If Ryan Writz

Pritam Shah

ECE 4/581

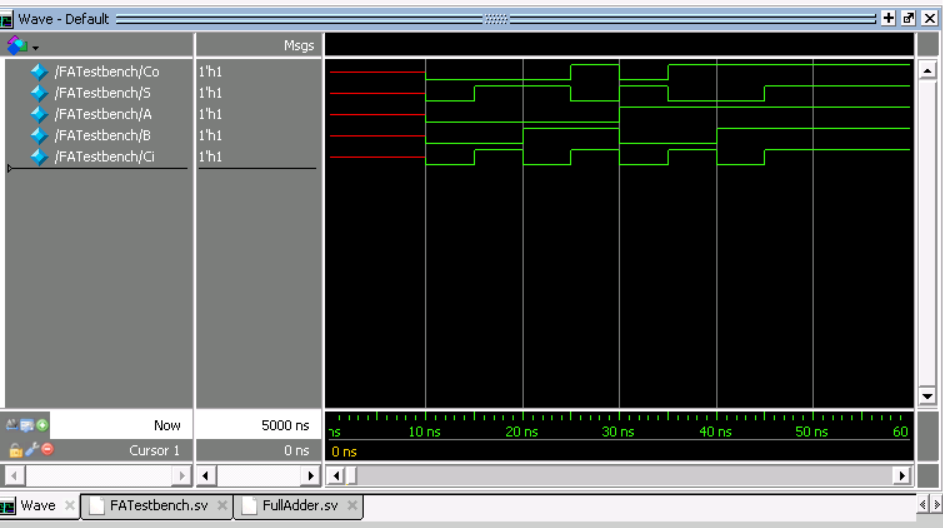
Project 1

7/15/19

1:

Full Adder Truth Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | Cin | Cout | S |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



FullAdder.sv:

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 1

//SystemVerilog description of a full adder as a netlist of AND and OR gates

//and inverters w/o gate delays

//Side note: Standard Full Adder would be

/\*module FA(output logic Co, S, input logic A, B, Ci);

logic X,Y,Z;

xor E1(X,A,B); // X = A ^ B

xor E2(S,Ci,X); // S = Ci ^ X

and A1(Y,Ci,X); // Y = Ci & X

and A2(Z,A,B); // Z = A & B

or O1(Co,Y,Z); // Co = Y | Z

endmodule \*/

//However as we are limited to AND, OR and inverters must make do

module FA(output logic Co, S, input logic A, B, Ci);

logic Q,R,T,U,V,W,X,Y,Z;

//Effectively becomes X = A ^ B

and A1(Q,A,B); // Q = A & B

not I1(R,Q); // R = ~Q

or O1(T,A,B); // T = A | B

and A2(X,R,T); // X = R & T

//Effectively becomes S = Ci ^ X

and A3(U,X,Ci); // U = X & Ci

not I2(W,U); // W = ~U

or O2(V,X,Ci); // V = X | Ci

and A4(S,W,V); // S = W & V

and A5(Y,Ci,X); // Y = Ci & X

and A6(Z,A,B); // Z = A & B

or O3(Co,Y,Z); // Co = Y | Z

endmodule

FATestbench.sv:

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 1

//SystemVerilog description of a full adder as a netlist of AND and OR gates

//and inverters w/o gate delays

module FATestbench;

logic Co, S;

logic A, B, Ci;

FA FullAdd(Co, S, A, B, Ci);

initial

begin

#5ns;

#5ns A = 0; B = 0; Ci = 0; //{A,B,Ci} = 000

#5ns Ci = 1; //{A,B,Ci} = 001

#5ns B = 1; Ci = 0; //{A,B,Ci} = 010

#5ns Ci = 1; //{A,B,Ci} = 011

#5ns A = 1; B = 0; Ci = 0; //{A,B,Ci} = 100

#5ns Ci = 1; //{A,B,Ci} = 101

#5ns B = 1; Ci = 0; //{A,B,Ci} = 110

#5ns Ci = 1; //{A,B,Ci} = 111

#5ns;

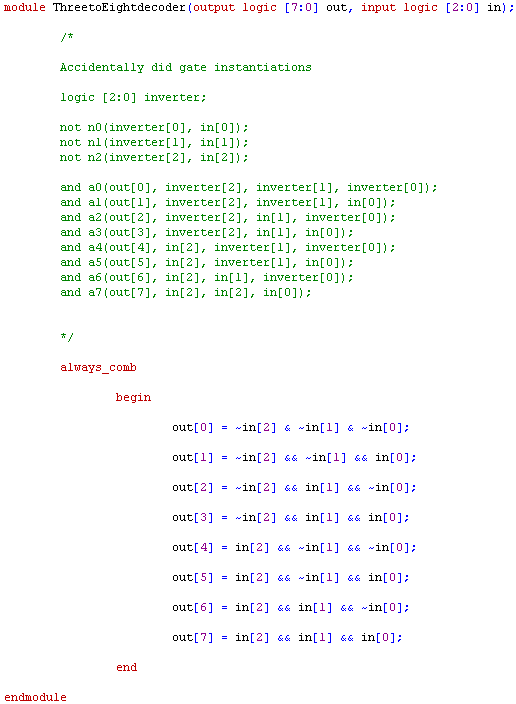
end

endmodule

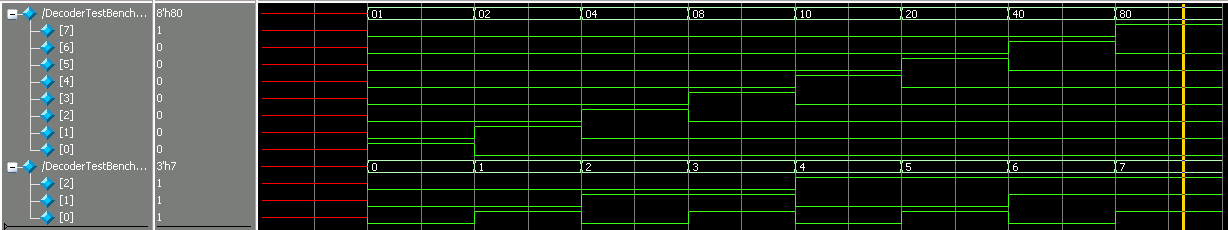
2:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I2 | I1 | I0 | O8 | O7 | O6 | O5 | O4 | O3 | O2 | O1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

