Bialystok University of Technology

Faculty of Computer Science

EMBEDDED SYSTEMS FCS-00072

Report

Lab 3: Registers and counters

Teacher: Adam Klimowicz

Student: Sevda Ghasemzadehnaghadehy

Task 1

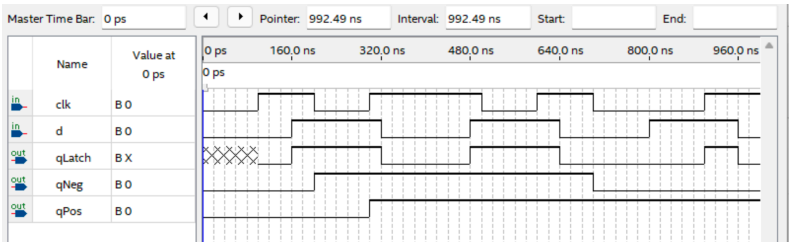
Objective:

1. Create a new Quartus project.
2. Write a Verilog file that instantiates the three storage elements. Figure 2 gives a behavioral style of Verilog code that specifies the gated D latch in Figure 1. Use a similar style of code to specify the flip-flops in Figure 1.
3. Compile your code and use the Technology Map Viewer to examine the implemented circuit. Verify that the latch uses one lookup table and that the flip-flops are implemented using the flip-flops provided in the target FPGA.
4. Create a Vector Waveform File (.vwf) that specifies the inputs and outputs of the circuit. Draw the inputs D and Clock as indicated in Figure 1. Use functional simulation to obtain the three output signals. Observe the different behavior of the three storage elements.

Verilog code:

```
module Lab3_Part1 (
    input wire clk,
    input wire d,
    output reg qLatch,
    output reg qPos,
    output reg qNeg
);

always @ (clk or d)
begin
    if (clk)
        qLatch = d;
end

always @ (posedge clk)
begin
    qPos <= d;
end

always @ (negedge clk)
begin
    qNeg <= d;
end

endmodule
```

Simulation



Conclusions:

In this part of the lab, I implemented three different storage elements: a gated D latch, a positive-edge triggered D flip-flop, and a negative-edge triggered D flip-flop. I used the same D and clock signals as inputs to all three components and observed their behavior through waveform simulation. The gated latch followed the input D only when the clock was high, while the flip-flops updated their output only on the rising or falling edge of the clock, depending on their type. The simulation clearly showed the differences between level-sensitive and edge-triggered storage elements, and confirmed that each component behaved as expected.

Task 2

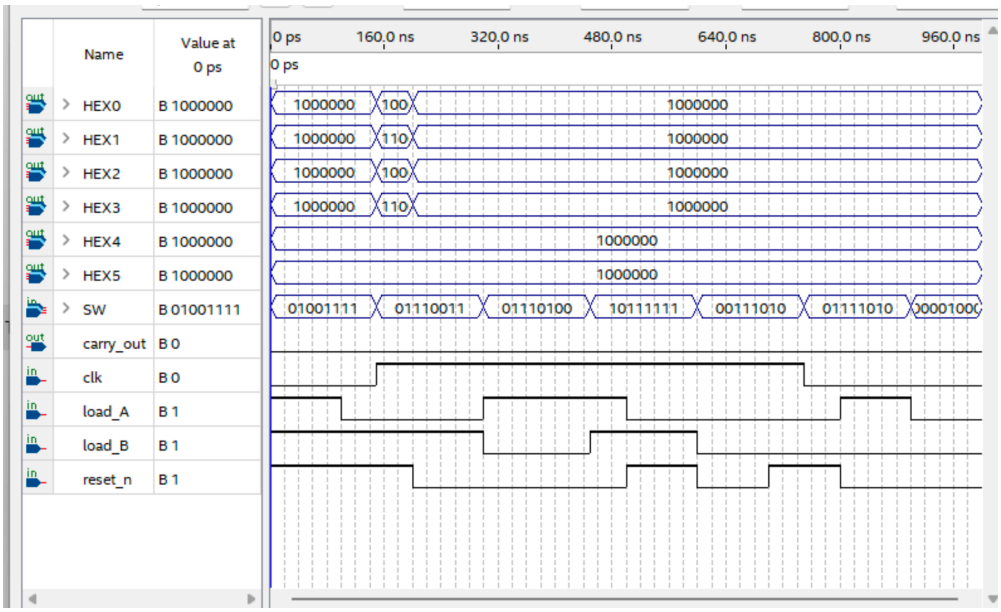
Objective:

1. Create a new Quartus project which will be used to implement the desired circuit
2. Write a Verilog file that provides the necessary functionality. Use an active-low asynchronous reset and a clock input.
3. Compile the circuit.
4. Test its functionality by changing input signal in proper order and observing the output displays in simulator.

