EECS 151/251A Homework 2

Due Monday, Feb 6th, 2023

For this HW Assignment

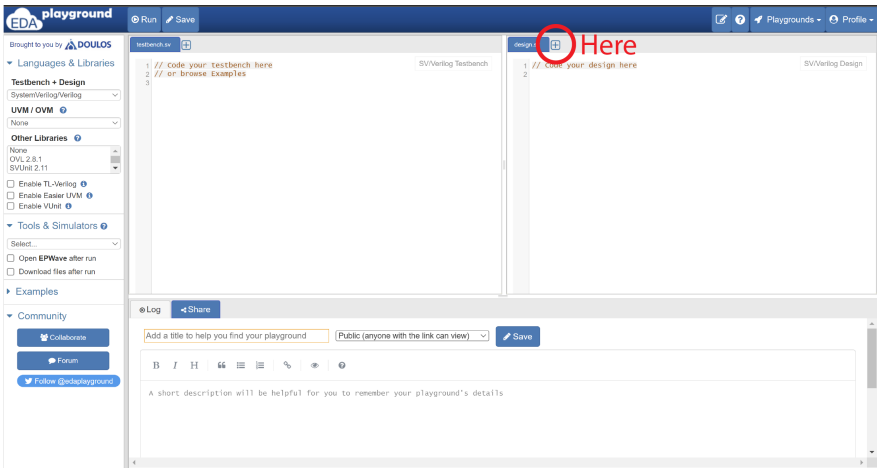
You will be asked to write several Verilog modules as part of this HW assignment. You will need to test your modules by running them through a simulator. As shown in discussion 2, a highly suggested simulator is `https://www.edaplayground.com` which is a free, online, Verilog simulator.

For all problems except 2(b), turn in:

1. Circuit diagram (neatly drawn by hand or with a tool). When drawing there are some rules we'd like you to follow:
   - Draw solder dots at wire junctions
   - Label bus widths for multi-bit wires
2. Verilog code
3. Testbench
4. Test result

***Warning*: We enforce no register inference policy in this class. You must use the register library in `EECS151.v` whenever using registers in your Verilog. `EECS151.v` is located at `https://inst.eecs.berkeley.edu/~eecs151/sp23/files/lib/EECS151.v`. We will only accept solutions using this library!**

To import the library on *EDA playground*, click the plus button shown below and upload the file. Then, add `` `include "EECS151.v" `` to your Verilog code (`design.sv`). You can import other modules (such as modules you designed before) in the same way.



Problem 1: Binary Decoder and Encoder

In this problem except (a)3, you are not allowed to use any of the following operators: {`!`, `~`, `&`, `|`, `~&`, `~|`, `*`, `&&`, `||`}.

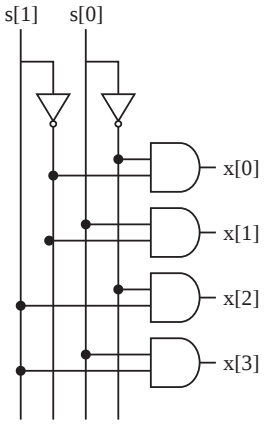
(a) Write five different Verilog implementations of a 2-bit binary decoder, based on the formats listed below. For the first one (structural Verilog), also draw a gate-level circuit diagram. Write a single testbench that exhaustively tests all of your implementations (put all of them in `design.sv` on *EDA playground*).

1. Structural Verilog
2. Behavioral Verilog using continuous assignment (i.e. no `always` blocks)
3. Behavioral Verilog using continuous assignment and bit-wise operators
4. Behavioral Verilog using `if`
5. Behavioral Verilog using `case`

(b) Write the behavioral Verilog using continuous assignment and the ternary operator (?:), for a 2-bit binary *encoder*. (A encoder has the inverse function of a decoder.) The input is guaranteed to be one-hot. Write a testbench using one of the decoders you designed (import the decoder in the testbench).

