HDLbits resolution

Date Created	@January 5, 2022 12:43 PM

▼ Getting Started

▼ Getting Started

```
module top_module( output one );

  assign one = 1'b1;

endmodule
```

▼ Output Zero

```
module top_module ( output zero );

  assign zero = 1'b0;

endmodule
```

▼ Verilog Language

▼ Basics

▼ Simple wire

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

  assign out = in;
  // Note that wires are directional, so "assign in = out" is not equivalent.

endmodule
```

▼ Four wires

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

  assign w = a;
  assign x = b;
  assign y = b;
  assign z = c;

  // If we're certain about the width of each signal, using
  // the concatenation operator is equivalent and shorter:
  // assign {w,x,y,z} = {a,b,b,c};

endmodule
```

▼ Inverter

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

```
  assign out = ~in;

endmodule
```

▼ AND gate

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

    assign out = a & b;
endmodule
```

▼ NOR gate

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

    assign out = ~(a | b);

endmodule
```

▼ XNOR gate

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

    assign out = ~ a^b;

endmodule
```

▼ Declaring wire

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

  wire w1, w2;    // Declare two wires (named w1 and w2)
  assign w1 = a&b;  // First AND gate
  assign w2 = c&d;  // Second AND gate
  assign out = w1|w2; // OR gate: Feeds both 'out' and the NOT gate

  assign out_n = ~out;  // NOT gate

endmodule
```

▼ 7458 chip

```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y );

    assign p1y = (p1a & p1b & p1c)|(p1d & p1e & p1f);
    assign p2y = (p2a & p2b)|(p2c & p2d);

endmodule
```

▼ Vectors

▼ Vectors

```
module top_module(
  input [2:0] vec,
  output [2:0] outv,
  output o2,
  output o1,
  output o0
);

  assign outv = vec;

  // This is ok too: assign {o2, o1, o0} = vec;
  assign o0 = vec[0];
  assign o1 = vec[1];
  assign o2 = vec[2];

endmodule
```

▼ Vectors in more detail

```
module top_module (
  input [15:0] in,
  output [7:0] out_hi,
  output [7:0] out_lo
);

  assign out_hi = in[15:8];
  assign out_lo = in[7:0];

  // Concatenation operator also works: assign {out_hi, out_lo} = in;

endmodule
```

▼ Vector part seledt

```
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
```

▼ Biwise operators

```
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[2:0] = ~a; // Part-select on left side is o.
  assign out_not[5:3] = ~b; //Assigning to [5:3] does not conflict with [2:0]

endmodule
```

▼ Four-input gates

```
//solution 1
module top_module(
    input [3:0] in,
    output out_and,
    output out_or,
    output out_xor
);

    assign out_and = in[0]&in[1]&in[2]&in[3];
    assign out_or = in[0]|in[1]|in[2]|in[3];
    assign out_xor = in[0]^in[1]^in[2]^in[3];

endmodule

//solution 2
module top_module(
    input [3:0] in,
    output out_and,
    output out_or,
    output out_xor
);

    assign out_and = &in;//reduction operator
    assign out_or = |in;
    assign out_xor = ^in;

endmodule
```

▼ Vector concatenation operator

```
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};
    // assign { ... } = { ... };

endmodule
```

▼ Vector reversal 1

```
//solution 1
module top_module (
  input [7:0] in,
  output [7:0] out
);

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

endmodule

//solution 2
module top_module (
  input [7:0] in,
  output [7:0] out
);

  always @(*) begin
    for (int i=0; i<8; i++) // int is a SystemVerilog type. Use integer for pure Verilog.
      out[i] = in[8-i-1];
  end

endmodule

//solution 3
module top_module (
  input [7:0] in,
  output [7:0] out
);

  generate
```

```
    genvar i;
    for (i=0; i<8; i = i+1) begin: my_block_name
      assign out[i] = in[8-i-1];
    end
  endgenerate

endmodule
```

▼ Replication operator

```
module top_module (
  input [7:0] in,
  output [31:0] out
);

  // Concatenate two things together:
  // 1: {in[7]} repeated 24 times (24 bits)
  // 2: in[7:0] (8 bits)
  assign out = { {24{in[7]}}, in };//notice the braces

endmodule
```

▼ More replication

```
module top_module (
  input a, b, c, d, e,
  output [24:0] out
);

  wire [24:0] top, bottom;
  assign top    = { {5{a}}, {5{b}}, {5{c}}, {5{d}}, {5{e}} };
  assign bottom = {5{a,b,c,d,e}};
  assign out = ~top ^ bottom; // Bitwise XNOR

  // This could be done on one line:
  // assign out = ~{ {5{a}}, {5{b}}, {5{c}}, {5{d}}, {5{e}} } ^ {5{a,b,c,d,e}};

endmodule
```

▼ Modules: Hierarchy

▼ Modules

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

  // Create an instance of "mod_a" named "inst1", and connect ports by name:
  mod_a inst1 (
    .in1(a),  // Port"in1"connects to wire "a"
    .in2(b),  // Port "in2" connects to wire "b"
    .out(out) // Port "out" connects to wire "out"
        // (Note: mod_a's port "out" is not related to top_module's wire "out".
        // It is simply coincidence that they have the same name)
  );

/*
  // Create an instance of "mod_a" named "inst2", and connect ports by position:
  mod_a inst2 ( a, b, out );  // The three wires are connected to ports in1, in2, and out, respectively.
*/

endmodule
```

▼ Connecting ports by position

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

    mod_a instance1 ( out1, out2, a, b, c, d );

endmodule
```

▼ Connecting ports by name

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

    mod_a instance1 ( .out1(out1), .out2(out2), .in1(a), .in2(b), .in3(c), .in4(d) );

endmodule
```

▼ Three modules

```
module top_module (
  input clk,
  input d,
  output q
);

  wire a, b;  // Create two wires. I called them a and b.

  // Create three instances of my_dff, with three different instance names (d1, d2, and d3).
  // Connect ports by position: ( input clk, input d, output q)
  my_dff d1 ( clk, d, a );
  my_dff d2 ( clk, a, b );
  my_dff d3 ( clk, b, q );

endmodule
```

▼ Adder 1

```
module top_module (
    input clk,
    input [7:0] d,
    input [1:0] sel,
    output [7:0] q
);
    wire [7:0] out1,out2,out3;
    my_dff8 instance1 (clk,d,out1);
    my_dff8 instance2 (clk,out1,out2);
    my_dff8 instance3 (clk,out2,out3);

    always @ (*)
    begin
        case (sel)
            2'b00: q = d;
            2'b01: q = out1;
            2'b10: q = out2;
            2'b11: q = out3;
        endcase
    end

endmodule
```

▼ Adder 2

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);
    wire [15:0] low16,high16;
    wire cout1,cout2;

    add16 add16_1 (a[15:0],b[15:0],1'b0,low16,cout1);
    add16 add16_2 (a[31:16],b[31:16],cout1,high16,cout2);

    assign sum = {high16,low16};

endmodule

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

    assign {cout,sum} = a + b + cin;
// Full adder module here

endmodul
```

▼ Carry-select adder

```
odule top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);

    wire [15:0] sumlow,sumhigh0,sumhigh1;
    wire coutlow,couthigh0,couthigh1;

    add16 add16low (a[15:0],b[15:0],1'b0,sumlow,coutlow);
    add16 add16high0 (a[31:16],b[31:16],1'b0,sumhigh0,couthigh0);
    add16 add16high1 (a[31:16],b[31:16],1'b1,sumhigh1,couthigh1);

    always @ (*)
    begin
        case (coutlow)
            1'b0: sum = {sumhigh0,sumlow};
            1'b1: sum = {sumhigh1,sumlow};
        endcase
    end

endmodule
```

▼ Adder-subtractor

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] sum
);

    wire [31:0] xb;
    wire [15:0] sumlow,sumhigh;
    wire coutlow,couthigh;

    add16 add16low (a[15:0],xb[15:0],sub,sumlow,coutlow);
    add16 add16high (a[31:16],xb[31:16],coutlow,sumhigh,couthigh);

    assign xb = b ^ {32{sub}};
    assign sum = {sumhigh,sumlow};


endmoduledule
```

▼ Procedures

▼ Always blocks(combinational)

```
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
```

▼ Always blocks(clocked)

```
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
```

▼ If statement

```
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 @ (*)
    begin
        if (sel_b1 && sel_b2) begin
            out_always <= b ;
        end
        else begin
            out_always <= a ;
        end
    end


endmodule
```

▼ If statement latches

```
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;
```

```
    end

    always @(*) begin
        if (~arrived)
           keep_driving = ~gas_tank_empty;//
        else
           keep_driving = 0;
    end

endmodule
```

▼ Case statement

```
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  // This is a combinational circuit
        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 = 3'b0000;
        endcase
    end

endmodule
```

▼ Priority encoder

```
//solution 1
module top_module (
    input [3:0] in,
    output reg [1:0] pos  );

    always @ (*)begin
        casex (in)
            4'bxxx1: pos = 2'd0;
            4'bxx10: pos = 2'd1;
            4'bx100: pos = 2'd2;
            4'b1000: pos = 2'd3;
            default: pos = 2'd0;
        endcase
    end

endmodule

//solution 2
module top_module (
    input [3:0] in,
    output reg [1:0] pos  );

    always @ (*)begin
        casez (in)
            4'b???1: pos = 2'd0;//按照case项的顺序比较，若匹配上了就不再比较
            4'b??10: pos = 2'd1;
            4'b?100: pos = 2'd2;
            4'b1000: pos = 2'd3;
            default: pos = 2'd0;
        endcase
    end

endmodule

//solution 3
module top_module (
```

```
  input [3:0] in,
  output reg [1:0] pos
);

  always @(*) begin     // Combinational always block
    case (in)
      4'h0: pos = 2'h0; // I like hexadecimal because it saves typing.
      4'h1: pos = 2'h0;
      4'h2: pos = 2'h1;
      4'h3: pos = 2'h0;
      4'h4: pos = 2'h2;
      4'h5: pos = 2'h0;
      4'h6: pos = 2'h1;
      4'h7: pos = 2'h0;
      4'h8: pos = 2'h3;
      4'h9: pos = 2'h0;
      4'ha: pos = 2'h1;
      4'hb: pos = 2'h0;
      4'hc: pos = 2'h2;
      4'hd: pos = 2'h0;
      4'he: pos = 2'h1;
      4'hf: pos = 2'h0;
      default: pos = 2'b0;  // Default case is not strictly necessary because all 16 combinations are covered.
    endcase
  end

  // There is an easier way to code this. See the next problem (always_casez).

endmodule
```

▼ Priority encoder with casez

```
module top_module (
    input [7:0] in,
    output reg [2:0] pos  );

    always @(*)begin
        casez(in)
            8'b???????1: pos = 3'd0;
            8'b??????10: pos = 3'd1;
            8'b?????100: pos = 3'd2;
            8'b????1000: pos = 3'd3;
            8'b???10000: pos = 3'd4;
            8'b??100000: pos = 3'd5;
            8'b?1000000: pos = 3'd6;
            8'b10000000: pos = 3'd7;
            default: pos = 3'd0;
        endcase
    end
endmodule
```

▼ Avoiding latches

