Experiment 2 - Sequential Synthesis and FPGA Programming

Amir Naddaf Fahmideh,
810101540,
Mostafa Kermani nia,
810101575

***Abstract*— This document is the report of second experiment of logic design lab. In this experiment we build a multi-channel serial transmitter and implement it on an FPGA to get familiar with concepts of state machines, synthesising our module and FPGA implementation.**

***Keywords*— FPGA, Synthesis, One Pulser, SSD, Finite State Machines, Demultiplexer, MSSD, Serial Transmitter**

I. Introduction

The experiment aims to accomplish two main objectives: firstly, to introduce the concepts of state machines that are mostly used for controllers; and secondly, to get familiar with FPGA devices and implementation. And throughout the experiment, participants will learn about state machines, sequence detectors, Huffman coding style, design simulation, synthesis techniques, and FPGA programming and implementation.

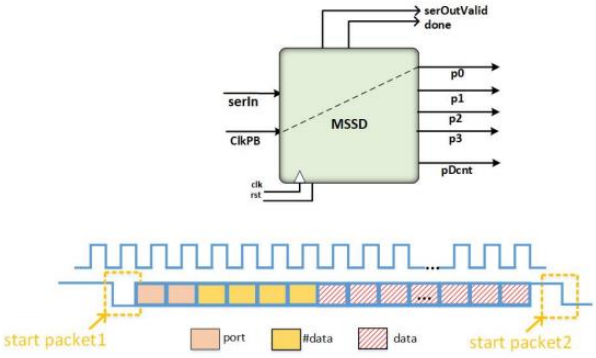


Fig. 1 Multi-channel Serial Transmitter

II. RTL Design of Multi-channel Serial Transmitter

In this experiment, we will design a Multi-channel Synchronous Serial communication Demultiplexer referred to as MSSD. The top-level view and ports of this design are shown in fig 1, and figure 2 shows the RTL design of this experiment.

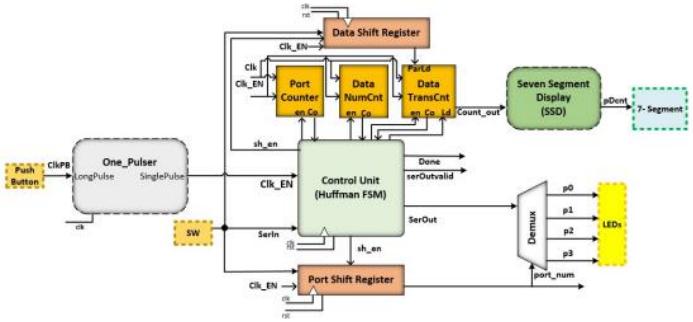


Fig. 2 Overall view of the design

A. Onepulser

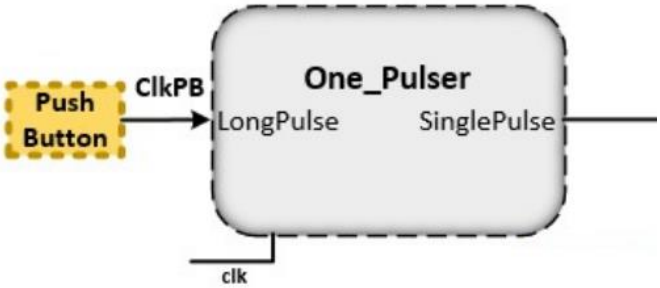


Fig. 3 One_pulser in RTL design

1. Explanation

This part of our RTL design that shown in fig 3, provides a clock-enable input for the controller of this design. This input (clkEn) is used for controlling the clock when the circuit is implemented on an FPGA board. The one-pulser connects to a push-button on your board (clkPB) and when pressed it creates a single pulse that is synchronized with the system clock. The output of this circuit connects to the clock enable input (clkEn) of the sequence detector, shift registers, and counters.

