KUZEY KIBRIS KAMPÜSÜ
ODTÜ
METU
NORTHERN CYPRUS CAMPUS

Lab 6 Report

Name: Mert Can Bilgin

Student ID: 2453025

EXPERIMENTAL RESULTS

6.3 DATAPATH DESIGN

D FLIP-FLOP DESIGN

```
module d_flip_flop(Q, D, Clk);

output Q;
input D, Clk;
reg Q = 0;

always @(posedge Clk)
Q <= D;

endmodule
```

Figure 1: Verilog Code of D Flip-Flop

```
module d_flip_flop_tb();

wire Q;
reg D, Clk;

d_flip_flop DUT(Q, D, Clk);

always #50 Clk = ~Clk;

initial
begin
Clk = 0;
D = 1'b0; #100;
D = 1'b1; #100;
D = 1'b1; #100;
D = 1'b0; #100;
end
endmodule
```

Figure 2: TestBench of D Flip-Flop

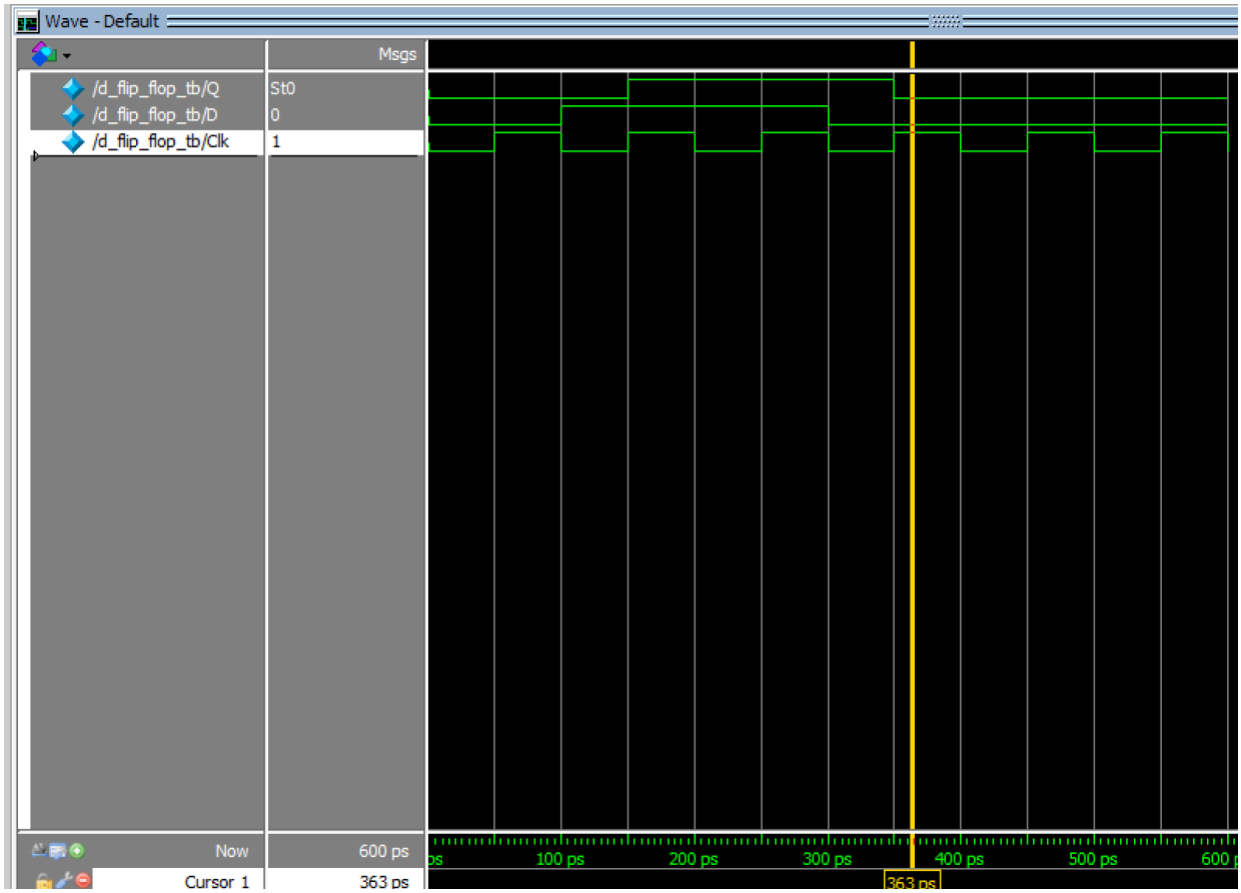

Figure 3: Simulation Results of DFF

Comments:

- I have implemented the D flip-flop by using behavioral approach.
- I wrote a testbench to be ensure that my results are correct, then I simulated it.
- For example, when clock is rising we see the output's value is the same as input value.
- Therefore, the results are correct.
- I will use this DFF in order to design 4-bit register.

