EECS 151/251A Homework 3

Due Monday, Feb 12th, 2024

Introduction

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. We recommend the following free, online Verilog simulator: `https://www.edaplayground.com`.

Important: Use the register library in `EECS151.v` when sequential logic is needed for all of the HW problems.

Problem 1: Parallel to Serial Converter

In lecture, we introduced the parallel to serial converter. This module is defined with the following Verilog code:

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

```
module ParToSer(ld, X, out, clk);
    input [3:0] X;
    input ld, clk;
    output out;
    wire [3:0] Q;
    wire [3:0] NS;
    assign NS =
    (ld) ? X : {Q[0], Q[3:1]};
    REGISTER state #(4)
    (.q(Q), .d(NS), .clk(clk));
    assign out = Q[0];
endmodule
```

1. Given the following waveforms for `X`,`ld`, and `clk`, please draw the corresponding waveform for `Q`, `NS`, and `out`.

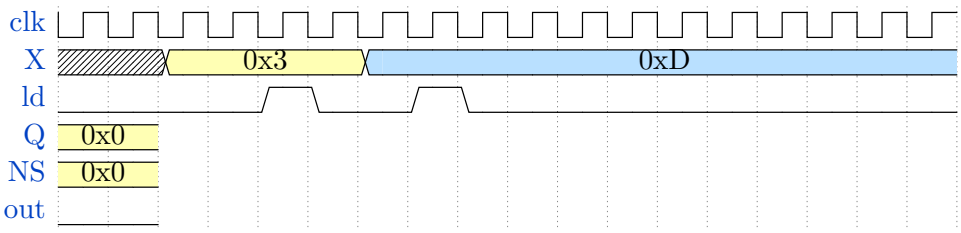
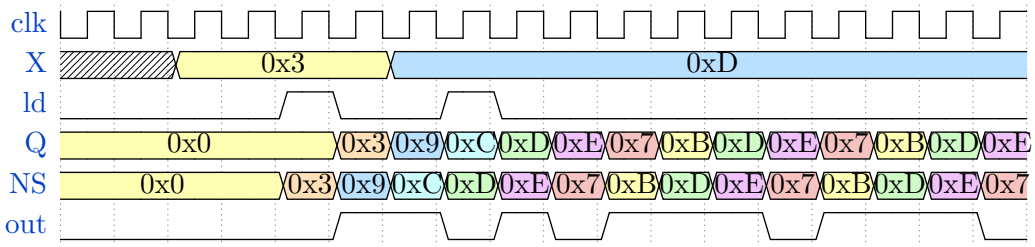


Figure 1: Serial to Parallel Waveform

2. Convert the above module to a generator that has takes as parameter `N` which represents the width of the `X` input.

Solution:

1. Note that the values only change on the positive edge of the clock signal, this is not clear from the image.



2.
```
module ParToSer_gen(ld, X, out, clk);
          parameter N = 4;
      input [N-1:0] X;
      input ld, clk;
      output out;
      wire [N-1:0] Q;
      wire [N-1:0] NS;
      assign NS =
      (ld) ? X : {Q[0], Q[N-1:1]};
          REGISTER state #(N)
      (.q(Q), .d(NS), .clk(clk));
      assign out = Q[0];
  endmodule
```

Problem 2: Up/Down Counter with Powers of 2

We have already seen many examples of counters in lecture. Here we introduce the Up/Down counter, which takes an additional bit `sign`, that determines whether the module counts up (`sign` = 0) or down (`sign` = 1). In addition we introduce a second output `pow2` that outputs 2 raised to the power of the value in the counter. The full specification is defined as follows:

Specification:

- Input: `clk` (the clock signal), `rst` (reset), `en` (enable), and `sign` (up vs. down sign).
- Outputs: 4-bit count named `cnt`, and 16-bit value for the power of two named `pow2`.
- 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 change `cnt` by 1 at a positive edge of `clk` if `rst` is 0 and `en` is 1. If `sign` is 0, the counter will increase the value, if `sign` is 1, the counter will decrement the value.

- When `cnt` is the maximum possible value ($2^4 - 1$ for 4-bit counters), `cnt` will become 0 next time it is incremented. Similarly, when `cnt` is 0, it will become $2^4 - 1$ when next decremented.

