HDLBits- Verilog Coding
Bharath Shenoy

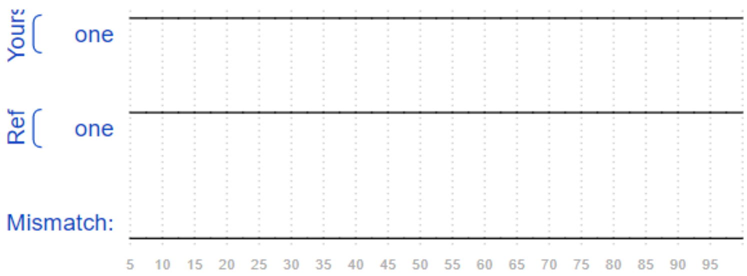
Week 1:

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

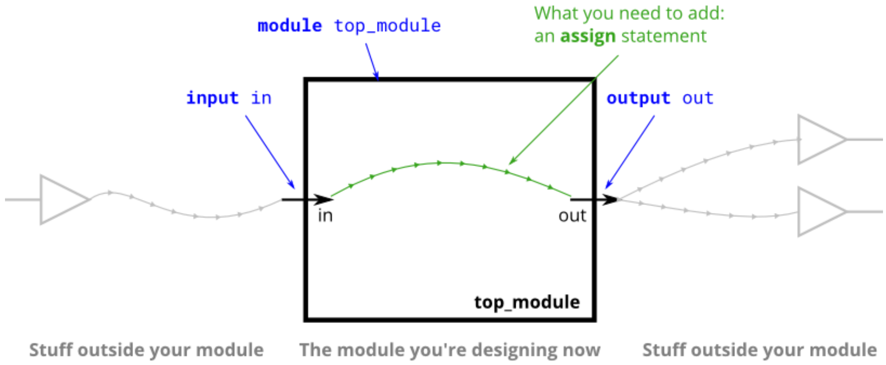
//Code

```
        module top_module( output one );

        // Insert your code here
            assign one = 1'b1;

endmodule
```

//Output



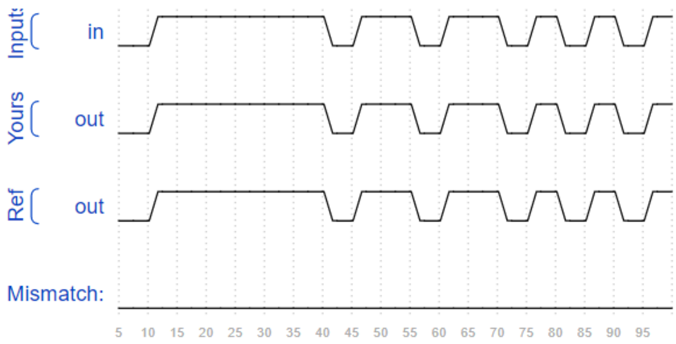
2. Create a module with one input and one output that behaves like a wire.



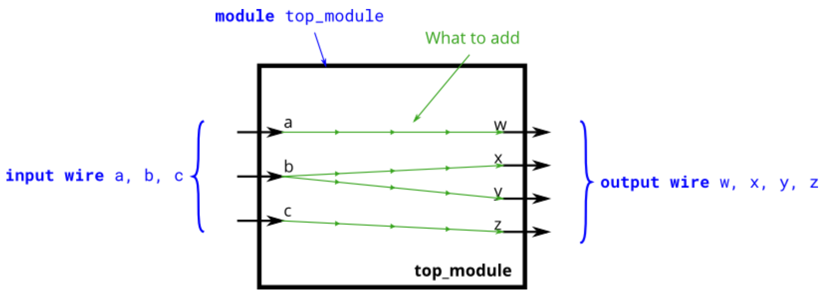
//Code

```
module top_module( input in, output out );

        assign out=in;
endmodule
```

//Output



3. 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

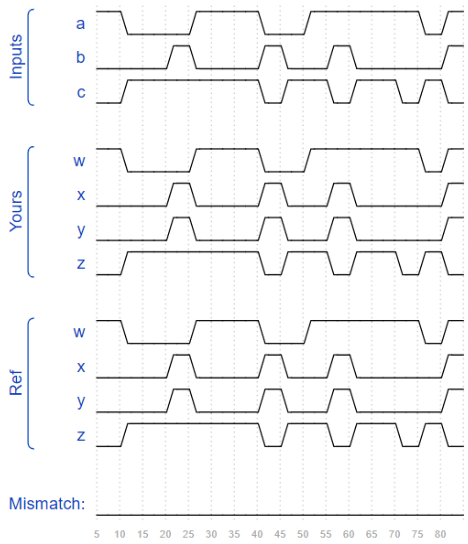


//Code

```
module top_module(
        input a,b,c,
        output w,x,y,z );
    assign w=a;
        assign y=b;
        assign x=b;
        assign z=c;
endmodule
```

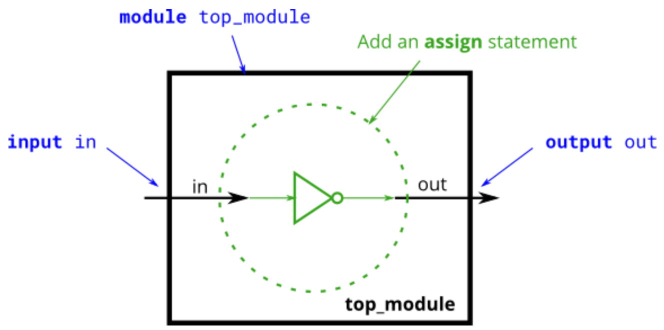
//Output

HDLBits- Verilog Coding
Bharath Shenoy



<u>Weak 2:</u>

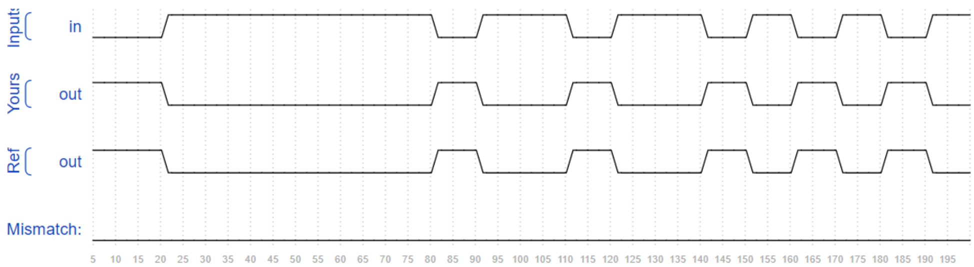
4. Create a module that implements a NOT gate.



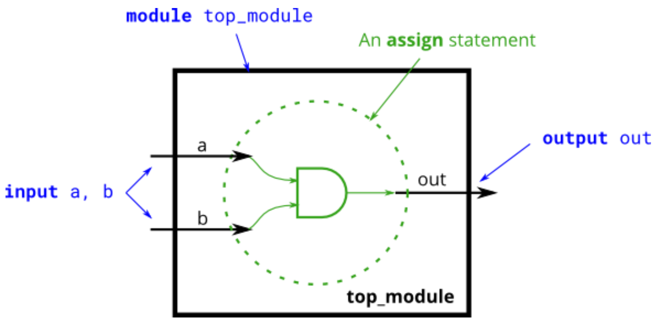
//Code

```
module top_module( input in, output out );
        assign out=~in;
endmodule
```

//Output



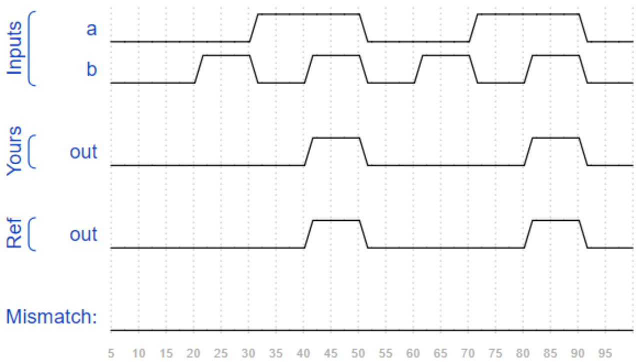
5. Create a module that implements an AND gate.



