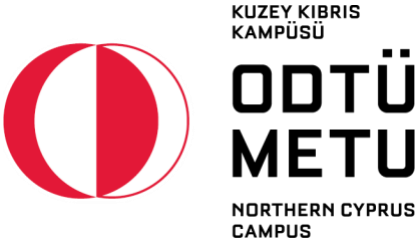


Lab 7 Project Report

Name: Mert Can Bilgin

Student ID: 2453025

Date: 13/06/2022

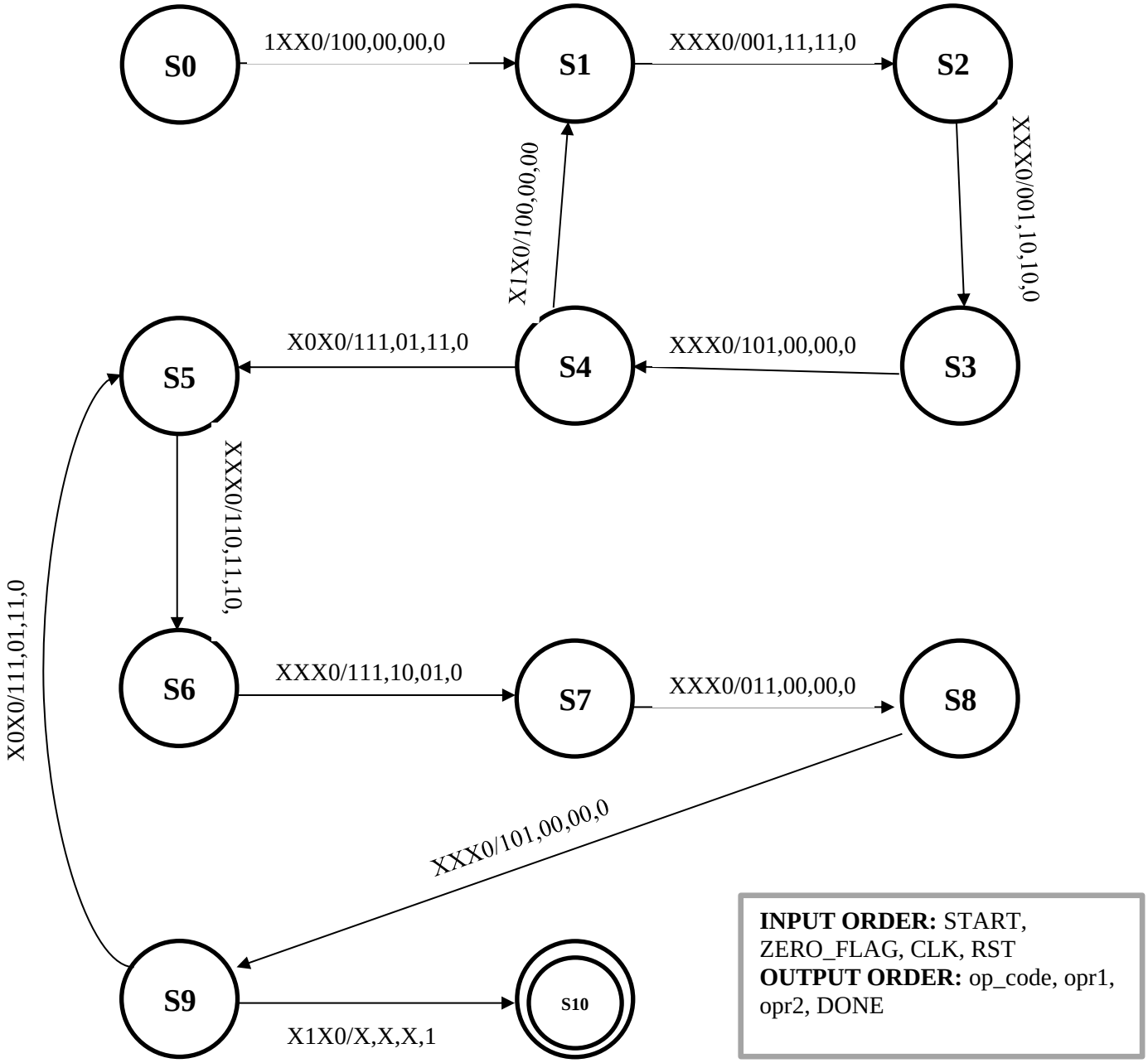
Objective

In this project, 4-bit Fibonacci Series Calculator is designed. It will be designed part by part:

1. Datapath
2. FSM Decoder
3. FSM
4. Fibonacci FSM
5. Fibonacci Series Calculator

Also, designing a Mealy FSM will be learned.