4-Bit Register Design

C:/Users/mertc/Desktop/fourbitRegister/fourbitRegister.v (/fourbi

```
module fourbitRegister(Q, D, Clk);

output [3:0] Q;
input [3:0] D;
input Clk;

d_flip_flop f3(Q[3], D[3], Clk);
d_flip_flop f2(Q[2], D[2], Clk);
d_flip_flop f1(Q[1], D[1], Clk);
d_flip_flop f0(Q[0], D[0], Clk);
endmodule
```

Figure 4: Verilog Code of 4-bit Register

C:/Users/mertc/Desktop/fourbitRegister/fourbitRegister_tb

```
module fourbitRegister_tb();
wire [3:0] Q;
reg [3:0] D;
reg Clk;

fourbitRegister DUT(Q, D, Clk);

always #50 Clk = ~Clk;

initial
begin
Clk = 0;
D = 4'b0000; #100;
D = 4'b0001; #100;
D = 4'b0010; #100;
D = 4'b0011; #100;
D = 4'b0100; #100;
D = 4'b0101; #100;
D = 4'b0110; #100;
D = 4'b0111; #100;
D = 4'b1000; #100;
D = 4'b1001; #100;
D = 4'b1010; #100;
D = 4'b1011; #100;
D = 4'b1100; #100;
D = 4'b1101; #100;
D = 4'b1110; #100;
D = 4'b1111; #100;
end
endmodule
```

Figure 5: TestBench of 4-bit Register

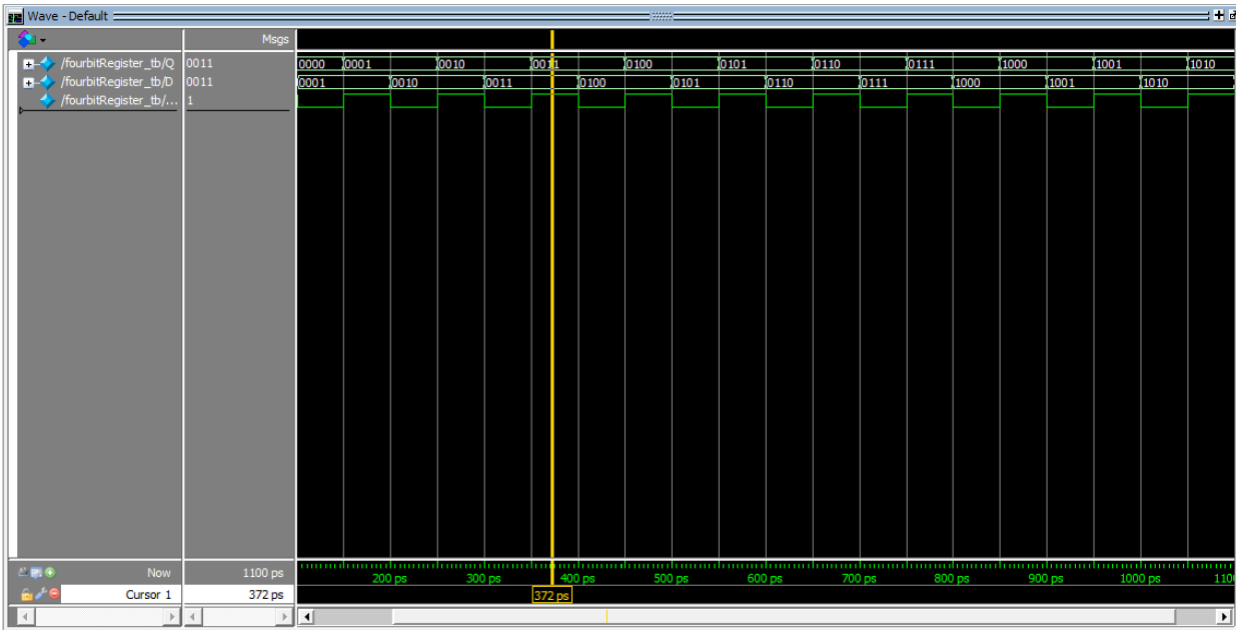

Figure 6: Simulation Results of 4-bit Register

Comments:

- I have used four DFF to design a 4-bit register.
- I wrote a testbench to be ensure that my results are correct, then I simulated it.
- For example, when clock is rising from ‘0’ to ‘1’, the output value is updated according to input value.
- Therefore, the results are correct.
- I will use five 4-bit register in my top-level design. These register will help to design a fibonacci series calculator.

2x1 Multiplexer Design

C:/Users/mertc/Desktop/twoXoneMux/twoXoneMux.v (/tw

```
module twoXoneMux(F, A, B, S);

output [3:0] F;
input [3:0] A, B;
input S;

assign F = (S) ? B:A;
endmodule
```

Figure 7: Verilog Code of 2x1 Multiplexer

C:/Users/mertc/Desktop/twoXoneMux/twoXoneMux_tb.v (/twoXoneMux_tb) - D

```
module twoXoneMux_tb();
wire [3:0] F;
reg [3:0] A, B;
reg S;

twoXoneMux DUT(F, A, B, S);

initial
begin
A = 4'b0000; B = 4'b0001; S = 1'b0; #100;
A = 4'b0000; B = 4'b0001; S = 1'b1; #100;
end
endmodule
```

Figure 8: TestBench of 2x1 Multiplexer

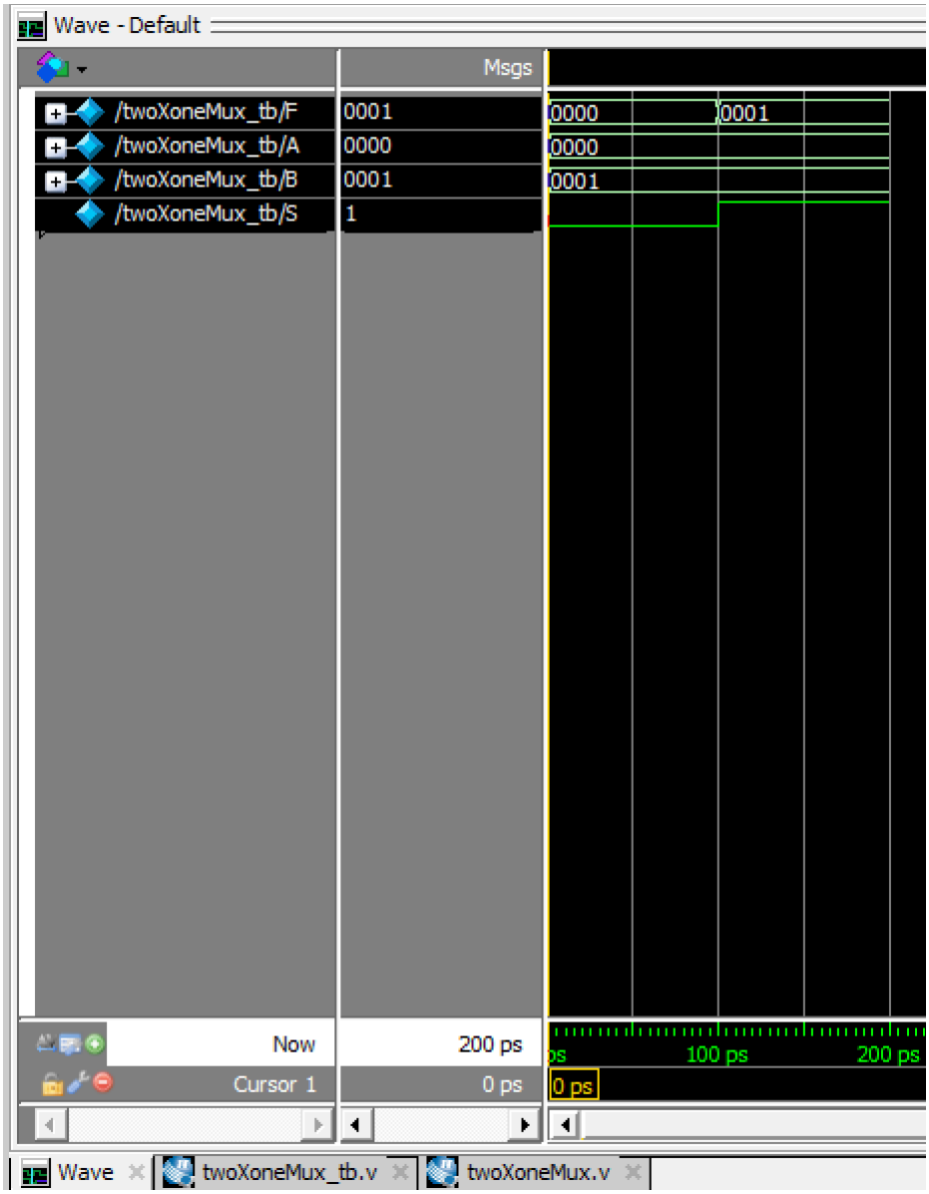

Figure 9: Simulation Results of 2x1 Mux

Comments:

- I have designed a 2x1 multiplexer by using behavioral approach.
