

1. Making Latches

(15%)

Here is one solution:

```
module latchInferred (latchOut, set, clear, gate, in);
    parameter W=4;
    input gate, set, clear;
    input [W-1:0] in; // parameterize the bitwidths
    output [W-1:0] latchOut; // parameterize the bitwidths
    reg [W-1:0] latchOut;
    reg [W-1:0] in_reg;

    always @(set or clear or gate or in) begin
        if ( ~clear ) begin
            if ( ~set ) begin
                if( gate ) begin
                    latchOut = in;
                    in_reg = in;
                end
            else
                latchOut = in_reg;
            end
        else
            latchOut = 1;
        end
    else
        latchOut = 0;
    end
endmodule // latchInferred

module test;
    reg [3:0] in;
    reg gate, set, clear;
    wire [3:0] latchOut;

    latchInferred #10 myLatchInferred (latchOut, set, clear, gate, in);

    initial begin
        $monitor( $time,,"in=%d, gate=%d, set=%d, clear=%d,
latchOut=%d", in, gate, set, clear, latchOut);
        in=15; gate=0; set=0; clear=0;
        #10; in=15; gate=0; set=0; clear=1;
        #10; in=15; gate=0; set=1; clear=0;
        #10; in=15; gate=0; set=1; clear=1;
        #10; in=15; gate=1; set=0; clear=0;
        #10; in=15; gate=1; set=0; clear=1;
        #10; in=15; gate=1; set=1; clear=0;
        #10; in=15; gate=1; set=1; clear=1;
        #10; in=10; gate=1; set=0; clear=0;
        #10; in=10; gate=0; set=0; clear=0;
    end
endmodule
```

```

        #10 $finish(0);
    end
endmodule // test

```

The simulation result is what follows:

```

0  in=15, gate=0, set=0, clear=0, latchOut= x
10 in=15, gate=0, set=0, clear=1, latchOut= 0
20 in=15, gate=0, set=1, clear=0, latchOut= 1
30 in=15, gate=0, set=1, clear=1, latchOut= 0
40 in=15, gate=1, set=0, clear=0, latchOut=15
50 in=15, gate=1, set=0, clear=1, latchOut= 0
60 in=15, gate=1, set=1, clear=0, latchOut= 1
70 in=15, gate=1, set=1, clear=1, latchOut= 0
80 in=10, gate=1, set=0, clear=0, latchOut=10
90 in=10, gate=0, set=0, clear=0, latchOut=10

```

The circuit diagram is also attached.

2. A D-FF with force/release

(10%)

Here is the Verilog and the simulation:

```

module theyForcedMeToWriteThisUglyDff (q, d, ck, reset_L);
    input d, ck, reset_L;
    output q;
    reg q;

    always @(posedge ck)
        q <= d;

    always @(reset_L) begin
        if(reset_L) release q;
        else force q = 0;
    end
endmodule

module testUglyDff;
    wire q;
    reg ck, d, reset_L;

    theyForcedMeToWriteThisUglyDff Udff (q, d, ck, reset_L);

    always #50 ck = ~ck;

    initial begin
        d = 1;
        ck = 0;
        reset_L = 0;

        $monitor($time, "d = %b, reset_L = %b, q = %b", d, reset_L,
q);

```

```

        @(posedge ck);
        @(posedge ck); reset_L = 1;
        @(posedge ck); d = 0;
        @(posedge ck); reset_L = 0; d = 1;
        @(posedge ck); reset_L = 1; d = 0;
        @(posedge ck); d= 1;
        $finish;
    end
endmodule

```

Here is the simulation result:

```

0 d = 1, reset_L = 0, q = 0
150 d = 1, reset_L = 1, q = 1
250 d = 0, reset_L = 1, q = 0
350 d = 1, reset_L = 0, q = 0
450 d = 0, reset_L = 1, q = 0

```

3. Hierarchical Names

(10%)

