EECS 151/251A Homework 2

Due Monday, Feb 8th, 2021

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** 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 N-bit wires
2. Verilog code, test bench, and test results.
3. A short (1-2 sentence) description of your code and how you arrived at your solution.

Warning: As per our EECS151 "no register inference policy", you will need to use the register library in `EECS151.v` when using registers in your Verilog. `EECS151.v` is located at `https://inst.eecs.berkeley.edu/~eecs151/sp21/files/lib/EECS151.v`. We will only accept solutions using this library!

You can import this library by adding `` `include "EECS151.v" `` to your Verilog code.

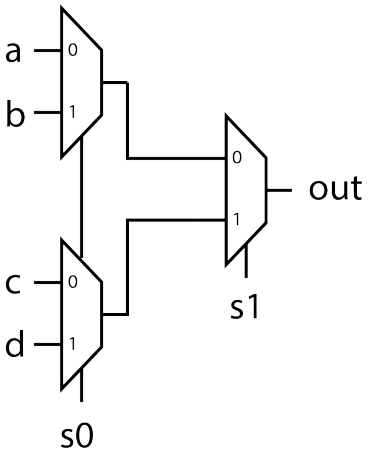
Examples of creating a testbench can be found in discussion 2 slides `https://inst.eecs.berkeley.edu/~eecs151/sp21/files/discussion2.pdf`.

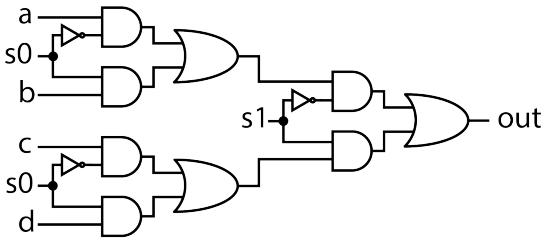
Problem 1: Multiplexer

Design a 4-to-1 multiplexer using only continuous assignment (i.e. no `always` blocks). Provide an exhaustive test.

Solution:

Diagram: Both approaches are acceptable.





Design: There are two possible solutions:

```
module mux(a, b, c, d, s, out);
    input a, b, c, d;
    input [1:0] s;
    output out;

    wire int_a, int_b;

    assign int_a = s[0] ? b : a;
    assign int_b = s[0] ? d : c;
    assign out = s[1] ? int_b : int_a;
endmodule
```

Alternatively,

```
module mux(a, b, c, d, s, out);
    input a, b, c, d;
    input [1:0] s;
    output out;

    wire int_a, int_b;

    assign int_a = (a & ~s[0]) | (b & s[0]);
    assign int_b = (c & ~s[0]) | (d & s[0]);
    assign out = (int_a & ~s[1]) | (int_b & s[1]);
endmodule
```

Testbench: (You won't need the fancy PASS/FAIL checking I'm doing here, but great job if you do include them!)

```
module mux_tb;
  reg [3:0] in_vec;
  reg [1:0] s;
  reg expected;
  wire out;

  // loop variables
  integer i, j;

  // instantiate dut
  mux dut (.a(in_vec[3]),
           .b(in_vec[2]),
           .c(in_vec[1]),
           .d(in_vec[0]),
           .s(s),
           .out(out));

  // make golden LUT
  always @(*) begin
    case (s)
      2'b00:    expected = in_vec[3];
      2'b01:    expected = in_vec[2];
      2'b10:    expected = in_vec[1];
      2'b11:    expected = in_vec[0];
      default:  expected = 1'b0;
    endcase
  end

  // begin test
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    in_vec = 4'b0000;
           s = 2'b00;
    for(i = 0; i < 16; i = i + 1) begin
      for (j = 0; j < 4; j = j + 1) begin
              in_vec = i;
        s = j;
        $strobe("a: %b, b: %b, c: %b, d: %b, s: %b, out: %b",
                in_vec[3], in_vec[2], in_vec[1], in_vec[0], s, out);
        #1;
        // Break early if test failed
        if (out != expected) begin
```

```
            $display("FAILED, expected %b, got %b", expected, out);
            $finish();
          end
        end
      end
      $display("ALL TESTS PASSSED!");
      $finish();
    end
  endmodule
```

Problem 2: More Multiplexers!