2. State machine diagram

As we can see in fig 4, we will stay in state A until push clkPB, and when we push that, we will go to state B, and drive clkEn signal, then at next posedge of clk, we will go to state C and while the clkPB is pushed, we will stay there and then we will go to state A after releasing the clkPB

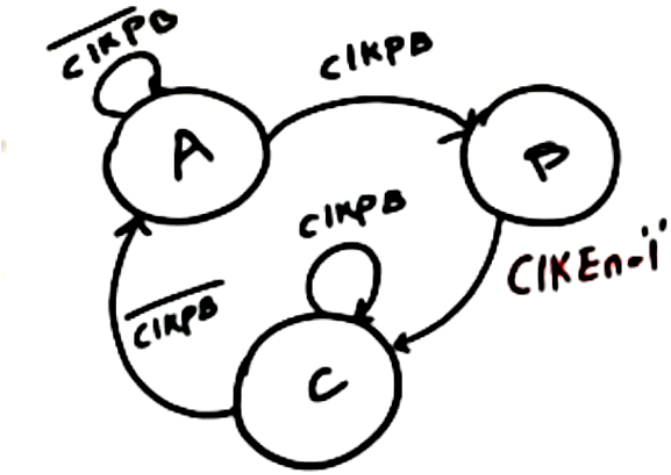


Fig. 4 State machine diagram of the one-pulser

3. Simulation result

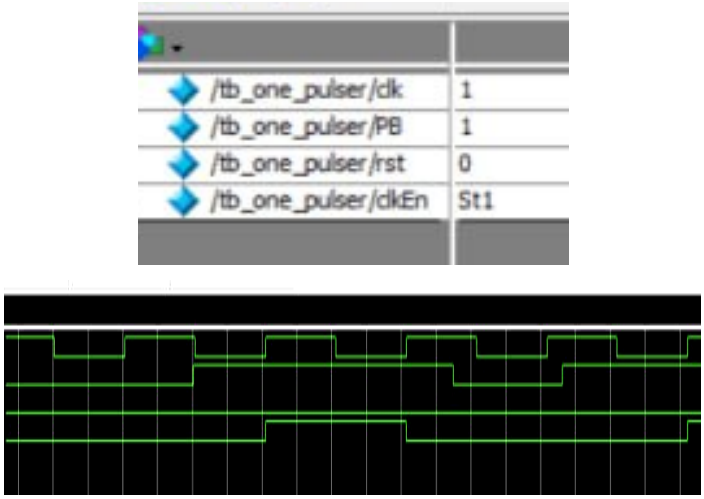


Fig. 5 simulation of the one-pulser

B. Finite State Machine and the Counters

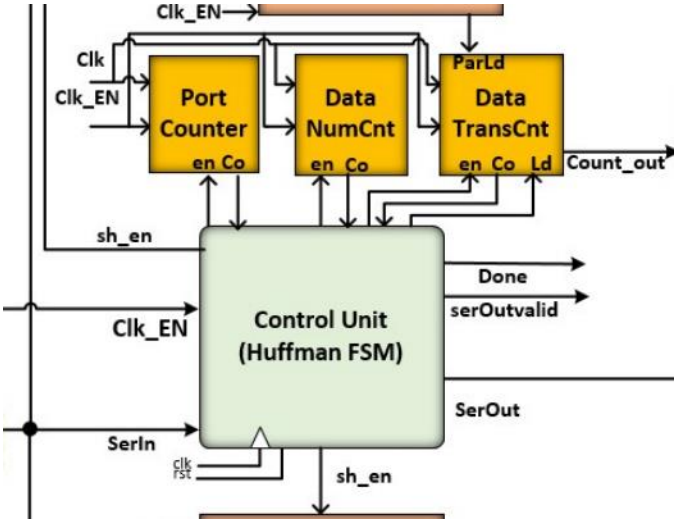


Fig. 6 Controller and counters in RTL design

1. Controller Explanation

When a zero comes on serIn it begins to work. In next two clocks it gets a two bit number which selects the output port (saving in a shift register). After that in the next four clocks it receives a 4 bit number that shows how many bits are going to transfer(saving in a shift register) and it loads the data into the down counter. Let the 4 bit number be k. In the next k clocks the data is going to the selected port and the out valid is 1(down counter counts from k to 0). Then after that the Done signal is 1 for 1 clock and then it goes to Idle state or get port state based on SerIn(if 0 it goes to get port, else Idle).