Solution:

(a) Diagram:



Design:

1. Structural Verilog

```
module decoder1(
                input [1:0]  s,
                output [3:0] x
                );

   wire                      s0bar, s1bar;
```

```
    not(s0bar, s[0]);
    not(s1bar, s[1]);

    and(x[0], s0bar, s1bar);
    and(x[1], s[0],  s1bar);
    and(x[2], s0bar, s[1]);
    and(x[3], s[0],  s[1]);

endmodule
```

2. Behavioral Verilog using continuous assignment

```
module decoder2(
                input [1:0]  s,
                output [3:0] x
                );

    assign x[0] = (s == 2'b00);
    assign x[1] = (s == 2'b01);
    assign x[2] = (s == 2'b10);
    assign x[3] = (s == 2'b11);

endmodule
```

3. Behavioral Verilog using continuous assignment and bit-wise operators

```
module decoder3(
                input [1:0]  s,
                output [3:0] x
                );

    assign x[0] = ~s[1] & ~s[0];
    assign x[1] = ~s[1] &  s[0];
    assign x[2] =  s[1] & ~s[0];
    assign x[3] =  s[1] &  s[0];

endmodule
```

4. Behavioral Verilog using `if`

```
module decoder4(
                input [1:0]      s,
                output reg [3:0] x
                );

    always @(*) begin
        x = 4'b0;
        if(s == 2'b00)
            x[0] = 1;
        if(s == 2'b01)
```

```
            x[1] = 1;
        if(s == 2'b10)
            x[2] = 1;
        if(s == 2'b11)
            x[3] = 1;
    end

endmodule
```

5. Behavioral Verilog using `case`

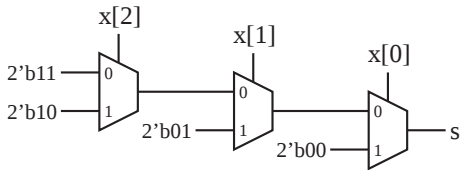
```
module decoder5(
                input [1:0]      s,
                output reg [3:0] x
                );

    always @(*) begin
        case(s)
          2'b00: x = 4'b0001;
          2'b01: x = 4'b0010;
          2'b10: x = 4'b0100;
          2'b11: x = 4'b1000;
        endcase
    end

endmodule
```

Testbench:

```
module decoder_tb;
    reg [1:0] s;
    reg [3:0] expected;
    wire [3:0] xs [0:4];

    // loop variables
    integer    i, j;

    // instantiate duts
    decoder1 d1(.s(s), .x(xs[0]));
    decoder2 d2(.s(s), .x(xs[1]));
    decoder3 d3(.s(s), .x(xs[2]));
    decoder4 d4(.s(s), .x(xs[3]));
    decoder5 d5(.s(s), .x(xs[4]));

    // expected outputs
    always @(*) begin
        case (s)
          2'b00:   expected = 4'b0001;
          2'b01:   expected = 4'b0010;
```

```
            2'b10:   expected = 4'b0100;
            2'b11:   expected = 4'b1000;
            default: expected = 4'bxxxx;
         endcase
      end

      // begin test
      initial begin
         $dumpfile("dump.vcd");
         $dumpvars;
         for(i = 0; i < 4; i = i + 1) begin
            s = i;
            #1;
            $display("s: %b, xs[0]: %b, xs[1]: %b, xs[2]: %b, xs[3]: %b,
↪  xs[4]: %b, expected: %b",
                     s, xs[0], xs[1], xs[2], xs[3], xs[4], expected);
            // Break early if failed
            for(j = 0; j < 5; j = j + 1) begin
               if(xs[j] !== expected) begin
                  $display("FAILED at decoder%1d, expected %b, got %b",
                           j + 1, expected, xs[j]);
                  $finish();
               end
            end
         end
         $display("ALL TESTS PASSED FOR ALL DESIGNS!");
         $finish();
      end
   endmodule
```

(b) Diagram:



Design:

```
module encoder(
               input [3:0]  x,
               output [1:0] s
               );

   assign s = (x[0] == 1'b1)? 2'b00:
              (x[1] == 1'b1)? 2'b01:
```

```
                (x[2] == 1'b1)? 2'b10:
                                2'b11;

endmodule
```