Consider a 6-to-1 multiplexer, with a 6-bit wide data input `X`, 3-bit wide `select` input, and single bit output `y`. If `select` is 000, then `y=x[0]`, if `select=001`, then `y=x[1]`, etc. When `select=110` or `select=111`, the output is unspecified (i.e. "don't care").

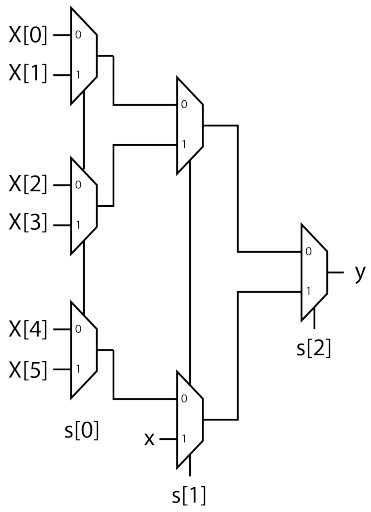
(a) Using the definition of a 2-to-1 multiplexer from the class notes, define a hierarchical digital system to represent this 6-to-1 mux.

(b) Repeat (a), but use a `case` block with a flat (i.e. no instantiation) module.

You can submit one testbench that tests both designs together.

Solution:

Diagram:



(a) Design:

```
module mux_2_hier(
  input [1:0] X,
  input sel,
  output y);

  assign y = sel ? X[1] : X[0];
endmodule

module mux_4_hier(
  input [3:0] X,
  input [1:0] sel,
  output y);

  wire int_AA, int_AB;

  mux_2_hier mux_AA(.X(X[3:2]), .sel(sel[0]), .y(int_AA));
  mux_2_hier mux_AB(.X(X[1:0]), .sel(sel[0]), .y(int_AB));
  mux_2_hier mux_B(.X({int_AA, int_AB}), .sel(sel[1]), .y(y));
endmodule

module mux_6_hier(
  input [5:0] X,
  input [2:0] sel,
  output y);

  wire int_AA, int_AB;
```

```
  mux_4_hier mux_AA(.X({2'bxx,X[5:4]}), .sel(sel[1:0]), .y(int_AA));
  mux_4_hier mux_AB(.X(X[3:0]), .sel(sel[1:0]), .y(int_AB));
  mux_2_hier mux_B(.X({int_AA, int_AB}), .sel(sel[2]), .y(y));
endmodule
```

(b) Design:

```
module mux_6_flat(
  input [5:0] X,
  input [2:0] sel,
  output reg y
  );

  always @ ( * ) begin
    case (sel)
      3'b000:   y = X[0];
      3'b001:   y = X[1];
      3'b010:   y = X[2];
      3'b011:   y = X[3];
      3'b100:   y = X[4];
      3'b101:   y = X[5];
      default:  y = 1'bx;
    endcase
  end
endmodule
```

Testbench:

```
module mux_6_tb;
  reg [5:0] in_vec;
  reg [2:0] s;
  reg expected;
  wire out_hier;
  wire out_flat;

  // loop variables
  integer i, j;

  // instantiate duts
  mux_6_hier dut_hier (.X(in_vec),
                       .sel(s),
                       .y(out_hier));

  mux_6_flat dut_flat (.X(in_vec),
                       .sel(s),
                       .y(out_flat));
```

```
  // make golden LUT
  always @(*) begin
    case (s)
      3'b000:   expected = in_vec[0];
      3'b001:   expected = in_vec[1];
      3'b010:   expected = in_vec[2];
      3'b011:   expected = in_vec[3];
      3'b100:   expected = in_vec[4];
      3'b101:   expected = in_vec[5];
      default:  expected = 1'bx;
    endcase
  end

  // begin test
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    in_vec = 4'b0000;
           s = 2'b00;
    for(i = 0; i < 256; i = i + 1) begin
      for (j = 0; j < 8; j = j + 1) begin
              in_vec = i;
        s = j;
        $strobe("in_vec: %b, s: %b, out_flat: %b, out_hier: %b, expected:
        ↪  %b",
                in_vec, s, out_flat, out_hier, expected);
        #1;
        // Break early if failed; Using case equivalence to check for x
        if (out_hier !== expected) begin
          $display("HIERARCHICAL FAILED, expected %b, got %b", expected,
          ↪  out_hier);
          $finish();
        end
        if (out_flat !== expected) begin
          $display("FLAT FAILED, expected %b, got %b", expected,
          ↪  out_flat);
          $finish();
        end
      end
    end
    $display("ALL TESTS PASSED FOR ALL DESIGNS!");
    $finish();
  end
endmodule
```