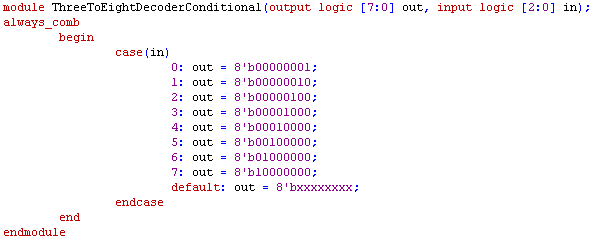
Code for Boolean Operator Implementation of the 3 to 8 Decoder



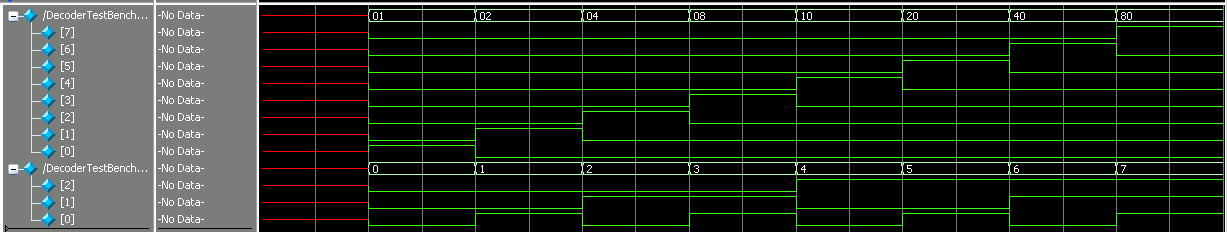
Simulation of the Boolean Operator Implementation of the 3 To 8 Decoder:



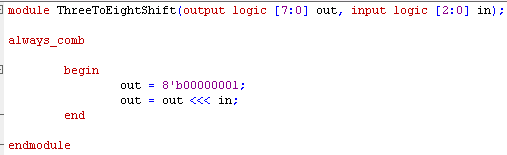
Code of the Conditional Operator Implementation of the 3 To 8 Decoder:



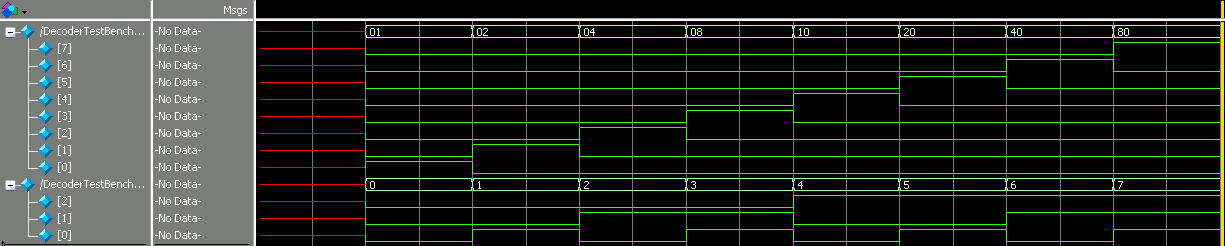
Simulation of the Conditional Operator Implementation of the 3 To 8 Decoder:



Code for the Shift Operator Implementation of the 3 to 8 Decoder:

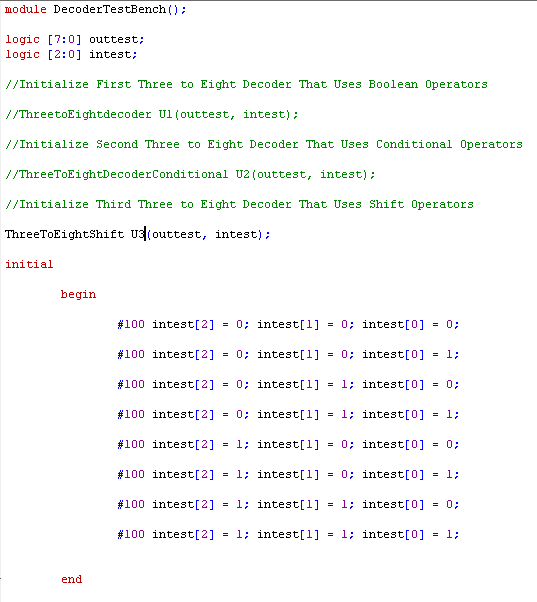


Simulation of the Shift Operator Implementation of the 3 to 8 Decoder.



The three different implementations match with each other and also match the expected outcome.

Test Bench For 3 To 8 Decoder

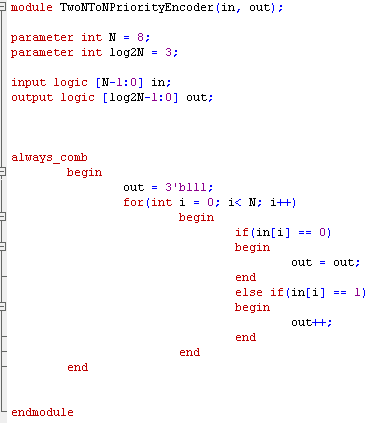


3:

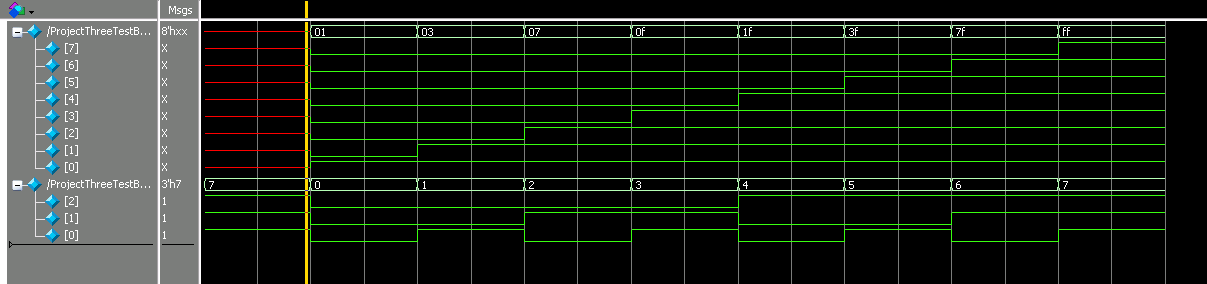
Truth Table

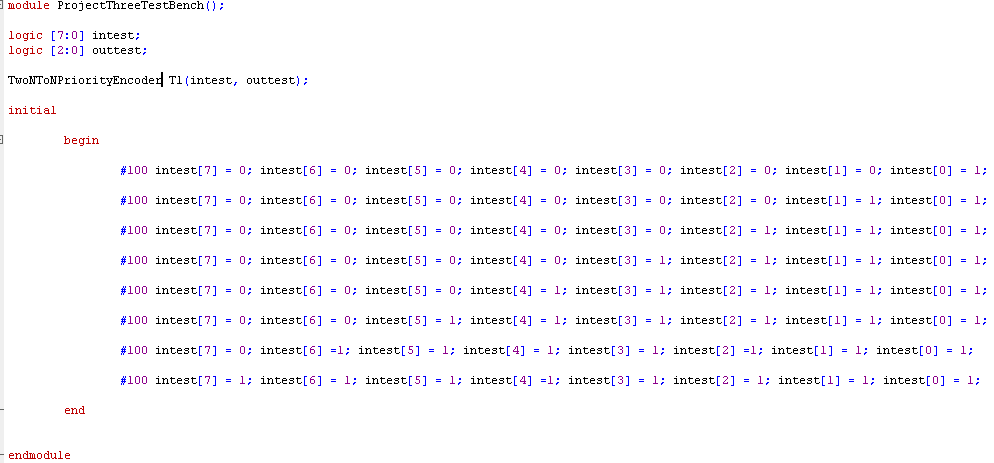
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I7 | I6 | I5 | I4 | I3 | I2 | I1 | I0 | O2 | O1 | O0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | x | x | 0 | 1 | 0 |
| 0 | 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 |

We used an 8 bit input and a 3 bit output to test.



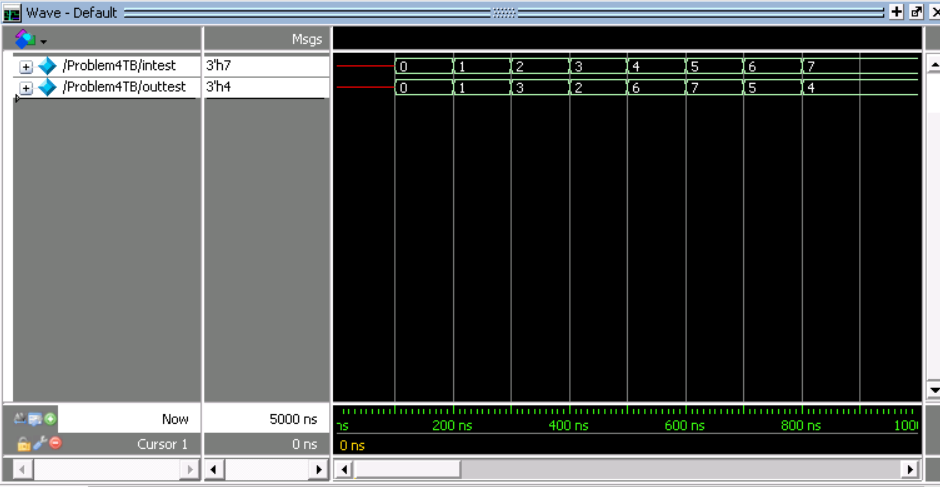
Simulation Picture of Priority Encoder With 8 bit Input and 3 bit Output.



Test Bench Code:

4:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| B2 | B1 | B0 | I2 | I1 | I0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |



Problem4.sv

module BinaryToGreyCode(in, out);

parameter int Bits = 3;

input logic [Bits-1:0] in;

output logic [Bits-1:0] out;

int i = 0;

always\_comb

begin

for(i = 0; i < Bits; i++)

begin

if(i == Bits- 1)

begin

out[i] = in[Bits-1];

end

else

begin

out[i] = in[i] ^ in[i+1];

end

end

end

endmodule

Problem4TB.sv

module Problem4TB();

logic [2:0] intest;

logic [2:0] outtest;

BinaryToGreyCode F1(intest, outtest);

initial

begin

#100 intest[2] = 0; intest[1] = 0; intest[0] = 0;

#100 intest[2] = 0; intest[1] = 0; intest[0] = 1;

#100 intest[2] = 0; intest[1] = 1; intest[0] = 0;

#100 intest[2] = 0; intest[1] = 1; intest[0] = 1;

#100 intest[2] = 1; intest[1] = 0; intest[0] = 0;

#100 intest[2] = 1; intest[1] = 0; intest[0] = 1;

#100 intest[2] = 1; intest[1] = 1; intest[0] = 0;

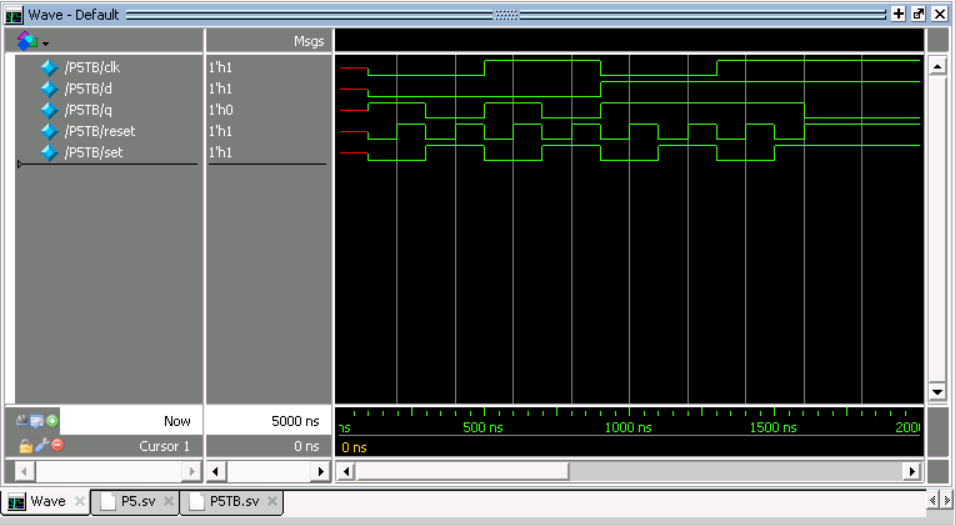
#100 intest[2] = 1; intest[1] = 1; intest[0] = 1;

end

endmodule

5:

This was a simulation of the D flip flop which takes what is on the input at the negative edge of the clock and puts it on the output. The synchronous reset will send the output to zero while the asynchronous set will send the output to one. This is what we see.



P5.sv

module d\_ff(input logic d, clk, set, reset, output logic q);

logic synch\_reset;

always\_comb

begin

synch\_reset = reset && clk;

end

always\_ff @(negedge clk, set, posedge synch\_reset)

begin

if(set == 0)

q <= 1;

else if (synch\_reset == 1)

q <= 0;

else

q <= d;

end

endmodule

P5TB.sv

module P5TB();

logic d, q, set, reset, clk;

d\_ff I1(d, clk, set, reset, q);

initial

begin

#100 d = 0; clk = 0; set = 0; reset = 0;

#100 d = 0; clk = 0; set = 0; reset = 1;

#100 d = 0; clk = 0; set = 1; reset = 0;

#100 d = 0; clk = 0; set = 1; reset = 1;

#100 d = 0; clk = 1; set = 0; reset = 0;

#100 d = 0; clk = 1; set = 0; reset = 1;

#100 d = 0; clk = 1; set = 1; reset = 0;

#100 d = 0; clk = 1; set = 1; reset = 1;

#100 d = 1; clk = 0; set = 0; reset = 0;

#100 d = 1; clk = 0; set = 0; reset = 1;

#100 d = 1; clk = 0; set = 1; reset = 0;

#100 d = 1; clk = 0; set = 1; reset = 1;

#100 d = 1; clk = 1; set = 0; reset = 0;

#100 d = 1; clk = 1; set = 0; reset = 1;

#100 d = 1; clk = 1; set = 1; reset = 0;

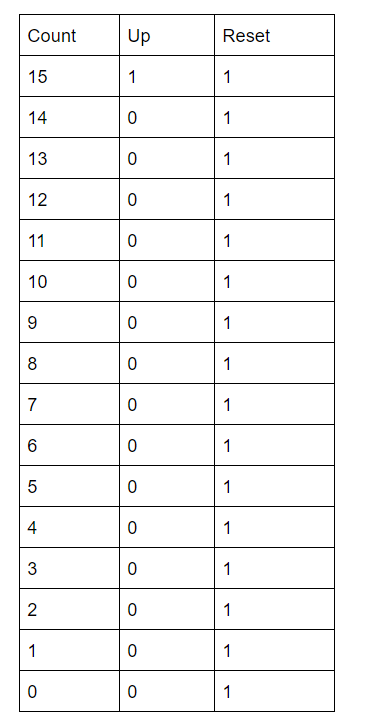
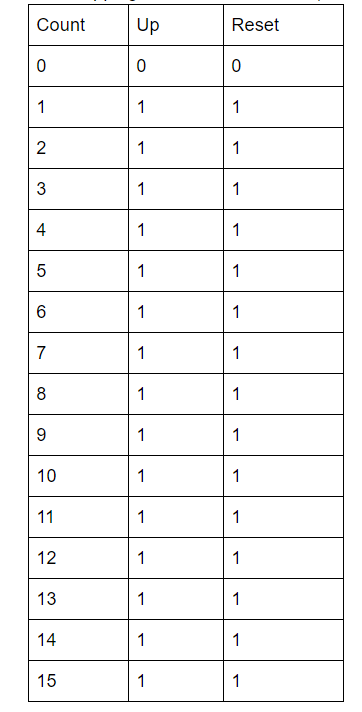
#100 d = 1; clk = 1; set = 1; reset = 1;

