CSC258H1: Pre-lab Exercise for Lab 5

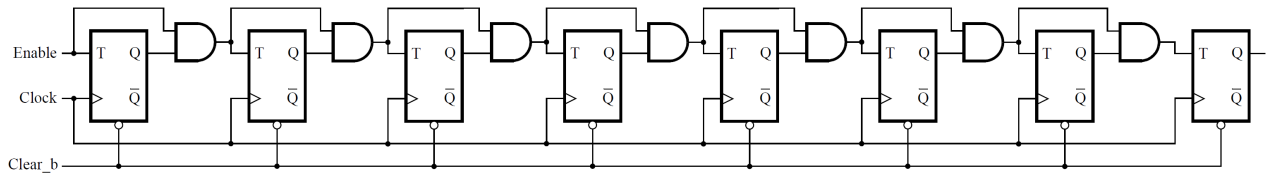
Wang, Weiqing

Due: February 11th, 2018 before 12 a.m.

1 Part I

1. Draw the schematic for an 8-bit counter using the same structure as shown in Figure 1.

Solution.



2. Annotate all Q outputs of your schematic with the bit of the counter ($Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$) that they correspond to.

Solution.
The output Q of the T-type Flip-Flops from right to left are respectively Q_{7-0}.

3. Write the Verilog corresponding to your schematic. Your code should use a module for the flip-flop that is instantiated 8 times to create the counter. Note that you should not name your module `TFF` as this is a Quartus primitive and thus Quartus will ignore that entity after issuing a warning. (*e.g.* use a name such as `MyTFF`).

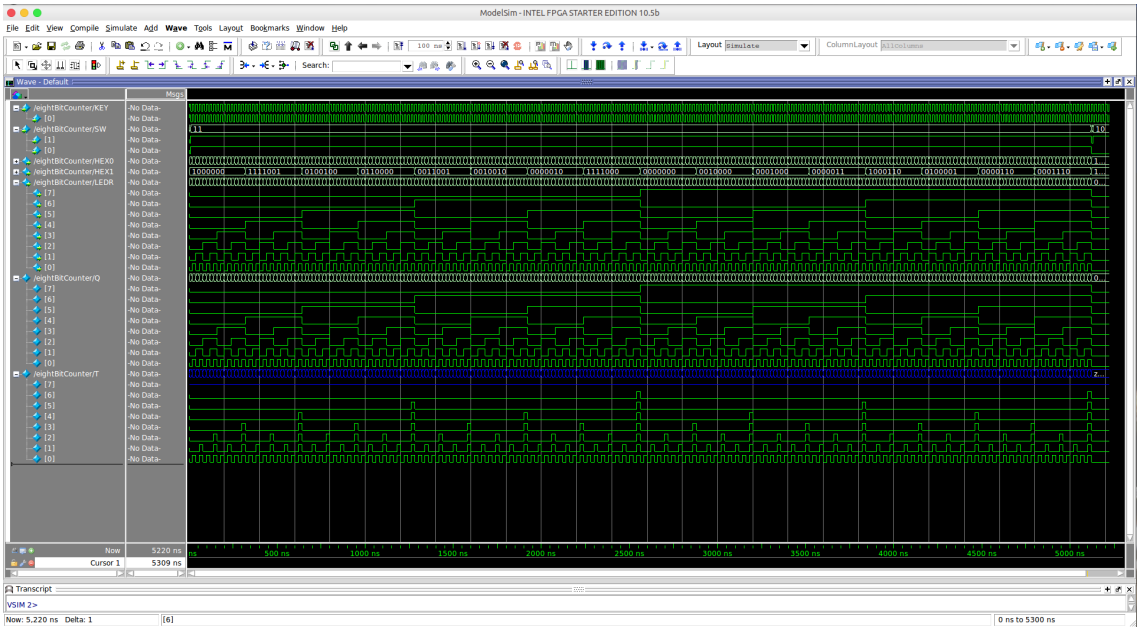
Solution.
Here is the Verilog for my 8-bit counter.

```
module eightBitCounter(KEY, SW, HEX0, HEX1, LEDR);
    input [0:0] KEY;    //Clock.
    input [1:0] SW;     //SW[0] for Clear_b, SW[1] for Enable.
    output [6:0] HEX0;
    output [6:0] HEX1;
    output [7:0] LEDR;
    wire [6:0] T;

    assign T[0] = SW[1] & LEDR[0];
    assign T[1] = T[0] & LEDR[1];
    assign T[2] = T[1] & LEDR[2];
    assign T[3] = T[2] & LEDR[3];
    assign T[4] = T[3] & LEDR[4];
    assign T[5] = T[4] & LEDR[5];
    assign T[6] = T[5] & LEDR[6];

    tTypeFlipFlop TFF0(.Clk(KEY[0]), .T(SW[0]),.Q(LEDR[0]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF1(.Clk(KEY[0]), .T(T[0]), .Q(LEDR[1]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF2(.Clk(KEY[0]), .T(T[1]), .Q(LEDR[2]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF3(.Clk(KEY[0]), .T(T[2]), .Q(LEDR[3]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF4(.Clk(KEY[0]), .T(T[3]), .Q(LEDR[4]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF5(.Clk(KEY[0]), .T(T[4]), .Q(LEDR[5]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF6(.Clk(KEY[0]), .T(T[5]), .Q(LEDR[6]), .Clear_n(SW[0]));
    tTypeFlipFlop TFF7(.Clk(KEY[0]), .T(T[6]), .Q(LEDR[7]), .Clear_n(SW[0]));

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