Problem 3: Decoder

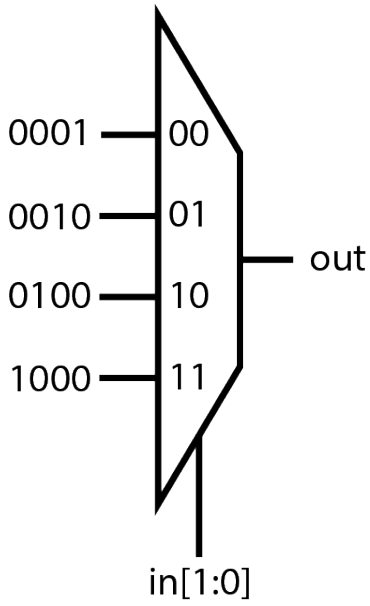
For a 2-bit binary decoder, devise two behavioral Verilog descriptions without using the `case` statement (*hint: conditional statements and Boolean Logic*).

You can submit one testbench that tests both designs together.

Solution:

(a) **Conditional Statements**

Diagram:

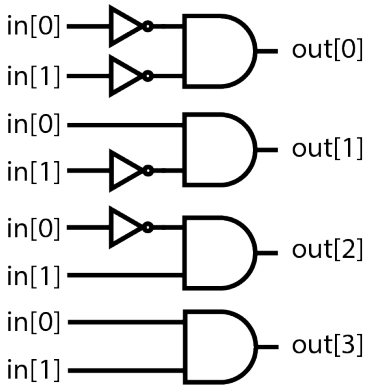


Design:

```
module decoder_2_cond(
  input [1:0] in,
  output reg [3:0] out
  );

  always @ ( * ) begin
    if (in == 2'b00) out = 4'b0001;
    if (in == 2'b01) out = 4'b0010;
    if (in == 2'b10) out = 4'b0100;
    if (in == 2'b11) out = 4'b1000;
  end
endmodule
```

(b) **Boolean Logic**

Diagram:



Design:

```
module decoder_2_bool(
  input [1:0] in,
  output [3:0] out
  );

  assign out[0] = ~in[0] & ~in[1];
  assign out[1] = in[0] & ~in[1];
  assign out[2] = ~in[0] & in[1];
  assign out[3] = in[0] & in[1];
endmodule
```

Testbench:

```
module decoder_2_tb;
  reg [1:0] in_vec;
  reg [3:0] expected;
  wire [3:0] out_cond;
  wire [3:0] out_bool;

  // loop variables
  integer i;

  // instantiate duts
  decoder_2_cond dut_cond (.in(in_vec), .out(out_cond));
  decoder_2_bool dut_bool (.in(in_vec), .out(out_bool));

  // make golden LUT
  always @(*) begin
    case (in_vec)
      2'b00:   expected = 4'b0001;
      2'b01:   expected = 4'b0010;
      2'b10:   expected = 4'b0100;
      2'b11:   expected = 4'b1000;
```

```
        default:  expected = 4'bx;
      endcase
    end

    // begin test
    initial begin
      $dumpfile("dump.vcd");
      $dumpvars;
      in_vec = 2'b00;
      for(i = 0; i < 4; i = i + 1) begin
              in_vec = i;
        $strobe("in_vec: %b, out_cond: %b, out_bool: %b, expected: %b",
                in_vec, out_cond, out_bool, expected);
        #1;
        // Break early if failed; Using case equivalence to check for x
        if (out_cond !== expected) begin
          $display("CONDITIONAL FAILED, expected %b, got %b", expected,
          ↪  out_cond);
          $finish();
        end
        if (out_bool !== expected) begin
          $display("BOOLEAN FAILED, expected %b, got %b", expected,
          ↪  out_bool);
          $finish();
        end
      end
      $display("ALL TESTS PASSED FOR ALL DESIGNS!");
      $finish();
    end
  endmodule
```