STATE DIAGRAM OF THE FIBO_FSM



PARAMETERIZED VERILOG CODE

```
module FIBO_DATAPATH(wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode, count, zero_flag, data);
output [3:0] data;
output zero_flag;
input [3:0] count;
input [2:0] alu_opcode;
input [1:0] wrt_addr, rd_addr1, rd_addr2;
input wrt_en, clk, load_data;
wire [3:0] deco;//output of the decoder
wire [3:0] andgates; //output of and gates
wire [3:0] MUXL0, MUXL1, MUXL2, MUXL3; //output of 2x1 muxs
wire [3:0] DFFL0, DFFL1, DFFL2, DFFL3; //output of dffs
wire [3:0] Rxx, Ryy;
wire [3:0] outputofALU;
wire [3:0] bottomMux;
wire [3:0] outputofBottomDFF;
twoXfourDecoder U1(wrt_addr, deco);
//and gates
and A1(andgates[3], deco[3], wrt_en);
and A2(andgates[2], deco[2], wrt_en);
and A3(andgates[1], deco[1], wrt_en);
and A4(andgates[0], deco[0], wrt_en);
//top 2x1 muxs
twoXoneMux M3(MUXL3, bottomMux, DFFL3, andgates[3]);
twoXoneMux M2(MUXL2, bottomMux, DFFL2, andgates[2]);
twoXoneMux M1(MUXL1, bottomMux, DFFL1, andgates[1]);
twoXoneMux M0(MUXL0, bottomMux, DFFL0, andgates[0]);
//Registers
fourbitRegister R0(DFFL3, MUXL3, clk);
fourbitRegister R1(DFFL2, MUXL2, clk);
fourbitRegister R2(DFFL1, MUXL1, clk);
fourbitRegister R3(DFFL0, MUXL0, clk);
//4x1 mux
fourXoneMux F1(Rxx, DFFL3, DFFL2, DFFL1, DFFL0, rd_addr1);
fourXoneMux F2 (Ryy, DFFL3, DFFL2, DFFL1, DFFL0, rd_addr2);
//ALU
ALU G0(Rxx, Ryy, alu_opcode, outputofALU, zero_flag);
//bottom 2x1 mux
//it needs to be negedge because of the delay issue
fourbitRegister R4(outputofBottomDFF, outputofALU, ~clk);
twoXoneMux M4(bottomMux, count, outputofBottomDFF, load_data);
assign data = outputofBottomDFF;
endmodule
```

Figure 1: Verilog Code of Datapath

Comments:

- I have designed this datapath in lab 6. It is a hierarchial design. I firstly implemented 2x1 multiplexer, 4x1 multiplexer, 4-bit register, 2x4 Decoder and ALU.
- There are wires which represents the outputs of multiplexers and 4-bit registers.
- I connected them according to the Figure 1 in the lab manual.

```
module FSM_DECO(op_code, opr1, opr2, ALU, rd_addr1, rd_addr2, wrt_addr, wrt_en, load_data);
parameter size = 4;
input [size - 3: 0] opr1, opr2;
input [size - 2: 0] op_code;
output reg [size - 2:0] ALU;
output reg [size - 3:0] rd_addr1, rd_addr2, wrt_addr;
output reg wrt_en, load_data;


always @(op_code, opr1, opr2)
begin
ALU = op_code; //default
rd_addr1 = 2'b11;
rd_addr2 = 2'b10;
wrt_addr = 2'b11;
wrt_en = 0;
load_data = 0;
case(op_code)
3'b000: begin rd_addr1 <= rd_addr1; rd_addr2 <= rd_addr2; wrt_addr <= wrt_addr; wrt_en = 0; load_data <= load_data; end
3'b001: begin rd_addr1 <= rd_addr1; rd_addr2 <= rd_addr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 0; end
3'b010: begin rd_addr1 <= opr1; rd_addr2 <= rd_addr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 0; end
3'b011: begin rd_addr1 <= opr1; rd_addr2 <= rd_addr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 0; end
3'b100: begin rd_addr1 <= rd_addr1; rd_addr2 <= rd_addr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 1; end
3'b101: begin rd_addr1 <= opr1; rd_addr2 <= rd_addr2; wrt_addr <= wrt_addr; wrt_en = 0; load_data <= load_data; end
3'b110: begin rd_addr1 <= opr1; rd_addr2 <= opr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 0; end
3'b111: begin rd_addr1 <= opr2; rd_addr2 <= rd_addr2; wrt_addr <= opr1; wrt_en = 1; load_data <= 0; end
endcase
end
endmodule
```

Figure 2: Verilog Code of FSM_DECO

Comments:

- Outputs of the FSM_DECO is used for inputs of the datapath. ALU opcode is directly assigned, and rd_addr1 and rd_addr2 is assigned Register1 and Register in default.
- I implemented the case part according to the table 3 in the lab manual.

```
module FSM(START, ZERO_FLAG, CLK, RST, op_code, opr1, opr2, DONE);
parameter size = 4;
input START, ZERO_FLAG, CLK, RST;
output reg [size - 2:0] op_code;
output reg [size - 3:0] opr1, opr2;
output reg DONE;
reg [size - 1:0] state, nextstate;
parameter S0 = 4'b0000, S1 = 4'b0001, S2 = 4'b0010, S3 = 4'b0011, S4 = 4'b0100, S5 = 4'b0101, S6 = 4'b0110, S7 = 4'b0111, S8 = 4'b1000, S9 = 4'b1001, S10 = 4'b1010;
always @(posedge CLK or posedge RST) //to update state
if(RST) state <= S0; else state <= nextstate;
always @(state or START or ZERO_FLAG) //compute output
begin
   op_code = 3'b000; opr1 = 2'b00; opr2 = 2'b00; DONE = 0;
   case(state)
   S0: begin if(START) op_code = 3'b100; end
   S1: begin op_code = 3'b001; opr1 = 2'b11; opr2 = 2'b11; end
   S2: begin op_code = 3'b001; opr1 = 2'b10; opr2 = 2'b10; end
   S3: op_code = 3'b101;
   S4: begin if(ZERO_FLAG) op_code = 3'b100; else begin op_code = 3'b111; opr1 = 2'b01; opr2 = 2'b11; end end
   S5: begin op_code = 3'b110; opr1 = 2'b11; opr2 = 2'b10; end
   S6: begin op_code = 3'b111; opr1 = 2'b10; opr2 = 2'b01; end
   S7: op_code = 3'b011;
   S8: op_code = 3'b101;
   S9: begin if(ZERO_FLAG) DONE = 1; else op_code = 3'b111; end
   endcase
end
always @(state or START or ZERO_FLAG) //compute next state
begin
   nextstate = S0;
   case(state)
      S0: if(START) nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S3;
      S3: nextstate = S4;
      S4: begin if(ZERO_FLAG) nextstate = S1; else nextstate = S5; end
      S5: nextstate = S6;
      S6: nextstate = S7;
      S7: nextstate = S8;
      S8: nextstate = S9;
      S9: begin if(ZERO_FLAG) nextstate = S10; else nextstate = S5; end
      endcase
end
endmodule
```

Figure 3: Verilog Code of FSM

Comments:

- I implemented the FSM according to the state diagram.
- Firstly, all the states are declared as parameters.
- In the first always block, it checks the rising RST. If RST is rising it directly goes to the state S0.
- In the second always block, it computes the output. For example, op_code is “001”, opr1 is “10” and opr2 is “10” in S2. It means it sets the Register2 to 1.
- In the third always block, it computes the next state. For instance, if ZERO_FLAG is ‘1’, next state will be S4 in S1, otherwise it will be S5.

```
module FIBO_FSM(START, ZERO_FLAG, CLK, RST, DONE, wrt_addr, wrt_en, load_data, rd_addr1, rd_addr2, alu_opcode);
parameter size = 4;

input START, ZERO_FLAG, CLK, RST;
output [size-2:0] alu_opcode;
output [size - 3:0] wrt_addr, rd_addr1, rd_addr2;
output DONE, wrt_en, load_data;

wire [size - 2:0] alu;
wire [size - 3:0] operand1, operand2;



FSM F0(START, ZERO_FLAG, CLK, RST, alu, operand1, operand2, DONE);
FSM_DECO D0(alu, operand1, operand2, alu_opcode, rd_addr1, rd_addr2, wrt_addr, wrt_en, load_data);
endmodule
```

Figure 4:Verilog Code of FIBO_FSM

Comments:

- FIBO_FSM consists of two parts which are FSM and FSM_DECO.
- Firstly, FSM is called, and according to the alu, operand1 and operand2, FSM_DECO sends signals to the datapath.

```
module TOP(START, RST, DONE, CLK, count, data);

parameter size = 4;

input START, RST, CLK;
input [size - 1:0] count;
output DONE;
output [size - 1:0] data;

wire zero_flag, wrt_en, load_data;
wire [size - 2:0] Alu;
wire [size - 3:0] wrt_addr, rd_addr1, rd_addr2;


FIBO_FSM F0(START, zero_flag, CLK, RST, DONE, wrt_addr, wrt_en, load_data, rd_addr1, rd_addr2, Alu);
FIBO_DATAPATH D0(wrt_addr, wrt_en, CLK, load_data, rd_addr1, rd_addr2, Alu, count, zero_flag, data);
endmodule

```

Figure 5: Verilog Code of Top-Level Design

Comments:

- Top-level design consists of FIBO_FSM and datapath.
- First, FSM is called, and according to its inputs datapath is called.
- FIBO_FSM decides the wrt_addr, wrt_en, load_data, rd_addr1, rd_addr2 and opcode of alu.
- Also, it has an count input. When count is ‘0’ operation will be stopped and DONE will be ‘1’.