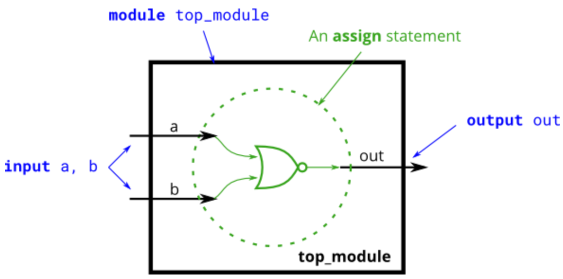
//Code

```
module top_module(
            input a,
            input b,
            output out );
        assign out=a&b;
endmodule
```

//Output



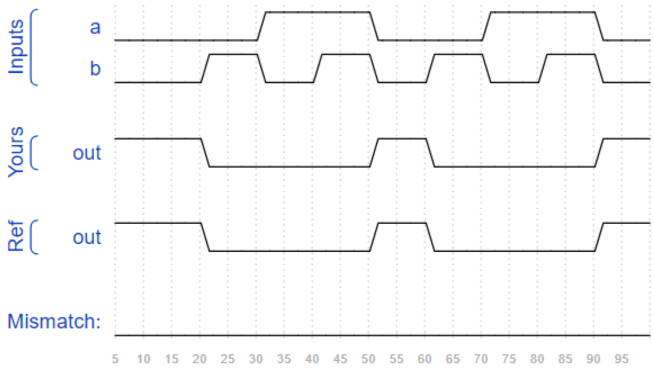
6. Create a module that implements a NOR gate. A NOR gate is an OR gate with its output inverted. A NOR function needs two operators when written in Verilog.



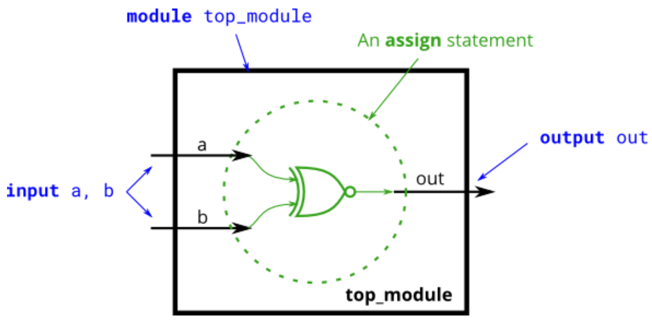
//Code

```
module top_module(
        input a,
        input b,
        output out );
        assign out= ~(a|b);
endmodule
```

//Output



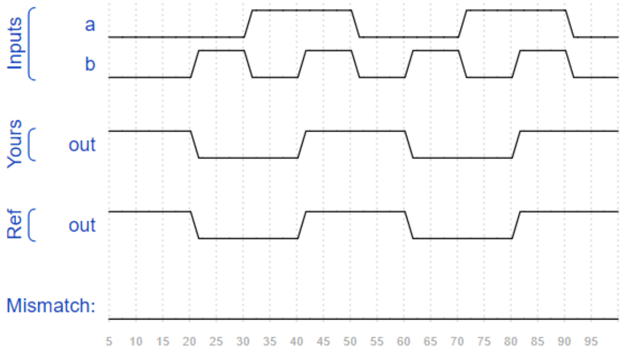
7. Create a module that implements an XNOR gate.



//Code

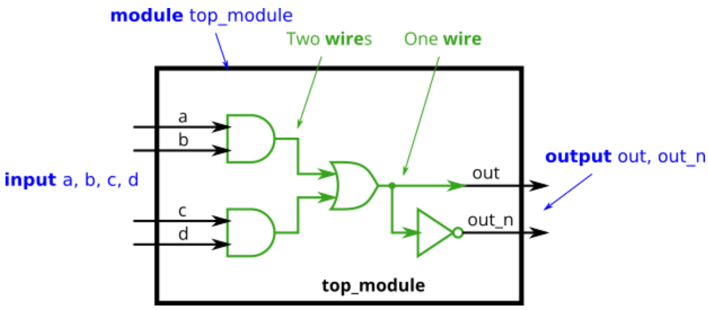
```
module top_module(
        input a,
        input b,
        output out );
        assign out=~(a^b);
endmodule
```

//Output



<u>Weak 3:</u>

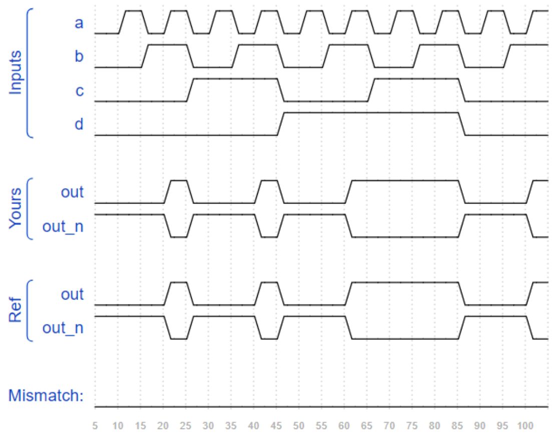
8. Implement the following circuit. Create two intermediate wires (named anything you want) to connect the AND and OR gates together. Note that the wire that feeds the NOT gate is really wire out, so you do not necessarily need to declare a third wire here. Notice how wires are driven by exactly one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



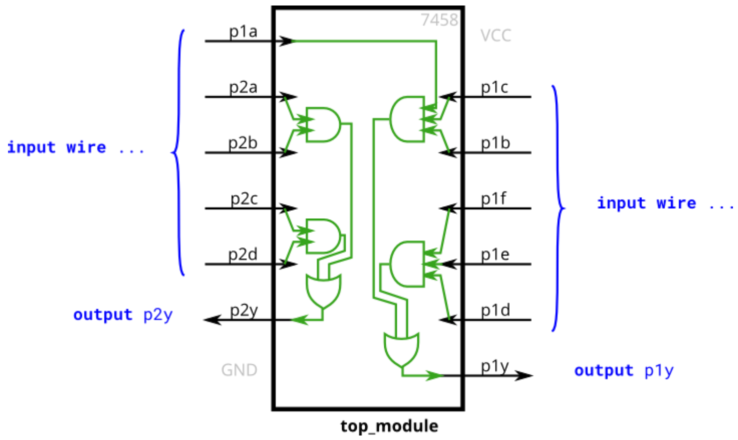
//Code

```
module top_module(
            input a,
            input b,
            input c,
            input d,
            output out,
            output out_n   );
            wire a1,a2;
         assign a1=a&b;
            assign a2=c&d;
            assign out= a1|a2;
            assign out_n=~(a1|a2);
endmodule
```

//Output



9. Create a module with the same functionality as the 7458 chip. It has 10 inputs and 2 outputs. You may choose to use an assign statement to drive each of the output wires, or you may choose to declare (four) wires for use as intermediate signals, where each internal wire is driven by the output of one of the AND gates. For extra practice, try it both ways.

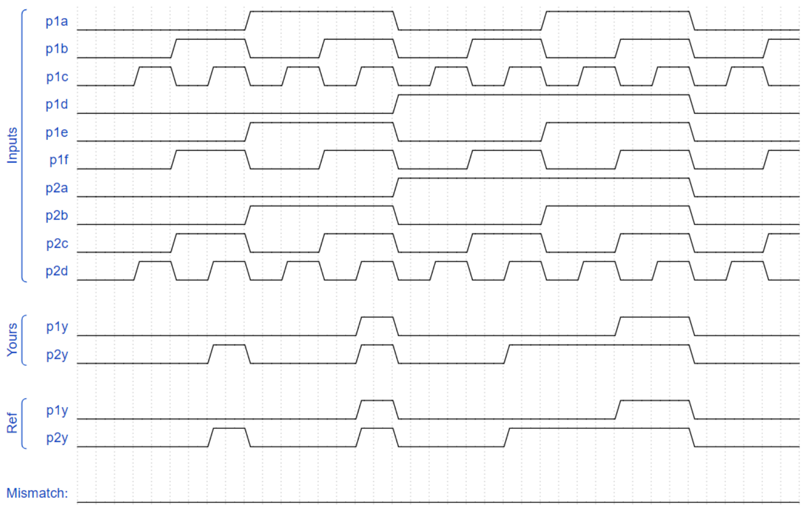


//Code

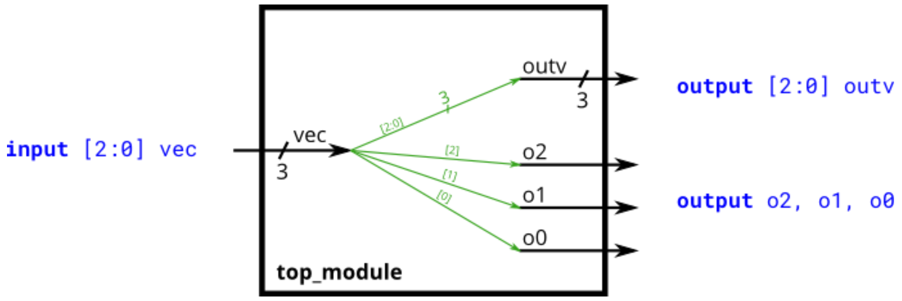
```
module top_module (
        input p1a, p1b, p1c, p1d, p1e, p1f,
        output p1y,
        input p2a, p2b, p2c, p2d,
        output p2y );
    wire a1,a2,a3,a4;
        assign a1=p2a&p2b;
    assign a2=p2c&p2d;
        assign p2y=a1|a2;
        assign a3=p1a&p1c&p1b;
```

```
        assign a4=p1f&p1e&p1d;
        assign p1y=a3|a4;
endmodule
```

//Output



10. Build a circuit that has one 3-bit input, then outputs the same vector, and also splits it into three separate 1-bit outputs. Connect output o0 to the input vector's position 0, o1 to position 1, etc. In a diagram, a tick mark with a number next to it indicates the width of the vector (or "bus"), rather than drawing a separate line for each bit in the vector.