end

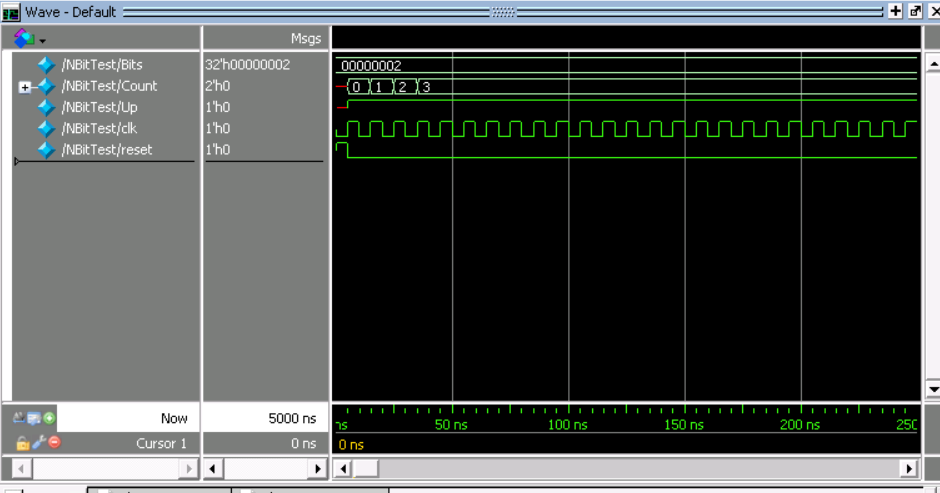
endmodule

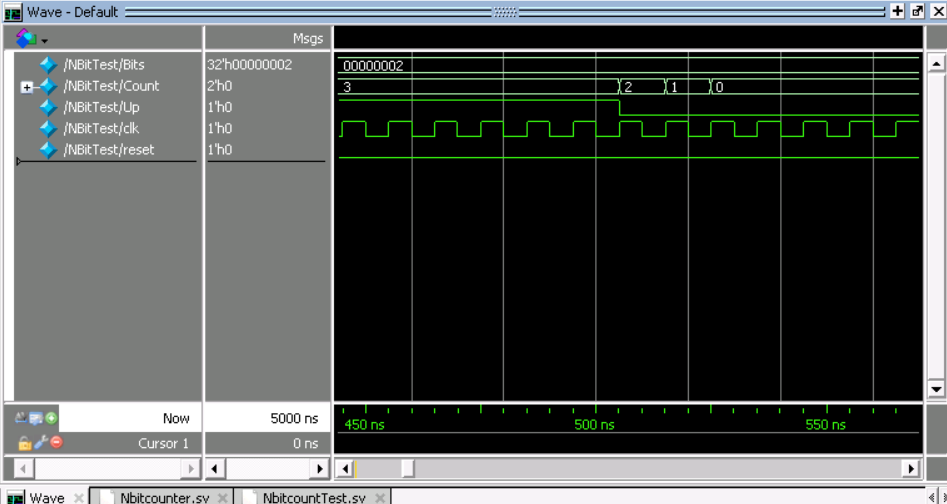
6:

N Bit Non-Wrapping Counter Expected behavior ( Bits = 4)