Problem 4: Serializer

Many modern high-speed communications are sent over serial links. This means that the data buses from the processor must be serialized before going to the transmit block to be sent over the link. On slide 22 of Lecture 4, there is an example of a 4-bit parallel-to-serial converter (also known as a serializer).

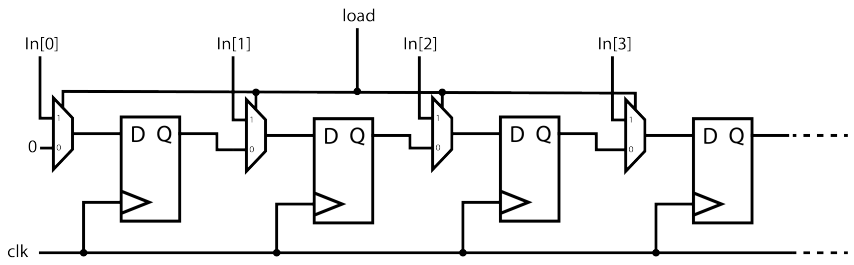
(a) Adapt this design to be parametrizable to N-bits, but also modify it to shift in 0s instead of rotating the bits.

(b) Instantiate a 4-bit version of the serializer and load with the value `4'b0110`. Simulate for 10 cycles and take a screenshot of the waveform.

Note: when using registers in your Verilog, you will need to use the register library in `EECS151.v`.

Solution:

(a) Diagram:



Design: There are two ways to approach this problem.

Using `generate` Block

```
// Don't forget to include the register library!
`include "EECS151.v"

module serializer_generate#(
  parameter N = 1
  )(
  input load,
  input clk,
  input rst,
  input [N-1:0] in,
  output out
  );

  wire [N-1:0] Q;
  wire [N-1:0] NS;
  genvar i;

  // Shift in 0 from the left
  assign NS = load ? in : {1'b0, Q[N-1:1]};
  assign out = Q[0];

  generate
    for (i=N-1; i>= 0; i=i-1) begin:stage
      REGISTER_R shift_r (.d(NS[i]), .q(Q[i]), .rst(rst), .clk(clk));
    end
  endgenerate
endmodule
```

Using Parameters

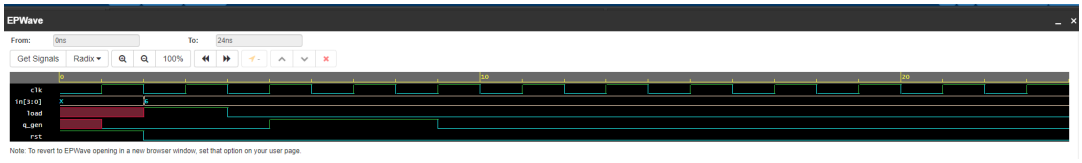
```
// Don't forget to include the register library!
`include "EECS151.v"

module serializer_param#(
  parameter N = 1
  )(
  input load,
  input clk,
  input rst,
  input [N-1:0] in,
  output out
  );

  wire [N-1:0] Q;
  wire [N-1:0] NS;

  // Shift in 0 from the left
  assign NS = load ? in : {1'b0, Q[N-1:1]};
  assign out = Q[0];

  REGISTER_R #(.N(N), .INIT(0)) shift_r (.d(NS), .q(Q), .rst(rst),
  ↪  .clk(clk));
endmodule
```