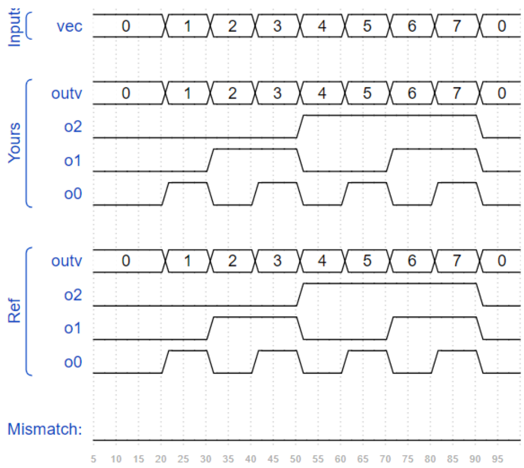


//Code

```
module top_module (

        input wire [2:0] vec,
        output wire [2:0] outv,
        output wire o2,
        output wire o1,
        output wire o0  ); // Module body starts after module declaration
     assign outv=vec;
        assign o0=vec[0];
```

```
        assign o1=vec[1];
        assign o2=vec[2];
endmodule
```

//Output



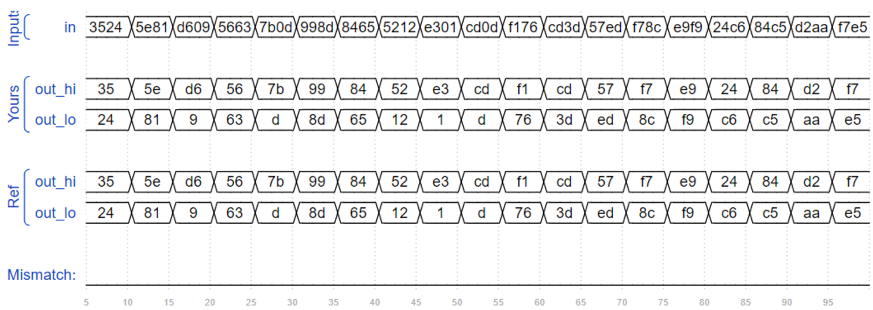
Weak 4:

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

//Code

```
module top_module(
        input wire [15:0] in,
        output wire [7:0] out_hi,
        output wire [7:0] out_lo );
        assign out_lo[7:0]=in[7:0];
        assign out_hi[7:0]=in[15:8];
endmodule
```

//Output



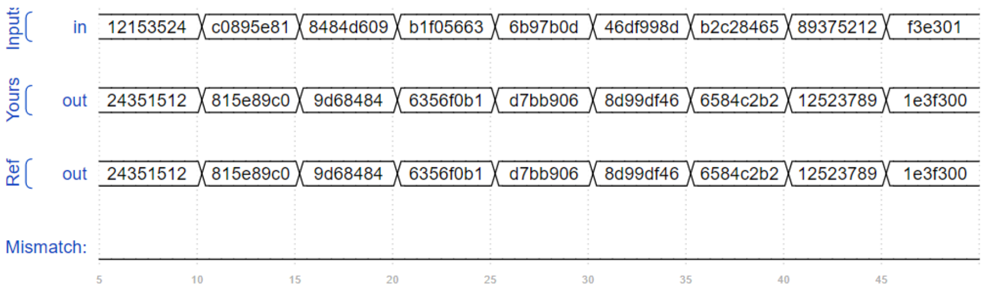
12. Build a circuit that will reverse the *byte* ordering of the 4-byte word.

AaaaaaaaBbbbbbbbCcccccccDddddddd => DdddddddCcccccccBbbbbbbbAaaaaaaa

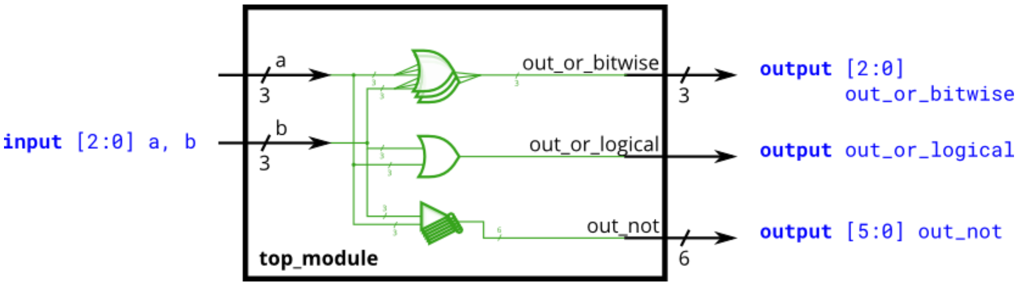
//Code

```
module top_module(
        input [31:0] in,
        output [31:0] out );//
        assign out[31:24]=in[7:0];
        assign out[23:16]=in[15:8];
        assign out[15:8]=in[23:16];
        assign out[7:0]=in[31:24];
endmodule
```

//Output



13. 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.

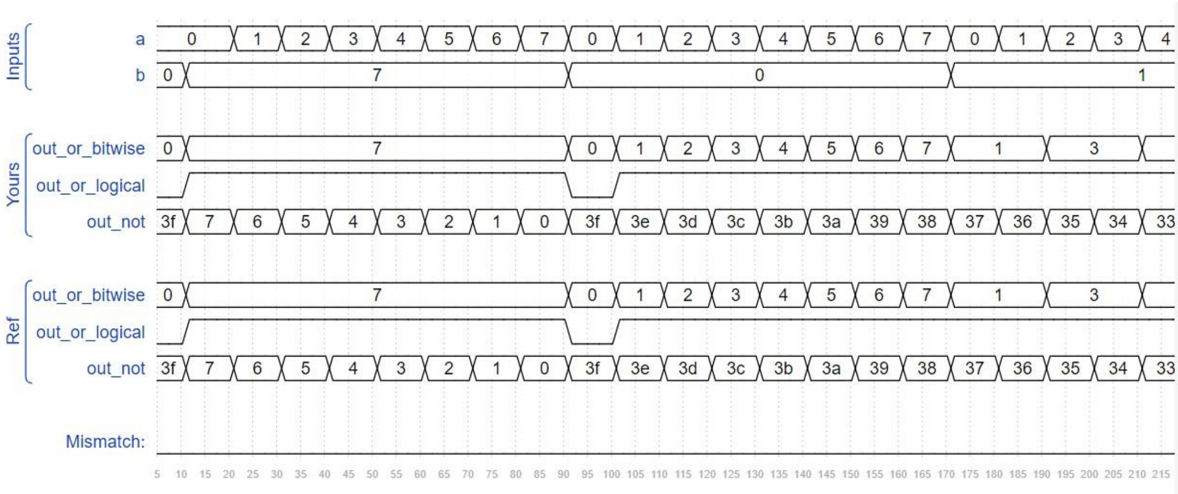


//Code

```
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={~b,~a};
endmodule
```

//Output



Weak 5:

14. Build a combinational circuit with four inputs, in[3:0].
    There are 3 outputs,

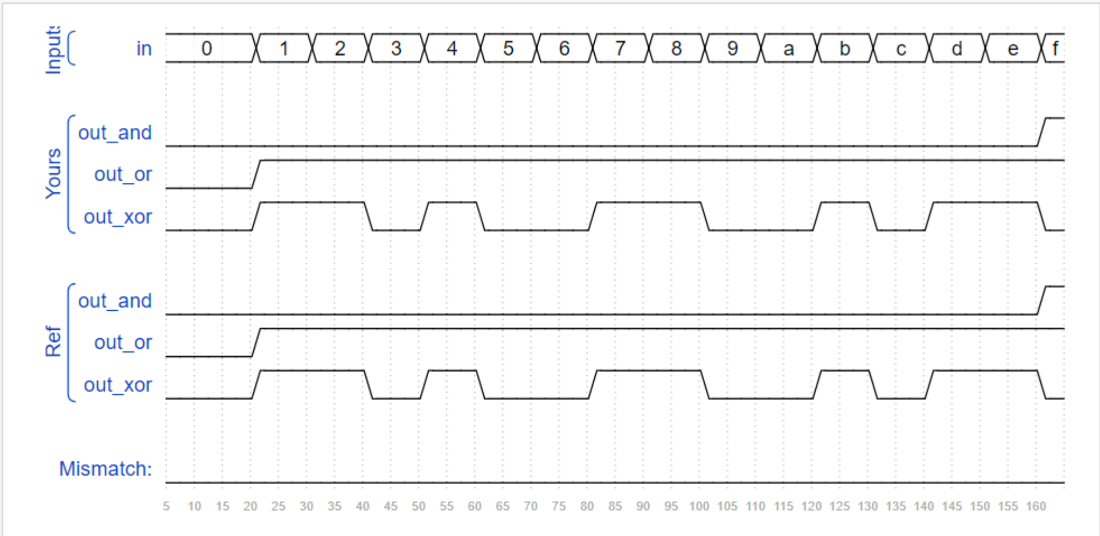
- out_and: output of a 4-input AND gate.
- out_or: output of a 4-input OR gate.
- out_xor: output of a 4-input XOR gate.