```
    hexDecoder hex1(.SW(LEDR[7:4]), .HEX(HEX1[6:0]));
endmodule

module tTypeFlipFlop(Clk, T, Clear_n, Q);
    input Clk;
    input T;
    input Clear_n;
    output reg Q;

    always @(posedge Clk, negedge Clear_n)
    begin
        if (Clear_n == 1'b0)
            Q <= 1'b0;
        else if  (T == 1'b1)
            Q <= !Q;
    end
endmodule

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

    always @(*)
    begin
        case (SW[3:0])
            4'b0000: HEX[6:0] = 7'b1000000;
            4'b0001: HEX[6:0] = 7'b1111001;
            4'b0010: HEX[6:0] = 7'b0100100;
            4'b0011: HEX[6:0] = 7'b0110000;
            4'b0100: HEX[6:0] = 7'b0011001;
            4'b0101: HEX[6:0] = 7'b0010010;
            4'b0110: HEX[6:0] = 7'b0000010;
            4'b0111: HEX[6:0] = 7'b1111000;
            4'b1000: HEX[6:0] = 7'b0000000;
            4'b1001: HEX[6:0] = 7'b0010000;
            4'b1010: HEX[6:0] = 7'b0001000;
            4'b1011: HEX[6:0] = 7'b0000011;
            4'b1100: HEX[6:0] = 7'b1000110;
            4'b1101: HEX[6:0] = 7'b0100001;
