Experiment 2 - Sequential Synthesis and FPGA Programming

Erfan Mokhtari 810100208
AmirHosein Samoudi 810100108

***Abstract*— In this experiment, we have designed a Multi-Channel Synchronous Serial communication Demultiplexer (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.**

***Keywords*— MSSD - State machines - Sequence detectors - Huffman coding style – Synthesis - FPGA**

I. Introduction

In this experiment, we've designed a Multi-Channel Synchronous Serial communication Demultiplexer that takes a stream of 1-bit data and specify the destination port, number of data bits and the actual data. At the end we synthesised the design and implemented it on the FPGA board.

II. Multi-channel Serial Transmitter

Serial bits of data appear on the serIn input of MSSD. Transmission begins when serIn makes a 1 to 0 transition. Then, the two bits that follow indicate the port number and the next four bits indicate number of bits. The done signal is issued when the transmission is over and another transmission begins with another start-bit. We have used four seven-segment modules on the board (each of them for one destination port) to show that specific destination port's data count on it.

III. RTL Design

In the following parts we will explain every component in the RTL design.

1. One-pulser

The one-pulser module provides a clock-enable input for the controller of this design and other components in the data path. 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 (with the length of our main clock) and (clkPB) should get back to zero in order to generate a pulse next time that is having a 0 to 1 transition. The reason for using (clkEn) in our design is that we provide the inputs of the circuit by push buttons on the board. However, the main clock of our system (which is the FPGA internal clock) is so fast that we would miss multiple clock cycle in the time we are giving one data to it and the circuit would lose its functionality. Thus we control the flow of our circuit by using (clkPB).

Figure 1 shows the Verilog description of one-pulser.

```
module Onepulser(input clk,rst,clkPB , output reg clkEn);
    parameter [1:0] A = 0, B = 1, C = 2;
    reg [1:0] ps, ns ;

    always@(ps, clkPB) begin
      case (ps)
        A : ns = clkPB ? B : A;
        B : ns = C;
        C : ns = clkPB ? C : A;
        default : ns = A;

        endcase
    end

    always@(ps) begin

        clkEn = 1'b0;
        case (ps)
            B : clkEn = 1'b1;
        endcase
    end

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

endmodule
```

Fig 1. Verilog description of one-pulser

```
`timescale 1ns/1ns
module TB_Onepulser();
reg clk = 0 , rst = 0 , clkPB = 0;
wire clkEn;

    Onepulser UUt(clk , rst , clkPB , clkEn);

    always #5 clk = ~clk;

    initial begin
        #10 rst = 1;
        #10 rst = 0;
        #10 clkPB = 1;
        #30 clkPB = 0;

        #20 clkPB = 1;
        #40 clkPB = 1;
        #50 $stop;
    end
endmodule
```

Fig 2. Test-bench of one-pulser module

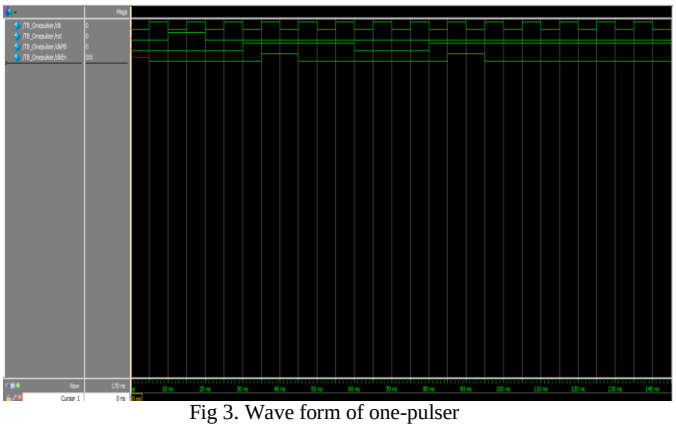

Fig 3. Wave form of one-pulser

In figure 3 it is shown that clkEn gets one in the first positive edge of the clock (after clkPB gets one) and return to zero in the next clock edge.

2. *Finite State Machine and the counters*

The FSM waits for the serIn to become one and after that, it again waits for it to be zero to start sending data. In the next state, the port number will be extracted. This requires the port counter (which is a 2-bits counter that count up to 2) to be enabled and wait in this state until all port bits are extracted. In the next state, the number of bits, n, will be extracted. This state takes four clock cycles (requires the num-data counter which counts up to four), and then the load value of the data counter is ready. After that, the data counter will be enabled and will count for the next n consecutive clock cycles. During this state, the serOutvalid remains one. The signal done will be set in the last state of the controller.

