CSC258H1: Pre-lab Exercise for Lab 4

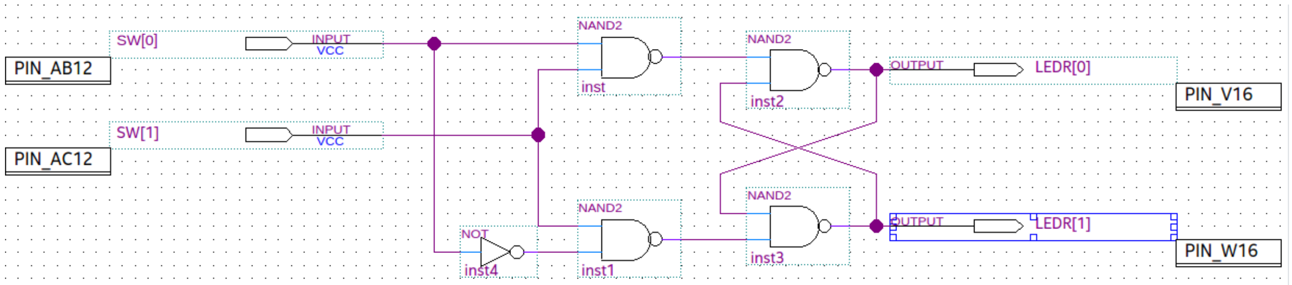
Wang, Weiqing

Due: February 5th, 2018 before 12 a.m.

1 Part I

1. In your pre-lab, draw a schematic of the gated D latch using interconnected 7400-series chips. For this exercise you are allowed to use NAND gates. Recall from Lab 1 what a gate-level schematic looks like

Solution.
Here is the schematic for my gated D latch:



2. Are thare any input combinations of Clk and D that should NOT be the first you test? Explain this in your pre-lab and list them if applicable.

Solution.
We should not test any cases for Clk = 0 as my first test, because the previous value of D is uncertain.

2 Part II

1. Create a Verilog module for the simple ALU with register, with the following specifications:

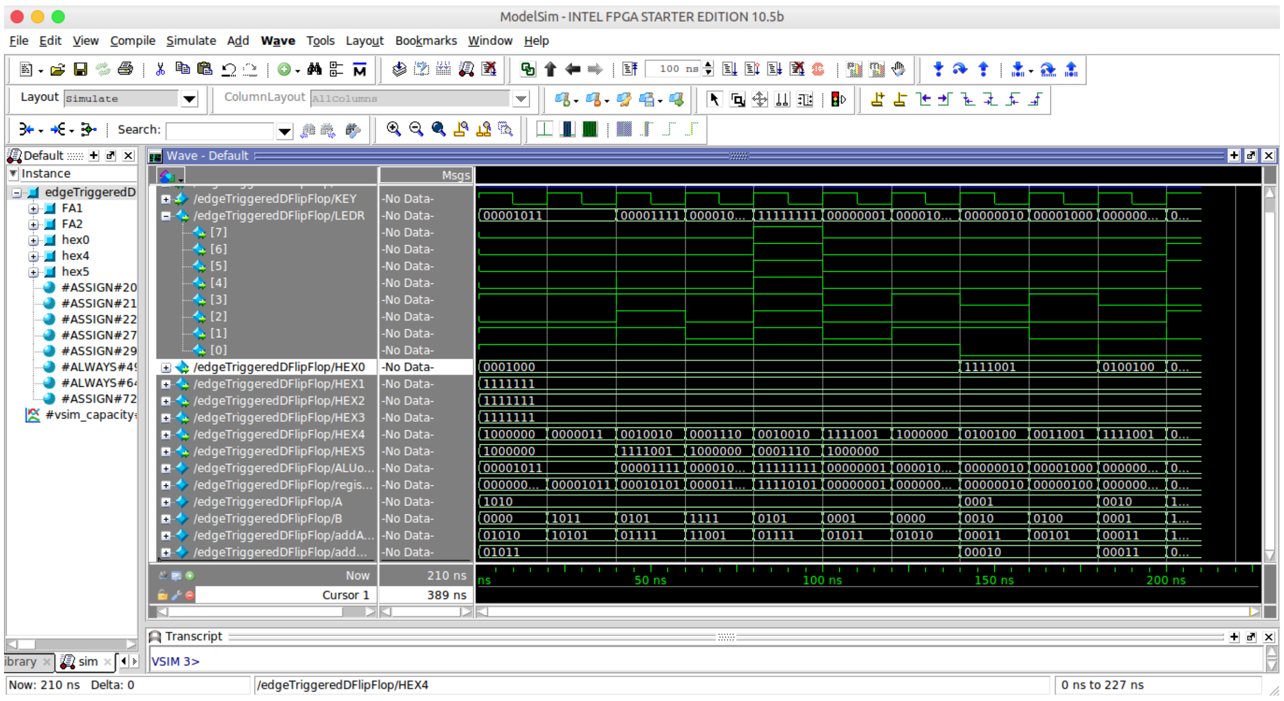
- Use the code in Figure 2 as the model for your register code.
- Connect $Data(A)$ input of your ALI to switches SW_{3-0}.
- Connect KEY_0 to the clock input for register, SW_9 to $reset\_n$ and use SW_{7-5} for the ALU function inputs.
- Display ALU outputs on $LEDR_{7-0}$; have $HEX0$ display the value of $Data(A)$ in hexadecimal and set $HEX1$, $HEX2$, $HEX3$ to display nothing (all segments off).
- $HEX4$ and $HEX5$ should display the least-significant and most-significant four bits of $Register$ (ALU outputs) respectively, also in your pre-lab.