2. State machine diagram of controller

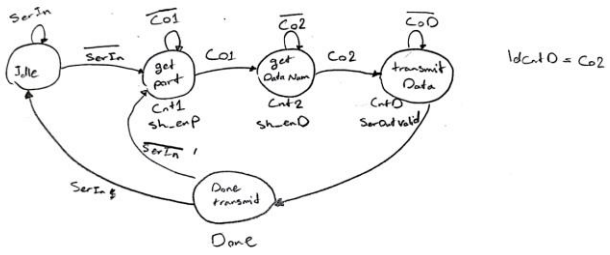


Fig. 7 State machine diagram of the controller

3. Verilog codes of Controller

```
module Controller(input clk, clkEn, rst, SerIn, Co1, Co2, CoD,
                  output reg Cnt1, Cnt2, CntD, ldcntD, Sh_enP, Sh_enD, SerOutValid, Done);
        parameter Idle = 3'b000, get_port = 3'b001, get_DataNum = 3'b010,
                  transmit_Data = 3'b011, Done_transmit = 3'b100;
        reg [2:0] ps, ns;
        always @(posedge clk, posedge rst)begin
                if (rst)
                        ps <= Idle;
                else if (clkEn)
                        ps <= ns;
        end

        always @(ps, SerIn, Co1, Co2, CoD)begin
                case(ps)
                Idle: ns = SerIn ? Idle : get_port;
                get_port: ns = Co1 ? get_DataNum : get_port;
                get_DataNum: ns = Co2 ? transmit_Data : get_DataNum;
                transmit_Data: ns = CoD ? Done_transmit : transmit_Data;
                Done_transmit: ns = SerIn ? Idle : get_port;
                default ns = Idle;
                endcase
        end

        always @(ps)begin
                {Cnt1, Cnt2, CntD, ldcntD, Sh_enP, Sh_enD, SerOutValid, Done} = 8'b0000_0000;
                case(ps)
                Idle: ;
                get_port: {Cnt1,Sh_enP} = 2'b11 ;
                get_DataNum: {Cnt2,Sh_enD} = 2'b11;
                transmit_Data: {CntD,SerOutValid} = 2'b11;
                Done_transmit: Done = 1'b1;
                endcase
        end
        assign ldcntD = Co2;
endmodule
```

Fig. 8 Verilog codes of Controller

4. Counters Explanation

There are Three counters in this design (As you can see in fig 5) . First of all we have port_cnt. Its duty is to count for two clocks (1 bit counter) and let the portNum shift register gets the port number which output goes into it.

Next one is DataNum_cnt. Its duty is to count for four clocks (2 bit counter) so the DataNum shift register gets the number of bits that are going to transfer.

Last one is a down counter named DataTrans_cnt. This counter counts from the number which is in DataNum shift register and it counts down to 0 and when its carryout becomes one it means the desired number of data is transmitted otherwise it has to transmit data until the carry out becomes one.

5. Simulation result

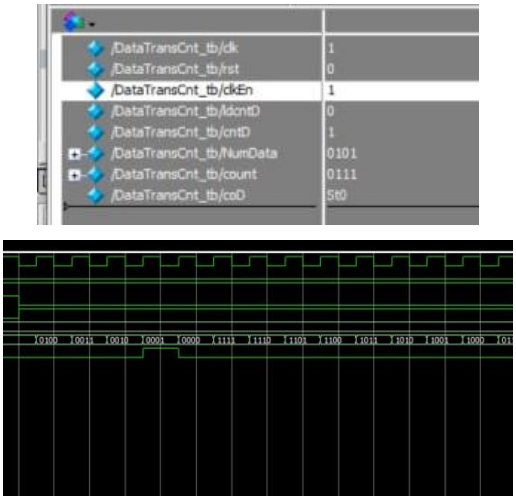


Fig. 9 simulation of the down-counter

C. Shift Registers, Demultiplexer, and SSD

1. Shift registers explanation

There are two shift registers (Shown in fig 9). One is for Port number (PortNum_shr). Its duty is to get the two bit of port number in two clock cycles. This port number will be used as the select input of the demultiplexer.