TESTBENCH

```
module TB_FIBO_DATAPATH();
 parameter size=4;
 reg [size-1:0] count;
 reg [size-3:0] wrt_addr,rd_addr1,rd_addr2;
 reg [size-2:0] alu_opcode;
 reg clk,wrt_en,load_data;

 wire [size-1:0] data;
 wire zero_flag;
 FIBO_DATAPATH DUT (wrt_addr,wrt_en,clk,load_data,rd_addr1,rd_addr2,alu_opcode,count,zero_flag, data);
always begin

 wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b1; rd_addr1=2'b01; rd_addr2=2'b10; alu_opcode= 3'b001; count= 4'b0100; #100;///load
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b000; count= 4'b0011; #100;//setnum1
 wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b111; count= 4'b0011; #100;//set num2 rd1

 wrt_addr= 2'b01; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b110; count= 4'b0011; #100;//copy , rd1 and rd2
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b01; rd_addr2=2'b01; alu_opcode= 3'b111; count= 4'b0011; #100;//add,rd1
 wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b00; rd_addr2=2'b00; alu_opcode= 3'b011; count= 4'b0011; #100;//copy,rd1
 wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b11; alu_opcode= 3'b111; count= 4'b0011; #100;//decr
 wrt_addr= 2'b01; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b110; count= 4'b0011; #100;
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b01; rd_addr2=2'b01; alu_opcode= 3'b111; count= 4'b0011; #100;
 wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b00; rd_addr2=2'b00; alu_opcode= 3'b011; count= 4'b0011; #100;
 wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b11; alu_opcode= 3'b111; count= 4'b0011; #100;
 wrt_addr= 2'b01; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b110; count= 4'b0011; #100;
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b01; rd_addr2=2'b01; alu_opcode= 3'b111; count= 4'b0011; #100;
 wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b00; rd_addr2=2'b00; alu_opcode= 3'b011; count= 4'b0011; #100;
 wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b11; alu_opcode= 3'b111; count= 4'b0011; #100;//decr
 wrt_addr= 2'b01; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b10; alu_opcode= 3'b110; count= 4'b0011; #100;
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b01; rd_addr2=2'b01; alu_opcode= 3'b111; count= 4'b0011; #100;
 wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b00; rd_addr2=2'b00; alu_opcode= 3'b011; count= 4'b0011; #100;
 wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b11; alu_opcode= 3'b111; count= 4'b0011; #100;//decr
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b01; alu_opcode= 3'b101; count= 4'b0011; #100;
 wrt_addr= 2'b11; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b01; alu_opcode= 3'b101; count= 4'b0011; #100;

 end

always begin
 clk=0; #50;
 clk=1; #50;
end
endmodule
```

Figure 6: Testbench of Datapath

Comments:

- In the testbench, firstly it loads the count value to Register 4, then it sets Register1 and Register2 as 1. After that, it copies the Register1 value to Register3, and sum of the Register1 and Register2 is assigned to Register1. Then it decrements the value of Register4 because it holds the value of count.
- All operations are performed according to the algorithm in lab manual.

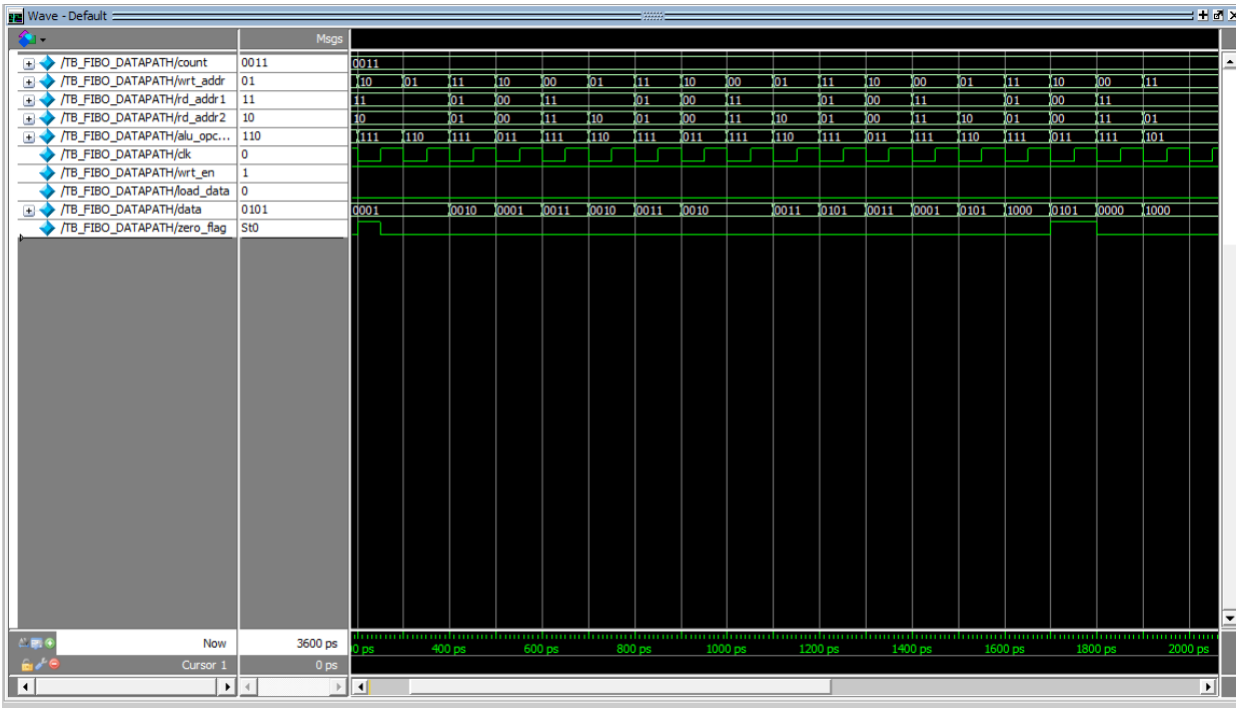


Figure 7: Simulation Results of Datapath

Comments:

- As we can see the fibonacci numbers can be seen at 200ps, 700ps, 1100ps, 1600ps and 1900ps which are 1-2-3-5-8.
- Therefore, the results are correct.

```
1  module TB_FSM_DECO();
2  parameter size = 4;
3  wire [size - 2: 0] ALU;
4  wire [size - 3: 0] rd_addr1, rd_addr2, wrt_addr;
5  wire wrt_en, load_data;
6  reg [size - 2: 0] op_code;
7  reg [size - 3: 0] opr1, opr2;
8
9  FSM_DECO DUT(op_code, opr1, opr2, ALU, rd_addr1, rd_addr2, wrt_addr, wrt_en, load_data);
10
11 initial
12 begin
13 op_code = 3'b000; opr1 = 2'b01; opr2 = 2'b10; #100; //noop
14 op_code = 3'b001; opr1 = 2'b01; opr2 = 2'b10; #100; //set
15 op_code = 3'b010; opr1 = 2'b01; opr2 = 2'b10; #100; //inc
16 op_code = 3'b011; opr1 = 2'b01; opr2 = 2'b10; #100; //dec
17 op_code = 3'b100; opr1 = 2'b01; opr2 = 2'b10; #100; //load
18 op_code = 3'b101; opr1 = 2'b01; opr2 = 2'b10; #100; //store
19 op_code = 3'b110; opr1 = 2'b01; opr2 = 2'b10; #100; //add
20 op_code = 3'b111; opr1 = 2'b01; opr2 = 2'b10; #100; //copy
21 end
22 endmodule
```

Figure 8: Testbench of FSM_DECO

Comments:

- All operation codes are tested, opr1 is represented as 1 and opr2 is represented as 2.