(b) Testbench: I am testing both types of serializer here. Ignore the lines specific to the other type for your solution.

```
`timescale 1ns / 1ns

module serializer_tb;
  parameter N = 4;

  reg [N-1:0] in;
  reg load, clk, rst;
  reg [9:0] out_trace_g, out_trace_p;
  wire q_gen, q_param;
  // 10-bit golden vector
  wire [9:0] expected = {6'h0, in};

  integer i = 0;

  // define clock
  initial clk = 0;
  always #(1) clk = ~clk;
```

```
    // instantiate designs
    serializer_generate #(.N(N)) dut_gen (.load(load), .clk(clk),
    ↪  .rst(rst), .in(in), .out(q_gen));
    serializer_param #(.N(N)) dut_param (.load(load), .clk(clk),
    ↪  .rst(rst), .in(in), .out(q_param));

    initial begin
      $dumpfile("dump.vcd");
      $dumpvars;
      // Reset registers
      rst = 1'b1;
      #2;
      // Begin test vector
      rst = 1'b0;
      in = 4'b0110;
      load = 1'b1;
      #2;
      load = 1'b0;
      // Run for 10 cycles
      for (i = 0; i < 10; i = i + 1) begin
        out_trace_g[i] = q_gen; out_trace_p[i] = q_param;
        if (expected[i] !== q_gen) begin
          $display("GENERATED SERIALIZER FAILED, expected %b, got %b",
          ↪  expected[i], q_gen);
          $display("gen output trace: %b", out_trace_g);
          $finish();
        end
        if (expected[i] !== q_param) begin
          $display("PARAMETRIZED SERIALIZER FAILED, expected %b, got
          ↪  %b", expected[i], q_param);
          $display("param output trace: %b", out_trace_p);
          $finish();
        end
        #2;
      end
      $display("TEST PASSED FOR ALL DESIGNS!");
      $display("gen output trace: %b", out_trace_g);
      $display("param output trace: %b", out_trace_p);
      $finish();
    end
  endmodule
```

Waveform:



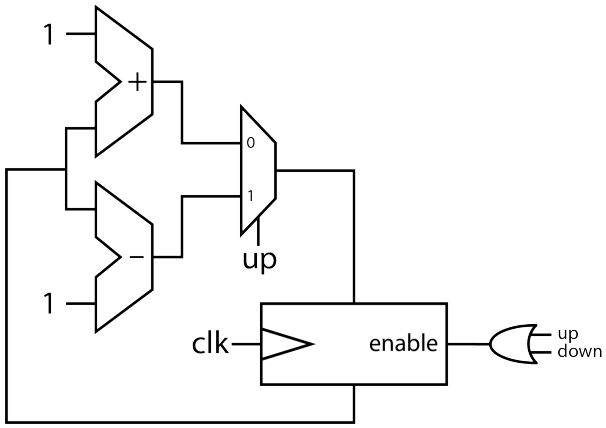
Problem 5: Wrap-Around Counter

Design an N-bit parametrizable wrap-around counter with separate `up` and `down` control inputs. When both control inputs are 1, the `up` input takes priority, that is the counter will count up in such a situation. If both signals are 0, the counter stops counting. Test your counter for N=3. The testbench should cover the following situations:

- Count up from 0 and wrap around
- Count down from maximum and wrap around
- Count up from 0 with both `up` and `down` at 1 and wrap around
- Stop count at 0 with both `up` and `down` at 0
- Count up halfway then stop the count for 2 clock cycles, then count down to 0

Solution:

Diagram:



Design:

```
// Don't forget to include the register library!
`include "EECS151.v"

module diff_ctrl_counter#(
  parameter N=1
  )(
```

```
  input clk,
  input rst,
  input up,
  input down,
  output [N-1:0] val);

  wire [N-1:0] next = up ? val + 1: val - 1;
  wire ce = up | down;

  REGISTER_R_CE #(.N(N), .INIT({N{1'b0}})) counter_reg (.d(next),
                                                        .q(val),
                                                        .rst(rst),
                                                        .ce(ce),
                                                        .clk(clk));
endmodule
```