The other shift register is to get number of transmitted data (DataNum_shr). It gets the number of transmitted data, n in 4 clock cycles. The parallel output of this shift register is used as the load value of the data transfer counter.

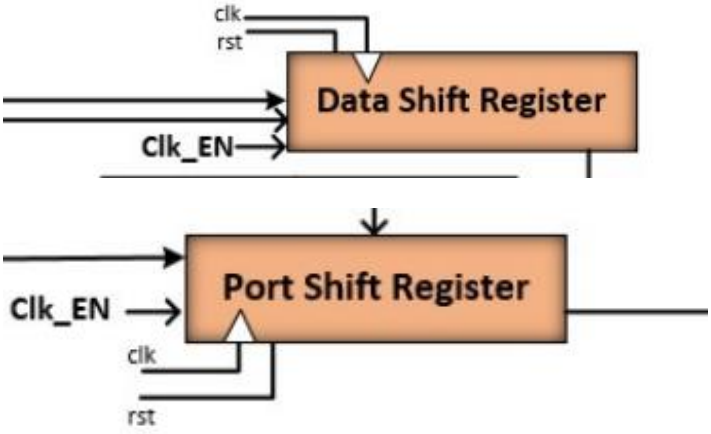


Fig. 10 Shift registers in RTL design

2. Data shift register simulation result

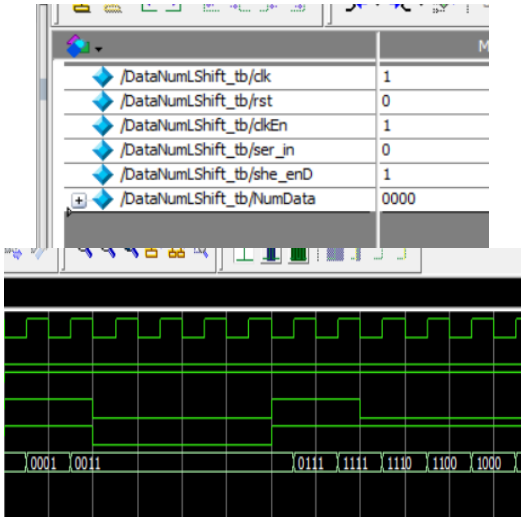


Fig. 11 simulation of the port shift register

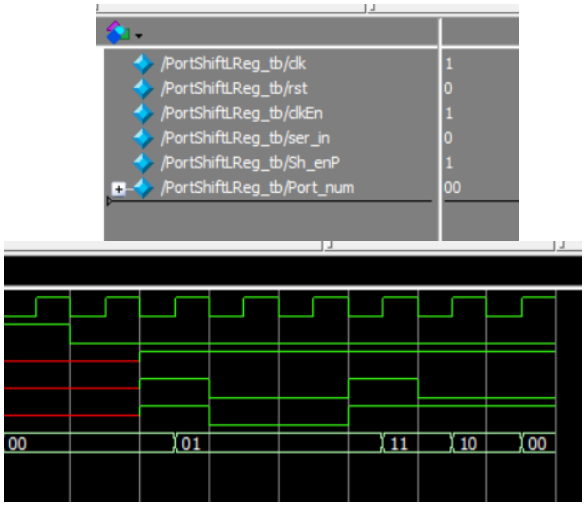


Fig. 12 simulation of the port shift register

3. Demultiplexer explanation and simulation result

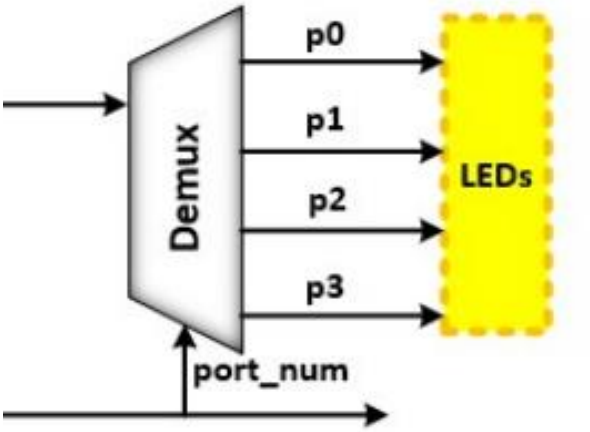
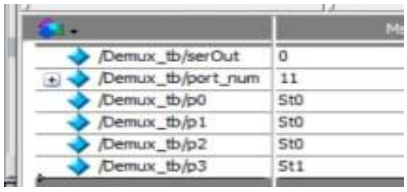