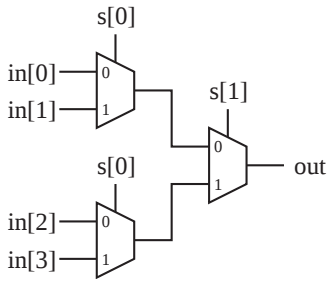
Testbench:

```
`include "decoder.v"

module encoder_tb;
   reg [1:0] s;
   reg [1:0] expected;
   wire [3:0] x;
   wire [1:0] out;

   // loop variables
   integer    i, j;

   // instantiate duts
   decoder1 d1(.s(s), .x(x));
   encoder enc(.x(x), .s(out));

   // expected outputs
   always @(*) begin
      expected = s;
   end

   // begin test
   initial begin
      $dumpfile("dump.vcd");
      $dumpvars;
      for(i = 0; i < 4; i = i + 1) begin
         s = i;
         #1;
         $display("s: %b, x: %b, out: %b, expected: %b",
                  s, x, out, expected);
         // Break early if failed
         if(out !== expected) begin
            $display("FAILED, expected %b, got %b",
                     expected, out);
            $finish();
         end
      end
      $display("ALL TESTS PASSED!");
      $finish();
   end
endmodule
```

Problem 2: Decoder-Based Multiplexer

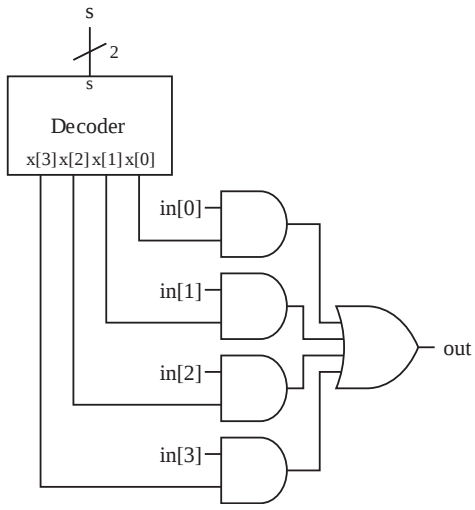
(a) Design a 4-to-1 multiplexer using one of the decoders you designed above. The select signals must be input to the decoder and must not be used anywhere else. Provide an exhaustive test.

(b) What could be a potential benefit of using this decoder-based multiplexer against the following design:



Solution:

(a) Diagram:



Design:

```
`include "decoder.v"

module multiplexer(
                   input [1:0] s,
                   input [3:0] in,
                   output      out
                   );

   wire [3:0]                  x;
```

```
    wire                        t0, t1, t2, t3;

    decoder1 dc1(.s(s), .x(x));

    and(t0, in[0], x[0]);
    and(t1, in[1], x[1]);
    and(t2, in[2], x[2]);
    and(t3, in[3], x[3]);
    or(out, t0, t1, t2, t3);

endmodule
```

Testbench:

```
module multiplexer_tb;
    reg [1:0] s;
    reg [3:0] in;
    reg       expected;
    wire      out;

    // loop variables
    integer   i, j;

    // instantiate duts
    multiplexer mux1(.s(s), .in(in), .out(out));

    // expected outputs
    always @(*) begin
        case (s)
          2'b00:   expected = in[0];
          2'b01:   expected = in[1];
          2'b10:   expected = in[2];
          2'b11:   expected = in[3];
          default: expected = 1'bx;
        endcase
    end

    // begin test
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        for(j = 0; j < 16; j = j + 1) begin
            in = j;
            for(i = 0; i < 4; i = i + 1) begin
                s = i;
                #1;
                $display("s: %b, in: %b, out: %b, expected: %b",
```

```
                    s, in, out, expected);
            // Break early if failed
            if(out !== expected) begin
               $display("FAILED, expected %b, got %b",
                        expected, out);
               $finish();
            end
         end
      end
      $display("ALL TESTS PASSED!");
      $finish();
   end
endmodule
```

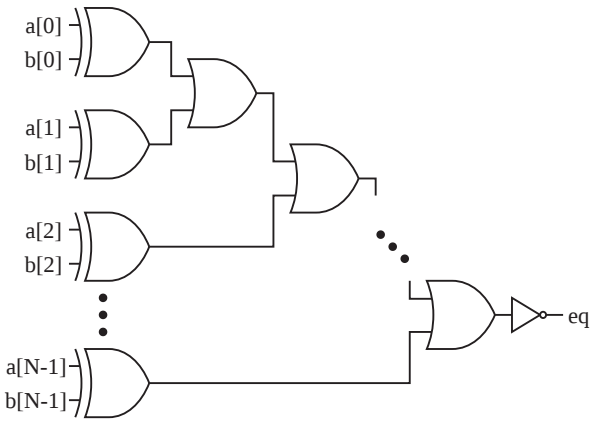
(b) Smaller delay (fewer logic levels) from data input to output, trading off delay from select signals to output.

