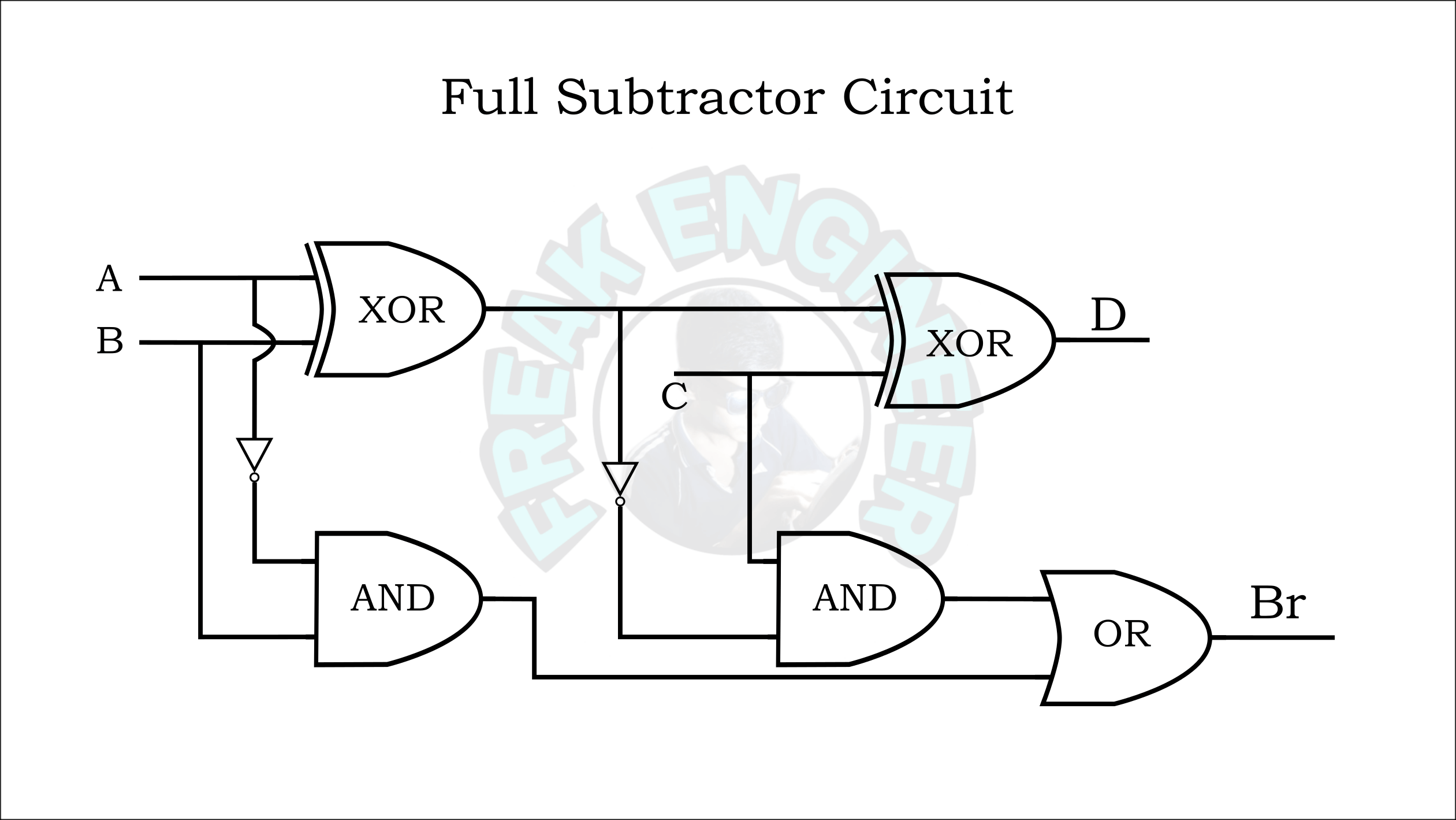
**FULL SUBSTRACTOR:**

A full subtractor is a combinational circuit that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit. This circuit has three inputs and two outputs. The three inputs A, B and Bin, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and Bout represent the difference and output borrow, respectively.

**Truth Table:**

****

**Circuit Diagram:**

****

**RTL Code:**

module fullsubbehavioral(a,b,bin,dif,bout);

input a,b,bin;

output reg dif,bout;

always @(\*)

begin

case ({a,b,bin})

3'b000: dif=0;

3'b001: dif=1;

3'b010: dif=1;

3'b011: dif=0;

3'b100: dif=1;

3'b101: dif=0;

3'b110: dif=0;

3'b111: dif=1;

endcase

case({a,b,bin})

3'b000: bout=0;

3'b001: bout=1;

3'b010: bout=1;

3'b011: bout=1;

3'b100: bout=0;

3'b101: bout=0;

3'b110: bout=0;

3'b111: bout=1;

endcase

end

endmodule

**Test Bench:**

module fullsubstractor\_tb();

reg a,b,bin;

wire dif,bout;

fullsubbehavioral s(a,b,bin,dif,bout);

initial

begin

a=0;

b=0;

bin=0;

#500 $finish();

end

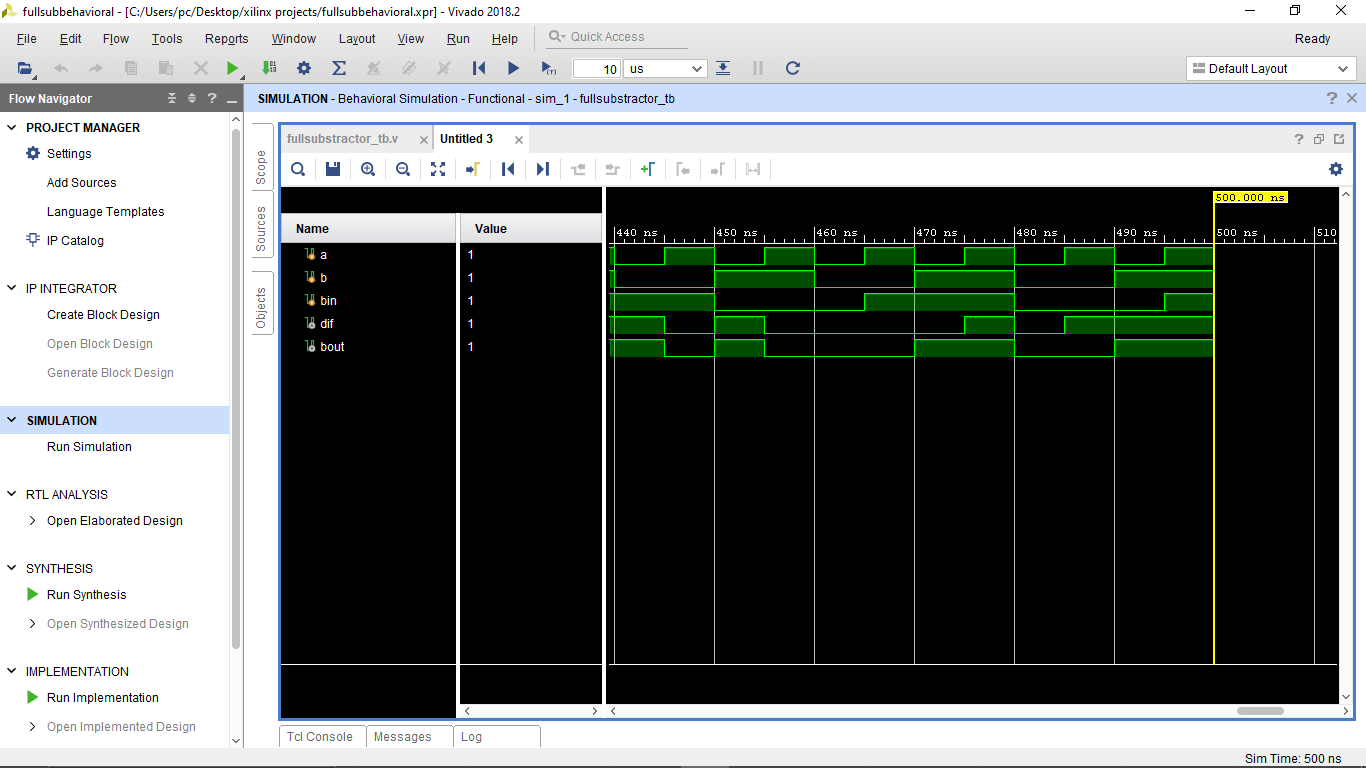
always #5 a=~a;

always #10 b=~b;

always #15 bin=~bin;

endmodule

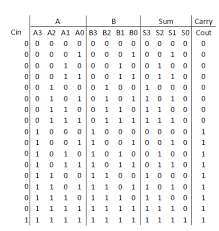
**Wave Forms:**



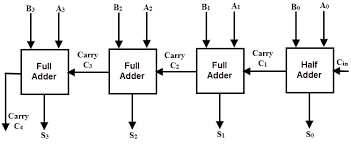
**FOUR BIT ADDER:**

The 4-bit adder-subtractor is a digital circuit capable of performing arithmetic operations on binary numbers of four bits in length.

**Truth Table:**

****

**Circuit Diagram:**

****

**RTL Code:**

module four\_bit\_adder(a,b,cin,sum,cout);

input [3:0]a,b,cin;

output [3:0]sum;

output cout;

wire [2:0]c;

fulladderdataflow a1(sum[0],c[0],a[0],b[0],cin);

fulladderdataflow a2(sum[1],c[1],a[1],b[1],c[0]);

fulladderdataflow a3(sum[2],c[2],a[2],b[2],c[1]);

fulladderdataflow a4(sum[3],cout,a[3],b[3],c[2]);

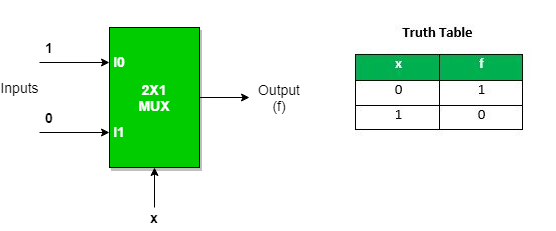
endmodule

**Test Bench:**

**2x1 MUX :**

A 2x1 multiplexer (mux) is a digital logic circuit that selects one of two input signals and forwards it to the output based on a control signal. It has two data inputs (A and B), one select input (S), and one output (Y). The select input (S) determines which of the two data inputs (A or B) is transmitted to the output.

**Truth Table:**



**RTL Code:**

module muxbehavioral(i0,i1,s0,o);

input i0,i1,s0;

output reg o;

always@(\*)

begin

case({s0})

1'b0: o=i0;

1'b1: o=i1;

default: o=0;

endcase

end

endmodule

**Test bench**:

module mux2x1\_tb();

reg i0,i1,s0;

wire o;

muxbehavioral m(i0,i1,s0,o);

initial

begin

i0=0;

i1=0;

s0=0;

#500 $finish();

end

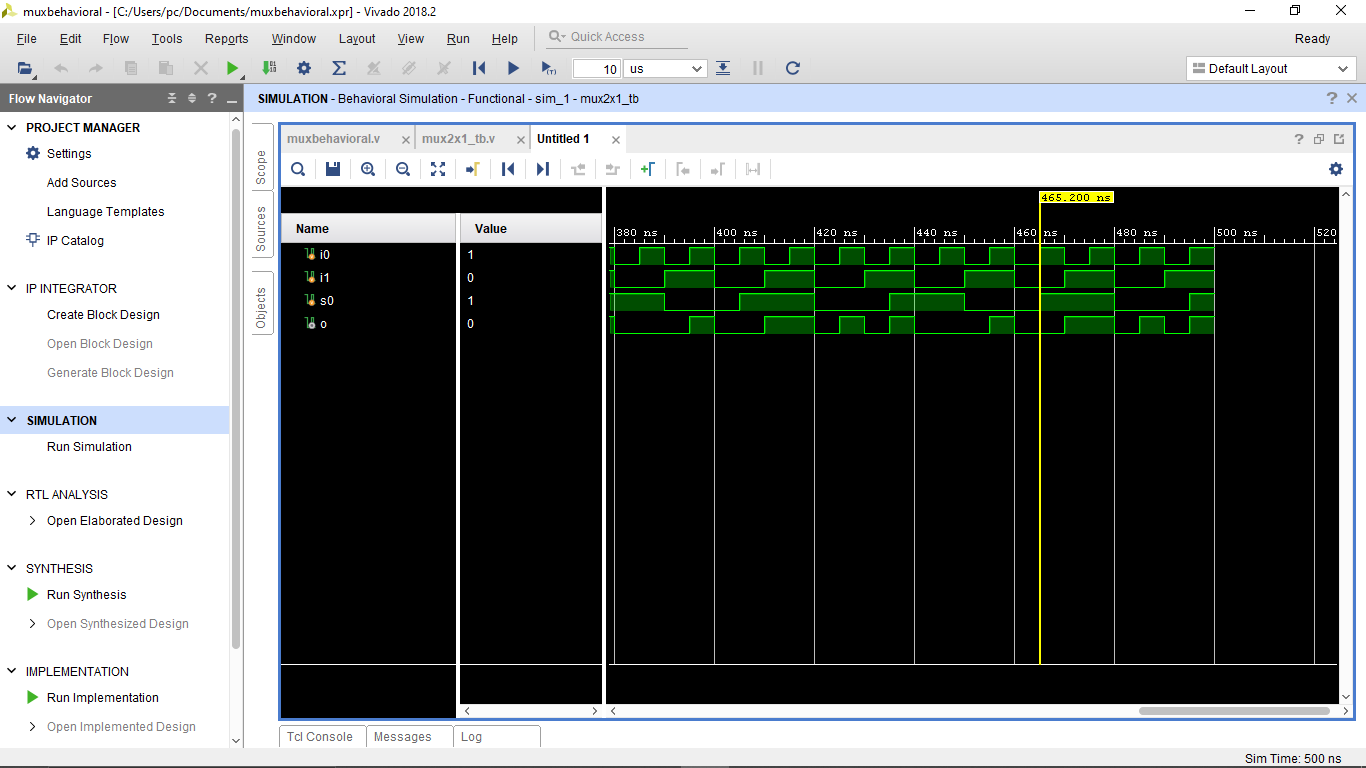
always #5 i0=~i0;

always #10 i1=~i1;

always #15 s0=~s0;

endmodule

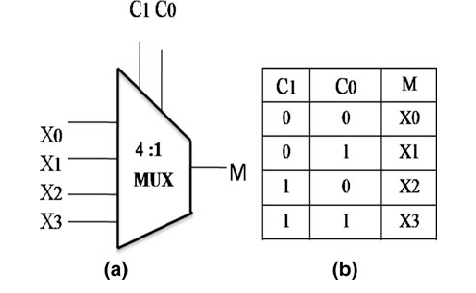
**Waveforms:**

****

**4x1 MUX:**

A 4x1 multiplexer (mux) is a digital logic circuit that selects one of four input signals and forwards it to the output based on a set of control signals. It has four data inputs (A, B, C, and D), two select inputs (S0 and S1), and one output (Y). The two select inputs (S0 and S1) determine which of the four data inputs is transmitted to the output.