```
module top_module (
    input [15:0] scancode,
    output reg left,
    output reg down,
    output reg right,
    output reg up  );

    always @(*)begin
        left = 1'b0;right = 1'b0;down = 1'b0;up = 1'b0;
        case(scancode)
            16'he06b: left = 1'b1;
      16'he072: down = 1'b1;
      16'he074: right = 1'b1;
      16'he075: up = 1'b1;
        endcase
    end
endmodule
```

▼ More Verilog Features

▼ Conditional ternary operator

```
module top_module (
    input [7:0] a, b, c, d,
    output [7:0] min);//

    wire[7:0] abmin,cdmin;

    assign abmin = (a < b)? a:b;
    assign cdmin = (c < d)? c:d;
    assign min = (abmin < cdmin)? abmin:cdmin;

    // assign intermediate_result1 = compare? true: false;

endmodule
```

▼ Reduction operators

```
module top_module (
    input [7:0] in,
    output parity);

    assign parity = ^in;
endmodule
```

▼ Reduction: Even wider gates

```
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
```

▼ Combinational for-loop: Vector reversal 2

```
//solution 1
module top_module(
    input [99:0] in,
    output [99:0] out
);
    always@(*)begin
        for(integer i = 0;i<100;i=i+1)
            out[i] = in[99-i];
    end

endmodule

//solution 2
module top_module (
  input [99:0] in,
  output reg [99:0] out
);

  always @(*) begin
    for (int i=0;i<$bits(out);i++)    // $bits() is a system function that returns the width of a signal.
      out[i] = in[$bits(out)-i-1];  // $bits(out) is 100 because out is 100 bits wide.
  end

endmodule

//solution 3
module top_module (
  input [99:0] in,
  output reg [99:0] out
```

```
);

    genvar i;//生成generate中的循环变量
    generate for (i=0;i<$bits(out);i=i+1)
        begin:Go//begin_end和命名一定要有
            assign out[i] = in[$bits(out)-1-i];//此处要用assign语句
        end
    endgenerate

endmodule
```

▼ Combinational for-loop: 255-bit population count

```
module top_module(
    input [254:0] in,
    output [7:0] out );

    always@(*)begin
        out = 8'd0;
        for(int i=0;i<$bits(in);i=i+1)
            out = out + in[i];
    end

endmodule
```

▼ Generate for-loop: 100-bit binary adder 2

```
module top_module(
    input [99:0] a, b,
    input cin,
    output [99:0] cout,
    output [99:0] sum );

    always @(*)begin
        {cout[0],sum[0]} = a[0] + b[0] + cin;//全加器
        for (int i=1;i<$bits(a);i=i+1)
        {cout[i],sum[i]} = a[i] + b[i] + cout[i-1];
    end

endmodule
```

▼ Generate for-loop: 100-digit BCD adder

```
module top_module(
    input [399:0] a, b,
    input cin,
    output cout,
    output [399:0] sum );

    wire [400:0] midcout;

    assign cout = midcout[400];

    bcd_fadd bcd_fadd_0(.a(a[3:0]), .b(b[3:0]), .cin(cin), .cout(midcout[4]), .sum(sum[3:0]));

    generate//子模块不可在always模块内部调用，可以生成模块重复调用
        genvar i;
        for(i=4;i<$bits(a);i=i+4)
            begin:Go
                bcd_fadd bcd_fadd_i(.a(a[i+3:i]), .b(b[i+3:i]), .cin(midcout[i]), .cout(midcout[i+4]), .sum(sum[i+3:i]));
            end
    endgenerate

endmodule
```

▼ Circuits

▼ Combinational Logics

▼ Basic Gates

▼ Wire

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

    assign out = in;

endmodule
```

▼ GND

```
module top_module (
    output out);

    assign out = 1'b0;

endmodule
```

▼ NOR

```
module top_module (
    input in1,
    input in2,
    output out);

    assign out = ~(in1|in2);

endmodule
```

▼ Another gate

```
module top_module (
    input in1,
    input in2,
    output out);

    assign out = in1 & (!in2);

endmodule
```

▼ Two gates

```
module top_module (
    input in1,
    input in2,
    input in3,
    output out);

    assign out = ~(in1^in2) ^ in3;

endmodule
```

▼ More logic gates

```
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
```

▼ 7420 chip

```
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
```

▼ Truth table

```
module top_module (
  input x3,
  input x2,
  input x1,
  output f
);
  // This truth table has four minterms.
  assign f = ( ~x3 & x2 & ~x1 ) |
        ( ~x3 & x2 & x1 ) |
        ( x3 & ~x2 & x1 ) |
        ( x3 & x2 & x1 ) ;

  // It can be simplified, by boolean algebra or Karnaugh maps.
  // assign f = (~x3 & x2) | (x3 & x1);

  // You may then notice that this is actually a 2-to-1 mux, selected by x3:
  // assign f = x3 ? x1 : x2;

endmodule
```

▼ Two-bit equality

```
module top_module(
  input [1:0] A,
  input [1:0] B,
  output z);

  assign z = (A[1:0]==B[1:0]);  // Comparisons produce a 1 or 0 result.

  // Another option is to use a 16-entry truth table ( {A,B} is 4 bits, with 16 combinations ).
  // There are 4 rows with a 1 result.  0000, 0101, 1010, and 1111.

endmodule
```

▼ Simple circuit A

```
module top_module (input x, input y, output z);

    assign z = (x^y) & x;

endmodule
```

▼ Simple circuit B

```
module top_module ( input x, input y, output z );

    assign z = ~x^y;

endmodule
```

▼ Combine circuits A and B

```
module top_module (input x, input y, output z);

    wire z1,z2,z3,z4;


    A_module IA1(x,y,z1);
    B_module IB1(x,y,z2);
    A_module IA2(x,y,z3);
    B_module IB2(x,y,z4);

    assign z = (z1 | z2) ^ (z3 & z4);

endmodule

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

    assign z = (x^y) & x;

endmodule

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

    assign z = ~x^y;

endmodule
```

▼ Ring or vibrate?

```
module top_module (
    input ring,
    input vibrate_mode,
    output ringer,       // Make sound
    output motor         // Vibrate
);

    assign ringer = ~vibrate_mode & ring;//list truth table
    assign motor = vibrate_mode & ring;

endmodule
```

▼ Thermostat

```
module top_module (
    input too_cold,
    input too_hot,
    input mode,
    input fan_on,
    output heater,
    output aircon,
    output fan
);

    assign fan = fan_on | (mode & too_cold) | (~mode & too_hot);//truth table
    assign heater = mode & too_cold;
    assign aircon = ~mode & too_hot;

endmodule
```

▼ 3-bit population count

```
module top_module(
    input [2:0] in,
    output [1:0] out );

    assign out = in[0] + in[1] + in[2];

endmodule
```

▼ Gates and vectors

```
//solution 1
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[3:2],&in[2:1],&in[1:0]};
    assign out_any = {|in[3:2],|in[2:1],|in[1:0]};
    assign out_different = {in[0]^in[3],^in[3:2],^in[2:1],^in[1:0]};

endmodule

//solution 2
module top_module (
  input [3:0] in,
  output [2:0] out_both,
  output [3:1] out_any,
  output [3:0] out_different
);

  // Use bitwise operators and part-select to do the entire calculation in one line of code
  // in[3:1] is this vector:             in[3]  in[2]  in[1]
  // in[2:0] is this vector:             in[2]  in[1]  in[0]
  // Bitwise-OR produces a 3 bit vector.         |      |      |
  // Assign this 3-bit result to out_any[3:1]:  o_a[3] o_a[2] o_a[1]

  // Thus, each output bit is the OR of the input bit and its neighbour to the right:
  // e.g., out_any[1] = in[1] | in[0];
  // Notice how this works even for long vectors.
  assign out_any = in[3:1] | in[2:0];

  assign out_both = in[2:0] & in[3:1];

  // XOR 'in' with a vector that is 'in' rotated to the right by 1 position: {in[0], in[3:1]}
  // The rotation is accomplished by using part selects[] and the concatenation operator{}.
  assign out_different = in ^ {in[0], in[3:1]};

endmodule
```

▼ Even longer vectors

```
module top_module(
    input [99:0] in,
    output [98:0] out_both,
    output [99:1] out_any,
    output [99:0] out_different );

    assign out_any = in[99:1] | in[98:0];
    assign out_both = in[98:0] & in[99:1];
    assign out_different = in ^ {in[0], in[99:1]};

endmodule
```

▼ Mutiplexers

▼ 2-to-1 multiplexer

```
//solution 1
module top_module (
  input a,
```

```
  input b,
  input sel,
  output out
);

  assign out = (sel & b) | (~sel & a);  // Mux expressed as AND and OR

  // Ternary operator is easier to read, especially if vectors are used:
  // assign out = sel ? b : a;

endmodule

//solution 2
module top_module(
    input a, b, sel,
    output out );

    assign out = sel? b:a;//ternary operator

endmodule
```

▼ 2-to-1 bus multiplexer

```
module top_module(
    input [99:0] a, b,
    input sel,
    output [99:0] out );

    assign out = sel? b:a;

endmodule
```

▼ 9-to-1 multiplexer

```
//solution 1
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'd0: out = a;
            4'd1: out = b;
            4'd2: out = c;
            4'd3: out = d;
            4'd4: out = e;
            4'd5: out = f;
            4'd6: out = g;
            4'd7: out = h;
            4'd8: out = i;
            default:out = 16'hffff;
        endcase
    end
endmodule


//solution 2
module top_module (
  input [15:0] a,, b, c, d, e, f, g, h, i,
  output logic [15:0] out
);

  // Case statements can only be used inside procedural blocks (always block)
  // This is a combinational circuit, so use a combinational always @(*) block.
  always @(*) begin
    out = '1;   // '1 is a special literal syntax for a number with all bits set to 1.
            // '0, 'x, and 'z are also valid.
            // I prefer to assign a default value to 'out' instead of using a
            // default case.
    case (sel)
      4'h0: out = a;
      4'h1: out = b;
      4'h2: out = c;
      4'h3: out = d;
```

```
      4'h4: out = e;
      4'h5: out = f;
      4'h6: out = g;
      4'h7: out = h;
      4'h8: out = i;
    endcase
  end

endmodule
```

▼ 256-to-1 multiplexer

```
module top_module (
  input [255:0] in,
  input [7:0] sel,
  output  out
);

  // Select one bit from vector in[]. The bit being selected can be variable.
  assign out = in[sel];

endmodule
```