            4'b1110: HEX[6:0] = 7'b0000110;
            4'b1111: HEX[6:0] = 7'b0001110;
            default: HEX[6:0] = 7'b1000000;
        endcase
    end
endmodule
```

4. Simulate your circuit in ModelSim to verify its correctness. You will need to reset (clear) all your flip-flops early in your simulation, so your circuit is in a known state. Include screenshots of simulation output in your prelab.

Solution.
Here is the screenshot for my ModelSim simulation:



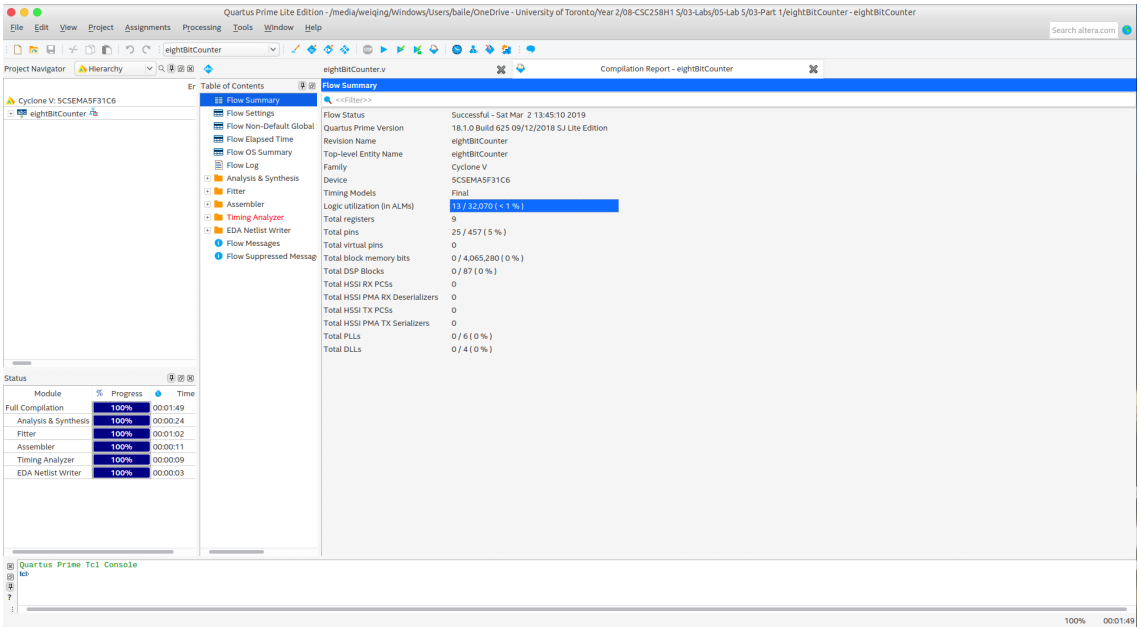
5. Augment your Verilog code to use the push button KEY 0 as the Clock input, switches SW_1 and SW_0 as *Enable* and *Clear_b* inputs, and 7-segment displays $HEX0$ and $HEX1$ to display the hexadecimal value of your counter as your circuit operates. Simulate your circuit to ensure that you have done this correctly (include at least one screenshot).

Solution.
See my solution to question 4. The simulation in Question 4 is enough to ensure I have done correctly.