- I have written a testbench to be ensure that my results are correct.
- As we can see in the Figure 9, when S is 0, it selects A, otherwise it selects B.
- Therefore, my results are correct.
- I will use five 2x1 mux in the top-level design. Four of them will decide whether the registers' value is updated or not. One of them will decide whether the data is stored or the new data(count) is stored in registers.

2-to-4 Line Decoder Design

C:/Users/mertc/Desktop/twoXfourDecoder/twoXfourDecoder.v (/twoXfourDec

```
module twoXfourDecoder(wrt_addr, F);
input [1:0] wrt_addr;
output [3:0] F;

reg [3:0] F = 4'b0000;

always @(wrt_addr)
begin
if(wrt_addr == 2'b00) F <= 4'b0001;
else if(wrt_addr == 2'b01) F <= 4'b0010;
else if(wrt_addr == 2'b10) F <= 4'b0100;
else if(wrt_addr == 2'b11) F <= 4'b1000;
end
endmodule
```

Figure 10:Verilog Code of 2-to-4 Line Decoder

C:/Users/mertc/Desktop/twoXfourDecoder/twoXfourDecoder_tb.v (/

```
module twoXfourDecoder_tb();
reg [1:0] wrt_addr;
wire [3:0] F;

twoXfourDecoder DUT(wrt_addr, F);

initial
begin
  wrt_addr = 2'b00; #100;
  wrt_addr = 2'b01; #100;
  wrt_addr = 2'b10; #100;
  wrt_addr = 2'b11; #100;
end
endmodule
```