Here is the complete code, with answers highlighted.

```

module a;
// Just for assigning function output
    reg temp;

    always
        begin: d
            begin: e
                //Call xyzzy from here, pass s (3 lines below) to it
                temp = xyzzy(f.s);
            end
            begin: f
                reg q, r, s;
            end
        end
endmodule

module b;
    reg q, u, v;

    // Show a $display statement here that prints all
    // the qs in module as instantiations
    initial
        begin
            $display("%b, %b, %b",a0.d.f.q,a1.d.f.q,a2.d.f.q);
        end

    a        a0 ();
    a        a1 ();
    a        a2 ();

    function xyzzy;

```

```

        input s;

        begin
            xyzzy = s;
        end
    endfunction
endmodule

module tb;
    reg temp;

    initial
        begin
            $display("%b",top.design.a1.d.f.q);
            temp = top.design.xyzzy(top.design.a0.d.f.s);
        end
endmodule

module top();
    b    design();
    tb    t(); //From inside tb, $display register q in a1
           //From inside tb, call xyzzy, pass s in a0 to it
endmodule

```

4. Wire You Asking This

(15%)

Here are the Verilog descriptions for (a) and (b):

```

module wireAdd_good (sum, A, B, carryout);
    output sum, carryout;
    input A,B;

    wand c,d, carryout;
    wor sum;

    // carryout = AB
    buf(carryout, A);
    buf(carryout, B);

    // sum = A ^ B
    buf(c, A);
    not(c, B);
    buf(d, B);
    not(d, A);
    buf(sum, c);
    buf(sum, d);
endmodule

module wireAdd_bad (sum, A, B, carryout);
    output sum, carryout;
    input A,B;

    wire c,d, carryout;

```

```

    wire sum;

    // carryout = AB
    buf(carryout, A);
    buf(carryout, B);

    // sum = A ^ B
    buf(c, A);
    not(c, B);
    buf(d, B);
    not(d, A);
    buf(sum, c);
    buf(sum, d);
endmodule

// Testbench
module testbench;
    reg A, B;
    wire sum1, carryout1;
    wire sum2, carryout2;

    wireAdd_good w1(sum1, A, B, carryout1);
    wireAdd_bad w2(sum2, A, B, carryout2);

    initial
        begin
            $monitor($time,, " A = %d, B = %d, sum_good = %d,
carry_good = %d, sum_bad = %d, carry_bad = %d", A, B, sum1, carryout1, sum2,
carryout2);

            #10 A = 0; B = 0;
            #10 A = 0; B = 1;
            #10 A = 1; B = 0;
            #10 A = 1; B = 1;
        end
endmodule

```

Here is the simulation result

```

0 A = x, B = x, sum_good = x, carry_good = x, sum_bad = x, carry_bad = x
10 A = 0, B = 0, sum_good = 0, carry_good = 0, sum_bad = x, carry_bad = 0
20 A = 0, B = 1, sum_good = 1, carry_good = 0, sum_bad = x, carry_bad = x
30 A = 1, B = 0, sum_good = 1, carry_good = 0, sum_bad = x, carry_bad = x
40 A = 1, B = 1, sum_good = 0, carry_good = 1, sum_bad = x, carry_bad = 1

```

(c). The Synthesis tool correctly recognizes the wand/wor example and produces the correct Verilog description. In the wire example it gives an incorrect Verilog description. See below for the correct and incorrect synthesized Verilog descriptions:

```

// Correct Verilog description for wand/wor design
module wireAdd_good ( sum, A, B, carryout );
    input A, B;
    output sum, carryout;

```

```

    and2_1x U1 ( .B(B), .A(A), .Y(carryout) );
    xor2_1x U2 ( .B(B), .A(A), .Y(sum) );
endmodule

```

// Incorrect Verilog description using 'wire'

```

module wireAdd_bad ( sum, A, B, carryout );
    input A, B;
    output sum, carryout;
    wire c, d;

    GTECH_BUF B_0 ( .A(A), .Z(carryout) );
    GTECH_BUF B_1 ( .A(B), .Z(carryout) );
    GTECH_BUF B_2 ( .A(A), .Z(c) );
    GTECH_NOT I_0 ( .A(B), .Z(c) );
    GTECH_BUF B_3 ( .A(B), .Z(d) );
    GTECH_NOT I_1 ( .A(A), .Z(d) );
    GTECH_BUF B_4 ( .A(c), .Z(sum) );
    GTECH_BUF B_5 ( .A(d), .Z(sum) );
endmodule