**Truth Table and Circuit diagram:**



**RTL Code:**

module f\_o\_muxbehavioral(i0,i1,i2,i3,s0,s1,y);

input i0,i1,i2,i3,s0,s1;

output reg y;

always@(\*)

begin

case({s0,s1})

2'b00: y=i0;

2'b01: y=i1;

2'b10: y=i2;

2'b11: y=i3;

endcase

end

endmodule

**Test Bench:**

module mux4x1\_tb();

reg i0,i1,i2,i3,s0,s1;

wire y;

f\_o\_muxbehavioral m(i0,i1,i2,i3,s0,s1,y);

initial

begin

i0=0;

i1=0;

i2=0;

i3=0;

s0=0;

s1=0;

#500 $finish();

end

always #5 i0=~i0;

always #10 i1=~i1;

always #15 i2=~i2;

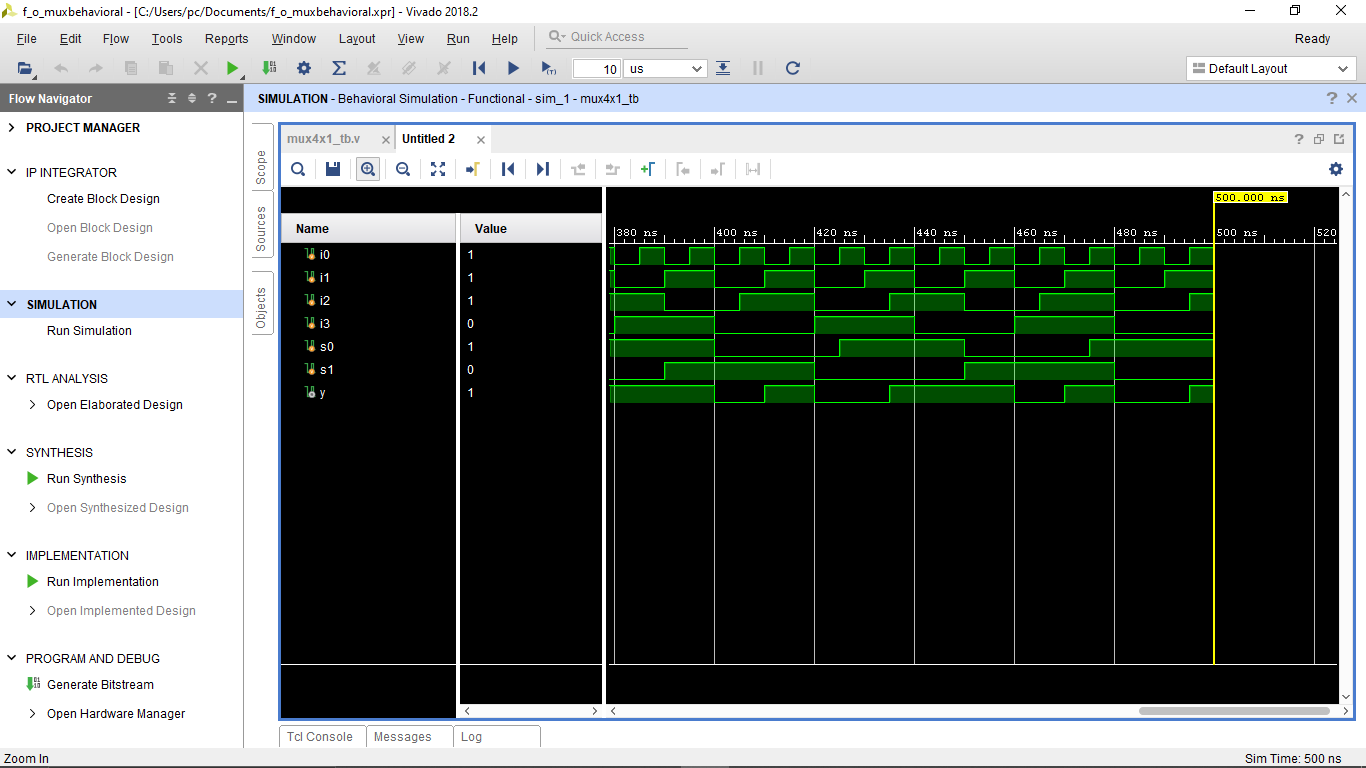
always #20 i3=~i3;

always #25 s0=~s0;

always #30 s1=~s1;

endmodule

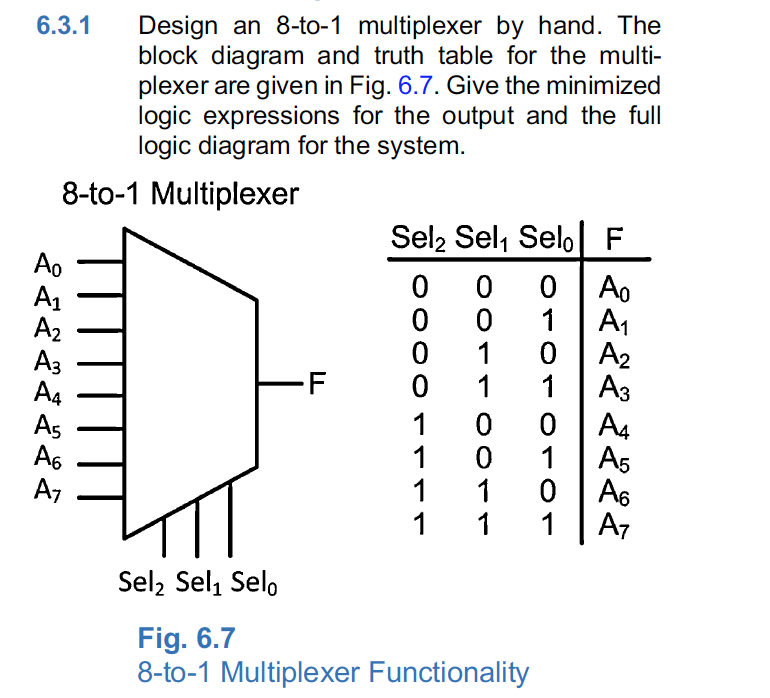
**Waveforms:**

****

**8x1 MUX:**

An 8x1 multiplexer (mux) is a digital logic circuit that selects one of eight input signals and forwards it to the output based on a set of control signals. It has eight data inputs (A0 to A7), three select inputs (S2, S1, and S0), and one output (Y). The three select inputs determine which of the eight data inputs is transmitted to the output.

**Truth Table and Circuit Diagram:**



**RTL Code:**

module eomuxstructural(i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2,y);

input i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2;

output y;

wire w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11;

not g1(w1,s0);

not g2(w2,s1);

not g3(w3,s2);

and g4(w4,w1,w2,w3,i0);

and g5(w5,w1,w2,s2,i1);

and g6(w6,w1,s1,w3,i2);

and g7(w7,w1,s1,s2,i3);

and g8(w8,s0,w2,w3,i4);

and g9(w9,s0,w2,s2,i5);

and g10(w10,s0,s1,w3,i6);

and g11(w11,s0,s1,s2,i7);

or g12(y,w4,w5,w6,w7,w8,w9,w10,w11);

endmodule

**Testbench:**

**module mux8x1\_tb();**

**reg i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2;**

**wire y;**

**eomuxbehavioral m(i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2,y);**

**initial**

**begin**

**i0=0;**

**i1=0;**

**i2=0;**

**i3=0;**

**i4=0;**

**i5=0;**

**i6=0;**

**i7=0;**

**s0=0;**

**s1=0;**

**s2=0;**

**#500 $finish();**

**end**

**always #5 i0=~i0;**

**always #10 i1=~i1;**

**always #15 i2=~i2;**

**always #20 i3=~i3;**

**always #25 i4=~i4;**

**always #30 i5=~i5;**

**always #35 i6=~i6;**

**always #40 i7=~i7;**

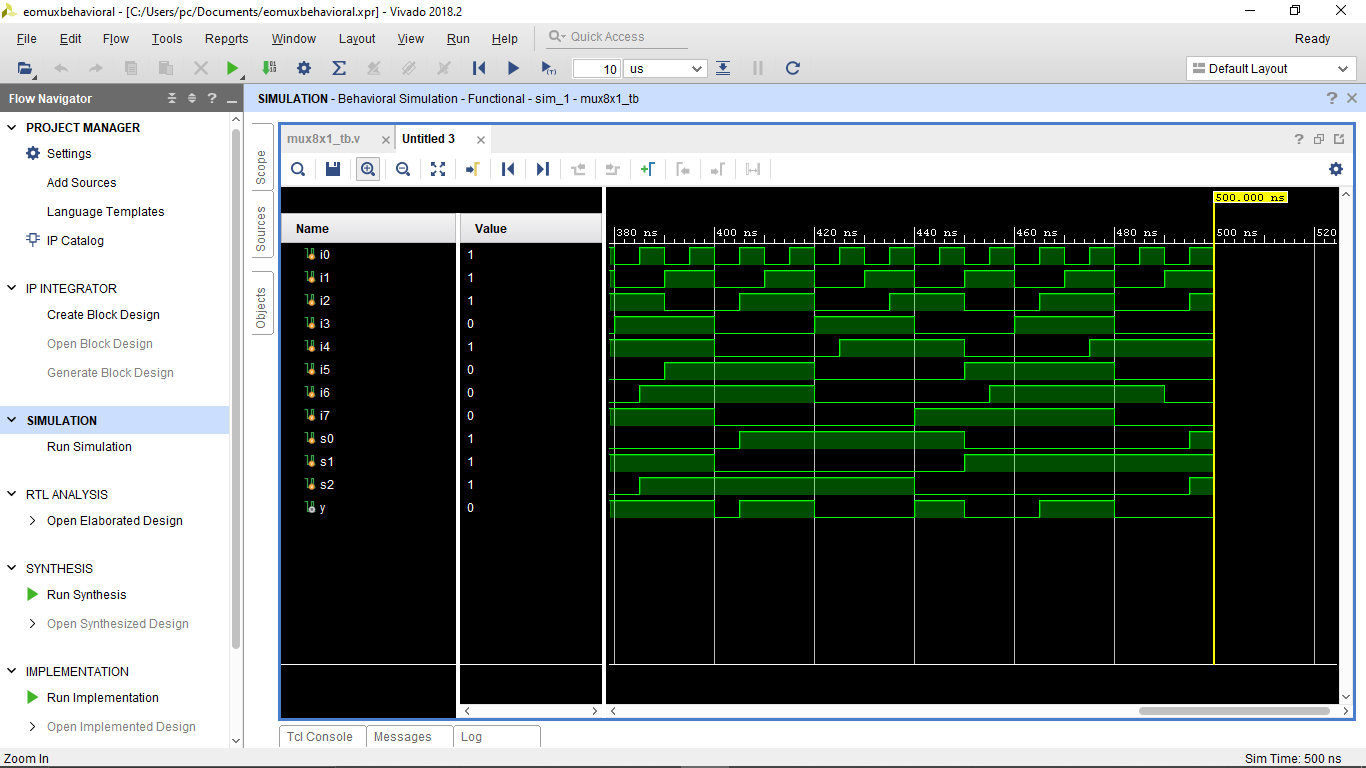
**always #45 s0=~s0;**

**always #50 s1=~s1;**

**always #55 s2=~s2;**

**endmodule**

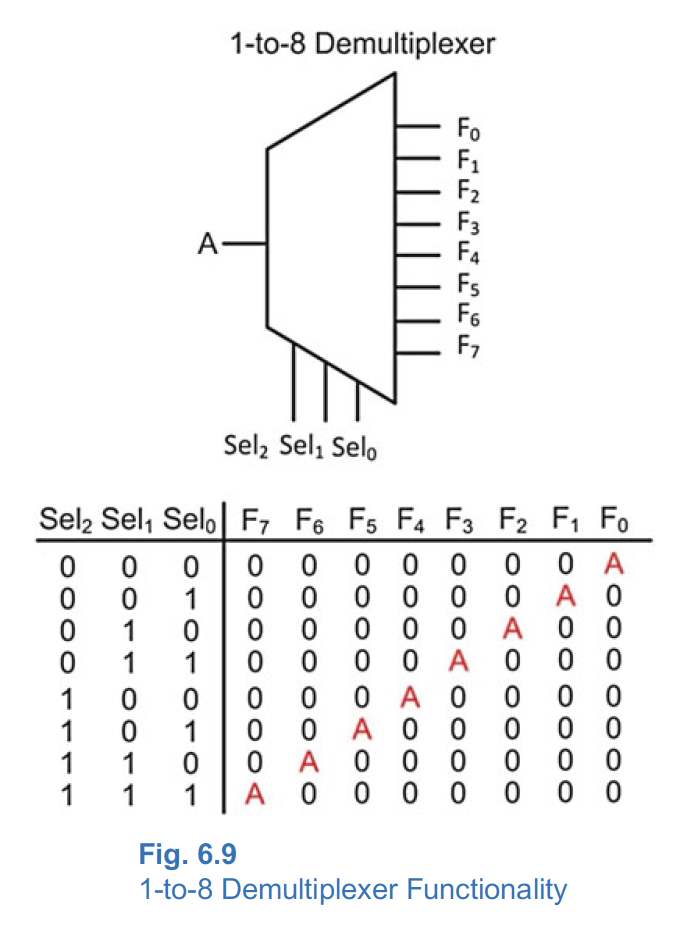
**Waveforms:**

****

**1x8 DEMUX**

A 1x8 demultiplexer (1:8 Demux) is a digital logic circuit that takes a single input and directs it to one of eight possible output lines based on the value of control signals. It essentially divides a single input signal into one of eight possible output channels**.**

**Truth table and circuit diagram:**



module oneeightdemuxdataflow(i,s0,s1,s2,y0,y1,y2,y3,y4,y5,y6,y7);

input i,s0,s1,s2;

output y0,y1,y2,y3,y4,y5,y6,y7;

assign y0=(~s0&~s1&~s2&i);

assign y0=(~s0&~s1&s2&i);

assign y0=(~s0&s1&~s2&i);

assign y0=(~s0&s1&s2&i);

assign y0=(s0&~s1&~s2&i);

assign y0=(s0&~s1&s2&i);

assign y0=(s0&s1&~s2&i);

assign y0=(s0&s1&s2&i);

endmodule

**Test Bench:**

module demux1x8\_tb();

reg i,s0,s1,s2;

wire y0,y1,y2,y3,y4,y5,y6,y7;

oneeightdemuxbehavioral m(i,s0,s1,s2,y0,y1,y2,y3,y4,y5,y6,y7);

initial

begin

i=0;

s0=0;

s1=0;

s2=0;

#500 $finish();

end

always #5 i=~i;

