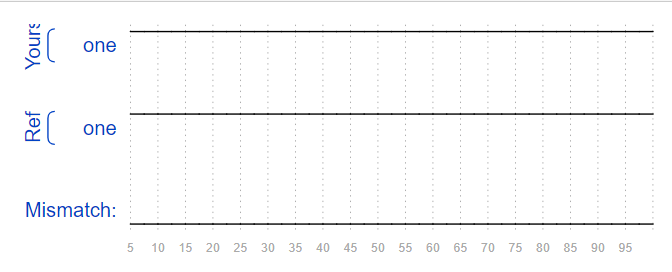
**Verilog assignment -1**

**Answers**

1. Build a circuit with no inputs and one output. That output should always drive 1 (or logic high).

Ans;

module top\_module(

output reg one

);

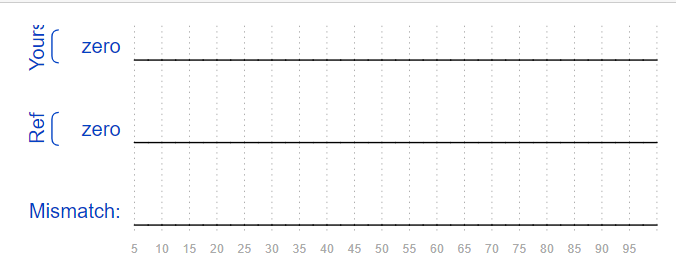
always @\* begin

one = 1;

end

endmodule

2. Build a circuit with no inputs and one output that outputs a constant 0

Ans;

module top\_module(

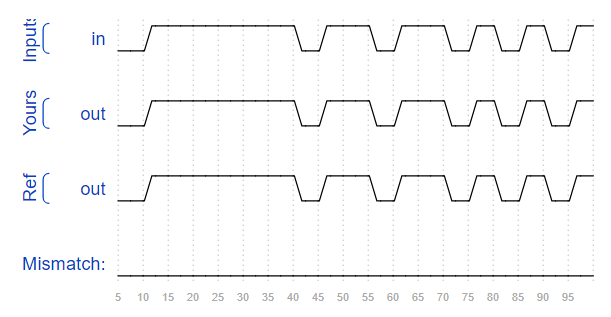
output zero

);

assign zero=0;

endmodule

3. Create a module with one input and one output that behaves like a wire.

module top\_module(

input in,

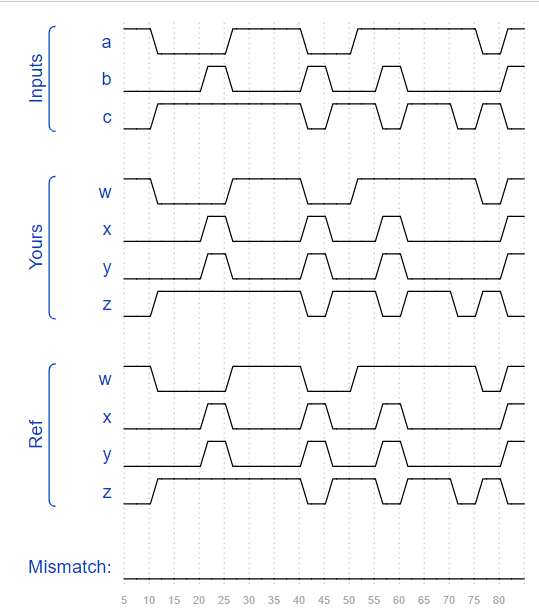
output out

);

assign out = in;

endmodule

4. Create a module with 3 inputs and 4 outputs that behaves like wires that makes these connections:

a -> w

b -> x

b -> y

c -> z

ans;

module top\_module(

input a, b, c,

output w, x, y, z

);

assign w = a;

assign x = b;

assign y = b;

assign z = c;

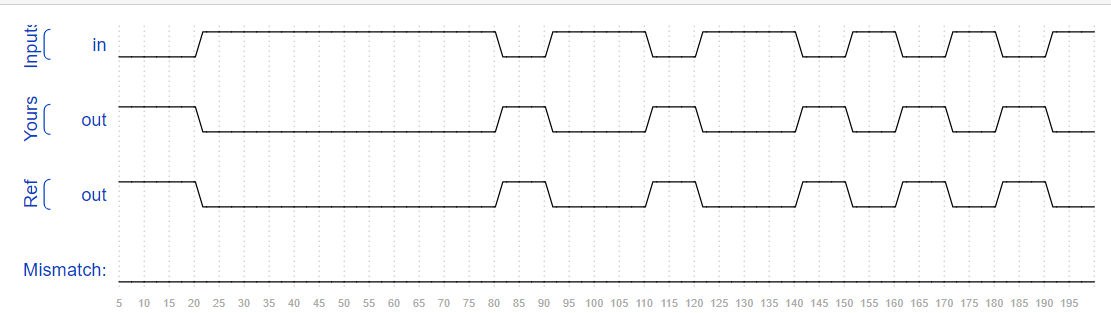
endmodule

5. Create a module that implements a NOT gate.

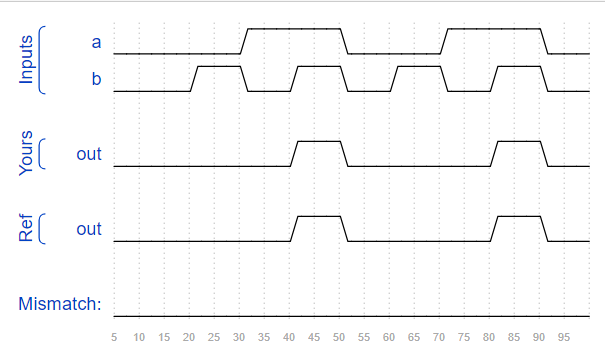
Ans;

module top\_module( input in, output out );

assign out=~in;

endmodule

6. Create a module that implements an AND gate.

Ans;

module top\_module(

input a,

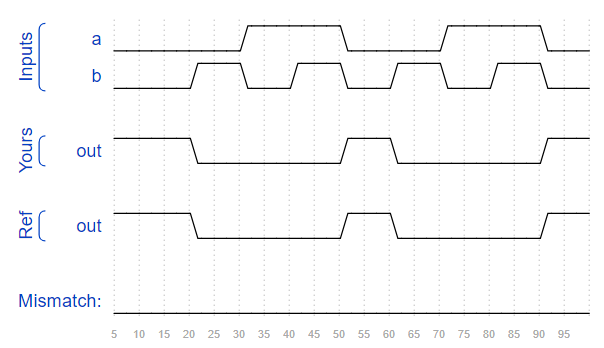
input b,

output out );

assign out=a&b;

endmodule

7. Create a module that implements a NOR gate. A NOR gate is an OR gate with its output inverted. A NOR function needs two operators when written in Verilog.

Ans;

module top\_module(

input a,

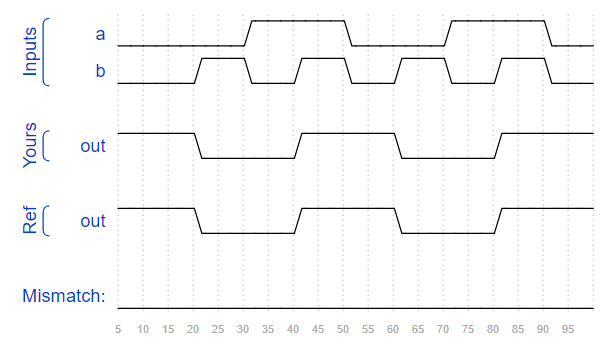
input b,

output out );

assign out=~(a|b);

endmodule

8. Create a module that implements an XNOR gate.

Ans;

module top\_module(

input a,

input b,

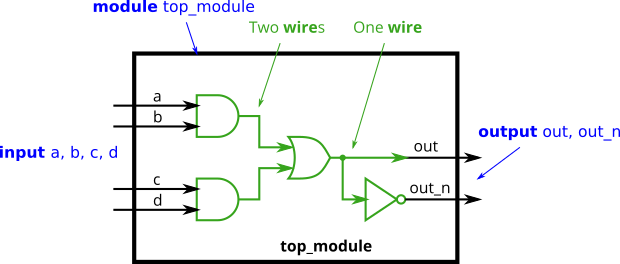
output out );

assign out=~(a^b);

endmodule

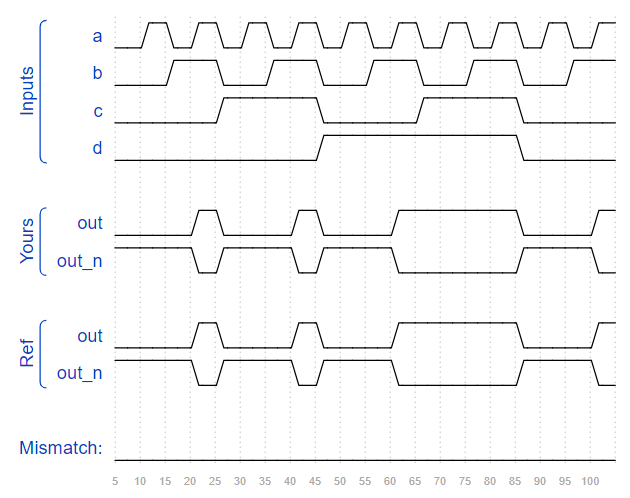
9. Implement the following circuit. Create two intermediate wires (named anything you want) to connect the AND and OR gates together. Note that the wire that feeds the NOT gate is really wire out, so you do not necessarily need to declare a third wire here. Notice how wires are driven by exactly one source (output of a gate), but can feed multiple inputs.

If you're following the circuit structure in the diagram, you should end up with four assign statements, as there are four signals that need a value assigned.



Ans;

`default\_nettype none

module top\_module(

input a,

input b,

input c,

input d,

output out,

output out\_n );

wire x,y;

assign x=a&b;

assign y=c&d;

assign out=x|y;

assign out\_n=~out;

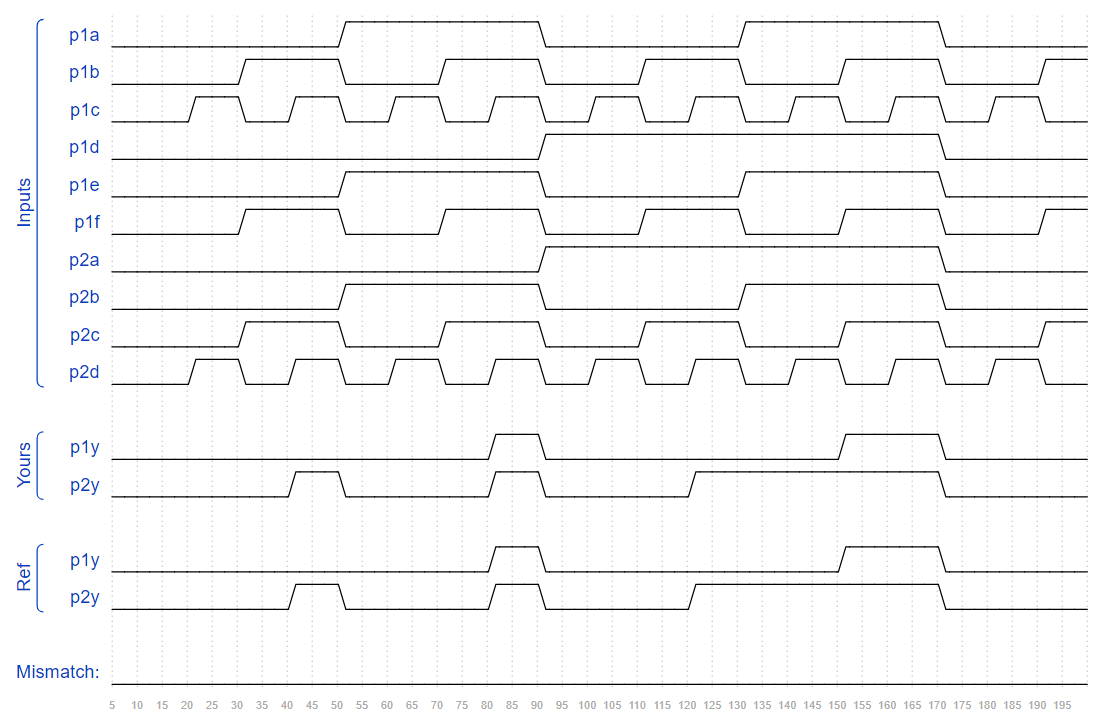
endmodule

10. The 7458 is a chip with four AND gates and two OR gates. This problem is slightly more complex than 7420.

Ans;

module top\_module (

input p1a, p1b, p1c, p1d, p1e, p1f,

 output p1y,

input p2a, p2b, p2c, p2d,

output p2y );

wire w1,w2,w3,w4;

and(w1,p1a,p1b,p1c);

and(w2,p1d,p1e,p1f);

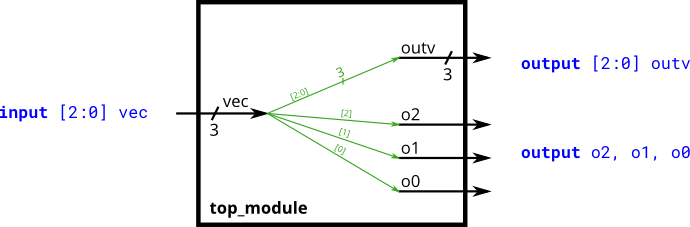
or(p1y,w1,w2);

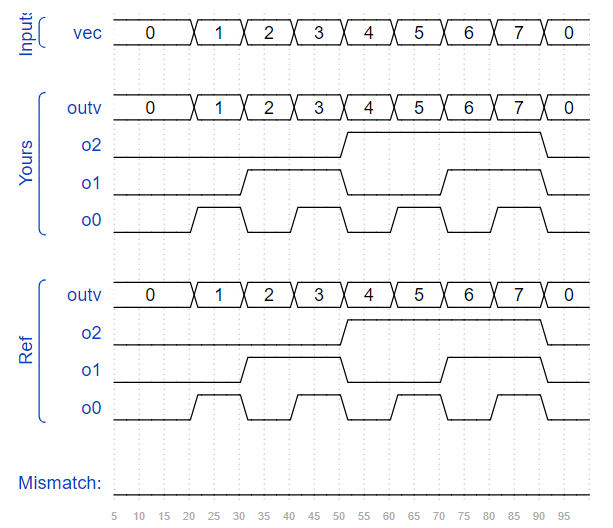
and(w3,p2a,p2b);

and(w4,p2c,p2d);

or(p2y,w3,w4);

endmodule

11. 

Ans;

module top\_module (

input wire [2:0] vec,

output wire [2:0] outv,

output wire o2,

output wire o1,

output wire o0 );

assign outv = vec;

assign o0 = vec[0];

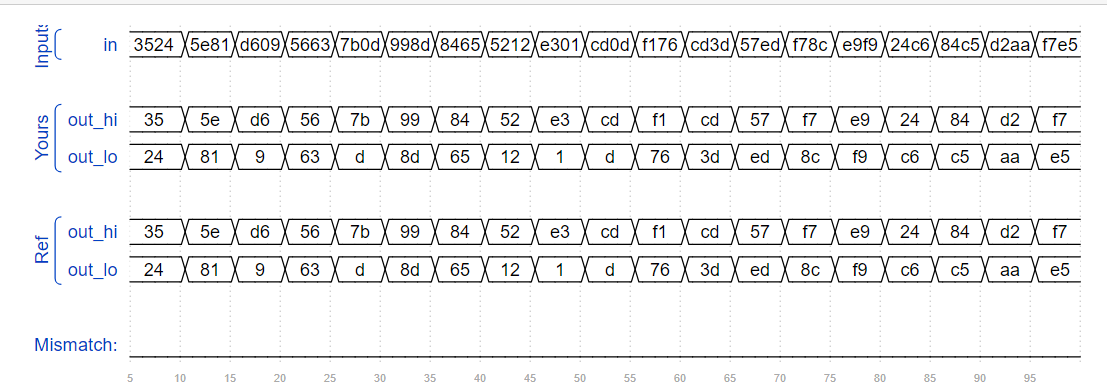
assign o1 = vec[1];

assign o2 = vec[2];

endmodule

12. Build a combinational circuit that splits an input half-word (16 bits, [15:0] ) into lower [7:0] and upper [15:8] bytes.

Ans;

module top\_module(

input wire [15:0] in,

output wire [7:0] out\_hi,

output wire [7:0] out\_lo );

assign out\_hi = in[15:8];

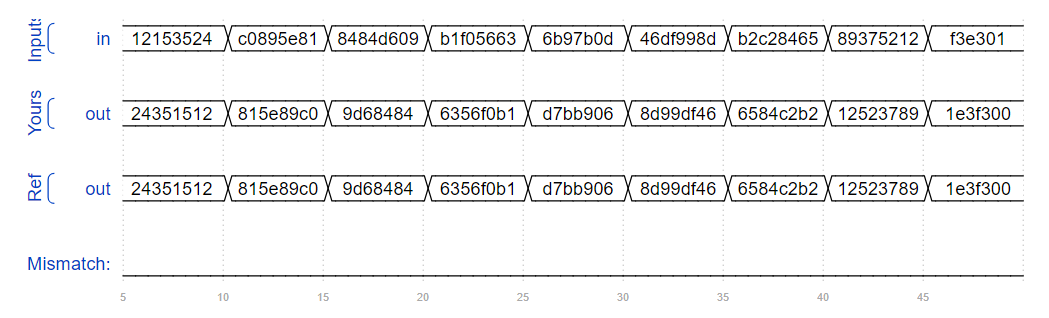
assign out\_lo = in[7:0];

endmodule

13. A 32-bit vector can be viewed as containing 4 bytes (bits [31:24], [23:16], etc.). Build a circuit that will reverse the byte ordering of the 4-byte word.

AaaaaaaaBbbbbbbbCcccccccDddddddd => DdddddddCcccccccBbbbbbbbAaaaaaaa

Ans;

module top\_module(

input [31:0] in,

output [31:0] out );

assign out[7:0] = in[31:24];

assign out[15:8] = in[23:16];

assign out[23:16] = in[15:8];

assign out[31:24] = in[7:0];

endmodule

14. Build a circuit that has two 3-bit inputs that computes the bitwise-OR of the two vectors, the logical-OR of the two vectors, and the inverse (NOT) of both vectors. Place the inverse of b in the upper half of out\_not (i.e., bits [5:3]), and the inverse of a in the lower half.

Ans;

module top\_module(

input [2:0] a,

input [2:0] b,

output [2:0] out\_or\_bitwise,

output out\_or\_logical,

output [5:0] out\_not

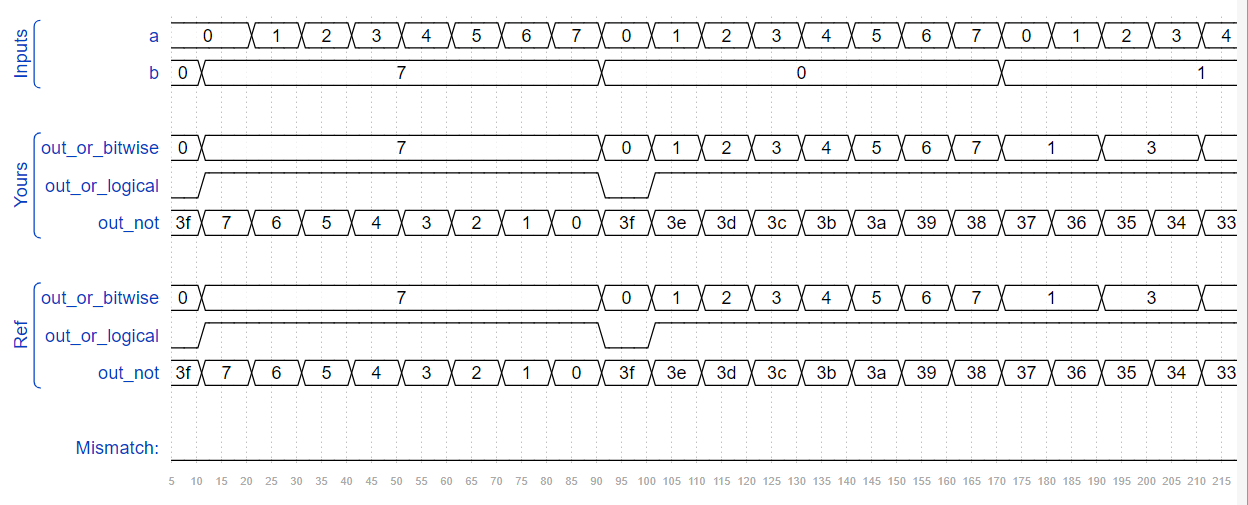
);

assign out\_or\_bitwise = a|b;

assign out\_or\_logical = a||b;

assign out\_not[5:3] = ~b;

assign out\_not[2:0] = ~a;

endmodule

15. Build a circuit that has two 3-bit inputs that computes the bitwise-OR of the two vectors, the logical-OR of the two vectors, and the inverse (NOT) of both vectors. Place the inverse of b in the upper half of out\_not (i.e., bits [5:3]), and the inverse of a in the lower half.

Ans;

module top\_module(

input [2:0] a,

input [2:0] b,

output [2:0] out\_or\_bitwise,

output out\_or\_logical,

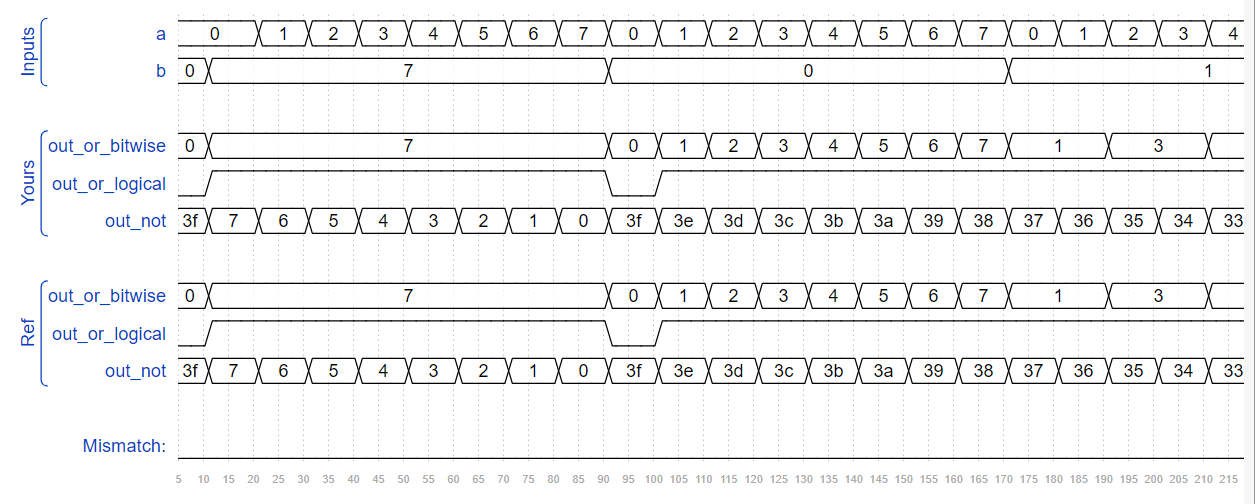
output [5:0] out\_not);

assign out\_or\_bitwise = a|b;

assign out\_or\_logical = a||b;

assign out\_not[5:3] = ~b;

assign out\_not[2:0] = ~a;

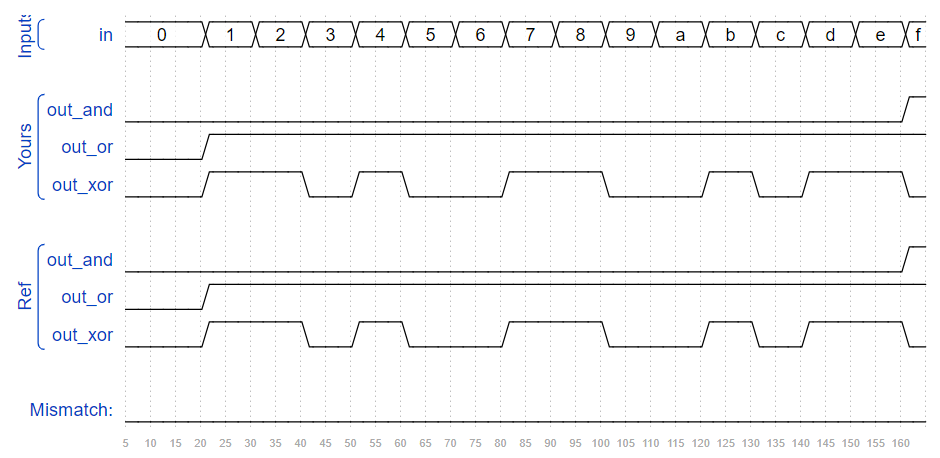
endmodule

16. Build a combinational circuit with four inputs, in[3:0].

There are 3 outputs:

out\_and: output of a 4-input AND gate.

out\_or: output of a 4-input OR gate.

out\_xor: output of a 4-input XOR gate.

Ans;

module top\_module(

input [3:0] in,

output out\_and,

output out\_or,

output out\_xor

);

assign out\_and = in[3] && in[2] && in[1] && in[0];

assign out\_or = in[3] || in[2] || in[1] || in[0];

assign out\_xor = in[3] ^ in[2] ^ in[1] ^ in[0];

endmodule

17. Given several input vectors, concatenate them together then split them up into several output vectors. There are six 5-bit input vectors: a, b, c, d, e, and f, for a total of 30 bits of input. There are four 8-bit output vectors: w, x, y, and z, for 32 bits of output. The output should be a concatenation of the input vectors followed by two 1 bits:

Ans;

module top\_module (

input [4:0] a, b, c, d, e, f,

output [7:0] w, x, y, z );

wire [31:0] concat\_reg;

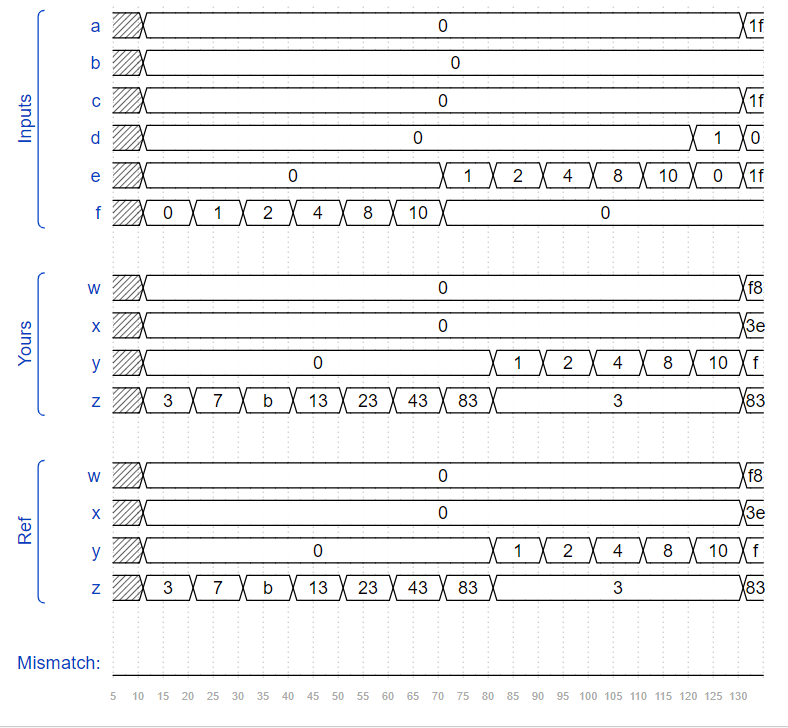
assign concat\_reg = {a[4:0], b[4:0], c[4:0], d[4:0], e[4:0], f[4:0], 2'b11};

assign w = concat\_reg[31:24];

assign x = concat\_reg[23:16];

assign y = concat\_reg[15:8];

assign z = concat\_reg[7:0];

endmodule

18. Given an 8-bit input vector [7:0], reverse its bit ordering.

Ans;

module top\_module(

input [7:0] in,

output [7:0] out

);

assign out = {in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7]};

endmodule

19. Build a circuit that sign-extends an 8-bit number to 32 bits. This requires a concatenation of 24 copies of the sign bit (i.e., replicate bit[7] 24 times) followed by the 8-bit number itself.

Ans;

module top\_module (

input [7:0] in,

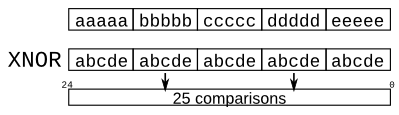
output [31:0] out );

assign out = { {24{in[7]}} , in[7:0] };

endmodule

20. Given five 1-bit signals (a, b, c, d, and e), compute all 25 pairwise one-bit comparisons in the 25-bit output vector. The output should be 1 if the two bits being compared are equal.

out[24] = ~a ^ a; // a == a, so out[24] is always 1.

out[23] = ~a ^ b;

out[22] = ~a ^ c;

...

out[ 1] = ~e ^ d;

out[ 0] = ~e ^ e;

Vector5.png

As the diagram shows, this can be done more easily using the replication and concatenation operators.

The top vector is a concatenation of 5 repeats of each input

The bottom vector is 5 repeats of a concatenation of the 5 inputs

Ans;

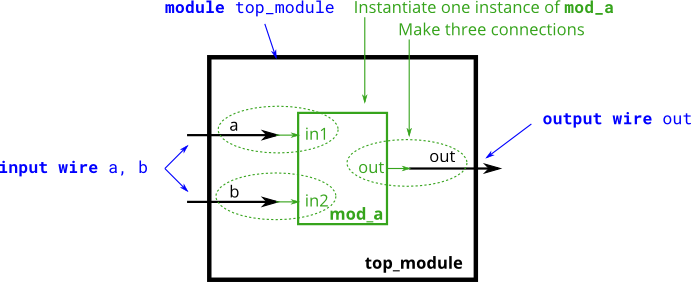
module top\_module (

input a, b, c, d, e,

output [24:0] out );//

assign out = ~{ {5{a}},{5{b}},{5{c}},{5{d}},{5{e}}} ^ { {5{a,b,c,d,e}}};

endmodule

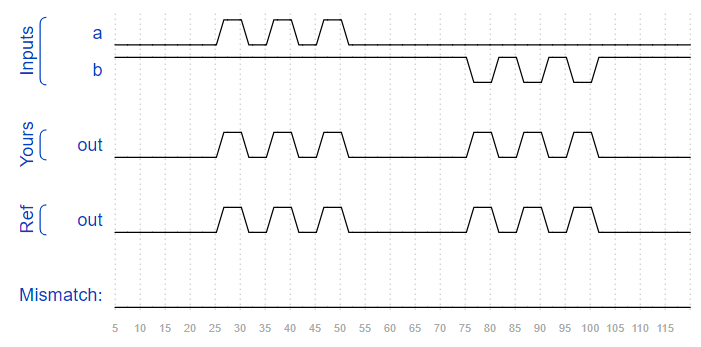
21. connect signals to the module by port name or port position.