```

5. Loopy Synthesis

(20%)

Here is the Verilog description for (a) and (b):

```

// using a construct
module lfsr_loop(q, d);
    parameter n = 6;
    parameter [1:n] c = 6'h16;
    input [1:n] q;
    output d;
    reg d;
    integer i;

    always @(q)
        begin
            d = 0;
            for (i = 1; i <= n; i = i+1) begin
                if (c[i] == 1'b1)
                    d = d + q[i];
            end
        end
endmodule // lfsr_loop

// using a non-looping construct
module lfsr_nonloop(q,d);
    input [1:6] q;
    output d;
    reg d;

    always @(q)
        begin
            d = 0;

```

```

        d = d + q[2] + q[4] + q[5];
    end
endmodule // lfsr_nonloop

// A simple register with a parameterizable reset value.
module my_register(clk, reset, d, q);
    parameter reset_value = 0;
    input clk, reset, d;
    output q;
    reg q;

    always @(posedge clk or negedge reset)
    begin
        if (~reset)
            q <= reset_value;
        else
            q <= d;
        end
    endmodule // my_register

module testbench();
    parameter n = 6;
    wire [1:n] q0, q1;
    wire out0, out1;
    reg clk, reset;

    lfsr_loop    lfsr0(q0, out0);
    my_register #(1) testreg0(clk, reset, out0, q0[1]);
    my_register #(0) testreg1(clk, reset, q0[1], q0[2]);
    my_register #(0) testreg2(clk, reset, q0[2], q0[3]);
    my_register #(1) testreg3(clk, reset, q0[3], q0[4]);
    my_register #(0) testreg4(clk, reset, q0[4], q0[5]);

    lfsr_nonloop lfsr1(q1, out1);
    my_register #(1) testreg6(clk, reset, out1, q1[1]);
    my_register #(0) testreg7(clk, reset, q1[1], q1[2]);
    my_register #(0) testreg8(clk, reset, q1[2], q1[3]);
    my_register #(1) testreg9(clk, reset, q1[3], q1[4]);
    my_register #(0) testreg10(clk, reset, q1[4], q1[5]);
    my_register #(1) testreg11(clk, reset, q1[5], q1[6]);

    always
        #5 clk = ~clk;

    initial
        begin
            $monitor ($time,, "\nq0[1:6] = %x, %x, %x, %x, %x,
%x\t out0 = %x \nq1[1:6] = %x, %x, %x, %x, %x, %x\t out1 = %x \n", q0[1],
q0[2], q0[3], q0[4], q0[5], q0[6], out0, q1[1], q1[2], q1[3], q1[4], q1[5],
q1[6], out1);

            clk = 0;
            reset = 0;
            #15 reset = 1;
            #200 $finish;
        end
endmodule