Problem 3: Equality Comparator Generator

An equality comparator is a combinational logic circuit that takes as input two N-bit signals and outputs 1 iff the two signals match in every bit position. Design a generator for equality comparators of size N in the structural Verilog. Write a testbench that tests $N = 1$ and $N = 4$ exhaustively.

Solution:

Diagram:



Design:

```
module eq #(
            parameter N = 4
            ) (
```

```
                input [N-1:0] a,
                input [N-1:0] b,
                output        eq
                );

   wire [N-1:0]               s;
   wire [N-1:0]               t;

   genvar                     i;

   generate
      for(i = 0; i < N; i = i + 1)
        xor(s[i], a[i], b[i]);
   endgenerate

   buf(t[0], s[0]);
   generate
      for(i = 1; i < N; i = i + 1)
        or(t[i], s[i], t[i-1]);
   endgenerate

   not(eq, t[N-1]);

endmodule
```

Testbench:

```
module eq_tb;
   reg a1, b1;
   reg [3:0] a4, b4;
   reg       expected1, expected4;
   wire      out1, out4;

   // loop variables
   integer   i, j;

   // instantiate duts
   eq #(.N(1)) eq1 (.a(a1), .b(b1), .eq(out1));
   eq #(.N(4)) eq4 (.a(a4), .b(b4), .eq(out4));

   // expected outputs
   always @(*) begin
      expected1 = (a1 == b1);
      expected4 = (a4 == b4);
   end
```

```
    // begin test
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        for(i = 0; i < 2; i = i + 1) begin
            for(j = 0; j < 2; j = j + 1) begin
                a1 = i;
                b1 = j;
                #1;
                $display("a: %b, b: %b, out: %b, expected: %b",
                         a1, b1, out1, expected1);
                if(out1 !== expected1) begin
                    $display("FAILED, expected %b, got %b",
                             expected1, out1);
                    $finish();
                end
            end
        end
        for(i = 0; i < 16; i = i + 1) begin
            for(j = 0; j < 16; j = j + 1) begin
                a4 = i;
                b4 = j;
                #1;
                $display("a: %b, b: %b, out: %b, expected: %b",
                         a4, b4, out4, expected4);
                if(out4 !== expected4) begin
                    $display("FAILED, expected %b, got %b",
                             expected4, out4);
                    $finish();
                end
            end
        end
        $display("ALL TESTS PASSED FOR ALL DESIGNS!");
        $finish();
    end
endmodule
```

Problem 4: Counter Generator

Write a Verilog implementation for an N-bit counter generator for counters with the following specification. Write a testbench for $N = 4$.

Specification:

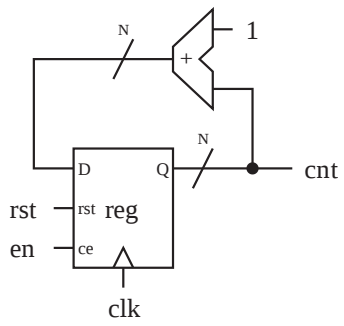
- Counters have as input `clk` (the clock signal), `rst` (reset), and `en` (enable).
- Counters have an N-bit output named `cnt`.

- Counters hold the value of `cnt` constant when both `rst` and `en` are 0.
- Counters set `cnt` to 0 on a positive edge of `clk` if `rst` is 1.
- Counters increment `cnt` by 1 at a positive edge of `clk` if `rst` is 0 and `en` is 1.
- When `cnt` is the maximum possible value ($2^N - 1$ in N-bit counters), `cnt` will become 0 next time it is incremented.

Warning: Use the register library in `EECS151.v`.