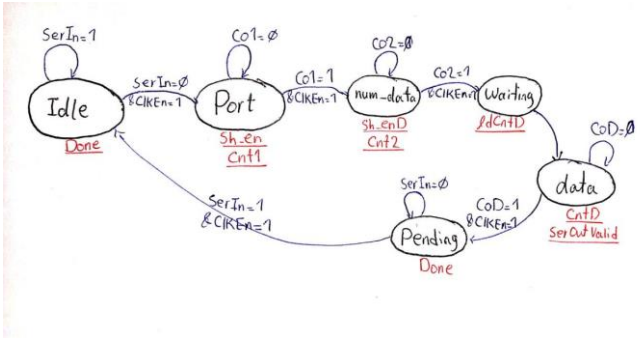

Fig 4. state diagram of the sequence detector

```
module Controller(input clk, rst, serIn, clkEn, co1, co2, coD, output reg cnt1, cnt2, cntD, ldcntD, sh_en, sh_enD, SerOutValid, done);

    parameter [2:0] Idle = 0, port = 1, num_data = 2 ,waiting = 3, data = 4, pending = 5;
    reg [2:0] ps,ns;

    always@(ps, serIn, clkEn, co1, co2, coD) begin

        case(ps)

            Idle :     ns = clkEn ?(serIn? Idle : port): Idle;
            port :     ns = clkEn ? (co1 ? num_data : port): port;
            num_data : ns = clkEn?(co2 ? waiting : num_data):num_data;
            waiting :  ns = data;
            data :     ns = clkEn?(coD ? pending : data): data;
            pending :  ns = clkEn?(serIn ? Idle : pending):pending;
            default :  ns = Idle;
        endcase
    end

    always@(ps) begin
        {cnt1, cnt2, cntD, ldcntD, sh_en, sh_enD, SerOutValid, done} = 8'b0;
        case(ps)
            Idle : done = 1'b1;
            port : {sh_en, cnt1} = 2'b11;
            num_data : {sh_enD, cnt2} = 2'b11;
            waiting : ldcntD = 1'b1;
            data : {cntD, SerOutValid} = 2'b11;
            pending : done = 1'b1;
        endcase
    end

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

endmodule
```

Fig 5. Verilog description of the FSM

```
module port_cnt(input clk, rst, clkEn, cnt1, output co1);

    reg [1:0] PO;
    always @ (posedge clk, posedge rst) begin
        if(rst) PO <= 2'b0;
        else PO <= clkEn ? (cnt1 ? PO+1'b1 : PO) : PO;
    end

    assign co1 = (PO == 2'b01)? 1'b1 : 1'b0;


endmodule
```

Fig 6. Verilog description of the port counter

```
module Datanum_cnt(input clk, rst, clkEn, cnt2, output co2);

    reg [1:0] PO;
    always @ (posedge clk, posedge rst) begin
        if(rst) PO <= 2'b0;
        else PO <= clkEn ? (cnt2 ? PO+ 1'b1 : PO) : PO;
    end

    assign co2 = & PO;


endmodule
```

Fig 7. Verilog description of the number of data counter

```
module DataTrans_cnt(input clk, rst, clkEn, cntD, ldcntD,input [3:0] NumData, output reg [3:0] count, output coD);

    always @ (posedge clk, posedge rst) begin
        if(rst) count <= 4'b0;
        else begin
            if (ldcntD) count <= NumData;
            else count <= clkEn ? (cntD ? count- 1'b1 : count) : count;
        end
    end

    assign coD = (count == 4'b0)? 1'b1 : 1'b0;


endmodule
```

Fig 8. Verilog description of the data counter

Data-counter is a down counter which shows the remaining bits of the data to be transferred.

3. *Shift Registers and Demultiplexers*

To extract the port number, we have used a shift register unit that shifts in and stores the value of the serial input for two clock cycles. This port number will be used as the select input of the demultiplexer. We need another shift register to store the number of data bits, n. The parallel output of this shift register is used as the load value of the data counter.