Fig. 13 Demultiplexer in RTL design

The demultiplexer is used to connect the SerIn into the selected output port by the number of selected port as input (fig 11).



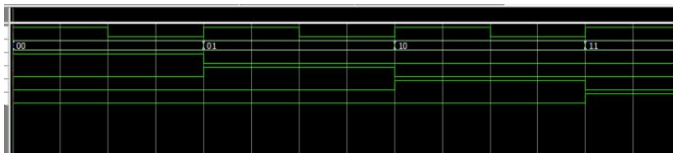


Fig. 14 simulation of the dmux module

4. SSD explanation and simulation result

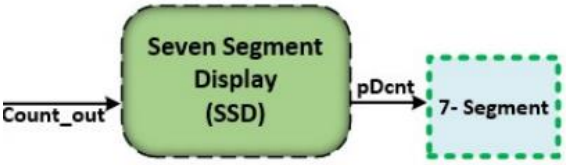


Fig. 15 SSD in RTL design

SSD is for displaying the counter output on the seven segments of the FPGA board. Seven-segment receives a 4-bit input and displays the HEX value on its 7-bit output (fig 12).

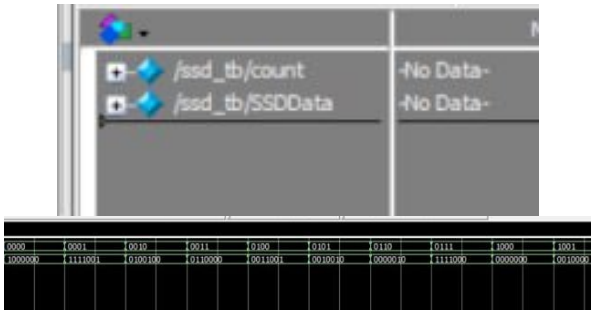


Fig. 16 simulation of the SSD module

D. MSSD (top module) Simulation

1. Explanation

As you can see in fig 1, MSSD is going to transfer a desired number of data into a selected ports. It has a SerIn as input which is used for all of functions that it needs. At first It won't do anything until the SerIn becomes zero. After that it gets two bit port number which is the number of selected output port. After that it gets a 4 bit number, n that shows how many bits are going to be transmit. After that it counts from n to 0 and transmit each SerIn input to the selected port.

For getting the portNum we need a two bit shift register and a one bit counter that shows when is the 2 bit received by shift register.

For getting the number of data that is going to transmit, n , we need a 4 bit shift register to save n and a 2 bit counter that shows the 4 bit data is loaded into shift register.

For counting from n to 0 we need a down counter and while it is counting the data is transmitting to the selected

output port which is selected by the Demultiplexer and its portNum input.

We also show the counter count down with the HEX number on our FPGA.