Solution:

Diagram:



Design:

```
`include "EECS151.v"

module counter #(
                  parameter N = 4
                  ) (
                     input          clk, rst, en,
                     output [N-1:0] cnt
                     );

   wire [N-1:0]                     cnt_inc;

   REGISTER_R_CE #(.N(N)) r(.q(cnt), .d(cnt_inc), .rst(rst), .ce(en),
↪   .clk(clk));

   assign cnt_inc = cnt + 1;

endmodule
```

Testbench:

```
module counter_tb;
   reg clk, rst, en;
   reg [3:0] expected;
   wire [3:0] out;

   // loop variables
   integer    i;

   // define clock
   initial clk = 0;
   always #1 clk = !clk;

   // instantiate duts
   counter #(.N(4)) c1(.clk(clk), .rst(rst), .en(en), .cnt(out));

   // task to compare the output with its expected value
   task check;
      begin
         #2;
         // verify in the timestep clock is going down
         $display("rst: %b, en: %b, out: %b, expected: %b",
                  rst, en, out, expected);
         if(out !== expected) begin
            $display("FAILED, expected %b, got %b",
                     expected, out);
            $finish();
         end
      end
   endtask

   // begin test
   initial begin
      $dumpfile("dump.vcd");
      $dumpvars;
      // check reset works regardless enable
      rst = 1;
      en = 1'bx;
      expected = 4'b0;
      check;
      // check increment works
      rst = 0;
      en = 1;
      for(i = 0; i < 16; i = i + 1) begin
         expected = expected + 1;
         check;
      end
      // check enable works
```

```
        for(i = 0; i < 16; i = i + 1) begin
            // count doesn't change if enable is 0
            en = 0;
            check;
            check;
            check;
            // count is incremented if enable is 1
            en = 1;
            expected = expected + 1;
            check;
        end
        $display("ALL TESTS PASSSED!");
        $finish();
    end
endmodule
```

Problem 5: Serial To Parallel Converter

Imagine a situation that you need to design an interface between two circuits A and B. A generates one-bit data at a time, while B receives 10-bit data in parallel. These circuits also partially implement the hand-shaking protocol. A raises `SerRdy` when the data starts coming out. On the other hand, B reads the data when `ParRdy` is on. Assume all circuits (including the one you are going to design) share the clock signal `clk`.

(a) Using the equality comparator generator and the counter generator you designed, design a saturating counter that saturates at 9 (it works the same as the original counter up to 9, but then it will get stuck at 9 until the reset signal is activated). Equip it with an extra output port that tells whether it is saturated (1) or not (0).

(b) Design a 10-bit shift-register with enable (it works as a shift-register if enable is 1, otherwise it just holds the current value).

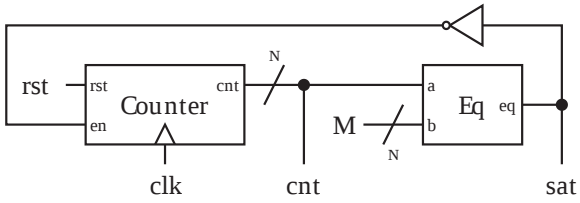
(c) Design the interface circuit using the saturating counter and the shift-register. Its detailed specification is shown below. You may assume that `SerRdy` is 1 at the first positive edge.

Specification:

- Receives three binary inputs: `SerDat`, `SerRdy`, and `clk`.
- Generates two outputs: a 10-bit signal `ParDat` and a binary signal `ParRdy`.
- Stores a 10-bit sequence that `SerDat` takes at each positive edge of `clk` after `SerRdy` becomes 1.
- It is guaranteed that once `SerRdy` becomes 1, it will remain 1 until next positive edge of `clk`. Just after that, it will drop to 0 and remain 0 for at least 10 cycles.
- Turns on `ParRdy` when finished storing the 10-bit sequence, which is output as `ParDat`.
- Turns off `ParRdy` when next sequence comes in.