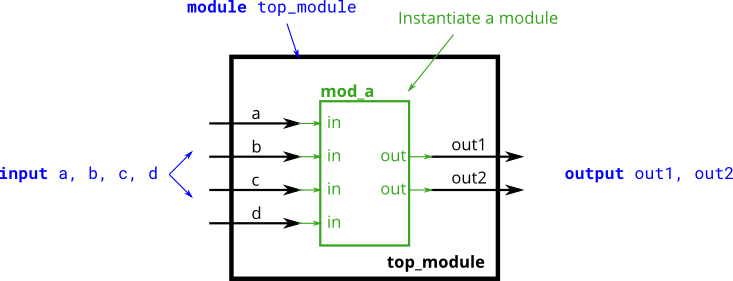
Ans;

module top\_module ( input a, input b, output out );

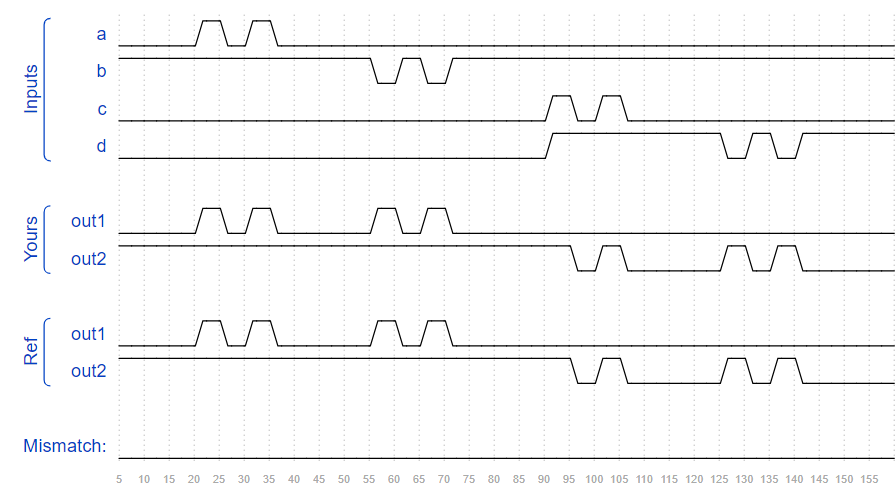
mod\_a instance\_1(a,b,out);

endmodule



22. You must connect the 6 ports by position to your top-level module's ports out1, out2, a, b, c, and d, in that order.

Ans;

module top\_module (

input a,

input b,

input c,

input d,

output out1,

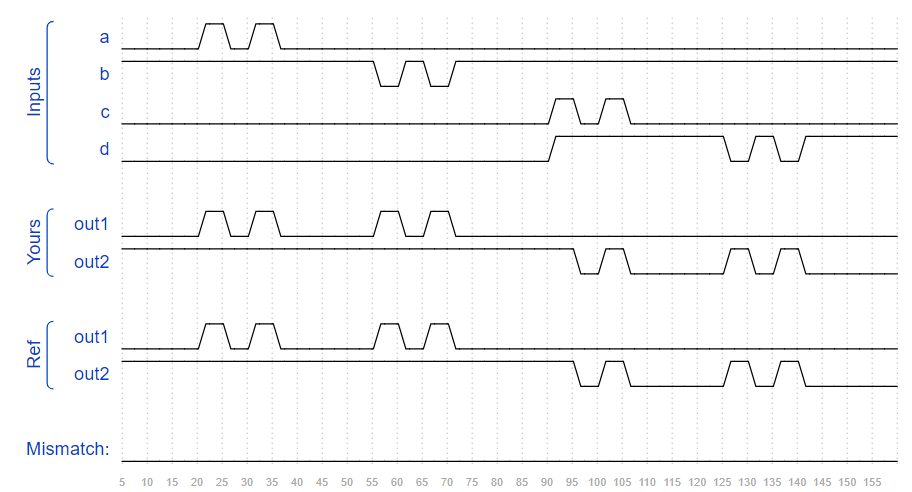
output out2

);

mod\_a inst\_1(out1,out2,a,b,c,d);

endmodule

23. You must connect the 6 ports by name to your top-level module's ports

Ans;

module top\_module (

input a,

input b,

input c,

input d,

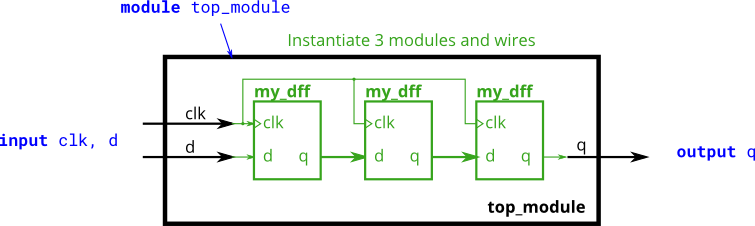
output out1,

output out2

);

mod\_a inst\_1(.in1(a),.in2(b),.in3(c),.in4(d),.out1(out1),.out2(out2));

endmodule

24. Instantiate three of them, then chain them together to make a shift register of length 3. The clk port needs to be connected to all instances.

Ans;

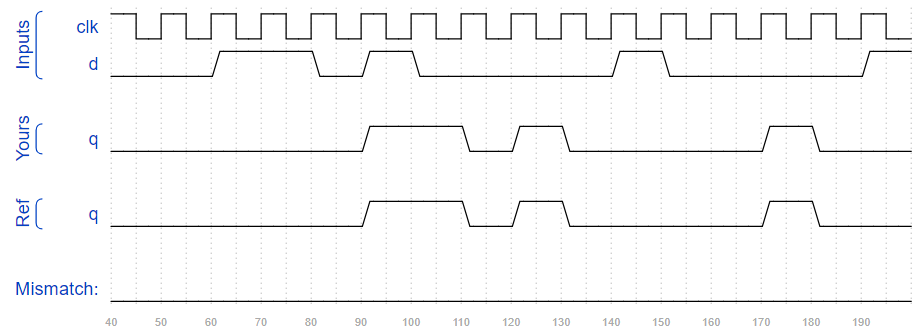
module top\_module ( input clk, input d, output q );

wire con1,con2;

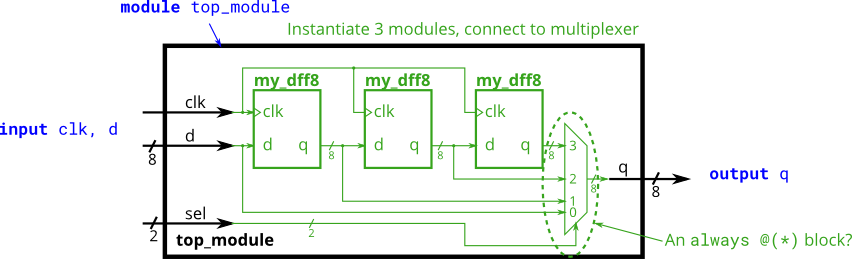
my\_dff d\_flop1(.clk(clk),.d(d),.q(con1));

my\_dff d\_flop2(.clk(clk),.d(con1),.q(con2));

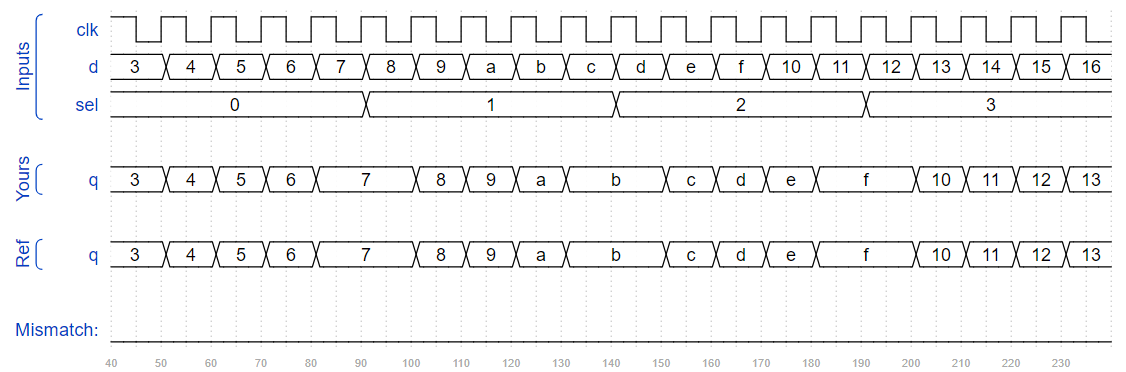
my\_dff d\_flop3(.clk(clk),.d(con2),.q(q));

endmodule

25. You are given a module my\_dff8 with two inputs and one output (that implements a set of 8 D flip-flops). Instantiate three of them, then chain them together to make a 8-bit wide shift register of length 3. In addition, create a 4-to-1 multiplexer (not provided) that chooses what to output depending on sel[1:0]: The value at the input d, after the first, after the second, or after the third D flip-flop. (Essentially, sel selects how many cycles to delay the input, from zero to three clock cycles.)



Ans;

module top\_module (

input clk,

input [7:0] d,

input [1:0] sel,

output reg [7:0] q

);

wire [7:0]con1,con2,con3;

my\_dff8 d\_flop1(.clk(clk), .d(d), .q(con1));

my\_dff8 d\_flop2(.clk(clk), .d(con1), .q(con2) );

my\_dff8 d\_flop3(.clk(clk), .d(con2), .q(con3));

always @ (\*) begin

case(sel)

0 : q = d ;

1 : q = con1;

2 : q = con2;

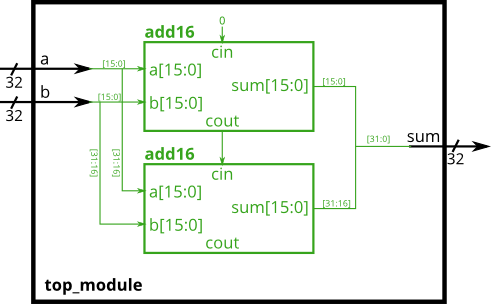
3 : q = con3;

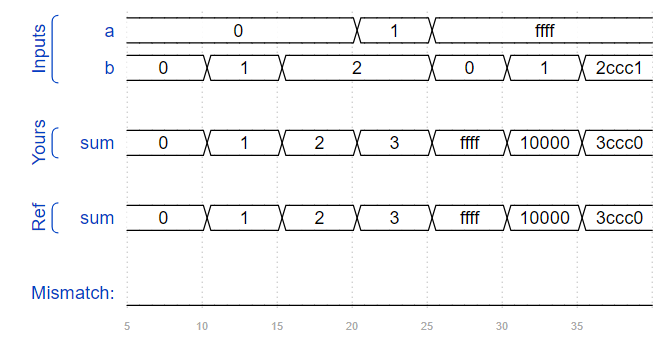
endcase

end

endmodule

26. You are given a module add16 that performs a 16-bit addition. Instantiate two of them to create a 32-bit adder. One add16 module computes the lower 16 bits of the addition result, while the second add16 module computes the upper 16 bits of the result, after receiving the carry-out from the first adder. Your 32-bit adder does not need to handle carry-in (assume 0) or carry-out (ignored), but the internal modules need to in order to function correctly. (In other words, the add16 module performs 16-bit a + b + cin, while your module performs 32-bit a + b).

Connect the modules together as shown in the diagram below. The provided module add16 has the following declaration:

Ans;

module top\_module(

input [31:0] a,

input [31:0] b,

output [31:0] sum

);

wire con1, con2;

add16 adder\_1(a[15:0], b[15:0], 0, sum[15:0], con1);

add16 adder\_2(a[31:16], b[31:16], con1, sum[31:16], con2);

endmodule

27. you are given a module add16 that performs a 16-bit addition. You must instantiate two of them to create a 32-bit adder. One add16 module computes the lower 16 bits of the addition result, while the second add16 module computes the upper 16 bits of the result. Your 32-bit adder does not need to handle carry-in (assume 0) or carry-out (ignored).

Connect the add16 modules together as shown in the diagram below. The provided module add16 has the following declaration:

module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );

Within each add16, 16 full adders (module add1, not provided) are instantiated to actually perform the addition. You must write the full adder module that has the following declaration:

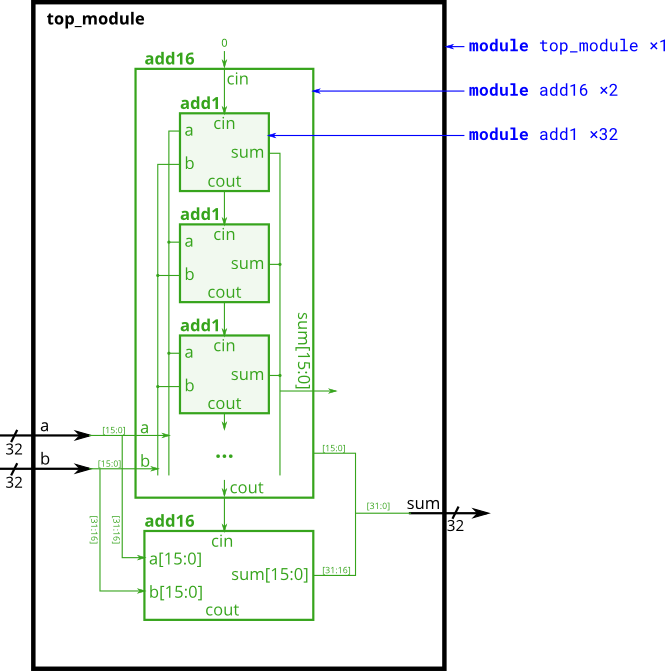
module add1 ( input a, input b, input cin, output sum, output cout );

Recall that a full adder computes the sum and carry-out of a+b+cin.

In summary, there are three modules in this design:

top\_module — Your top-level module that contains two of...

add16, provided — A 16-bit adder module that is composed of 16 of...

add1 — A 1-bit full adder module.

Ans;

module top\_module (

input [31:0] a,

input [31:0] b,

output [31:0] sum

);

wire con1, con2;

add16 adder\_1(a[15:0], b[15:0], 0, sum[15:0], con1);

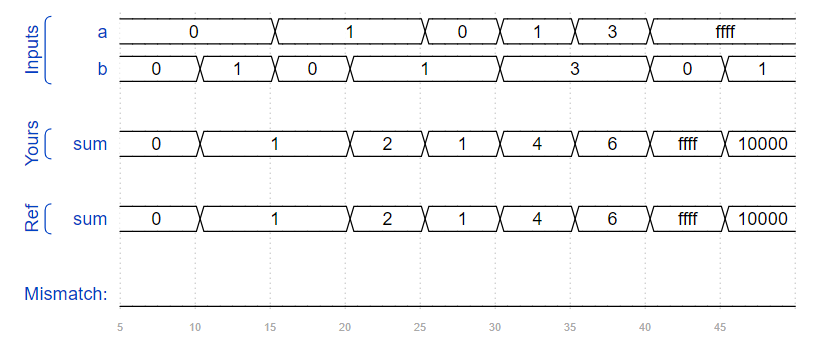
add16 adder\_2(a[31:16], b[31:16], con1, sum[31:16], con2);

endmodule

module add1 ( input a, input b, input cin, output sum, output cout );

assign sum = a^b^cin;

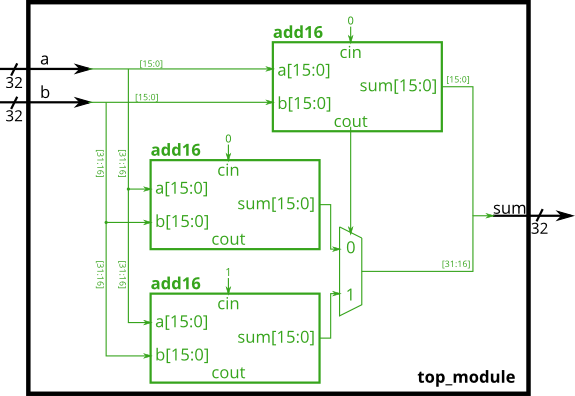
assign cout = (a&b)|(a&cin)|(b&cin);

endmodule

28. In this exercise, you are provided with the same module add16 as the previous exercise, which adds two 16-bit numbers with carry-in and produces a carry-out and 16-bit sum. You must instantiate three of these to build the carry-select adder, using your own 16-bit 2-to-1 multiplexer.

Connect the modules together as shown in the diagram below. The provided module add16 has the following declaration:

module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );

ans;

module top\_module(

input [31:0] a,

input [31:0] b,

output reg [31:0] sum

);

wire [15:0] cout,con2;

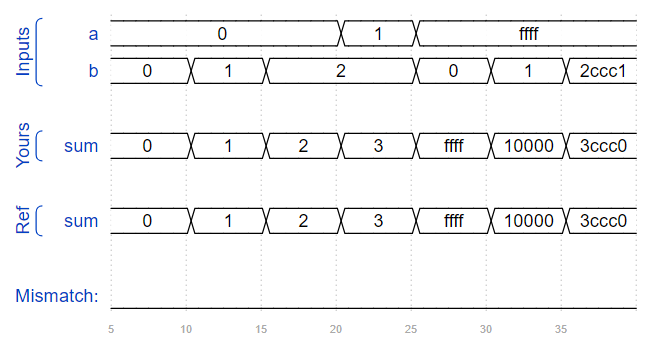
wire [15:0]alt\_sum1, alt\_sum2;

add16 adder1(a[15:0], b[15:0], 0, sum[15:0], cout);

add16 adder\_sel1(a[31:16], b[31:16], 0, alt\_sum1, con2);

add16 adder\_sel2(a[31:16], b[31:16], 1, alt\_sum2, con2);

always @(cout, alt\_sum1, alt\_sum2) begin

 case(cout)

0 : sum[31:16] = alt\_sum1;

1 : sum[31:16] = alt\_sum2;

endcase

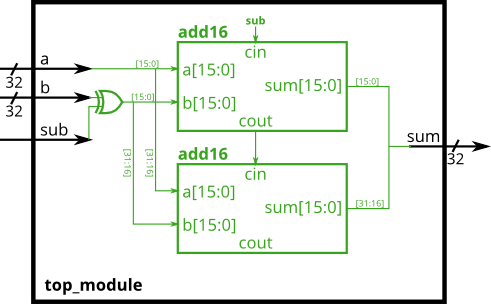
end

endmodule

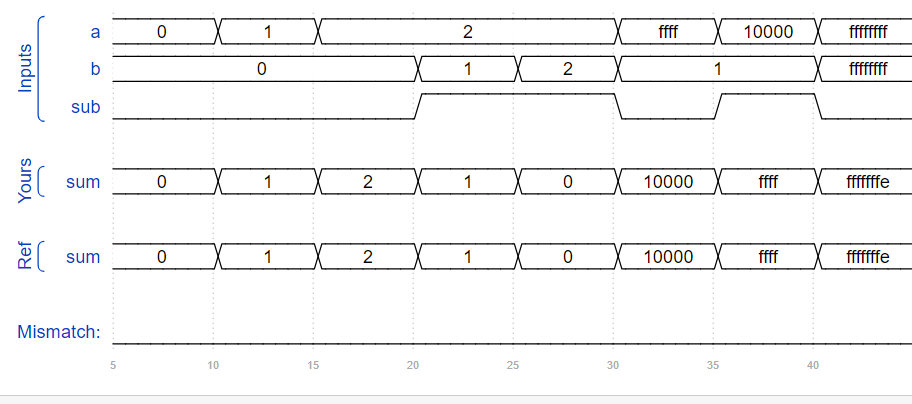
29. Build the adder-subtractor below.

You are provided with a 16-bit adder module, which you need to instantiate twice:

module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );



Ans;

module top\_module(

input [31:0] a,

input [31:0] b,

input sub,

output [31:0] sum

);

wire wire1;

wire [31:0]b\_xor;

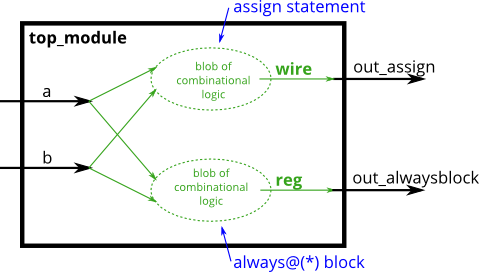
assign b\_xor = {32{sub}}^b;

add16 adder1(a[15:0], b\_xor[15:0], sub, sum[15:0], wire1);

add16 adder2(a[31:16], b\_xor[31:16], wire1, sum[31:16]);

endmodule

30. Build an AND gate using both an assign statement and a combinational always block. (Since assign statements and combinational always blocks function identically, there is no way to enforce that you're using both methods. But you're here for practice, right?...)

Ans;

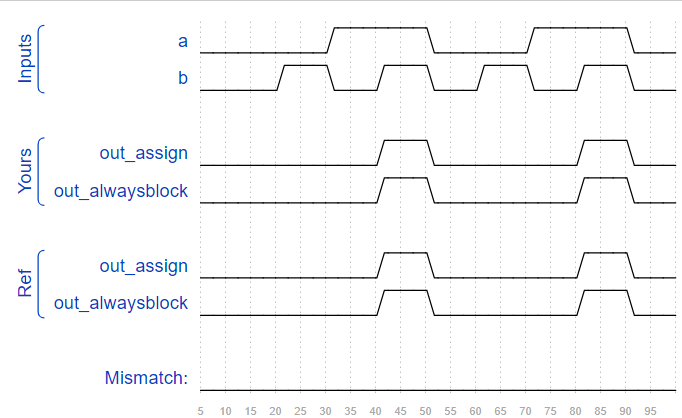
module top\_module(

input a,

input b,

output wire out\_assign,

output reg out\_alwaysblock

);

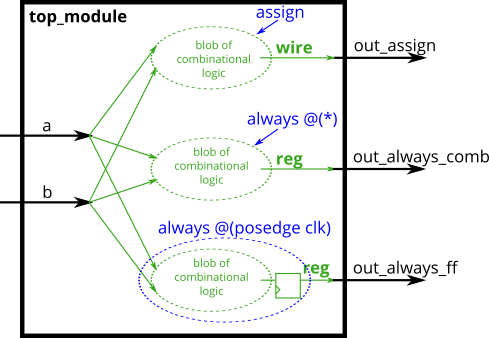
assign out\_assign = a&&b;

always @(\*) begin

out\_alwaysblock = a&&b;

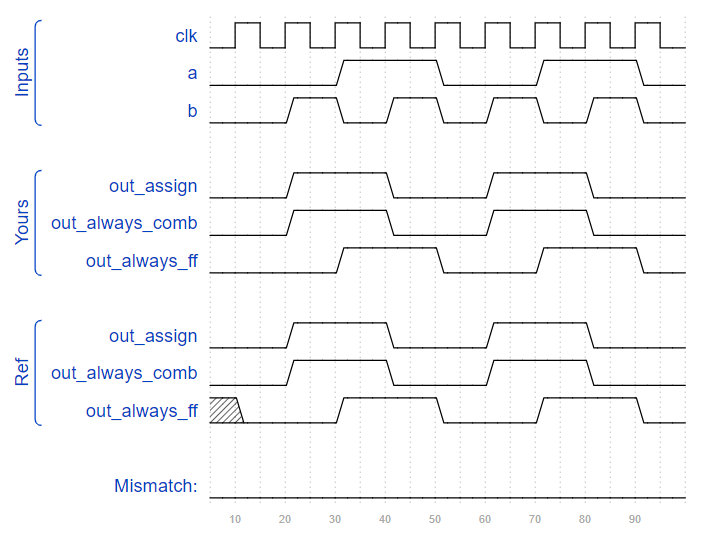
end

endmodule

31. Build an XOR gate three ways, using an assign statement, a combinational always block, and a clocked always block. Note that the clocked always block produces a different circuit from the other two: There is a flip-flop so the output is delayed.

Ans;

module top\_module(

 input clk,

input a,

input b,

output wire out\_assign,

output reg out\_always\_comb,

output reg out\_always\_ff );

assign out\_assign = a^b;

always @ (\*) begin

out\_always\_comb = a^b;

end

always @ (posedge clk) begin

out\_always\_ff <= a^b;

end

endmodule

32. Build a 2-to-1 mux that chooses between a and b. Choose b if both sel\_b1 and sel\_b2 are true. Otherwise, choose a. Do the same twice, once using assign statements and once using a procedural if statement.

Ans;

module top\_module(

input a,

input b,

input sel\_b1,

input sel\_b2,

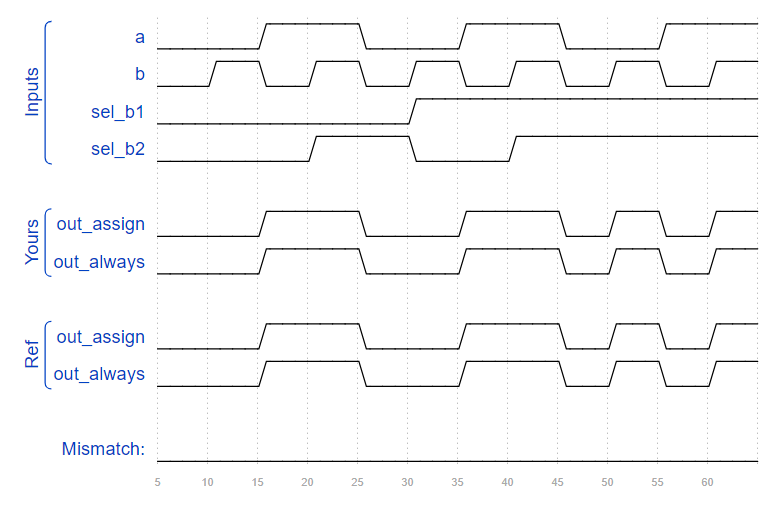
output wire out\_assign,

output reg out\_always );

assign out\_assign= (sel\_b1==1'b1 && sel\_b2==1'b1)? b:a;

always @(\*)begin

if (sel\_b1==1'b1 && sel\_b2==1'b1) begin

 out\_always = b;

end

else begin

out\_always = a;

end

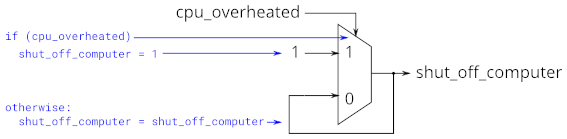
end

endmodule

33. The following code contains incorrect behaviour that creates a latch. Fix the bugs so that you will shut off the computer only if it's really overheated, and stop driving if you've arrived at your destination or you need to refuel.

always @(\*) begin

if (cpu\_overheated)

 shut\_off\_computer = 1;

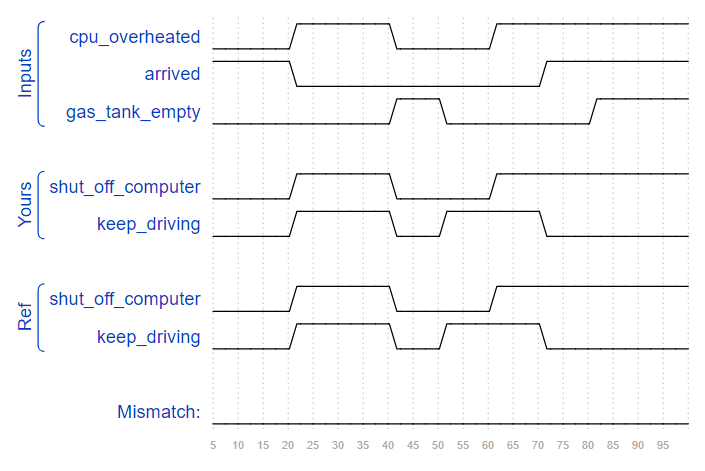
end

always @(\*) begin

if (~arrived)

keep\_driving = ~gas\_tank\_empty;

end



ans;

module top\_module (

input cpu\_overheated,

output reg shut\_off\_computer,

input arrived,

input gas\_tank\_empty,

output reg keep\_driving );

always @(\*) begin

if (cpu\_overheated)

shut\_off\_computer = 1;

else

shut\_off\_computer = 0; // Fix: Ensure shut\_off\_computer is driven to 0 when not overheated

end

always @(\*) begin

if (~arrived)

keep\_driving = ~gas\_tank\_empty;

else

keep\_driving = 0; // Fix: Ensure keep\_driving is driven to 0 when arrived

end

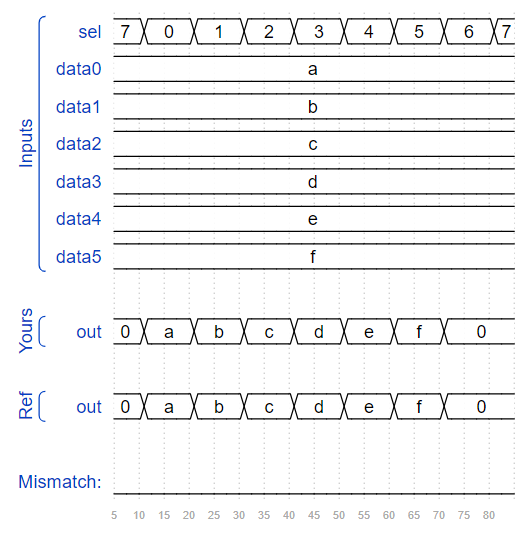
endmodule

34. Case statements are more convenient than if statements if there are a large number of cases. So, in this exercise, create a 6-to-1 multiplexer. When sel is between 0 and 5, choose the corresponding data input. Otherwise, output 0. The data inputs and outputs are all 4 bits wide.

Ans;

module top\_module (

input [2:0] sel,

 input [3:0] data0,

input [3:0] data1,

input [3:0] data2,

input [3:0] data3,

input [3:0] data4,

input [3:0] data5,

output reg [3:0] out );

always@(\*) begin

case(sel)

3'b000: out = data0;

3'b001: out = data1;

3'b010: out = data2;

3'b011: out = data3;

3'b100: out = data4;

3'b101: out = data5;

default: out = 4'b0000;

endcase

end

endmodule

35. Build a 4-bit priority encoder. For this problem, if none of the input bits are high (i.e., input is zero), output zero. Note that a 4-bit number has 16 possible combinations.