Verilog code:

```
module Lab3_Part2 (
    input [7:0] SW,
    input clk,
    input reset_n,
    input load_A,
    input load_B,
    output reg [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
    output carry_out
);

    reg [7:0] A, B;
    reg [8:0] S;

    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            A <= 8'd0;
            B <= 8'd0;
        end else begin
            if (load_A)
                A <= SW;
            if (load_B)
                B <= SW;
        end
    end


    always @(*) begin
        S = A + B;
    end

    assign carry_out = S[8];

    function [6:0] seg;
        input [3:0] val;
        begin
            case (val)
                4'h0: seg = 7'b1000000;
                4'h1: seg = 7'b1111001;
                4'h2: seg = 7'b0100100;
                4'h3: seg = 7'b0110000;
                4'h4: seg = 7'b0011001;
                4'h5: seg = 7'b0010010;
                4'h6: seg = 7'b0000010;
                4'h7: seg = 7'b1111000;
                4'h8: seg = 7'b0000000;
                4'h9: seg = 7'b0010000;
                4'hA: seg = 7'b0001000;
                4'hB: seg = 7'b0000011;
                4'hC: seg = 7'b1000110;
                4'hD: seg = 7'b0100001;
                4'hE: seg = 7'b0000110;
                4'hF: seg = 7'b0001110;
                default: seg = 7'b1111111;
            endcase
        end
    endfunction

      always @(*) begin
        HEX0 = seg(S[3:0]);
        HEX1 = seg(S[7:4]);
        HEX2 = seg(B[3:0]);
        HEX3 = seg(B[7:4]);
        HEX4 = seg(A[3:0]);
        HEX5 = seg(A[7:4]);
    end

endmodule
```

Simulation:



Conclusions:

In this part of the lab, I designed a circuit that loads two 8-bit binary numbers, adds them, and displays the result using 7-segment displays. I used two control signals, `load_A` and `load_B`, to load the values of A and B from the switches. The addition was done using an 8-bit adder, and the outputs were displayed on HEX displays. During simulation, I loaded the value 79 into A and 34 into B. The sum was 113, which exceeds the 2-digit decimal display range, so the carry-out signal correctly went high. Although the 7-segment outputs could not display all 3 digits of the result, the carry-out signal confirmed the overflow. The simulation proved that the register loading, addition, and carry detection parts of the circuit were working correctly.and changed S during the simulation to check if the output M matched the correct input. The waveform results showed that the output changed as expected in each case.

Task 3

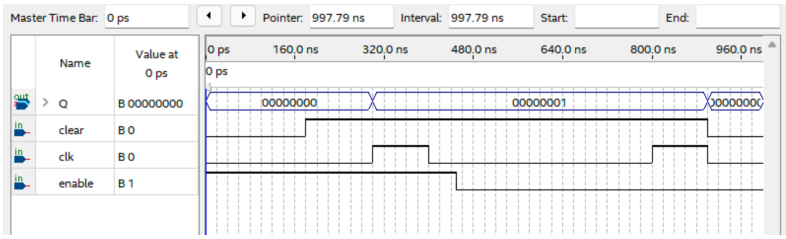
Objective:

1. Write a Verilog file that defines an 8-bit counter by using the structure depicted in Figure 3. Your code should include a T flip-flop module that is instantiated eight times to create the counter. Compile the circuit. How many logic elements (LEs) are used to implement your circuit?
2. Simulate your circuit to verify its correctness.

Verilog code:

```
module Lab3_Part3 (
    input clk,
    input enable,
    input clear,
    output reg [7:0] Q
);

    always @(posedge clk or negedge clear) begin
        if (!clear)
            Q <= 8'd0;
        else if (enable) begin
            Q[0] <= ~Q[0];
            Q[1] <= Q[1] ^ Q[0];
            Q[2] <= Q[2] ^ (Q[1] & Q[0]);
            Q[3] <= Q[3] ^ (Q[2] & Q[1] & Q[0]);
            Q[4] <= Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
            Q[5] <= Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
            Q[6] <= Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
            Q[7] <= Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
        end
    end

endmodule
```

Simulation:



Conclusions:

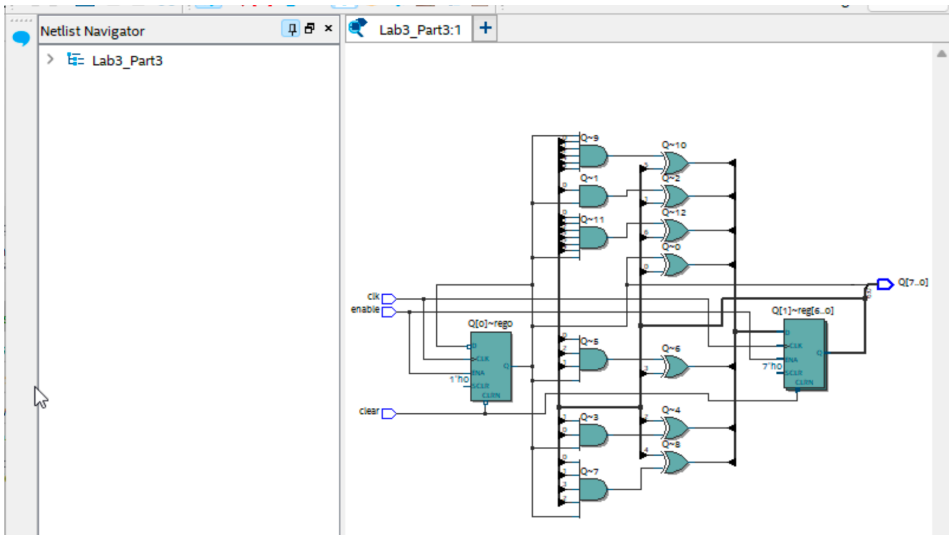
In this part of the lab, I implemented an 8-bit synchronous counter using T-type flip-flops. I used active-low asynchronous clear and an enable signal to control the counter. The design was constructed by connecting the T flip-flops so that each stage toggled based on the state of the previous ones, allowing binary counting. I tested the circuit using a waveform simulation, and the results confirmed correct behavior: the counter incremented its value on each positive clock edge when enable was high, and reset to zero when the clear signal was low. The simulation matched the expected synchronous operation of the counter.

3. Implement a four-bit version of your circuit and use the Quartus RTL Viewer to see how the Quartus software synthesized the circuit. What are the differences in comparison with Figure 3?

Verilog code:

```
module Lab3_Part3_4bit (
    input clk,
    input enable,
    input clear,
    output reg [3:0] Q
);

    always @(posedge clk or negedge clear) begin
        if (!clear)
            Q <= 4'd0;
        else if (enable) begin
            Q[0] <= ~Q[0];
            Q[1] <= Q[1] ^ Q[0];
            Q[2] <= Q[2] ^ (Q[1] & Q[0]);
            Q[3] <= Q[3] ^ (Q[2] & Q[1] & Q[0]);
        end
    end

endmodule
```

RTL



Conclusions:

In the final part of this task, I implemented a 4-bit version of the T flip-flop-based synchronous counter and examined its structure using the Quartus RTL Viewer. The visualized design showed that each output bit was generated using a flip-flop connected through a chain of logic gates, matching the structure depicted in Figure 3. The enable and clear signals were properly routed to all flip-flops, and the toggle behavior was driven by the logical AND of previous outputs, confirming the correct T-type counter architecture. This view helped me understand how my Verilog code was synthesized into actual hardware components.

Task 4

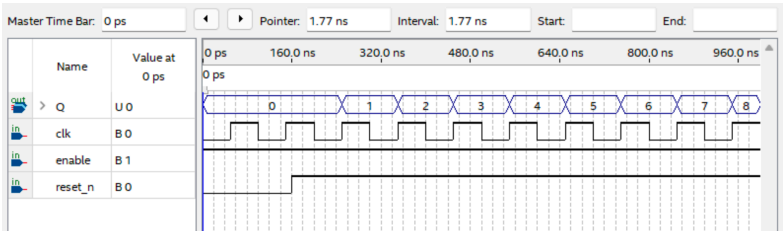
Objective:

Another way to specify a counter is by using a register and adding 1 to its value. Compile a 16-bit version of this counter and determine the number of LEs needed. Use the RTL Viewer to see the structure of this implementation and comment on the differences with the design from Part IV. Simulate circuit using waveforms.

Verilog code:

```
module Lab3_Part4 (
    input clk,
    input enable,
    input reset_n,
    output reg [15:0] Q
);

    always @(posedge clk or negedge reset_n) begin
        if (!reset_n)
            Q <= 0;
        else if (enable)
            Q <= Q + 1;
    end

endmodule
```

Simulation:



Conclusions:

In this part of the lab, I implemented a 16-bit counter using a behavioral approach by defining a register and incrementing its value with the statement `Q <= Q + 1`. The design included an active-low asynchronous reset and an enable signal. After compiling the design, I observed that the counter utilized 9 logic elements (ALMs) and 16 registers, which is reasonable for a 16-bit wide counter. The RTL Viewer showed a simpler structure compared to the T flip-flop design in Part III, where the counter was represented as a single register block and an adder. I also simulated the circuit using waveform inputs and confirmed that the counter behaved correctly, resetting when `reset_n` was low and incrementing on each positive clock edge when enabled.