▼ 256-to-1 4-bit multiplexer

```
//solution 1
module top_module (
  input [1023:0] in,
  input [7:0] sel,
  output [3:0] out
);

  assign out = {in[sel*4+3], in[sel*4+2], in[sel*4+1], in[sel*4+0]};//combine all bits

endmodule

//solution 2
module top_module(
    input [1023:0] in,
    input [7:0] sel,
    output [3:0] out );

    assign out = in[sel * 4 +: 4];// up to 4 bits
    //assign out = in[4*sel+3-:4]; //down to 4 bits
    //in[4*sel+3:4*sel] can't identify the width

endmodule
```

▼ Arithmetic Circuits

▼ Half adder

```
module top_module(
    input a, b,
    output cout, sum );

    assign {cout,sum} = a + b ;

endmodule
```

▼ Full adder

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

    assign {cout,sum} = a + b + cin;
```

```
endmodule
```

▼ 3-bit binary adder

```
module top_module(
    input [2:0] a, b,
    input cin,
    output [2:0] cout,
    output [2:0] sum );

    full_adder adder1(a[0],b[0],cin,cout[0],sum[0]);
    full_adder adder2(a[1],b[1],cout[0],cout[1],sum[1]);
    full_adder adder3(a[2],b[2],cout[1],cout[2],sum[2]);

endmodule

//submodule
module full_adder(
    input a, b, cin,
    output cout, sum );

    assign {cout,sum} = a + b + cin;

endmodule
```

▼ Adder

```
//solution 1
module top_module (
  input [3:0] x,
  input [3:0] y,
  output [4:0] sum
);

  assign sum = x+y; // Verilog addition automatically produces the carry-out bit.

endmodule

//solution 2
module top_module (
    input [3:0] x,
    input [3:0] y,
    output [4:0] sum);

    wire co1,co2,co3,co4;

    full_adder FA1(x[0],y[0],1'b0,co1,sum[0]);
    full_adder FA2(x[1],y[1],co1,co2,sum[1]);
    full_adder FA3(x[2],y[2],co2,co3,sum[2]);
    full_adder FA4(x[3],y[3],co3,co4,sum[3]);

    assign sum[4] = co4;

endmodule

//submodule
module full_adder(
    input a, b, cin,
    output cout, sum );

    assign {cout,sum} = a + b + cin;

endmodule
```

▼ Signed addition overflow

```
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]:1'b0 ;
    //Determine if the sign bits are the same
    //if they are, determine if the sign of the sum is the same as the sign of the addition

endmodule
```

▼ 100-bit binary adder

```
module top_module(
    input [99:0] a, b,
    input cin,
    output cout,
    output [99:0] sum );

    assign {cout,sum} = a + b + cin;

endmodule
```

▼ 4-digit BCD adder

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

    wire mcout[2:0];

    bcd_fadd BCDadd1(a[3:0],b[3:0],cin,mcout[0],sum[3:0]);
    bcd_fadd BCDadd2(a[7:4],b[7:4],mcout[0],mcout[1],sum[7:4]);
    bcd_fadd BCDadd3(a[11:8],b[11:8],mcout[1],mcout[2],sum[11:8]);
    bcd_fadd BCDadd4(a[15:12],b[15:12],mcout[2],cout,sum[15:12]);

endmodule
```

▼ Karnaugh Map to Circuit

▼ 3-variable

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

    assign out = a | b | c ;  // sum-of-products
    // assign out = ~(~a & ~b & ~c) ;  // product-of-sums

endmodule
```

▼ 4-variable

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

    // assign out = ~a&~d | ~b&~c | ~a&b&c | a&c&d ; // sum-of-products
    assign out = (~a|~b|c) & (~b|c|~d) & (~a|~c|d) & (a|b|~c|~d); // product-of-sums

endmodule
```

▼ 4-variable

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

    assign out = a | ~b&c ; // sum-of-products
    // assign out = (a|~b) & (a|c) ; //product-of-sums

endmodule
```

▼ 4-variable

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

    assign out = a^b^c^d ; // The number of 1's is odd

endmodule
```

▼ Minimum SOP and POS

```
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|d) & (~a|d) ;

endmodule
```

▼ Karnaugh map

```
module top_module (
    input [4:1] x,
    output f );

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

endmodule
```

▼ Karnaugh map

```
module top_module (
    input [4:1] x,
    output f
);

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

▼ K-map implemented with a multiplexer

```
module top_module (
    input c,
    input d,
    output [3:0] mux_in
);

    always @(*)begin
        mux_in = 4'b0000; // default
        case({c,d})
            2'b00: mux_in = 4'b0100;
            2'b01: mux_in = 4'b0001;
            2'b11: mux_in = 4'b1001;
            2'b10: mux_in = 4'b0101;
        endcase
    end

endmodule
```

▼ Sequential Logic

▼ Latches and Flip-Flops

▼ D flip-flop

```
module top_module (
    input clk,    // Clocks are used in sequential circuits
    input d,
    output reg q );//

    always@(posedge clk)
        q <= d ;

    // Use a clocked always block
    //   copy d to q at every positive edge of clk
    //   Clocked always blocks should use non-blocking assignments

endmodule
```

▼ D flip-flops

```
module top_module (
    input clk,
    input [7:0] d,
    output [7:0] q
);

    always@(posedge clk)
        q <= d;

endmodule
```

▼ DFF with reset

```
module top_module (
    input clk,
    input reset,            // Synchronous reset
    input [7:0] d,
    output [7:0] q
);

    always@(posedge clk)begin
        if (reset)
            q <= 8'd0;
        else
            q <= d;
    end

endmodule
```

▼ DFF with reset value

```
module top_module (
    input clk,
    input reset,
    input [7:0] d,
    output [7:0] q
);

    always@(negedge clk)begin
        if(reset)
            q = 8'h34;
      else
            q <= d;
    end

endmodule
```

▼ DFF with asynchronous reset

```
module top_module (
    input clk,
    input areset,   // active high asynchronous reset
    input [7:0] d,
    output [7:0] q
);
    // posedge areset：mixed single- and double-edge expressions are not supported
    // or ：can’t be changed to ||
    always@(posedge clk or posedge areset)begin
        if(areset)
            q <= 8'd0;
        else
            q <= d;
    end

endmodule
```

▼ DFF with byte enable

```
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)
            q <= 16'd0;
        else
            begin
            if(byteena[0])
                q[7:0] <= d[7:0];
          if(byteena[1])
              q[15:8] <= d[15:8];
            end
    end

endmodule
```

▼ D Latch

```
module top_module (
    input d,
    input ena,
    output q);

    always@(*)begin
        if(ena)
            q <= d;
```

```
    end
endmodule
```

▼ DFF

```
module top_module (
    input clk,
    input d,
    input ar,   // asynchronous reset
    output q);

    always@(posedge clk or posedge ar)begin
        if(ar)
            q <= 1'b0;
      else
            q <= d;
    end

endmodule
```

▼ DFF

```
module top_module (
    input clk,
    input d,
    input r,   // synchronous reset
    output q);

    always@(posedge clk)begin
        if(r)
            q <= 1'b0;
      else
            q <= d;
    end

endmodule
```

▼ DFF+gate

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

    always@(posedge clk)out <= out^in;

endmodule
```

▼ Mux and DFF

```
module top_module (
  input clk,
  input L,
  input r_in,
  input q_in,
  output reg Q);

    wire D;
    assign D = L? r_in:q_in;
    always@(posedge clk) Q <= D;

endmodule
```

▼ Mux and DFF

```
module top_module (
    input clk,
```

```
    input w, R, E, L,
    output Q
);

    wire O1,D;
    assign O1 = E? w:Q;
    assign D = L? R:O1;
    always@(posedge clk) Q <= D;

endmodule
```

▼ DFFs and gates

```
module top_module (
    input clk,
    input x,
    output z
);
  wire D1,D2,D3,Q1,Q2,Q3;

    assign D1 = x ^ Q1;
    assign D2 = x & ~Q2;
    assign D3 = x | ~Q3;
    assign z = ~(Q1|Q2|Q3);

    always@(posedge clk)begin
        Q1 <= D1;
        Q2 <= D2;
        Q3 <= D3;
    end

endmodule
```

▼ Create circuit from truth table

```
module top_module (
    input clk,
    input j,
    input k,
    output Q);

    wire D;

    assign D = ~k&Q | j&~Q;
    always@(posedge clk) Q <= D;

endmodule
```

▼ Detect an edge

```
module top_module (
    input clk,
    input [7:0] in,
    output [7:0] pedge
);
    reg [7:0] mid;

    always@(posedge clk) mid <= in;
    always@(posedge clk) pedge <= in&~mid;

endmodule
```

▼ Detect both edges

```
module top_module (
    input clk,
    input [7:0] in,
    output [7:0] anyedge
);
    reg [7:0] mid;
```

```
    always@(posedge clk) mid <= in;
    always@(posedge clk) anyedge <= in^mid;

endmodule
```

▼ Edge capture register

```
module top_module (
    input clk,
    input reset,
    input [31:0] in,
    output [31:0] out
);
    reg [31:0]  in_reg;

    always@(posedge clk)begin
        in_reg <= in;
    end

    always@(posedge clk)begin
        if(reset)
            out <= 32'd0;
        else
            out <= ~in & in_reg | out;//if ~in & in_reg == 32'd0, out keeps old value. If not, identifies negedge
    end

endmodule
```

▼ Dual-edge triggered flip-flop

```
// solution 1
module top_module (
    input clk,
    input d,
    output q
);

    wire q1,q2;

    always@(posedge clk)
        q1 <= d;
    always@(negedge clk)
        q2 <= d;

    assign q = clk? q1:q2;

endmodule

// solution 2
module top_module(
  input clk,
  input d,
  output q);

  reg p, n;

  // A positive-edge triggered flip-flop
    always @(posedge clk)
        p <= d ^ n;

    // A negative-edge triggered flip-flop
    always @(negedge clk)
        n <= d ^ p;

    // Why does this work?
    // After posedge clk, p changes to d^n. Thus q = (p^n) = (d^n^n) = d.
    // After negedge clk, n changes to p^n. Thus q = (p^n) = (p^d^p) = d.
    // At each (positive or negative) clock edge, p and n FFs alternately
    // load a value that will cancel out the other and cause the new value of d to remain.
    assign q = p ^ n;


  // Can't synthesize this.
  /*always @(posedge clk, negedge clk) begin
```

```
    q <= d;
  end*/


endmodule
```

▼ Counters

▼ Four-bit binary counter

```
module top_module (
    input clk,
    input reset,      // Synchronous active-high reset
    output [3:0] q);

    always@(posedge clk)begin
        if(reset)
            q <= 4'd0;
      else
            q <= q + 1'b1;
    end
endmodule
```

▼ Decade counter

```
module top_module (
    input clk,
    input reset,        // Synchronous active-high reset
    output [3:0] q);

    always@(posedge clk)begin
        if(reset || q == 9)//can't be changed for "or"
            q <= 4'd0;
        else
            q <= q + 1'b1;
    end

endmodule
```