Ans;

module top\_module (

input [3:0] in,

output reg [1:0] pos );

always @(\*) begin

if (in[0]==1'b1)

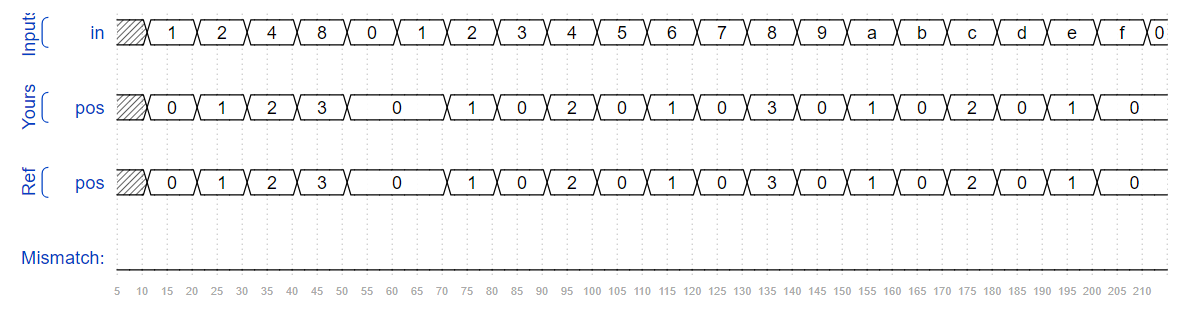
pos = 0;

else if (in[1]==1'b1)

pos = 1;

else if (in[2]==1'b1)

pos = 2;

 else if (in[3]==1'b1)

pos = 3;

else

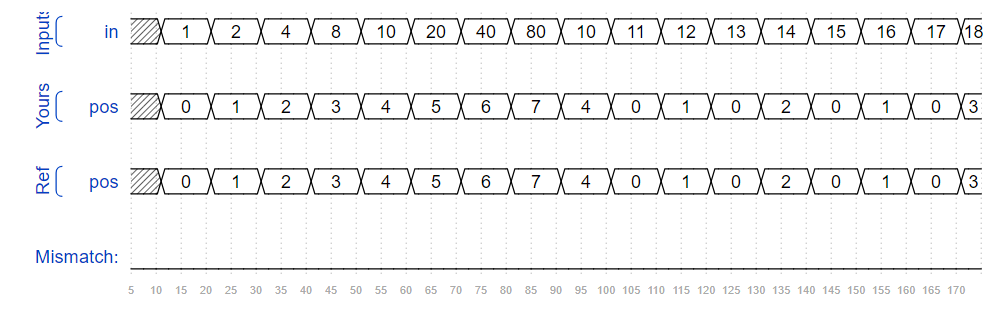
pos = 0;

end

endmodule

36. Build a priority encoder for 8-bit inputs. Given an 8-bit vector, the output should report the first (least significant) bit in the vector that is 1. Report zero if the input vector has no bits that are high. For example, the input 8'b10010000 should output 3'd4, because bit[4] is first bit that is high.

Ans;

module top\_module (

input [7:0] in,

output reg [2:0] pos );

always @(\*) begin

casez (in[7:0])

8'bzzzzzzz1: pos = 0;

8'bzzzzzz1z: pos = 1;

8'bzzzzz1zz: pos = 2;

8'bzzzz1zzz: pos = 3;

8'bzzz1zzzz: pos = 4;

8'bzz1zzzzz: pos = 5;

8'bz1zzzzzz: pos = 6;

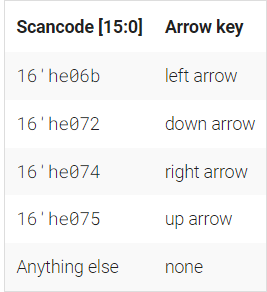
8'b1zzzzzzz: pos = 7;

default: pos = 0;

endcase

end

endmodule

37. you're building a circuit to process scancodes from a PS/2 keyboard for a game. Given the last two bytes of scancodes received, you need to indicate whether one of the arrow keys on the keyboard have been pressed. This involves a fairly simple mapping, which can be implemented as a case statement (or if-elseif) with four cases. 

Build this circuit that recognizes these four scancodes and asserts the correct output.

Ans;

module top\_module (

input [15:0] scancode,

output reg left,

output reg down,

output reg right,

output reg up );

always @ (scancode) begin

up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;

case (scancode)

16'he06b:begin

up = 1'b0; down = 1'b0; left = 1'b1; right = 1'b0;

end

16'he072:begin

up = 1'b0; down = 1'b1; left = 1'b0; right = 1'b0;

end

16'he074:begin

up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b1;

end

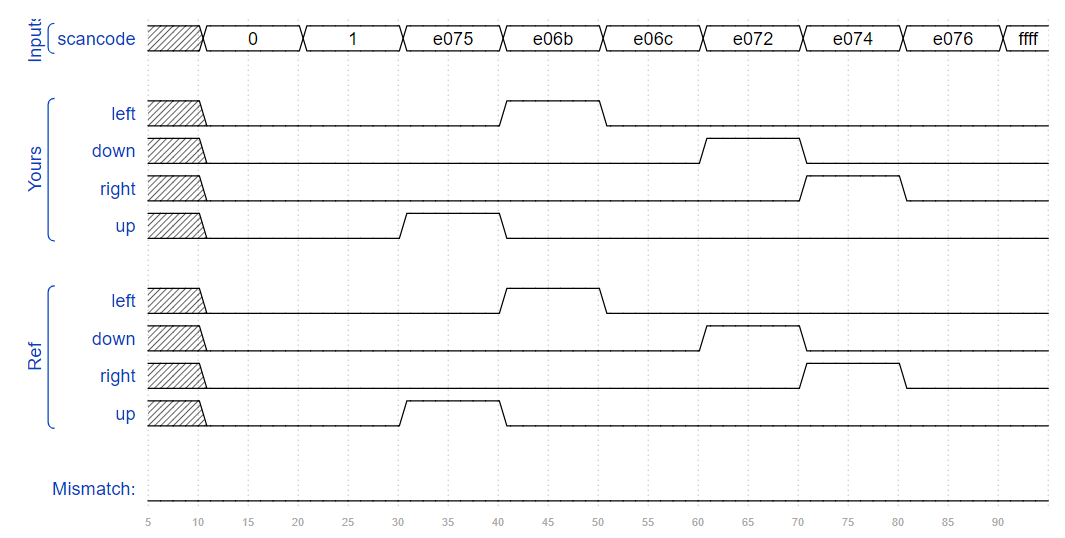
16'he075:begin

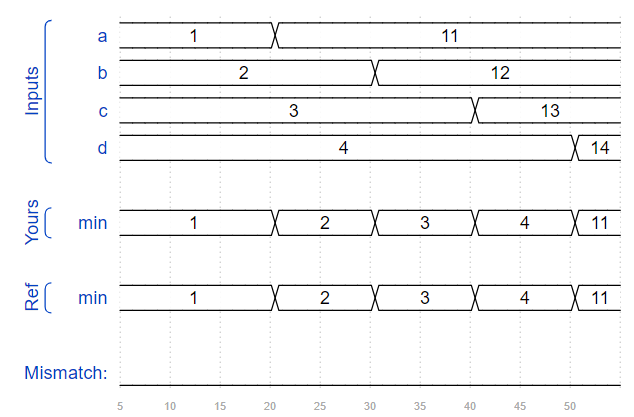
up = 1'b1; down = 1'b0; left = 1'b0; right = 1'b0;

end

endcase

end

endmodule

38. Given four unsigned numbers, find the minimum. Unsigned numbers can be compared with standard comparison operators (a < b). Use the conditional operator to make two-way min circuits, then compose a few of them to create a 4-way min circuit. You'll probably want some wire vectors for the intermediate results.

Ans;

module top\_module (

input [7:0] a, b, c, d,

output [7:0] min);

wire [7:0] int1, int2;

assign int1 = (a<b)? a:b;

assign int2 = (int1<c)? int1:c;

assign min = (int2<d)? int2:d;

endmodule

39. Parity checking is often used as a simple method of detecting errors when transmitting data through an imperfect channel. Create a circuit that will compute a parity bit for a 8-bit byte (which will add a 9th bit to the byte). We will use "even" parity, where the parity bit is just the XOR of all 8 data bits.

Expected solution length: Around 1 line.

Ans;

module top\_module (

input [7:0] in,

output parity);

assign parity = ^in;

endmodule

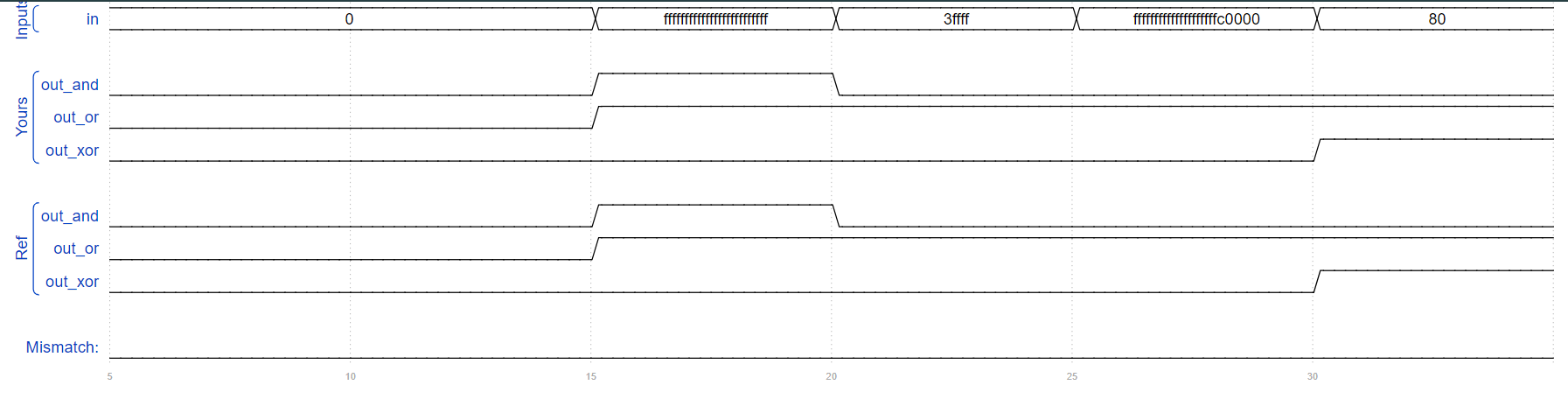
40. Build a combinational circuit with 100 inputs, in[99:0].

There are 3 outputs:

out\_and: output of a 100-input AND gate.

out\_or: output of a 100-input OR gate.

out\_xor: output of a 100-input XOR gate.

Ans;

module top\_module(

input [99:0] in,

output out\_and,

output out\_or,

output out\_xor

);

assign out\_and = &in;

assign out\_or = |in;

assign out\_xor = ^in;

endmodule

41. Given a 100-bit input vector [99:0], reverse its bit ordering.

Ans;

module top\_module(

input [99:0] in,

output [99:0] out

);

integer i;

always @ (in) begin

for (i=0;i<100; i=i+1) begin

out[99-i] = in[i];

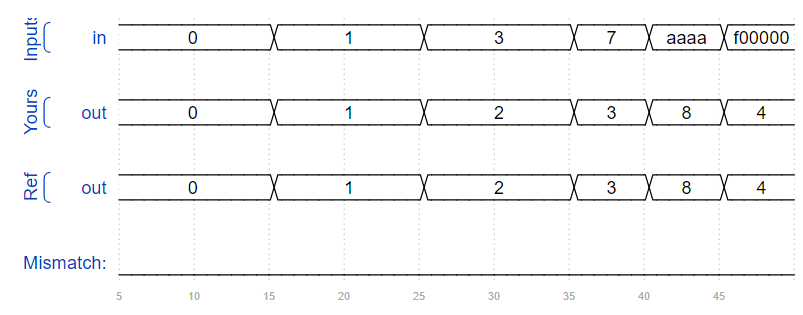
end

end

endmodule

42. A "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 255-bit input vector.

Ans;

module top\_module(

input [254:0] in,

output [7:0] out );

integer i;

reg [7:0] counter;

always @ (in) begin

counter = 0; for (i=0; i<255; i=i+1) begin

if (in[i]==1'b1)

counter = counter+1'b1;

end

out = counter;

end

endmodule

43. Create a 100-bit binary ripple-carry adder by instantiating 100 full adders. The adder adds two 100-bit numbers and a carry-in to produce a 100-bit sum and carry out. To encourage you to actually instantiate full adders, also output the carry-out from each full adder in the ripple-carry adder. cout[99] is the final carry-out from the last full adder, and is the carry-out you usually see

Ans;

module one\_bit\_FA(

input a,b,

input cin,

output cout,sum);

assign sum = a^b^cin;

assign cout = (a&b)|(b&cin)|(cin&a);

endmodule

module top\_module(

input [99:0] a, b,

input cin,

output [99:0] cout,

output [99:0] sum );

genvar i;

one\_bit\_FA FA1(a[0],b[0],cin,cout[0],sum[0]);

for (i=1; i<100; i=i+1) begin : Full\_adder\_block

one\_bit\_FA FA(a[i],b[i],cout[i-1],cout[i],sum[i]);

end

endgenerate

endmodule

44. You are provided with a BCD one-digit adder named bcd\_fadd that adds two BCD digits and carry-in, and produces a sum and carry-out.

module bcd\_fadd (

input [3:0] a,

input [3:0] b,

input cin,

output cout,

output [3:0] sum );

Instantiate 100 copies of bcd\_fadd to create a 100-digit BCD ripple-carry adder. Your adder should add two 100-digit BCD numbers (packed into 400-bit vectors) and a carry-in to produce a 100-digit sum and carry out.

Ans;

module top\_module(

input [399:0] a, b,

input cin,

output cout,

output [399:0] sum );

wire[99:0] cout\_wires;

genvar i;

generate

bcd\_fadd(a[3:0], b[3:0], cin, cout\_wires[0],sum[3:0]);

for (i=4; i<400; i=i+4) begin: bcd\_adder\_instances

bcd\_fadd bcd\_adder(a[i+3:i], b[i+3:i], cout\_wires[i/4-1],cout\_wires[i/4],sum[i+3:i]);

end

endgenerate

assign cout = cout\_wires[99];

endmodule

45. Implement the following circuit:

Exams m2014q4h.png

Ans;

module top\_module (

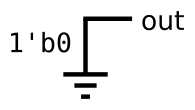
input in,

output out);

assign out = in;

endmodule

46. Implement the following circuit:



Ans;

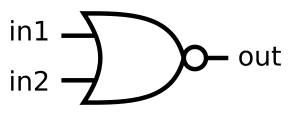
module top\_module (

output out);

assign out = 1'b0;

endmodule

47. Implement the following circuit:



Ans;

module top\_module (

input in1,

input in2,

output out);

assign out = ~(in1|in2);

endmodule

48. Implement the following circuit:



Ans;

module top\_module (

input in1,

input in2,

output out);

assign out = in1&&~in2;

endmodule

49. Implement the following circuit:



Ans;

module top\_module (

input in1,

input in2,

input in3,

output out);

wire wire1;

assign wire1 = ~(in1^in2);

assign out = wire1^in3;

endmodule

50. Build a combinational circuit with two inputs, a and b.

There are 7 outputs, each with a logic gate driving it:

out\_and: a and b

out\_or: a or b

out\_xor: a xor b

out\_nand: a nand b

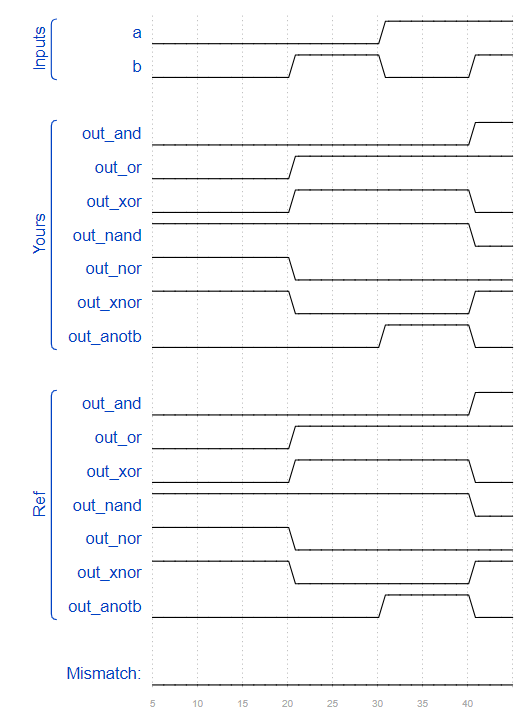
out\_nor: a nor b

out\_xnor: a xnor b

out\_anotb: a and-not b

ans;

module top\_module(

 input a, b,

output out\_and,

output out\_or,

output out\_xor,

output out\_nand,

output out\_nor,

output out\_xnor,

output out\_anotb

);

assign out\_and = a && b;

assign out\_or = a || b;

assign out\_xor = a ^ b;

assign out\_nand = ~(a&&b);

assign out\_nor = ~(a||b);

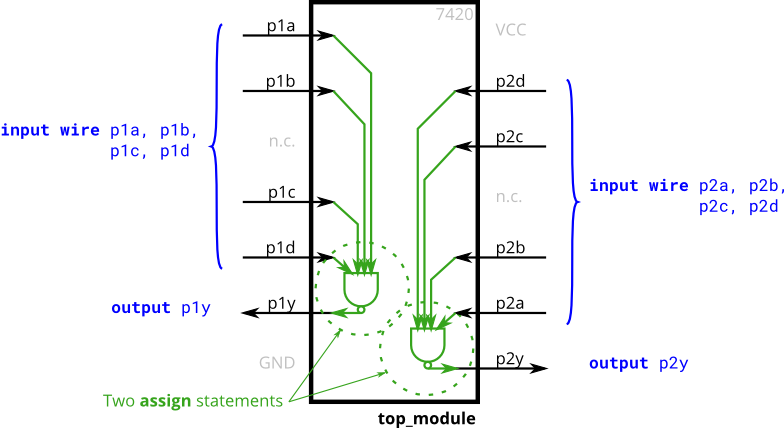
assign out\_xnor = ~(a^b);

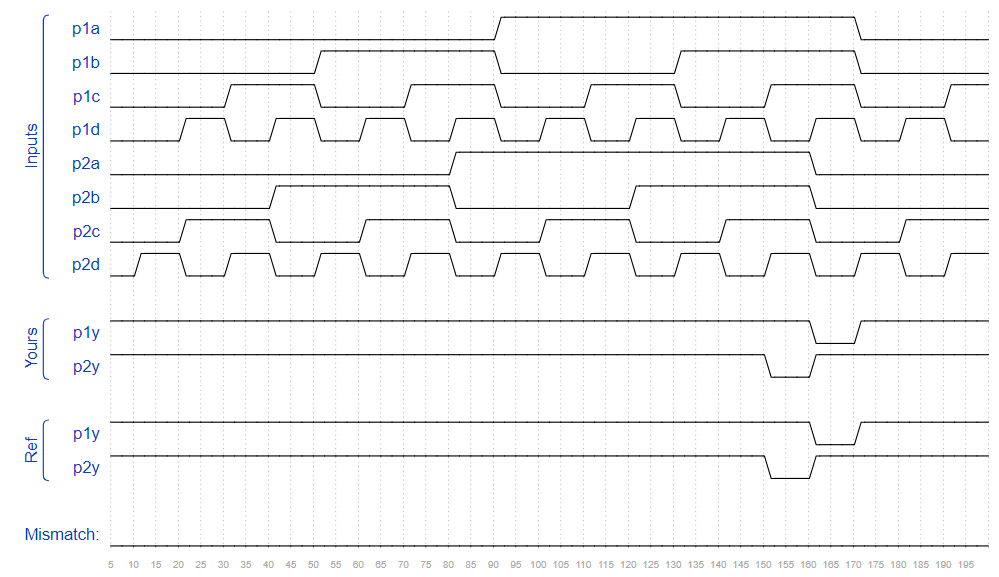
assign out\_anotb = a&&~b;

endmodule

51. The 7400-series integrated circuits are a series of digital chips with a few gates each. The 7420 is a chip with two 4-input NAND gates.

Create a module with the same functionality as the 7420 chip. It has 8 inputs and 2 outputs.



Ans;

module top\_module (

input p1a, p1b, p1c, p1d,

output p1y,

input p2a, p2b, p2c, p2d,

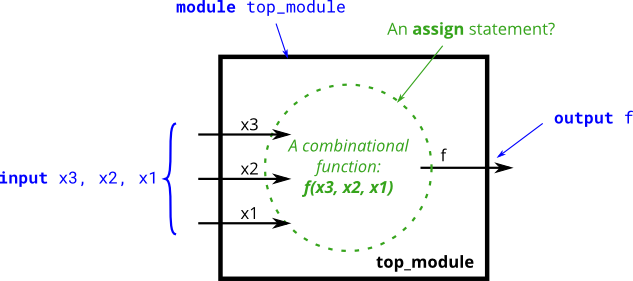
output p2y );

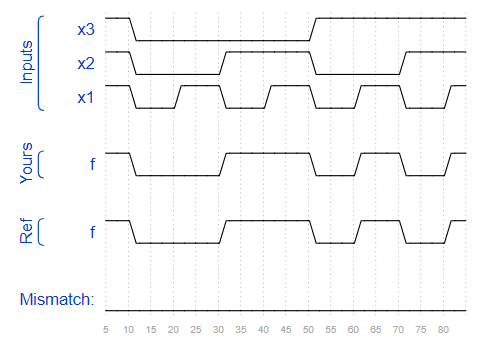
assign p1y = ~(p1a && p1b && p1c && p1d);

assign p2y = ~(p2a && p2b && p2c && p2d);

endmodule

52. Create a combinational circuit that implements the above truth table.



Ans;

module top\_module(

input x3,

input x2,

input x1,

output f

);

assign f = (~x3&x2&~x1) | (~x3&x2&x1) | (x3&~x2&x1) | (x3&x2&x1);

endmodule

53. Create a circuit that has two 2-bit inputs A[1:0] and B[1:0], and produces an output z. The value of z should be 1 if A = B, otherwise z should be 0.

Ans;

module top\_module ( input [1:0] A, input [1:0] B, output z );

assign z = (A==B);

endmodule

54. Module A is supposed to implement the function z = (x^y) & x. Implement this module.

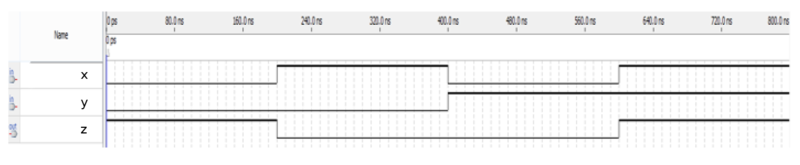
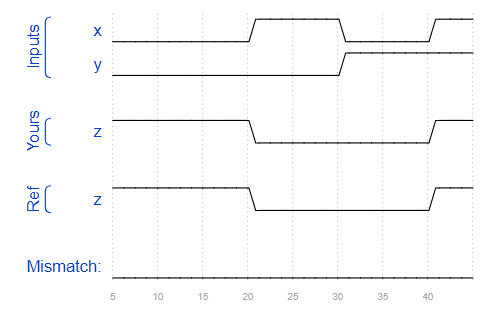
Ans;

module top\_module (input x, input y, output z);

assign z = (x^y) & x;

endmodule

55. Circuit B can be described by the following simulation waveform: Mt2015 q4b.png

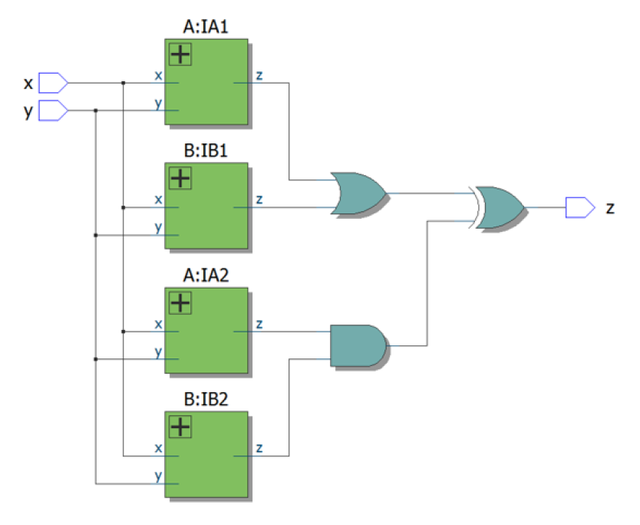


Ans;

module top\_module ( input x, input y, output z );

assign z= ~(x^y);

endmodule

56. The top-level design consists of two instantiations each of subcircuits A and B, as shown below. 

Implement this circuit.

Ans;

module top\_module (input x, input y, output z);

wire z1, z2, z3, z4,z5,z6;

A IA1(x,y,z1);

B IB1(x,y,z2);

A IA2(x,y,z3);

B IB2(x,y,z4);

assign z5 = z1||z2;

assign z6 = z3&&z4;

assign z = z5^z6;

endmodule

module B( input x, input y, output z );

assign z= ~(x^y);

endmodule

module A(input x, input y, output z);

assign z = (x^y) & x;

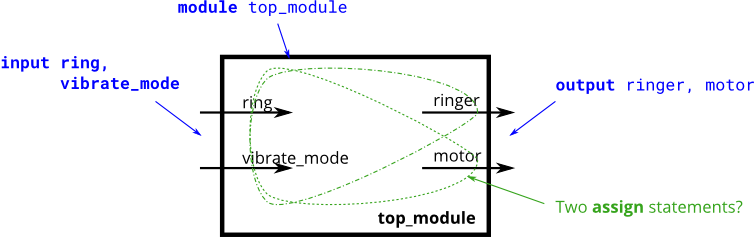
endmodule

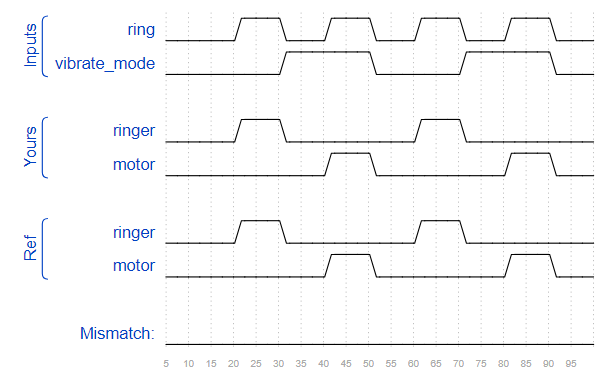
57. Suppose you are designing a circuit to control a cellphone's ringer and vibration motor. Whenever the phone needs to ring from an incoming call (input ring), your circuit must either turn on the ringer (output ringer = 1) or the motor (output motor = 1), but not both. If the phone is in vibrate mode (input vibrate\_mode = 1), turn on the motor. Otherwise, turn on the ringer.

Try to use only assign statements, to see whether you can translate a problem description into a collection of logic gates.

Design hint: When designing circuits, one often has to think of the problem "backwards", starting from the outputs then working backwards towards the inputs. This is often the opposite of how one would think about a (sequential, imperative) programming problem, where one would look at the inputs first then decide on an action (or output). For sequential programs, one would often think "If (inputs are \_\_\_ ) then (output should be \_\_\_ )". On the other hand, hardware designers often think "The (output should be \_\_\_ ) when (inputs are \_\_\_ )".

The above problem description is written in an imperative form suitable for software programming (if ring then do this), so you must convert it to a more declarative form suitable for hardware implementation (assign ringer = \_\_\_). Being able to think in, and translate between, both styles is one of the most important skills needed for hardware design.



Ans;

module top\_module (

input ring,

input vibrate\_mode,

output ringer, // sound

output motor // Vibrate

);

assign motor = vibrate\_mode && ring;

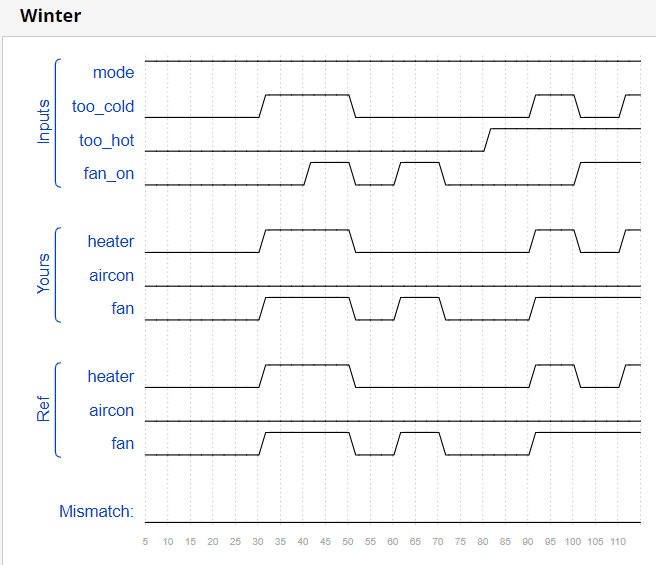
assign ringer = (~vibrate\_mode) && ring;

endmodule

58. A heating/cooling thermostat controls both a heater (during winter) and an air conditioner (during summer). Implement a circuit that will turn on and off the heater, air conditioning, and blower fan as appropriate.

The thermostat can be in one of two modes: heating (mode = 1) and cooling (mode = 0). In heating mode, turn the heater on when it is too cold (too\_cold = 1) but do not use the air conditioner. In cooling mode, turn the air conditioner on when it is too hot (too\_hot = 1), but do not turn on the heater. When the heater or air conditioner are on, also turn on the fan to circulate the air. In addition, the user can also request the fan to turn on (fan\_on = 1), even if the heater and air conditioner are off.

Try to use only assign statements, to see whether you can translate a problem description into a collection of logic gates.