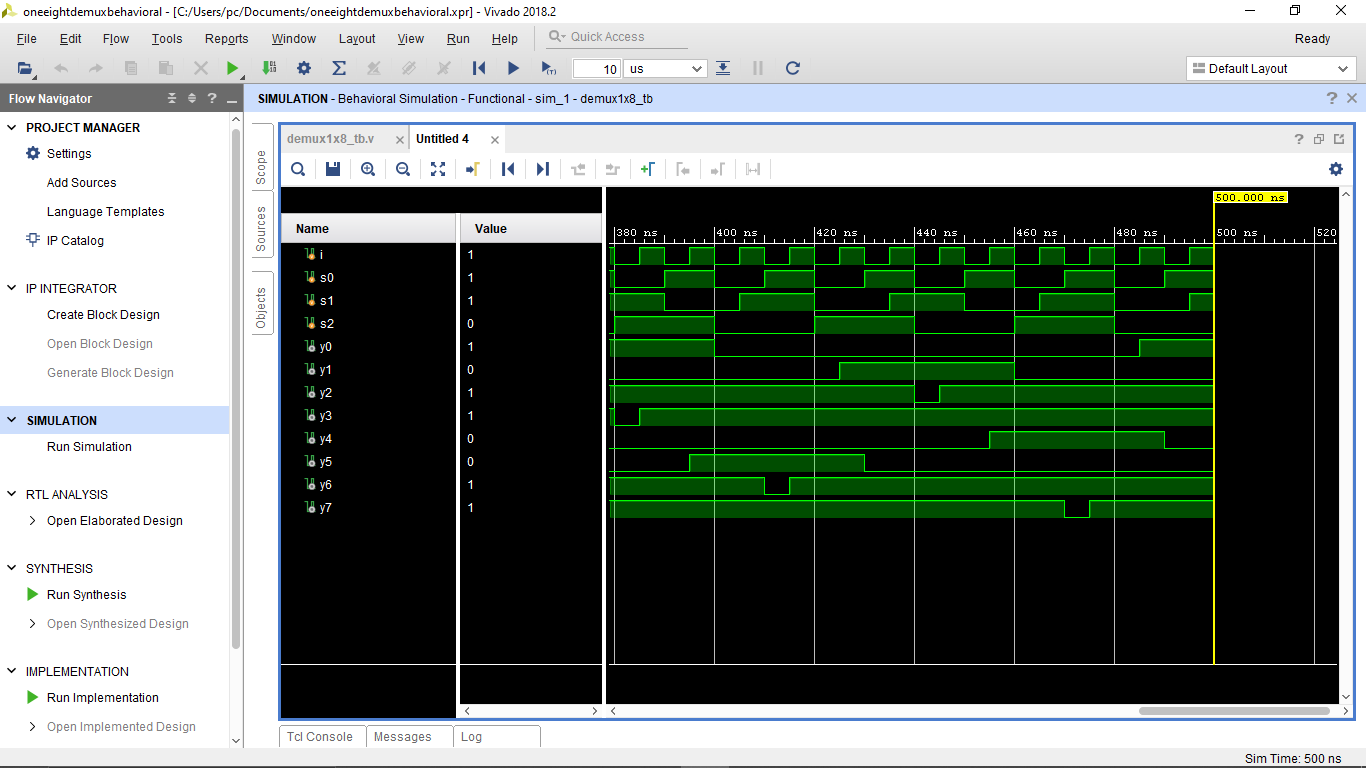
always #10 s0=~s0;

always #15 s1=~s1;

always #20 s2=~s2;

endmodule

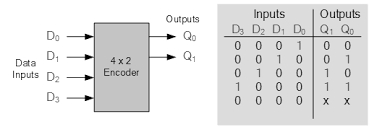
**Wave forms:**

****

**4x2 ENCODER**

A 4x2 encoder is a digital logic circuit that takes four input lines and produces two output lines based on the input data. It encodes one of the active input lines (if any) into a binary code on the output lines. This type of encoder is often used in digital systems to reduce the number of input lines required to represent data**.**

**Truth table and circuit diagram:**

****

**RTL Code:**

module fourtotwoencoderbehavioral(i0,i1,i2,i3,b0,b1);

input i0,i1,i2,i3;

output reg b0,b1;

always@(\*)

begin

case({i0,i1,i2,i3})

4'b1000: b0=0;

4'b0100: b0=0;

4'b0010: b0=1;

4'b0001: b0=1;

endcase

case({i0,i1,i2,i3})

4'b1000: b1=0;

4'b0100: b1=1;

4'b0010: b1=0;

4'b0001: b1=1;

endcase

end

endmodule

**Test Bench:**

module encoder4\_2\_tb();

reg i0,i1,i2,i3;

wire b0,b1;

fourtotwoencoderbehavioral e(i0,i1,i2,i3,b0,b1);

initial

begin

i0=0;

i1=0;

i2=0;

i3=0;

#500 $finish();

end

always #5 i0=~i0;

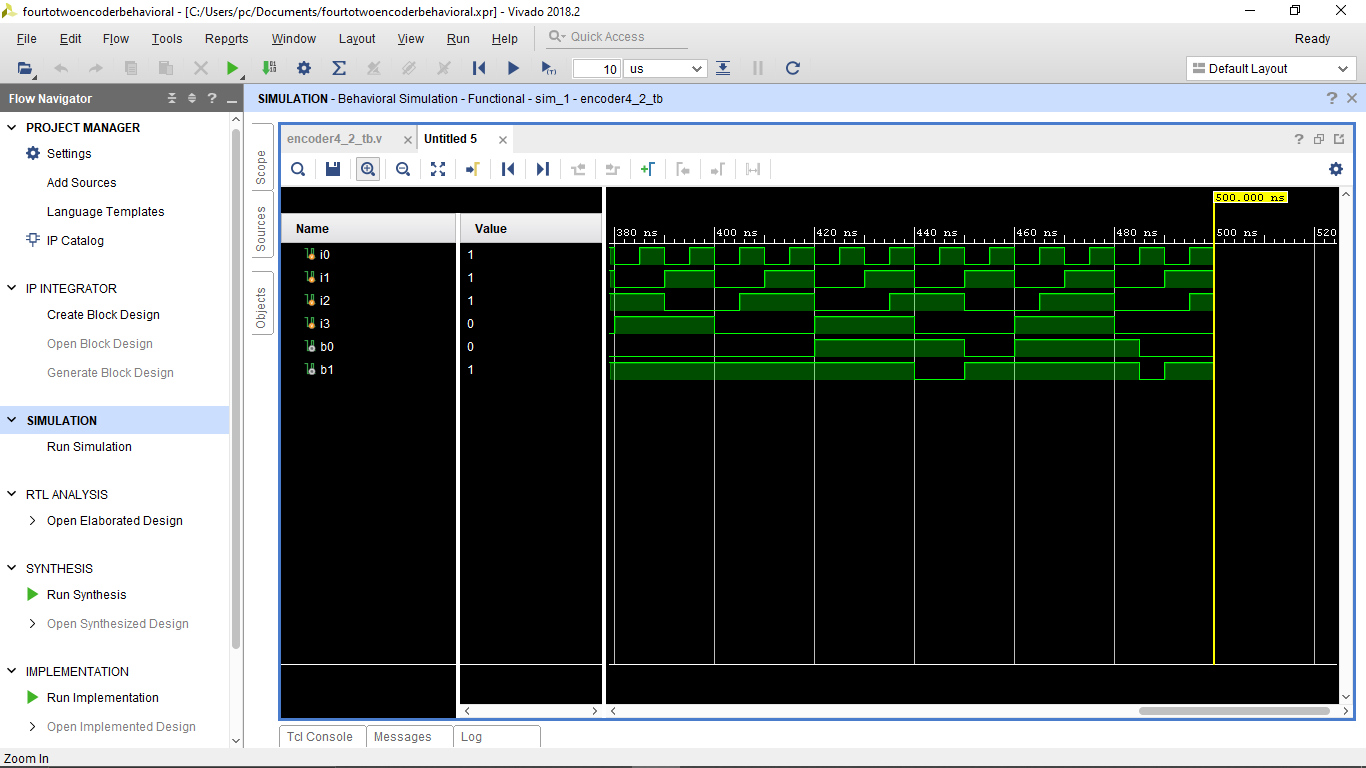
always #10 i1=~i1;

always #15 i2=~i2;

always #20 i3=~i3;

endmodule

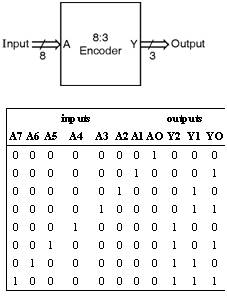
**Waveforms:**

****

**8x3 ENCODER**

An 8x3 encoder is a digital logic circuit that takes 8 input lines and produces 3 output lines, which represent the binary code for the active input line. It is used to reduce the number of input lines required to represent data by encoding one of the active input lines into binary form on the output lines.

**TruthTable and Circuit Diagram:**



**RTL Code:**

module eighttothreeencoderdataflow(i0,i1,i2,i3,i4,i5,i6,i7,b0,b1,b2);

input i0,i1,i2,i3,i4,i5,i6,i7;

output b0,b1,b2;

assign b0=i4|i5|i6|i7;

assign b1=i2|i3|i6|i7;

assign b2=i1|i3|i5|i7;

endmodule

**Test bench:**

**2x4 Decoder**

module decoder24\_behaviour(en,a,b,y);

input en,a,b;

output reg [3:0]y;

always @(en,a,b)

begin

if(en==0)

begin

if(a==1'b0 & b==1'b0) y=4'b1110;

else if(a==1'b0 & b==1'b1) y=4'b1101;

else if(a==1'b1 & b==1'b0) y=4'b1011;

else if(a==1 & b==1) y=4'b0111;

else y=4'bxxxx;

end

else

y=4'b1111;

end

endmodule

**3:8 Decoder:**

module decoder3\_to\_8( in,out, en);

input [2:0] in;

input en;

output [7:0] out;

reg [7:0] out;

always @( in or en)

begin

if (en)

begin

out=8'd0;

case (in)

3'b000: out[0]=1'b1;

3'b001: out[1]=1'b1;

3'b010: out[2]=1'b1;

3'b011: out[3]=1'b1;

3'b100: out[4]=1'b1;

3'b101: out[5]=1'b1;

3'b110: out[6]=1'b1;

3'b111: out[7]=1'b1;

default: out=8'd0;

endcase

end

else

out=8'd0;

end

endmodule

**4:16 Decoder:**

module decoder\_4to16 (

input enable,

input [3:0] binary\_in,

output reg [15:0] decoder\_out);

always @ (enable or binary\_in)

begin

if (enable) begin

case (binary\_in)

4'h0 : decoder\_out = 16'h0001;

4'h1 : decoder\_out = 16'h0002;

4'h2 : decoder\_out = 16'h0004;

4'h3 : decoder\_out = 16'h0008;

4'h4 : decoder\_out = 16'h0010;

4'h5 : decoder\_out = 16'h0020;

4'h6 : decoder\_out = 16'h0040;

4'h7 : decoder\_out = 16'h0080;

4'h8 : decoder\_out = 16'h0100;

4'h9 : decoder\_out = 16'h0200;

4'hA : decoder\_out = 16'h0400;

4'hB : decoder\_out = 16'h0800;

4'hC : decoder\_out = 16'h1000;

4'hD : decoder\_out = 16'h2000;

4'hE : decoder\_out = 16'h4000;

4'hF : decoder\_out = 16'h8000;

default : decoder\_out = 0;

endcase

end

else begin

decoder\_out = 0;

end

end

endmodule

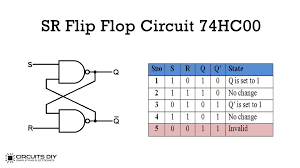
**Sequential Circuits:**

**Flipflops:**

**SR Flipflop:**

A Set-Reset (SR) flip-flop, also known as a flip-flop or latch, is a fundamental digital electronic circuit used in digital logic and sequential logic systems. It's a bistable multivibrator, which means it has two stable states and can store one bit of information. The two stable states are often denoted as "0" and "1" or "reset" and "set."

**Truth table and Circuit Diagram:**

****

**RTL Code:**

module sr\_ff\_behavioral(s,r,clk,q,qb);

input s,r,clk;

output reg q,qb;

always @(posedge clk)

begin

if(s==1)

begin

q=1;

qb=0;

end

else if(r==1)

begin

q=0;

qb=1;

end

else if(s==0&r==0)

begin

q <= q;

qb <= qb;

end

end

endmodule

**Testbench:**

module sr\_ff\_tb();

reg s,r,clk;

wire q,qb;

sr\_ff\_behavioral f(s,r,clk,q,qb);

initial

begin

s=0;

r=0;

clk=0;

#500 $finish();

end

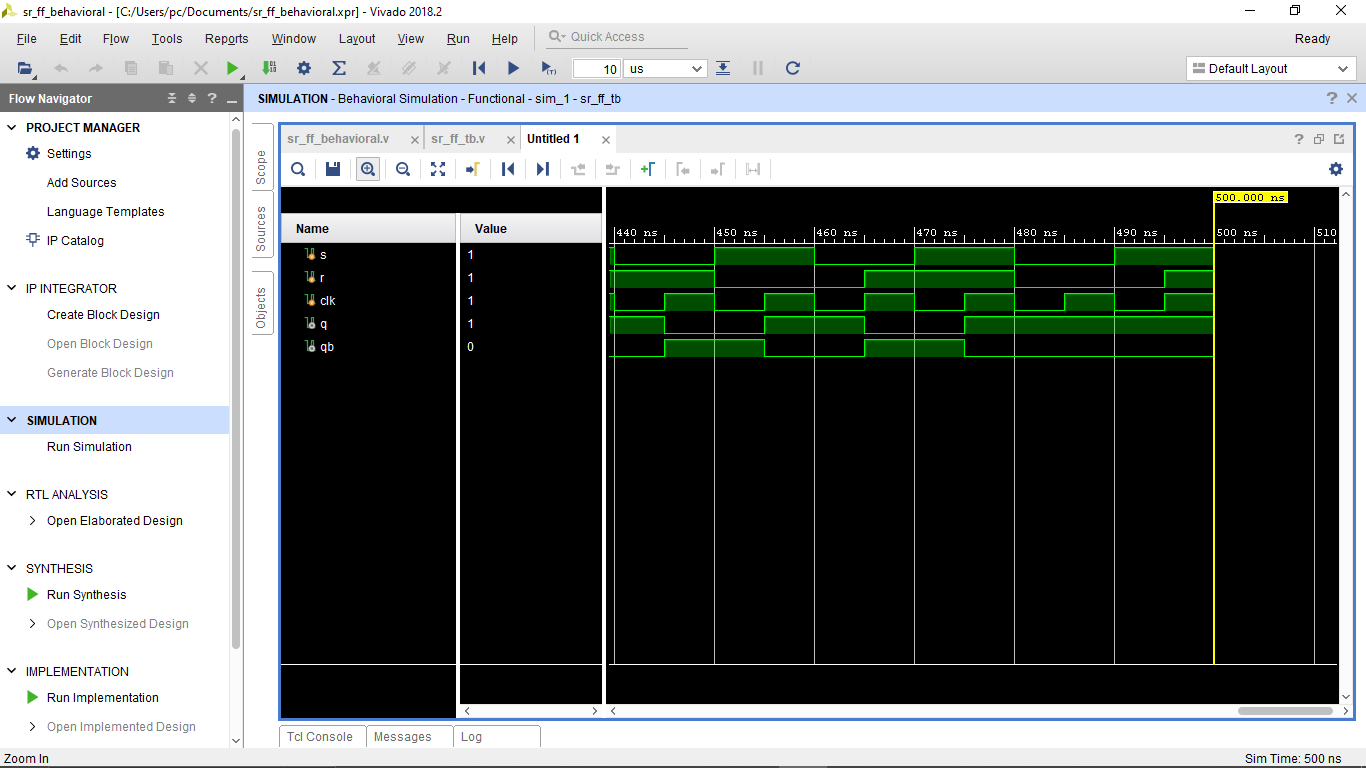
always #5 clk=~clk;

always #10 s=~s;

always #15 r=~r;

endmodule

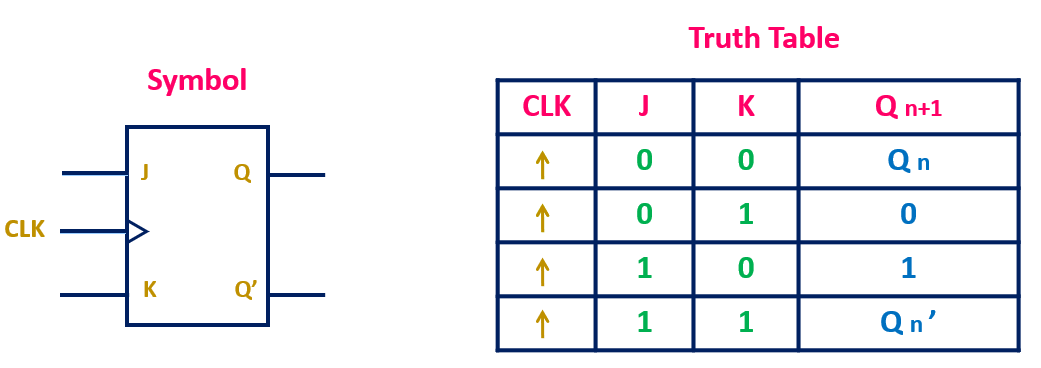
**Waveforms:**

****

**JK Flip flop:**

A JKflip-flop is another type of digital flip-flop or bistable multivibrator, commonly used in digital electronic circuits. It is an improvement over the SR (Set-Reset) flip-flop because it eliminates the problematic "forbidden" state and offers more functionality. The name "JK" comes from the inputs J (Jack) and K (Kilby), who were two of the inventors of the flip-flop circuit.