//Code

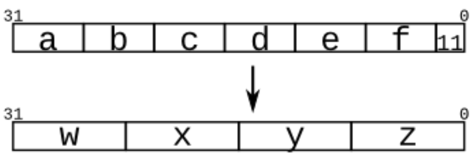
```
module top_module(
 input [3:0] in,
```

```
  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
```

//Output



15. 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:

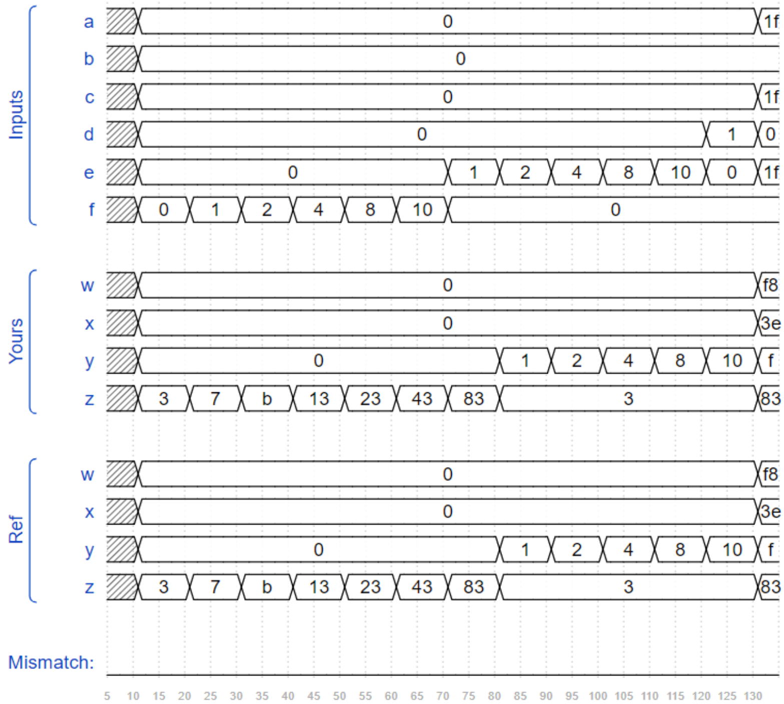


//Code

```
module top_module (

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

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

```
    assign {w,x,y,z}={a,b,c,d,e,f,2'b11};
endmodule
```

//Output

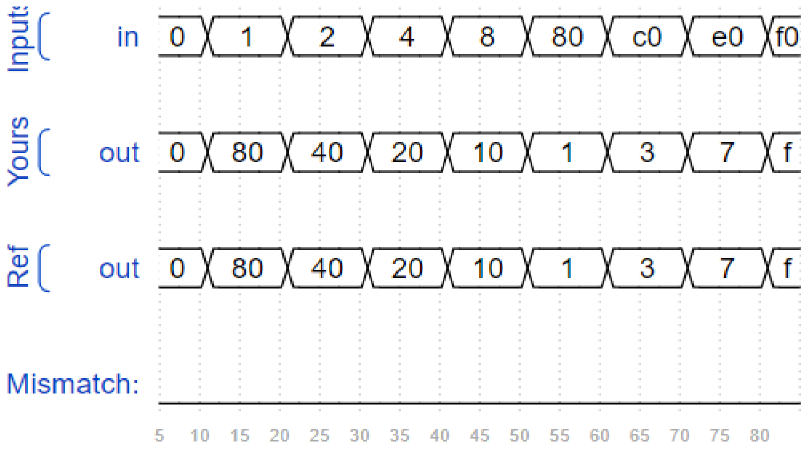


16. Given an 8-bit input vector [7:0], reverse its bit ordering.

//Code

```
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
```

//Output



17. 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.

//Code

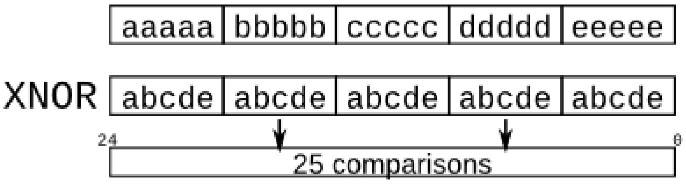
```
module top_module (
        input [7:0] in,
        output [31:0] out );
    assign out = { {24{in[7]}}, in };

endmodule
```

<u>Weak 6:</u>

18. 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;



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

//Code

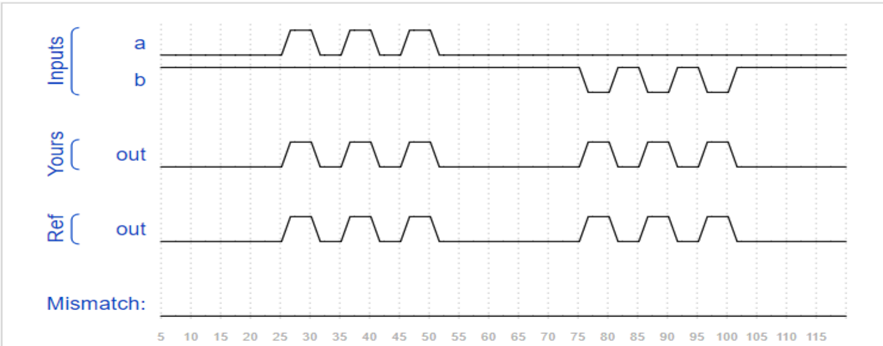
```
module top_module (
    input a, b, c, d, e,
    output [24:0] out );//
 assign out[24:20]=~{5{a}}^{a,b,c,d,e};
    assign out[19:15]=~{5{b}}^{a,b,c,d,e};
    assign out[14:10]=~{5{c}}^{a,b,c,d,e};
    assign out[9:5]=~{5{d}}^{a,b,c,d,e};
    assign out[4:0]=~{5{e}}^{a,b,c,d,e};

endmodule
```

19. In this exercise, create one *instance* of module **mod_a**, then connect the module's three pins (in1, in2, and out) to your top-level module's three ports (wires a, b, and out). The module mod_a is provided for you — you must instantiate it.You may connect signals to the module by port name or port position. For extra practice, try both methods.

//Code

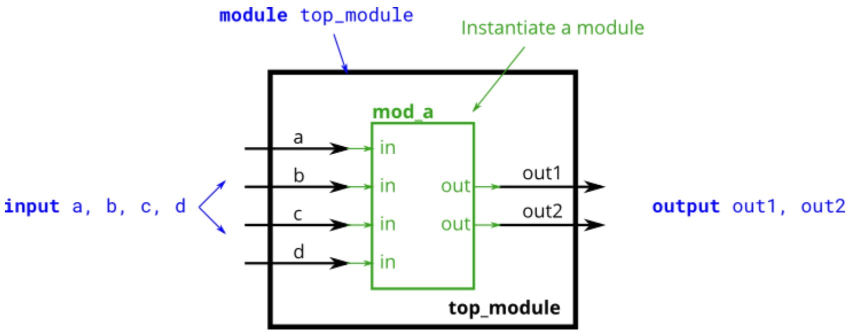
```
module top_module ( input a, input b, output out );

        mod_a inst1 (.in1(a), .in2(b), .out(out) );

endmodule
```

//Output



20. You are given a module named mod_a that has 2 outputs and 4 inputs, in that order. 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. You are given the following module:

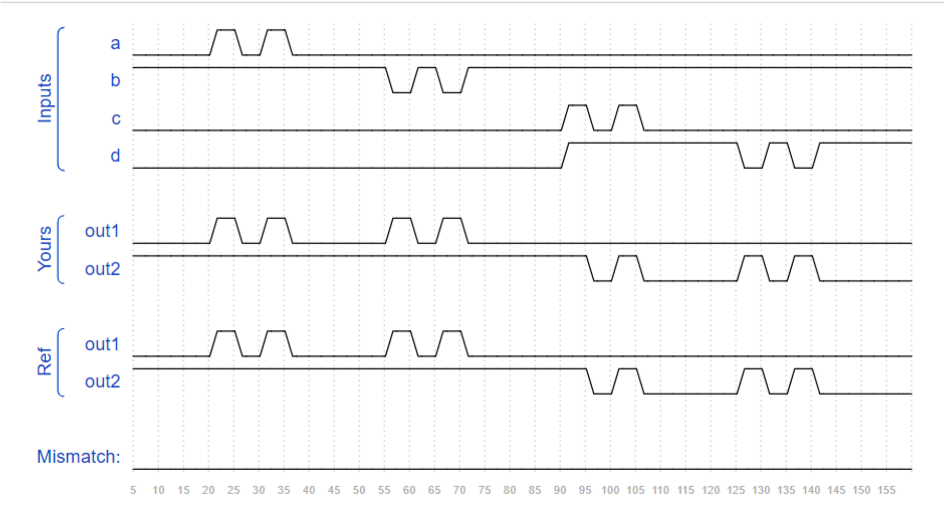
    module mod_a (output, output, input, input, input, input);



//Code