▼ Decade counter again

```
module top_module (
    input clk,
    input reset,
    output [3:0] q);

    always @(posedge clk)begin
        if(reset || q == 10)
            q <= 1'b1;
        else
            q <= q + 1'b1;
    end
endmodule
```

▼ Slow decade counter

```
module top_module (
    input clk,
    input slowena,
    input reset,
    output [3:0] q);

    always@(posedge clk)begin
        if(reset)
            q <= 0;
        else if(slowena)begin
            if (q == 9)//slowena is high
                q <= 0;
            else
```

```
                q <= q + 1;
        end
        else
            q <= q;
    end
endmodule
```

▼ Counter 1-12

```
module top_module (
    input clk,
    input reset,
    input enable,
    output [3:0] Q,
    output c_enable,
    output c_load,
    output [3:0] c_d
); //

    assign c_enable = enable;
    always@(posedge clk)begin
        if(reset || Q == 12)begin
            c_load <= 0;
            c_d <= 1;
        end
        else
            c_load <= 1;

    end

    count4 the_counter (clk, c_enable, c_load, c_d , Q );



endmodule
```

▼ Counter 1000

```
module top_module (
    input clk,
    input reset,
    output OneHertz,
    output [2:0] c_enable
); //

    reg [3:0] Q0,Q1,Q2;

    always@(posedge clk)begin
        if(reset)
            c_enable[0] = 1;
    end

    assign c_enable[1] = (Q0 == 4'd9);
    assign c_enable[2] = ({Q1,Q0} == 8'h99);
    assign OneHertz = ({Q2,Q1,Q0} == 12'h999)? 1'b1:1'b0;

    bcdcount counter0 (clk, reset, c_enable[0], Q0);
    bcdcount counter1 (clk, reset, c_enable[1], Q1);
    bcdcount counter2 (clk, reset, c_enable[2], Q2);

endmodule
```

▼ 4-digit decimal counter

```
// solution 1
module top_module (
    input clk,
    input reset,   // Synchronous active-high reset
    output [3:1] ena,
    output [15:0] q);

  wire ena0;
```

```
    always@(posedge clk)begin
        if(reset)
            ena0 = 1;
    end

    assign ena[1] = (q[3:0] == 4'h9);
    assign ena[2] = (q[7:0] == 8'h99);
    assign ena[3] = (q[11:0] == 12'h999);

    modulo_10 counter0 (clk, reset, ena0, q[3:0]);
    modulo_10 counter1 (clk, reset, ena[1], q[7:4]);
    modulo_10 counter2 (clk, reset, ena[2], q[11:8]);
    modulo_10 counter3 (clk, reset, ena[3], q[15:12]);

endmodule

module modulo_10 (
    input clk,
    input reset,
    input slowena,
    output [3:0] q);

    always@(posedge clk)begin
        if(reset)
            q <= 0;
        else if(slowena)begin
            if (q == 9)//slowena is high
                q <= 0;
            else
                q <= q + 1;
        end
        else
            q <= q;
    end
endmodule

// solution 2
module top_module (
    input clk,
    input reset,   // Synchronous active-high reset
    output [3:1] ena,
    output [15:0] q);

    reg [3:0] ones;
    reg [3:0] tens;
    reg [3:0] hundreds;
    reg [3:0] thousands;

    always@(posedge clk)begin
        if(reset)begin
            ones <= 4'd0;
        end
        else if(ones == 4'd9)begin
            ones <= 4'd0;
        end
        else begin
            ones <= ones + 1'b1;
        end
    end

    always@(posedge clk)begin
        if(reset)begin
            tens <= 4'd0;
        end
        else if(tens == 4'd9 && ones == 4'd9)begin
            tens <= 4'd0;
        end
        else if(ones == 4'd9) begin
            tens <= tens + 1'b1;
        end
    end

    always@(posedge clk)begin
        if(reset)begin
            hundreds <= 4'd0;
        end
        else if(hundreds == 4'd9 && tens == 4'd9 && ones == 4'd9)begin
            hundreds <= 4'd0;
        end
        else if(tens == 4'd9 && ones == 4'd9) begin
            hundreds <= hundreds + 1'b1;
```

```
        end
    end

    always@(posedge clk)begin
        if(reset)begin
            thousands <= 4'd0;
        end
        else if(thousands == 4'd9 && hundreds == 4'd9 && tens == 4'd9 && ones == 4'd9)begin
            thousands <= 4'd0;
        end
        else if(hundreds == 4'd9 && tens == 4'd9 && ones == 4'd9) begin
            thousands <= thousands + 1'b1;
        end
    end

    assign q = {thousands, hundreds, tens, ones};
    assign ena[1] = (ones == 4'd9) ? 1'b1 : 1'b0;
    assign ena[2] = (tens == 4'd9 && ones == 4'd9) ? 1'b1 : 1'b0;
    assign ena[3] = (hundreds == 4'd9 && tens == 4'd9 && ones == 4'd9) ? 1'b1 : 1'b0;

endmodule
_________________
```


▼ 12-hour clock

```
module top_module(
    input clk,
    input reset,
    input ena,
    output pm,
    output [7:0] hh,
    output [7:0] mm,
    output [7:0] ss);

    reg     pm_temp;
    reg [3:0] ss_ones;
    reg [3:0] ss_tens;
    reg [3:0] mm_ones;
    reg [3:0] mm_tens;
    reg [3:0] hh_ones;
    reg [3:0] hh_tens;
    wire    add_ss_ones;
    wire    end_ss_ones;
    wire    add_ss_tens;
    wire    end_ss_tens;
    wire    add_mm_ones;
    wire    end_mm_ones;
    wire    add_mm_tens;
    wire    end_mm_tens;
    wire    add_hh_ones;
    wire    end_hh_ones_0;
    wire    end_hh_ones_1;
    wire    add_hh_tens;
    wire    end_hh_tens_0;
    wire    end_hh_tens_1;
    wire    pm_ding;

    always@(posedge clk)begin
        if(reset)begin
            ss_ones <= 4'd0;
        end
        else if(add_ss_ones)begin
            if(end_ss_ones)begin
                ss_ones <= 4'd0;
            end
            else begin
                ss_ones <= ss_ones + 1'b1;
            end
        end
    end

    assign add_ss_ones = ena;
    assign end_ss_ones = add_ss_ones && ss_ones == 4'd9;

    always@(posedge clk)begin
        if(reset)begin
```

```
            ss_tens <= 4'd0;
        end
        else if(add_ss_tens)begin
            if(end_ss_tens)begin
                ss_tens <= 4'd0;
            end
            else begin
                ss_tens <= ss_tens + 1'b1;
            end
        end
    end

    assign add_ss_tens = end_ss_ones;
    assign end_ss_tens = add_ss_tens && ss_tens == 4'd5;

    always@(posedge clk)begin
        if(reset)begin
            mm_ones <= 4'd0;
        end
        else if(add_mm_ones)begin
            if(end_mm_ones)begin
                mm_ones <= 4'd0;
            end
            else begin
                mm_ones <= mm_ones + 1'b1;
            end
        end
    end

    assign add_mm_ones = end_ss_tens;
    assign end_mm_ones = add_mm_ones && mm_ones == 4'd9;

    always@(posedge clk)begin
        if(reset)begin
            mm_tens <= 4'd0;
        end
        else if(add_mm_tens)begin
            if(end_mm_tens)begin
                mm_tens <= 4'd0;
            end
            else begin
                mm_tens <= mm_tens + 1'b1;
            end
        end
    end

    assign add_mm_tens = end_mm_ones;
    assign end_mm_tens = add_mm_tens && mm_tens == 4'd5;

    always@(posedge clk)begin
        if(reset)begin
            hh_ones <= 4'd2;
        end
        else if(add_hh_ones)begin
            if(end_hh_ones_0)begin
                hh_ones <= 4'd0;
            end
            else if(end_hh_ones_1)begin
                hh_ones <= 4'd1;
            end
            else begin
                hh_ones <= hh_ones + 1'b1;
            end
        end
    end

    assign add_hh_ones = end_mm_tens;
    assign end_hh_ones_0 = add_hh_ones && hh_ones == 4'd9;
    assign end_hh_ones_1 = add_hh_ones && (hh_tens == 4'd1 && hh_ones == 4'd2);

    always@(posedge clk)begin
        if(reset)begin
            hh_tens <= 4'd1;
        end
        else if(add_hh_tens)begin
            if(end_hh_tens_0)begin
                hh_tens <= 4'd0;
            end
            else if(end_hh_tens_1)begin
                hh_tens <= hh_tens + 1'b1;
            end
```

```
        end
    end

    assign add_hh_tens = end_mm_tens;
    assign end_hh_tens_0 = add_hh_tens && end_hh_ones_1;
    assign end_hh_tens_1 = add_hh_tens && end_hh_ones_0;

    always@(posedge clk)begin
        if(reset)begin
            pm_temp <= 1'b0;
        end
        else if(pm_ding)begin
            pm_temp <= ~pm_temp;
        end
    end

    assign pm_ding = hh_tens == 4'd1 && hh_ones == 4'd1 && end_mm_tens;

    assign ss = {ss_tens, ss_ones};
    assign mm = {mm_tens, mm_ones};
    assign hh = {hh_tens, hh_ones};
    assign pm = pm_temp;

endmodule
```

▼ Shift Registers

▼ 4-bit shift register

```
module top_module(
    input clk,
    input areset,  // async active-high reset to zero
    input load,
    input ena,
    input [3:0] data,
    output reg [3:0] q);

    always@(posedge clk or posedge areset)begin
        if(areset)
            q <= 4'd0;
        else if(load)
            q <= data;
        else if(ena)
            q <= {1'b0,q[3:1]};//q <= q[3:1]; Use vector part select to express a shift.
        else
            q <= q;
    end

endmodule
```

▼ Left/right rotator

```
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
                case(ena)
                    2'b01: q <= {q[0],q[99:1]};
                    2'b10: q <= {q[98:0],q[99]};
          default: q <= q;
                endcase
            end
    end

endmodule
```

▼ Left/right arithmatic shift by 1 or 8

```
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 if(ena)
            case(amount)
                2'b00: q <= {q[62:0],1'd0};
                2'b01: q <= {q[55:0],8'd0};
                2'b10: q <= {q[63],q[63:1]};
                2'b11: q <= {{8{q[63]}},q[63:8]};
            endcase
        else
            q <= q;
    end

endmodule
```

▼ 5-bit LFSR

```
module top_module(
    input clk,
    input reset,    // Active-high synchronous reset to 5'h1
    output [4:0] q
);

    always@(posedge clk)begin
        if(reset)
            q <= 5'h1;
        else begin
            q[4] <= 1'b0 ^ q[0];
            {q[3],q[1],q[0]} <= {q[4],q[2],q[1]};
            q[2] <= q[3] ^ q[0];
        end
    end
endmodule
```