**Truthtable and Circuit Diagram:**



**RTL Code:**

module jkff(j,k,clk,rst,q); input j,k,clk,rst;

output reg q; always@(posedge clk) begin

if(rst) q<=0;

else begin

end

case({j,k}) 2'b00:q<=q;

2'b01:q<=0;

2'b10:q<=1;

2'b11:q<=~q; endcase

end endmodule

**TestBench:**

module jkfft\_b(); reg j,k,rst,clk; wire q;

jkff uut(j,k,clk,rst,q); module jkff\_testbench;

// Inputs reg j;

reg k; reg clk; reg rst;

// Outputs wire q;

// Instantiate the Unit Under Test (UUT) jkff uut (

.j(j),

.k(k),

.clk(clk),

.rst(rst),

.q(q)

);

always

#5 clk=~clk;

task jkin(input[1:0]a); begin

end

@(negedge clk)

{j,k}=a;

endtask

task reset(input b); begin

end

@(negedge clk) rst=b;

endtask initial

begin rst=0;

j=0; k=0;

clk=0; #5;

jkin(0);

jkin(1);

jkin(2);

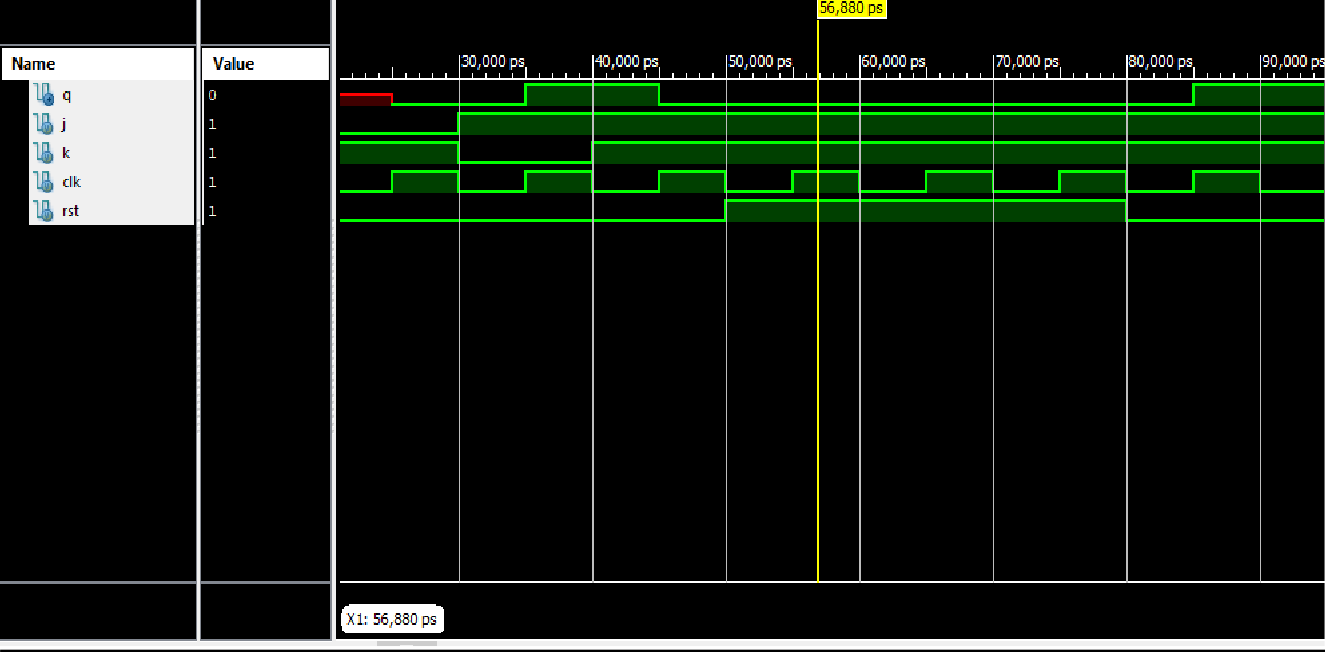
jkin(3);

reset(1); #30;

reset(0); end

endmodule

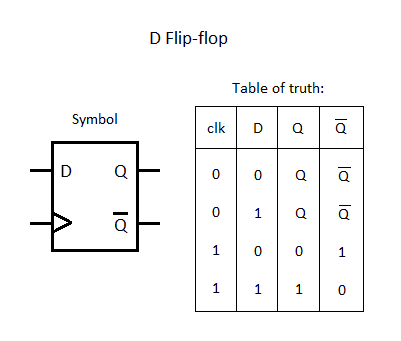
**WaveForms:**



**D-Flipflop:**

A data flip-flop is a type of digital electronic circuit that can store a single bit of binary data and transfer it to an output in response to a clock signal. It typically has a data input (D), a clock input (CLK), and outputs (Q and Q̅), and it latches or captures the value at the data input when triggered by the clock signal's edge.

**Truth Table and Circuit:**



**RTL Code:**

module dff(q,rst,clk,d); input clk,d,rst;

output reg q; always@(posedge clk) begin

if(rst) q<=0;

else q<=d;

end

endmodule

**Testbench:**

module d\_ff\_testbench(); reg d,clk,rst;

wire q;

d\_ff uut(q,rst,clk,d); always

begin

#5 clk=~clk;

end

task din(input a); begin

end

@(negedge clk)

d=a;

endtask

task reset(input b); begin

end

@(negedge clk) rst=b;

endtask initial begin

d=0;

rst=0; clk=0; #5;

din(0);

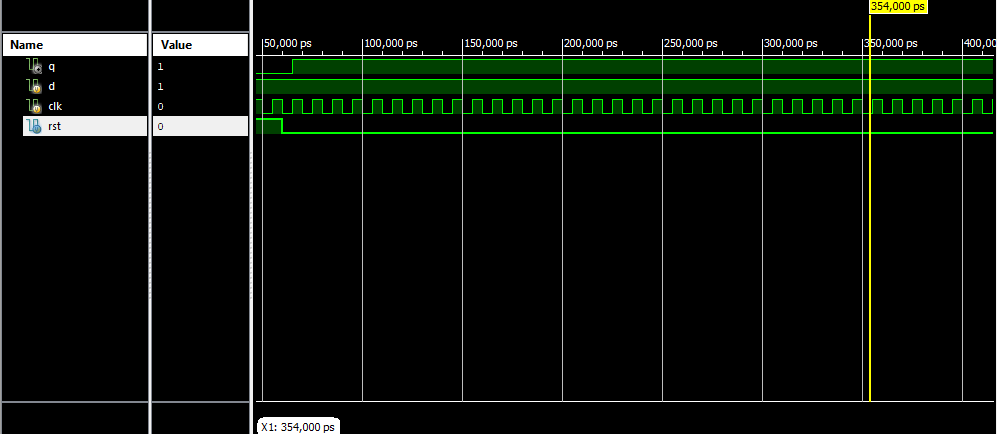
din(1);

reset(1); #30;

reset(0);

end

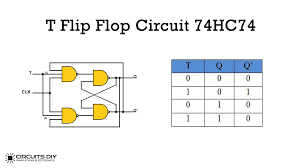
endmodule

**Waveforms:**

**T-flipflop:**

A T flip-flop, also known as a Toggle flip-flop, is a type of digital flip-flop or bistable multivibrator used in digital electronic circuits. It is similar in functionality to a JK flip-flop but simplified, as it has a single input (T) and two outputs (Q and Q̅). The T input is used to toggle or change the state of the flip-flop on the rising or falling edge of a clock signal.

**Truthtable and Circuit Diagram:**

****

module t\_ff\_behavioral(t,clk,reset,q,qb);

input t,clk,reset;

output reg q,qb;

always @(posedge clk)

begin

if(reset)

q<=0;

else

if(t)

q<=~q;

else

q<=q;

end

assign qb=~q;

endmodule

**Testbench:**

module t\_ff\_tb();

reg t,clk,reset;

wire q,qb;

t\_ff\_behavioral tf(t,clk,reset,q,qb);

initial

begin

t=0;

clk=0;

reset=1;

#10 reset=0;

#500 $finish();

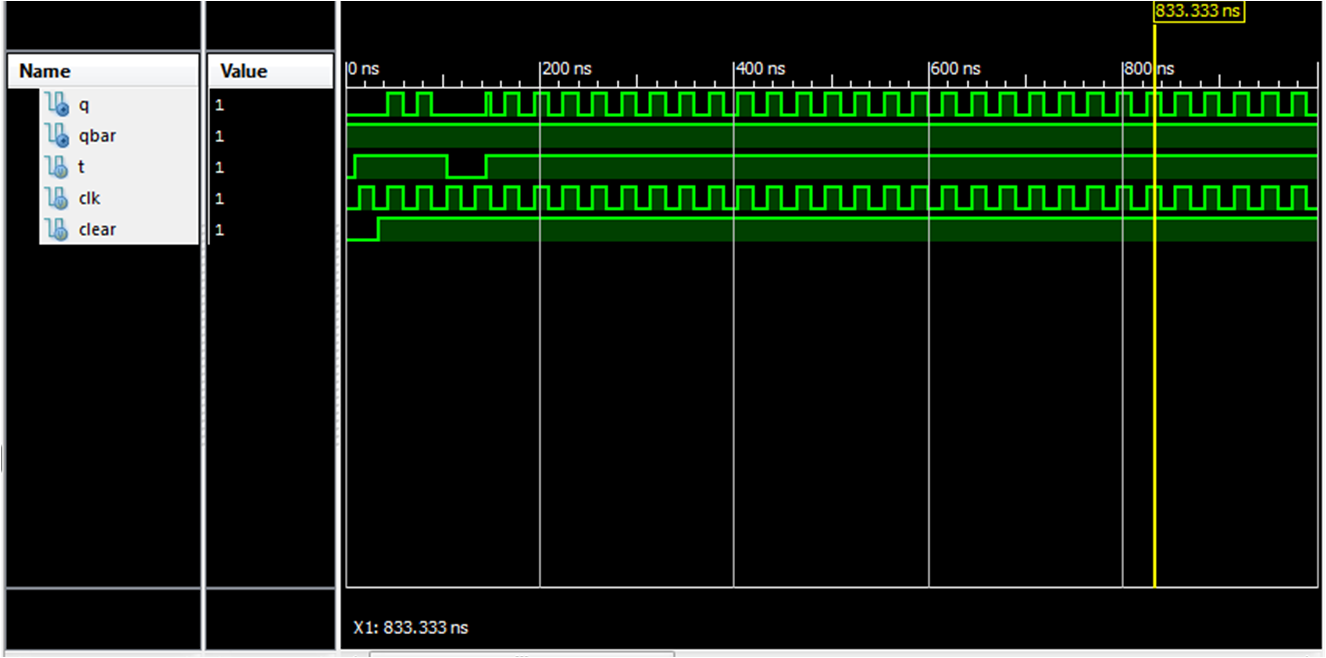
end

always #5 clk=~clk;

always #30 t=~t;

endmodule

**Waveforms:**



**N-Bit updown Counter:**

`timescale 1ns / 1ps

module updown\_counter(data\_in,reset\_n,ld,U\_D,clk,count);

input reset\_n,ld,U\_D,clk;

input [2:0] data\_in;

output reg [2:0] count;

always@(posedge clk or posedge reset\_n)

begin

if(reset\_n)

count<=3'b000;

else if(ld)

count<=data\_in;

else

begin

if(U\_D)

count<=count+1'b1;

else

count<=count-1'b1;

end

end

endmodule

**Testbench:**

`timescale 1ns / 1ps

module updown\_counter\_tb();

reg reset\_n, ld, U\_D, clk;

reg [2:0] data\_in;

wire [2:0] count;

updown\_counter uut(.data\_in(data\_in),.reset\_n(reset\_n),.ld(ld),

.U\_D(U\_D),.clk(clk),.count(count));

always #5 clk = ~clk;

initial begin

reset\_n = 1;ld = 0;U\_D = 0; clk = 0;

data\_in = 3'b000;

#5 reset\_n = 0;

#5 ld = 1;

#5 ld = 0;

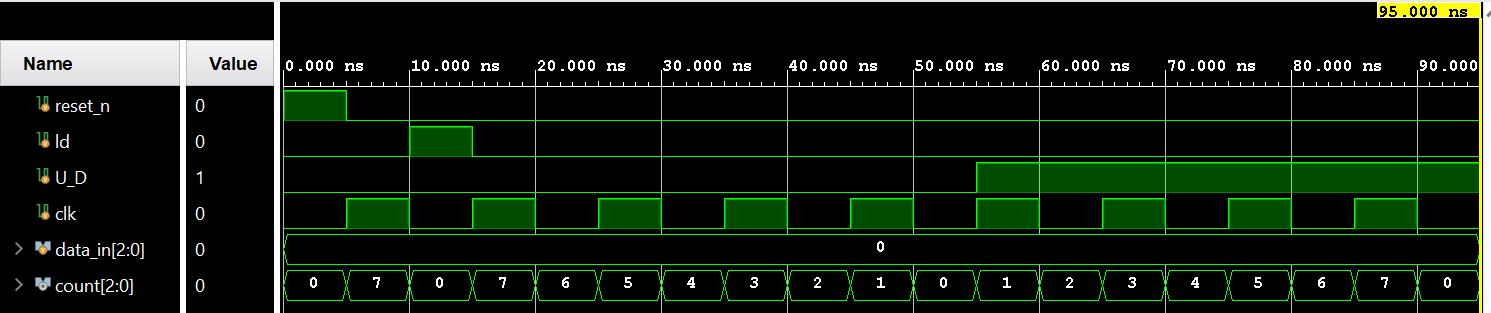
#40 U\_D = 1;

#40 $finish;

end

endmodule

**Waveforms:**



**MOD N Counter(N=10):**

module MOD10\_counter(clk,rst,dout,pwr,loaden,load); input clk ,rst,pwr,loaden;

output reg[3:0]dout; input [3:0]load; always@(posedge clk)

begin

if(~pwr) dout<=0; else begin

end endmodule

end

if(rst) dout<=0;

else if(loaden) begin if(load>4'b1001) dout<=dout+1; else

dout<=load; end

else if(dout==4'b1001) dout<=0;

else if(dout!=4'b1001) dout<=dout+1;

**Testbench:**

module MOD10\_counter\_tb;

// Inputs reg clk;

reg rst; reg pwr;

reg loaden; reg [3:0] load;

// Outputs wire [3:0] dout;

// Instantiate the Unit Under Test (UUT) MOD10\_counter uut (

.clk(clk),

.rst(rst),

.dout(dout),

.pwr(pwr),

.loaden(loaden),

.load(load)

);

always

#5 clk=~clk; task initialise; begin

end

clk=0; rst=0; pwr=0;

loaden=0;

load=0;

endtask

task power(input a); begin

end

@(negedge clk) pwr=a;

endtask task reset; begin

@(negedge clk) rst=1; @(negedge clk) rst=0;

end

endtask

task loadtask(input [3:0]b); begin

@(negedge clk) loaden=1; load=b; @(negedge clk) loaden=0;

end

endtask initial begin

end

initialise; power(1); #100;

reset; #50;

loadtask(4'b0111); #20;

loadtask(4'b1011); #20;

$finish;

Endmodule

**Gray Code Counter:**

`timescale 1ns / 1ps

module gray\_counter(clk,data\_in,ld,rst, count);

input clk, rst,ld;

input [3:0] data\_in;

output reg [3:0] count;

reg q0, q1, q2;

reg [3:0] temp;

always @ (posedge clk)

begin

if (rst)

begin

temp <= 4'b0;