Ans;

module top\_module (

input too\_cold,

input too\_hot,

input mode,

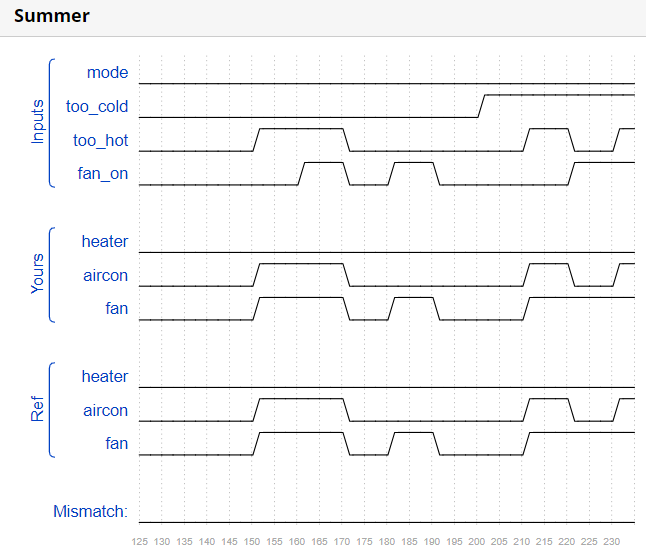
input fan\_on,

output heater,

output aircon,

output fan

);

 assign heater = mode && too\_cold;

assign aircon = (~mode) && too\_hot;

assign fan = aircon || heater || fan\_on;

endmodule

59. A "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 3-bit input vector.

Ans;

module top\_module(

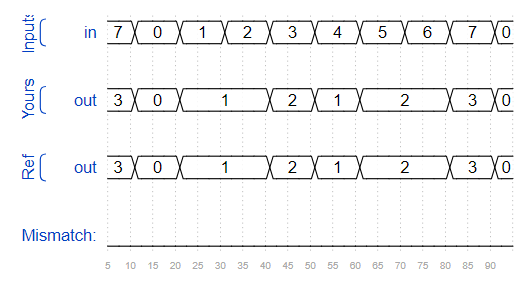
input [2:0] in,

output [1:0] out );

integer i,count;

always @ \* begin

count = 0;

 for (i=0; i<3;i=i+1) begin

if (in[i]==1'b1) count=count+1;

end

out = count;

end

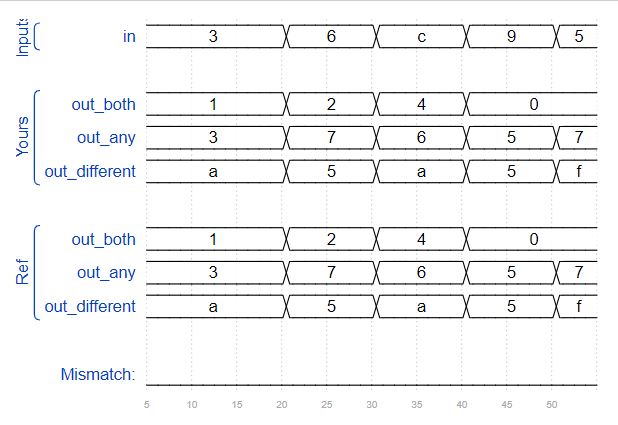
endmodule

60. You are given a four-bit input vector in[3:0]. We want to know some relationships between each bit and its neighbour:

out\_both: Each bit of this output vector should indicate whether both the corresponding input bit and its neighbour to the left (higher index) are '1'. For example, out\_both[2] should indicate if in[2] and in[3] are both 1. Since in[3] has no neighbour to the left, the answer is obvious so we don't need to know out\_both[3].

out\_any: Each bit of this output vector should indicate whether any of the corresponding input bit and its neighbour to the right are '1'. For example, out\_any[2] should indicate if either in[2] or in[1] are 1. Since in[0] has no neighbour to the right, the answer is obvious so we don't need to know out\_any[0].

out\_different: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbour to the left. For example, out\_different[2] should indicate if in[2] is different from in[3]. For this part, treat the vector as wrapping around, so in[3]'s neighbour to the left is in[0].

Ans;

module top\_module(

input [3:0] in,

output [2:0] out\_both,

output [3:1] out\_any,

output [3:0] out\_different );

assign out\_both = in[2:0] & in[3:1];

assign out\_any = in[3:1] | in[2:0];

assign out\_different = in ^ {in[0],in[3:1]};

endmodule

61. You are given a 100-bit input vector in[99:0]. We want to know some relationships between each bit and its neighbour:

out\_both: Each bit of this output vector should indicate whether both the corresponding input bit and its neighbour to the left are '1'. For example, out\_both[98] should indicate if in[98] and in[99] are both 1. Since in[99] has no neighbour to the left, the answer is obvious so we don't need to know out\_both[99].

out\_any: Each bit of this output vector should indicate whether any of the corresponding input bit and its neighbour to the right are '1'. For example, out\_any[2] should indicate if either in[2] or in[1] are 1. Since in[0] has no neighbour to the right, the answer is obvious so we don't need to know out\_any[0].

out\_different: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbour to the left. For example, out\_different[98] should indicate if in[98] is different from in[99]. For this part, treat the vector as wrapping around, so in[99]'s neighbour to the left is in[0].

Ans;

module top\_module(

input [99:0] in,

output [98:0] out\_both,

output [99:1] out\_any,

output [99:0] out\_different );

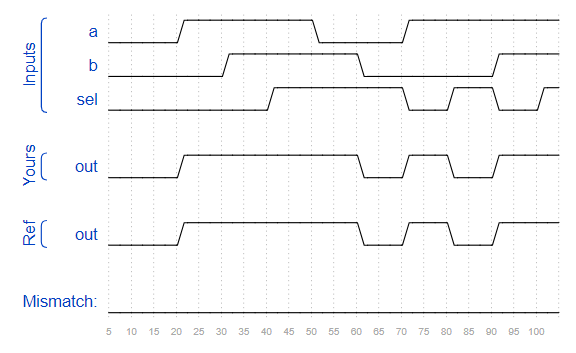
assign out\_both = in[98:0] & in[99:1];

assign out\_any = in[99:1] | in[98:0];

assign out\_different = in ^ {in[0],in[99:1]};

endmodule

62. Create a one-bit wide, 2-to-1 multiplexer. When sel=0, choose a. When sel=1, choose b.

Ans;

module top\_module(

input a, b, sel,

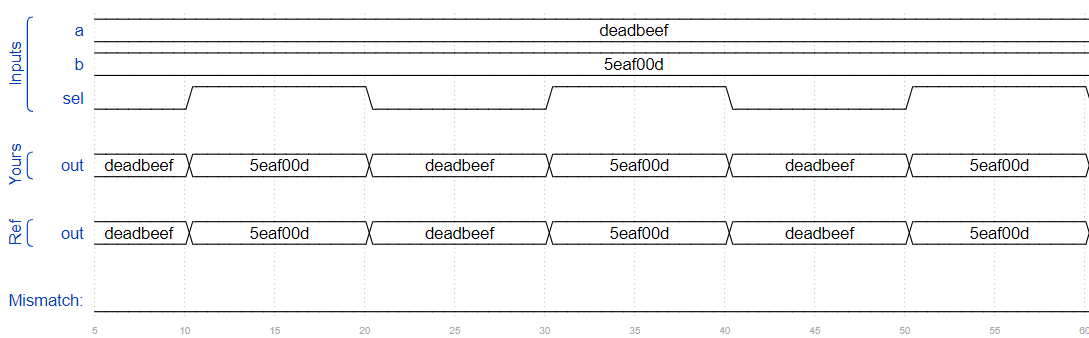
output out );

assign out = sel?b:a;

endmodule

63. Create a 100-bit wide, 2-to-1 multiplexer. When sel=0, choose a. When sel=1, choose b.

Ans;

module top\_module(

input [99:0] a, b,

input sel,

output [99:0] out );

assign out = sel?b:a;

endmodule

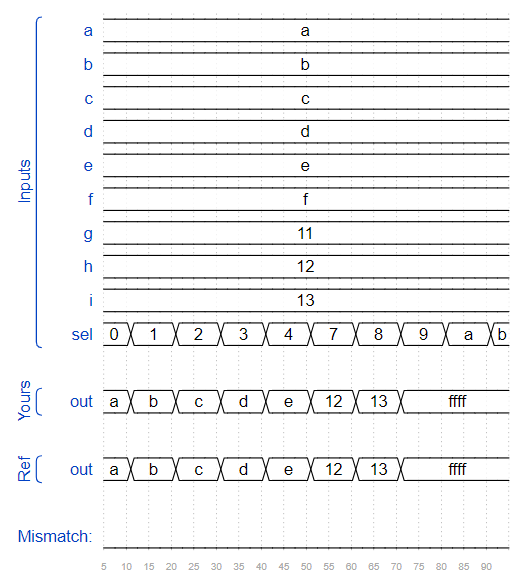
64. Create a 16-bit wide, 9-to-1 multiplexer. sel=0 chooses a, sel=1 chooses b, etc. For the unused cases (sel=9 to 15), set all output bits to '1'.

Ans;

module top\_module(

input [15:0] a, b, c, d, e, f, g, h, i,

input [3:0] sel,

 output [15:0] out );

always @ \* begin

case (sel)

4'b0000 : out = a;

4'b0001 : out = b;

4'b0010 : out = c;

4'b0011 : out = d;

4'b0100 : out = e;

4'b0101 : out = f;

4'b0110 : out = g;

4'b0111 : out = h;

4'b1000 : out = i;

default : out = {16{1'b1}};

endcase

end

endmodule

65. Create a 1-bit wide, 256-to-1 multiplexer. The 256 inputs are all packed into a single 256-bit input vector. sel=0 should select in[0], sel=1 selects bits in[1], sel=2 selects bits in[2], etc.

Ans;

module top\_module(

input [255:0] in,

input [7:0] sel,

output out

);

assign out = in[sel];

endmodule

66. Create a 4-bit wide, 256-to-1 multiplexer. The 256 4-bit inputs are all packed into a single 1024-bit input vector. sel=0 should select bits in[3:0], sel=1 selects bits in[7:4], sel=2 selects bits in[11:8], etc.

Ans;

module top\_module(

input [1023:0] in,

input [7:0] sel,

output [3:0] out );

integer range;

wire [7:0]con1;

always @ (\*) begin

range = sel;

out = in[sel\*4 +:4];

end

endmodule

67. Create a half adder. A half adder adds two bits (with no carry-in) and produces a sum and carry-out.

Ans;

module top\_module(

input a, b,

output cout, sum );

assign sum = a^b;

assign cout = a&b;

endmodule

68. Create a full adder. A full adder adds three bits (including carry-in) and produces a sum and carry-out.

Ans;

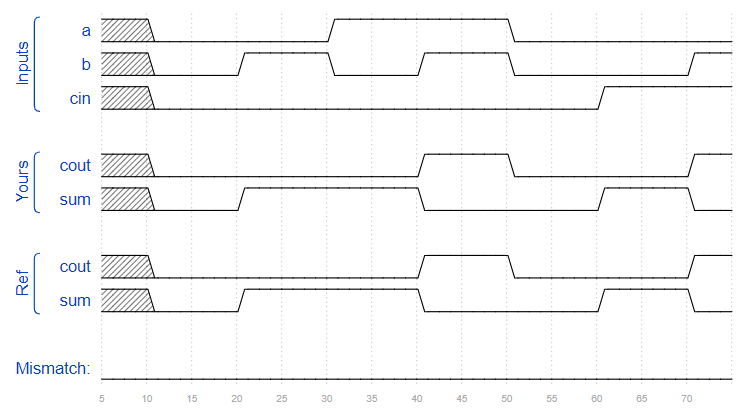
module top\_module(

input a, b, cin,

output cout, sum );

assign cout = a&b | b&cin | a&cin;

assign sum = a^b^cin;

endmodule

69. ripple carry adder 3-bit.

Ans;

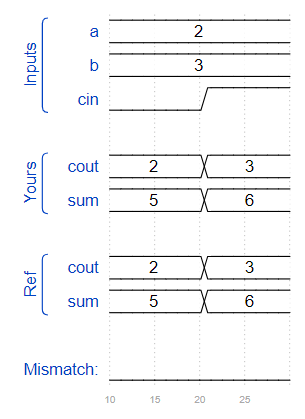
module top\_module(

input [2:0] a, b,

input cin,

output [2:0] cout,

output [2:0] sum );



FA FA1(a[0],b[0],cin,cout[0],sum[0]);

FA FA2(a[1],b[1],cout[0],cout[1],sum[1]);

FA FA3(a[2],b[2],cout[1],cout[2],sum[2]);

endmodule

module FA(

input a, b, cin,

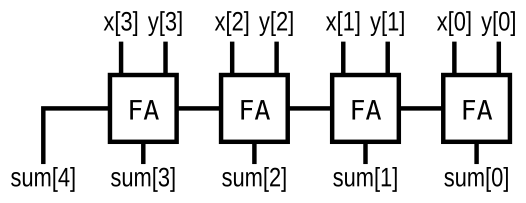
output cout, sum );

assign cout = a&b | b&cin | a&cin;

assign sum = a^b^cin;

endmodule

70. Implement the following circuit:



Ans;

module top\_module (

input [3:0] x,

input [3:0] y,

output [4:0] sum);

wire [3:0]cout;

FA FA1(x[0],y[0],0,cout[0],sum[0]);

FA FA2(x[1],y[1],cout[0],cout[1],sum[1]);

FA FA3(x[2],y[2],cout[1],cout[2],sum[2]);

FA FA4(x[3],y[3],cout[2],sum[4],sum[3]);

endmodule

module FA(

input a, b, cin,

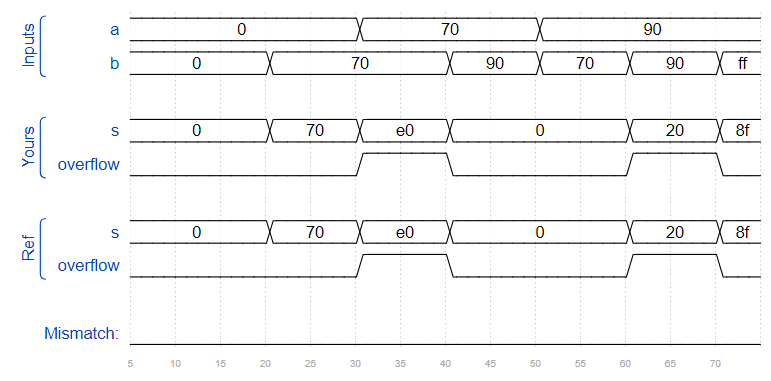
output cout, sum );

assign cout = a&b | b&cin | a&cin;

assign sum = a^b^cin;

endmodule

71. Assume that you have two 8-bit 2's complement numbers, a[7:0] and b[7:0]. These numbers are added to produce s[7:0]. Also compute whether a (signed) overflow has occurred.

Ans;

module top\_module (

input [7:0] a,

input [7:0] b,

output [7:0] s,

output overflow

);

assign s = a+b;

assign overflow = a[7]&&b[7]&&(~s[7]) || (~a[7])&&(~b[7])&&(s[7]);

endmodule

module top\_m(

input a, b, cin,

output cout, sum );

assign cout = a&b | b&cin | a&cin;

assign sum = a^b^cin;

endmodule

72. Create a 100-bit binary adder. The adder adds two 100-bit numbers and a carry-in to produce a 100-bit sum and carry out.

Ans;

module top\_module(

input [99:0] a, b,

input cin,

output cout,

output [99:0] sum );

genvar i;

wire [98:0]con\_vect;

one\_bit\_FA FA1(a[0],b[0],cin,con\_vect[0],sum[0]);

one\_bit\_FA FA2(a[99],b[99],con\_vect[98],cout,sum[99]);

generate

for (i=1; i<99; i=i+1) begin : Full\_adder\_block

one\_bit\_FA FA(a[i],b[i],con\_vect[i-1],con\_vect[i],sum[i]);

end

endgenerate

endmodule

module one\_bit\_FA(

input a,b,

input cin,

output cout,sum);

assign sum = a^b^cin;

assign cout = (a&b)|(b&cin)|(cin&a);

endmodule

73. You are provided with a BCD (binary-coded decimal) one-digit adder named bcd\_fadd that adds two BCD digits and carry-in, and produces a sum and carry-out.

module bcd\_fadd (

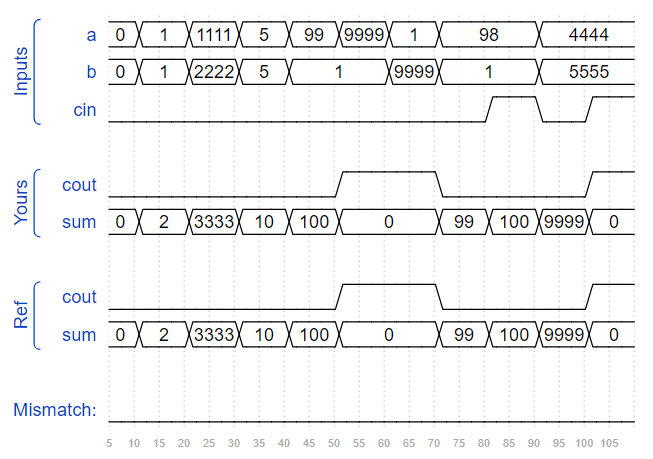
input [3:0] a,

input [3:0] b,

input cin,

output cout,

output [3:0] sum );

Instantiate 4 copies of bcd\_fadd to create a 4-digit BCD ripple-carry adder. Your adder should add two 4-digit BCD numbers (packed into 16-bit vectors) and a carry-in to produce a 4-digit sum and carry out.

Ans;

module top\_module(

input [15:0] a, b,

input cin,

output cout,

output [15:0] sum );

wire [2:0]con1;

bcd\_fadd B\_add1(a[3:0],b[3:0],cin,con1[0],sum[3:0]);

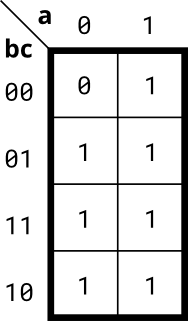
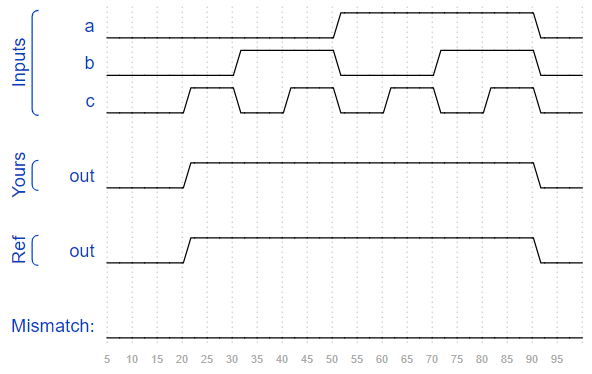
bcd\_fadd B\_add2(a[7:4],b[7:4],con1[0],con1[1],sum[7:4]);

bcd\_fadd B\_add3(a[11:8],b[11:8],con1[1],con1[2],sum[11:8]);

bcd\_fadd B\_add4(a[15:12],b[15:12],con1[2],cout,sum[15:12]);

endmodule

74. Implement the circuit described by the Karnaugh map below.



Ans;

module top\_module(

input a,

input b,

input c,

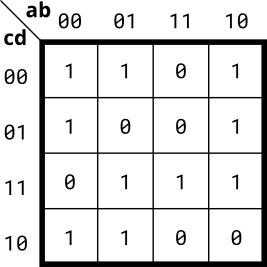
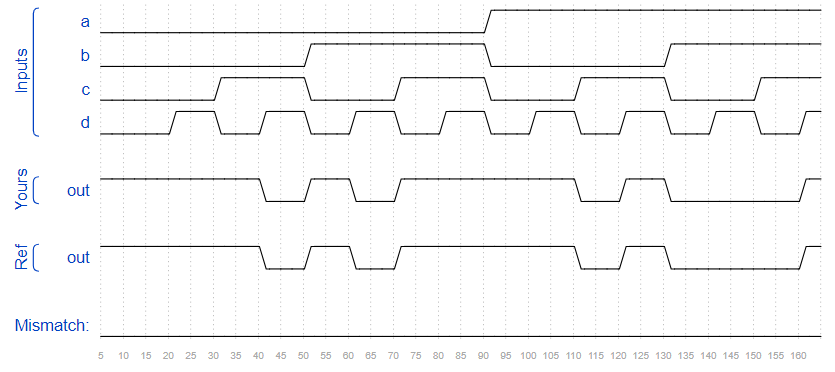
output out

);

assign out = (a | b | c);

endmodule

75. Implement the circuit described by the Karnaugh map below.



Ans;

module top\_module(

input a,

input b,

input c,

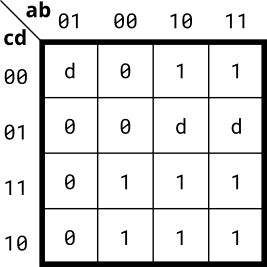
input d,

output out );

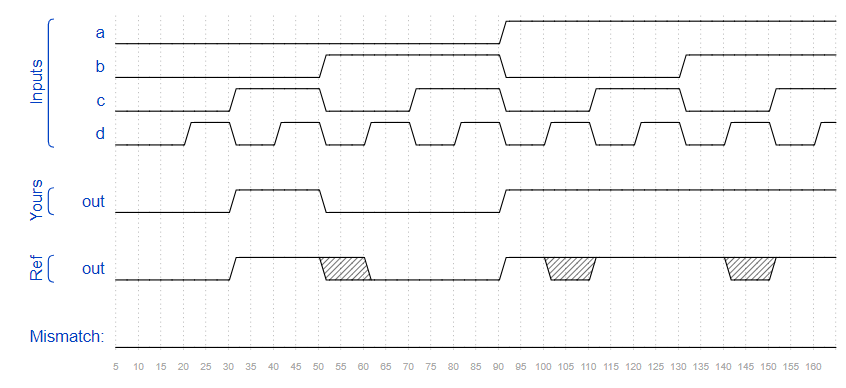
assign out = (c|!d|!b) & (!a|!b|c) & (a|b|!c|!d) & (!a|!c|d);

endmodule

76. Implement the circuit described by the Karnaugh map below.



Ans;

module top\_module(

input a,

input b,

input c,

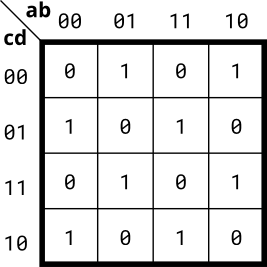
input d,

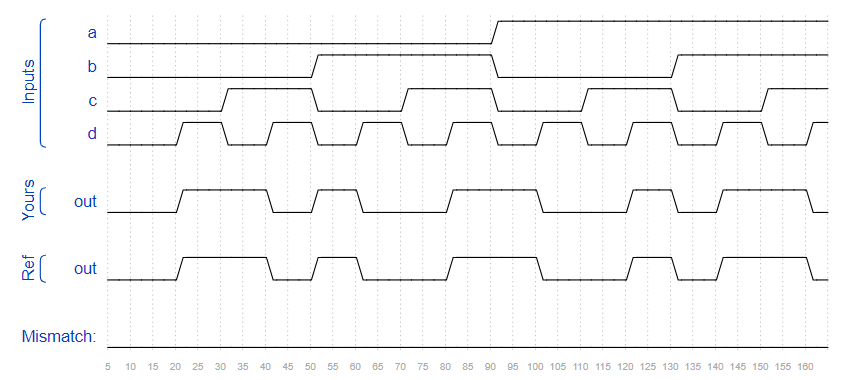
output out );

assign out = a|(~a&~b&c);

endmodule

77. Implement the circuit described by the Karnaugh map below.



Ans; module top\_module(

input a,

input b,

input c,

input d,

output out );

assign out = a^b^c^d;

endmodule

78. A single-output digital system with four inputs (a,b,c,d) generates a logic-1 when 2, 7, or 15 appears on the inputs, and a logic-0 when 0, 1, 4, 5, 6, 9, 10, 13, or 14 appears. The input conditions for the numbers 3, 8, 11, and 12 never occur in this system. For example, 7 corresponds to a,b,c,d being set to 0,1,1,1, respectively.

Determine the output out\_sop in minimum SOP form, and the output out\_pos in minimum POS form.

Ans;

module top\_module (

input a,

input b,

input c,

input d,

output out\_sop,

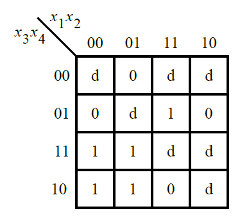
output out\_pos

);

assign out\_sop = (c&d)|(~a&~b&c);

assign out\_pos = c&(~b|~c|d)&(~a|~c|d);

endmodule

79. Consider the function f shown in the Karnaugh map below.

Ans;

module top\_module (

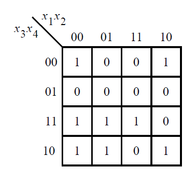
input [4:1] x,

output f );

assign f = (x[2]&x[4])|(x[3]&x[4])|(x[3]&~x[1]);

endmodule

80. Consider the function f shown in the Karnaugh map below. Implement this function.



Ans;

module top\_module (

input [4:1] x,

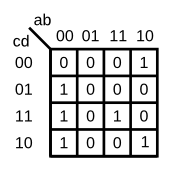
output f

);

assign f = (~x[1]&x[3])|(~x[2]&~x[4])|(x[2]&x[3]&x[4]);

endmodule

81. For the following Karnaugh map, give the circuit implementation using one 4-to-1 multiplexer and as many 2-to-1 multiplexers as required, but using as few as possible. You are not allowed to use any other logic gate and you must use a and b as the multiplexer selector inputs, as shown on the 4-to-1 multiplexer below.

You are implementing just the portion labelled top\_module, such that the entire circuit (including the 4-to-1 mux) implements the K-map.

Ans;

module top\_module (

input c,

input d,

output [3:0] mux\_in

);

assign mux\_in[0] = c | d;

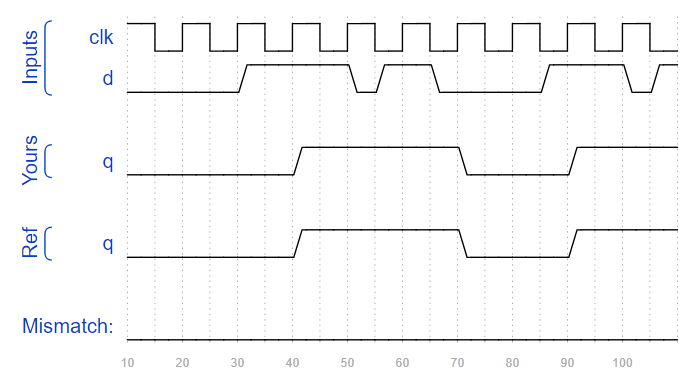
assign mux\_in[1] = 0;

assign mux\_in[2] = ~d;

assign mux\_in[3] = c & d;

endmodule

82. Create a single D flip-flop.



Ans;

module top\_module (

input clk,

input d,

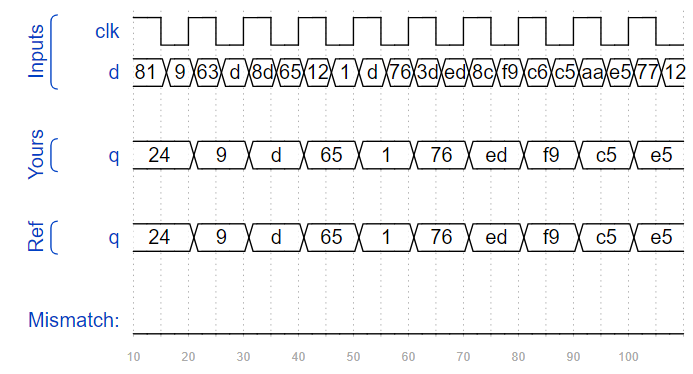
output reg q );

always @ (posedge clk) begin

q <= d;

end

endmodule

83. Create 8 D flip-flops. All DFFs should be triggered by the positive edge of clk.

Ans;

module top\_module (

input clk,

input [7:0] d,

output [7:0] q

);

always @ (posedge clk) begin

q <= d;

end

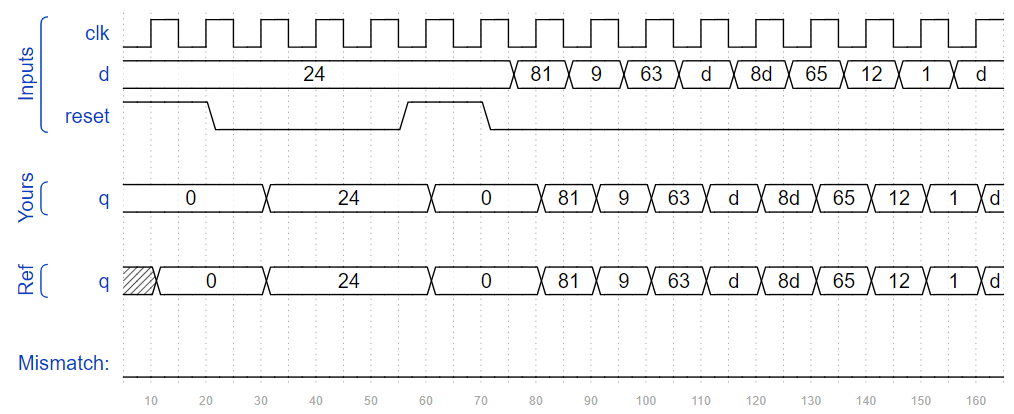
endmodule

84. Create 8 D flip-flops with active high synchronous reset. All DFFs should be triggered by the positive edge of clk.

Ans;

module top\_module (

input clk,

 input reset,

input [7:0] d,

output [7:0] q

);

always @ (posedge clk) begin

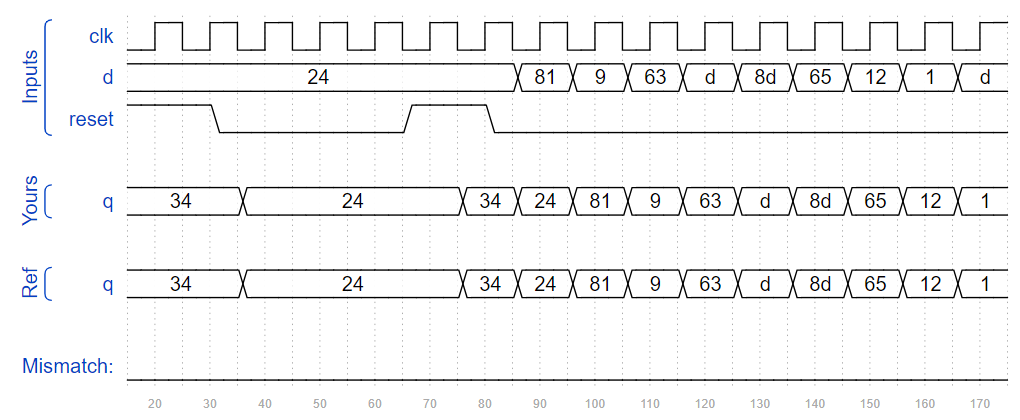
if (reset) q<=8'b0;

else q <= d;

end

endmodule

85. Create 8 D flip-flops with active high synchronous reset. The flip-flops must be reset to 0x34 rather than zero. All DFFs should be triggered by the negative edge of clk.