Solution.
Here is the Verilog module for ALU with register:

```
module edgeTriggeredDFlipFlop(SW,
                              KEY,
                              LEDR,
                              HEX0,
                              HEX1,
                              HEX2,
                              HEX3,
                              HEX4,
                              HEX5);

    input [9:0] SW;
    input [0:0] KEY;
    output [7:0] LEDR;
    output [6:0] HEX0;  //HEX0 value of Date(A)
    output [6:0] HEX1;  //permantly off
    output [6:0] HEX2;  //permanantly off
    output [6:0] HEX3;  //permantly off
    output [6:0] HEX4;  //4 least significant bits of register
    output [6:0] HEX5;  //4 most significant bits of register
    assign HEX1[6:0] = 7'b1111111;
    assign HEX2[6:0] = 7'b1111111;
    assign HEX3[6:0] = 7'b1111111;

    reg [7:0] ALUout;
    reg [7:0] register;
    wire [3:0] A;
        assign A[3:0] = SW[3:0];
    wire [3:0] B;
        assign B[3:0] = register[3:0];

    wire [4:0] addAToB;
    fourBitAdder FA1(
        .SW({2'b00, A[3:0], B[3:0]}),
        .LEDR(addAToB[4:0])
    );

    wire [4:0] addOneToA;
    fourBitAdder FA2(
        .SW({2'b00, A[3:0], 4'b0001}),
        .LEDR(addOneToA[4:0])
    );

    hexDecoder hex0(
        .SW(A[3:0]),
        .HEX(HEX0[6:0])
    );

    always @(*)
    begin
        case (SW[7:5])
            3'b000: ALUout[7:0] = {3'b000, addOneToA[4:0]};
            3'b001: ALUout[7:0] = {3'b000, addAToB[4:0]};
            3'b010: ALUout[7:0] = {3'b000, A[3:0] + B[3:0]};
            3'b011: ALUout[7:0] = {A[3:0] | B[3:0], A[3:0] ^ B[3:0]};
            3'b100: ALUout[7:0] = {7'b0000000, (|{A[3:0], B[3:0]})};
            3'b101: ALUout[7:0] = B[3:0] << A[3:0];
            3'b110: ALUout[7:0] = B[3:0] >> A[3:0];
            3'b111: ALUout = A[3:0] * B[3:0];
            default: ALUout[7:0] = 8'b0000_0000;
        endcase
    end

    always @(posedge KEY[0])
    begin
        if (SW[9] == 1'b0)
            register[7:0] <= 8'b0000_0000;
        else
            register[7:0] <= ALUout[7:0];
    end

    assign LEDR[7:0] = ALUout[7:0];

    hexDecoder hex4(
        .SW(register[3:0]),
        .HEX(HEX4[6:0])
    );

    hexDecoder hex5(
        .SW(register[7:4]),
        .HEX(HEX5[6:0])
    );
endmodule

module hexDecoder (SW, HEX);
    input [3:0] SW;
    output [6:0] HEX;

    assign HEX[0] = ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
            SW[3] & ~SW[2] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[1] = SW[2] & SW[1] & ~SW[0] |
            SW[3] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[2] = SW[3] & SW[2] & ~SW[0] |
            SW[3] & SW[2] & SW[1] |
            ~SW[3] & ~SW[2] & SW[1] & ~SW[0];

    assign HEX[3] = SW[2] & SW[1] & SW[0] |
            ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
            SW[3] & ~SW[2] & SW[1] & ~SW[0];

    assign HEX[4] = ~SW[3] & SW[0] |
            ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] |
            ~SW[3] & SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & SW[1] & SW[0] |
            SW[3] & ~SW[2] & ~SW[1] & SW[0];

    assign HEX[5] = ~SW[3] & ~SW[2] & SW[0] |
            ~SW[3] & ~SW[2] & SW[1] |
            ~SW[3] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[6] = ~SW[3] & ~SW[2] & ~SW[1] |
            ~SW[3] & SW[2] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & ~SW[0];
endmodule

module fourBitAdder(SW, LEDR);
    input [9:0] SW;
    output [4:0] LEDR;
    wire [2:0] cable;

    fullAdder FA1(
        .a(SW[4]),
        .b(SW[0]),
        .cin(SW[9]),
        .s(LEDR[0]),
        .cout(cable[0])
    );

    fullAdder FA2(
        .a(SW[5]),
        .b(SW[1]),
        .cin(cable[0]),
        .s(LEDR[1]),
        .cout(cable[1])
    );

    fullAdder FA3(
        .a(SW[6]),
        .b(SW[2]),
        .cin(cable[1]),
        .s(LEDR[2]),
        .cout(cable[2])
    );

    fullAdder FA4(
            .a(SW[7]),
            .b(SW[3]),
            .cin(cable[2]),
            .s(LEDR[3]),
            .cout(LEDR[4])
      );
endmodule

module fullAdder(a, b, cin, s, cout);
      input a;
      input b;
      input cin;
      output s;
      output cout;

      assign s = a^b^cin;
      assign cout = (a & b) | (cin & (a^b));
endmodule
```