Testbench:

```
`timescale 1ns / 1ns

module diff_counter_tb;
  parameter N = 3;

  reg up, down, clk, rst;
  wire [N-1:0] val;
  reg [N-1:0] expected;

  integer i = 0;

  // define clock
  initial clk = 0;
  always #(1) clk = ~clk;

  diff_ctrl_counter #(.N(N)) dut (.clk(clk), .rst(rst), .up(up),
  ↪  .down(down), .val(val));

  // defining reset task since this is used a lot
  task reset;
    begin
    up = 1'b0;
    down = 1'b0;
    expected = 3'b000;
    rst = 1'b1;
    #2;
    rst = 1'b0;
    end
```

```
    endtask

    // begin test
    initial begin
      $dumpfile("counter_rb.vcd");
      $dumpvars;
      // reset counter
      reset;
      // count up, pick something arbitrarily >8
      $display("TESTING COUNT UP");
      for(i=0; i<10; i=i+1) begin
        up = 1'b1;
        if (val !== expected) begin
          $display("FAILED COUNT UP NORMAL expected %b, got %b", expected,
          ↪  val);
          $finish();
        end
        expected = expected + 1;
        #2;
      end
      // reset for next test
      reset;
      // count down from maximum, can test wrap immediately
      $display("TESTING COUNT DOWN");
      for(i=0; i<10; i=i+1) begin
        down = 1'b1;
        if (val !== expected) begin
          $display("FAILED COUNT DOWN expected %b, got %b", expected, val);
          $finish();
        end
        expected = expected - 1;
        #2;
      end
      // reset for next test
      reset;
      // count up, both up and down high
      $display("TESTING COUNT UP OVERCTRL");
      for(i=0; i<10; i=i+1) begin
        up = 1'b1;
        down = 1'b1;
        if (val !== expected) begin
          $display("FAILED COUNT UP OVERCTRL expected %b, got %b", expected,
          ↪  val);
          $finish();
        end
        expected = expected + 1;
        #2;
```

```
    end
    // reset for next test
    reset;
    // stop count, arbitrarily use same number of cycles
    $display("TESTING COUNT STOP");
    for(i=0; i<10; i=i+1) begin
      if (val !== expected) begin
        $display("FAILED COUNT STOP expected %b, got %b", expected, val);
        $finish();
      end
      expected = expected;
      #2;
    end
    // reset for next test
    reset;
    // count up then down from 4
    $display("TESTING COUNT AROUND");
    // count up to 4
    for(i=0; i<4; i=i+1) begin
      up = 1'b1;
      if (val !== expected) begin
        $display("FAILED COUNT AROUND AT COUNT UP expected %b, got %b",
        ↪  expected, val);
        $finish();
      end
      expected = expected + 1;
      #2;
    end
    // pause for 2 cycles
    for(i=0; i<1; i=i+1) begin
      up = 1'b0;
      down = 1'b0;
      if (val !== expected) begin
        $display("FAILED COUNT AROUND AT COUNT STOP expected %b, got %b",
        ↪  expected, val);
        $finish();
      end
      expected = expected;
      #2;
    end
    // count down to 0
    for(i=0; i<4; i=i+1) begin
      up = 1'b0;
      down = 1'b1;
      if (val !== expected) begin
        $display("FAILED COUNT AROUND AT COUNT DOWN expected %b, got %b",
        ↪  expected, val);
```

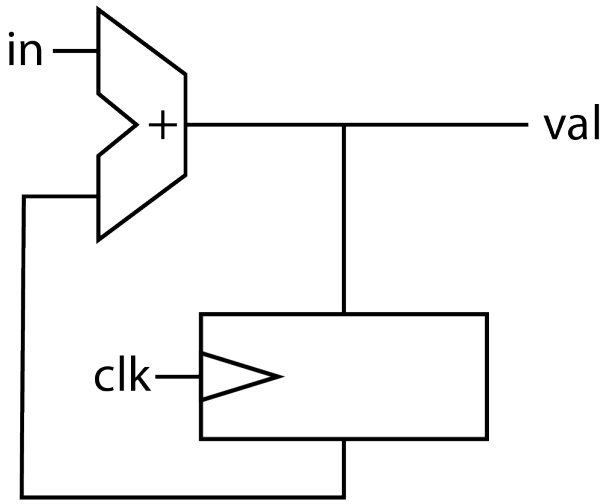
```
            $finish();
          end
          expected = expected - 1;
          #2;
        end
        $display("ALL TESTS PASSED!");
        $finish();
      end
    endmodule
```

