Experiment 2 - Sequential Synthesis and FPGA Programming

Shirin Jamshidi - 810199570, Mahya Shahshahani - 810199598

***Abstract*:** In this experiment we learned about state machines, and we got familiar with FPGA programming and implementation.

***Keywords*:** *state machine - sequence detector - Huffman coding style - design simulation - one pulser - FPGA programing*

Introduction

The first goal of this experiment was to introduce the concepts of state machines that are mostly used for controllers. The second goal was to get familiar with FPGA devices and implementation.

I. Experiments

A. Onepulser

The system needs a ClkEn to control the clock when the circuit is implemented on an FPGA board.

One pulser is connected to a push-button on the board so when the button is pressed it creates a single pulse which is synchronized with the system's clock. So, the output of one pulser is ClkEn that will be the input of sequence detector and the counter.

The state machine diagram of this structure is shown in Figure 1.

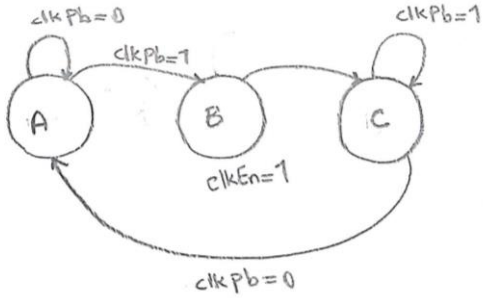

Figure 1: Onepulser state machine diagram.

The implemented onepulser worked properly and its simulation result is shown in figure 2.

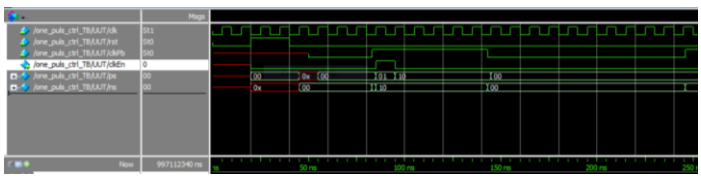

Figure 2: Onepulser simulation results.

B. Finite State Machine and the Counters

In the Huffman style FSM, when the push-button is pressed, the FSM checks the serIn input to determine whether to transition to the start state or remain in the current state. The FSM follows the start and end flow with the push-button pressed for each input received. Initially, the FSM waits for serIn to become 1 before starting data transmission. In the next state, the port number is extracted, requiring the port counter to be enabled and waiting until all port bits are extracted. Subsequently, the number of bits, n, is extracted over four clock cycles, and the load value of the data transfer counter is prepared. The data transfer counter is then enabled and counts for the next n consecutive clock cycles, while serOutvalid remains 1 during this state. Finally, the "done" signal is set in the last state of the controller. We added an empty state, E, to make sure data has loaded properly.

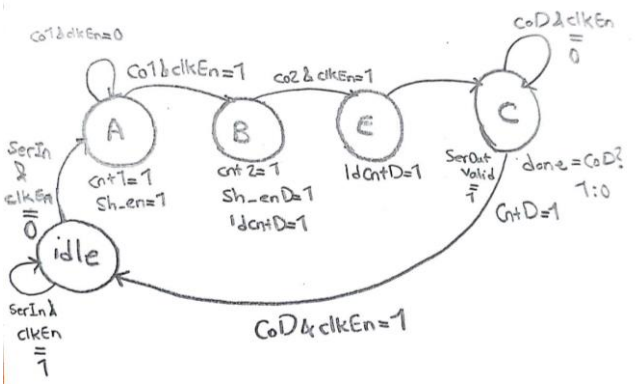

Figure 3: Controller state machine diagram.

Verilog code:

```
always@(ps,clkEn, co1, co2, coD, serIn)begin
    case (ps)
        idle: ns = serIn & clkEn? idle : A;
        A: ns = co1&clkEn ? B : A;
        B: ns = co2&clkEn ? E : B;
                E: ns = C;
        C: ns = coD&clkEn ? idle : C;
        default: ns = idle;
    endcase
end

always @(ps,coD) begin
    {cnt1, sh_en, cnt2, sh_enD, ldcntD, serOutValid, cntD, done} = 8'b00000000;
    case (ps)
        idle: ;//rst = 2'b1;
        A: {cnt1, sh_en} = 2'b11;
        B: {cnt2, sh_enD, ldcntD} = 3'b111;
        E: ldcntD = 1'b1;
        C: begin
            {serOutValid, cntD} = 2'b11;
            done = coD ? 1 : 0;
        end
        default;
    endcase
end

always @(posedge clk , posedge rst) begin
   if (rst) ps <= idle;
   else ps <= ns;
end
```

We can see the counter counts down properly.