2. Simulate your circuit with ModelSim, ensuring the output waveforms are correct for each of your test cases. Choose test cases that make you feel confident about your ALU's correctness, in preparation for you in-lab demo. Your pre-lab should include at least one test for each ALU operation, though more test cases per operation is advisable. Make sure to include a few selected screenshots of these cases when you hand in your pre-lab.

Solution.
Here is the screenshot of my ModelSim simulation:



```
/edgeTriggeredDFlipFlop/KEY      -No Data-
/edgeTriggeredDFlipFlop/LEDR     -No Data-  00001011 | 00001111 | 000010... | 11111111 | 00000001 | 000010... | 00000010 | 00001000 | 000000... | 0...
  [7]                            -No Data-
  [6]                            -No Data-
  [5]                            -No Data-
  [4]                            -No Data-
  [3]                            -No Data-
  [2]                            -No Data-
  [1]                            -No Data-
  [0]                            -No Data-
/edgeTriggeredDFlipFlop/HEX0     -No Data-  0001000 | 1111001 | 0100100 | 0...
/edgeTriggeredDFlipFlop/HEX1     -No Data-  1111111
/edgeTriggeredDFlipFlop/HEX2     -No Data-  1111111
/edgeTriggeredDFlipFlop/HEX3     -No Data-  1111111
/edgeTriggeredDFlipFlop/HEX4     -No Data-  1000000 | 0000011 | 0010010 | 0001110 | 0010010 | 1111001 | 1000000 | 0100100 | 0011001 | 1111001 | 0...
/edgeTriggeredDFlipFlop/HEX5     -No Data-  1000000 | 1111001 | 1000000 | 0001110 | 1000000
/edgeTriggeredDFlipFlop/ALUo...  -No Data-  00001011 | 00001111 | 000010... | 11111111 | 00000001 | 000010... | 00000010 | 00001000 | 000000... | 0...
/edgeTriggeredDFlipFlop/regis... -No Data-  000000... | 00001011 | 00010101 | 000011... | 11110101 | 00000001 | 000000... | 00000010 | 00000100 | 000000... | 0...
/edgeTriggeredDFlipFlop/A        -No Data-  1010 | 0001 | 0010 | 1...
/edgeTriggeredDFlipFlop/B        -No Data-  0000 | 1011 | 0101 | 1111 | 0101 | 0001 | 0000 | 0010 | 0100 | 0001 | 1...
/edgeTriggeredDFlipFlop/addA...  -No Data-  01010 | 10101 | 01111 | 11001 | 01111 | 01011 | 01010 | 00011 | 00101 | 00011 | 1...
/edgeTriggeredDFlipFlop/add...   -No Data-  01011 | 00010 | 00011 | 0...

Now       210 ns    ns   50 ns   100 ns   150 ns   200 ns
Cursor 1  389 ns

VSIM 3>

Now: 210 ns   Delta: 0      /edgeTriggeredDFlipFlop/HEX4      0 ns to 227 ns
```