```

```

        end // initial begin
endmodule // testbench

```

The following is the simulation result for (a) and (b).

```

          0
q0[1:6] = 1, 0, 0, 1, 0, 1      out0 = 1
q1[1:6] = 1, 0, 0, 1, 0, 1      out1 = 1

          15
q0[1:6] = 1, 1, 0, 0, 1, 0      out0 = 0
q1[1:6] = 1, 1, 0, 0, 1, 0      out1 = 0

          25
q0[1:6] = 0, 1, 1, 0, 0, 1      out0 = 1
q1[1:6] = 0, 1, 1, 0, 0, 1      out1 = 1

          35
q0[1:6] = 1, 0, 1, 1, 0, 0      out0 = 1
q1[1:6] = 1, 0, 1, 1, 0, 0      out1 = 1

          45
q0[1:6] = 1, 1, 0, 1, 1, 0      out0 = 1
q1[1:6] = 1, 1, 0, 1, 1, 0      out1 = 1

          55
q0[1:6] = 1, 1, 1, 0, 1, 1      out0 = 0
q1[1:6] = 1, 1, 1, 0, 1, 1      out1 = 0

          65
q0[1:6] = 0, 1, 1, 1, 0, 1      out0 = 0
q1[1:6] = 0, 1, 1, 1, 0, 1      out1 = 0

          75
q0[1:6] = 0, 0, 1, 1, 1, 0      out0 = 0
q1[1:6] = 0, 0, 1, 1, 1, 0      out1 = 0

          85
q0[1:6] = 0, 0, 0, 1, 1, 1      out0 = 0
q1[1:6] = 0, 0, 0, 1, 1, 1      out1 = 0

          95
q0[1:6] = 0, 0, 0, 0, 1, 1      out0 = 1
q1[1:6] = 0, 0, 0, 0, 1, 1      out1 = 1

          105
q0[1:6] = 1, 0, 0, 0, 0, 1      out0 = 0
q1[1:6] = 1, 0, 0, 0, 0, 1      out1 = 0

          115
q0[1:6] = 0, 1, 0, 0, 0, 0      out0 = 1
q1[1:6] = 0, 1, 0, 0, 0, 0      out1 = 1

          125

```

```

q0[1:6] = 1, 0, 1, 0, 0, 0      out0 = 0
q1[1:6] = 1, 0, 1, 0, 0, 0      out1 = 0

```

```

      135
q0[1:6] = 0, 1, 0, 1, 0, 0      out0 = 0
q1[1:6] = 0, 1, 0, 1, 0, 0      out1 = 0

```

```

      145
q0[1:6] = 0, 0, 1, 0, 1, 0      out0 = 1
q1[1:6] = 0, 0, 1, 0, 1, 0      out1 = 1

```

```

      155
q0[1:6] = 1, 0, 0, 1, 0, 1      out0 = 1
q1[1:6] = 1, 0, 0, 1, 0, 1      out1 = 1

```

```

      165
q0[1:6] = 1, 1, 0, 0, 1, 0      out0 = 0
q1[1:6] = 1, 1, 0, 0, 1, 0      out1 = 0

```

```

      175
q0[1:6] = 0, 1, 1, 0, 0, 1      out0 = 1
q1[1:6] = 0, 1, 1, 0, 0, 1      out1 = 1

```

```

      185
q0[1:6] = 1, 0, 1, 1, 0, 0      out0 = 1
q1[1:6] = 1, 0, 1, 1, 0, 0      out1 = 1

```

```

      195
q0[1:6] = 1, 1, 0, 1, 1, 0      out0 = 1
q1[1:6] = 1, 1, 0, 1, 1, 0      out1 = 1

```

```

      205
q0[1:6] = 1, 1, 1, 0, 1, 1      out0 = 0
q1[1:6] = 1, 1, 1, 0, 1, 1      out1 = 0

```

Here is the comparison result of Synthesis for (c):

looping version:

estimated period - 7.590ns

number of gates: 6 (this includes 4 buffer gates, which do not perform logical computation)

non-looping version:

estimated period - 7.590ns

number of gates: 6 (also includes 4 buffers)

Synplify Pro synthesized both designs identically. Note that the non-looping version simplifies the Verilog description, as C[1:6] is fixed in this case.