Solution:

(a) Diagram:



Design:

```
`include "counter.v"
`include "eq.v"

module sat_counter #(
                     parameter N = 4,
                     parameter M = 4'd9
                     ) (
                        input           clk, rst,
                        output [N-1:0] cnt,
                        output          sat
                        );

   wire                                 en;

   counter #(.N(N)) c1(clk, rst, en, cnt);
   eq #(.N(N)) eq1(cnt, M, sat);
   not(en, sat);

endmodule
```

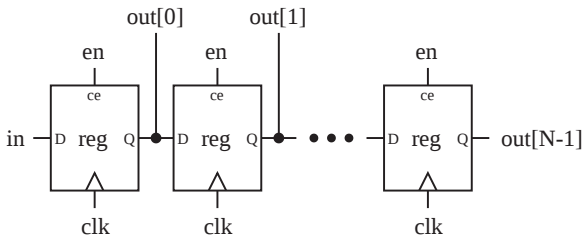
Testbench:

```
module sat_counter_tb;
   reg clk, rst;
   reg [3:0] expected;
   reg       expected_sat;
   wire [3:0] out;
   wire       sat;

   // loop variables
   integer    i;

   // define clock
   initial clk = 0;
```

```
    always #1 clk = !clk;

    // instantiate duts
    sat_counter #(.N(4), .M(4'd9)) c1(.clk(clk), .rst(rst), .cnt(out),
↪  .sat(sat));

    task check;
        begin
            #2;
            // verify in the timestep clock is going down
            $display("rst: %b, out: %b, sat: %d, expected: %b, expected_sat:
↪  %b",
                     rst, out, sat, expected, expected_sat);
            if(out !== expected) begin
                $display("FAILED at out, expected %b, got %b",
                         expected, out);
                $finish();
            end
            if(sat !== expected_sat) begin
                $display("FAILED at sat, expected %b, got %b",
                         expected_sat, sat);
                $finish();
            end
        end
    endtask

    // begin test
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        // initial reset
        rst = 1;
        expected = 4'b0;
        expected_sat = 0;
        check;
        // check increment works before saturation
        rst = 0;
        for(i = 0; i < 8; i = i + 1) begin
            expected = expected + 1;
            check;
        end
        // check it saturates
        expected = expected + 1;
        expected_sat = 1;
        check;
        check;
        check;
```

```
        check;
        // check reset works after saturation
        rst = 1;
        expected = 4'b0;
        expected_sat = 0;
        check;
        $display("ALL TESTS PASSSED!");
        $finish();
    end
endmodule
```

(b) Diagram:



Design:

```
`include "EECS151.v"

module shift_reg #(
                    parameter N = 10
                    )(
                      input           clk, in, en,
                      output [N-1:0] out
                      );

    wire [N-1:0]                    d;

    REGISTER_CE #(.N(N)) r(.q(out), .d(d), .ce(en), .clk(clk));

    assign d = {out, in};

endmodule
```

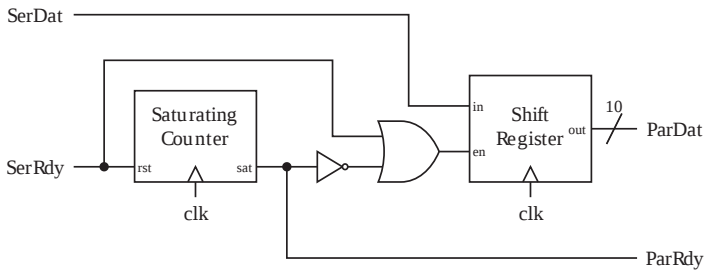
Testbench:

```
module sat_counter_tb;
    reg clk, in, en;
    reg [9:0] expected;
    wire [9:0] out;

    // loop variables
```

```
integer    i;

// define clock
initial clk = 0;
always #1 clk = !clk;

// instantiate duts
shift_reg #(.N(10)) c1(.clk(clk), .in(in), .en(en), .out(out));

task check;
   begin
      #2;
      // verify in the timestep clock is going down
      $display("in: %b, en: %b, out: %b, expected: %b",
               in, en, out, expected);
      if(out !== expected) begin
         $display("FAILED, expected %b, got %b",
                  expected, out);
         $finish();
      end
   end
endtask

// begin test
initial begin
   $dumpfile("dump.vcd");
   $dumpvars;
   // fill register with random numbers
   en = 1;
   for(i = 0; i < 9; i = i + 1) begin
      in = $random % 2;
      expected = {expected[8:0], in};
      #2;
   end
   in = $random % 2;
   expected = {expected[8:0], in};
   check;
   // check enable
   en = 0;
   check;
   check;
   check;
   // check 10 more cycles with random numbers
   en = 1;
   for(i = 0; i < 10; i = i + 1) begin
      en = 1;
      in = $random % 2;
```

```
            expected = {expected[8:0], in};
            check;
        end
        $display("ALL TESTS PASSSED!");
        $finish();
    end
endmodule
```