▼ 3-bit LFSR

```
module top_module (
  input [2:0] SW,      // R
  input [1:0] KEY,     // L and clk
  output [2:0] LEDR);  // Q

    wire b;
    assign b = LEDR[1] ^ LEDR[2];
    submodule module1(KEY[0], KEY[1], {SW[0], LEDR[2]}, LEDR[0]);
    submodule module2(KEY[0], KEY[1], {SW[1], LEDR[0]}, LEDR[1]);
    submodule module3(KEY[0], KEY[1], {SW[2], b}, LEDR[2]);

endmodule

module submodule(
    input clk,
    input L,
    input[1:0] in,
    output Q);

    always@(posedge clk)begin
        case(L)
            1'b0: Q <= in[0];
            1'b1: Q <= in[1];
        endcase
    end
endmodule
```

▼ 32-bit LFSR

```
module top_module(
    input clk,
    input reset,    // Active-high synchronous reset to 32'h1
    output [31:0] q
);

    always@(posedge clk)begin
        if(reset)
            q <= 32'h1;
        else begin
            {q[30:22],q[20:2]} <= {q[31:23],q[21:3]};
            q[31] <= q[0]^1'b0;
            q[21] <= q[0]^q[22];
            q[1] <= q[0]^q[2];
            q[0] <= q[0]^q[1];
        end
    end
endmodule
```

▼ Shift register

```
module top_module (
    input clk,
    input resetn,   // synchronous reset
    input in,
    output out);

    reg[2:0] q;
    always@(posedge clk)begin
        if(~resetn)
          {q[2:0],out} <= 4'd0;
        else
          {q[2:0],out} <= {in,q[2:0]};
    end
endmodule
```

▼ Shift register

```
module top_module (
    input [3:0] SW,
    input [3:0] KEY,
    output [3:0] LEDR
); //

    MUXDFF module3(KEY[0],KEY[1],KEY[2],SW[3],KEY[3],LEDR[3]);
    MUXDFF module2(KEY[0],KEY[1],KEY[2],SW[2],LEDR[3],LEDR[2]);
    MUXDFF module1(KEY[0],KEY[1],KEY[2],SW[1],LEDR[2],LEDR[1]);
    MUXDFF module0(KEY[0],KEY[1],KEY[2],SW[0],LEDR[1],LEDR[0]);

endmodule

module MUXDFF (
    input clk,
  input E,
  input L,
  input R,
  input W,
  output Q);

    wire b1,b2;
    assign b1 = E? W:Q;
    assign b2 = L? R:b1;
    always@(posedge clk) Q <= b2;
endmodule
```

▼ 3-input LUT

```
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)
            Q[7:0] <= {Q[6:0],S};
        else
            Q <= Q;
    end

    always@(A or B or C)begin
        case({A,B,C})
            3'd0: Z <= Q[0];
            3'd1: Z <= Q[1];
            3'd2: Z <= Q[2];
            3'd3: Z <= Q[3];
            3'd4: Z <= Q[4];
            3'd5: Z <= Q[5];
            3'd6: Z <= Q[6];
            3'd7: Z <= Q[7];
        endcase
    end

endmodule
```

▼ More Circuits

▼ Rule 90

```
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};//q <= q[511:1] ^ {q[510:0], 1'b0} ;
    end
endmodule
```

▼ Rule 110

```
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 <= q^{q[510:0],1'b0} | ~{1'b0,q[511:1]}&{q[510:0],1'b0};//Karnaugh map
            //left is high bit, right is low bit.
    end
endmodule
```

▼ Conway's Game of Life 16×16

```
module top_module(
    input clk,
    input load,
```

```
    input [255:0] data,
    output [255:0] q );

    reg [3:0] sum;
    integer i;

    always@(posedge clk)begin
        if(load)
            q <= data;
        else begin
            for(i=0;i<256;i=i+1)begin
                if(i==0)
                    sum = q[255]+q[240]+q[241]+q[15]+q[1]+q[31]+q[16]+q[17];
                else if(i==15)
                    sum = q[254]+q[255]+q[240]+q[14]+q[0]+q[30]+q[31]+q[16];
                else if(i==240)
                    sum = q[239]+q[224]+q[225]+q[255]+q[241]+q[15]+q[0]+q[1];
                else if(i==255)
                    sum = q[238]+q[239]+q[224]+q[254]+q[240]+q[14]+q[15]+q[0];
                else if(i>0 && i<15)
                    sum = q[i+239]+q[i+240]+q[i+241]+q[i-1]+q[i+1]+q[i+15]+q[i+16]+q[i+17];
                else if(i>240 && i<255)
                    sum = q[i-17]+q[i-16]+q[i-15]+q[i-1]+q[i+1]+q[i-241]+q[i-240]+q[i-239];
                else if((i+1)%16==0)
                    sum = q[i-17]+q[i-16]+q[i-31]+q[i-1]+q[i-15]+q[i+15]+q[i+16]+q[i+1];
                else if(i%16==0)
                    sum = q[i-1]+q[i-16]+q[i-15]+q[i+15]+q[i+1]+q[i+31]+q[i+16]+q[i+17];
                else
                    sum = q[i-17]+q[i-16]+q[i-15]+q[i-1]+q[i+1]+q[i+15]+q[i+16]+q[i+17];

                case(sum)
                    4'd2: q[i]<=q[i];
                    4'd3: q[i]<=1'b1;
                    default:q[i]<=1'b0;
                endcase
            end
        end
    end

endmodule
```

▼ Finite State Machines

▼ Simple FSM 1 (asynchronous reset)

```
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? A:B;
            B: next_state <= in? 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 == ...);
    assign out = (state==B);

endmodule
```

▼ Simple FSM 1 (synchronous reset)

```
// 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;//
    reg out;

    // Fill in state name declarations

    reg present_state, next_state;

    always @(posedge clk) begin
        if (reset) begin
            // Fill in reset logic
            present_state = 1'b1;
            out <= 1'b1;
        end
        else begin
            case (present_state)
                // Fill in state transition logic
                1'b0: next_state = in? 1'b0:1'b1;
              1'b1: next_state = in? 1'b1:1'b0;
            endcase

            // State flip-flops
            present_state = next_state;

            case (present_state)
                // Fill in output logic
                1'b0: out <= 1'b0;
                1'b1: out <= 1'b1;
            endcase
        end
    end

endmodule
```

▼ Simple FSM 2 (asynchronous reset)

```
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
        // State transition logic
        case(state)
            OFF: next_state <= j? ON:OFF;
            ON: next_state <= k? OFF:ON;
        endcase
    end

    always @(posedge clk, posedge areset) begin
        // State flip-flops with asynchronous reset
        if(areset)
            state <= OFF;
        else
            state <= next_state;
    end

    // Output logic
    // assign out = (state == ...);
    assign out = (state==ON);

endmodule
```

▼ Simple FSM 2 (synchronous reset)

```
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
        // State transition logic
        case(state)
            OFF: next_state <= j? ON:OFF;
            ON: next_state <= k? OFF:ON;
        endcase
    end

    always @(posedge clk) begin
        // State flip-flops with synchronous reset
        state <= reset? OFF:next_state;
    end

    // Output logic
    // assign out = (state == ...);
    assign out = (state == ON);

endmodule
```

▼ Simple state transitions 3

```
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;
    reg [1:0] present_state;
    // State transition logic: next_state = f(state, in)
    always@(*)begin
        case(state)
            A: next_state <= in? B:A;
            B: next_state <= in? B:C;
            C: next_state <= in? D:A;
            D: next_state <= in? B:C;
        endcase
    end
    // Output logic:  out = f(state) for a Moore state machine
    assign out = (state==D);
endmodule
```

▼ Simple one-hot state transition 3

```
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;

    // State transition logic: Derive an equation for each state flip-flop.
    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;

    // Output logic:
    assign out = (state[D] == 1);

endmodule
```

▼ Simple FSM 3 (asynchronous reset)

```
//solution 1
module top_module(
    input clk,
    input in,
    input areset,
    output out); //

    parameter A=0,B=1,C=2,D=3;
    reg[3:0] state,next;

    // State transition logic
  assign next[A] = state[A]&~in | state[C]&~in;
    assign next[B] = state[A]&in | state[B]&in | state[D]&in;
    assign next[C] = state[B]&~in | state[D]&~in;
    assign next[D] = state[C]&in;


    // State flip-flops with asynchronous reset
    always@(posedge clk,posedge areset)begin
        if(areset)
            state <= 4'b0001;
        else
            state <= next;
    end

    // Output logic
    assign out = (state[D] == 1);

endmodule

//solution 2
module top_module (
  input clk,
  input in,
  input areset,
  output out
);

  // Give state names and assignments. I'm lazy, so I like to use decimal numbers.
  // It doesn't really matter what assignment is used, as long as they're unique.
  parameter A=0, B=1, C=2, D=3;
  reg [1:0] state;    // Make sure state and next are big enough to hold the state encodings.
  reg [1:0] next;




    // Combinational always block for state transition logic. Given the current state and inputs,
    // what should be next state be?
    // Combinational always block: Use blocking assignments.
    always@(*) begin
    case (state)
      A: next = in ? B : A;
      B: next = in ? B : C;
      C: next = in ? D : A;
      D: next = in ? B : C;
    endcase
    end



    // Edge-triggered always block (DFFs) for state flip-flops. Asynchronous reset.
    always @(posedge clk, posedge areset) begin
    if (areset) state <= A;
        else state <= next;
  end



  // Combinational output logic. In this problem, an assign statement is the simplest.
  assign out = (state==D);

endmodule
```

▼ Simple FSM 3 (synchronous reset)

```
module top_module(
    input clk,
    input in,
    input reset,
    output out); //

    parameter A=0,B=1,C=2,D=3;
    reg[3:0] state,next;

    // State transition logic
  assign next[A] = state[A]&~in | state[C]&~in;
    assign next[B] = state[A]&in | state[B]&in | state[D]&in;
    assign next[C] = state[B]&~in | state[D]&~in;
    assign next[D] = state[C]&in;


    // State flip-flops with asynchronous reset
    always@(posedge clk)begin
        if(reset)
            state <= 4'b0001;
        else
            state <= next;
    end

    // Output logic
    assign out = (state[D] == 1);

endmodule
```

▼ Design a Moore FSM

```
module top_module (
    input clk,
    input reset,
    input [3:1] s,
    output fr3,
    output fr2,
    output fr1,
    output dfr
);
    // set parameter
    //A2:001->000,B1:000->001,B2:011->001,C1:001->011,C2:111->011,D1:011->111
    parameter A2=0, B1=1, B2=2, C1=3, C2=4, D1=5;
    wire[2:0] state, next;

    // state transition (combination logic)
    always@(*)begin
        case(state)
            A2: next = s[1]? B1:A2;
            B1: next = s[2]? C1:(s[1]? B1:A2);
            B2: next = s[2]? C1:(s[1]? B2:A2);
            C1: next = s[3]? D1:(s[2]? C1:B2);
            C2: next = s[3]? D1:(s[2]? C2:B2);
            D1: next = s[3]? D1:C2;
        endcase
    end

    always@(posedge clk)begin
        if(reset)
            state <= A2;
        else
            state <= next;
    end

    // output control()sequential logic
    always@(*)begin
        case(state)
            A2: {fr3,fr2,fr1,dfr} = 4'b1111;
            B1: {fr3,fr2,fr1,dfr} = 4'b0110;
            B2: {fr3,fr2,fr1,dfr} = 4'b0111;
            C1: {fr3,fr2,fr1,dfr} = 4'b0010;
            C2: {fr3,fr2,fr1,dfr} = 4'b0011;
            D1: {fr3,fr2,fr1,dfr} = 4'b0000;
        endcase
    end
endmodule
```