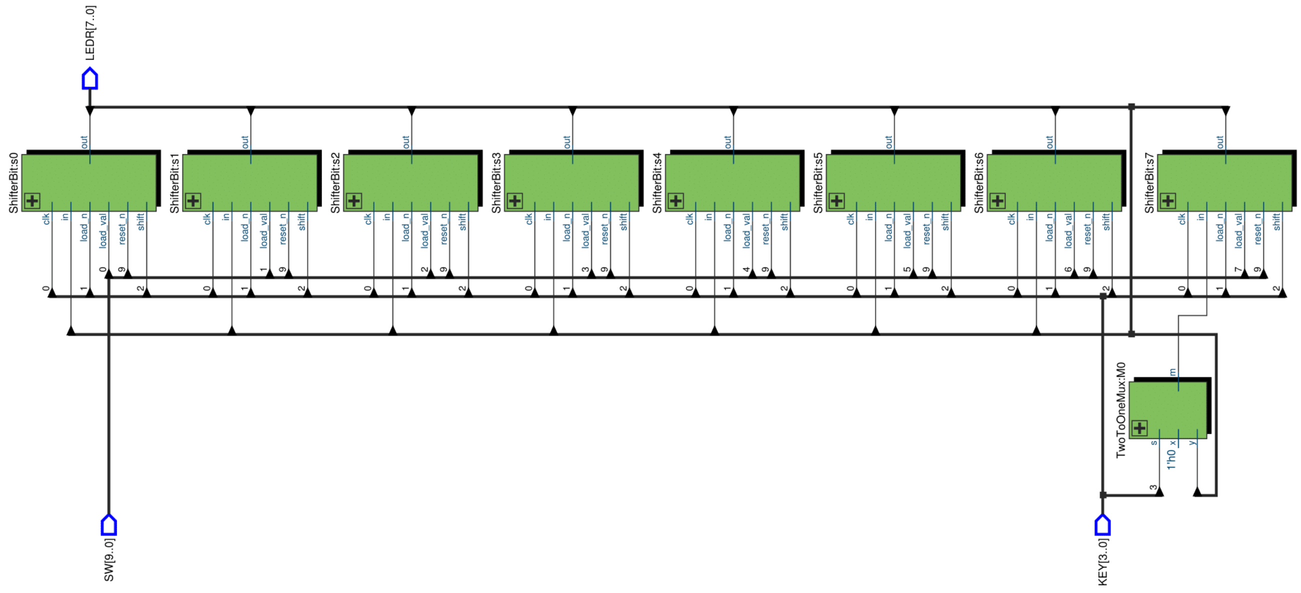
3 Part III

1. What is the behaviour of the 8-bit shift register shown in Figure 6 when $Load\_n = 1$ and $ShiftRight = 0$? Briefly explain in your pre-lab.

Solution.
The value of the 8-bit shift stays constant throughout the whole process, because both $Load\_n$ and $ShiftRight$ are connected to the Multiplexer, which only decides the input of input D, not the register.

2. Draw a schematic for the 8-bit shift register shown in Figure 6 including the necessary connections. Your schematic should contain eight instances of the one-bit shifter shown in Figure 5 and all the wiring required to implement the desired behaviour. Label the signals on your schematic with the same names you will use in your Verilog code.

Solution.



3. Starting with the code in Figure 2 for a positive edge-triggered D flip-flop, use this D flip-flop with instances of the mux2to1 module from Lab 2 to build the one-bit shifter shown in Figure 5. To get you started, Figure 7 is a code snippet that shows how the D flip-flop will be connected to one of the 2-to-1 multiplexers.