2. simulation result

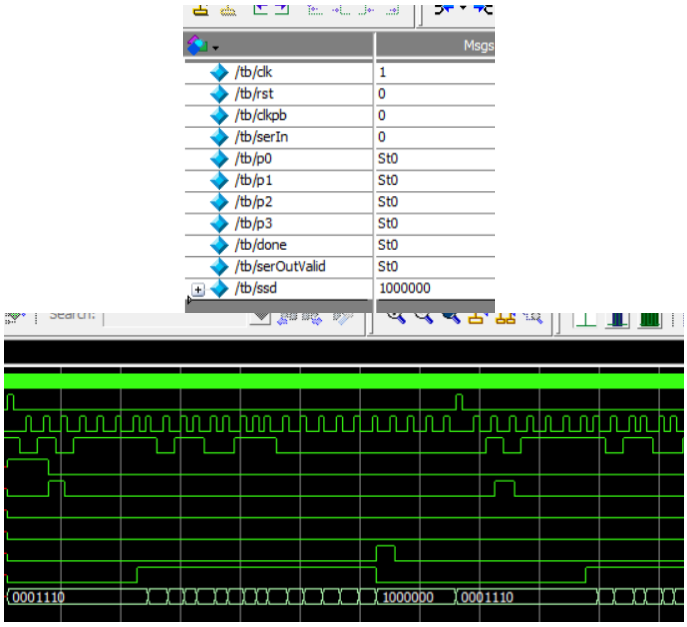


Fig. 17 simulation of the MSSD

E. MSSD (top module) Implementation

1. Explanation

We made a project and added all of our files into it. We set the settings of quartus project by considering the model of our FPGA board. Then we compiled it to see if it has any errors. After that we set our pin assignment to use it on FPGA board. Then we connect our computer to FPGA by an USB-blaster and programmed our FPGA board by that.