{q0,q1,q2}<=3'b000;

count <= 4'b0;

end

else if(ld)

begin

temp <=data\_in;

{q0,q1,q2}<={data\_in[2],data\_in[1],data\_in[0]};

count<={data\_in[3],data\_in[2],data\_in[1],data\_in[0]};

end

else

begin

temp <= temp + 1'b1;

q2 <= temp[3] ^ temp[2];

q1 <= temp[2] ^ temp[1];

q0 <= temp[1] ^ temp[0];

count = {temp[3], q2, q1, q0};

end

end

endmodule

**Testbench**:

`timescale 1ns / 1ps

module gray\_counter\_tb;

reg clk;

reg rst;

reg ld;

reg [3:0] data\_in;

wire [3:0] count;

gray\_counter uut (

.clk(clk),

.rst(rst),

.ld(ld),

.data\_in(data\_in),

.count(count)

);

always #5 clk=~clk;

initial

begin

clk=0;rst=1;data\_in=4'b0000;ld=0;

#5 rst=0;ld=1;

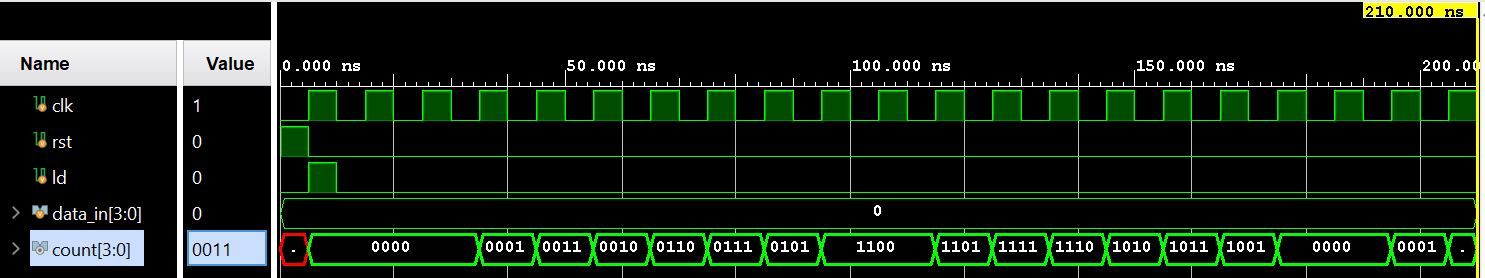
#5 ld=0;

#200 $finish();

end

endmodule

**Waveforms:**



**Ring Counter:**

`timescale 1ns / 1ps

module ringcounter(

input clk,

input set,

output reg [3:0] count

);

always @ (posedge clk)

begin

if(set)

count<=4'b1000;

else

begin

count<=(count<<1);

count[0]<=count[3];

end

end

endmodule

**Testbench:**

`timescale 1ns / 1ps

module testbench();

reg clk,set;

wire [3:0] count;

ringcounter dut(.clk(clk),.set(set),.count(count));

always #5 clk=~clk;

initial begin

clk=0;

set=0;

#5 set=1;

#5 set=0;

#80 $finish();

end

endmodule

**Waveforms:**



**Twisted Ring Counter:**

`timescale 1ns / 1ps

module johnson\_counter(

input clk,

input reset\_n,

output reg [3:0] count

);

always @ (posedge clk or negedge reset\_n)

begin

if(~reset\_n)

count<=4'b0000;

else

count<={{count[2:0]},{~count[3]}};

end

endmodule

**Testbench:**

`timescale 1ns / 1ps

module testbench();

reg clk,reset\_n;

wire [3:0] count;

johnson\_counter dut(.clk(clk),.reset\_n(reset\_n),.count(count));

always #5 clk=~clk;

initial begin

clk=0;

reset\_n=0;

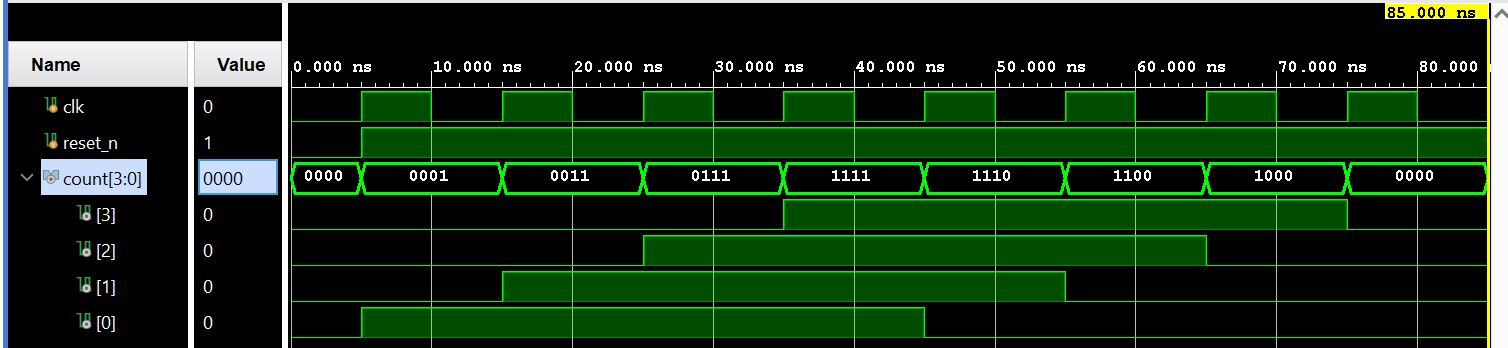
#5 reset\_n=1;

#80 $finish();

end

endmodule

**Waveforms:**



**Minor Projects:**

#### **1.FIFO:**

module FIFO(clk,Data\_in,Data\_out,rst,wr\_en,rd\_en,full,empty

);

parameter width = 8; parameter depth = 16; parameter addr\_width = 4;

input clk,rd\_en,wr\_en,rst; input [width-1:0]Data\_in;

output reg[width-1:0]Data\_out; output full,empty;

reg [addr\_width-1:0]wr\_ptr; reg [addr\_width-1:0]rd\_ptr;

reg [addr\_width:0]status\_count; reg [width-1:0] MEM [depth-1:0];

assign full = (status\_count == depth)?1:0; assign empty = (status\_count == 0)?1:0; integer i;

//write ponter

always @(posedge clk) begin

if(rst)

wr\_ptr<=0;

end

else if(wr\_en)

wr\_ptr<=wr\_ptr+1;

//read ponter always@(posedge clk) begin

if(rst)

rd\_ptr<=0;

end

else if(rd\_en)

rd\_ptr<=rd\_ptr+1;

//status count

always @(posedge clk) begin

if(rst)

status\_count<=0;

end

else if((wr\_en && !full)&&(!rd\_en)) status\_count<=status\_count+1;

else if((rd\_en && !empty)&&(!wr\_en)) status\_count<=status\_count-1;

else if((wr\_en && !full)&&(rd\_en && !empty)) status\_count<=status\_count;

always@(posedge clk) begin

if(rst)

begin

for(i=0; i<depth; i=i+1)

begin

else

end begin

end

if(wr\_en)

MEM[i]<=0;

end endmodule

**Testbench:**

module FIFO\_TB;

parameter width=8; parameter depth=16; integer i;

// Inputs reg clk;

reg [7:0] Data\_in; reg rst;

reg wr\_en; reg rd\_en;

// Outputs

wire [7:0] Data\_out; wire full;

wire empty;

always #5 clk = ~clk;

// Instantiate the Unit Under Test (UUT) FIFO uut (

.clk(clk),

.Data\_in(Data\_in),

.Data\_out(Data\_out),

.rst(rst),

.wr\_en(wr\_en),

.rd\_en(rd\_en),

.full(full),

.empty(empty)

);

task initialise;

begin clk = 0;

Data\_in = 0;

rst = 0;

wr\_en = 0;

rd\_en = 0; end

endtask

task reset;

begin

end endtask

@(negedge clk) rst<=1; @(negedge clk) rst<=0;

task write(input [width-1:0]D); begin

end endtask

@(negedge clk) Data\_in<=D;

initial

begin

initialise; reset; wr\_en<=1;

for(i=0;i<depth;i=i+1)

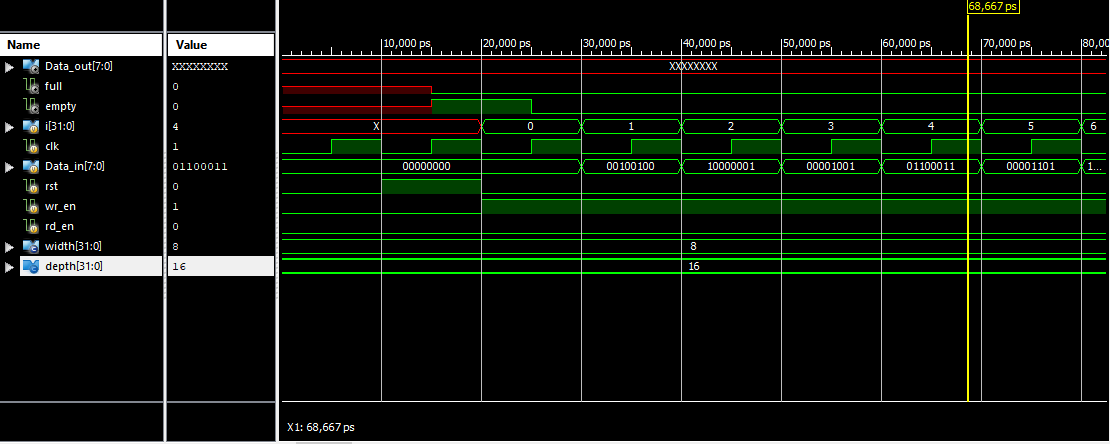
begin

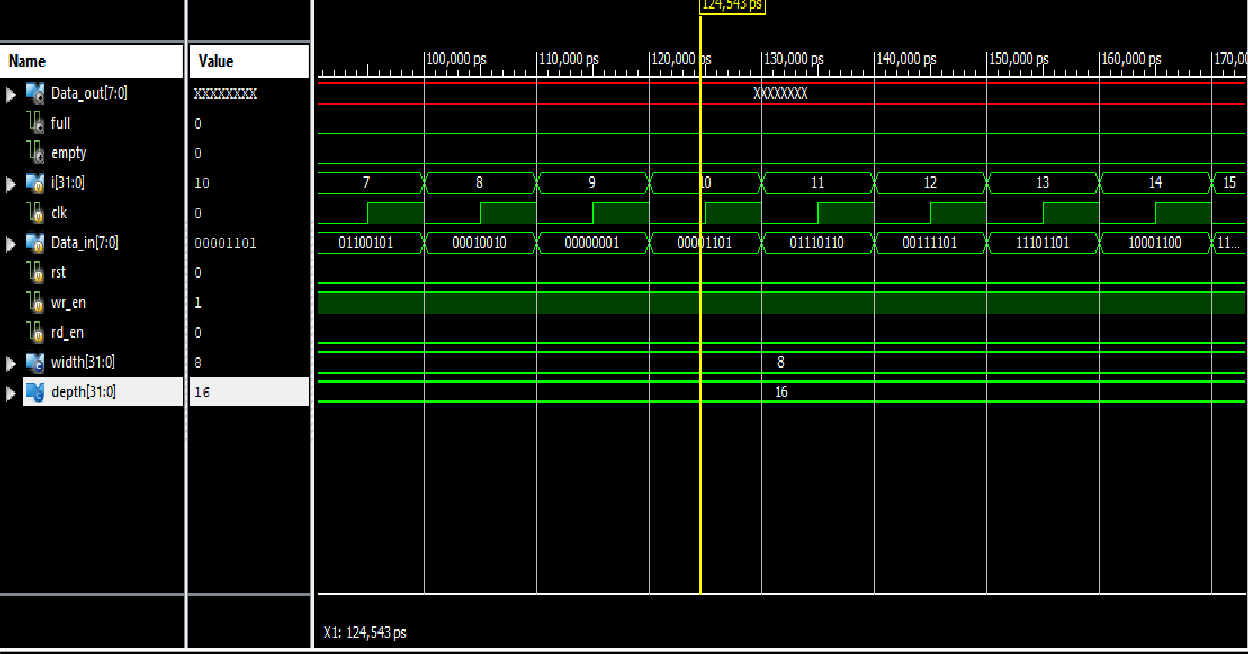
end

end wr\_en<=0; rd\_en<=1; #100;

write($random);

endmodule

**Waveforms:**



**2.Ram:**

module singleRAM(Clk,wr\_en,rd\_en,Addr,Data

);

input Clk, wr\_en, rd\_en; input [3:0] Addr;

inout [7:0] Data;

reg [7:0] temp\_data;

reg [7:0] MEMORY [15:0];

assign Data = (rd\_en && !wr\_en)? temp\_data: 8'bzzzzzzzz; always@(posedge Clk)

begin

if(wr\_en && !rd\_en) MEMORY[Addr] <= Data;

else if (rd\_en && !wr\_en) temp\_data <= MEMORY[Addr];

end endmodule

**Testbench:**

module singleRAM\_testbench;

// Inputs reg Clk; reg wr\_en; reg rd\_en;

reg [3:0] Addr;

reg [7:0]temp;

// Bidirs

wire [7:0] Data; integer i,j;

// Instantiate the Unit Under Test (UUT) singleRAM uut (

.Clk(Clk),

.wr\_en(wr\_en),

.rd\_en(rd\_en),

.Addr(Addr),

.Data(Data)

);

always #10 Clk = ~Clk;

assign Data = (wr\_en && !rd\_en)?temp:8'hzz; task initialise;

begin

end endtask

Clk = 0;

wr\_en = 0;

rd\_en = 0;

Addr = 0;

temp = 0;

task write( input [3:0]A, input [7:0] D); begin

end endtask

@(negedge Clk) wr\_en = 1;

rd\_en = 0; Addr = A; temp = D;

task read(input [3:0]Ar); begin

end endtask initial

@(negedge Clk) wr\_en = 0;

rd\_en = 1; Addr = Ar;

begin

initialise;

for(i =0; i<16; i = i+1) begin

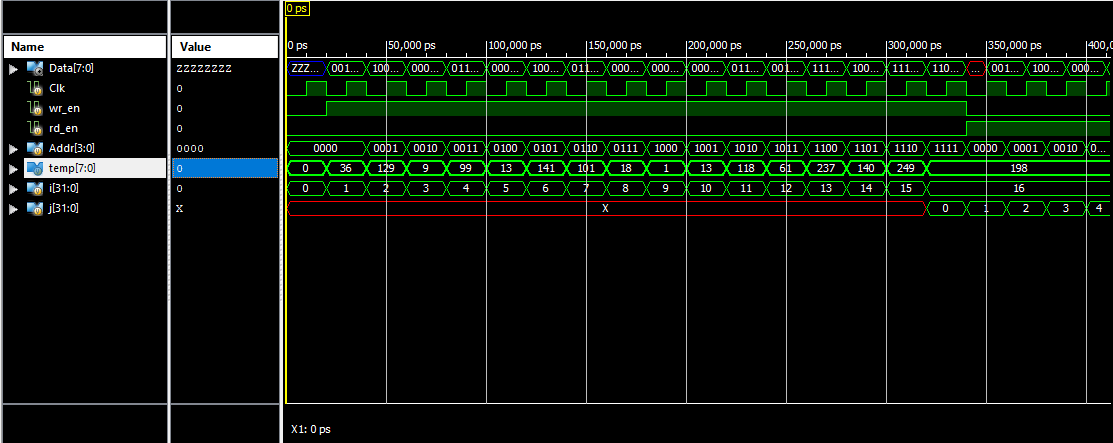
write(i, $random);