Problem 6: Accumulator

Consider the accumulator on Lecture 4 Slide 10. Design a parametrizable version for N-bits. Test exhaustively for N=4

Solution:

Diagram:



Design:

```
// Don't forget to include the register library!
`include "EECS151.v"

module accumulator#(
  parameter N=1
  )(
  input clk,
  input rst,
  input signed [N-1:0] in,
  output signed [N-1:0] val);
```

```
  wire signed [N-1:0] last;
  assign val = last + in;


  REGISTER_R #(.N(N)) acc (.d(val), .q(last), .rst(rst), .clk(clk));
endmodule
```

Testbench:

```
`timescale 1ns / 10ps

module accumulator_tb;
  parameter N = 4;

  reg clk, rst;
  wire signed [N-1:0] val;
  reg signed [N-1:0] in, expected;

  integer i;
  integer j;

  // define clock
  initial clk = 0;
  always #(1) clk = ~clk;

  accumulator #(.N(N)) dut (.clk(clk), .rst(rst), .in(in), .val(val));

  // defining reset task since this is used a lot
  task reset;
    begin
    in = 4'b0000;
    expected = 4'b0000;
    rst = 1'b1;
    wait(~clk);
    wait(clk);
    wait(~clk);
    rst = 1'b0;
    end
  endtask

  task preload;
  input [N-1:0] preload_val;
  begin
    in = preload_val;
    #2;
```

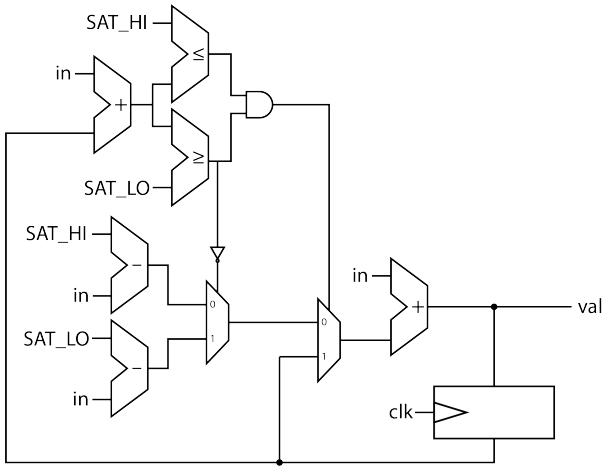
```
    end
    endtask

    initial begin
      $dumpfile("acc.vcd");
      $dumpvars;
      i = 0;
      j = 0;
      // exhaustively test adding
      // loop over all possible stored values of register
      for(i=-8; i<7; i=i+1) begin
        reset;
        preload(i);
        // loop over all possible input values for acc
        for(j=-8; j<7; j=j+1) begin
          in = j;
          expected = i + j;
          #0; // Hack to pause a bit b/c checks on same time step as change
          ↪  fail (on some sims)
          if (val != expected) begin
            $display("ACCUMULATION FAILED expected %b, got %b", expected,
            ↪  val);
            $finish();
          end
        end
      end
      $display("ALL TESTS PASSED!");
      $finish();
    end
  endmodule
```

Problem 7: 251A only — *Optional* Challenge Question for 151

Modify the accumulator in Problem 6 to have a parametrizable saturation value (both positive and negative 2's complement), such that when the accumulator reaches this value, it will no longer increment or decrement.