▼ Lemmings 1

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    output walk_left,
    output walk_right);

    parameter LEFT=0, RIGHT=1;
    reg state, next_state;

    always @(*) begin
        // State transition logic
        case(state)
            LEFT: next_state = bump_left? RIGHT:LEFT;
            RIGHT: next_state = bump_right? LEFT:RIGHT;
        endcase
    end

    always @(posedge clk, posedge areset) begin
        // State flip-flops with asynchronous reset
        if(areset)
            state <= LEFT;
        else
            state <= next_state;
    end

    // Output logic
    assign walk_left = (state == LEFT);
    assign walk_right = (state == RIGHT);

endmodule
```

▼ Lemmings 2

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    input ground,
    output walk_left,
    output walk_right,
    output aaah );

    parameter left=0, right=1, fall_l=2, fall_r=3;
    wire[1:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            left: next = ground? (bump_left? right:left):fall_l;
            right: next = ground? (bump_right? left:right):fall_r;
            fall_l: next = ground? left:fall_l;
            fall_r: next = ground? right:fall_r;
        endcase
    end

    // flip-flop and areset logic
    always@(posedge clk, posedge areset)begin
        if(areset)
            state <= left;
        else
            state <= next;
    end

    // output
    assign walk_left = (state == left);
    assign walk_right = (state == right);
    assign aaah = (state == fall_l || state == fall_r);

endmodule
```

▼ Lemmings 3

```
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 );

    parameter left=0,right=1,dig_l=2,dig_r=3,fall_l=4,fall_r=5;
    wire[2:0] state,next;

    // state transition logic
    always@(*)begin
        case(state)
            left: next = ground? (dig? dig_l:(bump_left? right:left)):fall_l;
            right: next = ground? (dig? dig_r:(bump_right? left:right)):fall_r;
            dig_l: next = ground? dig_l:fall_l;
            dig_r: next = ground? dig_r:fall_r;
            fall_l: next = ground? left:fall_l;
            fall_r: next = ground? right:fall_r;
        endcase
    end

    // flip-flop and areset
    always@(posedge clk, posedge areset)begin
        if(areset)
            state <= left;
        else
            state <= next;
    end

    // output
    assign walk_left = (state == left);
    assign walk_right = (state == right);
    assign digging = (state == dig_l || state == dig_r);
    assign aaah = (state == fall_l || state == fall_r);

endmodule
```

▼ Lemmings 4

```
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 );

    parameter left=0, right=1, dig_l=2, dig_r=3, fall_l=4, fall_r=5, splatter=6;
    wire[2:0] state, next;

    wire[4:0] timelen;// counter of time
    wire signal;// time of falling is over 20 cycles

    // state transition logic
    always@(*)begin
        case(state)
            left: next = ground? (dig? dig_l:(bump_left? right:left)):fall_l;
            right: next = ground? (dig? dig_r:(bump_right? left:right)):fall_r;
            dig_l: next = ground? dig_l:fall_l;
            dig_r: next = ground? dig_r:fall_r;
            fall_l: begin
                if(signal)
                    next = ground? splatter:fall_l;
                else
```

```
                next = ground? left:fall_l;
            end
            fall_r: begin
                if(signal)
                    next = ground? splatter:fall_r;
                else
                  next = ground? right:fall_r;
            end
            splatter: next = splatter;
        endcase
    end

    // timing
    always@(posedge clk, posedge areset)begin
        if(areset)
            timelen <= 0;
        else if(state==fall_l || state==fall_r)begin
            if(timelen==19)
              signal <= 1;
            else
                timelen <= timelen + 1;
        end
        else begin
            signal <= 0;
            timelen <=0;
        end
    end

    // flip-flop and areset
    always@(posedge clk, posedge areset)begin
        if(areset)
            state <= left;
        else
            state <= next;
    end

    // output
    assign walk_left = (state == left);
    assign walk_right = (state == right);
    assign digging = (state == dig_l || state == dig_r);
    assign aaah = (state == fall_l || state == fall_r);
endmodule
```

▼ One-hot FSM

```
module top_module(
    input in,
    input [9:0] state,
    output [9:0] next_state,
    output out1,
    output out2);

    // state transition logic
    assign next_state[0] = state[0]&~in | state[1]&~in | state[2]&~in | state[3]&~in | state[4]&~in | state[7]&~in | state
    assign next_state[1] = state[0]&in | state[8]&in | state[9]&in;
    assign next_state[6:2] = {state[5]&in, state[4]&in, state[3]&in, state[2]&in, state[1]&in};
    assign next_state[7] = state[6]&in | state[7]&in;
    assign next_state[9:8] = {state[6]&~in, state[5]&~in};

    // output
    assign out1 = state[8] | state[9];
    assign out2 = state[7] | state[9];

endmodule
```

▼ PS/2 packed parser

```
module top_module(
    input clk,
    input [7:0] in,
    input reset,    // Synchronous reset
    output done); //

    parameter s0=0, s1=1, s2=2, s3=3;
    wire[1:0] state, next;
```

```
    // State transition logic (combinational)
    always@(*)begin
        case(state)
            s0: next = in[3]? s1:s0;
            s1: next = s2;
            s2: next = s3;
            s3: next = in[3]? s1:s0;
        endcase
    end

    // State flip-flops (sequential)
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    // Output logic
    assign done = (state == s3);

endmodule
```

▼ PS/2 packed parser and datapath

```
module top_module(
    input clk,
    input [7:0] in,
    input reset,    // Synchronous reset
    output [23:0] out_bytes,
    output done); //

    // FSM from fsm_ps2
    parameter s0=0, s1=1, s2=2, s3=3;
    wire[1:0] state, next;

    // State transition logic (combinational)
    always@(*)begin
        case(state)
            s0: next = in[3]? s1:s0;
            s1: next = s2;
            s2: next = s3;
            s3: next = in[3]? s1:s0;
        endcase
    end

    // State flip-flops (sequential)
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    // Output logic
    assign done = (state == s3);

    // New: Datapath to store incoming bytes.
    always@(posedge clk)begin
        case(state)
            s0: out_bytes[23:16] <= in;
            s1: out_bytes[15:8] <= in;
            s2: out_bytes[7:0] <= in;
            s3: out_bytes[23:16] <= in;
        endcase
    end

endmodule
```

▼ Serial receiver

```
module top_module(
    input clk,
    input in,
```

```
    input reset,    // Synchronous reset
    output done
);
    parameter s0=0, s1=1,s2=2, s3=3, s4=4, s5=5, s6=6, s7=7, s8=8, s9=9, s10=10, s11=11;
    wire [3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = in? s0:s1;
            s1: next = s2;
            s2: next = s3;
            s3: next = s4;
            s4: next = s5;
            s5: next = s6;
            s6: next = s7;
            s7: next = s8;
            s8: next = s9;
            s9: next = in? s10:s11;
            s10: next = in? s0:s1;
            s11: next = in? s0:s11;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    //output
    assign done = (state == s10);

endmodule
```

▼ Serial receiver and datapath

```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
); //

    // Use FSM from Fsm_serial
    parameter s0=0, s1=1,s2=2, s3=3, s4=4, s5=5, s6=6, s7=7, s8=8, s9=9, s10=10, s11=11;
    wire [3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = in? s0:s1;
            s1: next = s2;
            s2: next = s3;
            s3: next = s4;
            s4: next = s5;
            s5: next = s6;
            s6: next = s7;
            s7: next = s8;
            s8: next = s9;
            s9: next = in? s10:s11;
            s10: next = in? s0:s1;
            s11: next = in? s0:s11;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    //output
```

```
    assign done = (state == s10);

    // New: Datapath to latch input bits.
    always@(posedge clk)begin
        case(state)
            s1: out_byte[0] <= in;
            s2: out_byte[1] <= in;
            s3: out_byte[2] <= in;
            s4: out_byte[3] <= in;
            s5: out_byte[4] <= in;
            s6: out_byte[5] <= in;
            s7: out_byte[6] <= in;
            s8: out_byte[7] <= in;
            default: out_byte <= out_byte;
        endcase
    end

endmodule
```

▼ Serial receiver with parity checking

```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
); //

    // Modify FSM and datapath from Fsm_serialdata
    // Use FSM from Fsm_serial
    parameter s0=0, s1=1, s2=2, s3=3, s4=4, s5=5, s6=6, s7=7, s8=8, s9=9, s10=10, s11=11, s12=12;
    wire [3:0] state, next;
    wire odd_bit; //odd parity bit
    reg odd;
    wire res; //reset of submodule

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = in? s0:s1;
            s1: next = s2;
            s2: next = s3;
            s3: next = s4;
            s4: next = s5;
            s5: next = s6;
            s6: next = s7;
            s7: next = s8;
            s8: next = s9;
            s9: next = s10;
            s10: next = in? s11:s12;
            s11: next = in? s0:s1;
            s12: next = in? s0:s12;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    //output
    assign done = (odd == 0 && state == s11);

    // New: Datapath to latch input bits.
    always@(posedge clk)begin
        case(state)
            s1: out_byte[0] <= in;
            s2: out_byte[1] <= in;
            s3: out_byte[2] <= in;
            s4: out_byte[3] <= in;
            s5: out_byte[4] <= in;
            s6: out_byte[5] <= in;
            s7: out_byte[6] <= in;
            s8: out_byte[7] <= in;
```

```
            s9: odd_bit <= in;
            default: out_byte <= out_byte;
        endcase
    end

    // New: Add parity checking.
    always@(*)begin  //combination logic
        if(state == s0 || state == s11)
            res <= 1;
        else
            res <= 0;
    end

    parity submodule1(clk, res, in, odd);

endmodule
```

▼ Sequence recognition

```
module top_module(
    input clk,
    input reset,    // Synchronous reset
    input in,
    output disc,
    output flag,
    output err);

    parameter s0=0, s1=1, s2=2, s3=3, s4=4, s5=5, s6=6, se=7, sd=8, sf=9;
    wire[3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = in? s1:s0;
            s1: next = in? s2:s0;
            s2: next = in? s3:s0;
            s3: next = in? s4:s0;
            s4: next = in? s5:s0;
            s5: next = in? s6:sd;
            s6: next = in? se:sf;
            se: next = in? se:s0;
            sd: next = in? s1:s0;
            sf: next = in? s1:s0;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    // output
    assign disc = (state == sd);
    assign flag = (state == sf);
    assign err = (state == se);

endmodule
```