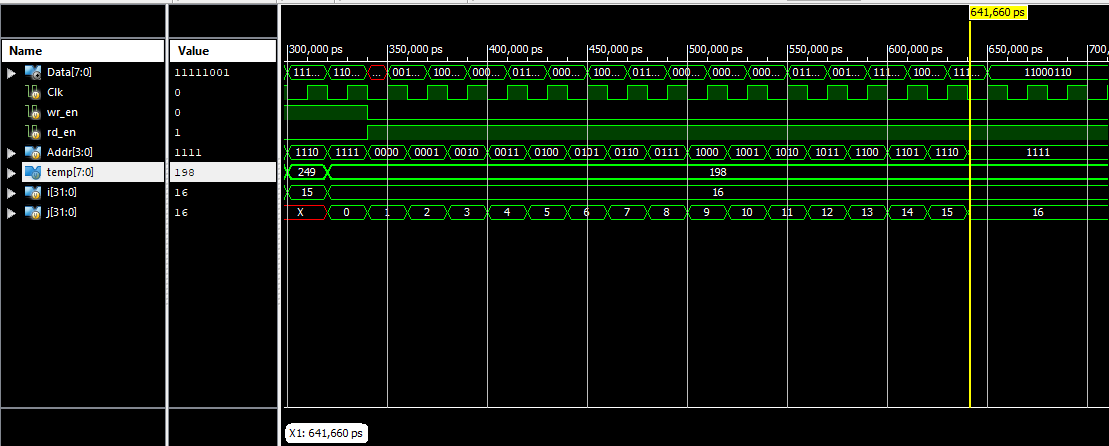
end

for(j=0; j<16; j=j+1) begin

end endmodule

**Waveforms:**

****



**3.Universal shift register:**

module universal\_shift\_register (

input wire clk,

input wire rst,

input wire shift\_left,

input wire shift\_right,

input wire shift\_in,

output wire [3:0] shift\_data

);

reg [3:0] shift\_reg;

always @(posedge clk or posedge rst) begin

if (rst)

shift\_reg <= 4'b0000;

else if (shift\_left)

shift\_reg <= {shift\_reg[2:0], shift\_in};

else if (shift\_right)

shift\_reg <= {shift\_in, shift\_reg[3:1]};

end

assign shift\_data = shift\_reg;

endmodule

**Testbench:**

module testbench;

reg clk;

reg rst;

reg shift\_left;

reg shift\_right;

reg shift\_in;

wire [3:0] shift\_data;

// Instantiate the universal\_shift\_register module

universal\_shift\_register uut (

.clk(clk),

.rst(rst),

.shift\_left(shift\_left),

.shift\_right(shift\_right),

.shift\_in(shift\_in),

.shift\_data(shift\_data)

);

// Clock generation

always begin

#5 clk = ~clk;

end

initial begin

clk = 0;

rst = 1;

shift\_left = 0;

shift\_right = 0;

shift\_in = 0;

// Reset initialization

#10 rst = 0;

// Shift in data

#10 shift\_in = 1;

#10 shift\_in = 0;

// Shift right

shift\_right = 1;

#20;

shift\_right = 0;

// Shift left

shift\_left = 1;

#20;

shift\_left = 0;

// Hold mode

#10;

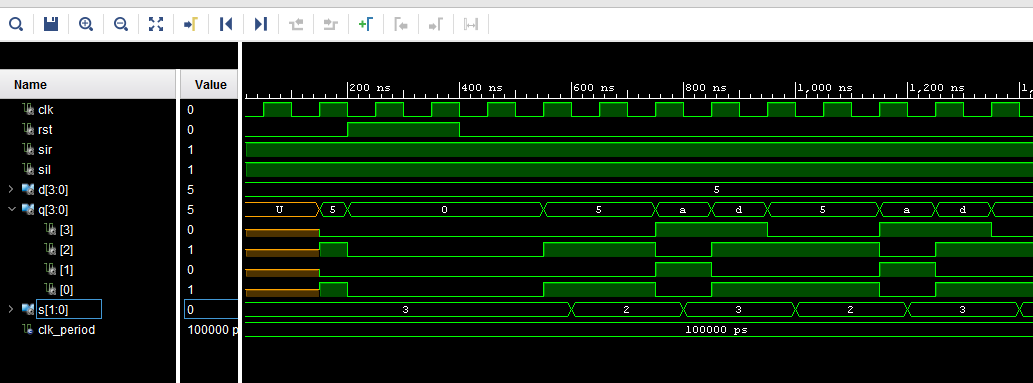
// Finish simulation

$finish;

end

endmodule

**Waveforms:**



**4.Carry lookahead adder:**

module carry\_lookahead\_adder (

input wire [3:0] A,

input wire [3:0] B,

input wire Cin,

output wire [4:0] Sum,

output wire Cout

);

wire [3:0] P, G;

wire [3:0] C;

assign P = A ^ B;

assign G = A & B;

assign C[0] = Cin;

assign C[1] = G[0] | (P[0] & Cin);

assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Cin);

assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & Cin);

assign Sum = A + B + Cin;

assign Cout = 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] & Cin);

endmodule

**Testbench:**

module testbench;

reg [3:0] A;

reg [3:0] B;

reg Cin;

wire [4:0] Sum;

wire Cout;

// Instantiate the carry\_lookahead\_adder module

carry\_lookahead\_adder uut (

.A(A),

.B(B),

.Cin(Cin),

.Sum(Sum),

.Cout(Cout)

);

initial begin

A = 4'b0000;

B = 4'b0000;

Cin = 0;

// Test Case 1

#10 A = 4'b0110;

#10 B = 4'b1011;

#10 Cin = 0;

// Test Case 2

#10 A = 4'b1100;

#10 B = 4'b0101;

#10 Cin = 1;

// Test Case 3

#10 A = 4'b1111;

#10 B = 4'b1111;

#10 Cin = 1;

// Add more test cases here if needed

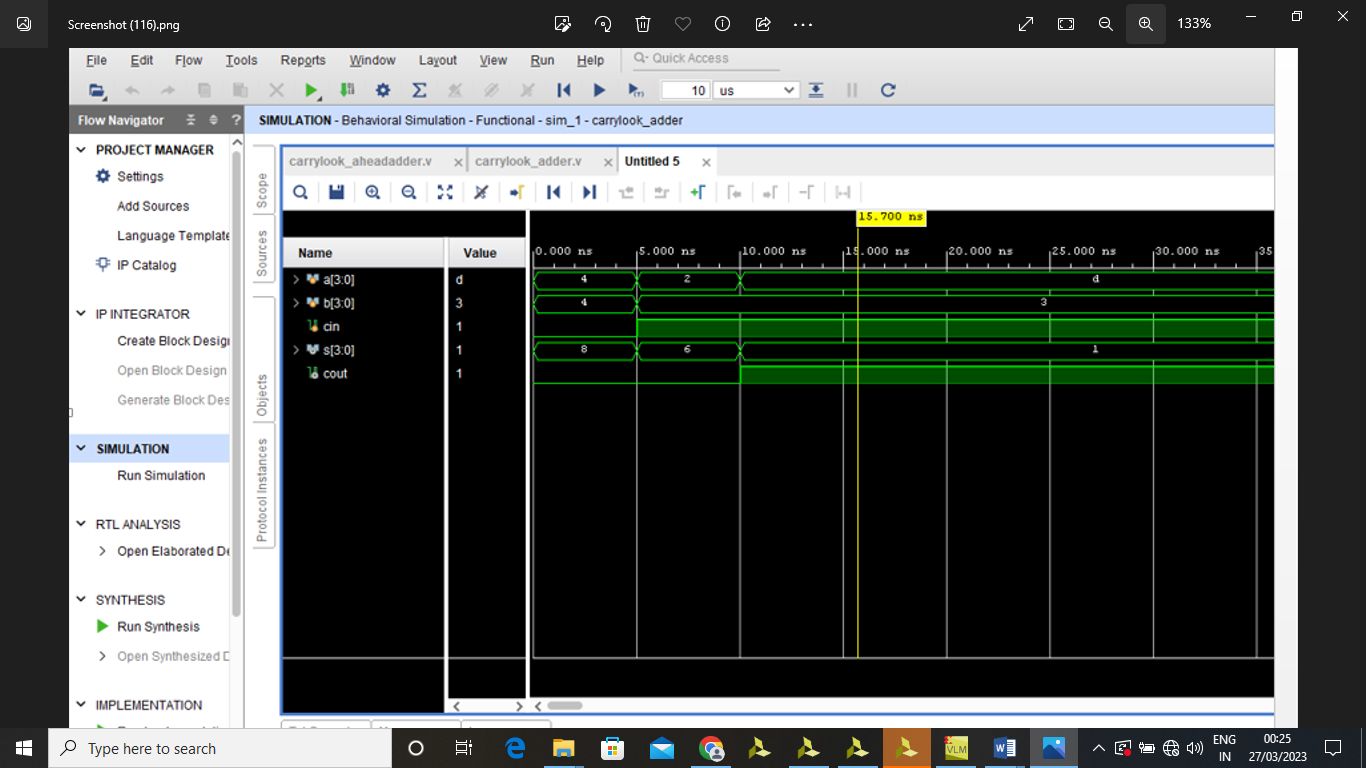
// Finish simulation

$finish;

end

endmodule

**Waveforms:**



**5.Universal Shift Register:**

module Universal\_Shift\_Register (

input wire clk,

input wire rst,

input wire shift\_en,

input wire serial\_in,

input wire parallel\_in,

input wire shift\_left,

output wire [7:0] serial\_out,

output wire [7:0] parallel\_out

);

reg [7:0] shift\_reg;

always @(posedge clk or posedge rst) begin

if (rst) begin

shift\_reg <= 8'b0;

end else if (shift\_en) begin

if (shift\_left) begin

shift\_reg <= {shift\_reg[6:0], serial\_in};

end else begin

shift\_reg <= {serial\_in, shift\_reg[7:1]};

end

end else if (!shift\_en && parallel\_in) begin

shift\_reg <= parallel\_in;

end

end

assign serial\_out = shift\_reg[0];

assign parallel\_out = shift\_reg;

endmodule

**Test bench:**

module testbench;

reg clk;

reg rst;

reg shift\_en;

reg serial\_in;

reg parallel\_in;

reg shift\_left;

wire [7:0] serial\_out;

wire [7:0] parallel\_out;

Universal\_Shift\_Register uut (

.clk(clk),

.rst(rst),

.shift\_en(shift\_en),

.serial\_in(serial\_in),

.parallel\_in(parallel\_in),

.shift\_left(shift\_left),

.serial\_out(serial\_out),

.parallel\_out(parallel\_out)

);

always begin

#5 clk = ~clk;

end

initial begin

clk = 0;

rst = 0;

shift\_en = 0;

serial\_in = 0;

parallel\_in = 8'b0;

shift\_left = 0;

rst = 1;

#10 rst = 0;

parallel\_in = 8'b10101010;

shift\_en = 0;

serial\_in = 0;

shift\_left = 0;

#10;

parallel\_in = 8'b0;

shift\_en = 1;

serial\_in = 0;

shift\_left = 0;

#10;

parallel\_in = 8'b0;

shift\_en = 1;

serial\_in = 0;

shift\_left = 1;

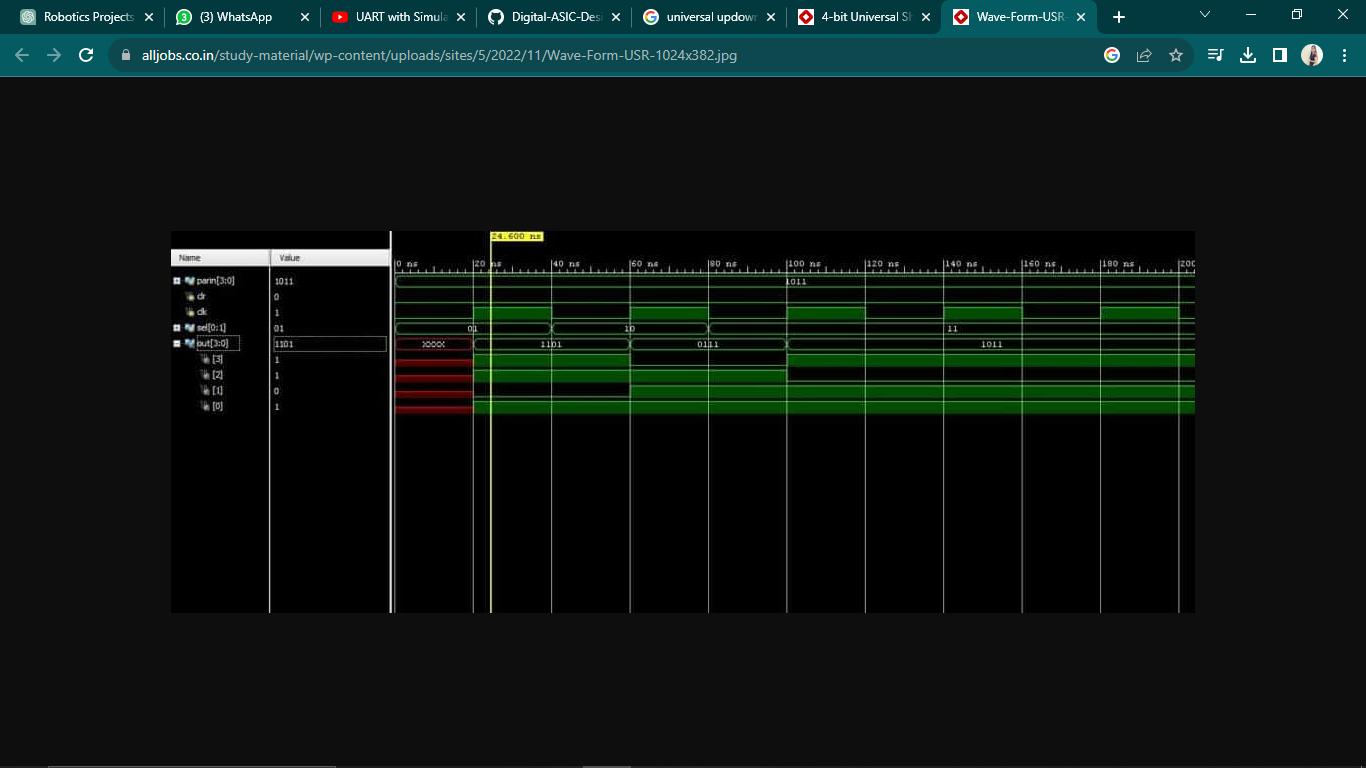
#10;

$finish;

end

endmodule

**Waveforms:**

****

**6.FSM:**

**Mealy FSM:**

module mealy(clk,X,Z,Rst

);

input clk,Rst; input [1:0] X; output reg [1:0]Z;

parameter S00 = 2'b00,

S01 = 2'b01, S10 = 2'b10, S11 = 2'b11;

reg [1:0] cur\_state,nxt\_state;

always @(posedge clk) begin

if (Rst)

end

else

nxt\_state<=S00; cur\_state <= nxt\_state;

always@(cur\_state,X) begin

case (cur\_state)

S00: if(X == 2'b00)

nxt\_state<=S00;

else if (X == 2'b01)

nxt\_state<=S01; else if (X == 2'b10)

nxt\_state<=S10; else if (X == 2'b11)

nxt\_state<=S11;

S01: if(X == 2'b00)

nxt\_state<=S00;

else if (X == 2'b01)

nxt\_state<=S01; else if (X == 2'b10)

nxt\_state<=S10; else if (X == 2'b11)

nxt\_state<=S11;

S10: if(X == 2'b00)

nxt\_state<=S00;

else if (X == 2'b01)

nxt\_state<=S01; else if (X == 2'b10)

nxt\_state<=S10; else if (X == 2'b11)

nxt\_state<=S11;