Ans;

module top\_module (

input clk,

input reset, input [7:0] d,

output [7:0] q

);

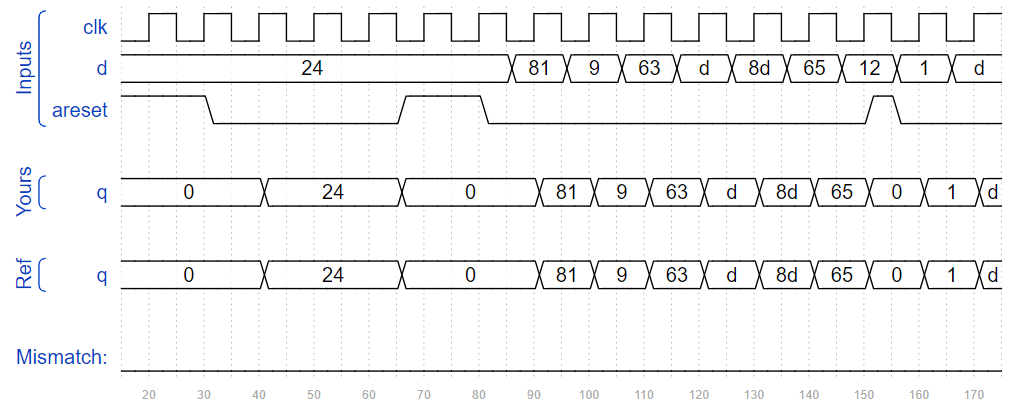
always @ (negedge clk) begin

if (reset==1'b1) q<=8'h34;

else q <= d;

end

endmodule

86. Create 8 D flip-flops with active high asynchronous reset. All DFFs should be triggered by the positive edge of clk.

Ans;

module top\_module (

input clk,

input areset,

input [7:0] d,

output [7:0] q);

always @ (posedge clk or posedge areset ) begin

if (areset==1'b1) q<= 1'b0;

else q <= d;

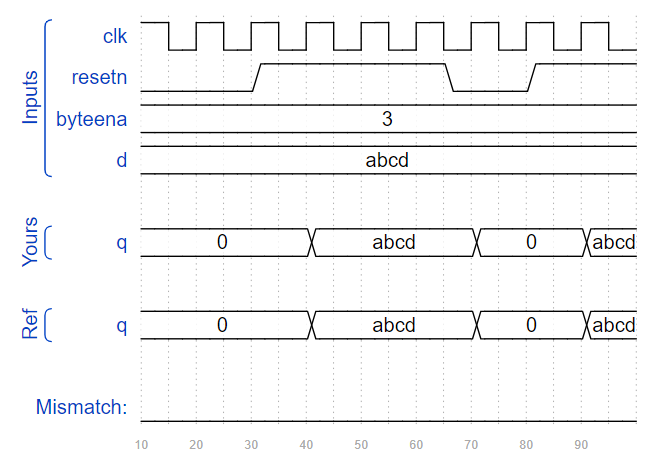
end

endmodule

87. Create 16 D flip-flops. It's sometimes useful to only modify parts of a group of flip-flops. The byte-enable inputs control whether each byte of the 16 registers should be written to on that cycle. byteena[1] controls the upper byte d[15:8], while byteena[0] controls the lower byte d[7:0].

resetn is a synchronous, active-low reset.

All DFFs should be triggered by the positive edge of clk.

Ans;

module top\_module (

input clk,

input resetn,

input [1:0] byteena,

input [15:0] d,

output [15:0] q

);

always @ (posedge clk) begin

if (resetn==1'b0) q<= 1'b0;

else begin

case(byteena)

2'b00: q <= q;

2'b01: q[7:0] <= d[7:0];

2'b10: q[15:8] <= d[15:8];

2'b11: q <= d;

endcase

end

end endmodule

88. Implement the following circuit:



Ans;

module top\_module (

input d,

input ena,

output q);

always @ (\*) begin

if (ena) q<=d;

end

endmodule

89. Implement the following circuit:



Ans;

module top\_module (

input clk,

input d,

input ar,

output q);

always @ (posedge clk, posedge ar) begin

if (ar) q<=1'b0;

else q<=d;

end

endmodule

90. Implement the following circuit:



Ans;

module top\_module (

input clk,

input d,

input r,

output q);

always @ (posedge clk) begin

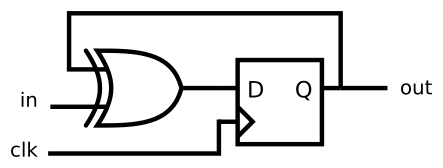
if (r) q <= 1'b0;

else q <= d;

end

endmodule

91. Implement the following circuit:



Ans;

module top\_module (

input clk,

input in,

output out);

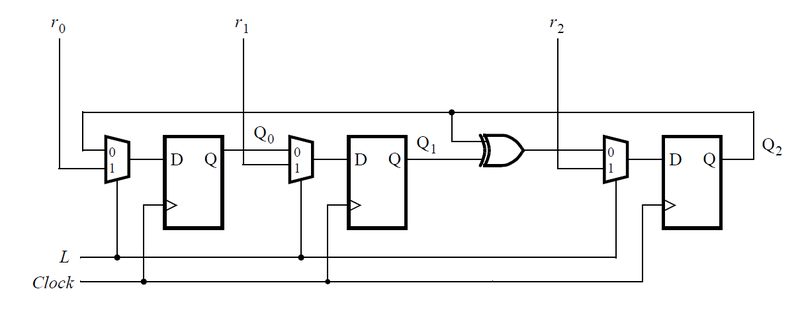
always @ (posedge clk) begin

out <= out ^ in;

end

endmodule

92. Consider the sequential circuit below:



Assume that you want to implement hierarchical Verilog code for this circuit, using three instantiations of a submodule that has a flip-flop and multiplexer in it. Write a Verilog module (containing one flip-flop and multiplexer) named top\_module for this submodule.

Ans;

module top\_module (

input clk,

input L,

input r\_in,

input q\_in,

output reg Q);

always @ (posedge clk) begin

case (L)

1'b0 : Q <= q\_in;

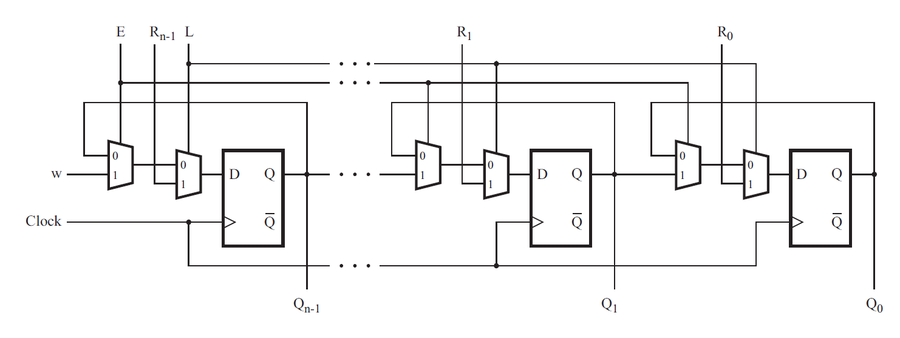
1'b1 : Q <= r\_in;

endcase

end

endmodule

93. Consider the n-bit shift register circuit shown below:



Ans;

module top\_module (

input clk,

input w, R, E, L,

output Q

);

wire [1:0] con;

assign con = {E,L};

always @ (posedge clk) begin

case (con)

2'b00 : Q <= Q;

2'b01 : Q <= R;

2'b10 : Q <= w;

2'b11 : Q <= R;

endcase

end

endmodule

94. Given the finite state machine circuit as shown, assume that the D flip-flops are initially reset to zero before the machine begins.

Build this circuit.

Ans;

module top\_module (

 input clk,

input x,

output z

);

reg [2:0] Q;

always@(posedge clk)begin

Q[0] <= Q[0] ^ x;

Q[1] <= ~Q[1] & x;

Q[2] <= ~Q[2] | x;

end

assign z = ~(| Q);

endmodule

95. A JK flip-flop has the below truth table. Implement a JK flip-flop with only a D-type flip-flop and gates. Note: Qold is the output of the D flip-flop before the positive clock edge. 

Ans;

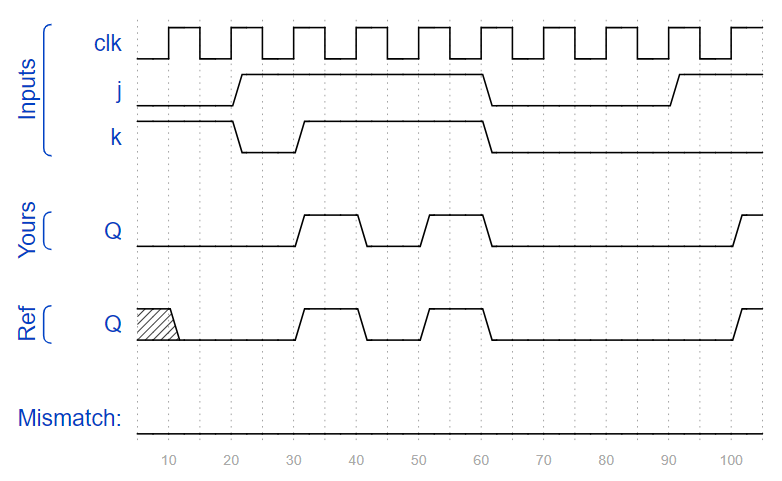
module top\_module (

input clk,

input j,

input k,

output Q);

 always @(posedge clk) begin

case({j, k})

2'b00 : Q <= Q;

2'b01 : Q <= 0;

2'b10 : Q <= 1;

2'b11 : Q <= ~Q;

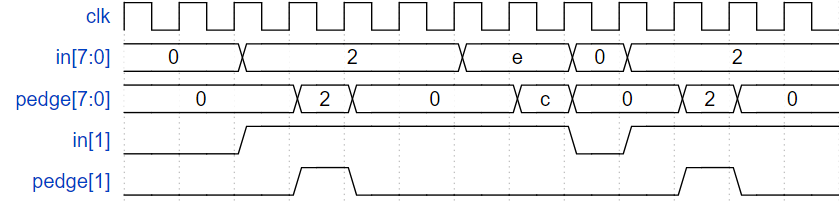
endcase

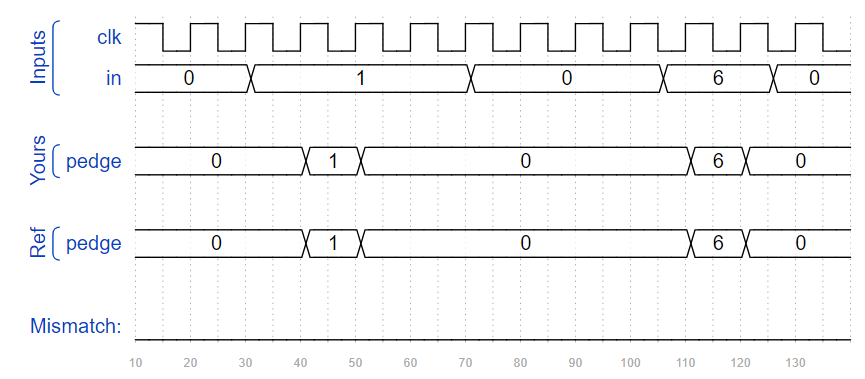
end

endmodule

96. For each bit in an 8-bit vector, detect when the input signal changes from 0 in one clock cycle to 1 the next (similar to positive edge detection). The output bit should be set the cycle after a 0 to 1 transition occurs.

Here are some examples. For clarity, in[1] and pedge[1] are shown separately.



Ans;

module top\_module (

input clk,

input [7:0] in,

output [7:0] pedge

);

reg [7:0]intermediate;

always @ (posedge clk) begin

intermediate <= in;

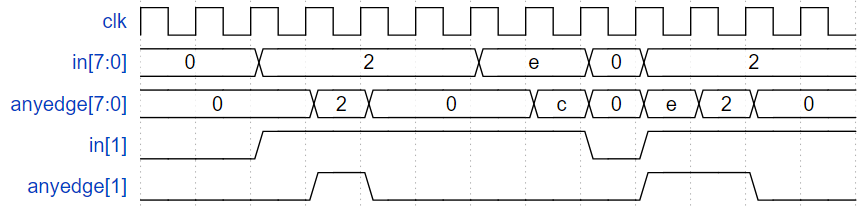
pedge <= (~intermediate) & in;

end

endmodule

97. For each bit in an 8-bit vector, detect when the input signal changes from one clock cycle to the next (detect any edge). The output bit should be set the cycle after a 0 to 1 transition occurs.

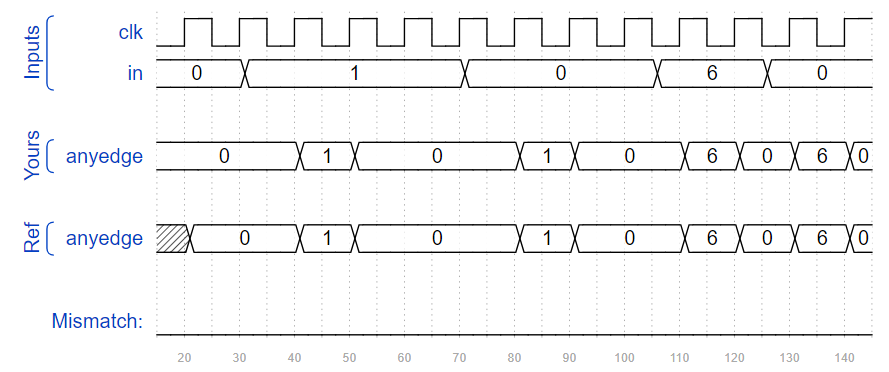
Here are some examples. For clarity, in[1] and anyedge[1] are shown separately



Ans;

module top\_module (

input clk,

 input [7:0] in,

output [7:0] anyedge

);

reg [7:0]intermediate;

always @ (posedge clk) begin

intermediate <= in;

anyedge <= intermediate ^ in;

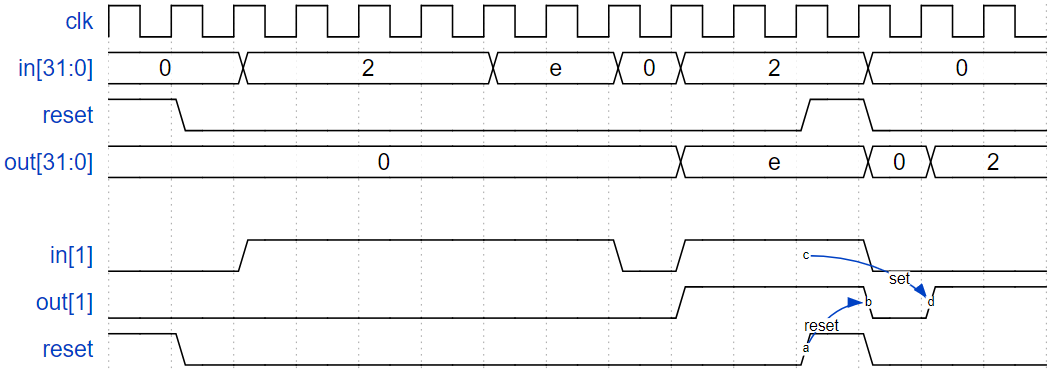
end

endmodule

98. For each bit in a 32-bit vector, capture when the input signal changes from 1 in one clock cycle to 0 the next. "Capture" means that the output will remain 1 until the register is reset (synchronous reset).

Each output bit behaves like a SR flip-flop: The output bit should be set (to 1) the cycle after a 1 to 0 transition occurs. The output bit should be reset (to 0) at the positive clock edge when reset is high. If both of the above events occur at the same time, reset has precedence. In the last 4 cycles of the example waveform below, the 'reset' event occurs one cycle earlier than the 'set' event, so there is no conflict here.

In the example waveform below, reset, in[1] and out[1] are shown again separately for clarity.



Ans;

module top\_module (

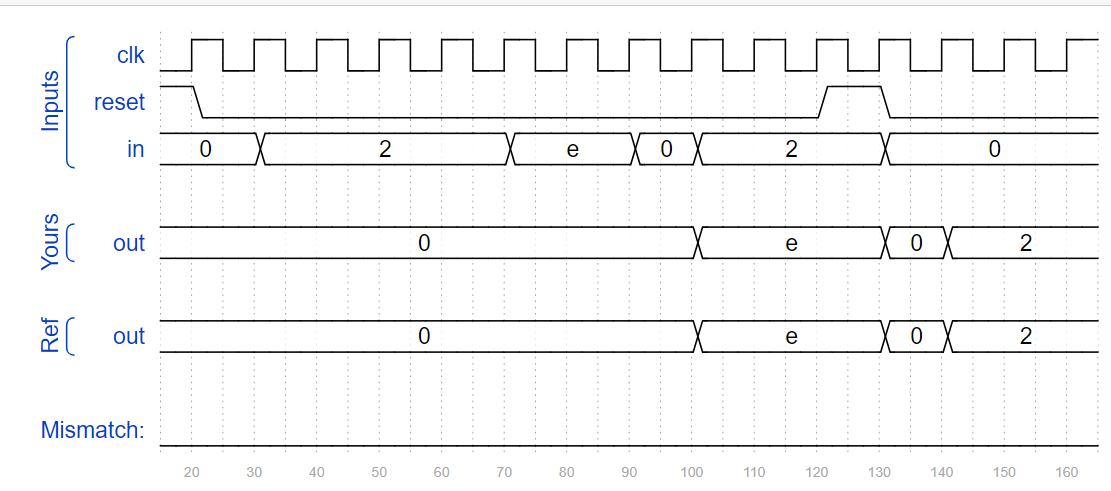
input clk,

input reset,

input [31:0] in,

output [31:0] out

);

 reg [31:0] in\_last; //the input in last clk

always @(posedge clk) begin

in\_last <= in;

if(reset) out <= 0;

else begin

out <= out | (in\_last & ~in);

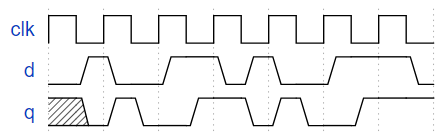
end

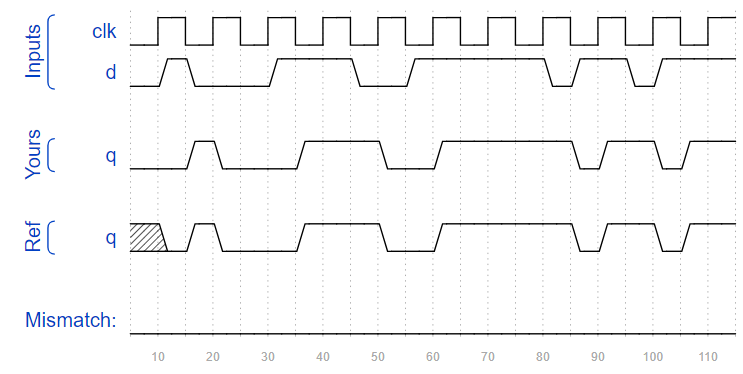
end

endmodule



99. Build a circuit that functionally behaves like a dual-edge triggered flip-flop:



Ans;

module top\_module (

input clk,

input d,

output q

);

reg q1,q2;

always @ (posedge clk) begin

q1 <= q2^d;

end

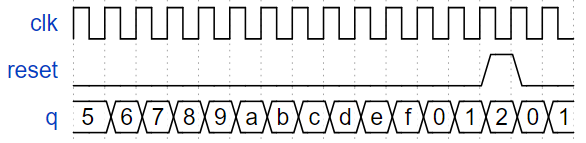
always @ (negedge clk) begin

q2 <= q1^d;

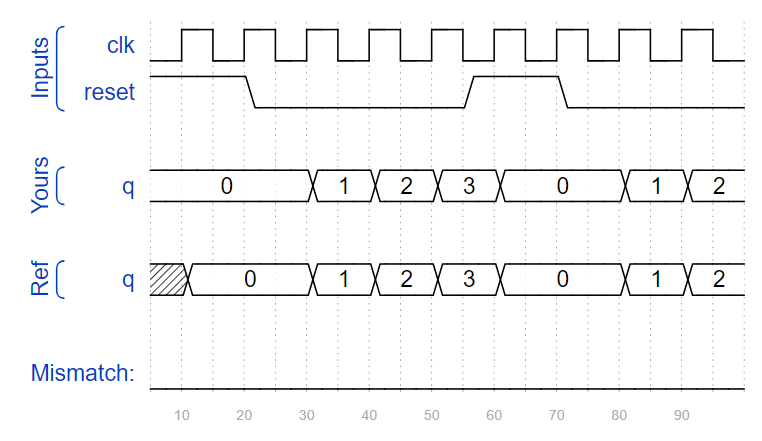
end

assign q = q1^q2;

endmodule

100. Build a 4-bit binary counter that counts from 0 through 15, inclusive, with a period of 16. The reset input is synchronous, and should reset the counter to 0. 

Ans;

module top\_module (

input clk,

input reset,

output [3:0] q);

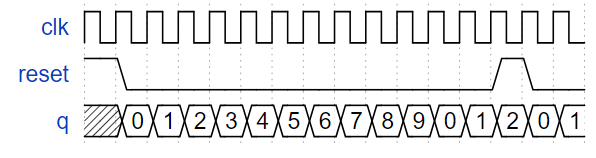
always @ (posedge clk) begin

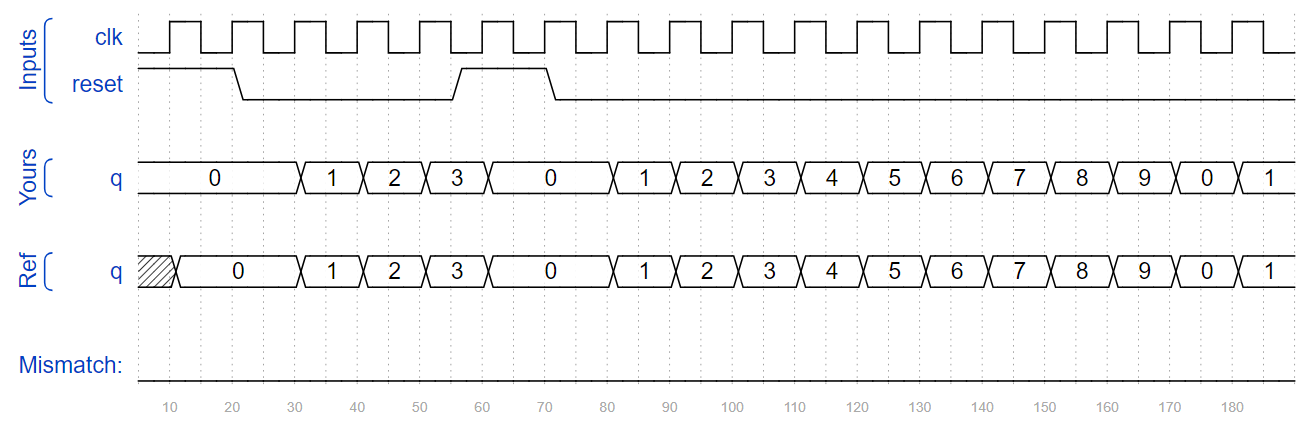
if (reset) q <= 4'd0;

else q <= q+4'd1;

end

endmodule

101. Build a decade counter that counts from 0 through 9, inclusive, with a period of 10. The reset input is synchronous, and should reset the counter to 0. 

Ans;

module top\_module (

input clk,

input reset,

output [3:0] q);

always @ (posedge clk) begin

if (reset) q <= 4'd0;

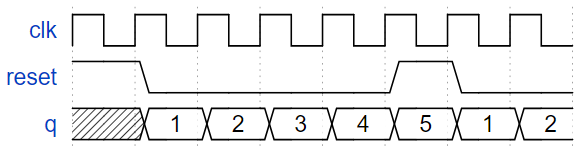
else if (q == 4'd9) q<= 4'd0;

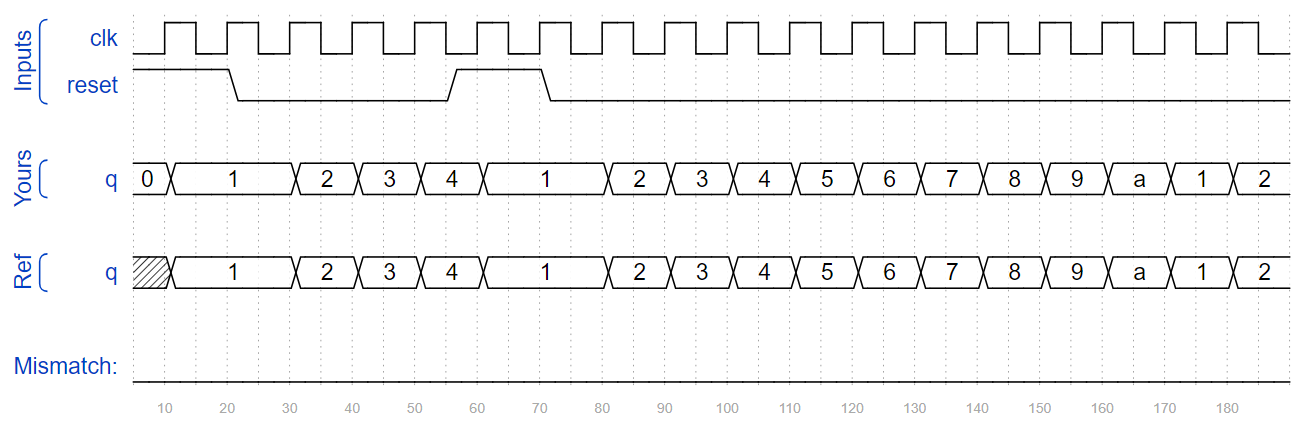
else q <= q+4'd1;

end

endmodule

102. Make a decade counter that counts 1 through 10, inclusive. The reset input is synchronous, and should reset the counter to 1.



Ans;

module top\_module (

input clk,

input reset,

output [3:0] q);

always @ (posedge clk) begin

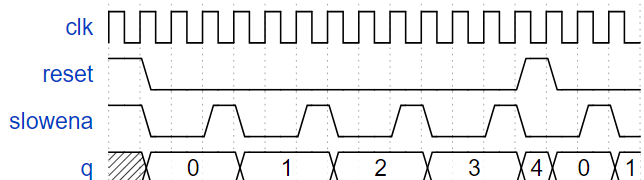
if (reset) q <= 4'd1;

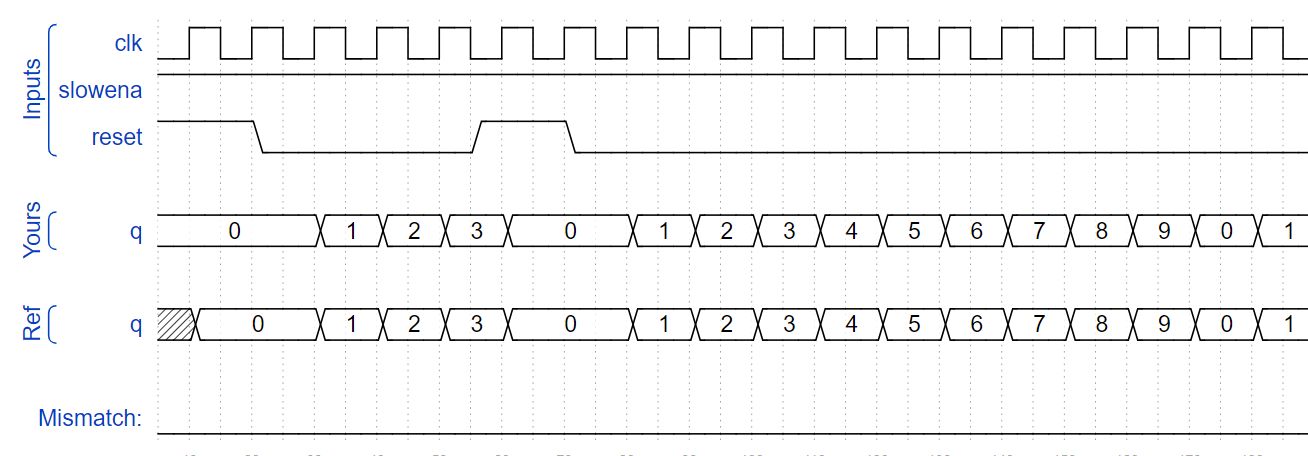
else if (q == 4'd10) q<= 4'd1;

else q <= q+4'd1;

end

endmodule

103. Build a decade counter that counts from 0 through 9, inclusive, with a period of 10. The reset input is synchronous, and should reset the counter to 0. We want to be able to pause the counter rather than always incrementing every clock cycle, so the slowena input indicates when the counter should increment. 

Ans;

module top\_module (

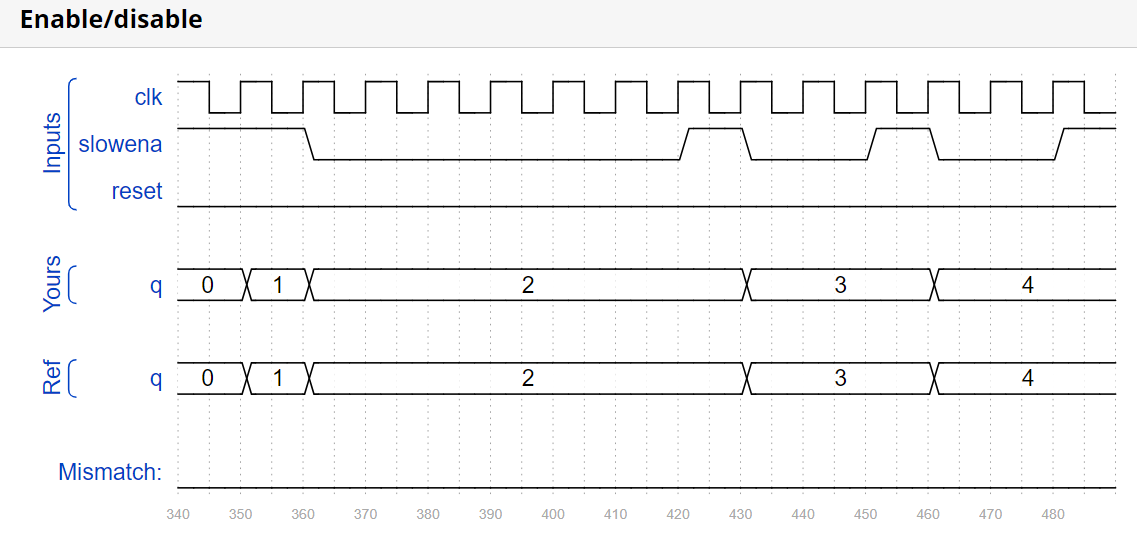
input clk,

input slowena,

input reset,

 output [3:0] q);

always @ (posedge clk) begin

 if (reset) q <= 4'd0;

else if (slowena) begin

if (q == 4'd9) q<= 4'd0;

else q <= q+4'd1;

end

end

endmodule

104. Design a 1-12 counter with the following inputs and outputs:

Reset Synchronous active-high reset that forces the counter to 1

Enable Set high for the counter to run

Clk Positive edge-triggered clock input

Q[3:0] The output of the counter

c\_enable, c\_load, c\_d[3:0] Control signals going to the provided 4-bit counter, so correct operation can be verified.