Solution:

Diagram:



Design:

```
// Don't forget to include the register library!
`include "EECS151.v"

module sat_accumulator#(
  parameter N=1,
  parameter signed SAT_HI={1'b0,{(N-1){1'b1}}},
  parameter signed SAT_LO={1'b1,{(N-1){1'b0}}}
  )(
  input clk,
  input rst,
  input signed [N-1:0] in,
  output signed [N-1:0] val);

  reg signed [N-1:0] diff;
  wire signed [N-1:0] last, addend;
  // To properly compare values, need N+1 bit wide result
  wire signed [N:0] int_sum = last + in;
  wire in_range = (int_sum >= $signed(SAT_LO)) && (int_sum <=
  ↪  $signed(SAT_HI));
  assign addend = (in_range) ? last : diff;
  assign val = addend + in;

  // make sure output always within saturation bounds
  always @(*) begin
    if (int_sum < SAT_LO) begin
      diff = SAT_LO - in;
    end else if (int_sum > SAT_HI) begin
      diff = SAT_HI - in;
    end
  end
```

```
  REGISTER_R #(.N(N)) acc (.d(val), .q(last), .rst(rst), .clk(clk));
endmodule
```

Testbench:

```
`timescale 1ns / 10ps

module accumulator_tb;
  parameter N = 4;
  parameter signed SAT_HI = 4'b0101;
  parameter signed SAT_LO = 4'b1010;

  reg clk, rst;
  wire signed [N-1:0] val;
  reg signed [N-1:0] in, raw_sum, expected;

  integer i;
  integer j;
  integer true_sum;

  // define clock
  initial clk = 0;
  always #(1) clk = ~clk;

  // Picking SAT_HI = 5 and SAT_LO = -6 arbitrarily
  sat_accumulator #(.N(N), .SAT_HI(SAT_HI), .SAT_LO(SAT_LO)) dut
  ↪  (.clk(clk), .rst(rst), .in(in), .val(val));

  // defining reset task since this is used a lot
  task reset;
    begin
    in = 4'b0000;
    expected = 4'b0000;
    rst = 1'b1;
    wait(~clk);
    wait(clk);
    wait(~clk);
    rst = 1'b0;
    end
  endtask

  // preload first value into accumulator
  task preload;
  input signed [N-1:0] preload_val;
  begin
```

```
    in = preload_val;
    #2;
  end
  endtask

  // emulate saturation behavior to check against
  task sat_check;
    input integer i;
    input integer j;
    begin
      expected = i;
      if (i < SAT_LO) expected = SAT_LO;
      if (i > SAT_HI) expected = SAT_HI;
      // Check true sum without bit width truncation
      true_sum = expected + j;
      if (true_sum > SAT_HI) begin
          expected = SAT_HI;
      end else if (true_sum < SAT_LO) begin
          expected = SAT_LO;
      end else begin
        expected = expected + j;
      end
    end
  endtask

  initial begin
    $dumpfile("acc.vcd");
    $dumpvars;
    i = 0;
    j = 0;
    // exhaustively test adding
    // loop over all possible stored values of register
    for(i=-8; i<7; i=i+1) begin
      reset;
      preload(i);
      // loop over all possible input values for acc
      for(j=-8; j<7; j=j+1) begin
        in = j;
        sat_check(i, j);
        #0; // Hack to pause a bit b/c checks on same time step as change
        ↪  fail (on some sims)
        // Check saturation behavior
        if (!(val >= SAT_LO && val <= SAT_HI)) begin
          if (val > SAT_HI) $display("SATURATION FAILED should clip at %b,
          ↪  got %b", SAT_HI, val);
          if (val < SAT_LO) $display("SATURATION FAILED should clip at %b,
          ↪  got %b", SAT_LO, val);
```

```
            $finish();
          end
          // Check for correct output value
          if (val !== expected) begin
            $display("ACCUMULATION FAILED expected %b, got %b", expected,
            ↪  val);
            $display("i: %4.b , j: %4.b", i, j);
            #2;
            $finish();
          end
        end
      end
      $display("ALL TESTS PASSED!");
      $finish();
    end
  endmodule
```