S11: if(X == 2'b00)

nxt\_state<=S00;

else if (X == 2'b01)

nxt\_state<=S01; else if (X == 2'b10)

nxt\_state<=S10; else if (X == 2'b11)

nxt\_state<=S11;

end

default: nxt\_state<= S00; endcase

always@(posedge clk)

begin

case(cur\_state)

S00: if (X == 2'b00)

Z<=2'b00;

else if (X==2'b01)

Z<=2'b10;

else if (X==2'b10)

Z<=2'b10;

else if (X==2'b11)

Z<=2'b10;

S01: if (X == 2'b00)

Z<=2'b01;

else if (X==2'b01)

Z<=2'b00;

else if (X==2'b10)

Z<=2'b10;

else if (X==2'b11)

Z<=2'b10;

S10: if (X == 2'b00)

Z<=2'b01;

else if (X==2'b01)

Z<=2'b01;

else if (X==2'b10)

Z<=2'b00;

else if (X==2'b11)

Z<=2'b10;

S11: if (X == 2'b00)

Z<=2'b01;

endmodule

end

endcase

else if (X==2'b01)

Z<=2'b01;

else if (X==2'b10)

Z<=2'b01;

else if (X==2'b11)

Z<=2'b00;

**Testbench:**

module mealy\_TB;

// Inputs reg clk;

reg [1:0] X;

reg Rst;

// Outputs wire [1:0] Z;

always #5 clk=~clk;

// Instantiate the Unit Under Test (UUT) mealy uut (

.clk(clk),

.X(X),

.Z(Z),

.Rst(Rst)

);

task initialise; begin

end

clk = 0;

X = 0;

Rst = 0;

endtask

task reset;

begin

end endtask

@(negedge clk); Rst<=1;

@(negedge clk); Rst<=0;

task Data\_in (input [1:0]D); begin

end endtask

@(negedge clk); X<=D;

initial begin

initialise; #5;

reset;

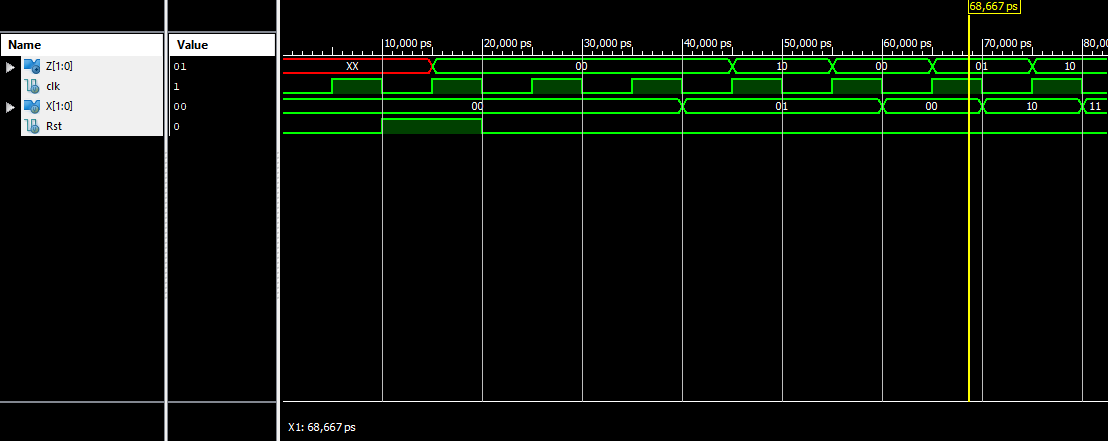
Data\_in (2'b00); Data\_in (2'b01); Data\_in (2'b01); Data\_in (2'b00);

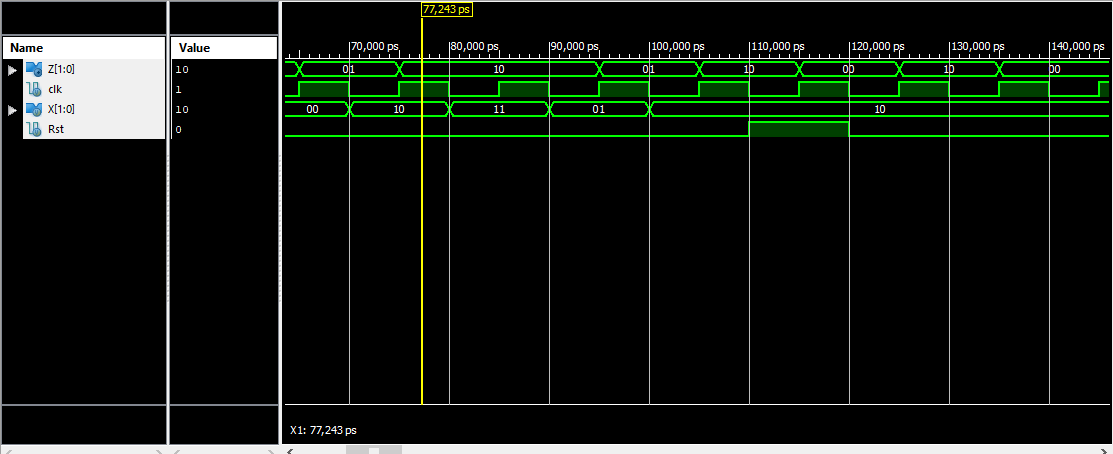
Data\_in (2'b10); Data\_in (2'b11); Data\_in (2'b01); Data\_in (2'b10);

reset;

end endmodule

**Waveforms:**

****

****

**Moore FSM:**

module moore\_fsm(Din,Dout,Rst,clk

);

input Din,Rst,clk; output Dout;

parameter IDLE= 4'b0000; parameter N01 = 4'b0001; parameter N10 = 4'b0010; parameter N11 = 4'b0011; parameter N02 = 4'b0100; parameter N20 = 4'b0101; parameter N12 = 4'b0110; parameter N21 = 4'b0111; parameter N22 = 4'b1000;

reg [3:0] cur\_state,nxt\_state;

assign Dout = (cur\_state == N22)? 1 : 0;

always @(posedge clk) begin

end

if (Rst) else

nxt\_state<=IDLE; cur\_state<=nxt\_state;

always @(cur\_state, Din) begin

case (cur\_state)

IDLE : if (Din == 1)

else N01 : if (Din==1)

else N10 : if (Din==1)

else N11 : if (Din==1)

nxt\_state<=N01; nxt\_state<=N10; nxt\_state<=N02; nxt\_state<=N11; nxt\_state<=N11; nxt\_state<=N20;

else N20 : if (Din==1)

else N02 : if (Din==1)

else

nxt\_state<=N12; nxt\_state<=N21; nxt\_state<=N21; nxt\_state<=N20; nxt\_state<=N02; nxt\_state<=N12;

N21 : if (Din==1)

else N12 : if (Din==1)

else N22 : if (Din==1)

else

nxt\_state<=N22; nxt\_state<= IDLE; nxt\_state<= IDLE; nxt\_state<=N22; nxt\_state<=N22; nxt\_state<=N22;

end endmodule

default nxt\_state<=IDLE; endcase

**Testbench:**

module moore\_fsm\_TB;

// Inputs reg Din; reg Rst; reg clk;

// Outputs wire Dout;

// Instantiate the Unit Under Test (UUT) moore\_fsm uut (

.Din(Din),

.Dout(Dout),

.Rst(Rst),

.clk(clk)

);

always #5 clk = ~clk;

task initialise; begin

end

// Initialize Inputs Din <= 0;

Rst <= 0;

clk <= 0;

endtask

task reset; begin

end

@(negedge clk) Rst<=1;

@(negedge clk) Rst<=0;

endtask

task Data (input D); begin

end

@(negedge clk)

Din <=D;

endtask

initial begin initialise; #5;

reset;

Data (0);

Data (0);

Data (1);

Data (1);

Data (0);

Data (1);

Data (1);

Data (0); reset;

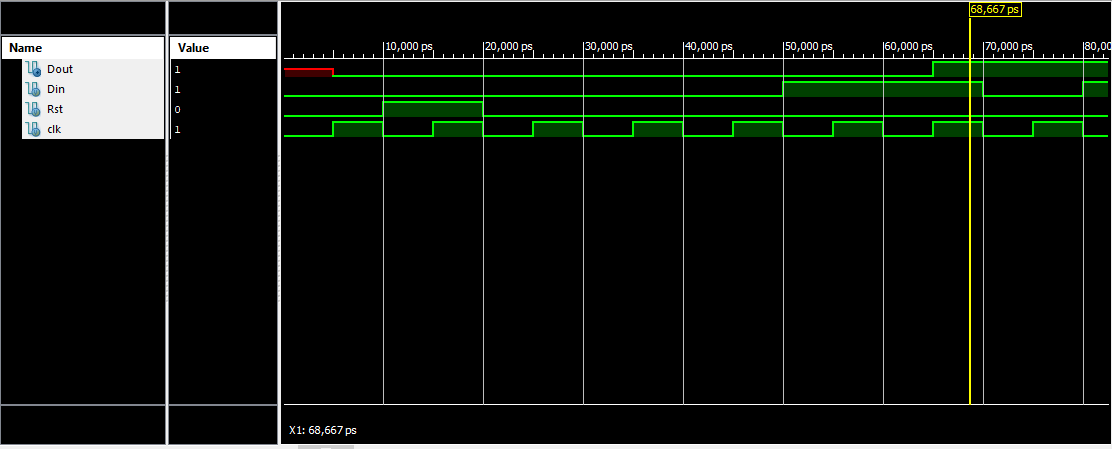
repeat (4)

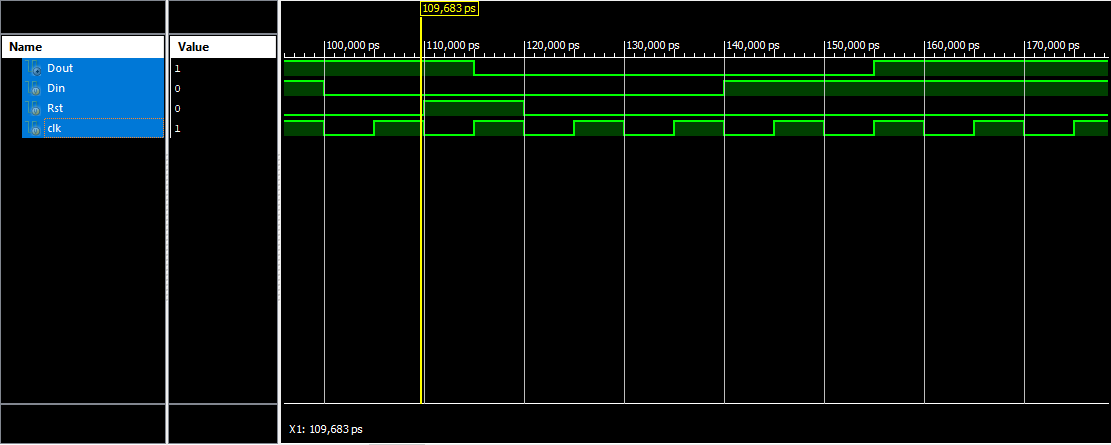
begin end

end

Data ($random%2);

endmodule

**Waveforms:**

****

**Major Project:**

**UART Protocol:**

**UART Transmitter Controller:**

module uart\_tx\_controller(

input clk,

input reset\_n,

input [7:0] i\_Tx\_Byte,

input i\_Tx\_Ready,

output o\_Tx\_Done,

output o\_Tx\_Active, // Asserted for 1 clk cycle after receiving one byte of data

output o\_Tx\_Data

);

localparam UART\_TX\_IDLE = 3'b000,

UART\_TX\_START = 3'b001,

UART\_TX\_DATA = 3'b010,

UART\_TX\_STOP = 3'b011;

reg [2:0] r\_Bit\_Index;

reg r\_Tx\_Done;

reg r\_Tx\_Data;

reg [2:0] r\_State;

reg r\_Tx\_Active;

//UART TX Logic Implementation

always @(posedge clk or negedge reset\_n)

begin

if(~reset\_n)

begin

r\_State <= UART\_TX\_IDLE;

r\_Bit\_Index <= 0;

r\_Tx\_Done <= 1'b0;

r\_Tx\_Data <= 1'b1;

r\_Tx\_Active <= 1'b0;

end

else

begin

case(r\_State)

UART\_TX\_IDLE: begin

r\_Bit\_Index <= 0;

r\_Tx\_Done <= 1'b0;

r\_Tx\_Data <= 1'b1;

if(i\_Tx\_Ready == 1'b1)

begin

r\_State <= UART\_TX\_START;

r\_Tx\_Active <= 1'b1;

end

else begin

r\_State <= UART\_TX\_IDLE;

end

end

UART\_TX\_START: begin

r\_Tx\_Data <= 1'b0;

r\_State <= UART\_TX\_DATA;

end

UART\_TX\_DATA: begin

r\_Tx\_Data <= i\_Tx\_Byte[r\_Bit\_Index];

if(r\_Bit\_Index < 7 )

begin

r\_Bit\_Index <= r\_Bit\_Index + 1;

r\_State <= UART\_TX\_DATA;

end

else begin

r\_Bit\_Index <= 0;

r\_State <= UART\_TX\_STOP;

end

end

UART\_TX\_STOP: begin

r\_State <= UART\_TX\_IDLE;

r\_Tx\_Done <= 1'b1;

r\_Tx\_Active <= 1'b0;

r\_Tx\_Data <= 1'b1;

end

default: begin

r\_State <= UART\_TX\_IDLE;

end

endcase

end

end

assign o\_Tx\_Done = r\_Tx\_Done;

assign o\_Tx\_Data = r\_Tx\_Data;

assign o\_Tx\_Active = r\_Tx\_Active;

endmodule

**UART Receiver Controller:**

module uart\_rx\_controller #(parameter RX\_OVERSAMPLE = 0)(

input clk,

input reset\_n,

input i\_Rx\_Data,

output o\_Rx\_Done, // Asserted for 1 clk cycle after receiving one byte of data

output [7:0] o\_Rx\_Byte

);

localparam UART\_RX\_IDLE = 3'b000,

UART\_RX\_START = 3'b001,

UART\_RX\_DATA = 3'b010,

UART\_RX\_STOP = 3'b011;

reg [7:0] r\_Rx\_Data;

reg [2:0] r\_Bit\_Index;

reg [4:0] r\_Clk\_Count;

reg r\_Rx\_Done;

reg [2:0] r\_State;

//UART RX Logic Implementation

always @(posedge clk or negedge reset\_n)

begin

if(~reset\_n)

begin

r\_State <= UART\_RX\_IDLE;

r\_Bit\_Index <= 0;

r\_Clk\_Count <= 0;

r\_Rx\_Done <= 1'b0;

r\_Rx\_Data <= 8'b00;

end

else

begin

case(r\_State)