```
module top_module (
        input a,
        input b,
        input c,
        input d,
        output out1,
        output out2);
         mod_a inst1 (out1, out2, a, b, c, d);
endmodule
```

//Output



Weak 7:

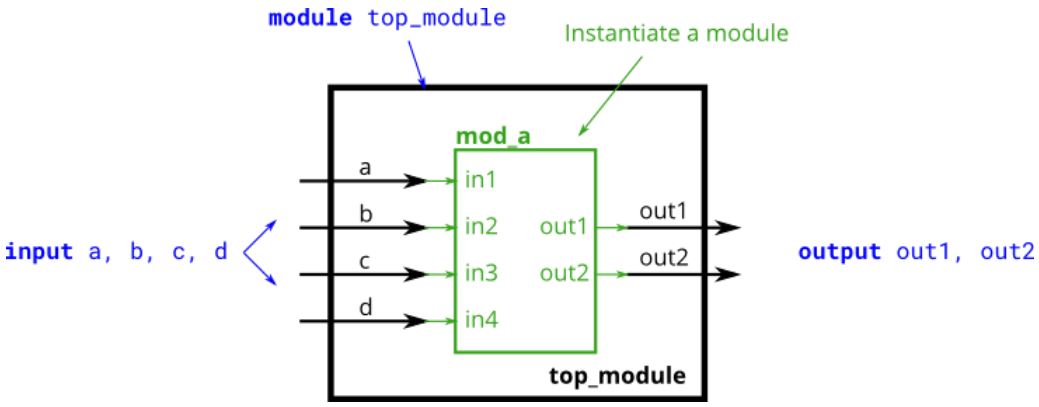
21. You are given a module named mod_a that has 2 outputs and 4 inputs, in some order. You must connect the 6 ports *by name* to your top-level module's ports:

Port in mod_a	Port in top_module
output out1	out1
output out2	out2
input in1	a
input in2	b
input in3	c
input in4	d

You are given the following module:

module mod_a (output out1, output out2, input in1, input in2, input in3, input in4);

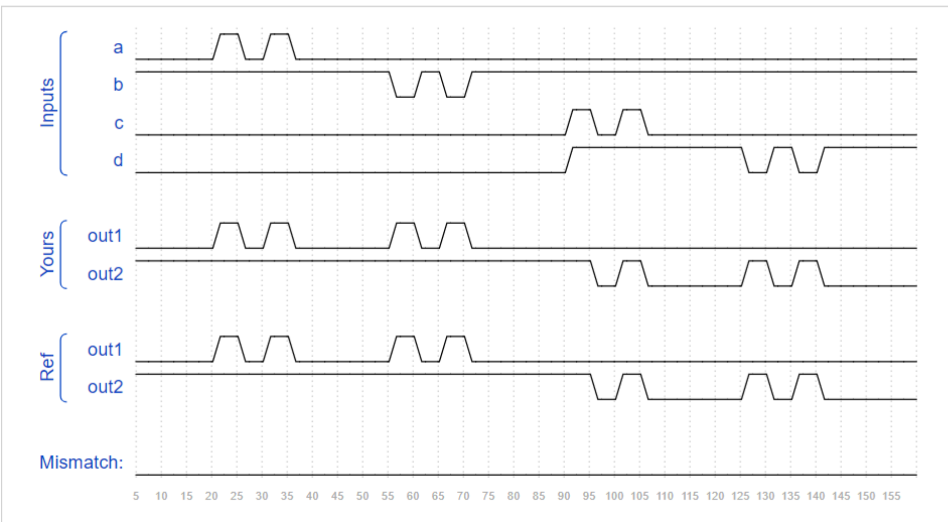
HDLBits- Verilog Coding
Bharath Shenoy



//Code

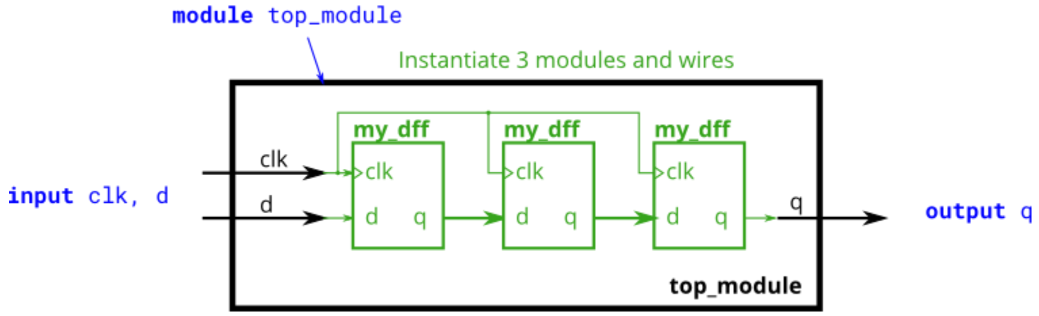
```
module top_module (
            input a,
            input b,
            input c,
            input d,
            output out1,
            output out2
         );
          mod_a inst1 (.in1(a), .in2(b), .in3(c), .in4(d), .out1(out1), .out2(out2) );