▼ Q8: Design a Mealy FSM

```
// solution 1
module top_module (
    input clk,
    input aresetn,    // Asynchronous active-low reset
    input x,
    output z );

    parameter idle=0, s1=1, s2=2, s3=3;
    wire [1:0] state, next;

    // state transition logic
    always@(*)begin
```

```
        case(state)
            idle: next = x? s1:idle;
            s1: next = x? s1:s2;
            s2: next = x? s3:idle;
            s3: next = x? s1:s2;
        endcase
    end

    // flip-flop and aresetn
    always@(posedge clk, negedge aresetn)begin
        if(~aresetn)
            state <= idle;
        else
            state <= next;
    end

    // output
    assign z = (state==s2 && x == 1);

endmodule

// solution 2
module top_module (
  input clk,
  input aresetn,
  input x,
  output reg z
);

  // Give state names and assignments. I'm lazy, so I like to use decimal numbers.
  // It doesn't really matter what assignment is used, as long as they're unique.
  parameter S=0, S1=1, S10=2;
  reg[1:0] state, next;   // Make sure state and next are big enough to hold the state encodings.



  // Edge-triggered always block (DFFs) for state flip-flops. Asynchronous reset.
  always@(posedge clk, negedge aresetn)
    if (!aresetn)
      state <= S;
    else
      state <= next;



    // Combinational always block for state transition logic. Given the current state and inputs,
    // what should be next state be?
    // Combinational always block: Use blocking assignments.
  always@(*) begin
    case (state)
      S: next = x ? S1 : S;
      S1: next = x ? S1 : S10;
      S10: next = x ? S1 : S;
      default: next = 'x;
    endcase
  end



  // Combinational output logic. I used a combinational always block.
  // In a Mealy state machine, the output depends on the current state *and*
  // the inputs.
  always@(*) begin
    case (state)
      S: z = 0;
      S1: z = 0;
      S10: z = x;   // This is a Mealy state machine: The output can depend (combinational) on the input.
      default: z = 1'bx;
    endcase
  end

endmodule
```

▼ Q5a: Serial two’s complementer(Moore FSM)

```
module top_module (
    input clk,
    input areset,
```

```
    input x,
    output z
);

    parameter A=0, B=1, C=2;
    wire[1:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = x? B:A;
            B: next = x? C:B;
            C: next = x? C:B;
        endcase
    end

    // flip-flop and areset
    always@(posedge clk, posedge areset)begin
        if(areset)
            state <= A;
        else
            state <= next;
    end

    // output
    assign z = (state == B);

endmodule
```

▼ Q5b: Serial two's complementer(Mealy FSM)

```
module top_module (
    input clk,
    input areset,
    input x,
    output z
);
  parameter A=2'b01, B=2'b10;
    wire[1:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = x? B:A;
            B: next = B;
            default: next = 'x;
        endcase
    end

    // flip-flop and areset
    always@(posedge clk, posedge areset)begin
        if(areset)
            state <= A;
        else
            state <= next;
    end

    // output
    always@(*)begin
        case(state)
            A: z = x;
            B: z = ~x;
        endcase
    end

endmodule
```

▼ Q3a: FSM

```
module top_module (
    input clk,
    input reset,   // Synchronous reset
    input s,
    input w,
    output z
```

```
);

    parameter A=0, B=1, C1=2, D1=3, E2=4, F2=5, G2=6, H3=7, I3=8;
    wire[3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = s? B:A;
            B: next = w? C1:D1;
            C1: next = w? E2:F2;
            D1: next = w? F2:G2;
            E2: next = w? H3:I3;
            F2: next = w? I3:H3;
            G2: next = H3;
            H3: next = w? C1:D1;
            I3: next = w? C1:D1;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= A;
        else
            state <= next;
    end

    assign z = (state == I3);
endmodule
```

▼ Q3b: FSM

```
module top_module (
    input clk,
    input reset,   // Synchronous reset
    input x,
    output z
);

    parameter s0=3'b000, s1=3'b001, s2=3'b010, s3=3'b011, s4=3'b100;
    wire [2:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = x? s1:s0;
            s1: next = x? s4:s1;
            s2: next = x? s1:s2;
            s3: next = x? s2:s1;
            s4: next = x? s4:s3;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    // output
    always@(*)begin
        case(state)
            s0: z = 0;
            s1: z = 0;
            s2: z = 0;
            s3: z = 1;
            s4: z = 1;
        endcase
    end

endmodule
```

▼ Q3c: FSM logic

```
module top_module (
    input clk,
    input [2:0] y,
    input x,
    output Y0,
    output z
);

    parameter s0=3'b000, s1=3'b001, s2=3'b010, s3=3'b011, s4=3'b100;
    wire [2:0] state, Y;

    // state transition logic
    always@(*)begin
        case(y[2:0])
            s0: Y = x? s1:s0;
            s1: Y = x? s4:s1;
            s2: Y = x? s1:s2;
            s3: Y = x? s2:s1;
            s4: Y = x? s4:s3;
        endcase
    end

    // output
    always@(*)begin
        case(y[2:0])
            s0: z = 0;
            s1: z = 0;
            s2: z = 0;
            s3: z = 1;
            s4: z = 1;
        endcase
    end

    assign Y0 = Y[0];

endmodule
```

▼ Q6b: FSM next-state logic

```
module top_module (
    input [3:1] y,
    input w,
    output Y2);

    parameter A=3'b000, B=3'b001, C=3'b010, D=3'b011, E=3'b100, F=3'b101;
    wire[3:1] Y;

    always@(*)begin
        case(y[3:1])
            A: Y = w? A:B;
            B: Y = w? D:C;
            C: Y = w? D:E;
            D: Y = w? A:F;
            E: Y = w? D:E;
            F: Y = w? D:C;
        endcase
    end

    assign Y2 = Y[2];

endmodule
```

▼ Q6c: FSM one-hot next-state logic

```
module top_module (
    input [6:1] y,
    input w,
    output Y2,
    output Y4);

    assign Y2 = y[1]&~w;
    assign Y4 = y[2]&w | y[3]&w | y[5]&w | y[6]&w;

endmodule
```

▼ Q6: FSM

```
module top_module (
    input clk,
    input reset,     // synchronous reset
    input w,
    output z);

    parameter A=3'b000, B=3'b001, C=3'b010, D=3'b011, E=3'b100, F=3'b101;
    wire[2:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = w? A:B;
            B: next = w? D:C;
            C: next = w? D:E;
            D: next = w? A:F;
            E: next = w? D:E;
            F: next = w? D:C;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= A;
        else
            state <= next;
    end

    // output
    assign z = (state == E || state == F);

endmodule
```

▼ Q2a: FSM

```
module top_module (
    input clk,
    input reset,   // Synchronous active-high reset
    input w,
    output z
);
    parameter A=3'b000, B=3'b001, C=3'b010, D=3'b011, E=3'b100, F=3'b101;
    wire[2:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = w? B:A;
            B: next = w? C:D;
            C: next = w? E:D;
            D: next = w? F:A;
            E: next = w? E:D;
            F: next = w? C:D;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= A;
        else
            state <= next;
    end

    // output
    assign z = (state == E || state == F);

endmodule
```

▼ Q2b: One-hot FSM equations

```
module top_module (
    input [5:0] y,
    input w,
    output Y1,
    output Y3
);

    assign Y1 = y[0]&w;
    assign Y3 = y[1]&~w | y[2]&~w | y[4]&~w | y[5]&~w;

endmodule
```

▼ Q2a: FSM

```
module top_module (
    input clk,
    input resetn,    // active-low synchronous reset
    input [3:1] r,   // request
    output [3:1] g   // grant
);

    parameter A=0, B=1, C=2, D=3;
    wire[2:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = r[1]? B:(r[2]? C:(r[3]? D:A) );
            B: next = r[1]? B:A;
            C: next = r[2]? C:A;
            D: next = r[3]? D:A;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(~resetn)
            state <= A;
        else
            state <= next;
    end

    // output
    assign g = {state==D, state==C, state==B};
endmodule
```

▼ Q2b: Another FSM

```
module top_module (
    input clk,
    input resetn,    // active-low synchronous reset
    input x,
    input y,
    output f,
    output g
);
    parameter A=0, B=1, S0=3, S1=4, S2=5, E1=8, E21=9, E22=10, E31=11, E32=12, E33=13;
    wire [3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            A: next = resetn? B:A;
            B: next = S0;
            S0: next = x? S1:S0;
            S1: next = x? S1:S2;
            S2: next = x? E1:S0;
            E1: next = y? E22:E21;
            E21: next = y? E32:E31;
            E22: next = E32;
            E31: next = E31;
            E32: next = E32;
        endcase
    end
```

```
    // flip-flop and reset
    always@(posedge clk)begin
        if(~resetn)
            state <= A;
        else
            state <= next;
    end

    // output
    assign f = (state == B);
    assign g = (state == E1 || state == E21 || state == E22 || state == E32);

endmodule
```

▼ Building Larger Circuits

▼ Counter with period 1000

```
module top_module (
    input clk,
    input reset,
    output [9:0] q);

    always@(posedge clk)begin
        if(reset)
            q <= 0;
        else if(q == 999)
            q <= 0;
        else
            q <= q + 1;
    end

endmodule
```

▼ 4-bit shift register and down counter

```
module top_module (
    input clk,
    input shift_ena,
    input count_ena,
    input data,
    output [3:0] q);

    always@(posedge clk)begin
        if(shift_ena) // shift in
            q <= {q[2:0],data};
        else if(count_ena)
            q <= q - 1; // count
        else
            q <= q;
    end

endmodule
```

▼ FSM: Sequence 1101 recognizer

```
module top_module (
    input clk,
    input reset,      // Synchronous reset
    input data,
    output start_shifting);

    parameter s0=0, s1=1, s2=2, s3=3, s4=4;
    wire[2:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            s0: next = data? s1:s0;
            s1: next = data? s2:s0;
            s2: next = data? s2:s3;
```

```
            s3: next = data? s4:s0;
            s4: next = s4;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= s0;
        else
            state <= next;
    end

    // output
    assign start_shifting = (state == s4);

endmodule
```

▼ FSM: Enable shift register

```
module top_module (
    input clk,
    input reset,      // Synchronous reset
    output shift_ena);

    wire[1:0] count;

    always@(posedge clk)begin
        if(reset)begin
            count <= 0;
            shift_ena <= 1;
        end
        else if(count == 3)
            shift_ena <= 0;
        else
            count <= count + 1;
    end

endmodule
```