You have the following components available:

the 4-bit binary counter (count4) below, which has Enable and synchronous parallel-load inputs (load has higher priority than enable). The count4 module is provided to you. Instantiate it in your circuit.

logic gates

module count4(

input clk,

input enable,

input load,

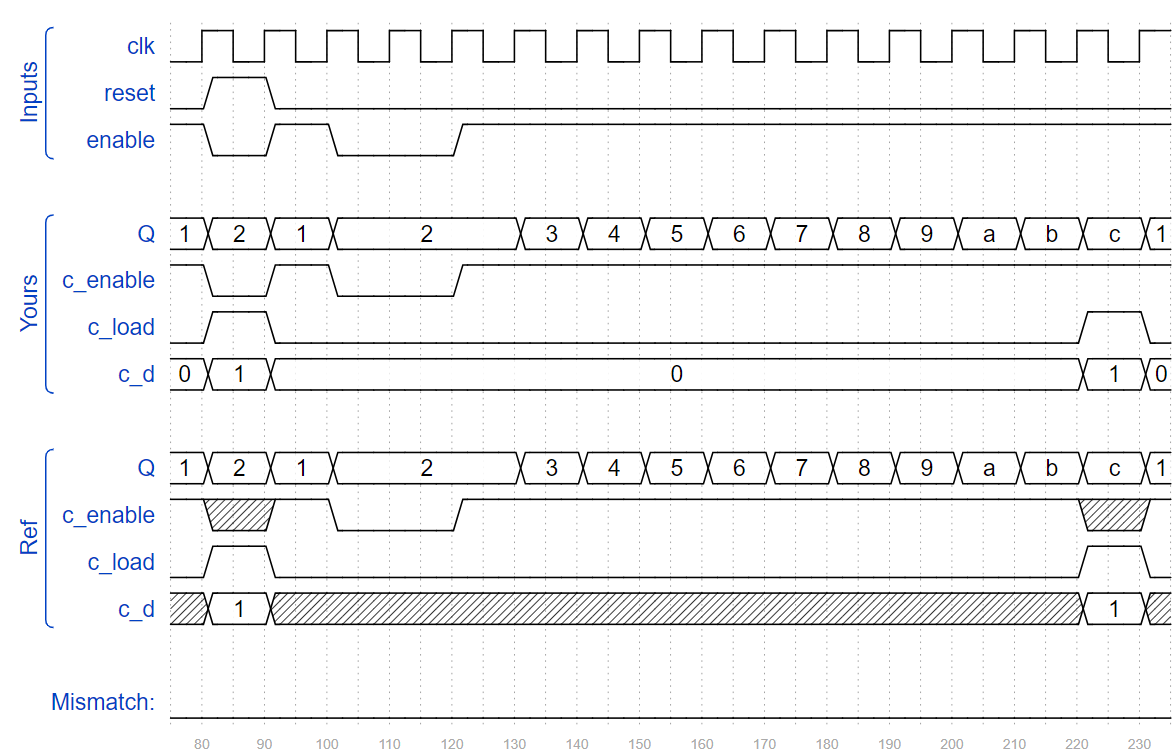
input [3:0] d,

output reg [3:0] Q

);

The c\_enable, c\_load, and c\_d outputs are the signals that go to the internal counter's enable, load, and d inputs, respectively. Their purpose is to allow these signals to be checked for correctness.

Ans;

module top\_module (

input clk,

input reset,

input enable,

output [3:0] Q,

output c\_enable,

output c\_load,

output [3:0] c\_d

);

initial Q <= 1;

always @(posedge clk) begin

if(reset | ((Q == 12) & enable)) Q <= 1;

else Q <= (enable) ? Q + 1 : Q;

end

assign c\_enable = enable;

assign c\_load = (reset | ((Q == 12) & enable));

assign c\_d = c\_load ? 1 : 0;

count4 the\_counter (clk, c\_enable, c\_load, c\_d /\*, ... \*/ );

endmodule

105. From a 1000 Hz clock, derive a 1 Hz signal, called OneHertz, that could be used to drive an Enable signal for a set of hour/minute/second counters to create a digital wall clock. Since we want the clock to count once per second, the OneHertz signal must be asserted for exactly one cycle each second. Build the frequency divider using modulo-10 (BCD) counters and as few other gates as possible. Also output the enable signals from each of the BCD counters you use (c\_enable[0] for the fastest counter, c\_enable[2] for the slowest).

The following BCD counter is provided for you. Enable must be high for the counter to run. Reset is synchronous and set high to force the counter to zero. All counters in your circuit must directly use the same 1000 Hz signal.

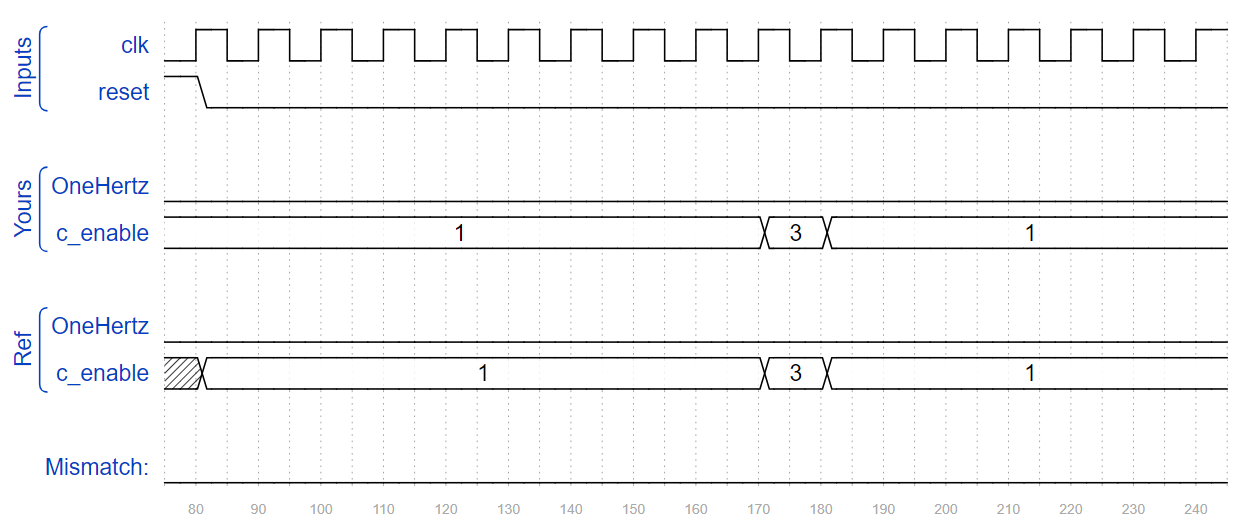
module bcdcount (

input clk,

input reset,

input enable,

output reg [3:0] Q);

ans;

module top\_module (

input clk,

input reset,

output OneHertz,

 output [2:0] c\_enable

);

wire [3:0] q0, q1, q2;

bcdcount counter0 (clk, reset, c\_enable[0], q0);

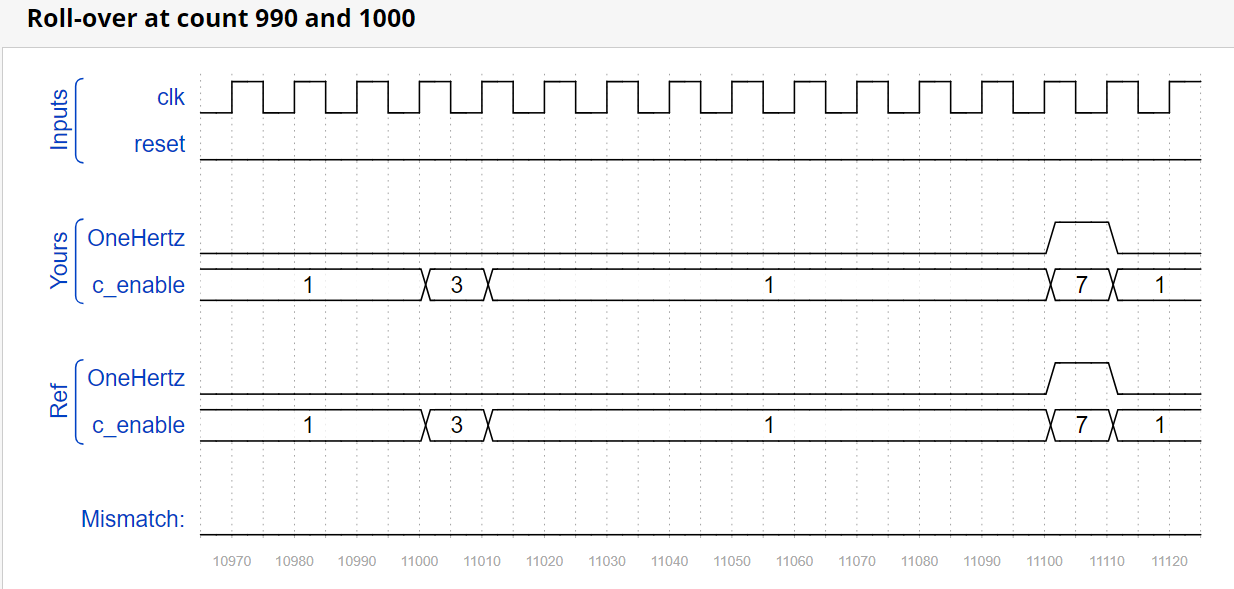
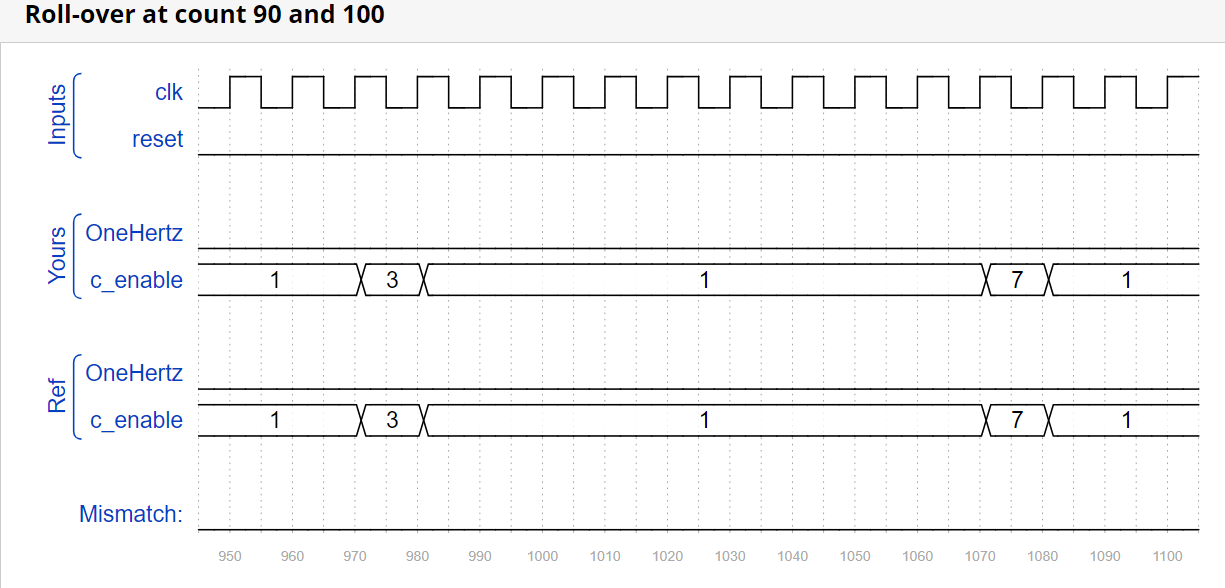
bcdcount counter1 (clk, reset, c\_enable[1], q1);

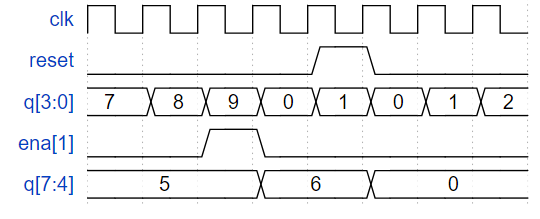
bcdcount counter2 (clk, reset, c\_enable[2], q2);

assign c\_enable = {(q1 == 4'd9) & (q0 == 4'd9), q0 == 4'd9, 1'b1};

assign OneHertz = (q2 == 4'd9) & (q1 == 4'd9) & (q0 == 4'd9);

endmodule



106. Build a 4-digit BCD (binary-coded decimal) counter. Each decimal digit is encoded using 4 bits: q[3:0] is the ones digit, q[7:4] is the tens digit, etc. For digits [3:1], also output an enable signal indicating when each of the upper three digits should be incremented. 

Ans;

module top\_module (

input clk,

input reset, // Synchronous active-high reset

output [3:1] ena,

output [15:0] q);

assign ena = {q[11:8] == 4'd9 && q[7:4] == 4'd9 && q[3:0] == 4'd9, q[7:4] == 4'd9 && q[3:0] == 4'd9, q[3:0] == 4'd9};

count4 inst1\_count4

(

.clk(clk),

.reset(reset),

.ena(1),

.q(q[3:0])

);

count4 inst2\_count4

(

.clk(clk),

.reset(reset),

.ena(ena[1]),

.q(q[7:4])

);

count4 inst3\_count4

(

.clk(clk),

.reset(reset),

.ena(ena[2]),

.q(q[11:8])

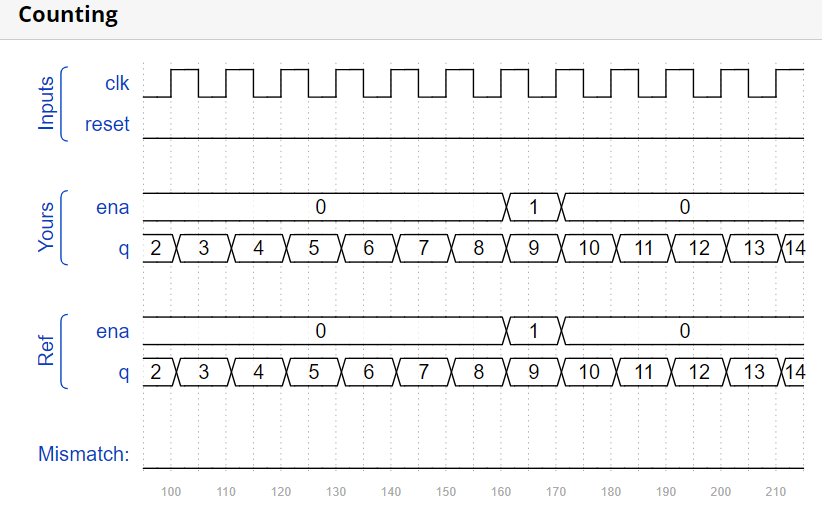
);

count4 inst4\_count4

(

.clk(clk),

.reset(reset),

 .ena(ena[3]),

.q(q[15:12])

);

endmodule

module count4

(

input clk,

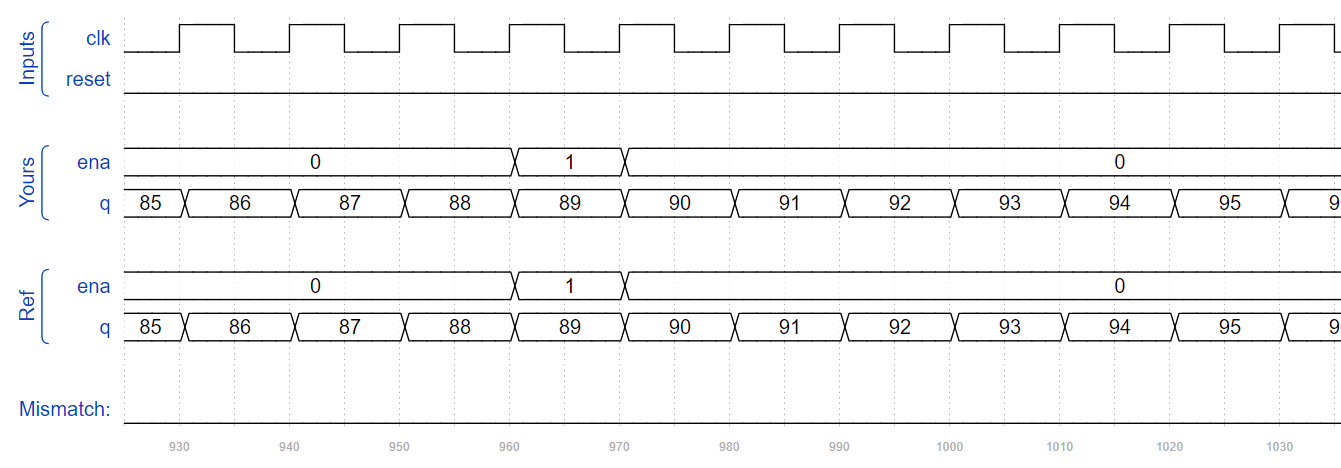
input reset,

input ena,

output reg[3:0] q

);

always @(posedge clk) begin

 if(reset) q <= 4'd0;

else begin

if(ena) begin

if(q == 4'd9) q <= 4'd0;

else q <= q + 1;

end

end

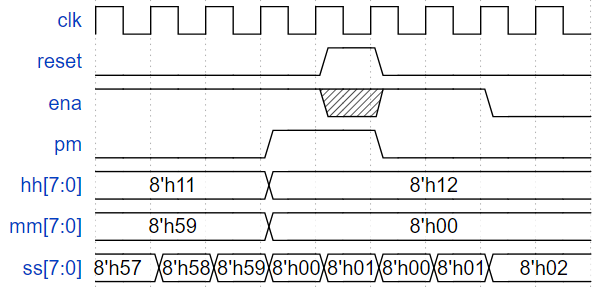
end

endmodule

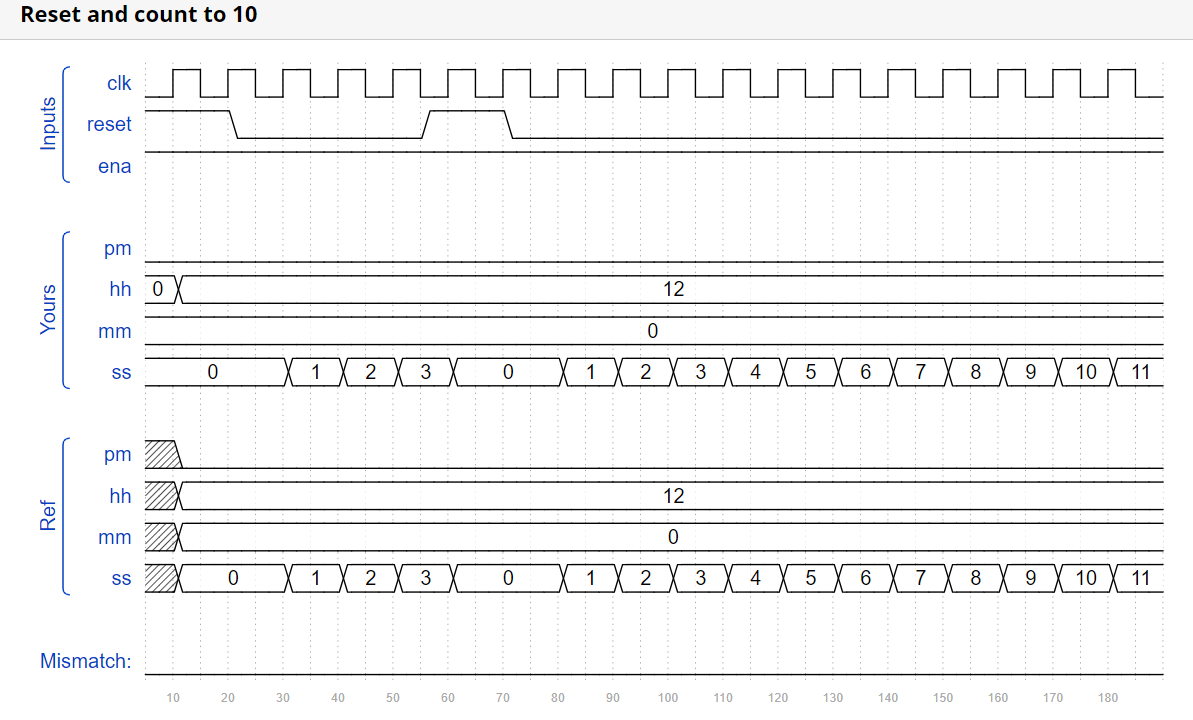
107. Create a set of counters suitable for use as a 12-hour clock (with am/pm indicator). Your counters are clocked by a fast-running clk, with a pulse on ena whenever your clock should increment (i.e., once per second).

reset resets the clock to 12:00 AM. pm is 0 for AM and 1 for PM. hh, mm, and ss are two BCD (Binary-Coded Decimal) digits each for hours (01-12), minutes (00-59), and seconds (00-59). Reset has higher priority than enable, and can occur even when not enabled.

The following timing diagram shows the rollover behaviour from 11:59:59 AM to 12:00:00 PM and the synchronous reset and enable behaviour.



Ans;

module top\_module(

input clk,

input reset,

input ena,

output pm,

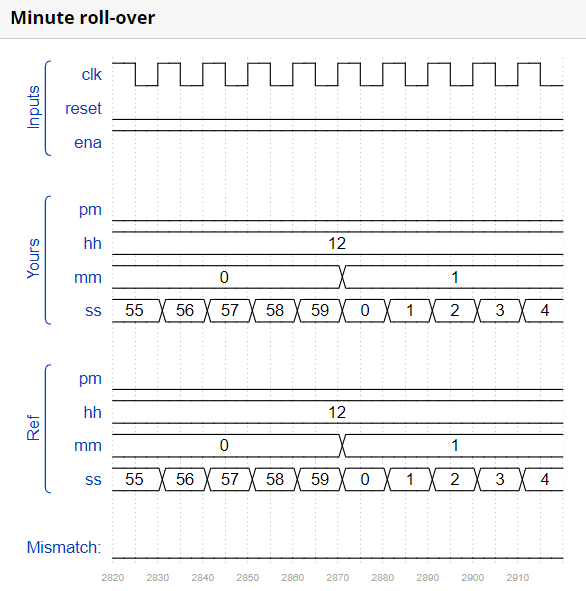
output [7:0] hh,

output [7:0] mm,

output [7:0] ss);

reg [2:0] ena\_hms; //determine when will "ss","mm" and "hh" need to be increased

assign ena\_hms = {(ena && (mm == 8'h59) && (ss == 8'h59)), (ena && (ss == 8'h59)), ena};



count60 count\_ss(

.clk(clk),

.reset(reset),

.ena(ena\_hms[0]),

.q(ss)

);

count60 count\_mm(

.clk(clk),

.reset(reset),

.ena(ena\_hms[1]),

 .q(mm)

);

always @(posedge clk) begin

if(reset) begin

hh <= 8'h12; //hh=12

pm <= 0;

end

else begin

if(ena\_hms[2]) begin //if mm=59 and ss=59

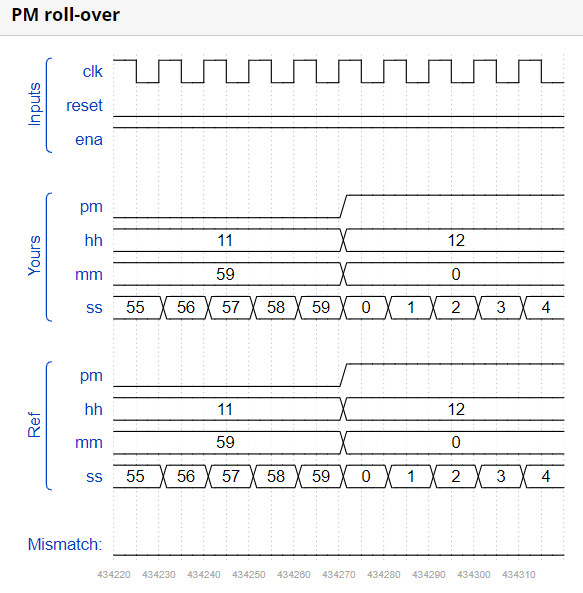
if(hh == 8'h12) hh <= 8'h1; //hh will change:12AM->1AM or 12PM->1PM

else if(hh == 8'h11) begin //if hh=11, then PM->AM or AM->PM

hh[3:0] <= hh[3:0] + 1'h1; //hh=12

pm <= ~pm;

end

 else begin

if(hh[3:0] == 4'h9) begin

hh[3:0] <= 4'h0;

hh[7:4] <= hh[7:4] + 1'h1;

end

else hh[3:0] = hh[3:0] + 1'h1;

end

end

else hh <= hh;

end

end

endmodule

module count60(

input clk,

input reset,

input ena,

output [7:0] q

);

always @(posedge clk) begin

if(reset) q <= 8'h0;

else begin

if(ena) begin

if(q[3:0] == 4'h9) begin

if(q[7:4] == 4'h5) q <= 8'h0;

else begin

q[7:4] <= q[7:4] + 1'h1;

q[3:0] <= 4'h0;

end

end

else q[3:0] <= q[3:0] + 1'h1;

end

else q <= q;

end

end

endmodule

108. Build a 4-bit shift register (right shift), with asynchronous reset, synchronous load, and enable.

areset: Resets shift register to zero.

load: Loads shift register with data[3:0] instead of shifting.

ena: Shift right (q[3] becomes zero, q[0] is shifted out and disappears).

q: The contents of the shift register.

If both the load and ena inputs are asserted (1), the load input has higher priority.

Ans;

module top\_module(

input clk,

input areset,

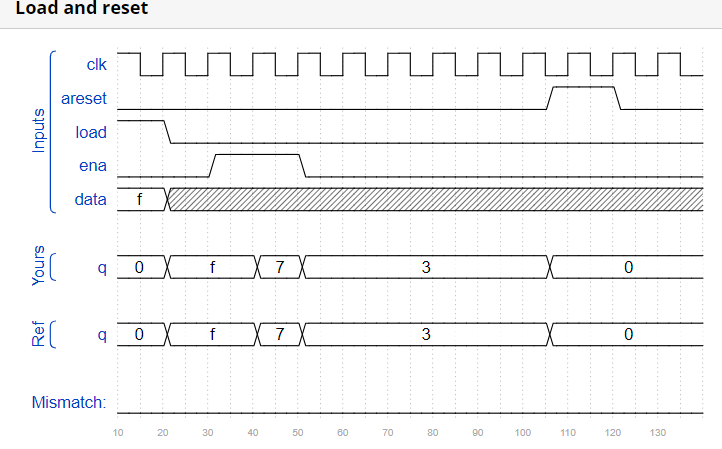
input load,

input ena,

input [3:0] data,

output reg [3:0] q);

always @(posedge clk or posedge areset) begin

 if(areset) begin

q <= 0;

end

else begin

if(load) begin

q <= data;

end

else begin

if(ena) begin

q <= (q >> 1);

end

end

end

end

endmodule

109. Build a 100-bit left/right rotator, with synchronous load and left/right enable. A rotator shifts-in the shifted-out bit from the other end of the register, unlike a shifter that discards the shifted-out bit and shifts in a zero. If enabled, a rotator rotates the bits around and does not modify/discard them.

load: Loads shift register with data[99:0] instead of rotating.

ena[1:0]: Chooses whether and which direction to rotate.

2'b01 rotates right by one bit

2'b10 rotates left by one bit

2'b00 and 2'b11 do not rotate.

q: The contents of the rotator.

Ans;

module top\_module(

input clk,

input load,

input [1:0] ena,

input [99:0] data,

output reg [99:0] q);

always @(posedge clk) begin

if(load) q <= data;

else begin

if(ena == 2'b01) q <= {q[0], q[99:1]};

else if(ena == 2'b10) q<= {q[98:0], q[99]};

else q <= q;

end

end

endmodule

110. Build a 64-bit arithmetic shift register, with synchronous load. The shifter can shift both left and right, and by 1 or 8 bit positions, selected by amount.

An arithmetic right shift shifts in the sign bit of the number in the shift register (q[63] in this case) instead of zero as done by a logical right shift. Another way of thinking about an arithmetic right shift is that it assumes the number being shifted is signed and preserves the sign, so that arithmetic right shift divides a signed number by a power of two.

There is no difference between logical and arithmetic left shifts.

load: Loads shift register with data[63:0] instead of shifting.

ena: Chooses whether to shift.

amount: Chooses which direction and how much to shift.

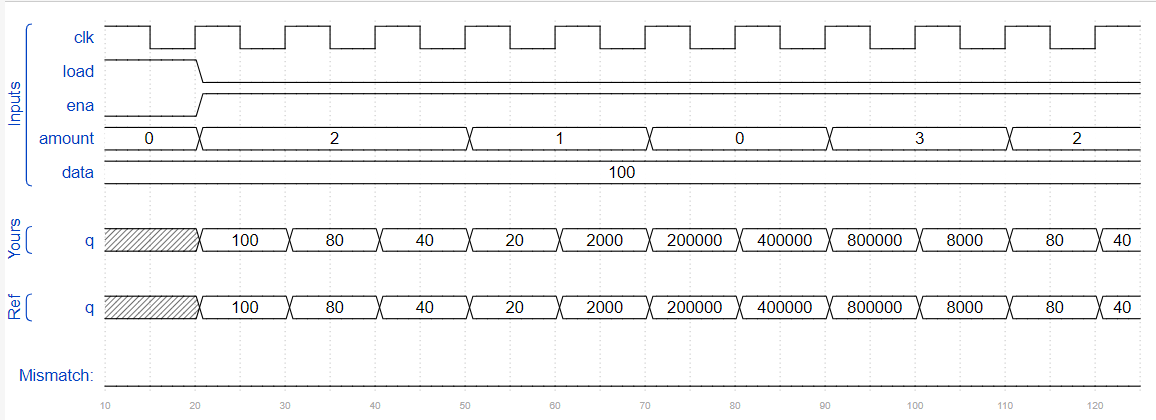
2'b00: shift left by 1 bit.

2'b01: shift left by 8 bits.

2'b10: shift right by 1 bit.

2'b11: shift right by 8 bits.

q: The contents of the shifter.

Ans;

module top\_module(

input clk,

input load,

input ena,

input [1:0] amount,

input [63:0] data,

output reg [63:0] q);

always @(posedge clk) begin

if(load) q <= data;

else begin

if(ena) begin

if(amount == 2'b00) q <= (q << 1);

else if(amount == 2'b01) q <= (q << 8);

else if(amount == 2'b10) begin

if(q[63] == 0) q <= (q >> 1);

else begin

q <= (q >> 1);

q[63] <= 1'b1;

end

end

else begin

if(q[63] == 0) q <= (q >> 8);

else begin

q <= (q >> 8);

q[63:56] <= {8{1'b1}};

end

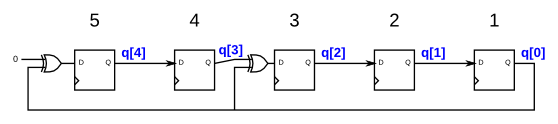
end

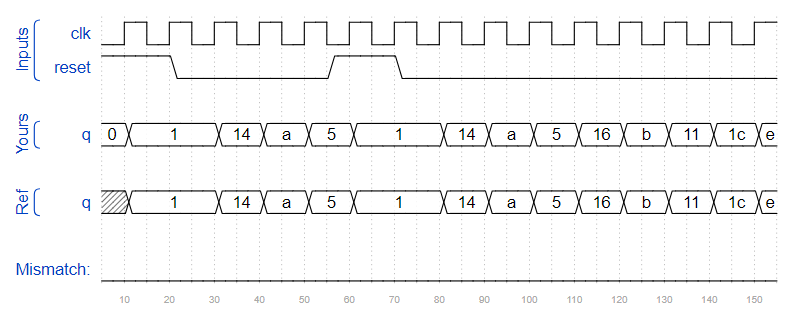
end

end

end

endmodule

111. The following diagram shows a 5-bit maximal-length Galois LFSR with taps at bit positions 5 and 3. (Tap positions are usually numbered starting from 1). Note that I drew the XOR gate at position 5 for consistency, but one of the XOR gate inputs is 0. 