endmodule
```

//Output



22. You are given a module my_dff with two inputs and one output (that implements a D flip-flop). Instantiate three of them, then chain them together to make a shift register of length the clk port needs to be connected to all instances. The module provided to you is:

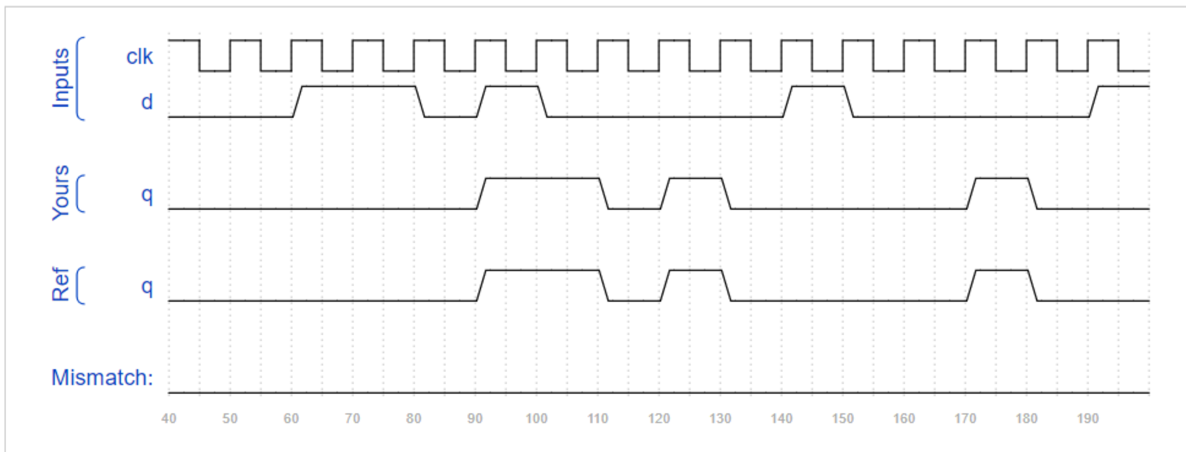
    module my_dff (input clk, input d, output q);



//Code

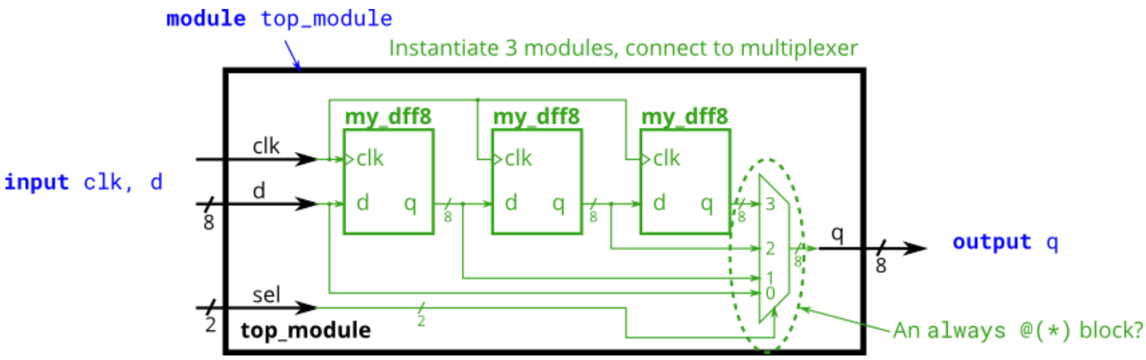
```
module top_module ( input clk, input d, output q );
 wire a, b;
   my_dff d1 ( clk, d, a );
   my_dff d2 ( clk, a, b );
   my_dff d3 ( clk, b, q );
endmodule
```

//Output



<u>Weak 8:</u>

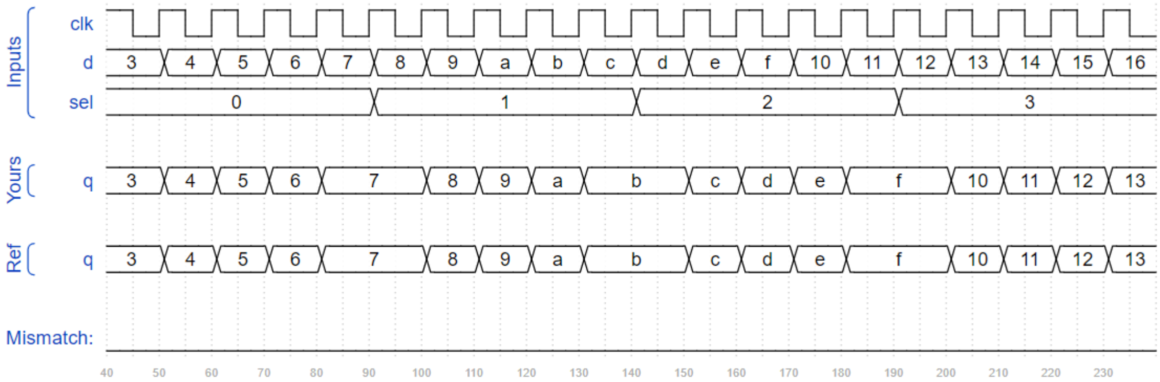
23. You are given a module my_dff8 with two inputs and one output (that implements a set of 8 D flip-flops). Instantiate three of them, then chain them together to make a 8-bit wide shift register of length 3. In addition, create a 4-to-1 multiplexer (not provided) that chooses what to output depending 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.)



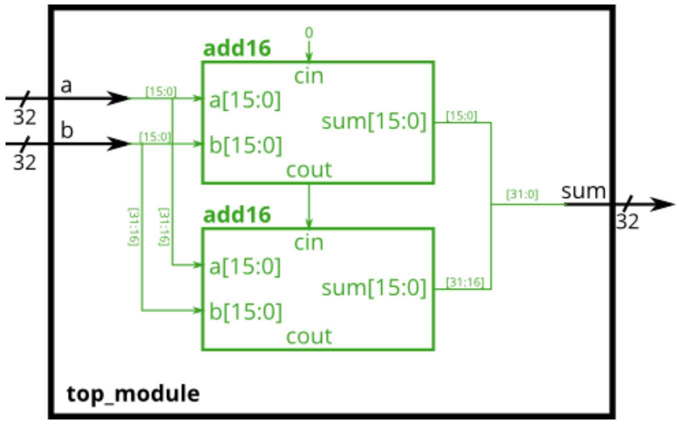
//Code

```
module top_module (
    input clk,
    input [7:0] d,
    input [1:0] sel,
    output [7:0] q
  );
  wire [7:0] o1, o2, o3;
    my_dff8 d1 ( clk, d, o1 );
    my_dff8 d2 ( clk, o1, o2 );
    my_dff8 d3 ( clk, o2, o3 );
    always @(*)
      case(sel)
        2'h0: q = d;
        2'h1: q = o1;
        2'h2: q = o2;
        2'h3: q = o3;
      endcase
endmodule
```

//Output



24. 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).

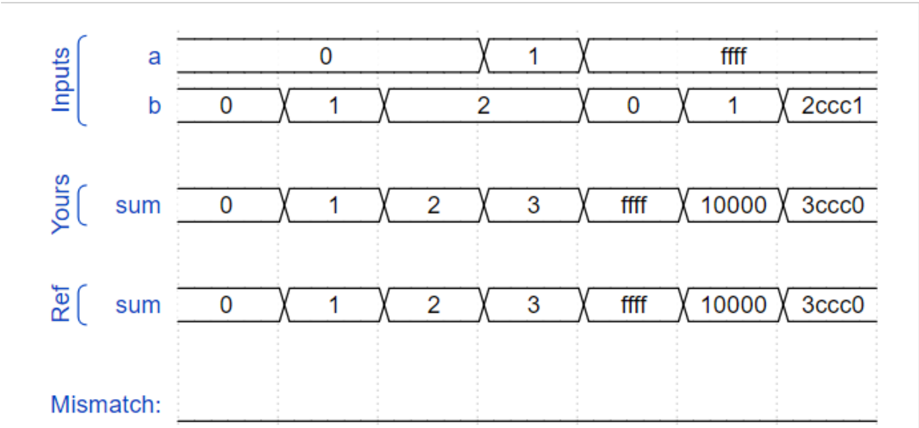


//Code

```
module top_module(
        input [31:0] a,
        input [31:0] b,
        output [31:0] sum
    );
    wire cin1,cout1,cout2;
        wire [15:0] sum1,sum2;
        assign cin1=1'b0;
```

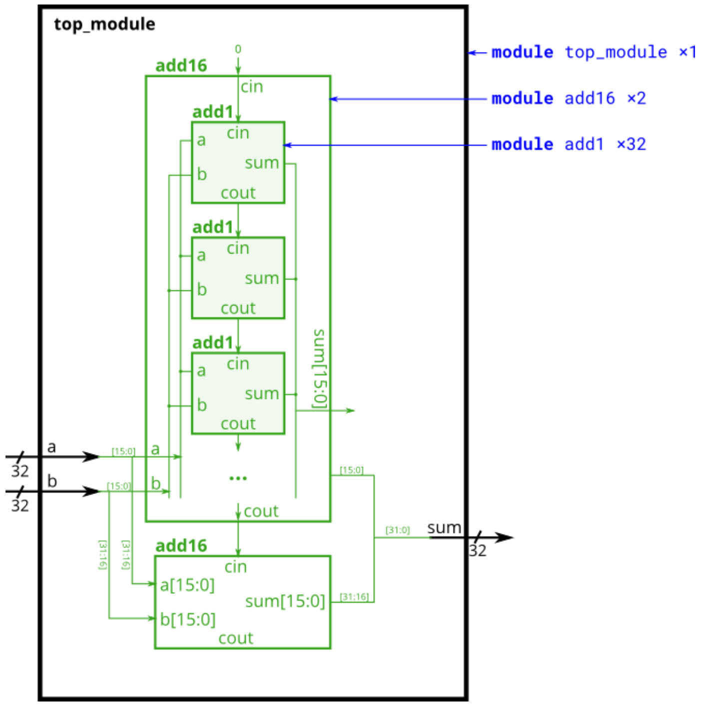
```
add16 inst1 (a[15:0],b[15:0],cin1,sum1,cout1);
add16 inst2 (a[31:16],b[31:16],cout1,sum2,cout2);
assign sum={sum2,sum1};