6. Assert Module**(15%)**

Here is possible solution to the assert module:

```
// Implement as an FSM: the following are the states:
`define state_idle      2'b00 // IDLE state
`define state_a        2'b01 // only A has been seen
`define state_b        2'b10 // only B has been seen
`define state_ab_or_ba 2'b11 // A and B have been seen (in either order)

module assert_AorBthenC(A, B, C, clock, reset);
    parameter msg0 = "Timeout error.",
               msg1 = "ERROR: Expected A.",
               msg2 = "ERROR: Expected B.",
               msg3 = "ERROR: Expected C.",
               msg4 = "ERROR: Expected A or B.",
               msg5 = "ERROR: More than one event asserted at once.",
               assert_name = "AorBthenC";

    input A, B, C, clock, reset;
    reg [1:0] state;
    reg [4:0] counter;

    always @(posedge clock or negedge reset) begin
        if (reset == 0) begin
            state <= `state_idle;
            counter <= 5'b0;
        end

        else begin
            // check for timeout error
            if (counter == 5'b10101) begin
                $display("\t%s %s", assert_name, msg0);
                state <= `state_idle;
                counter <= 0;
            end
            // check no more than one of A B C is true. */
            else if (((A&B) | (B&C) | (A&C)) == 1) begin
                $display("\t%s %s", assert_name, msg5);
                state <= `state_idle;
            end

            else begin
                case (state)
                    `state_idle: begin
                        if (A == 1'b1) begin
                            counter <= 1;
                            state <= `state_a;
                        end
                        else if (B == 1'b1) begin
                            counter <= 1;
                            state <= `state_b;
                        end
                        else if (C == 1'b1) begin
```

```

                                $display("\t%s  %s",  assert_name,
msg4);
                                state <= `state_idle;
                                end
                                end // case: `state_idle

`state_a: begin
    counter <= counter+1;
    if (A == 1'b1 || C == 1'b1) begin
msg2);
                                $display("\t%s  %s",  assert_name,
                                state <= `state_idle;
                                counter <= 0;
                                end
                                else if (B == 1'b1)
                                    state <= `state_ab_or_ba;
                                end

`state_ab_or_ba: begin
    counter <= counter+1;
    if (A == 1'b1 || B == 1'b1) begin
msg3);
                                $display("\t%s  %s",  assert_name,
                                state <= `state_idle;
                                counter <= 0;
                                end

                                else if (C == 1'b1) begin
                                    $display("\tsuccess!");
                                    state <= `state_idle;
                                    counter <= 0;
                                end
                                end

`state_b: begin
    counter <= counter+1;
    if (B == 1'b1 || C == 1'b1) begin
msg1);
                                $display("\t%s  %s",  assert_name,
                                state <= `state_idle;
                                counter <= 0;
                                end
                                else if (A == 1'b1)
                                    state <= `state_ab_or_ba;
                                end

`state_ab_or_ba: begin
    counter <= counter+1;
    if (A == 1'b1 || B == 1'b1) begin
msg3);
                                $display("\t%s  %s",  assert_name,
                                state <= `state_idle;
                                counter <= 0;
                                end
                                end

```

```

        else if (C == 1'b1) begin
            state <= `state_idle;
            counter <= 0;
        end
    end // case: `state_ab_or_ba
endcase // case(state)
end // else: !if((A&B) | (B&C) | (A&C)) == 1)
end // else: !if(reset == 0)
end // always @ (posedge clock or negedge reset)
endmodule // assert_AorBthenC

module testbench();
    reg A, B, C, clock, reset;

    assert_AorBthenC test_module(A, B, C, clock, reset);

    // Oscillate the clock every #5
    always
        #5 clock = ~clock;

    initial begin
        // $monitor($time,, "reset = %b, clock = %b, A = %b, B = %b,
        C = %b", reset, clock, A, B, C);
        clock = 0;
        reset = 0;
        A <= 0;
        B <= 0;
        C <= 0;
        #10 reset = 1;

        $display("\tTesting a valid sequence");
        #15;
        A <= 1;
        #10 A <= 0;
        #10 B <= 1;
        #10 B <= 0;
        #20 C <= 1;
        #10 C <= 0;
        #20;

        $display("\tTesting for timeout error");
        A <= 1;
        #10 A <= 0;
        #240;

        $display("\tTesting for error when expecting A");
        B <= 1;
        #10 B <= 0;
        #10 C <= 1;
        #10 C <= 0;
        #10;

        $display("\tTesting for error when expecting B");
        A <= 1;

```

```

#10 A <= 0;
#10 C <= 1;
#10 C <= 0;
#10;

$display("\tTesting for error when expecting C");
A <= 1;
#10 A <= 0;
#10 B <= 1;
#10 B <= 0;
#10 A <= 1;
#10 A <= 0;
#10;

$display("\tTesting for error when expecting A or B");
C <= 1;
#10 C <= 0;
#10;

$display("\tTesting for error by asserting more than one at
a time");
#10;
A <= 1;
B <= 1;
#10;
A <= 0;
B <= 0;
#10 $finish;
end // initial begin
endmodule