The demultiplexer gets the 7-bit output of the seven-segment display module as its input and base on the port number, put it on one of the 4 output port destinations (every output connects to a separate seven-segment).

```
module PortNum_shr(input clk, rst, clkEn, serIn, sh_en, output reg [1:0] PO);

    always@(posedge clk, posedge rst) begin
        if(rst) PO <= 2'b0;
        else
        PO <= clkEn?(sh_en ? {PO[0], serIn} : PO) : PO;
    end

endmodule
```

Fig 9. Shift register to store port number

```
module DataNum_shr(input clk, rst, clkEn, serIn, sh_enD, output reg [3:0] PO);

    always@(posedge clk, posedge rst) begin
        if(rst) PO <= 4'b0;
        else
        PO <= clkEn?(sh_enD ? {PO[2:0], serIn} : PO) : PO;
    end

endmodule
```

Fig 10. Shift register to store number of data bits (n)

```
module Demux(input [6:0] pdcnt, input [1:0] port_num, output reg [6:0] P0, P1, P2, P3);

    always@(port_num, pdcnt) begin
        {P0, P1, P2, P3} = 28'bz;
        case (port_num)
            2'b00 : P0 = pdcnt;
            2'b01 : P1 = pdcnt;
            2'b10 : P2 = pdcnt;
            2'b11 : P3 = pdcnt;
        endcase
    end
endmodule
```

Fig 11. Verilog description of the demultiplexer

4. *Seven Segment Display*

Seven Segment display module is used for displaying the counter output on the seven segments of the FPGA board. Seven-segment module receives a 4-bit input and displays the HEX value on its 7-bit output which is corresponded to the number of remaining data bits to be transmitted.

```
module SSD(input [3:0]count, output reg [6:0]pdcnt);
    always@(count)begin
        case (count)
            4'h0: pdcnt = 7'h40;
            4'h1: pdcnt = 7'h79;
            4'h2: pdcnt = 7'h24;
            4'h3: pdcnt = 7'h30;
            4'h4: pdcnt = 7'h19;
            4'h5: pdcnt = 7'h12;
            4'h6: pdcnt = 7'h02;
            4'h7: pdcnt = 7'h78;
            4'h8: pdcnt = 7'h00;
            4'h9: pdcnt = 7'h10;
            4'ha: pdcnt = 7'h08;
            4'hb: pdcnt = 7'h03;
            4'hc: pdcnt = 7'h46;
            4'hd: pdcnt = 7'h21;
            4'he: pdcnt = 7'h06;
            4'hf: pdcnt = 7'h0e;
            default:pdcnt = 7'h00;
        endcase
    end

endmodule
```

Fig 12. Verilog description of the Seven Segment Display module

```
module MSSD(input clk, rst, clkPB, serIn, output done, SerOutValid, output [6:0]P0, P1, P2, P3, output serOut);

    wire clkEn, co1, co2, coD, cnt1, cnt2, cntD, ldcntD, sh_en, sh_enD;
    wire [3:0] NumData, count;
    wire [1:0] PortNum;
    wire [6:0] pdcnt;

    Onepulser One_Pulser(clk, rst, clkPB, clkEn);
    PortNum_shr shr_port(clk, rst, clkEn, serIn, sh_en, PortNum);
    port_cnt counter1(clk, rst, clkEn, cnt1, co1);
    DataNum_shr shr_data( clk, rst, clkEn, serIn, sh_enD, NumData);
    Datanum_cnt counter2(clk, rst, clkEn, cnt2, co2);
    DataTrans_cnt counter3(clk, rst, clkEn, cntD, ldcntD, NumData, count, coD);
    Demux demux(pdcnt, PortNum, P0, P1, P2, P3);
    SSD Seven_Seg(count, pdcnt);

    Controller CU(clk, rst, serIn, clkEn, co1, co2, coD, cnt1, cnt2, cntD, ldcntD, sh_en, sh_enD, SerOutValid, done);

    assign serOut = serIn;

endmodule
```

Fig 13. Verilog description of the top level module MSSD

We have Add the top-level Verilog codes to your project then Connect the main FPGA clock to the clock input of your circuit after that Connect a switch key to the serIn of the serial transmitter circuit, another push-button to the input of the one-pulser circuit. We have synthesised the design and Programed the Cyclone II device.