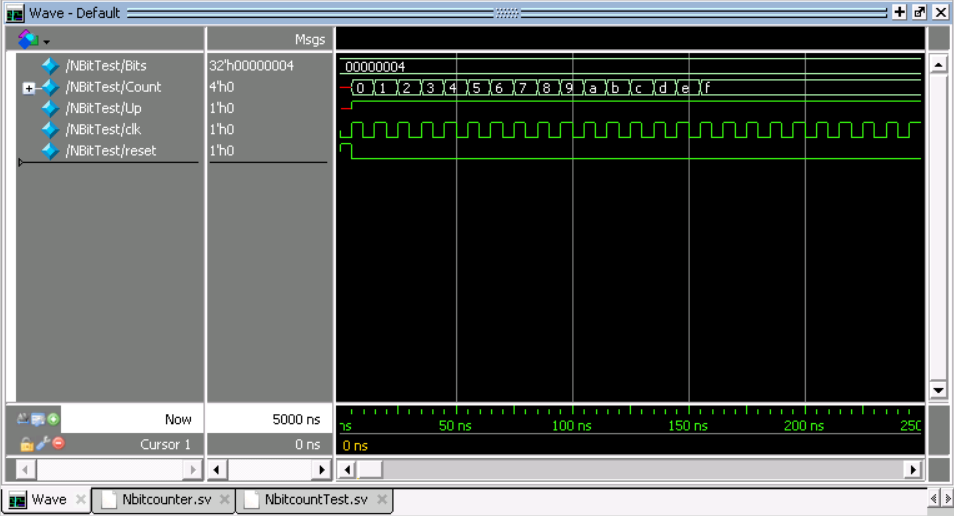


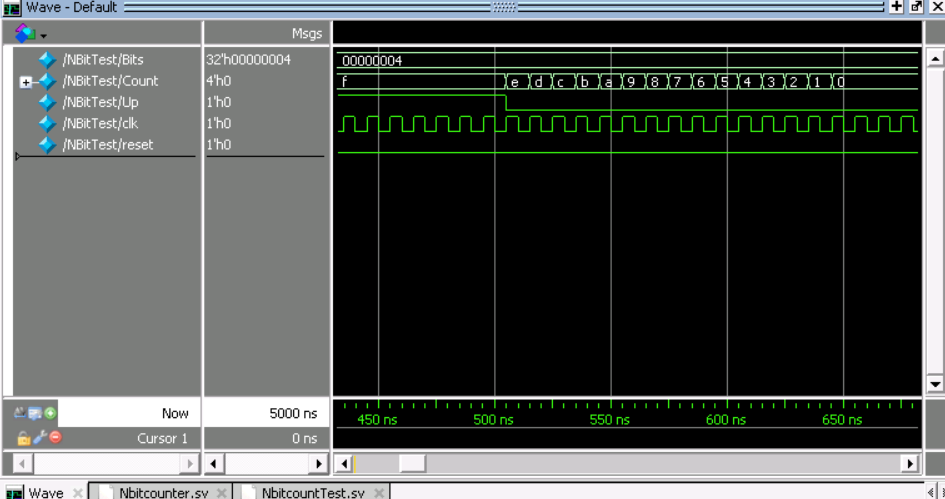
Bits = 2





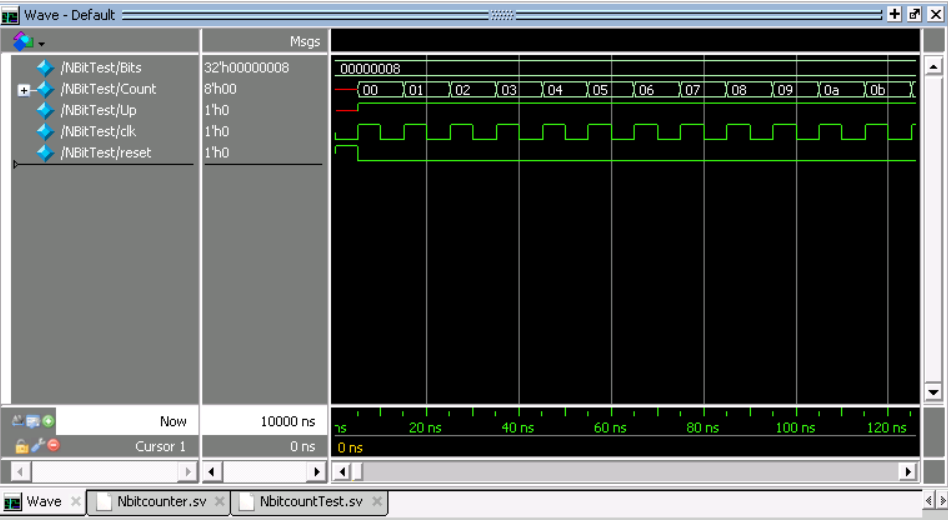
Bits = 4

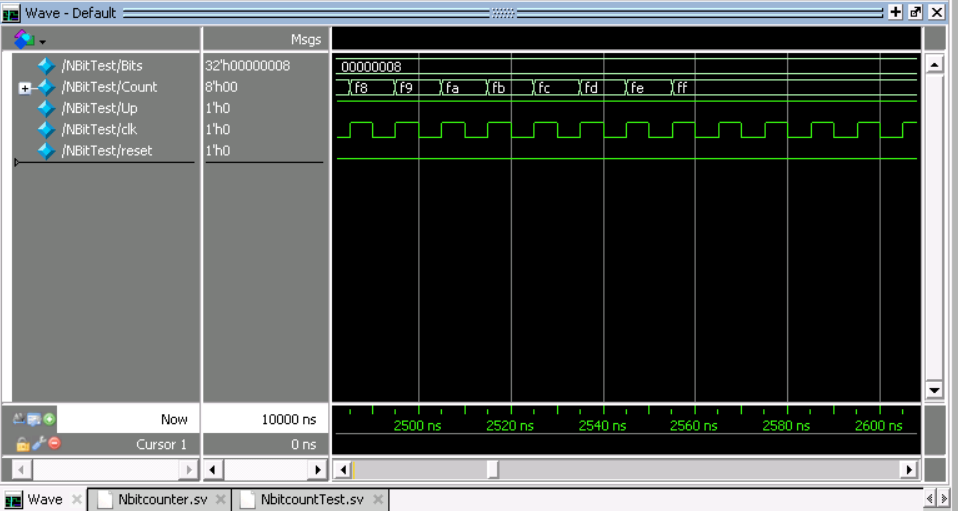


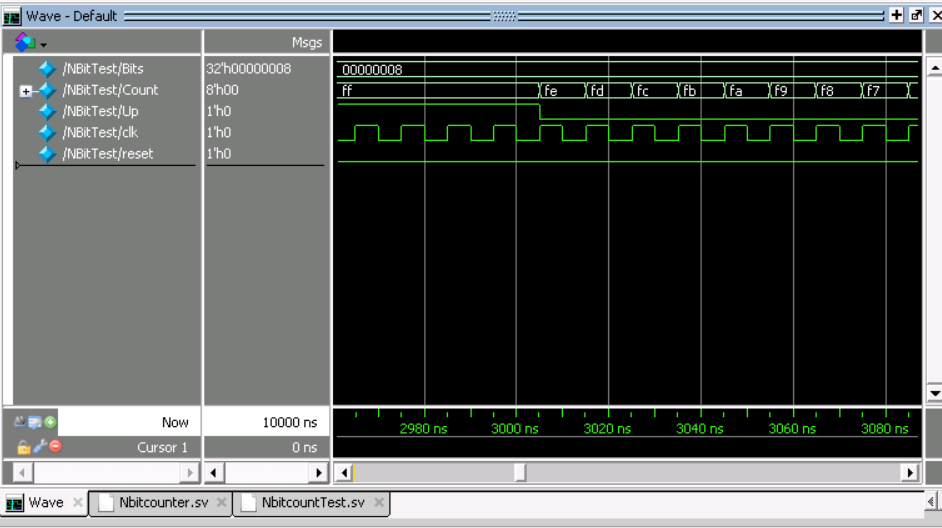


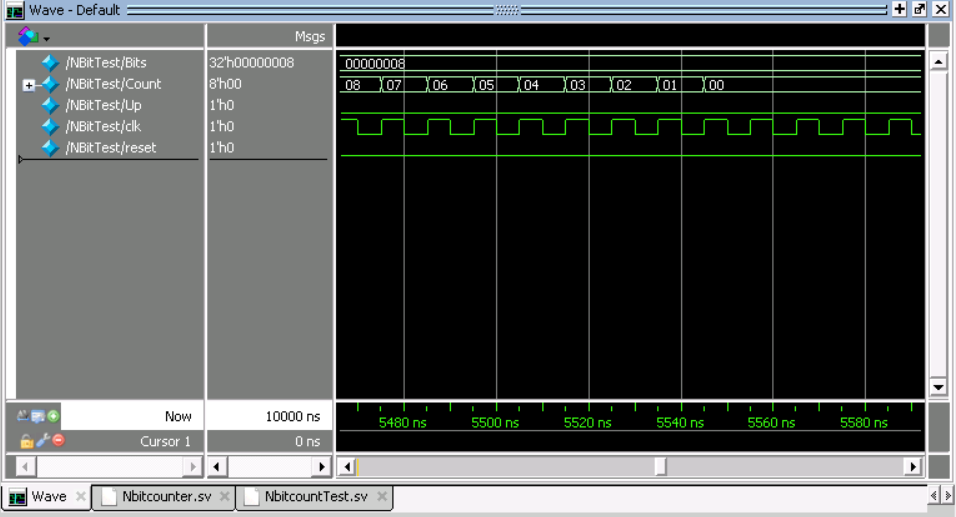
Bits = 8

Note in order to get all values to test for 8 bits we increased the delay between Up = 1 to Up = 0 to 3000 rather than the 500 used on the other tests









Note in order to change the value of the bits being used on the testbench, change the line in the testbench on the line:

parameter Bits = 8;

In order to change the value of the bits the actual module is being tested on change the number in the testbench on line:

Nbitcount #(8) TestNbit(Count, Up, clk, reset);

Nbitcounter.sv

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 6

//SystemVerilog model of an N-bit counter with a

//control input ‘Up’. When the control input is ‘1’ the

//counter counts up; when it is ‘0’ the counter counts down.