6. Create a new Quartus Prime project for your circuit. Make sure it is stored in your W: drive. Do not forget to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments. Compile the circuit in Quartus and answer the following questions:

   (a) What percentage of FPGA logic resources are used to implement your circuit?

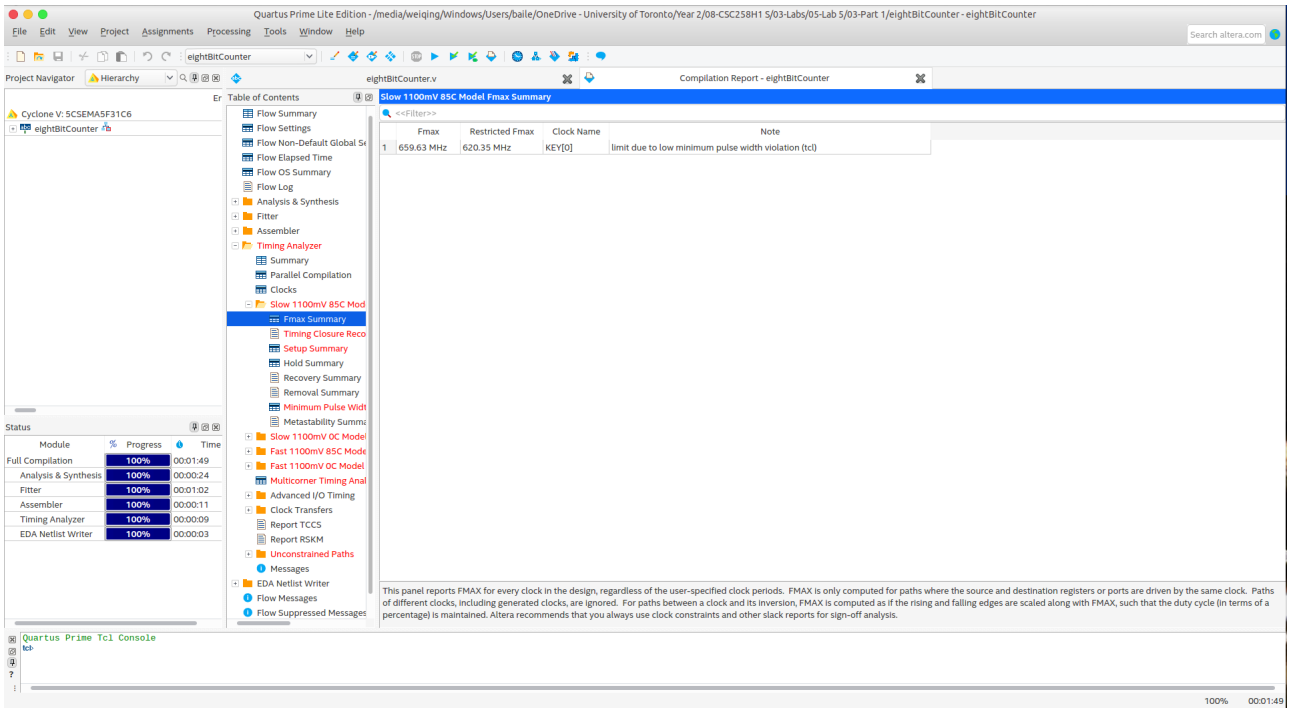
Solution.
See the screenshot below:



   (b) What is the maximum clock frequency, F_{max}, at which your circuit can be operated?

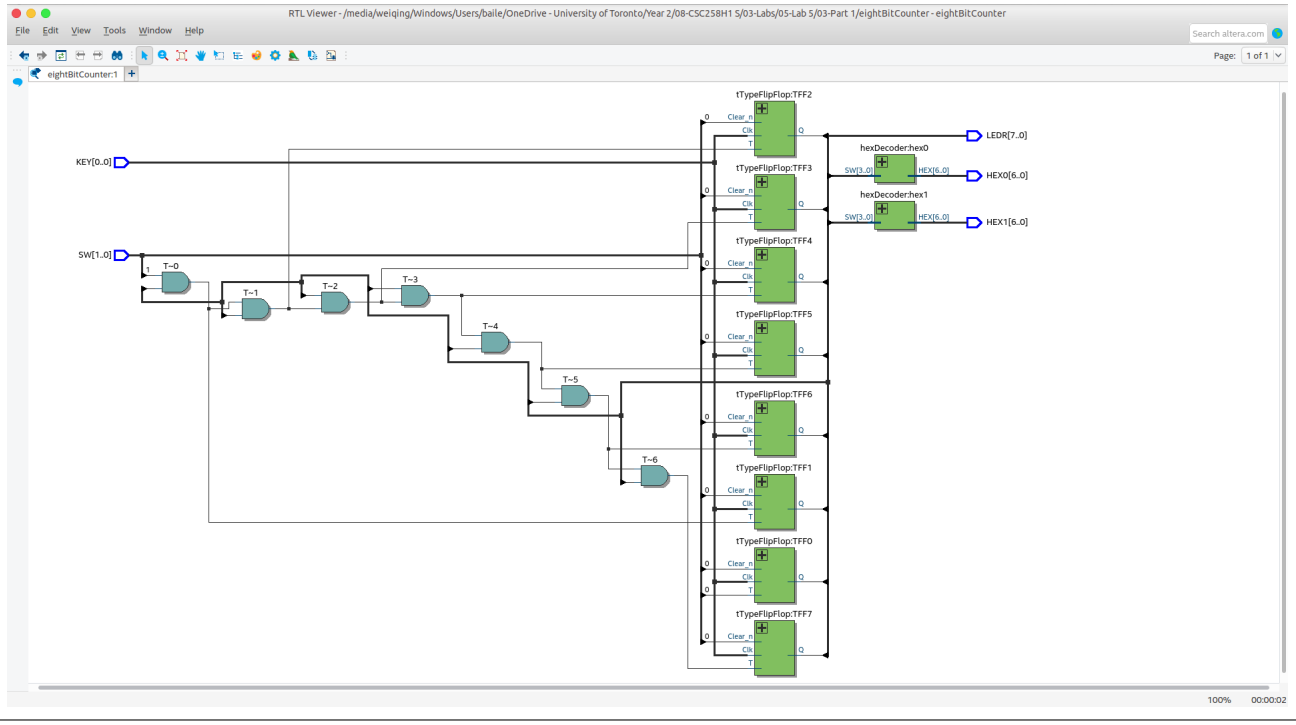
Solution.

See the screenshot below:



7. Use the Quartus Prime RTL Viewer to see how the Quartus Prime software synthesized your circuit. You can access the RTL viewer on Quartus via Tools -¿ Netlist Viewers -¿ RTL Viewer. You can zoom into the various building blocks of your circuit by double-clicking on them, to get more information about their implementation. What are the differences in comparison with Figure 1?

Solution.



2 Part II

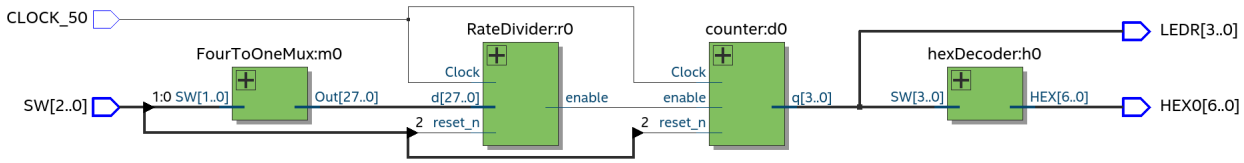
- I no longer need to check the maximum values, due to the limit in the number of digits.
- The following lines of code can be added to Verilog code below:

```
if (q == 4'b1111)
      q <= 1'b0
```

- I expect to see 8. Eagles may see something different.
- 50M requires 26-bit to represent in binary.

1. Draw a schematic of the circuit you wish to build. Work through the circuit manually to ensure that it will work according to your understanding.

Solution.



2. Write a Verilog module that realizes the behaviour described in your schematic. Your circuit should have a clock input and two switches inputs.

Solution.

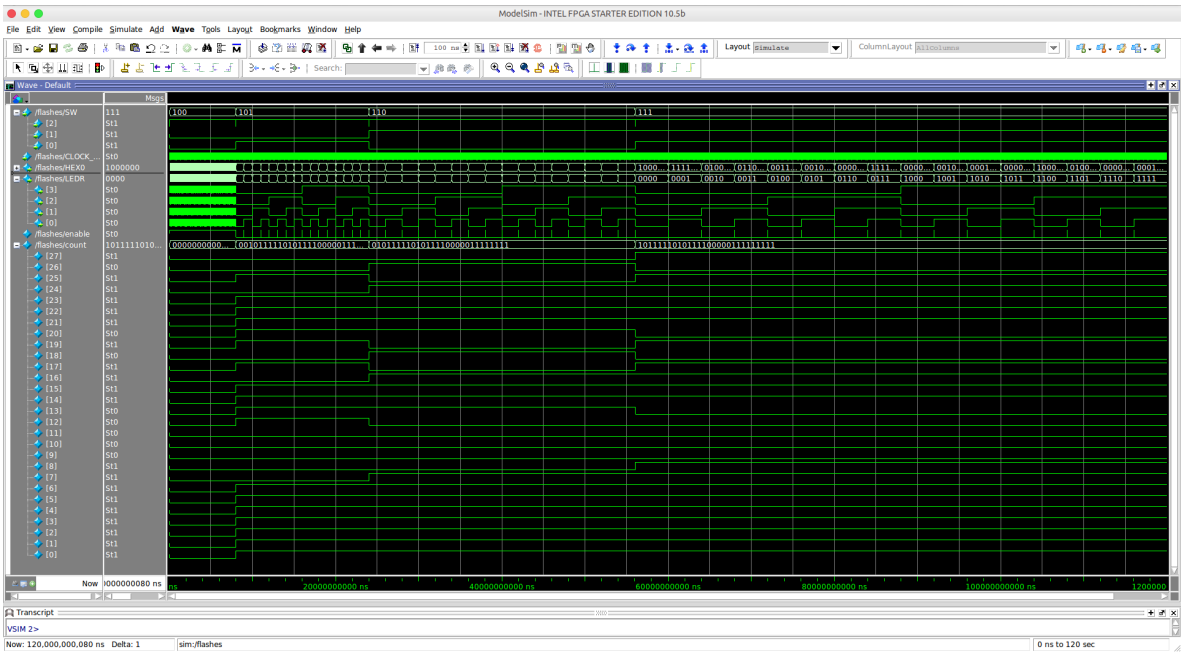
```
module flashes(SW, CLOCK_50, HEX0, LEDR);
    input [2:0] SW;        //SW[1:0] for rate selection, SW[2] for reset_n.
    input CLOCK_50;
    output [6:0] HEX0;     //Display the result in base 10.
    output [3:0] LEDR;     //Displaying the binary result.
    wire enable;           //Whether to enable the counter.
    reg [27:0] count;     //28 comes from log_2 <199_999_999> = 27.57.