```

Here is the simulation result

```

Testing a valid sequence
  success!
Testing for timeout error
AorBthenC Timeout error.
Testing for error when expecting A
AorBthenC ERROR: Expected A.
Testing for error when expecting B
AorBthenC ERROR: Expected B.
Testing for error when expecting C
AorBthenC ERROR: Expected C.
Testing for error when expecting A or B
AorBthenC ERROR: Expected A or B.
Testing for error by asserting more than one at a time
AorBthenC ERROR: More than one event asserted at once.

```

7. Half-Empty Case

(15%)

The given FSM can be expressed in verilog as follows, either using full/parallel case or no case attributes:

```

module oneHotFullParallel(y1, y0, x1, x0, clock);
    input    x1, x0, clock;
    output   y1, y0;
    reg      y1, y0;
    reg [3:0] state;

    always @(posedge clock)
        begin

            // full_case parallel_case
            casex(state)
                // A
                'b???1: begin
                    if ({x1, x0} == 2'b00) begin
                        {y1, y0} <= 2'b01;
                        state <= 4'b0010;
                    end
                    else begin
                        {y1, y0} <= 2'b10;
                        state <= 4'b1000;
                    end
                end
            end
            // B
            4'b??1?: begin
                if ({x1, x0} == 2'b00) begin
                    {y1, y0} <= 2'b00;
                    state <= 4'b0100;
                end
                else begin
                    {y1, y0} <= 2'b10;
                    state <= 4'b0010;
                end
            end
            end
            // C
            4'b?1??: begin
                if (x0 == 1'b0) begin
                    {y1, y0} <= 2'b01;
                    state <= 4'b0001;
                end
                else begin
                    {y1, y0} <= 2'b11;
                    state <= 4'b0100;
                end
            end
            end
            // D
            4'b1???: begin
                if (x1 == 1'b0) begin
                    {y1, y0} <= 2'b00;
                    state <= 4'b0100;
                end
                else begin
                    {y1, y0} <= 2'b11;
                    state <= 4'b0001;
                end
            end
            end
        end
end

```

```

        end
        endcase // case(state)
    end // always @ (posedge clock)
endmodule // oneHotFullParallel

module oneHotNoCase(y1, y0, x1, x0, clock);
    input    x1, x0, clock;
    output   y1, y0;
    reg      y1, y0;
    reg [3:0] state;

    always @(posedge clock)
        begin

            // no_case
            case(state)
                // A
                'b0001: begin
                    if ({x1, x0} == 2'b00) begin
                        {y1, y0} <= 2'b01;
                        state <= 4'b0010;
                    end
                    else begin
                        {y1, y0} <= 2'b10;
                        state <= 4'b1000;
                    end
                end
            end
            // B
            4'b0010: begin
                if ({x1, x0} == 2'b00) begin
                    {y1, y0} <= 2'b00;
                    state <= 4'b0100;
                end
                else begin
                    {y1, y0} <= 2'b10;
                    state <= 4'b0010;
                end
            end
            end
            // C
            4'b0100: begin
                if (x0 == 1'b0) begin
                    {y1, y0} <= 2'b01;
                    state <= 4'b0001;
                end
                else begin
                    {y1, y0} <= 2'b11;
                    state <= 4'b0100;
                end
            end
            end
            // D
            4'b1000: begin
                if (x1 == 1'b0) begin
                    {y1, y0} <= 2'b00;
                    state <= 4'b0100;
                end
            end
        end
endmodule

```

```
        end
        else begin
            {y1, y0} <= 2'b11;
            state <= 4'b0001;
        end
    end
endcase // case(state)
end // always @ (posedge clock)
endmodule // oneHotNoCase
```

It turns out that using don't care full/parallel case attributes optimizes the design with 41 instances. Using no case attributes doesn't optimize the design and requires 60 instances. See the attached schematics.