Figure 4: Down counter simulation results.

C. Shift Registers, Demultiplexer, and SSD

We need a shift register to the left that stores the serial input for two clock cycles to extract the port output which goes to a demultiplexer.

We need a demultiplexer to send the data based on the port number to the MSSD module with p0 to p3 output ports.

We need another shift register to the left so it can store the 4 bits after the two port bits, so we know how many bits are going to be taken for the actual data. Each clkEn (push-button pressed) posedge a shift to left will happen and the shift register will get the amount that serIn has.

Also, we have a seven-segment which shows the down count of the data transfer counter that had to count from n to 1 and then send the CoD signal. The data transfer counter output (count) will go to the seven-segment display. The segment display receives a 4-bit input and displays the HEX value on the output as stated in the manual.

Note: For the implementation part we changed the code and made Demux and SSD outside of our MSSD. The simulation is built from second session's code, not the finalized version of it.

Output of Data shift register is NumData, for Demux is P and SSD's output is Num.

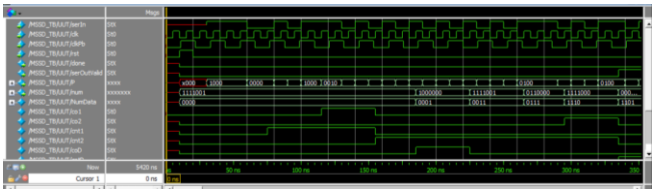

Figure 5: Data Shift Register, Demux and SSD simulation results.

D. MSSD Simulation

Multi-channel Synchronous Serial communication Demultiplexer is referred to as MSSD. The source provides a stream of 1-bit data that includes the size of the data to be transferred, the destination port, and the actual data.

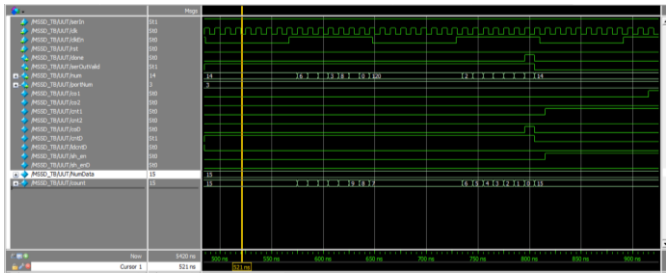

Figure 6: Top module simulation results. Sequence one.

The MSSD starts transmitting when serIn makes a 1 to 0 transition then the next two bits are the port number and after that, the next four bits are the number of bits, n. After 1+2+4+n clock cycles after serIn becomes 0, it is shown in Fig. 5, it will return to 1 and another transmission begins. So before receiving the actual data, we must wait for the seven bits.

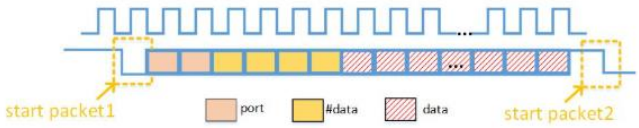

Figure 7: Transmission Flow

This top module should include all six parts (one-pulse, FSM, counters, shift registers, demultiplexer, and the seven-segment display modules) connected.

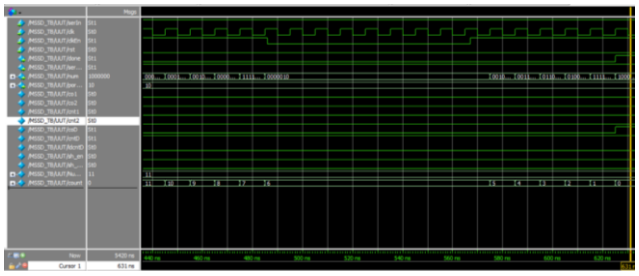

Figure 8: Top module simulation results. Sequence two.

E. MSSD Implementation

To implement the codes on an FPGA board we had to use the Quartus application. So we made a project and uploaded our codes on the Quartus app then we determined the pin planner part in the app so we could show the Done output on a green LED and SerOutvalid on a red LED on the board and the count-down was shown on the board's seven segments, moreover we specified one of the switches as the clkPB which was the one-pulser's input and two other switches that were used as serIn and rst inputs.