Ans;

module top\_module(

input clk,

input reset,

output [4:0] q

);

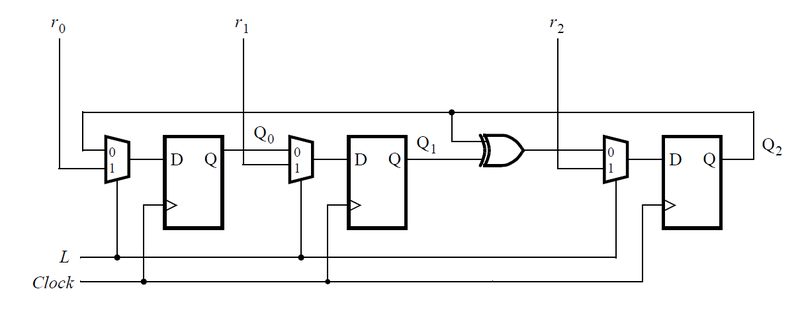
always @(posedge clk) begin

if(reset) q <= 5'h1;

else q <= {q[0]^1'b0, q[4], q[3]^q[0], q[2], q[1]};

end

endmodule

112. Write the Verilog code for this sequential circuit (Submodules are ok, but the top-level must be named top\_module). Assume that you are going to implement the circuit on the DE1-SoC board. Connect the R inputs to the SW switches, connect Clock to KEY[0], and L to KEY[1]. Connect the Q outputs to the red lights LEDR. 

Ans;

module top\_module (

input [2:0] SW,

input [1:0] KEY,

output [2:0] LEDR);

wire L;

wire clk;

wire [2:0] R;

reg [2:0] Q;

assign R = SW;

assign clk = KEY[0];

assign L = KEY[1];

always @(posedge clk) begin

if(L) Q <= R;

else Q <= {Q[2]^Q[1], Q[0], Q[2]};

end

assign LEDR = Q;

endmodule

113. Build a 32-bit Galois LFSR with taps at bit positions 32, 22, 2, and 1.

Ans;

module top\_module(

input clk,

input reset,

output [31:0] q

);

always @(posedge clk) begin

if(reset) q <= 32'h1;

else begin

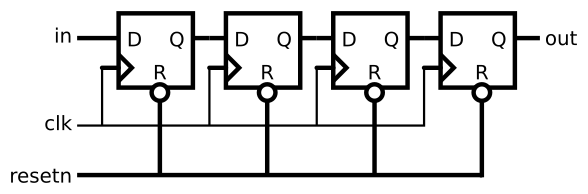
q <= {0^q[0], q[31:23], q[22]^q[0], q[21:3], q[2]^q[0], q[1]^q[0]};

end

end

endmodule

114. Implement the following circuit:



Ans;

module top\_module (

input clk,

input resetn,

input in,

output out);

reg [2:0] Q;

always @(posedge clk) begin

if(resetn) begin

Q[0] <= in;

Q[1] <= Q[0];

Q[2] <= Q[1];

out <= Q[2];

end

else begin

Q <= 3'b0;

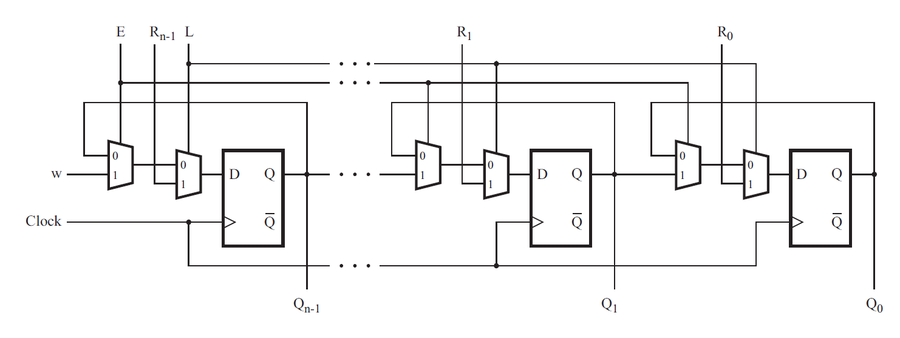
out <= 1'b0;

end

end

endmodule

115. Consider the n-bit shift register circuit shown below:



Ans;

module top\_module (

input [3:0] SW,

input [3:0] KEY,

output [3:0] LEDR

);

MUXDFF ins0(SW[3], KEY[0], KEY[1], KEY[2], KEY[3], LEDR[3]);

MUXDFF ins1(SW[2], KEY[0], KEY[1], KEY[2], LEDR[3], LEDR[2]);

MUXDFF ins2(SW[1], KEY[0], KEY[1], KEY[2], LEDR[2], LEDR[1]);

MUXDFF ins3(SW[0], KEY[0], KEY[1], KEY[2], LEDR[1], LEDR[0]);

endmodule

module MUXDFF (

input R,

input clk,

input E,

input L,

input w,

output out

);

wire [1:0] temp;

assign temp[0] = E ? w : out;

assign temp[1] = L ? R : temp[0];

always @(posedge clk) begin

out <= temp[1];

end

endmodule

116. In this question, you will design a circuit for an 8x1 memory, where writing to the memory is accomplished by shifting-in bits, and reading is "random access", as in a typical RAM. You will then use the circuit to realize a 3-input logic function.

First, create an 8-bit shift register with 8 D-type flip-flops. Label the flip-flop outputs from Q[0]...Q[7]. The shift register input should be called S, which feeds the input of Q[0] (MSB is shifted in first). The enable input controls whether to shift. Then, extend the circuit to have 3 additional inputs A,B,C and an output Z. The circuit's behaviour should be as follows: when ABC is 000, Z=Q[0], when ABC is 001, Z=Q[1], and so on. Your circuit should contain ONLY the 8-bit shift register, and multiplexers. (Aside: this circuit is called a 3-input look-up-table (LUT)).

Ans;

module top\_module (

input clk,

input enable,

input S,

input A, B, C,

output Z );

reg[7:0] q;

always @(posedge clk) begin

if(enable) begin

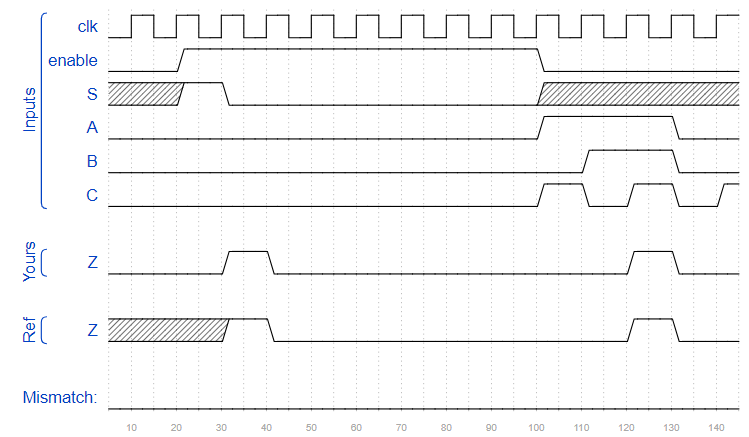
q <= {q[6:0], S}; //q[7]=q[6], q[6]=q[5]...q[0]=S

end

else q <= q;

end

always @(\*) begin

 case({A, B, C})

3'b000 : Z = q[0];

3'b001 : Z = q[1];

3'b010 : Z = q[2];

3'b011 : Z = q[3];

3'b100 : Z = q[4];

3'b101 : Z = q[5];

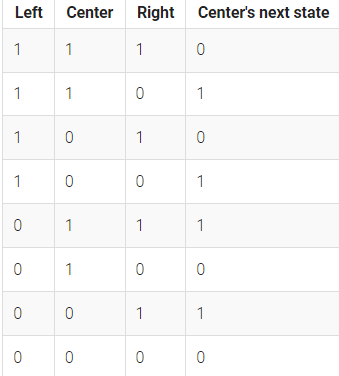
3'b110 : Z = q[6];

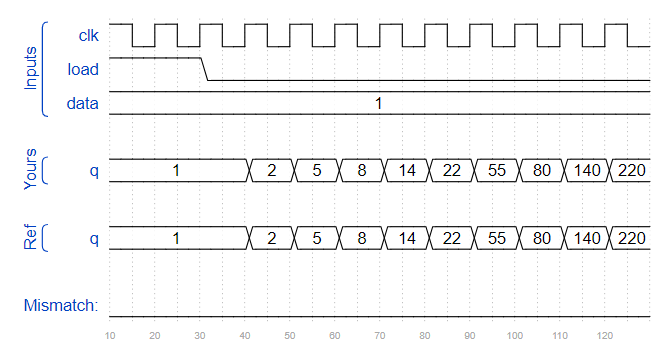
3'b111 : Z = q[7];

endcase

end

endmodule

117. In this circuit, create a 512-cell system (q[511:0]), and advance by one time step each clock cycle. The load input indicates the state of the system should be loaded with data[511:0]. Assume the boundaries (q[-1] and q[512]) are both zero (off).

Ans;

module top\_module(

input clk,

input load,

input [511:0] data,

output [511:0] q );

always@(posedge clk) begin

if(load) q <= data;

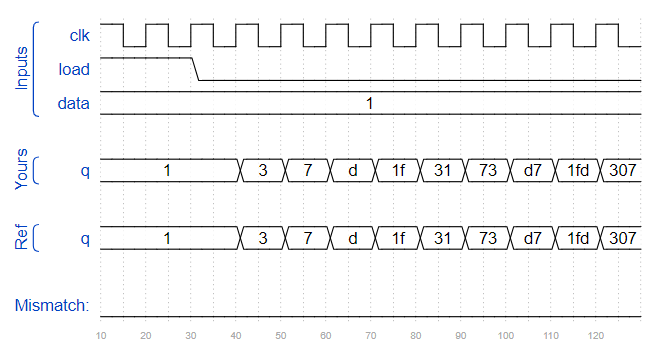
else q <= {1'b0, q[511:1]} ^ {q[510:0], 1'b0};

end

endmodule

118. There is a one-dimensional array of cells (on or off). At each time step, the state of each cell changes. In Rule 110, the next state of each cell depends only on itself and its two neighbours, according to the following table:

In this circuit, create a 512-cell system (q[511:0]), and advance by one time step each clock cycle. The load input indicates the state of the system should be loaded with data[511:0]. Assume the boundaries (q[-1] and q[512]) are both zero (off).

Ans;

module top\_module(

input clk,

input load,

input [511:0] data,

output [511:0] q

);

always @(posedge clk) begin

if (load) begin

q <= data;

end

else begin

q <= (((q[511:0] ^ {q[510:0], 1'b0}) & q[511:1]) | ((q[511:0] | {q[510:0], 1'b0}) & (~q[511:1])));

end

end

endmodule

119. The "game" is played on a two-dimensional grid of cells, where each cell is either 1 (alive) or 0 (dead). At each time step, each cell changes state depending on how many neighbours it has:

0-1 neighbour: Cell becomes 0.

2 neighbours: Cell state does not change.

3 neighbours: Cell becomes 1.

4+ neighbours: Cell becomes 0.

The game is formulated for an infinite grid. In this circuit, we will use a 16x16 grid. To make things more interesting, we will use a 16x16 toroid, where the sides wrap around to the other side of the grid. For example, the corner cell (0,0) has 8 neighbours: (15,1), (15,0), (15,15), (0,1), (0,15), (1,1), (1,0), and (1,15). The 16x16 grid is represented by a length 256 vector, where each row of 16 cells is represented by a sub-vector: q[15:0] is row 0, q[31:16] is row 1, etc. (This tool accepts SystemVerilog, so you may use 2D vectors if you wish.)

load: Loads data into q at the next clock edge, for loading initial state.

q: The 16x16 current state of the game, updated every clock cycle.

The game state should advance by one timestep every clock cycle.

Ans; module top\_module(

input clk,

input load,

input [255:0] data,

output [255:0] q );

reg [255:0] q\_next;

reg [3:0] sum;

always@(posedge clk) begin

if(load)

q <= data;

else begin

for(int i=0; i<256; i++) begin

if(i == 0)

sum = q[1] + q[16] + q[17] + q[240] + q[241] + q[15] + q[31] + q[255];

else if(i == 15)

sum = q[14] + q[16] + q[0] + q[240] + q[254] + q[30] + q[31] + q[255];

else if(i == 240)

sum = q[0] + q[15] + q[239] + q[241] + q[1] + q[224] + q[225] + q[255];

else if(i == 255)

sum = q[0] + q[15] + q[14] + q[224] + q[238] + q[240] + q[239] + q[254];

else if(0<i & i<15)

sum = q[i-1] + q[i+1] + q[i+15] + q[i+16] + q[i+17] + q[i+239] + q[i+240] + q[i+241];

else if(i%16 == 0)

sum = q[i-1] + q[i+1] + q[i+15] + q[i+16] + q[i+17] + q[i-16] + q[i-15] + q[i+31];

else if(i%16 == 15)

sum = q[i-1] + q[i+1] + q[i+15] + q[i+16] + q[i-17] + q[i-16] + q[i-15] + q[i-31];

else if(240<i & i<255)

sum = q[i-1] + q[i+1] + q[i-17] + q[i-16] + q[i-15] + q[i-239] + q[i-240] + q[i-241];

else

sum = q[i-1] + q[i+1] + q[i-17] + q[i-16] + q[i-15] + q[i+15] + q[i+16] + q[i+17];

case(sum)

2:q\_next[i] = q[i];

3:q\_next[i] = 1;

default:q\_next[i] = 0;

endcase

end

q = q\_next;

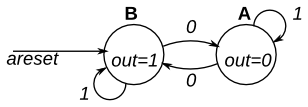
end

end

endmodule

120. This is a Moore state machine with two states, one input, and one output. Implement this state machine. Notice that the reset state is B.

This exercise is the same as fsm1s, but using asynchronous reset.



Ans;

module top\_module(

input clk,

input areset, // Asynchronous reset to state B

input in,

output out);

parameter A=0, B=1;

reg state, next\_state;

always @(\*) begin // This is a combinational always block

// State transition logic

case(state)

A : next\_state = (in == 1) ? A : B;

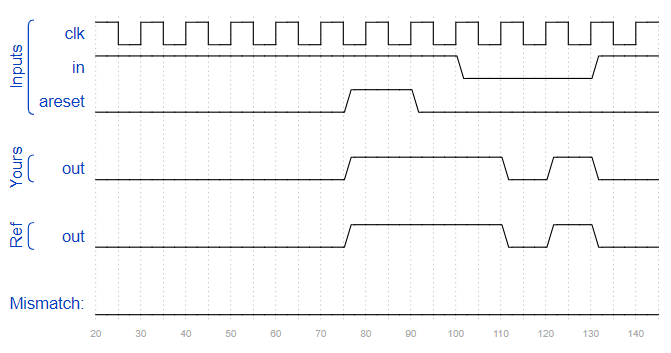
B : next\_state = (in == 1) ? B : A;

endcase

end

always @(posedge clk, posedge areset) begin // This is a sequential always block

// State flip-flops with asynchronous reset

 if(areset) state <= B;

else state <= next\_state;

end

// Output logic

assign out = (state == B);

endmodule

121. This is a Moore state machine with two states, one input, and one output. Implement this state machine. Notice that the reset state is B.

This exercise is the same as fsm1, but using synchronous reset. 

Ans;

// Note the Verilog-1995 module declaration syntax here:

module top\_module(clk, reset, in, out);

input clk;

input reset; // Synchronous reset to state B

input in;

output out;//

// Fill in state name declarations

parameter A = 0, B = 1;

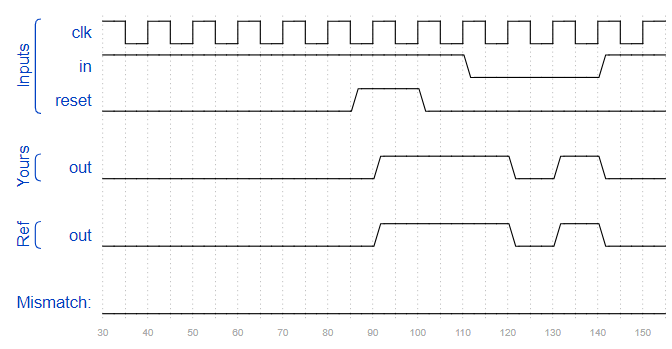
reg present\_state, next\_state;

always @(posedge clk) begin

if (reset) present\_state <= B;

else present\_state <= next\_state;

end

 always @(\*) begin

case (present\_state)

B : next\_state <= (in == 1) ? B : A;

A : next\_state <= (in == 1) ? A : B;

endcase

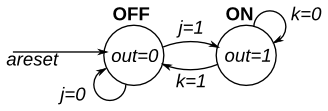
end

assign out = (present\_state == B);

endmodule

122. This is a Moore state machine with two states, two inputs, and one output. Implement this state machine.

This exercise is the same as fsm2s, but using asynchronous reset.

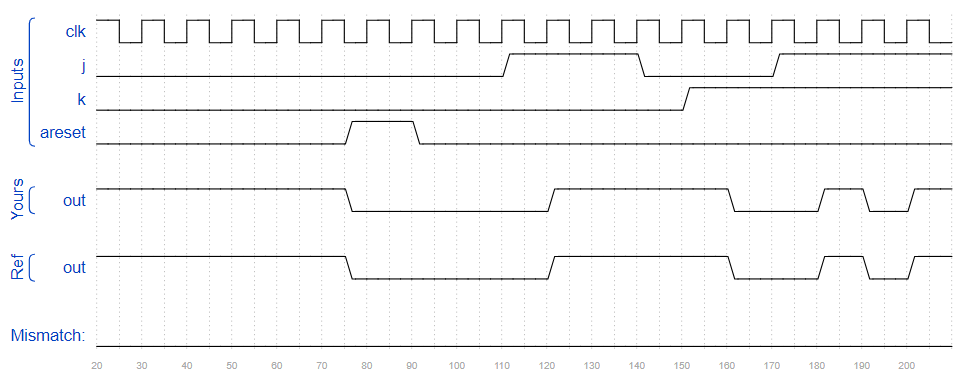


Ans;

module top\_module(

input clk,

input areset, // Asynchronous reset to OFF

 input j,

input k,

output out); //

parameter OFF=0, ON=1;

reg state, next\_state;

always @(\*) begin

case(state)

OFF:next\_state=(j==1)?ON:OFF;

ON:next\_state=(k==1)?OFF:ON;

endcase

end

always @(posedge clk, posedge areset) begin

if(areset)state<=OFF;

else state<=next\_state;

end

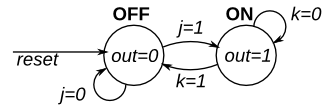
// Output logic

assign out = (state == ON);

endmodule

123. This is a Moore state machine with two states, two inputs, and one output. Implement this state machine.

This exercise is the same as fsm2, but using synchronous reset.



Ans;

module top\_module(

input clk,

input reset, // Synchronous reset to OFF

input j,

input k,

output out); //

parameter OFF=0, ON=1;

reg state, next\_state;

always @(\*) begin

case(state)

OFF:next\_state=(j==0)?OFF:ON;

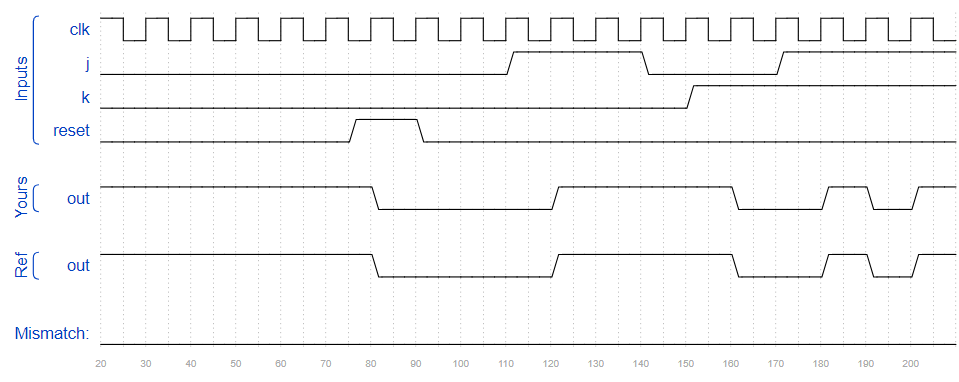
ON:next\_state=(k==0)?ON:OFF;

endcase

// State transition logic

end

always @(posedge clk) begin

 if(reset)state<=OFF;

else state<=next\_state;

end

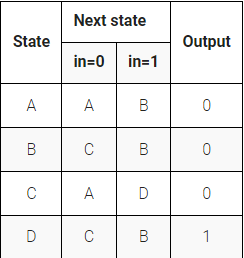
// Output logic

assign out = (state == ON);

endmodule

124. The following is the state transition table for a Moore state machine with one input, one output, and four states. Use the following state encoding: A=2'b00, B=2'b01, C=2'b10, D=2'b11.

Implement only the state transition logic and output logic (the combinational logic portion) for this state machine. Given the current state (state), compute the next\_state and output (out) based on the state transition table.



Ans;

module top\_module(

input in,

input [1:0] state,

output [1:0] next\_state,

output out); //

parameter A=0, B=1, C=2, D=3;

always@(\*)begin

case(state)

A:next\_state=(in==0)?A:B;

B:next\_state=(in==0)?C:B;

C:next\_state=(in==0)?A:D;

D:next\_state=(in==0)?C:B;

endcase

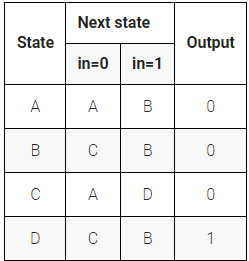
end

assign out=(state==D);

endmodule

125. The following is the state transition table for a Moore state machine with one input, one output, and four states. Use the following one-hot state encoding: A=4'b0001, B=4'b0010, C=4'b0100, D=4'b1000.

Derive state transition and output logic equations by inspection assuming a one-hot encoding. Implement only the state transition logic and output logic (the combinational logic portion) for this state machine. (The testbench will test with non-one hot inputs to make sure you're not trying to do something more complicated).



Ans;

module top\_module(

input in,

input [3:0] state,

output [3:0] next\_state,

output out);

parameter A=0, B=1, C=2, D=3;

assign next\_state[A] = (state[A] & ~in) | (state[C] & ~in);

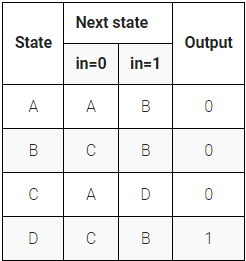
assign next\_state[B] = (state[A] & in) | (state[B] & in) | (state[D] & in);

assign next\_state[C] = (state[B] & ~in) | (state[D] & ~in);

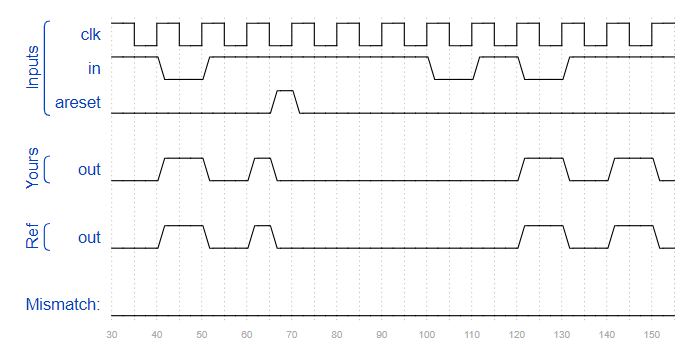
assign next\_state[D] = state[C] & in;

assign out = (state[D]);

endmodule

126. The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include an asynchronous reset that resets the FSM to state A.

Ans;

module top\_module(

input clk,

input in,

input areset,

output out); //

reg [2:0] state, next\_state;

parameter A = 1, B = 2, C = 3, D = 4;

always @(\*) begin

case(state)

A : next\_state = (in == 1) ? B : A;

B : next\_state = (in == 1) ? B : C;

C : next\_state = (in == 1) ? D : A;

D : next\_state = (in == 1) ? B : C;

endcase

end

always @(posedge clk or posedge areset) begin

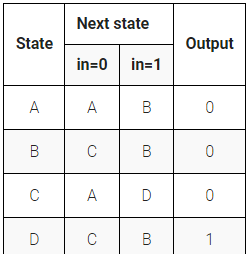
if(areset) state <= A;

else state <= next\_state;

end

assign out = (state == D);

endmodule

127. The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include a synchronous reset that resets the FSM to state A. (This is the same problem as Fsm3 but with a synchronous reset.) Ans;

module top\_module(

input clk,

input in,

input reset,

output out);

reg [2:0] state, next\_state;

parameter A = 1, B = 2, C = 3, D = 4;

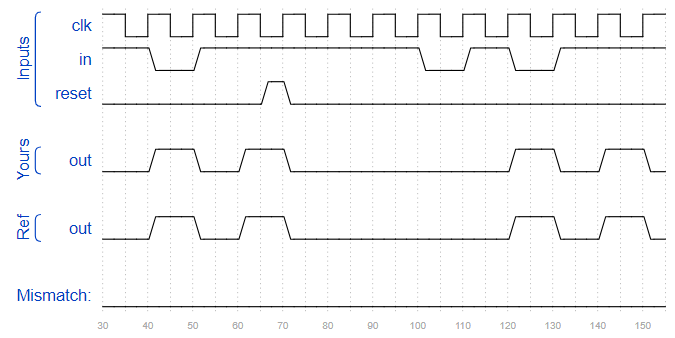
always @(\*) begin

case(state)

A : next\_state = (in == 1) ? B : A;

B : next\_state = (in == 1) ? B : C;

C : next\_state = (in == 1) ? D : A;

 D : next\_state = (in == 1) ? B : C;

endcase

end

always @(posedge clk) begin

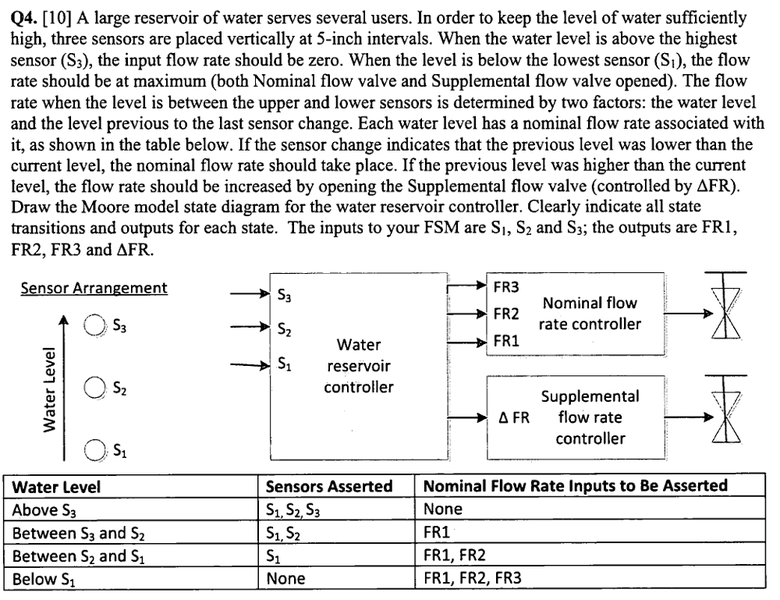
if(reset) state <= A;

else state <= next\_state;

end

assign out = (state == D);

endmodule

128. 

Ans;

module top\_module (

input clk,

input reset,

input [3:1] s,

output fr3,

output fr2,

output fr1,

output dfr

);

localparam [2:0] A = 3'd0, //water level:below s1

B0 = 3'd1, //s1~s2, and previous level is higher

B1 = 3'd2, //s1~s2, and previous level is lower

C0 = 3'd3, //s2~s3, and previous level is higher

C1 = 3'd4, //s2~s3, and previous level is lower

D = 3'd5; //above s3

reg [2:0] state, next\_state;

always @(posedge clk) begin

if(reset) state <= A;

else state <= next\_state;

end

always @(\*) begin

case(state)

A : next\_state = (s[1]) ? B1 : A;

B0 : next\_state = (s[2]) ? C1 : ((s[1]) ? B0 : A);

B1 : next\_state = (s[2]) ? C1 : ((s[1]) ? B1 : A);

C0 : next\_state = (s[3]) ? D : ((s[2]) ? C0 : B0);

C1 : next\_state = (s[3]) ? D : ((s[2]) ? C1 : B0);

D : next\_state = (s[3]) ? D : C0;

default : next\_state = 3'bxxx;

endcase

end

always @(\*) begin

case(state)

A : {fr3, fr2, fr1, dfr} = 4'b1111;

B0 : {fr3, fr2, fr1, dfr} = 4'b0111;

B1 : {fr3, fr2, fr1, dfr} = 4'b0110;

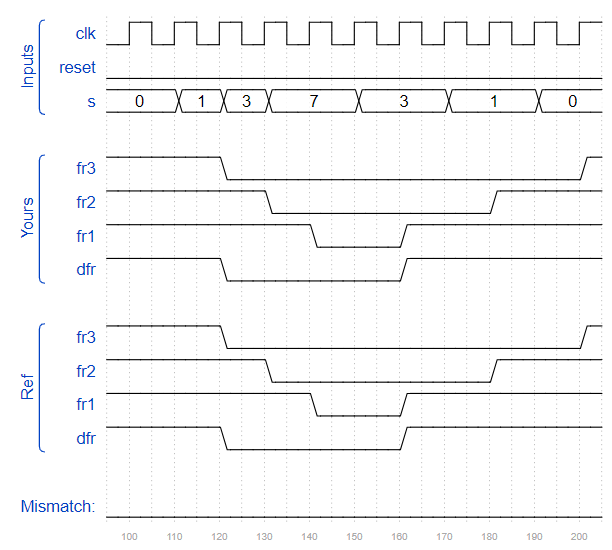
C0 : {fr3, fr2, fr1, dfr} = 4'b0011;

C1 : {fr3, fr2, fr1, dfr} = 4'b0010;

D : {fr3, fr2, fr1, dfr} = 4'b0000;

default : {fr3, fr2, fr1, dfr} = 4'bxxxx;

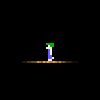
endcase

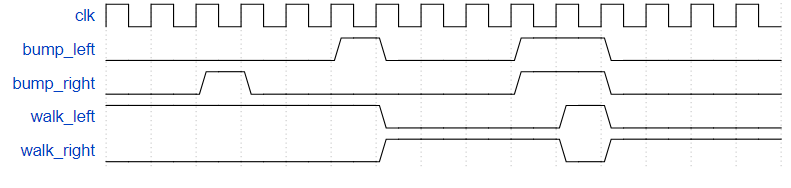
 end

endmodule

129. The game Lemmings involves critters with fairly simple brains. So simple that we are going to model it using a finite state machine.