1. Based on the specification above, write a Verilog module that behaves as required, but with only the use of one register.
2. Write a second Verilog module that uses 2 registers, one for the counter and a shift register for the power of 2.
3. Write a Verilog testbench that properly drives both modules. Ensure that the clock has a period of 10ns. Make sure to test:
   (a) Every combination of `rst` (reset), `en` (enable), and `sign`.
   (b) The counter going up and down for at least 5 clock periods with no changes to the input signals.
   (c) At 0, the counter should decrement to $2^4 - 1$, and at $2^4 - 1$ the counter should increment to 0.
   (d) On a reset, the counter should go to 0.
   (e) On `en` = 0, the counter value should not change.
   (f) Use the `$monitor` command to output the value of the input and output signals in a single repeating print statement (`$monitor` prints out a statement every time a given signal value changes, so you may find it useful for debugging your counter as well).

   Be sure to include all of the Verilog code you wrote, as well as the output of the `$monitor` command.

Important Note: There are several resources online to help write good test benches, but you can start with the Verilog Primer pdf on the class website. (Note that there is a Primer pdf and Primer slides. Both are helpful but the slides do not go as far into testbenches as you may need.)

Solution:

1.
```
module pow2_count(clk, rst, en, cnt, pow2, sign);
    input clk, rst, en, sign;
    output[3:0] cnt;
    output[15:0] pow2;
    wire[3:0] next_cnt;
    reg[15:0] pow2;

    REGISTER_R_CE #(.N(4), .INIT(0))
    state(.d(next_cnt), .q(cnt), .clk(clk), .rst(rst), .ce(en));

    // Simple assign to increase count
    assign next_cnt = sign ? cnt - 1 : cnt + 1;

    // Now to calculate the power of 2 using if statements
    // Can use case statement
     always @(*)
     if (cnt == 0) pow2 = 16'x0001;
     else if (cnt == 1) pow2 = 16'x0002;
```

```
    else if (cnt == 2) pow2 = 16'x0004;
    else if (cnt == 3) pow2 = 16'x0008;
    else if (cnt == 4) pow2 = 16'x0010;
    else if (cnt == 5) pow2 = 16'x0020;
    else if (cnt == 6) pow2 = 16'x0040;
    else if (cnt == 7) pow2 = 16'x0080;
    else if (cnt == 8) pow2 = 16'x0100;
    else if (cnt == 9) pow2 = 16'x0200;
    else if (cnt == 10) pow2 = 16'x0400;
    else if (cnt == 11) pow2 = 16'x0800;
    else if (cnt == 12) pow2 = 16'x1000;
    else if (cnt == 13) pow2 = 16'x2000;
    else if (cnt == 14) pow2 = 16'x4000;
    else if (cnt == 15) pow2 = 16'x8000;
    else pow2 = 0;


    // You can also replace the entire block with the
    // following line, not as clear how it's implemented
    assign pow2 = 1 << cnt;
    // This compiles to a barrel shifter! Implemented
    // as a decoder with a bunch of tri-state buffers

endmodule
```

2. The extra register can just be used as a shift register:

```
module pow2_count(clk, rst, en, cnt, pow2, sign);
  input clk, rst, en, sign;
  output[3:0] cnt;
  output[15:0] pow2;
  reg[15:0] pow2;
  // wire [15:0] next_pow;
  reg[15:0] next_pow;
  wire[3:0] next_cnt;

  REGISTER_R_CE #(.N(4), .INIT(0))
  state(.d(next_cnt), .q(cnt), .clk(clk), .rst(rst), .ce(en));

  REGISTER_R_CE #(.N(16), .INIT(1))
  pow_state (.d(next_pow), .q(pow2), .clk(clk), .rst(rst), .ce(en));

  // Simple assign to increase count
  assign next_cnt = sign ? cnt - 1 : cnt + 1;
  // If max val, reset is 1, else reset is max_val
  always @(*)
    if (sign)
      next_pow = pow2[0] == 1 ? 1 << 15: pow2 >> 1;
        else
      next_pow = pow2[15] == 1 ? 1: pow2 << 1;

endmodule
```

3. There are many ways to create a valid testbench, here is one way:

```
// Code your testbench here
// or browse Examples

module top_tb;
  reg clk, rst, en, sign;
  wire[3:0] out;
  wire[15:0] pow2;
  integer i;


  pow2_count count_reg(.clk(clk),
  .rst(rst), .en(en), .cnt(out),
  .sign(sign), .pow2(pow2));

  always #5 clk = ~clk;

  initial begin

    clk <= 0;
    rst <= 0;
    en <= 0;
    sign <= 0;

    $monitor ("t=%3d, c=%1d, rst=%1d,
    en=%1d, state=%4d pow2=%6d",
    $time, clk, rst, en,
    out, pow2);

    #10
    for (i = 0; i <=1; i = i + 1) begin
      sign = i;
      rst <= 0;
      en <= 1;
      #10
      rst <= 1;
      en <= 0;
      #20
      rst <= 0;
      en <= 1;
      #2000
      rst <= 1;
      #15
      rst <= 0;
      #20;
        end

  $finish();
  end
```

```
endmodule
```

Problem 3: Shift Register

A very common hardware structure is a shift register. They appear in many forms in hardware designs, and are an important precursor to pipelined processors.