2. resource utilization

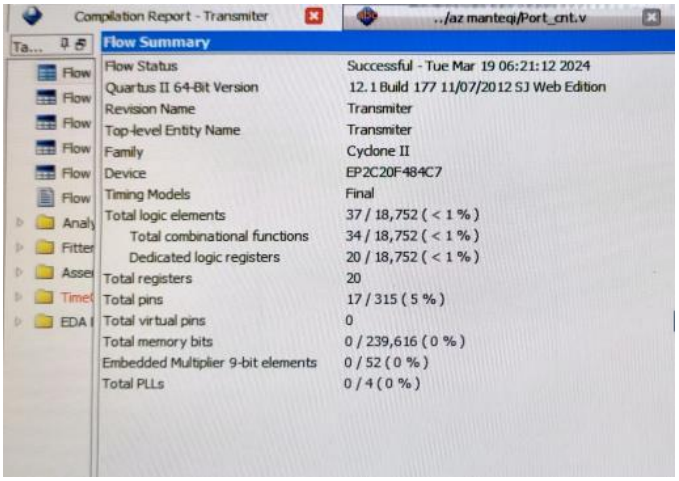


Fig. 18 Rresource utilization of the top module

3. pin assignment and FPGA board

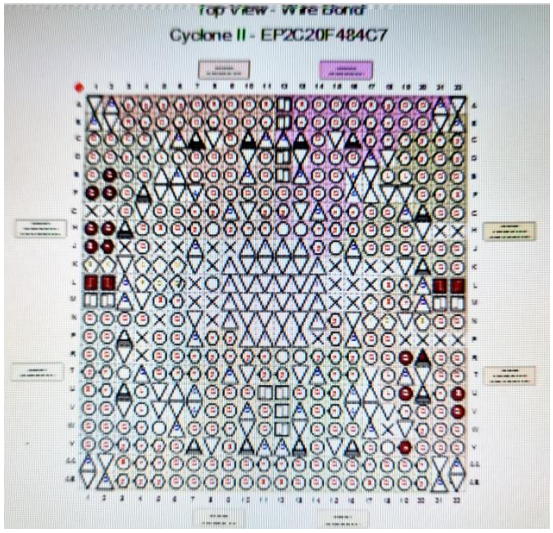


Fig. 19 Pin assignment selection in Quartus

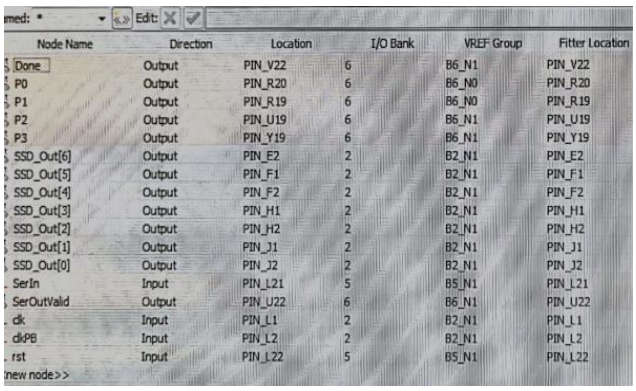


Fig. 20 Pin assignment selection in Quartus

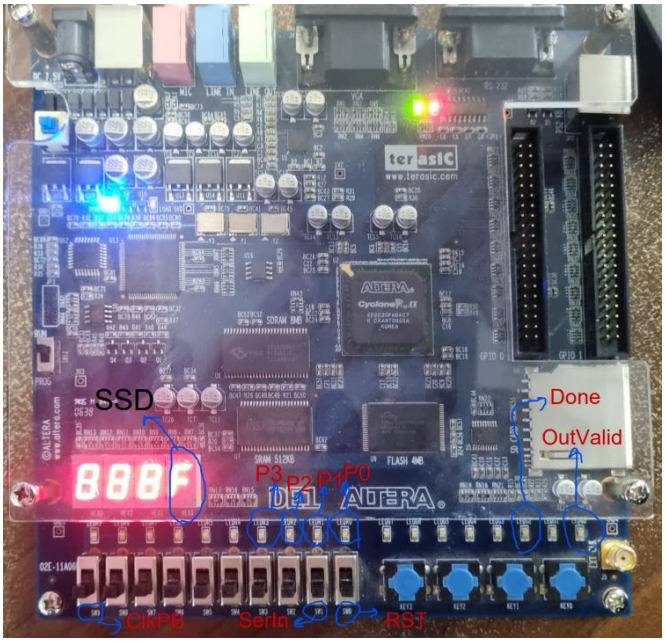


Fig. 21 Specify pins on the board

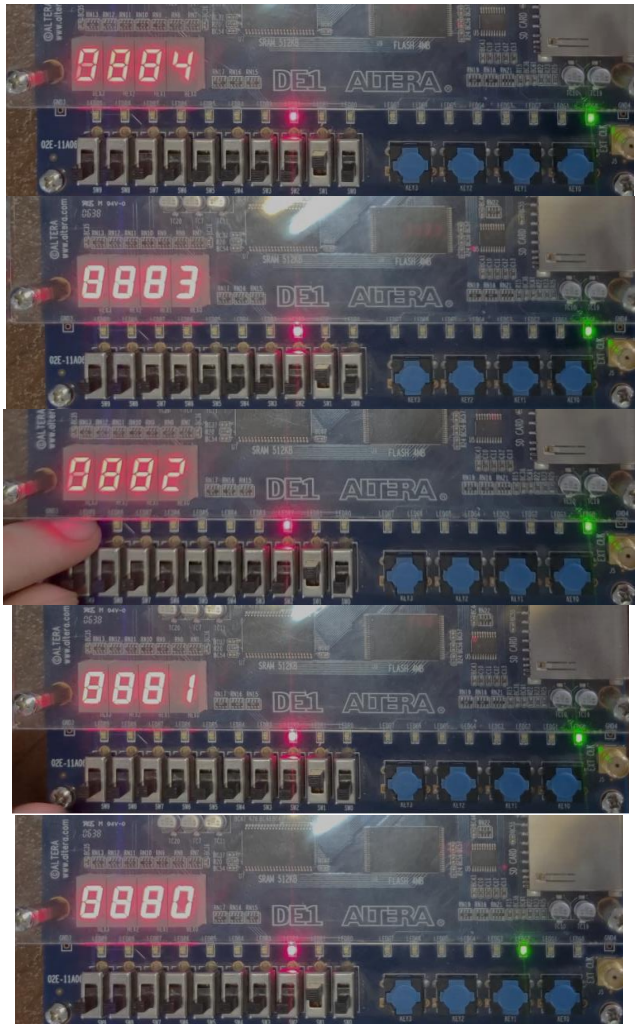


Fig. 22 Work procedure of module on FPGA

III. Conclusions

In this experiment, we designed a serial transmitter by using Haffman-coding-style. To visualize the result we used a FPGA and by programming it with our code we saw the functionality of serial transmitter visually. We learned how to program a FPGA and how to build a serial transmitter and we test it by the FPGA buttons and LEDs.

References

[1] K. Basharkhah “Experiment 2 Digital Logic Laboratory,” under supervision of Professor Z. Navabi, University of Tehran, Spring 1403