In the Lemmings' 2D world, Lemmings can be in one of two states: walking left or walking right. It will switch directions if it hits an obstacle. In particular, if a Lemming is bumped on the left, it will walk right. If it's bumped on the right, it will walk left. If it's bumped on both sides at the same time, it will still switch directions.

Implement a Moore state machine with two states, two inputs, and one output that models this behaviour.



Ans;

module top\_module(

input clk,

input areset,

input bump\_left,

input bump\_right,

output walk\_left,

output walk\_right);

parameter LEFT = 0, RIGHT = 1;

reg state, next\_state;

always @(\*) begin

case(state)

LEFT : next\_state = (bump\_left) ? RIGHT : LEFT;

RIGHT : next\_state = (bump\_right) ? LEFT : RIGHT;

endcase

end

always @(posedge clk, posedge areset) begin

if(areset) state <= LEFT;

else begin

state <= next\_state;

end

end

assign walk\_left = (state == LEFT);

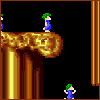
 assign walk\_right = (state == RIGHT);

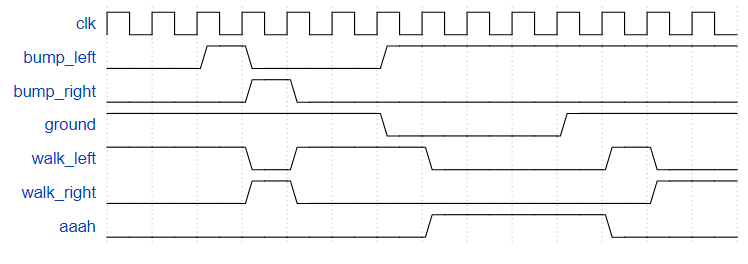
endmodule

130. In addition to walking left and right, Lemmings will fall (and presumably go "aaah!") if the ground disappears underneath them.

In addition to walking left and right and changing direction when bumped, when ground=0, the Lemming will fall and say "aaah!". When the ground reappears (ground=1), the Lemming will resume walking in the same direction as before the fall. Being bumped while falling does not affect the walking direction, and being bumped in the same cycle as ground disappears (but not yet falling), or when the ground reappears while still falling, also does not affect the walking direction.

Build a finite state machine that models this behaviour.





Ans;

module top\_module(

input clk,

input areset,

input bump\_left,

input bump\_right,

input ground,

output walk\_left,

output walk\_right,

output aaah );

localparam [1:0] WALK\_L = 2'b00,

WALK\_R = 2'b01,

FALL\_L = 2'b10,

FALL\_R = 2'b11;

reg [1:0] state, next;

always @(posedge clk or posedge areset) begin

if(areset) state <= WALK\_L;

else begin

state <= next;

end

end

always @(\*) begin

case(state)

WALK\_L : next = (ground == 0) ? FALL\_L : ((bump\_left == 1) ? WALK\_R : WALK\_L);

WALK\_R : next = (ground == 0) ? FALL\_R : ((bump\_right == 1) ? WALK\_L : WALK\_R);

FALL\_L : next = (ground == 1) ? WALK\_L : FALL\_L;

FALL\_R : next = (ground == 1) ? WALK\_R : FALL\_R;

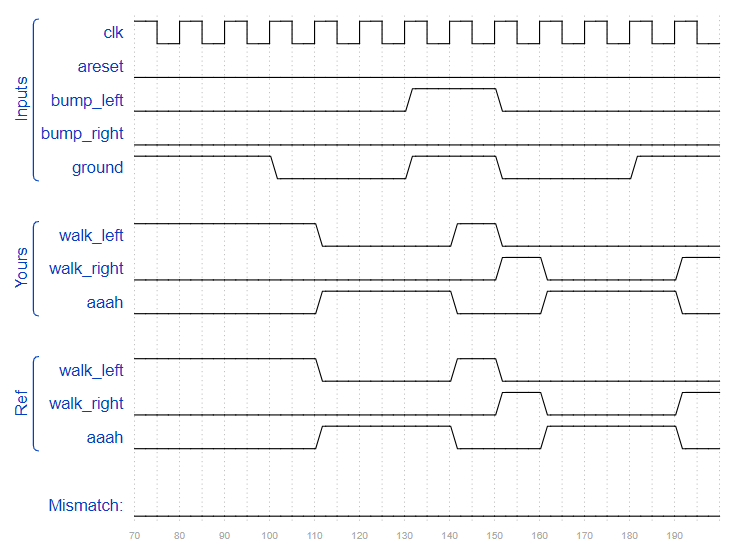
endcase

end

assign walk\_left = (state == WALK\_L);

assign walk\_right = (state == WALK\_R);

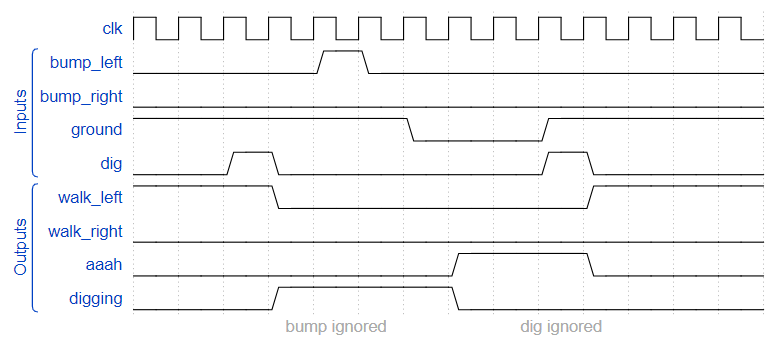
assign aaah = ((state == FALL\_L) || (state == FALL\_R));

endmodule

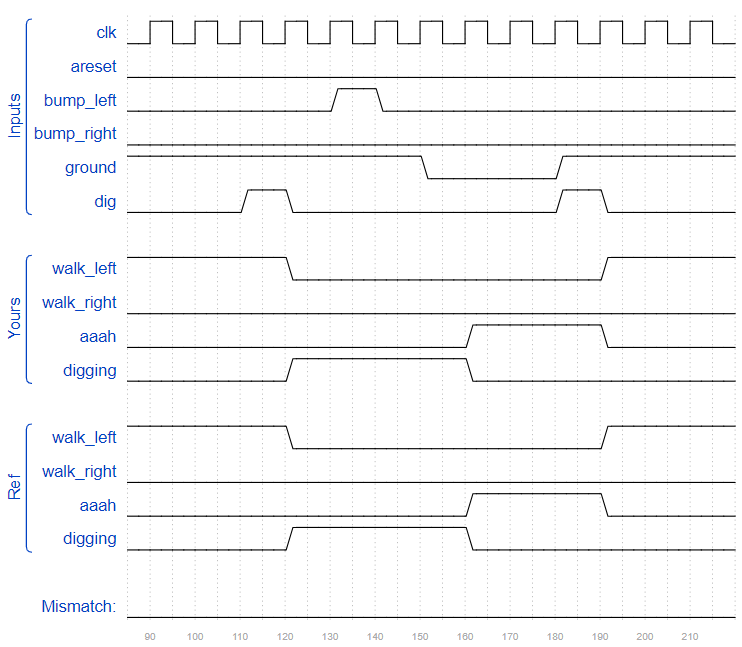
131. In addition to walking and falling, Lemmings can sometimes be told to do useful things, like dig (it starts digging when dig=1). A Lemming can dig if it is currently walking on ground (ground=1 and not falling), and will continue digging until it reaches the other side (ground=0). At that point, since there is no ground, it will fall (aaah!), then continue walking in its original direction once it hits ground again. As with falling, being bumped while digging has no effect, and being told to dig when falling or when there is no ground is ignored.

(In other words, a walking Lemming can fall, dig, or switch directions. If more than one of these conditions are satisfied, fall has higher precedence than dig, which has higher precedence than switching directions.)

Extend your finite state machine to model this behaviour.



Ans;

module top\_module(

input clk,

input areset,

input bump\_left,

input bump\_right,

input ground,

input dig,

output walk\_left,

output walk\_right,

output aaah,

output digging );

localparam [2:0] WALK\_L = 3'b000,

WALK\_R = 3'b001,

FALL\_L = 3'b010,

FALL\_R = 3'b011,

DIG\_L = 3'b100,

DIG\_R = 3'b101;

reg[2:0]state,next;

always@(posedge clk or posedge areset)begin

if(areset) state <= WALK\_L;

else state<=next;

end

always @(\*) begin

case(state)

WALK\_L : begin

if(!ground) next = FALL\_L;

else begin

if(dig) next = DIG\_L;

else begin

if(bump\_left) next = WALK\_R;

else next = WALK\_L;

end

end

end

WALK\_R : begin

if(!ground) next = FALL\_R;

else begin

if(dig) next = DIG\_R;

else begin

if(bump\_right) next = WALK\_L;

else next = WALK\_R;

end

end

end

FALL\_L : next = (ground) ? WALK\_L : FALL\_L;

FALL\_R : next = (ground) ? WALK\_R : FALL\_R;

DIG\_L : next = (ground) ? DIG\_L : FALL\_L;

DIG\_R : next = (ground) ? DIG\_R : FALL\_R;

endcase

end

assign walk\_left = (state == WALK\_L);

assign walk\_right = (state == WALK\_R);

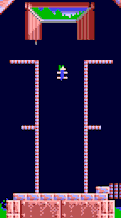
assign aaah = ((state == FALL\_L) || (state == FALL\_R));

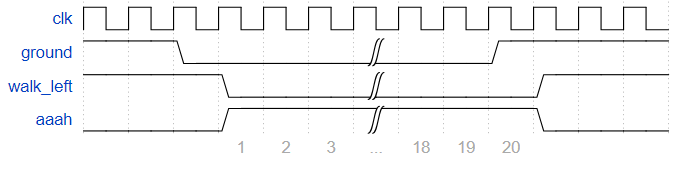
assign digging = ((state == DIG\_L) || (state == DIG\_R));

endmodule

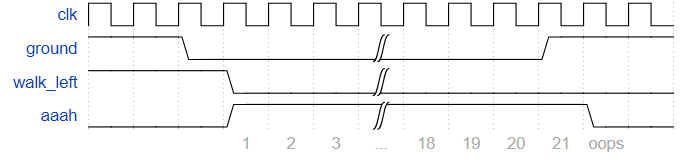
132. Although Lemmings can walk, fall, and dig, Lemmings aren't invulnerable. If a Lemming falls for too long then hits the ground, it can splatter. In particular, if a Lemming falls for more than 20 clock cycles then hits the ground, it will splatter and cease walking, falling, or digging (all 4 outputs become 0), forever (Or until the FSM gets reset). There is no upper limit on how far a Lemming can fall before hitting the ground. Lemmings only splatter when hitting the ground; they do not splatter in mid-air.

Extend your finite state machine to model this behaviour.

Falling for 20 cycles is survivable:



Falling for 21 cycles causes splatter:



Ans;

module top\_module(

input clk,

input areset, // Freshly brainwashed Lemmings walk left.

input bump\_left,

input bump\_right,

input ground,

input dig,

output walk\_left,

output walk\_right,

output aaah,

output digging );

localparam [2:0] WALK\_L = 3'b000,

WALK\_R = 3'b001,

FALL\_L = 3'b010,

FALL\_R = 3'b011,

DIG\_L = 3'b100,

DIG\_R = 3'b101,

SPLATTER = 3'b110;

reg [2:0] state, next;

reg [6:0] count;

always @(posedge clk or posedge areset) begin

if(areset) state <= WALK\_L;

else if(state == FALL\_R || state == FALL\_L) begin

count <= count + 1;

state <= next;

end

else begin

state <= next;

count <= 0;

end

end

always @(\*) begin

case(state)

WALK\_L : begin

if(!ground) next = FALL\_L;

else begin

if(dig) next = DIG\_L;

else begin

if(bump\_left) next = WALK\_R;

else next = WALK\_L;

end

end

end

WALK\_R : begin

if(!ground) next = FALL\_R;

else begin

if(dig) next = DIG\_R;

else begin

if(bump\_right) next = WALK\_L;

else next = WALK\_R;

end

end

end

FALL\_L : begin

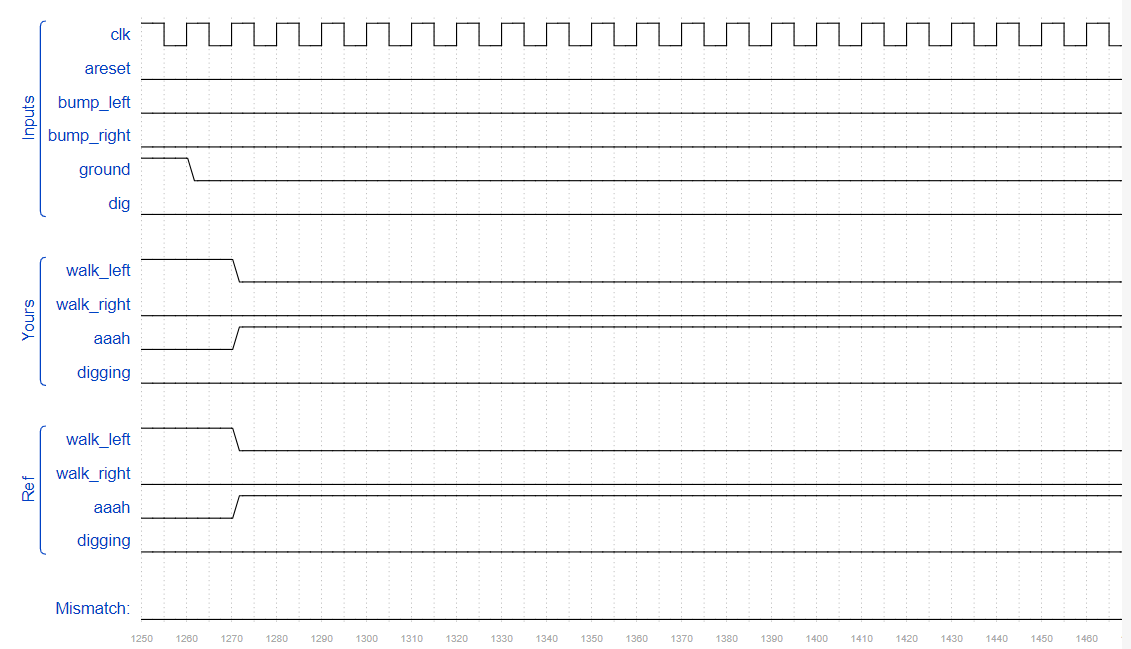
if(ground) begin

if(count > 19) next = SPLATTER;

else next = WALK\_L;

end

else next = FALL\_L;

 end

FALL\_R : begin

if(ground) begin

if(count > 19) next = SPLATTER;

else next = WALK\_R;

end

else next = FALL\_R;

end

DIG\_L : next = (ground) ? DIG\_L : FALL\_L;

DIG\_R : next = (ground) ? DIG\_R : FALL\_R;

SPLATTER : next = SPLATTER;

endcase

end

assign walk\_left = (state == WALK\_L);

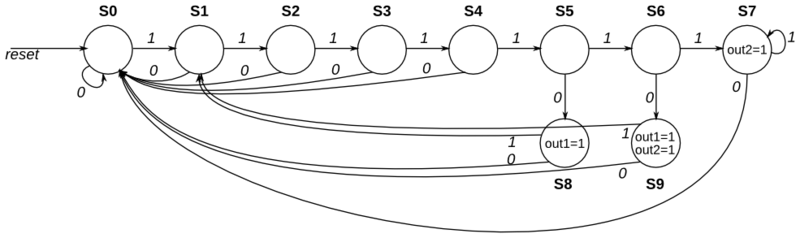
assign walk\_right = (state == WALK\_R);

assign aaah = ((state == FALL\_L) || (state == FALL\_R));

assign digging = ((state == DIG\_L) || (state == DIG\_R));

endmodule

133. Given the following state machine with 1 input and 2 outputs:



Suppose this state machine uses one-hot encoding, where state[0] through state[9] correspond to the states S0 though S9, respectively. The outputs are zero unless otherwise specified.

Implement the state transition logic and output logic portions of the state machine (but not the state flip-flops). You are given the current state in state[9:0] and must produce next\_state[9:0] and the two outputs. Derive the logic equations by inspection assuming a one-hot encoding. (The testbench will test with non-one hot inputs to make sure you're not trying to do something more complicated).

Ans;

module top\_module(

input in,

input [9:0] state,

output [9:0] next\_state,

output out1,

output out2);

localparam S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4,

S5 = 5, S6 = 6, S7 = 7, S8 = 8, S9 = 9;

//states

assign next\_state[S0] = (state[S0] & !in) | (state[S1] & !in) | (state[S2] & !in) | (state[S3] & !in) |

(state[S4] & !in) | (state[S7] & !in) | (state[S8] & !in) | (state[S9] & !in);

assign next\_state[S1] = (state[S0] & in) | (state[S8] & in) | (state[S9] & in);

assign next\_state[S2] = state[S1] & in;

assign next\_state[S3] = state[S2] & in;

assign next\_state[S4] = state[S3] & in;

assign next\_state[S5] = state[S4] & in;

assign next\_state[S6] = state[S5] & in;

assign next\_state[S7] = (state[S6] & in) | (state[7] & in);

assign next\_state[S8] = state[S5] & !in;

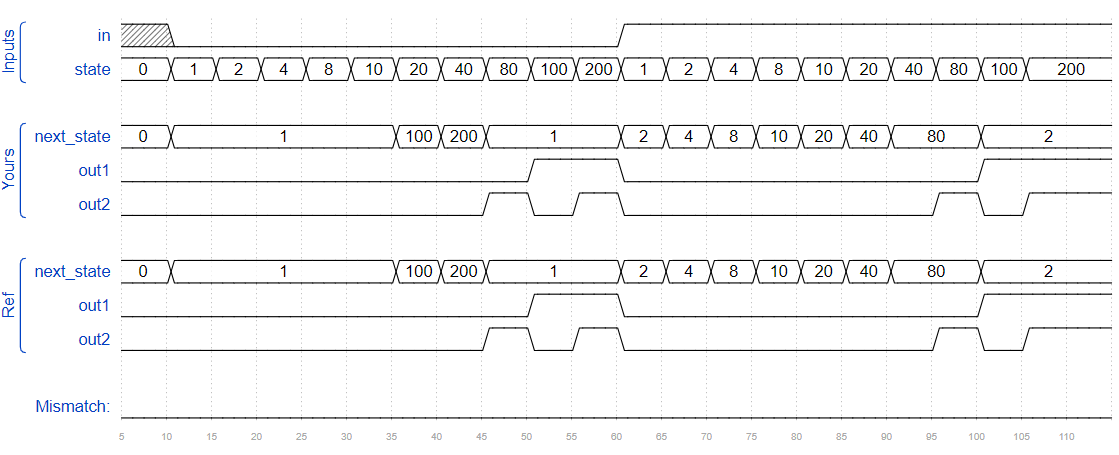
assign next\_state[S9] = state[S6] & !in;

//output

assign out1 = state[8] | state[9];

assign out2 = state[7] | state[9];

endmodule



134. The PS/2 mouse protocol sends messages that are three bytes long. However, within a continuous byte stream, it's not obvious where messages start and end. The only indication is that the first byte of each three byte message always has bit[3]=1 (but bit[3] of the other two bytes may be 1 or 0 depending on data).

We want a finite state machine that will search for message boundaries when given an input byte stream. The algorithm we'll use is to discard bytes until we see one with bit[3]=1. We then assume that this is byte 1 of a message, and signal the receipt of a message once all 3 bytes have been received (done).

The FSM should signal done in the cycle immediately after the third byte of each message was successfully received.

Ans;

module top\_module(

input clk,

input [7:0] in,

input reset, // Synchronous reset

output done); //

localparam [1:0] BYTE1 = 2'b00,

BYTE2 = 2'b01,

BYTE3 = 2'b10,

DONE = 2'b11;

reg [1:0] state, next;

// State transition logic (combinational)

always @(\*) begin

case(state)

BYTE1 : next = (in[3]) ? BYTE2 : BYTE1;

BYTE2 : next = BYTE3;

BYTE3 : next = DONE;

DONE : next = (in[3]) ? BYTE2 : BYTE1;

endcase

end

// State flip-flops (sequential)

always @(posedge clk) begin

if(reset) state <= BYTE1;

else state <= next;

end

// Output logic

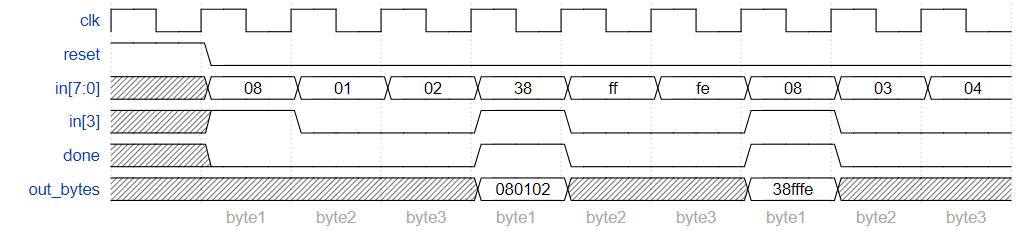
assign done = (state == DONE);

endmodule

135. Now that you have a state machine that will identify three-byte messages in a PS/2 byte stream, add a datapath that will also output the 24-bit (3 byte) message whenever a packet is received (out\_bytes[23:16] is the first byte, out\_bytes[15:8] is the second byte, etc.).

out\_bytes needs to be valid whenever the done signal is asserted. You may output anything at other times (i.e., don't-care).

For example:



Ans;

module top\_module(

input clk,

input [7:0] in,

input reset, // Synchronous reset

output [23:0] out\_bytes,

output done); //

localparam [1:0] BYTE1 = 2'b00,

BYTE2 = 2'b01,

BYTE3 = 2'b10,

DONE = 2'b11;

reg [1:0] state, next;

reg [23:0] data;

// State transition logic (combinational)

always @(\*) begin

case(state)

BYTE1 : next = (in[3]) ? BYTE2 : BYTE1;

BYTE2 : next = BYTE3;

BYTE3 : next = DONE;

DONE : next = (in[3]) ? BYTE2 : BYTE1;

endcase

end

// State flip-flops (sequential)

always @(posedge clk) begin

if(reset) state <= BYTE1;

else state <= next;

end

// New: Datapath to store incoming bytes.

always @(posedge clk) begin

if (reset) data <= 24'b0;

else data <= {data[15:8], data[7:0], in};

end

// Output logic

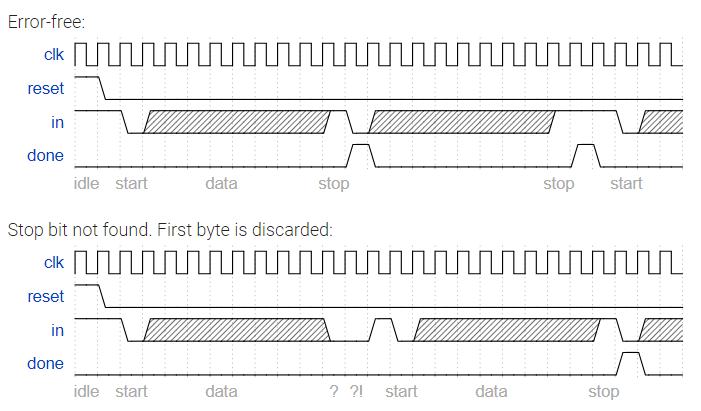
assign done = (state == DONE);

assign out\_bytes = (done) ? data : 23'b0;

endmodule

136. In many (older) serial communications protocols, each data byte is sent along with a start bit and a stop bit, to help the receiver delimit bytes from the stream of bits. One common scheme is to use one start bit (0), 8 data bits, and 1 stop bit (1). The line is also at logic 1 when nothing is being transmitted (idle).

Design a finite state machine that will identify when bytes have been correctly received when given a stream of bits. It needs to identify the start bit, wait for all 8 data bits, then verify that the stop bit was correct. If the stop bit does not appear when expected, the FSM must wait until it finds a stop bit before attempting to receive the next byte.



Ans;

module top\_module(

input clk,

input in,

input reset, // Synchronous reset

output done

);

localparam [2:0] IDLE = 3'b000,

START = 3'b001,

RECEIVE = 3'b010,

WAIT = 3'b011,

STOP = 3'b100;

reg [2:0] state, next;

reg [3:0] i;

always @(\*) begin

case(state)

IDLE : next = (in) ? IDLE : START;

START : next = RECEIVE;

RECEIVE : begin

if (i == 8) begin

if (in) next = STOP;

else next = WAIT;

end

else next = RECEIVE;

end

WAIT : next = (in) ? IDLE : WAIT;

STOP : next = (in) ? IDLE : START;

endcase

end

always @(posedge clk) begin

if(reset) state <= IDLE;

else state <= next;

end

always @(posedge clk) begin

if (reset) begin

done <= 0;

i <= 0;

end

else begin

case(next)

RECEIVE : begin

done <= 0;

i = i + 1;

end

STOP : begin

done <= 1;

i <= 0;

end

default : begin

done <= 0;

i <= 0;

end

endcase

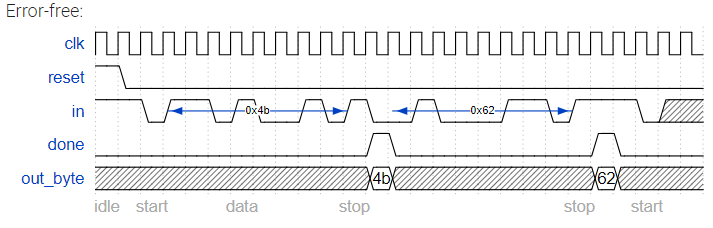
end

end

endmodule

137. Now that you have a finite state machine that can identify when bytes are correctly received in a serial bitstream, add a datapath that will output the correctly-received data byte. out\_byte needs to be valid when done is 1, and is don't-care otherwise.

Note that the serial protocol sends the least significant bit first.



Ans;

module top\_module(

input clk,

input in,

input reset, // Synchronous reset

output [7:0] out\_byte,

output done

);

// Use FSM from Fsm\_serial

localparam [2:0] IDLE = 3'b000,

START = 3'b001,

RECEIVE = 3'b010,

WAIT = 3'b011,

STOP = 3'b100;

reg [2:0] state, next;

reg [3:0] i;

reg [7:0] out;

always @(\*) begin

case(state)

IDLE : next = (in) ? IDLE : START;

START : next = RECEIVE;

RECEIVE : begin

if (i == 8) begin

if (in) next = STOP;

else next = WAIT;

end

else next = RECEIVE;

end

WAIT : next = (in) ? IDLE : WAIT;

STOP : next = (in) ? IDLE : START;

endcase

end

always @(posedge clk) begin

if(reset) state <= IDLE;

else state <= next;

end

always @(posedge clk) begin

if (reset) begin

done <= 0;

i <= 0;

end

else begin

case(next)

RECEIVE : begin

done <= 0;

i = i + 1;

end

STOP : begin

done <= 1;

i <= 0;

end

default : begin

done <= 0;

i <= 0;

end

endcase

end

end

// New: Datapath to latch input bits.

always @(posedge clk) begin

if (reset) out <= 0;

else if (next == RECEIVE)

out[i] <= in;

end

assign out\_byte = (done) ? out : 8'b0;

endmodule

139. Create a finite state machine to recognize these three sequences:

0111110: Signal a bit needs to be discarded (disc).

01111110: Flag the beginning/end of a frame (flag).

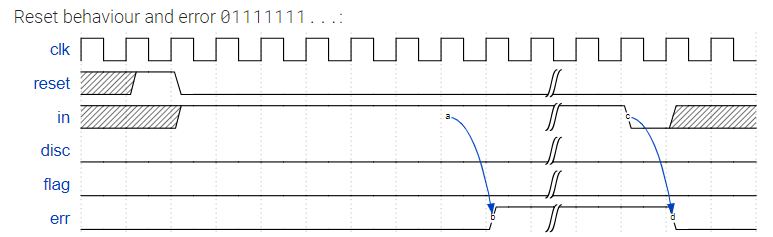
01111111...: Error (7 or more 1s) (err).

When the FSM is reset, it should be in a state that behaves as though the previous input were 0.

Here are some example sequences that illustrate the desired operation.

Discard 0111110:





Ans;

module top\_module(

input clk,

input reset, // Synchronous reset

input in,

output disc,

output flag,

output err);

localparam [3:0] NONE = 0,

ONE = 1,

TWO = 2,

THREE= 3,

FOUR = 4,

FIVE = 5,

SIX = 6,

DISC = 7,

FLAG = 8,

ERR = 9;

reg [3:0] state, next;

always @(\*) begin

case (state)

NONE : next = (in) ? ONE : NONE;

ONE : next = (in) ? TWO : NONE;

TWO : next = (in) ? THREE : NONE;

THREE: next = (in) ? FOUR : NONE;

FOUR : next = (in) ? FIVE : NONE;

FIVE : next = (in) ? SIX : DISC;

SIX : next = (in) ? ERR : FLAG;

DISC : next = (in) ? ONE : NONE;

FLAG : next = (in) ? ONE : NONE;

ERR : next = (in) ? ERR : NONE;

endcase

end

always @(posedge clk) begin

if (reset)

state <= NONE;

else

state <= next;

end

assign disc = (state == DISC);

assign flag = (state == FLAG);

assign err = (state == ERR);

endmodule

140. Implement a Mealy-type finite state machine that recognizes the sequence "101" on an input signal named x. Your FSM should have an output signal, z, that is asserted to logic-1 when the "101" sequence is detected. Your FSM should also have an active-low asynchronous reset. You may only have 3 states in your state machine. Your FSM should recognize overlapping sequences.

Ans;

module top\_module (

input clk,

input aresetn, // Asynchronous active-low reset

input x,

output z );

localparam [1:0] IDLE = 0,

ONE = 1,

ONE\_ZERO = 2;

reg [1:0] state, next;

always @(\*) begin

case (state)

IDLE : begin

next = (x) ? ONE : IDLE;

z = 0;

end

ONE : begin

next = (x) ? ONE : ONE\_ZERO;

z = 0;

end

ONE\_ZERO : begin

if (x) begin

next = ONE;

z = 1;

end

else begin

next = IDLE;

z = 0;

end

end

endcase

end

always @(posedge clk or negedge aresetn) begin

if (~aresetn) state <= IDLE;

else state <= next;

end

endmodule