▼ FSM: The complete FSM

```
module top_module (
    input clk,
    input reset,      // Synchronous reset
    input data,
    output shift_ena,
    output counting,
    input done_counting,
    output done,
    input ack );

    parameter S=0, S1=1, S11=2, S110=3, B0=4, B1=5, B2=6, B3=7, Count=8, Wait=9;
    wire[3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            S: next = data? S1:S;
            S1: next = data? S11:S;
            S11: next = data? S11:S110;
            S110: next = data? B0:S;
            B0: next = B1;
            B1: next = B2;
            B2: next = B3;
            B3: next = Count;
            Count: next = done_counting? Wait:Count;
            Wait: next = ack? S:Wait;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= S;
```

```
        else
            state <= next;
    end

    // output
    assign shift_ena = (state==B0 || state==B1 || state==B2 || state==B3);
    assign counting = (state == Count);
    assign done = (state == Wait);

endmodule
```

▼ The complete timer

```
module top_module(
    input clk,
    input reset,      // Synchronous reset
    input data,
    output [3:0] count,
    output counting,
    output done,
    input ack);

    wire shift_ena, done_counting, count_ena;

    ComFSM instance1(clk, reset, data, shift_ena, counting, done_counting, done, ack);
    Shift_Reg instance2(clk, shift_ena, count_ena, data, count);
    Counter instance3(clk, reset, count, counting, done_counting, count_ena);

endmodule

module ComFSM (
    input clk,
    input reset,      // Synchronous reset
    input data,
    output shift_ena,
    output counting,
    input done_counting,
    output done,
    input ack);

    parameter S=0, S1=1, S11=2, S110=3, B0=4, B1=5, B2=6, B3=7, Count=8, Wait=9;
    reg [3:0] state, next;

    // state transition logic
    always@(*)begin
        case(state)
            S: next = data? S1:S;
            S1: next = data? S11:S;
            S11: next = data? S11:S110;
            S110: next = data? B0:S;
            B0: next = B1;
            B1: next = B2;
            B2: next = B3;
            B3: next = Count;
            Count: next = done_counting? (ack? S:Wait):Count;
            Wait: next = ack? S:Wait;
        endcase
    end

    // flip-flop and reset
    always@(posedge clk)begin
        if(reset)
            state <= S;
        else
            state <= next;
    end

    // output
    assign shift_ena = (state==B0 || state==B1 || state==B2 || state==B3);
    assign counting = (state == Count) && (next != Wait) && (next != S);
    assign done = (state == Wait) || (state == Count)&(next == Wait) || (state == Count)&(next == S);

endmodule

module Shift_Reg (
    input clk,
    input shift_ena,
    input count_ena,
```

```
    input data,
    output [3:0] q);

    always@(posedge clk)begin
        if(shift_ena) // shift in
            q <= {q[2:0],data};
        else if(count_ena)
            q <= q - 1; // count
        else
            q <= q;
    end

endmodule

module Counter(
  input clk,
    input reset,
    input [3:0] count,
    input counting,
    output reg done_counting,
    output count_ena);

    wire [9:0] q;

    Counter1000 instance1(clk, reset, counting, q, count_ena);

    always@(posedge clk)begin
        if(count==0 && count_ena==1)
            done_counting <= 1;
        else
            done_counting <= 0;
    end

endmodule


module Counter1000 (
    input clk,
    input reset,
    input counting,
    output reg [9:0] q,
  output count);

    always@(posedge clk)begin
        if(reset | ~counting)
            q <= 0;
        else if(q == 999)
            q <= 0;
        else
            q <= q + 1;
    end

    assign count = (q == 999)? 1:0;

endmodule
```

▼ FSM: One-hot logic equations

```
module top_module(
    input d,
    input done_counting,
    input ack,
    input [9:0] state,    // 10-bit one-hot current state
    output B3_next,
    output S_next,
    output S1_next,
    output Count_next,
    output Wait_next,
    output done,
    output counting,
    output shift_ena
); //

    // You may use these parameters to access state bits using e.g., state[B2] instead of state[6].
    parameter S=0, S1=1, S11=2, S110=3, B0=4, B1=5, B2=6, B3=7, Count=8, Wait=9;

    assign B3_next = state[B2];
    assign S_next = state[S]&~d | state[S1]&~d | state[S110]&~d | state[Wait]& ack;
```

```
    assign S1_next = state[S]&d;
    assign Count_next = state[B3] | state[Count]&~done_counting;
    assign Wait_next = state[Count]&done_counting | state[Wait]&~ack;
    assign done = state[Wait];
    assign counting = state[Count];
    assign shift_ena = state[B0] | state[B1] | state[B2] | state[B3];

endmodule
```

▼ Verification: Reading Simulations

▼ Finding bugs in code

▼ Mux

```
module top_module (
    input sel,
    input [7:0] a,
    input [7:0] b,
    output [7:0] out  );

    always@(*)begin
        case(sel)
            1: out = a;
            0: out = b;
        endcase
    end

endmodule
```

▼ NAND

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

    wire o1;
    andgate inst1 (o1, a, b, c, 1'b1, 1'b1 );
    assign out = ~o1;

endmodule
```

▼ Mux

```
module top_module (
    input [1:0] sel,
    input [7:0] a,
    input [7:0] b,
    input [7:0] c,
    input [7:0] d,
    output [7:0] out  ); //

    wire [7:0] mux0, mux1;
    mux2 muxx0 ( sel[0],    a,    b, mux0 );
    mux2 muxx1 ( sel[0],    c,    d, mux1 );
    mux2 muxx2 ( sel[1], mux0, mux1,  out );

endmodule
```

▼ Add/sub

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input do_sub,
    input [7:0] a,
    input [7:0] b,
    output reg [7:0] out,
    output reg result_is_zero
);//

    always @(*) begin
```

```
        case (do_sub)
          0: out = a+b;
          1: out = a-b;
        endcase

        if (out==0)
            result_is_zero = 1;
        else
            result_is_zero = 0;
    end

endmodule
```

▼ Case statement

```
module top_module (
    input [7:0] code,
    output reg [3:0] out,
    output reg valid);//

    always @(*)begin
         valid = 1;
        case (code)
            8'h45: out = 0;
            8'h16: out = 1;
            8'h1e: out = 2;
            8'h26: out = 3;
            8'h25: out = 4;
            8'h2e: out = 5;
            8'h36: out = 6;
            8'h3d: out = 7;
            8'h3e: out = 8;
            8'h46: out = 9;
            default: begin
                out = 0;
                valid = 0;
            end
        endcase
    end

endmodule
```

▼ Build a circuit from a simulation waveform

▼ Combinational circuit 1

```
module top_module (
    input a,
    input b,
    output q );//

    assign q = a & b; // Fix me

endmodule
```

▼ Combinational circuit 2

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output q );//

    assign q = ~(a^b)^(c^d); // Fix me

endmodule
```

▼ Combinational circuit 3

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output q );//

    assign q = b&d | a&d | b&c | a&c ; // Fix me

endmodule
```

▼ Combinational circuit 4

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output q );//

    assign q = b | c; // Fix me

endmodule
```

▼ Combinational circuit 5

```
module top_module (
    input [3:0] a,
    input [3:0] b,
    input [3:0] c,
    input [3:0] d,
    input [3:0] e,
    output [3:0] q );

    always@(*)begin
        case(c)
            4'd0: q = b;
            4'd1: q = e;
            4'd2: q = a;
            4'd3: q = d;
            default: q = 4'hf;
        endcase
    end

endmodule
```

▼ Combinational circuit 6

```
module top_module (
    input [2:0] a,
    output [15:0] q );

    always@(*)begin
        case(a)
            3'd0: q = 16'h1232;
            3'd1: q = 16'haee0;
            3'd2: q = 16'h27d4;
            3'd3: q = 16'h5a0e;
            3'd4: q = 16'h2066;
            3'd5: q = 16'h64ce;
            3'd6: q = 16'hc526;
            3'd7: q = 16'h2f19;
        endcase
    end
endmodule
```

▼ Sequential circuit 7

```
module top_module (
    input clk,
    input a,
    output q );

    always@(posedge clk)begin
        if(a)
            q <= 0;
        else
            q <= 1;
    end

endmodule
```

▼ Sequential circuit 8

```
module top_module (
    input clock,
    input a,
    output p,
    output q );

    always@(*)begin
        if(clock)
            p = a;
        else
            p = p;
    end

    always@(negedge clock)begin
        q <= a;
    end
endmodule
```

▼ Sequential circuit 9

```
module top_module (
    input clk,
    input a,
    output [3:0] q );

    always@(posedge clk)begin
        if(a)
            q <= 4'd4;
        else if(q == 6)
            q <= 4'd0;
        else
            q <= q + 1;
    end

endmodule
```

▼ Sequential circuit 10

```
module top_module (
    input clk,
    input a,
    input b,
    output q,
    output state  );

    always@(posedge clk)begin
        if(a==b)
            state <= a;
        else
            state <= state;
    end

    assign q = state? ~a^b:a^b;

endmodule
```

▼ Verification: Writting Testbenches

▼ Clock

```
`timescale 1ps / 1ps
module top_module ( );

    parameter clk_period = 10;
    reg clk;

    initial begin
        clk = 1'b0;
        forever #(clk_period/2) clk = ~clk;
    end

    dut instance1(clk);

endmodule
```

▼ Testbench1

```
`timescale 1ps / 1ps
module top_module ( output reg A, output reg B );//

    // generate input patterns here
    initial begin
        A = 1'b0;
        B = 1'b0;

        #10 A =~ A;
        #5 B =~ B;
        #5 A =~ A;
        #20 B =~ B;
    end

endmodule
```

▼ AND gate

```
`timescale 1ps / 1ps
module top_module();

    reg [1:0] in;
    reg out;

    andgate andgate1(.in(in),.out(out));

    initial begin
        in = 2'b00;
        #10 in = 2'b01;
        #10 in = 2'b10;
        #10 in = 2'b11;
    end

endmodule
```

▼ Testbench2

```
module top_module();

    reg clk,in,out;
    reg [2:0] s;
    reg [2:0] num;

    initial begin
        clk = 1'b0;
        in = 1'b0;
        s = 3'd2;
        num = 3'd0;
```

```
        forever #5 clk =~ clk;
    end

    always@(posedge clk)begin
        num <= num + 1;
    end

    always@(negedge clk)begin
        case(num)
            3'd1: s <= 6;
            3'd2: begin
                s <= 2;
                in <= ~in;
            end
            3'd3: begin
                s <= 7;
                in <= ~in;
            end
            3'd4: begin
                s <= 0;
                in <= ~in;
            end
            3'd7: in <= ~in;
            default: begin
                s <= s;
                in <= in;
            end
        endcase
    end

    q7 instance1(clk, in, s, out);
endmodule
```

▼ T flip-flop

```
module top_module ();

    reg clk, reset, t, q;
    reg [2:0] num;
    initial begin
        clk = 1'b0;
        reset = 1'b1;
        t = 1'b1;
        num = 3'd0;
        forever #10 clk =~ clk;
    end

    always@(posedge clk)begin
        t <= ~t;
        num <= num + 1;
    end


    always@(negedge clk)begin
        if(num==2)
            reset <= 1'b1;
        else
            reset <= 1'b0;
    end

    tff instance1(clk, reset, t, q);

endmodule
```