endmodule
```

//Output



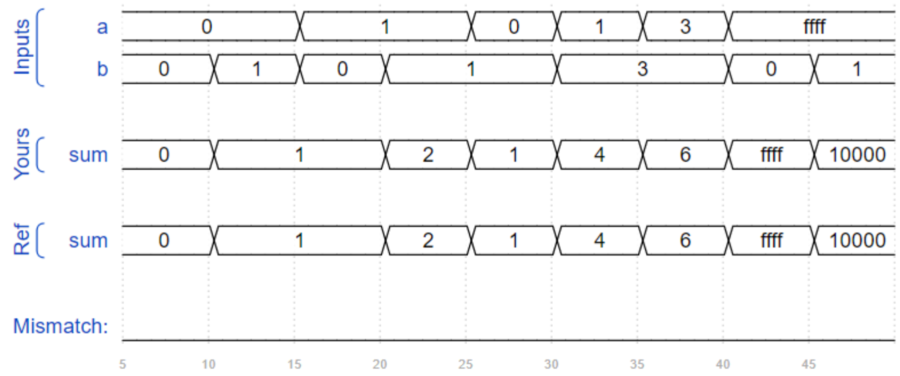
Weak 9:

25. 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).



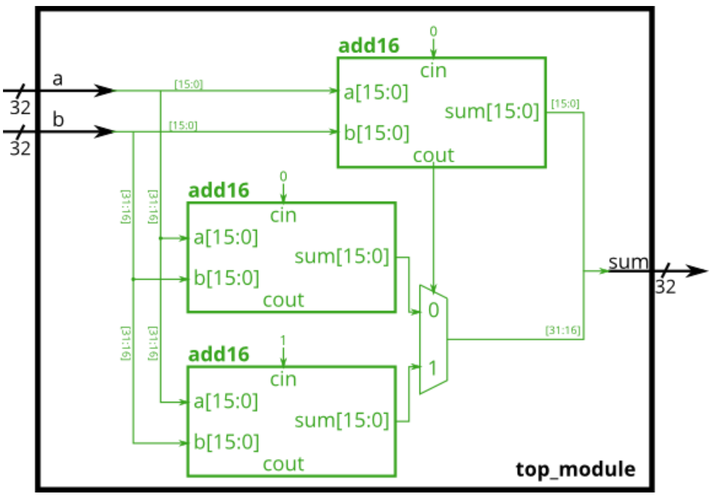

```
//Code
module top_module (
            input [31:0] a,
            input [31:0] b,
            output [31:0] sum
        );
            wire [15:0] sum1,sum2;
            wire out1,out2,in1;
            assign in1=1'b0;
            add16 inst1 (a[15:0],b[15:0],in1,sum1,out1);
            add16 inst2 (a[31:16],b[31:16],out1,sum2,out2);
            assign sum={sum2,sum1};
endmodule
module add1 ( input a, input b, input cin,   output sum, output cout );
            assign sum=a^b^cin;
            assign cout=(a|cin)&(b|cin)&(a|b);
endmodule
```

//Output



Weak 10:

26. 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.

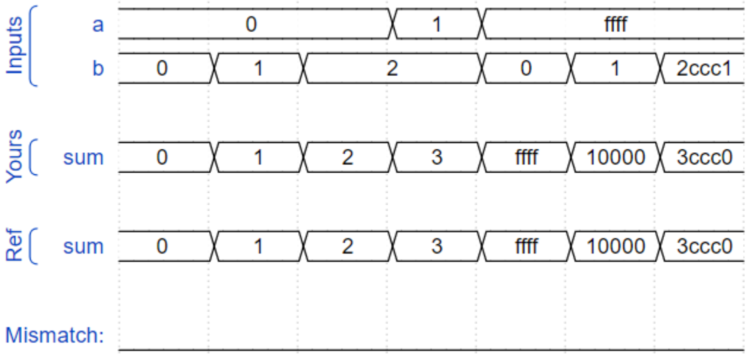


//Code

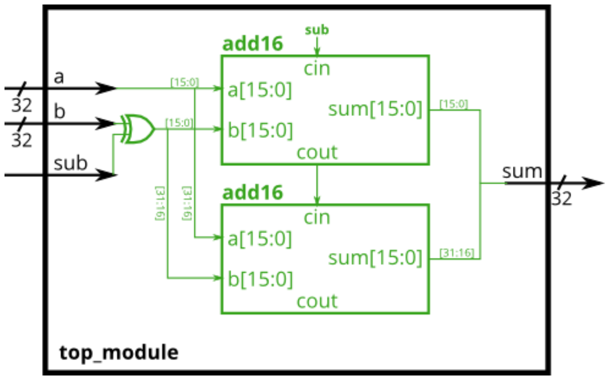
```
module top_module(
        input [31:0] a,
        input [31:0] b,
        output [31:0] sum
    );
    wire in1,in2,in3,o1,o2,o3;
        wire [15:0] sum1,sum2,sum3;
```

```
        assign in1=1'b0;
        assign in2=1'b0;
        assign in3=1'b1;
        add16 inst1 (a[15:0],b[15:0],in1,sum1,o1);
        add16 inst2 (a[31:16],b[31:16],in2,sum2,o2);
        add16 inst3 (a[31:16],b[31:16],in3,sum3,o3);
        always@(*)
          case(o1)
            1'b0:sum={sum2,sum1};
            1'b1:sum={sum3,sum1};
          endcase

endmodule
```

//Output



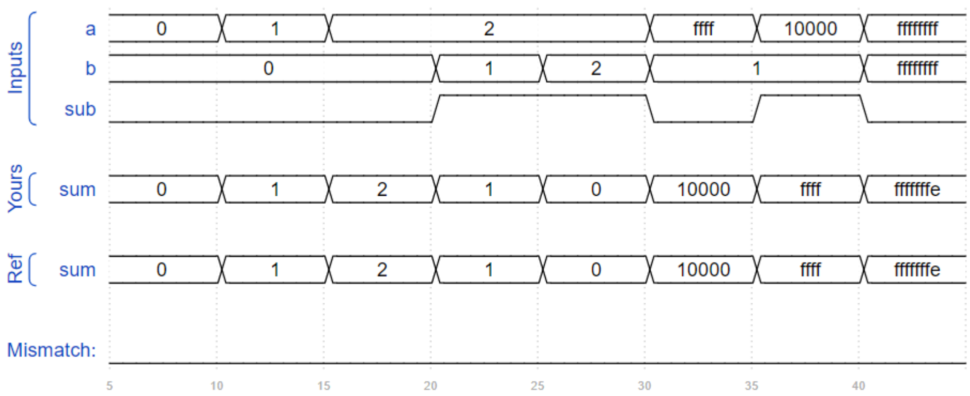
27. Build the adder-subtractor below. You are provided with a 16-bit adder module, which you need to instantiate twice:
    module add16 (input[15:0] **a**, input[15:0] **b**, input **cin**, output[15:0] **sum**, output **cout**);



//Code

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] sum
);
wire cout1,cout2;
    wire [31:0] bin;
    wire [15:0] sum1,sum2;
    assign bin={32{sub}}^b;
    add16 inst1(a[15:0],bin[15:0],sub,sum1,cout1);
    add16 inst2(a[31:16],bin[31:16],cout1,sum2,cout2);
    assign sum={sum2,sum1};
endmodule
```

//Output



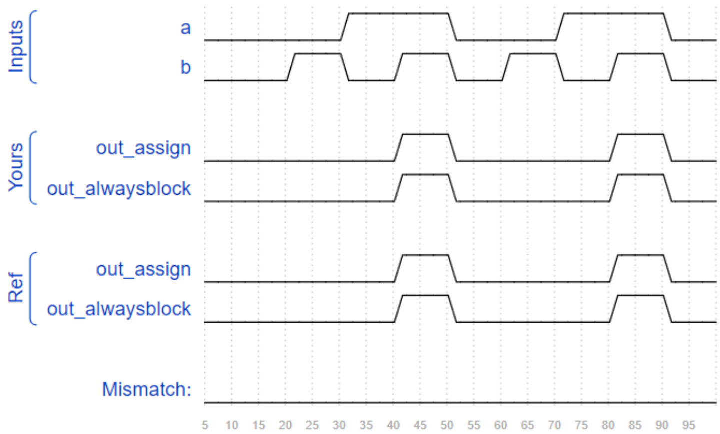
Weak 11:

28. Build an AND gate using both an assign statement and a combinational always block.

//Code

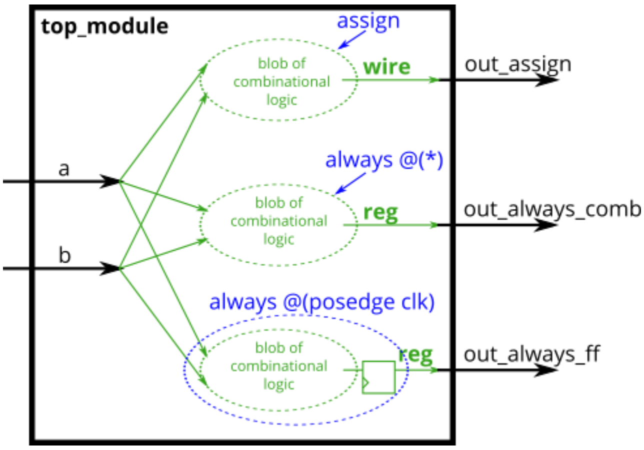
```
module top_module(
            input a,
            input b,
            output wire out_assign,
            output reg out_alwaysblock
         );
         assign out_assign = a & b;
            always @(*)
               out_alwaysblock = a & b;

endmodule
```