    always @(*)
    begin
        case (SW[1:0])
            2'b00: count[27:0] = 28'd0;
            2'b01: count[27:0] = 28'd49_999_999;
            2'b10: count[27:0] = 28'd99_999_999;
            2'b11: count[27:0] = 28'd199_999_999;
            default: count[27:0] = 28'd0;
        endcase
    end

    RateDivider r0(
        .Clock(CLOCK_50),
        .reset_n(SW[2]),
        .enable(enable),
        .d(count[27:0])
    );
    counter d0(
        .Clock(CLOCK_50),
        .enable(enable),
        .reset_n(SW[2]),
        .q(LEDR[3:0])
```

```
    );
    hexDecoder h0(
        .SW(LEDR[3:0]),
        .HEX(HEX0[6:0])
    );
endmodule

module counter(Clock, enable, reset_n, q);
    input Clock;
    input enable;
    input reset_n;
    output [3:0] q;
    reg [3:0] count;

    always @(posedge Clock, negedge reset_n)
    begin
        if (reset_n == 1'b0)
            count[3:0] <= 4'b0000;
        else if (enable == 1'b1)
            begin
                if (count[3:0] == 4'b1111)
                    count[3:0] <= 4'b0000;
                else
                    count[3:0] <= count[3:0] + 4'b0001;
            end
    end
    assign q[3:0] = count[3:0];
endmodule

module RateDivider(Clock, reset_n, enable, d);
    input [27:0] d;
    input reset_n;
    input Clock;
    output enable;
    reg [27:0] d_old;
    reg [27:0] count;

    always @(posedge Clock, negedge reset_n)
    begin
        if (reset_n == 1'b0)
            count[27:0] <= d[27:0];
        else if (count[27:0] == 28'd0)
            count[27:0] <= d[27:0];
        else if (d[27:0] != d_old[27:0])
            begin
                count[27:0] <= d[27:0];
                d_old[27:0] <= d[27:0];
            end
        else
            count[27:0] <= count[27:0] - 28'd1;
    end
    assign enable = (count[27:0] == 28'd0) ? 1'b1 : 1'b0;
endmodule

module hexDecoder (SW, HEX);
```

```
    input [3:0] SW;
    reg [6:0] result;
    output reg [6:0] HEX;

    always @(*)
    begin
        case (SW[3:0])
            4'b0000: HEX[6:0] = 7'b1000000;
            4'b0001: HEX[6:0] = 7'b1111001;
            4'b0010: HEX[6:0] = 7'b0100100;
            4'b0011: HEX[6:0] = 7'b0110000;
            4'b0100: HEX[6:0] = 7'b0011001;
            4'b0101: HEX[6:0] = 7'b0010010;
            4'b0110: HEX[6:0] = 7'b0000010;
            4'b0111: HEX[6:0] = 7'b1111000;
            4'b1000: HEX[6:0] = 7'b0000000;
            4'b1001: HEX[6:0] = 7'b0010000;
            4'b1010: HEX[6:0] = 7'b0001000;
            4'b1011: HEX[6:0] = 7'b0000011;
            4'b1100: HEX[6:0] = 7'b1000110;
            4'b1101: HEX[6:0] = 7'b0100001;
            4'b1110: HEX[6:0] = 7'b0000110;
            4'b1111: HEX[6:0] = 7'b0001110;
            default: HEX[6:0] = 7'b1000000;
        endcase
    end
endmodule
```

3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must include screenshots of simulation output in the prelab. You will also need to think about how to simulate this kind of circuit. For example, how many 50 MHz clock pulses will you need to simulate to show that the RateDivider is properly outputting a 1 Hz pulse?

Solution.



3 Part III

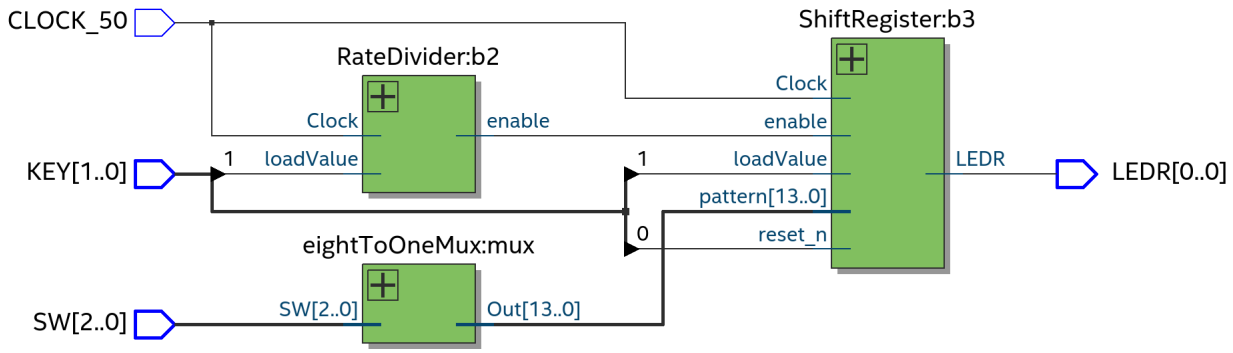
1. Use Table 1 to determine your codes and bit-width.

Solution.

Letter	Morse Code	Pattern Representation (Pattern is 14 its)
S	● ● ●	10101000000000
T	–	11100000000000
U	● ● –	10101110000000
V	● ● ●–	10101011100000
W	● – –	10111011100000
X	– ● ●–	11101010111000
Y	– ● ––	11101011101110
Z	– – ●●	11101110101000

2. Design your circuit by first drawing a schematic of the circuit. Think and work through your schematic to make sure that it will work according to your understanding. You can use Figure 4 as your starting point. You should include the schematic in your prelab.

Solution.



3. Write a Verilog module that realizes the behaviour described in your schematic. You should also inlucde the Verilog code as part of your prelab.

Solution.