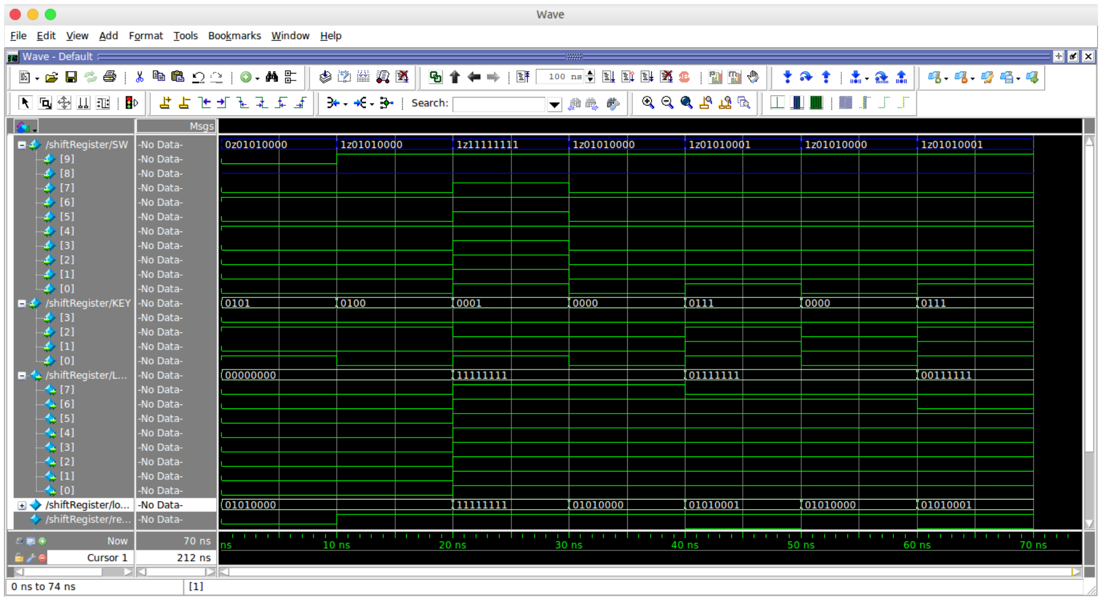
4. Write a Verilog module for the shift register that instantiates eight instances of one-bit shift register that you created in previous step. This Verilog module should match with the schematic in your pre-lab. Use SW_{7-0} as the inputs $LoadVal_{7-0}$ and SW_9 as a synchronous active low reset ($reset\_n$). Use KEY_1 as the $Load\_n$ input, KEY_2 as the $ShiftRight$ input and KEY_3 as the ASR input. Use KEY_0 as the clock (clk). The outputs Q_{7-0} should be displayed on $LEDR_{7-0}$.