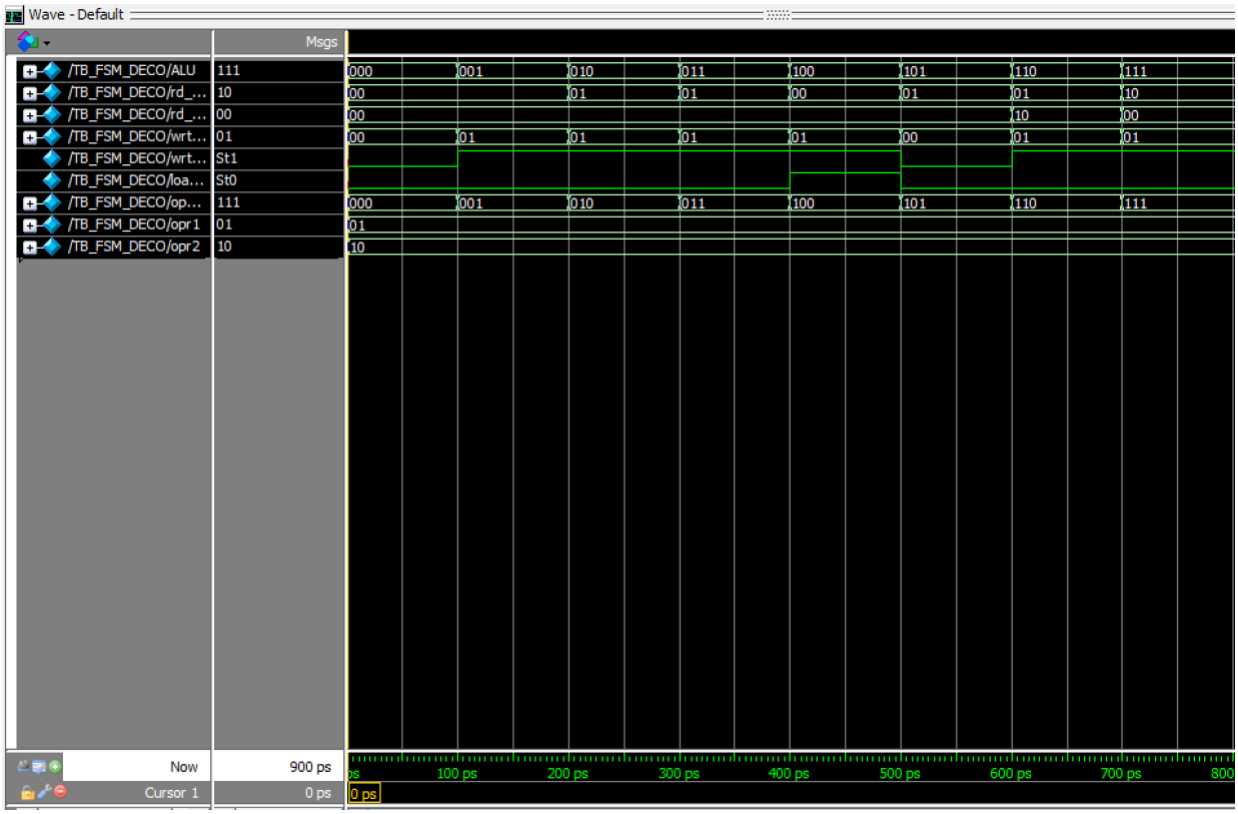


Figure 9: Simulation Results of FSM_DECO

Comments:

- All operations performed correctly. For instance, when op_code is “111”, it will perform copy operation. It means rd_addr1 will be the same as opr2. We can see that at 700ps.
- Therefore, the results are correct.

```
1  module FSM_TB();
2
3  parameter size = 4;
4
5  reg START, ZERO_FLAG, CLK, RST;
6  wire [size - 2:0] op_code;
7  wire [size - 3:0] opr1, opr2;
8  wire DONE;
9
10 FSM DUT(START, ZERO_FLAG, CLK, RST, op_code, opr1, opr2, DONE);
11 always
12 begin
13 START = 1; ZERO_FLAG = 0; RST = 0; #100;
14 START = 1; ZERO_FLAG = 0; RST = 0; #100;
15 START = 1; ZERO_FLAG = 0; RST = 0; #100;
16 START = 1; ZERO_FLAG = 0; RST = 0; #100;
17 START = 1; ZERO_FLAG = 0; RST = 0; #100;
18 START = 1; ZERO_FLAG = 0; RST = 0; #100;
19 START = 1; ZERO_FLAG = 0; RST = 0; #100;
20 START = 1; ZERO_FLAG = 0; RST = 0; #100;
21 START = 1; ZERO_FLAG = 0; RST = 0; #100;
22 START = 1; ZERO_FLAG = 0; RST = 0; #100;
23 START = 1; ZERO_FLAG = 0; RST = 0; #100;
24 START = 1; ZERO_FLAG = 0; RST = 0; #100;
25 START = 1; ZERO_FLAG = 0; RST = 0; #100;
26 START = 1; ZERO_FLAG = 0; RST = 0; #100;
27 START = 1; ZERO_FLAG = 0; RST = 0; #100;
28 START = 1; ZERO_FLAG = 0; RST = 0; #100;
29 START = 1; ZERO_FLAG = 1; RST = 0; #100;
30 START = 1; ZERO_FLAG = 1; RST = 0; #100;
31 START = 1; ZERO_FLAG = 1; RST = 0; #100;
32 START = 1; ZERO_FLAG = 1; RST = 0; #100;
33 START = 1; ZERO_FLAG = 1; RST = 0; #100;
34 end
35
36 always begin
37   CLK=0; #50;
38   CLK=1; #50;
39 end
40 endmodule
```

Figure 10: Testbench of FSM

Comments:

- I wrote the testbench according to the state diagram, and I will check my results in the simulation part. If the op_code, opr1 and op2 is correct then the results are correct.