1. Write a shift register generator with the following specifications:
   (a) 2 Parameters: `N` the bit length of each element (width), and `LEN` length of the shift register (depth)
   (b) Input: `clk`, `rst`, `en`, `ele_in` (a N-bit input to the shift register)
   (c) Output: `ele_out` (the N-bit value which was at the head of the shift register prior to `en` begin asserted)
   (d) Shift register elements are set to `'b0` if `rst` asserted
   (e) `ele_in` is shifted into shift register if `en` is asserted and `rst` is deasserted.
2. Write a Verilog module combining your shift register module with your counter module from Problem 2. Instantiate two shift registers: (1) with `N`=4 and `LEN`=5 and (2) `N`=1 and `LEN`=6. Connect the `cnt` output directly to the `ele_in` input of the first shift register. Write simple combinational logic which outputs '1' if an input is divisible by 4. Connect the `pow2` output to the input of this logic, and the output of the combinational logic to the `ele_in` input of the second shift register.
3. Extra Challenge: Write the above shift register with only a single register!

Solution:

1.
```
module ShiftRegister(clk, rst, en, ele_in, ele_out);
  parameter N = 3;
  parameter LEN = 4;
  input [N - 1:0] ele_in;
  output [N - 1:0] ele_out;
  input clk, rst, en;
  //wire [N*LEN -1: 0] all_vals;

  wire[N-1:0] array2D [LEN:0];


  assign array2D[0] = ele_in;
  assign ele_out = array2D[LEN];

  genvar i;

  generate
    for (i = 0; i < LEN; i = i + 1) begin
      REGISTER_R_CE #(N) sr_state (.q(array2D[i + 1]),
      .d(array2D[i]), .clk(clk), .rst(rst), .ce(en));
```

```
            end
        endgenerate

    endmodule
```

2. Combination:

```
    module top_tb;
        reg clk, rst, en, sign;
        reg [3:0] ele_in;
        reg [3:0] out;
        reg [15:0] pow2;
        reg pow2_div4;
        reg pow2_div4_out;
        integer i;


        ShiftRegister #(.N(4), .LEN(5)) sr (.clk(clk), .rst(rst),
        .en(en), .ele_in(ele_in), .ele_out(out));

        ShiftRegister #(.N(1), .LEN(6)) sr_2 (.clk(clk), .rst(rst),
        .en(en), .ele_in(pow2_div4), .ele_out(pow2_div4_out));

        pow2_count count_reg(.clk(clk), .rst(rst),
        .en(en), .cnt(ele_in), .sign(sign), .pow2(pow2));


        // Logic to detect if if pow
        assign pow2_div4 = ~|pow2[1:0];

    endmodule
```

Problem 4: Combining LUTs

1. Suppose you are only given 4-LUTs (any number of them) and no other modules (this includes simple logic gates such as inverters, ANDs, ORs, etc.). Please draw a diagram showing how you would combine these LUTs to generate a 5-LUT. Remember than a 5-LUT should be able to implement any boolean function on 5 inputs. Provide some justification (truth table, formula, etc.) why your diagram is correct.
2. Extending this, explain how you would generate a (N+1) LUTs from any number of N-LUTs.

Solution:

1. You need 3 4-LUTs. You can use the 3rd 4-LUT as a multiplexer that uses the top bit of the 5 input bits.
2. This holds for any N to N + 1 LUT combo. You always require 3 LUTs, where the last LUT is configured to a multiplexer on 1-bit.

Problem 5: LUT Mapping

Given the following circuit (Figure 2) with 5 inputs and 2 outputs:

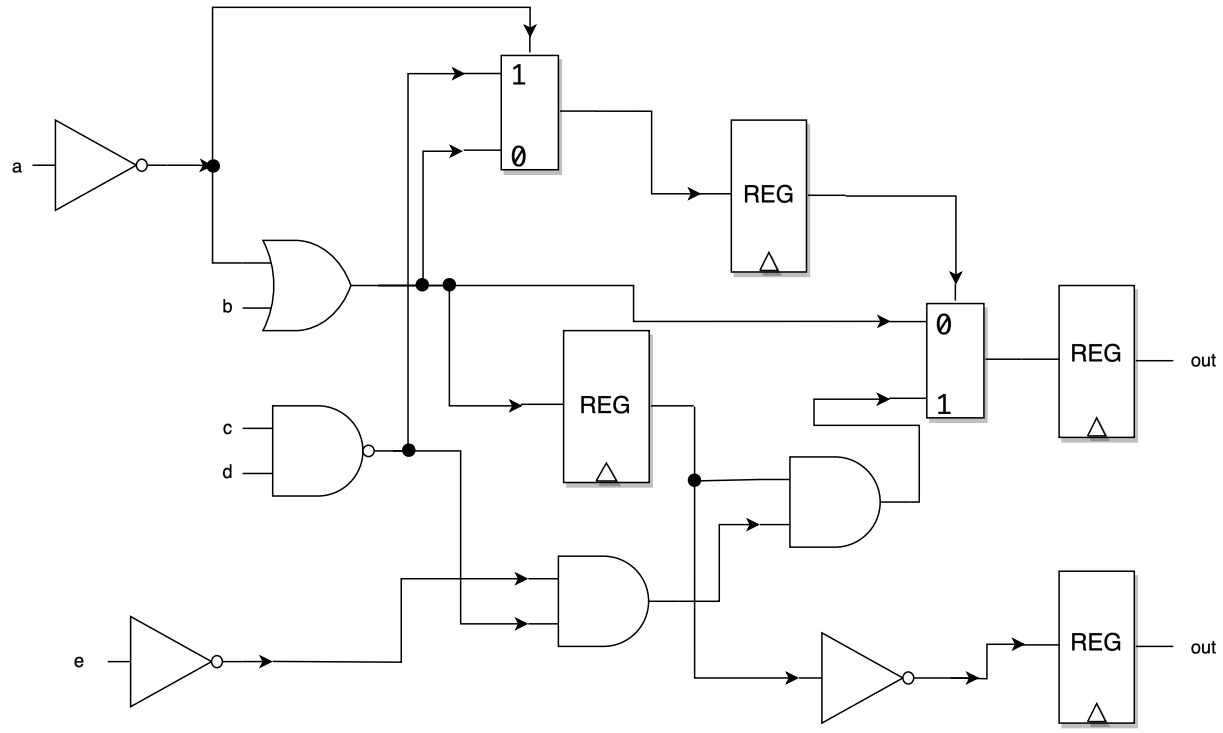
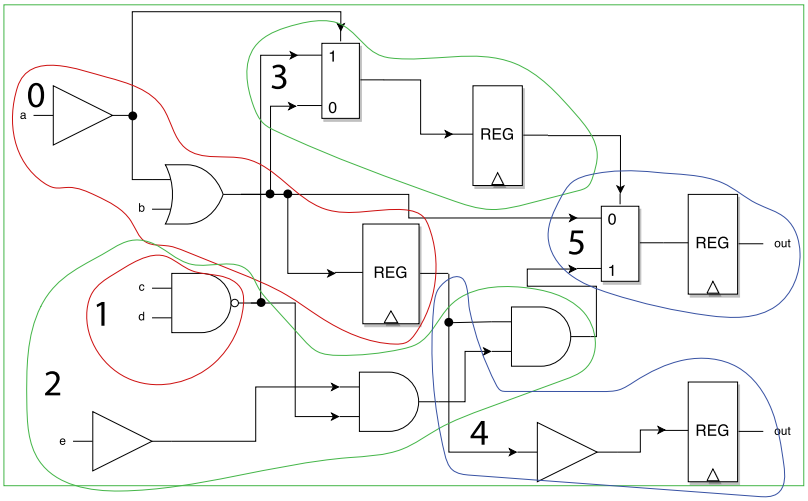


Figure 2: Problem 5 circuit

1. Please determine how many 3-LUTs are needed to fully represent this circuit (this means that the state in every register is equivalent between your LUT-based circuit and the original circuit).
2. For each LUT, please give the truth table that represents the function the LUT encodes.

Solution:

1. The strategy here is to find spots where you can find 3 or less inputs (can always pass in a 1 or 0) that maps to a single bit. Note that not all LUTs will be attached to a register. You can map the function with 6 3-LUTs, assuming that each LUT is connected to one register. If you assume that each LUT is connected to a shift register of depth 2 (which is available in some architectures) then you can map this with 5 3-LUTs (can squish together LUT4 and LUT 0). The mapping is drawn below:



2. For the LUTs in the above diagram, the truth tables are:

(a)

a	b	LUT0_out
0	0	1
0	1	1
1	0	0
1	1	1

(b)

c	d	LUT1_out
0	0	1
0	1	1
1	0	1
1	1	0

(c)

e	LUT1_out	LUT0_out	LUT2_out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(d)

a	LUT0_out	LUT1_out	LUT3_out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(e)

LUT0_out	LUT4_out
0	1
1	0

(f)

LUT0_out	LUT2_out	LUT3_out	LUT5_out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1