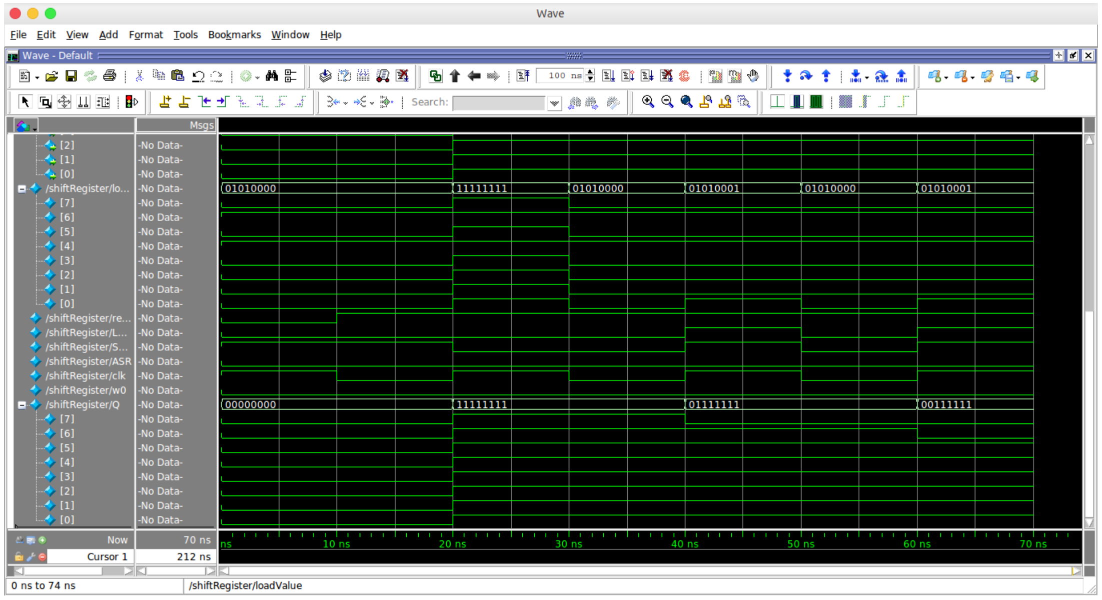
Here is the solution to Questions 3 and 4:

```
module shiftRegister(SW, KEY, LEDR);
    input [9:0] SW;
    input [3:0] KEY;
    output [7:0] LEDR;

    wire [7:0] loadValue;
        assign loadValue[7:0] = SW[7:0];
    wire reset_n;
        assign reset_n = SW[9];
    wire Load_n;
        assign Load_n = KEY[1];
    wire ShiftRight;
        assign ShiftRight = KEY[2];
    wire ASR;
        assign ASR = KEY[3];
    wire clk;
        assign clk = KEY[0];

    wire w0;
    wire [7:0] Q;

    TwoToOneMux M0(.x(1'b0),
                   .y(Q[7]),
                   .s(ASR),
                   .m(w0)
    );

    ShifterBit s7(
        .load_val(loadValue[7]),
        .in(w0), .out(Q[7]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s6(
        .load_val(loadValue[6]),
        .in(Q[7]), .out(Q[6]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s5(
        .load_val(loadValue[5]),
        .in(Q[6]),
        .out(Q[5]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s4(
        .load_val(loadValue[4]),
        .in(Q[5]),
        .out(Q[4]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s3(
        .load_val(loadValue[3]),
        .in(Q[4]),
        .out(Q[3]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s2(
        .load_val(loadValue[2]),
        .in(Q[3]),
        .out(Q[2]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s1(
        .load_val(loadValue[1]),
        .in(Q[2]),
        .out(Q[1]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    ShifterBit s0(
        .load_val(loadValue[0]),
        .in(Q[1]),
        .out(Q[0]),
        .reset_n(reset_n),
        .clk(clk),
        .load_n(Load_n),
        .shift(ShiftRight)
    );

    assign LEDR[7:0] = Q[7:0];
endmodule

module ShifterBit(load_val, in, out, reset_n, clk, load_n, shift);
    input in, load_val, reset_n, clk, load_n, shift;
    output out;
    wire w1, w2;

    TwoToOneMux M0(
        .x(out),
        .y(in),
        .s(shift),
        .m(w1)
    );

    TwoToOneMux M1(
        .x(load_val),
        .y(w1),
        .s(load_n),
        .m(w2)
    );

    FlipFlop F0(
        .d(w2),
        .q(out),
        .clock(clk),
        .reset_n(reset_n)
    );
endmodule


module FlipFlop(d, q, clock, reset_n);
    input clock, reset_n;
    input d;
    output reg q;

    always @(posedge clock)
    begin
        if (reset_n == 1'b0)
            q <= 8'b0000_0000;
        else
            q <= d;
    end
endmodule

module TwoToOneMux(x, y, s, m);
    input x;    //selected when s is 0
    input y;     //selected when s is 1
    input s;     //select signal
    output m;    //output

    assign m = s ? y : x;
endmodule
```

5. Compile your Verilog code and simulate the design with ModelSim. In your simulation, you should perform the reset operation on the first clock cycle, then do a parallel load of your register on the next cycle. Finally, clock the register for several cycles to demonstrate both types of shifts. (NOTE: If you do not perform a reset first, your simulation will not work! Try simulating without doing reset first and see what happens. Can you explain the results?) Include one (or a few) screenshot of simulation output in your pre-lab.

Solution.


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/shiftRegister/SW
0z01010000 1z01010000 1z11111111 1z01010000 1z01010001 1z01010000 1z01010001
/shiftRegister/KEY
0101 0100 0001 0000 0111 0000 0111
/shiftRegister/L...
00000000 11111111 01111111 00111111
/shiftRegister/lo...
01010000 11111111 01010000 01010001 01010000 01010001
/shiftRegister/re...
Now 70 ns
Cursor 1 212 ns
0 ns to 74 ns


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/shiftRegister/lo...
01010000 11111111 01010000 01010001 01010000 01010001
/shiftRegister/re...
/shiftRegister/L...
/shiftRegister/S...
/shiftRegister/ASR
/shiftRegister/clk
/shiftRegister/w0
/shiftRegister/Q
00000000 11111111 01111111 00111111
Now 70 ns
Cursor 1 212 ns
0 ns to 74 ns
/shiftRegister/loadValue