```
module Morse(LEDR, SW, KEY, CLOCK_50);
    input [2:0] SW;
    input [1:0] KEY;
    input CLOCK_50;
    output [0:0] LEDR;
    reg [13:0] Pattern;
    wire flash;

    always @(*)
    begin
        case(SW[2:0])
            3'b000: Pattern[13:0] = 14'b10101000000000;
            3'b001: Pattern[13:0] = 14'b11100000000000;
            3'b010: Pattern[13:0] = 14'b10101110000000;
            3'b011: Pattern[13:0] = 14'b10101011100000;
            3'b100: Pattern[13:0] = 14'b10111011100000;
            3'b101: Pattern[13:0] = 14'b11101010111000;
            3'b110: Pattern[13:0] = 14'b11101011101110;
```

```
            3'b111: Pattern[13:0] = 14'b11101110101000;
            default: Pattern[13:0] = 14'b00000000000000;
        endcase
    end

    RateDivider b2(
        .enable(flash),
        .loadValue(KEY[1]),
        .Clock(CLOCK_50)
    );

    ShiftRegister b3(
        .LEDR(LEDR[0]),
        .enable(flash),
        .Clock(CLOCK_50),
        .reset_n(KEY[0]),
        .loadValue(KEY[1]),
        .pattern(Pattern)
    );
endmodule

module RateDivider(enable, loadValue, Clock);
    input loadValue, Clock;
    output enable;
    reg [24:0] count;

    always @(posedge Clock)
    begin
        if (loadValue == 1'b0)
            count <= 25'd24_999_999;
        else
            begin
                if (count == 9'd0)
                    count <= 25'd24_999_999;
                else
                    count <= count - 25'd1;
            end
    end
    assign enable = (count == 25'd0) ? 1 : 0;
endmodule

module ShiftRegister(LEDR, enable, Clock, reset_n, loadValue, pattern);
    input enable, Clock, reset_n, loadValue;
    input [13:0] pattern;
    output reg LEDR;
    reg [13:0] Pattern;

    always @(posedge Clock, negedge reset_n)
    begin
        if (reset_n == 1'b0)
            begin
                LEDR <= 1'b0;
                Pattern <= 14'b0;
            end
        else if (loadValue == 1'b0)
```

```
74              begin
75                  LEDR <= 1'b0;
76                  Pattern <= pattern;
77              end
78          else if (enable == 1)
79              begin
80                  LEDR <= Pattern[13];
81                  Pattern[13:0] = {Pattern[12:0], 1'b0};
82              end
83      end
84  endmodule
```

4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. If you cannot compile your design to simulate it, then you should at the very least (a) decide what are the inputs settings you want to model and (b) draw waveforms of what you expect your output signals to be for those inputs.

Solution.