(c) Diagram:



Design:

```
`include "shift_reg.v"
`include "sat_counter.v"

module parallelizer(
                    input        clk,
                    input        SerDat, SerRdy,
                    output [9:0] ParDat,
                    output       ParRdy
                    );

    // shift-register to store the value of SerDat
    wire                         en;
    shift_reg #(.N(10)) sr(.clk(clk), .in(SerDat), .en(en), .out(ParDat));

    // saturating counter to count 10 - 1 cycles after SerRdy becomes on
    wire                         sat;
    sat_counter #(.N(4), .M(4'd9)) sc(.clk(clk), .rst(SerRdy), .sat(sat));

    // if counter is saturated, ParDat is ready
    assign ParRdy = sat;

    // if SerRdy is on (first bit) or counter is not saturated (following
↪    bits), enable shift-register
    assign en = SerRdy | !ParRdy;

endmodule
```

Testbench:

```
module parallelizer_tb;
   reg clk;
   reg SerDat, SerRdy;
   reg [9:0] expected;
   reg       expected_ParRdy;
   wire [9:0] ParDat;
   wire       ParRdy;

   // loop variables
   integer    i;

   // define clock
   initial clk = 0;
   always #1 clk = !clk;

   // instantiate duts
   parallelizer p1(.clk(clk), .SerDat(SerDat), .SerRdy(SerRdy),
↪  .ParDat(ParDat), .ParRdy(ParRdy));

   task check;
      begin
         #2;
         // verify in the timestep clock is going down
         $display("SerDat: %b, SerRdy: %b, ParDat: %b, ParRdy: %b,
↪  expected: %b, expected_ParRdy: %b",
                  SerDat, SerRdy, ParDat, ParRdy, expected,
↪  expected_ParRdy);
         // ParRdy must be always correct
         if(ParRdy !== expected_ParRdy) begin
            $display("FAILED at ParRdy, expected %b, got %b",
                     expected_ParRdy, ParRdy);
            $finish();
         end
         // ParDat must be correct if ParRdy
         if(ParRdy && ParDat !== expected) begin
            $display("FAILED at ParDat, expected %b, got %b",
                     expected, ParDat);
            $finish();
         end
      end
   endtask

   // begin test
   initial begin
      $dumpfile("dump.vcd");
```

```
        $dumpvars;
        // trigger SerRdy
        SerRdy = 1;
        SerDat = $random % 2;
        expected = {expected[8:0], SerDat};
        expected_ParRdy = 0;
        check;
        SerRdy = 0;
        // check behavior after SerRdy before ParRdy
        for(i = 0; i < 8; i = i + 1) begin
           SerDat = $random % 2;
           expected = {expected[8:0], SerDat};
           check;
        end
        // check behavior when turning on ParRdy
        expected_ParRdy = 1;
        SerDat = $random % 2;
        expected = {expected[8:0], SerDat};
        check;
        // check behavior after ParRdy
        for(i = 0; i < 20; i = i + 1) begin
           SerDat = $random % 2;
           check;
        end
        // trigger SerRdy again
        SerRdy = 1;
        SerDat = $random % 2;
        expected = {expected[8:0], SerDat};
        expected_ParRdy = 0;
        check;
        SerRdy = 0;
        // check behavior after SerRdy before ParRdy
        for(i = 0; i < 8; i = i + 1) begin
           SerDat = $random % 2;
           expected = {expected[8:0], SerDat};
           check;
        end
        // check behavior when turning on ParRdy
        expected_ParRdy = 1;
        SerDat = $random % 2;
        expected = {expected[8:0], SerDat};
        check;
        $display("ALL TESTS PASSSED!");
        $finish();
     end
  endmodule
```