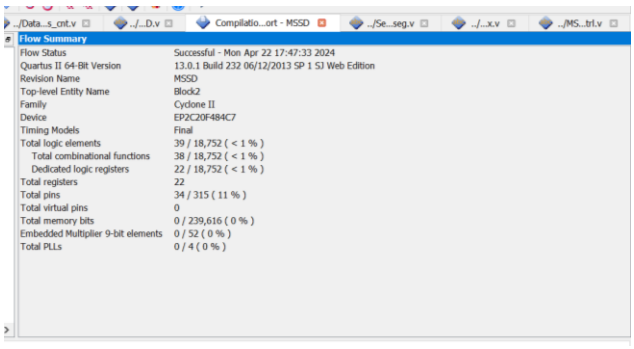

Figure 9: Resource Utilization

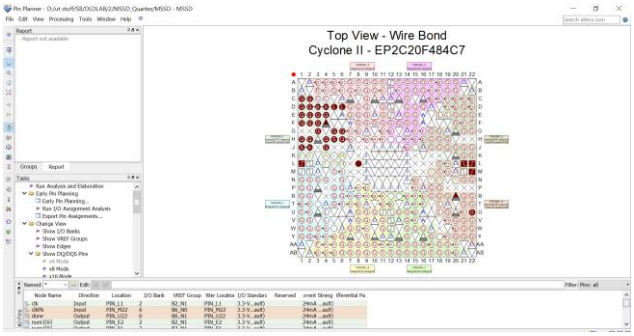

Figure 10: Top View of Pin Assignment

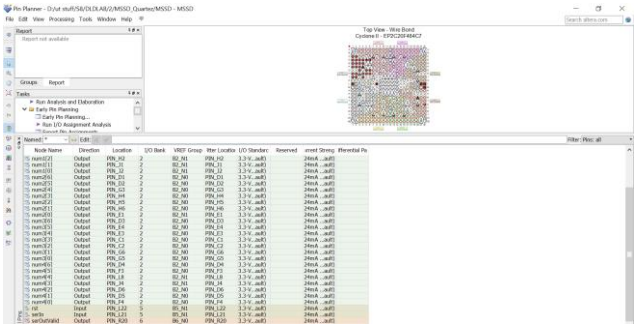

Figure 11: Detailed Pin Assignment

Final implementation on FPGA:

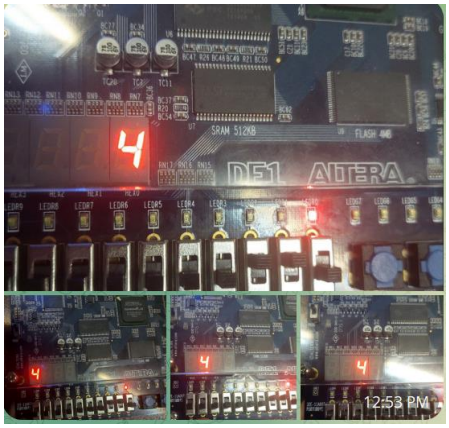

Figure 12: FPGA, showing the program works properly for all 4 SSDs.

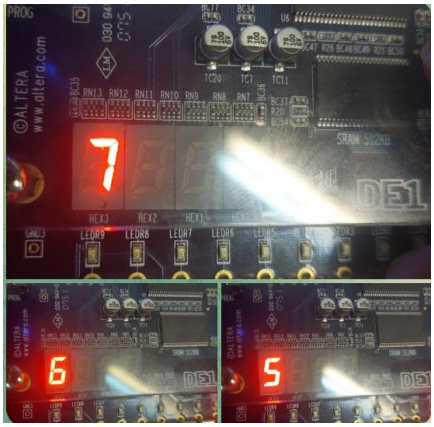

Figure 12: FPGA output counts down.

F. Conclusion

In this experiment, we were able to make a serial transmitter using one pulser, sequence detector, counter, and seven-segment display modules. First, we made our design in ModelSim and Quartus, and after making sure that our design worked correctly, we implemented it on the FPGA board and observed the results.

G. References

[1] Katayoon Basharkhah and Zahra Jahanpeima and Prof. Zain Navabi, *Digital Logic Laboratory*, University of Tehran, Spring 1403.