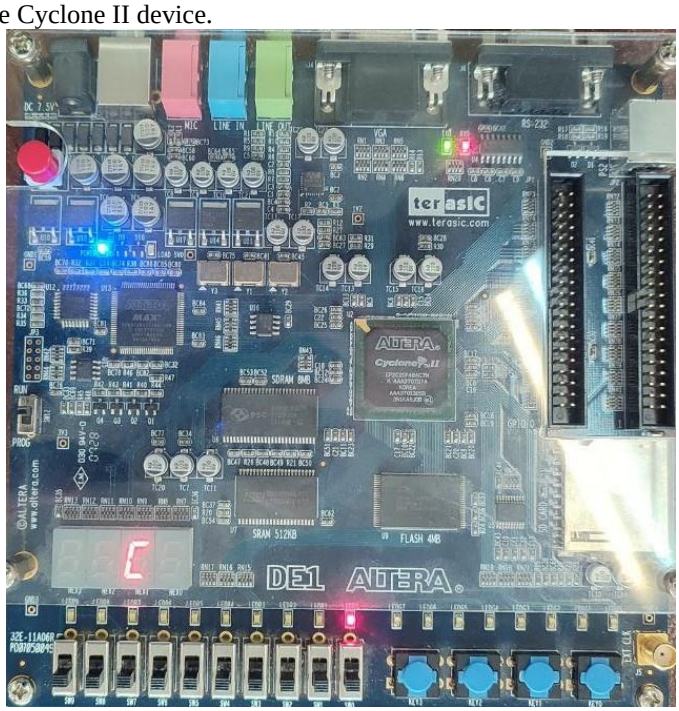

Fig 14. FPGA implementation

IV. MSSD IMPLEMENTATION

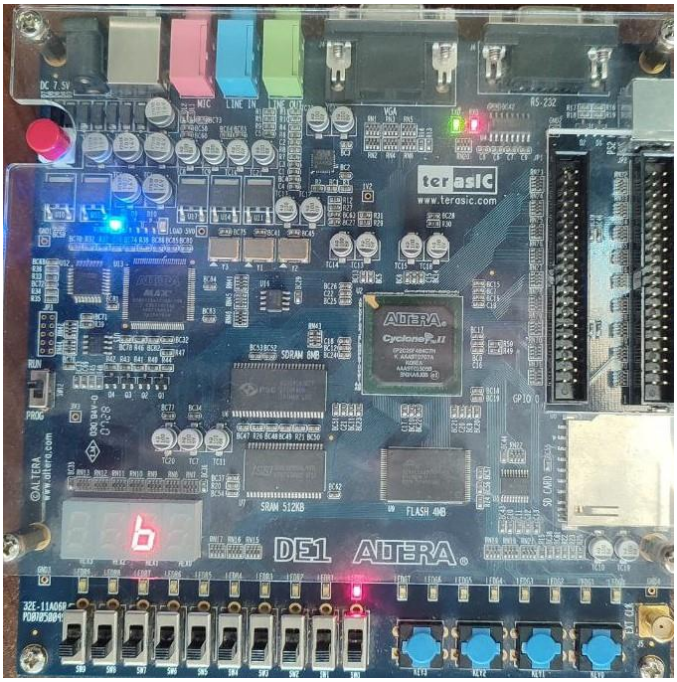

Fig 15. FPGA implementation

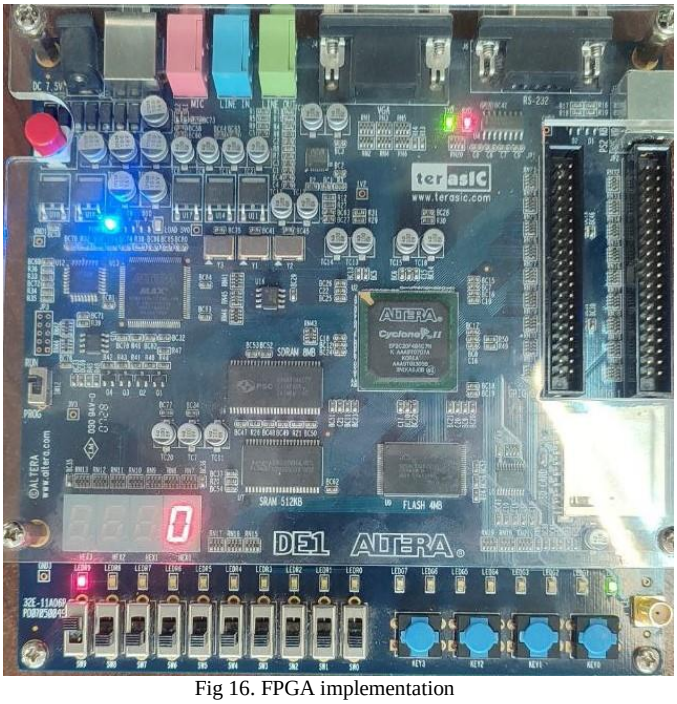

Fig 16. FPGA implementation

As we can see in the figure 14, we have selected the port number 2 (01 in binary) and the number of data bits is 12 (1100 in binary). Thus the second seven-segment module show the remaining bits of output to be transmitted.
The figure 16 demonstrate the end of the transmission and we can see the green LED (which is done signal) is on.