UART\_RX\_IDLE: begin

r\_Bit\_Index <= 0;

r\_Clk\_Count <= 0;

r\_Rx\_Done <= 1'b0;

if(i\_Rx\_Data == 1'b0)

begin

r\_State <= UART\_RX\_START;

end

else begin

r\_State <= UART\_RX\_IDLE;

end

end

UART\_RX\_START: begin

if(r\_Clk\_Count == RX\_OVERSAMPLE/2)

begin

if(i\_Rx\_Data == 1'b0)

begin

r\_State <= UART\_RX\_DATA;

r\_Clk\_Count <= 0;

end

else begin

r\_State <= UART\_RX\_IDLE;

end

end

else begin

r\_State <= UART\_RX\_START;

r\_Clk\_Count <= r\_Clk\_Count + 1;

end

end

UART\_RX\_DATA: begin

if(r\_Clk\_Count < (RX\_OVERSAMPLE))

begin

r\_State <= UART\_RX\_DATA;

r\_Clk\_Count <= r\_Clk\_Count + 1;

end

else begin

r\_Rx\_Data[r\_Bit\_Index] <= i\_Rx\_Data;

r\_Clk\_Count <= 0;

if(r\_Bit\_Index < 7 )

begin

r\_Bit\_Index <= r\_Bit\_Index + 1;

r\_State <= UART\_RX\_DATA;

end

else begin

r\_Bit\_Index <= 0;

r\_State <= UART\_RX\_STOP;

end

end

end

UART\_RX\_STOP: begin

if(r\_Clk\_Count < (RX\_OVERSAMPLE))

begin

r\_State <= UART\_RX\_STOP;

r\_Clk\_Count = r\_Clk\_Count + 1;

end

else begin

r\_State <= UART\_RX\_IDLE;

r\_Clk\_Count <= 0;

r\_Rx\_Done <= 1'b1;

end

end

default: begin

r\_State <= UART\_RX\_IDLE;

end

endcase

end

end

assign o\_Rx\_Done = r\_Rx\_Done;

assign o\_Rx\_Byte = r\_Rx\_Done ? r\_Rx\_Data : 8'h00;

endmodule

**UART Contrller:**

`include "defines.v"

`include "baudRateGenerator.sv"

`ifdef UART\_TX\_ONLY

`include "uart\_tx\_controller.v"

`elsif UART\_RX\_ONLY

`include "uart\_rx\_controller.sv"

`else

`include "uart\_tx\_controller.v"

`include "uart\_rx\_controller.sv"

`endif

module uart\_controller #(parameter CLOCK\_RATE = 0,

parameter BAUD\_RATE = 0,

parameter RX\_OVERSAMPLE = 1)(

input clk,

input reset\_n,

`ifdef UART\_TX\_ONLY

input i\_Tx\_Ready,

input [7:0] i\_Tx\_Byte,

output o\_Tx\_Active,

output o\_Tx\_Data,

output o\_Tx\_Done

`elsif UART\_RX\_ONLY

input i\_Rx\_Data,

output o\_Rx\_Done,

output [7:0] o\_Rx\_Byte

`else

input [7:0] i\_Tx\_Byte,

input i\_Tx\_Ready,

output o\_Rx\_Done, // Asserted for 1 clk cycle after receiving one byte of data

output [7:0] o\_Rx\_Byte

`endif

);

wire w\_Rx\_ClkTick,

w\_Tx\_ClkTick;

wire w\_Tx\_Data\_to\_Rx;

`ifdef UART\_TX\_ONLY

assign o\_Tx\_Data = w\_Tx\_Data\_to\_Rx;

`elsif UART\_RX\_ONLY

assign w\_Tx\_Data\_to\_Rx = i\_Rx\_Data;

`endif

//Instantiate Baud Rate Generator

baudRateGenerator #(CLOCK\_RATE, BAUD\_RATE, RX\_OVERSAMPLE) xbaudRateGenerator(

.clk (clk),

.reset\_n (reset\_n),

.o\_Rx\_ClkTick (w\_Rx\_ClkTick),

.o\_Tx\_ClkTick (w\_Tx\_ClkTick)

);

`ifdef UART\_TX\_ONLY

//Instantiation of TX Controller

uart\_tx\_controller xUART\_TX(

.clk (w\_Tx\_ClkTick),

.reset\_n (reset\_n),

.i\_Tx\_Byte (i\_Tx\_Byte),

.i\_Tx\_Ready (i\_Tx\_Ready),

.o\_Tx\_Done (o\_Tx\_Done),

.o\_Tx\_Active (o\_Tx\_Active),

.o\_Tx\_Data (w\_Tx\_Data\_to\_Rx)

);

`elsif UART\_RX\_ONLY

//Instantiation of RX Controller

uart\_rx\_controller #(RX\_OVERSAMPLE) xUART\_RX(

.clk (w\_Rx\_ClkTick),

.reset\_n (reset\_n),

.i\_Rx\_Data (w\_Tx\_Data\_to\_Rx),

.o\_Rx\_Done (o\_Rx\_Done),

.o\_Rx\_Byte (o\_Rx\_Byte)

);

`else

//Instantiation of TX Controller

uart\_tx\_controller xUART\_TX(

.clk (w\_Tx\_ClkTick),

.reset\_n (reset\_n),

.i\_Tx\_Byte (i\_Tx\_Byte),

.i\_Tx\_Ready (i\_Tx\_Ready),

.o\_Tx\_Done (),

.o\_Tx\_Active (),

.o\_Tx\_Data (w\_Tx\_Data\_to\_Rx)

);

//Instantiation of RX Controller

uart\_rx\_controller #(RX\_OVERSAMPLE) xUART\_RX(

.clk (w\_Rx\_ClkTick),

.reset\_n (reset\_n),

.i\_Rx\_Data (w\_Tx\_Data\_to\_Rx),

.o\_Rx\_Done (o\_Rx\_Done),

.o\_Rx\_Byte (o\_Rx\_Byte)

);

`endif

endmodule

**Baud Rate Generator:**

module baudRateGenerator#(parameter CLOCK\_RATE = 25000000,

parameter BAUD\_RATE = 115200,

parameter RX\_OVERSAMPLE = 16)(

// Global Signals

input clk,

input reset\_n,

// RX and TX Baud Rates

output reg o\_Rx\_ClkTick,

output reg o\_Tx\_ClkTick

);

parameter TX\_CNT = CLOCK\_RATE/(2\*BAUD\_RATE);

parameter RX\_CNT = CLOCK\_RATE/(2\*BAUD\_RATE\*RX\_OVERSAMPLE);

parameter TX\_CNT\_WIDTH = $clog2(TX\_CNT);

parameter RX\_CNT\_WIDTH = $clog2(RX\_CNT);

reg[TX\_CNT\_WIDTH - 1:0] r\_Tx\_Counter;

reg[RX\_CNT\_WIDTH - 1:0] r\_Rx\_Counter;

//RX Baud Rate

always @(posedge clk or negedge reset\_n)

begin

if(~reset\_n)

begin

o\_Rx\_ClkTick <= 1'b0;

r\_Rx\_Counter <= 0;

end

else if(r\_Rx\_Counter == RX\_CNT - 1) begin

o\_Rx\_ClkTick <= ~o\_Rx\_ClkTick;

r\_Rx\_Counter <= 0;

end

else begin

r\_Rx\_Counter <= r\_Rx\_Counter + 1;

end

end

//TX Baud Rate

always @(posedge clk or negedge reset\_n)

begin

if(~reset\_n)

begin

o\_Tx\_ClkTick <= 1'b0;

r\_Tx\_Counter <= 0;

end

else if(r\_Tx\_Counter == TX\_CNT - 1) begin

o\_Tx\_ClkTick <= ~o\_Tx\_ClkTick;

r\_Tx\_Counter <= 0;

end

else begin

r\_Tx\_Counter <= r\_Tx\_Counter + 1;

end

end

endmodule

**UART Testbench:**

`include "defines.v"

module UART\_TB ();

// Testbench uses a 25 MHz clock, Want to interface to 115200 baud UART , 25000000 / 115200 = 217 Clocks Per Bit.

parameter c\_CLOCK\_PERIOD\_NS = 40;

parameter c\_CLKS\_PER\_BIT = 217;

parameter c\_BIT\_PERIOD = 8680;

//Initialize Variables

reg Clock = 0;

reg reset\_n = 0;

`ifdef UART\_TX\_ONLY

wire Tx\_Done;

reg Tx\_Ready = 0;

wire Tx\_Active;

wire Tx\_Data;

reg [7:0] Tx\_Byte = 0;

`elsif UART\_RX\_ONLY

wire [7:0] Rx\_Byte;

reg UART\_Rx = 0;

wire Rx\_Done;

`else

wire Rx\_Done;

wire [7:0] Rx\_Byte;

reg [7:0] Tx\_Byte = 0;

reg Tx\_Ready = 0;

reg [7:0] DataToSend[0:7] = {8'h01, 8'h10, 8'h22, 8'h32, 8'h55, 8'hAA, 8'hAB, 8'h88};

reg [7:0] DataToSend\_1[0:7] = {8'h21, 8'h11, 8'h32, 8'h77, 8'hA0, 8'h0B, 8'hBB, 8'hFF};

reg [7:0] DataReceived[0:7];

integer ii;

`endif

`ifdef UART\_TX\_ONLY

uart\_controller #(.CLOCK\_RATE(25000000), .BAUD\_RATE(115200)) xUART\_TX(

.clk (Clock),

.reset\_n (reset\_n),

.i\_Tx\_Ready (Tx\_Ready),

.i\_Tx\_Byte (Tx\_Byte),

.o\_Tx\_Active (Tx\_Active),

.o\_Tx\_Data (Tx\_Data),

.o\_Tx\_Done (Tx\_Done)

);

`elsif UART\_RX\_ONLY

uart\_controller #(.CLOCK\_RATE(25000000), .BAUD\_RATE(115200), .RX\_OVERSAMPLE(16)) xUART\_RX(

.clk (Clock),

.reset\_n (reset\_n),

.i\_Rx\_Data (UART\_Rx),

.o\_Rx\_Done (Rx\_Done),

.o\_Rx\_Byte (Rx\_Byte)

);

`else

uart\_controller #(.CLOCK\_RATE(25000000), .BAUD\_RATE(115200), .RX\_OVERSAMPLE(16)) xUART(

.clk (Clock),

.reset\_n (reset\_n),

.i\_Tx\_Byte (Tx\_Byte),

.i\_Tx\_Ready (Tx\_Ready),

.o\_Rx\_Done (Rx\_Done),

.o\_Rx\_Byte (Rx\_Byte)

);

`endif

always

#(c\_CLOCK\_PERIOD\_NS/2) Clock <= !Clock;

`ifdef UART\_TX\_ONLY // UART TX Controller Test

reg[7:0] dataToSend\_TX = 8'b01010101;

initial begin

#5 reset\_n = 1;

@(posedge Clock);

@(posedge Clock);

Tx\_Ready = 1'b1;

@(posedge Clock); Tx\_Byte = DataToSend\_TX;

#100000;

$finish();

end

`elsif UART\_RX\_ONLY // UART RX Controller Test

reg[7:0] DataToSend\_RX = 8'b01010101;

integer i;

initial begin

#5 reset\_n = 1;

@(posedge Clock); UART\_Rx = 0;

for(i=0; i < 8 ; i = i+1)

begin

#(217\*40);

@(posedge Clock); UART\_Rx = DataToSend\_RX[i];

end

#85000;

$finish();

end

`else // UART TX + RX Controller Test

initial

begin

#5 reset\_n = 1;

@(posedge Clock);

@(posedge Clock);

Tx\_Ready <= 1'b1;

for (ii = 0; ii < 8; ii = ii + 1)

begin

Tx\_Byte = DataToSend[ii];

@(posedge Rx\_Done);

DataReceived[ii] = Rx\_Byte;

if (DataToSend[ii] == DataReceived[ii])

$display("Test Passed - Correct Byte Received. TX Data Byte = %h, RX Data Byte = %h", DataToSend[ii] , DataReceived[ii]);

else

$display("Test Failed - Incorrect Byte Received. TX Data Byte = %h, RX Data Byte = %h", DataToSend[ii] , DataReceived[ii]);

end

#800000;

$finish();

end

`endif

initial

begin

// Required to dump signals to EPWave

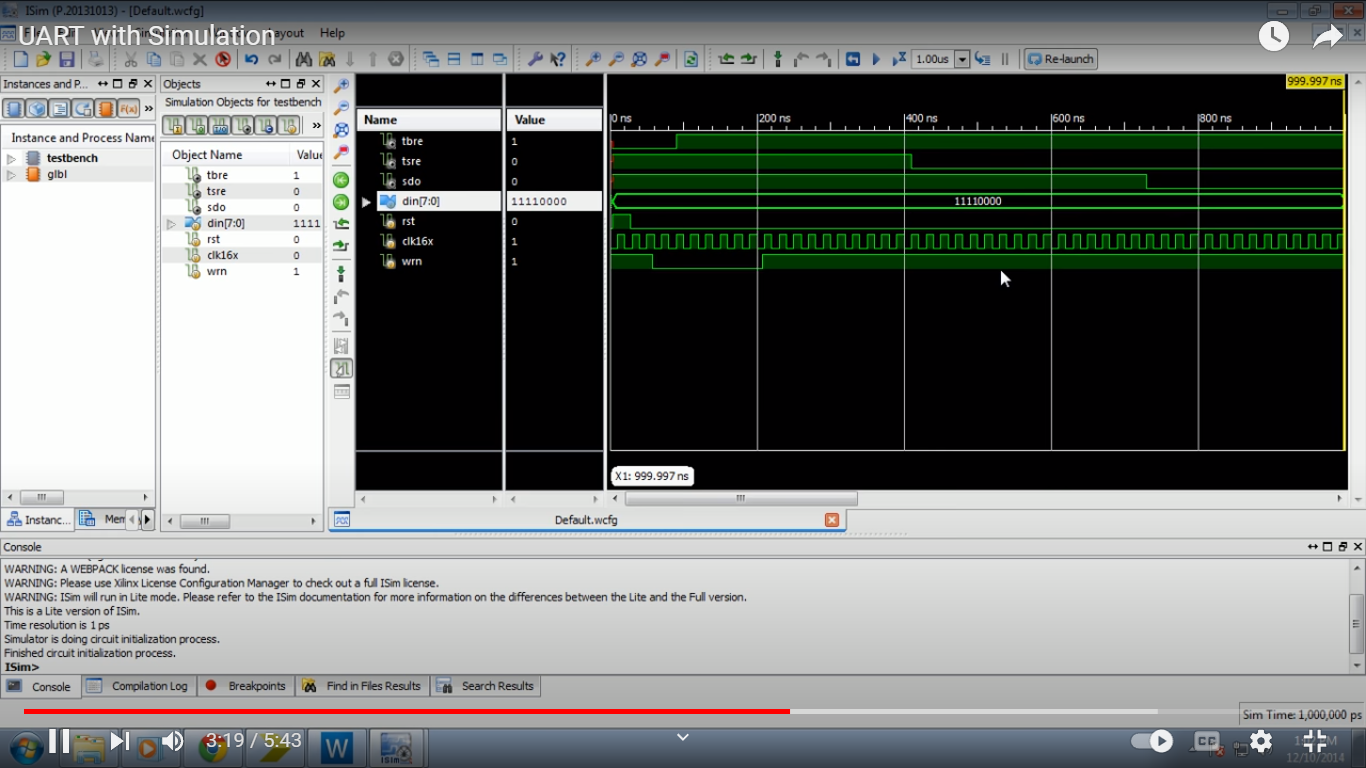
$dumpfile("dump.vcd");

$dumpvars(0);

end

endmodule

**Wave forms:**

****

1. **Student feedback**

I am very thankful to SURE Trust, because this is the place where I learnt both improving skills and discipline are must to achieve dreams. I am thankful to Prof. Ch. Radhakumari mam for introducing such wonderful courses to the students. I am thankful to my trainer Mr. Ryan Embenezer for training us VLSI. I am promising I will continue the same enthusiasm and work. Thank you.

## Uniqueness of this course

* + People can conceptually learn things.
  + Clearing doubts are fabulous.
  + Industry relevant skills.
  + Good trainer with good relevant skills.

## Concluding Remarks

SURE Trust provides good industry relevant skills to the students and makes them ready for their dream jobs. Both discipline and learning are important, I realized that thing clearly here.

This is the place where students can learn the thing with joy and enthusiasm. Because trainers are friendly so students can find free environment to do anything. Finally, because of SURE Trust I got some industry relevant skills. Thank you SURE Trust.