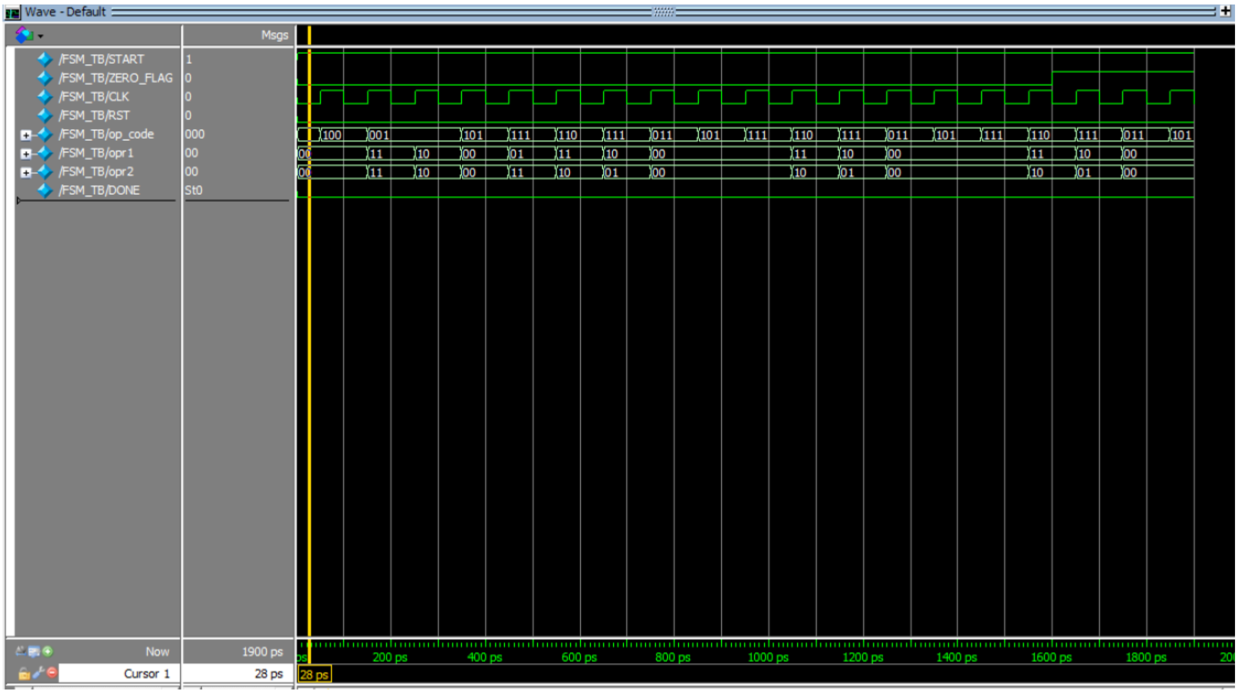


Figure 11: Simulation Results of FSM

Comments:

- The results are as expected. For example, in the state diagram the opcodes goes like that 100->001->001->101->111->110->111->011->101, and I can see the same results in the simulation.
- The opr1 is 00->11->10->… in the state diagram, and it is the same in the simulation.
- The opr2 is the same as in the state diagram.
- Therefore, results are correct.

```
module FIBO_FSM_TB();

parameter size = 4;

reg START, ZERO_FLAG, CLK, RST;
wire [size-2:0] alu_opcode;
wire [size - 3:0] wrt_addr, rd_addr1, rd_addr2;
wire DONE, wrt_en, load_data;

FIBO_FSM DUT(START, ZERO_FLAG, CLK, RST, DONE, wrt_addr, wrt_en, load_data, rd_addr1, rd_addr2, alu_opcode);

always
begin
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 0; RST = 0; #100;
START = 1; ZERO_FLAG = 1; RST = 0; #100;
START = 1; ZERO_FLAG = 1; RST = 0; #100;
START = 1; ZERO_FLAG = 1; RST = 0; #100;
START = 1; ZERO_FLAG = 1; RST = 0; #100;
START = 1; ZERO_FLAG = 1; RST = 0; #100;
end

always begin
 CLK=0; #50;
 CLK=1; #50;
end
endmodule
```

Figure 12: Testbench FIBO_FSM

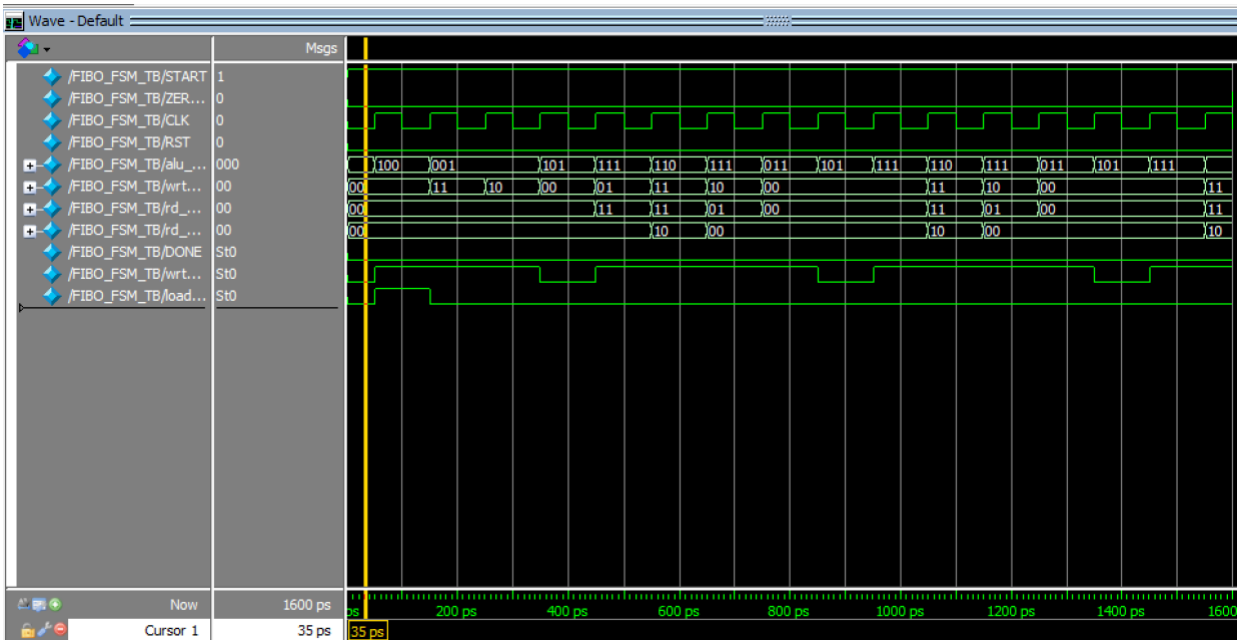


Figure 13: Simulation Results of FIBO_FSM

Comments:

- The state diagram, and algorithm will be cheched in this part.
- The first step is loading count, so opcode of load is “100”. The wrt_addr should be “00” to activate Register4.
- The second step is setting the Register1 and Register2 as ‘1’, so the opcode should be “001”, and the wrt_addr should be “11” then “10”.
- The third step checking if the count is ‘0’, so opcode should be “101” and the rd_addr1 should be “00”.
- The fourth step is Register1’s value is copied to Register3, so opcode should be “111”, and the wrt_addr should be “01”.
- The fifth step is setting Register1 as sum of Register1 and Register2, so opcode should be “110”, and wrt_addr should be “11”.
- The sixth step is setting Register2 as Register3, so opcode should be “111”, and the wrt_addr should be “10”.
- When we check all these conditions, we can see the same results in simulation.
- Therefore, the results are correct.

```
1  module TOP_TB();
2
3  parameter size = 4;
4
5  reg START, RST, CLK;
6  reg [size - 1:0] count;
7  wire [size - 1:0] data;
8  wire DONE;
9
10 TOP DUT(START, RST, DONE, CLK, count, data);
11
12 always
13 begin
14 START = 1; RST = 0; count = 4'b111; #100;
15 end
16
17 always
18 begin
19 CLK = 0; #50;
20 CLK = 1; #50;
21 end
22 endmodule
```

Figure 14: Testbench of Top-Level Design

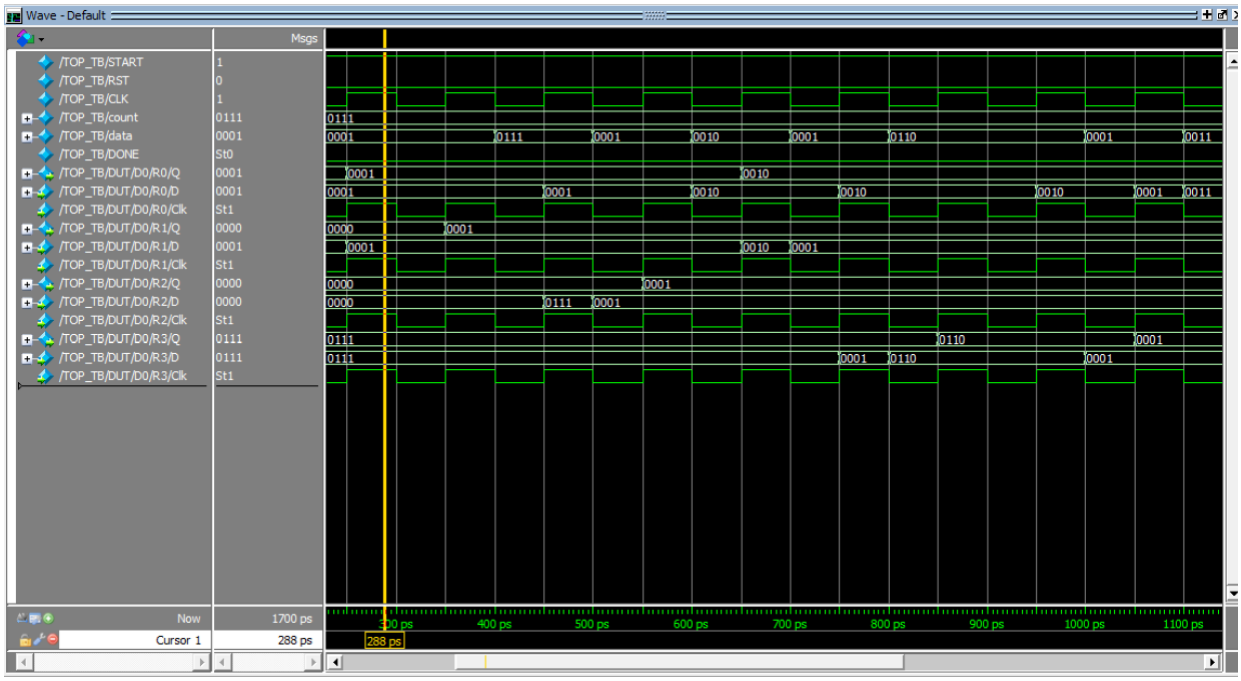


Figure 15: Simulation Results of Top-Level Design

Comments:

- When we check the Registers' values, it is the same as algorithm.
- Fibonacci series can be seen at 100ps, 600ps, 1100ps, and so on.
- fmax is 1/500ps.
- Therefore, the results are correct.

Conclusion

In this lab, I have completed a big design step by step. I have started to design from a 2x1 multiplexer, and it growed fastly. At the end of the design, I have a complex design. This lab improved my engineering and problem solving skills.