//Output



<u>Weak 12:</u>

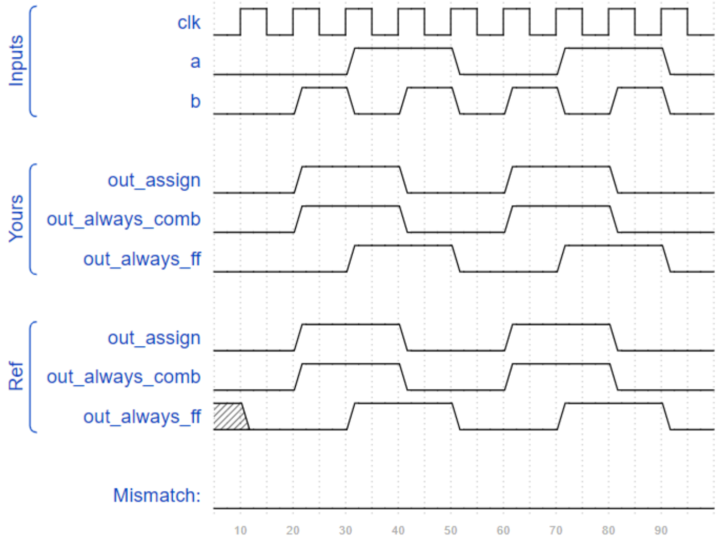
29. Build an XOR gate three ways, using an assign statement, a combinational always block, and a clocked always block. Note that the clocked always block produces a different circuit from the other two: There is a flip-flop so the output is delayed.



//Code

```
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@(*)
               out_always_comb=a^b;
            always@(posedge clk)
               out_always_ff<=a^b;

endmodule
```

//Output



30. 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.

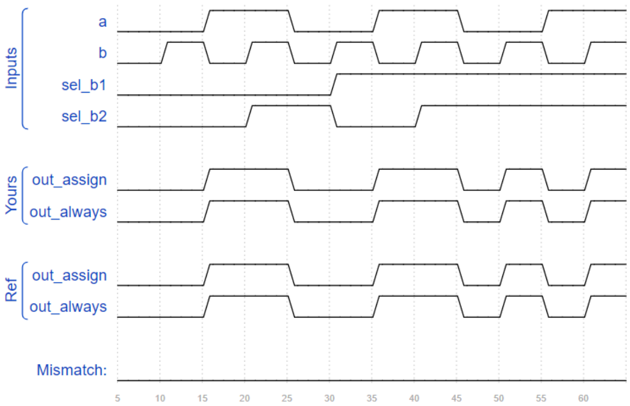
sel_b1	**sel_b2**	**out_assign** **out_always**
0	0	a
0	1	a
0	0	a
1	1	b

//Code

```
module top_module(
    input a,
    input b,
    input sel_b1,
    input sel_b2,
    output wire out_assign,
```

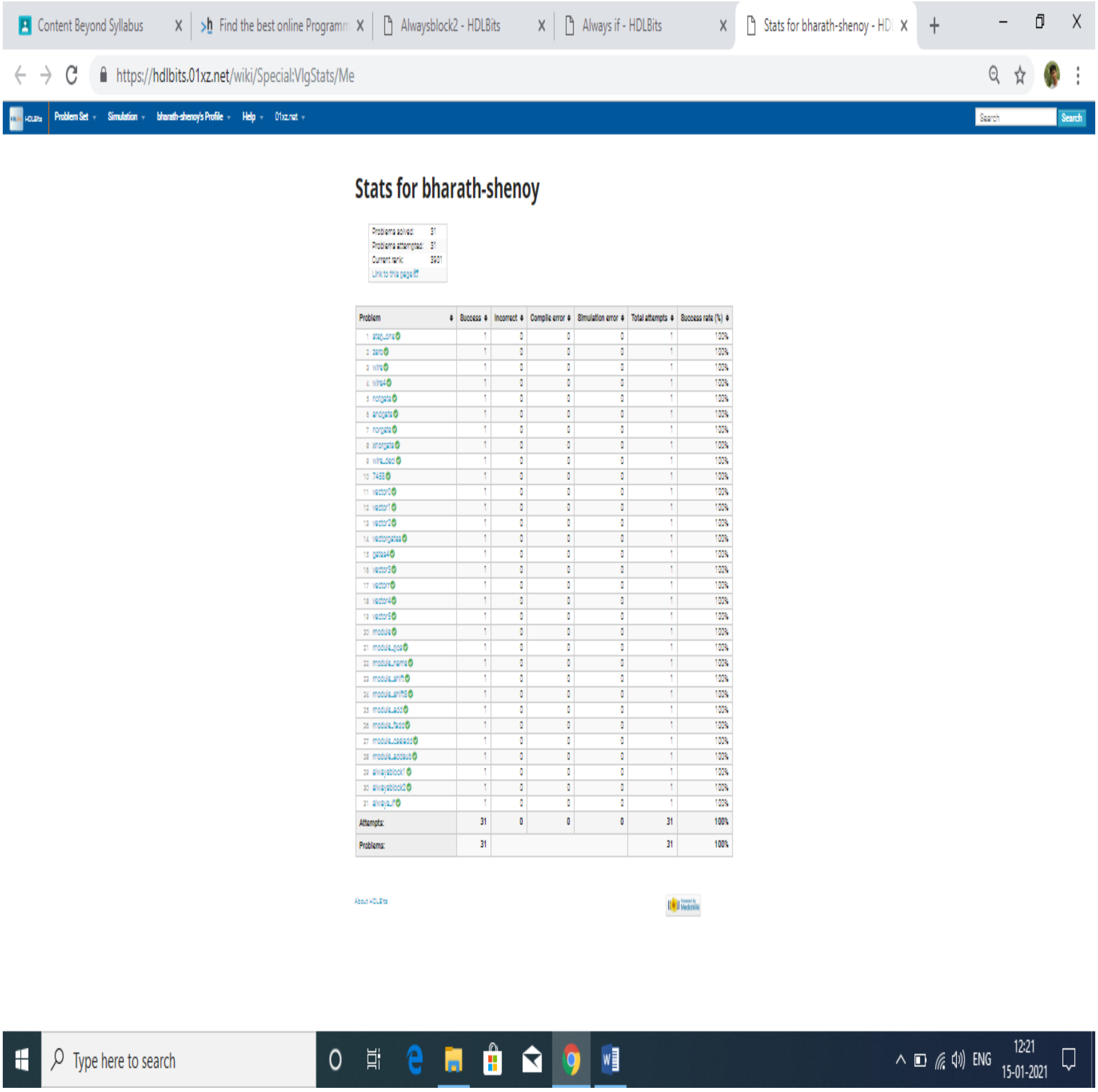
```
    output reg out_always   );
assign out_assign = (sel_b1&sel_b2) ? b : a;
    always@(*)
        if (sel_b1&sel_b2)
            out_always=b;
        else
            out_always=a;
endmodule
```

//Output



<u>Final Stats</u>:

HDLBits- Verilog Coding
Bharath Shenoy



Stats for bharath-shenoy

Problems solved:	31
Problems attempted:	31
Current rank:	3901
Link to this page	

Problem	Success	Incorrect	Compile error	Simulation error	Total attempts	Success rate (%)
1 step_one	1	0	0	0	1	100%
2 zero	1	0	0	0	1	100%
3 wire	1	0	0	0	1	100%
4 wire4	1	0	0	0	1	100%
5 notgate	1	0	0	0	1	100%
6 andgate	1	0	0	0	1	100%
7 norgate	1	0	0	0	1	100%
8 xnorgate	1	0	0	0	1	100%
9 wire_decl	1	0	0	0	1	100%
10 7458	1	0	0	0	1	100%
11 vector0	1	0	0	0	1	100%
12 vector1	1	0	0	0	1	100%
13 vector2	1	0	0	0	1	100%
14 vectorgates	1	0	0	0	1	100%
15 gates4	1	0	0	0	1	100%
16 vector3	1	0	0	0	1	100%
17 vectorr	1	0	0	0	1	100%
18 vector4	1	0	0	0	1	100%
19 vector5	1	0	0	0	1	100%
20 module	1	0	0	0	1	100%
21 module_pos	1	0	0	0	1	100%
22 module_name	1	0	0	0	1	100%
23 module_shift	1	0	0	0	1	100%
24 module_shift8	1	0	0	0	1	100%
25 module_add	1	0	0	0	1	100%
26 module_fadd	1	0	0	0	1	100%
27 module_cseladd	1	0	0	0	1	100%
28 module_addsub	1	0	0	0	1	100%
29 alwaysblock1	1	0	0	0	1	100%
30 alwaysblock2	1	0	0	0	1	100%
31 always_if	1	0	0	0	1	100%
Attempts:	31	0	0	0	31	100%
Problems:	31				31	100%

About HDLBits