//The counter should not wrap round. When the all ‘1’ s or

//all ‘0’ s states are reached the counter should stop

module Nbitcount

#(parameter Bits = 4)

(output logic [Bits-1:0]Count, input logic Up,clk,reset);

always\_ff @(posedge clk, negedge reset)

begin

if(~reset)

Count <= 0; //If reset asserted set count to 0

if(Up & Count < (2\*\*Bits -1)) //If less than highest count and Up

Count <= Count + 1; //Increase count

if (Up & Count == (2\*\*Bits -1)) //If equal to highest count and Up

Count <= Count; //Do not increase count to avoid wrapping

if (~Up & Count == 0) //If 0 and Up is low do no decrease

Count <= Count; //to avoid wrapping on other end

else if(~Up & Count != 0) //Otherwise decrement

Count <= Count - 1;

end

endmodule

NbitcountTest.sv

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 6

//SystemVerilog model of an N-bit counter with a

//control input ‘Up’. When the control input is ‘1’ the

//counter counts up; when it is ‘0’ the counter counts down.

//The counter should not wrap round. When the all ‘1’ s or

//all ‘0’ s states are reached the counter should stop

module NBitTest;

parameter Bits = 4;

logic [Bits - 1:0] Count;

logic Up, clk, reset;

Nbitcount #(4) TestNbit(Count, Up, clk, reset);

initial

begin

clk=0;

forever #5 clk=~clk;

end

initial

begin

reset = 1; //Initial reset to clear count

#5 reset = 0; Up = 1;

#500 Up = 0; //Multiple clock pulse to test increase and switch to decrease

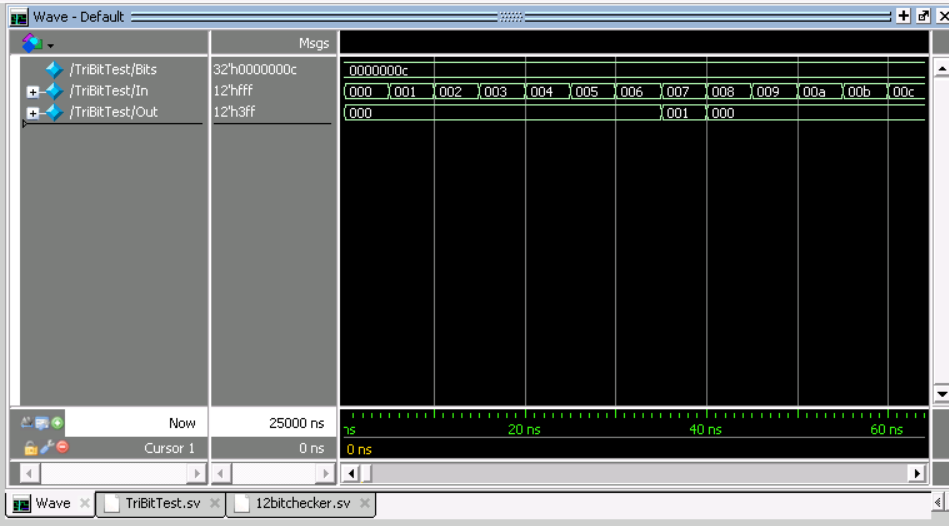
end

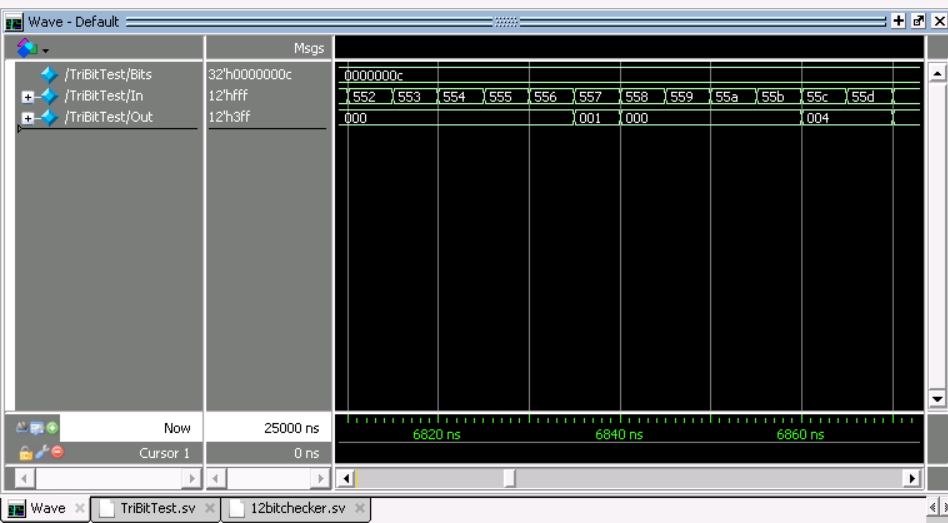
endmodule

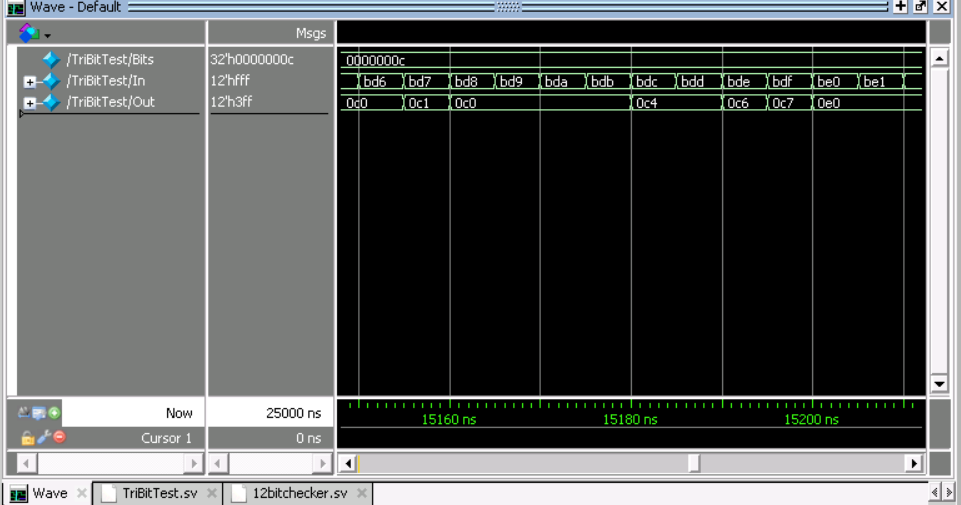
7:

12 bit Checker for 3 consecutive 1’s Expected behavior

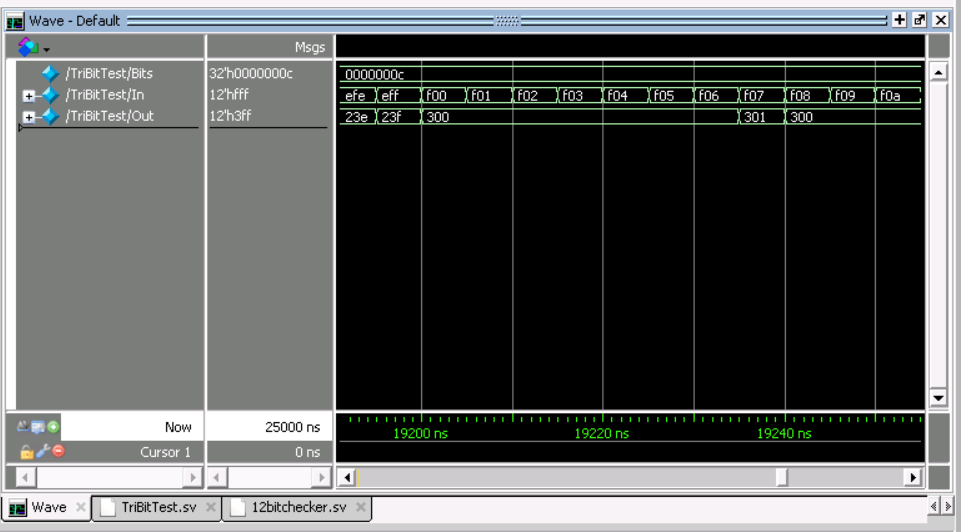
|  |  |  |  |
| --- | --- | --- | --- |
| In(Binary) | In(Hex) | Out(Binary) | Out(Hex) |
| 000000000000  To  000000000110 | 000  To  006 | 000000000000 | 000 |
| 010101010101 | 555 | 000000000000 | 000 |
| 101111011000 | BD8 | 000011000000 | 0C0 |
| 111000000111 | E07 | 001000000001 | 201 |
| 111011111111 | EFF | 001000111111 | 23F |
| 111111111110 | FFE | 001111111110 | 3FE |
| 111111111111 | FFF | 001111111111 | 3FF |

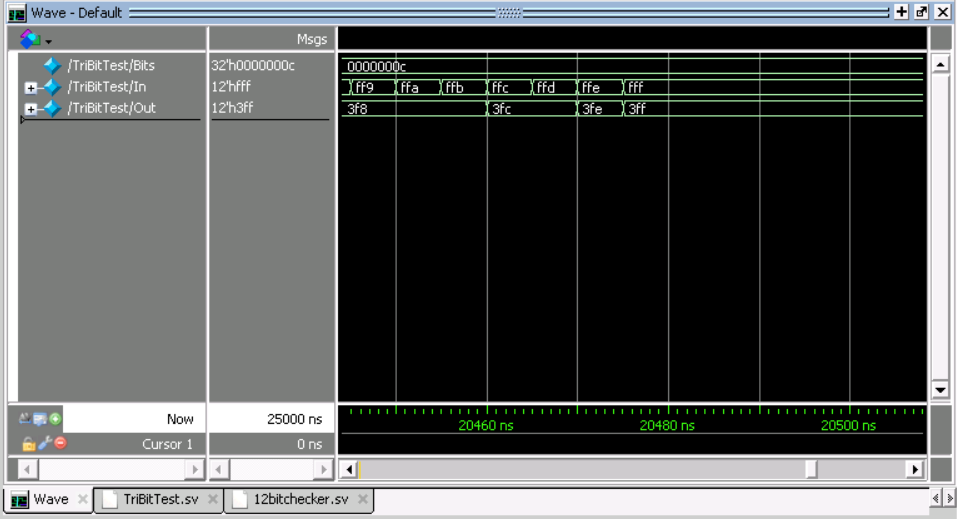












12bitchecker.sv

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 7

//SystemVerilog model of a state machine that detects a sequence

//of three logic 1 s occurring at the input and that asserts a logic 1

//at the end of each sequence when outputting

//E.g. the sequence 001011101111 would produce an output 000000100011.

/\* Model after this where {A,B,C} first 3 digits of input

if({A,B,C} === 3'b111) //If 3 1's in a row set last digit in three

{Ap,Bp,Cp} = 3'b001; //to 1

else

{Ap,Bp,Cp} = 3'b000;

\*/

module TriBitCheck

#(parameter Bits = 12) //Number of bits in bus

(output logic [Bits - 1: 0] Out, input logic [Bits - 1: 0] In);

int i;

always\_comb

begin

for(i = 0; i < (Bits -2) ; i++) //Increment till 1st to 10th bit are checked as leading bit in sequence

if({In[i+2],In[i+1],In[i]} === 3'b111) //If 3 1's in a row set last digit in three

{Out[i+2],Out[i+1],Out[i]} = 3'b001; //to 1

else

{Out[i+2],Out[i+1],Out[i]} = 3'b000; //else set to 0s

end

endmodule

TriBitTest.sv

//Ryan Writz and Priyam Shah (c) July 2019

//ECE 4/581 Project 1 Question 7

//SystemVerilog model of a state machine that detects a sequence

//of three logic 1 s occurring at the input and that asserts a logic 1

//at the end of each sequence when outputting

module TriBitTest;

parameter Bits = 12;

logic [Bits - 1:0] Out;

logic [Bits -1 :0] In ;

TriBitCheck #(12) TriBit(Out,In);

initial

begin

In = 12'b 0000\_0000\_0000;

while( In < 12'b 1111\_1111\_1111)

begin

#5 In = In + 1;

end

In = 12'b 1111\_1111\_1111;

end

endmodule