Figure 11: TestBench of 2-to-4 Line Decoder

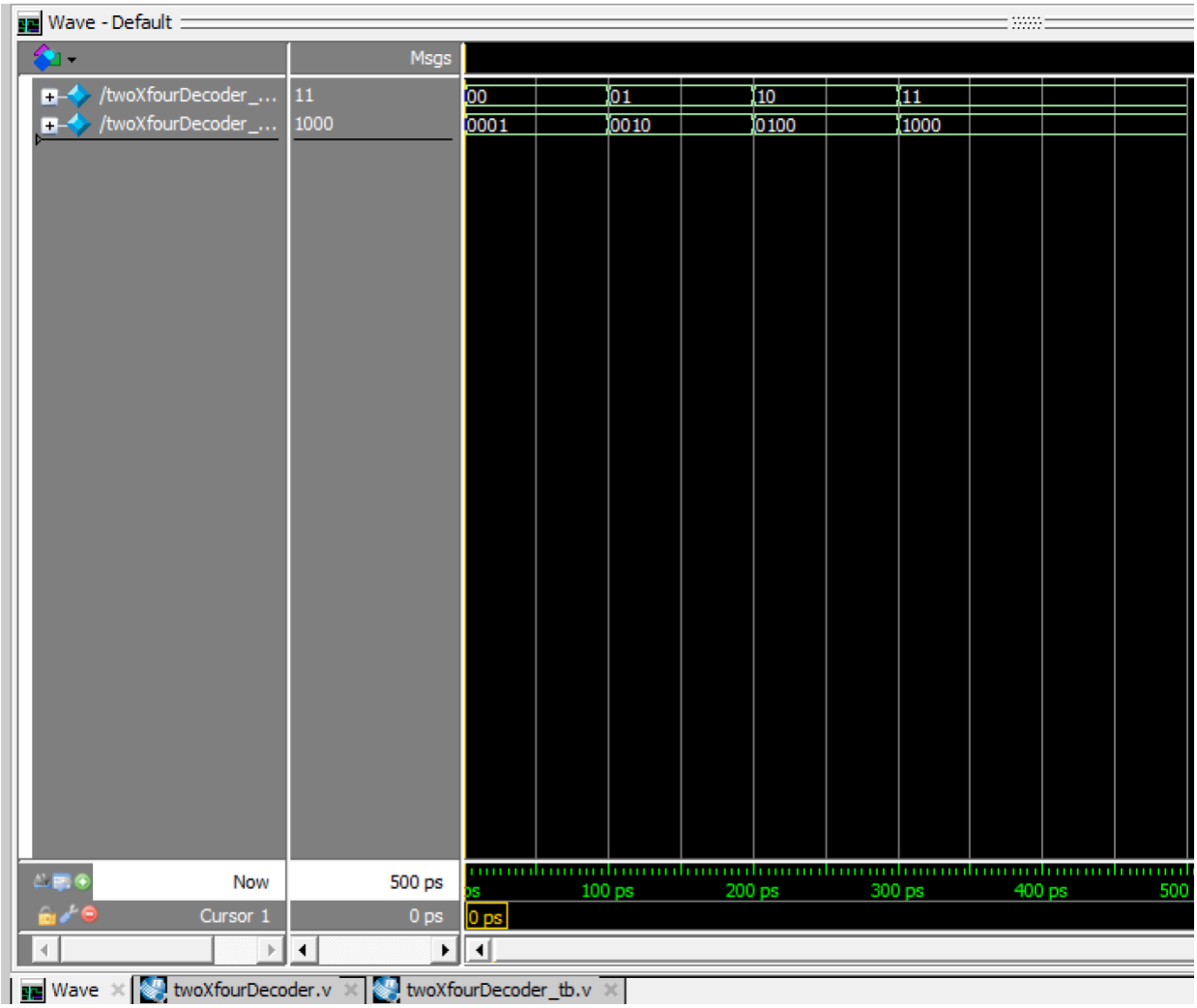

Figure 12: Simulation Results of 2-to-4 Line Decoder

Comments:

- I have implemented the decoder by using behavioral approach.
- I have writtent a testbench to be ensure that my results are correct, then I simulated it.
- According to the manual, 00->0001, 01->0010, 10->0100, and 11->1000 the inputs and outputs have to be like that. Hence, we can see the same thing in the simulation results, so it is correct.
- I will connect output of the decoder with wrt_en to selection of 2x1 multiplexers. By doing that, registers can be updated correctly.

4x1 Multiplexer Design

C:/Users/mertc/Desktop/fourXoneMux/fourXoneMux.v (/fourXoneMux_

```
module fourXoneMux(F, A, B, C, D, S);

output reg [3:0] F;
input [3:0] A, B, C, D;
input [1:0] S;

always @(A, B, C, D, S)
begin
case(S)
2'b00: F = A;
2'b01: F = B;
2'b10: F = C;
2'b11: F = D;
endcase
end
endmodule
```

Figure 13: Verilog Code of 4x1 Multiplexer

C:/Users/mertc/Desktop/fourXoneMux/fourXoneMux_tb.v (/fourXoneMux_tb) - Default

```
module fourXoneMux_tb();

wire [3:0] F;
reg [3:0] A, B, C, D;
reg [1:0] S;

fourXoneMux DUT(F, A, B, C, D, S);

initial
begin
  A = 4'b0000; B = 4'b0001; C = 4'b0010; D = 4'b0011; S = 2'b00; #100;
  A = 4'b0000; B = 4'b0001; C = 4'b0010; D = 4'b0011; S = 2'b01; #100;
  A = 4'b0000; B = 4'b0001; C = 4'b0010; D = 4'b0011; S = 2'b10; #100;
  A = 4'b0000; B = 4'b0001; C = 4'b0010; D = 4'b0011; S = 2'b11; #100;
end
endmodule
```

Figure 14: TestBench of 4x1 Multiplexer

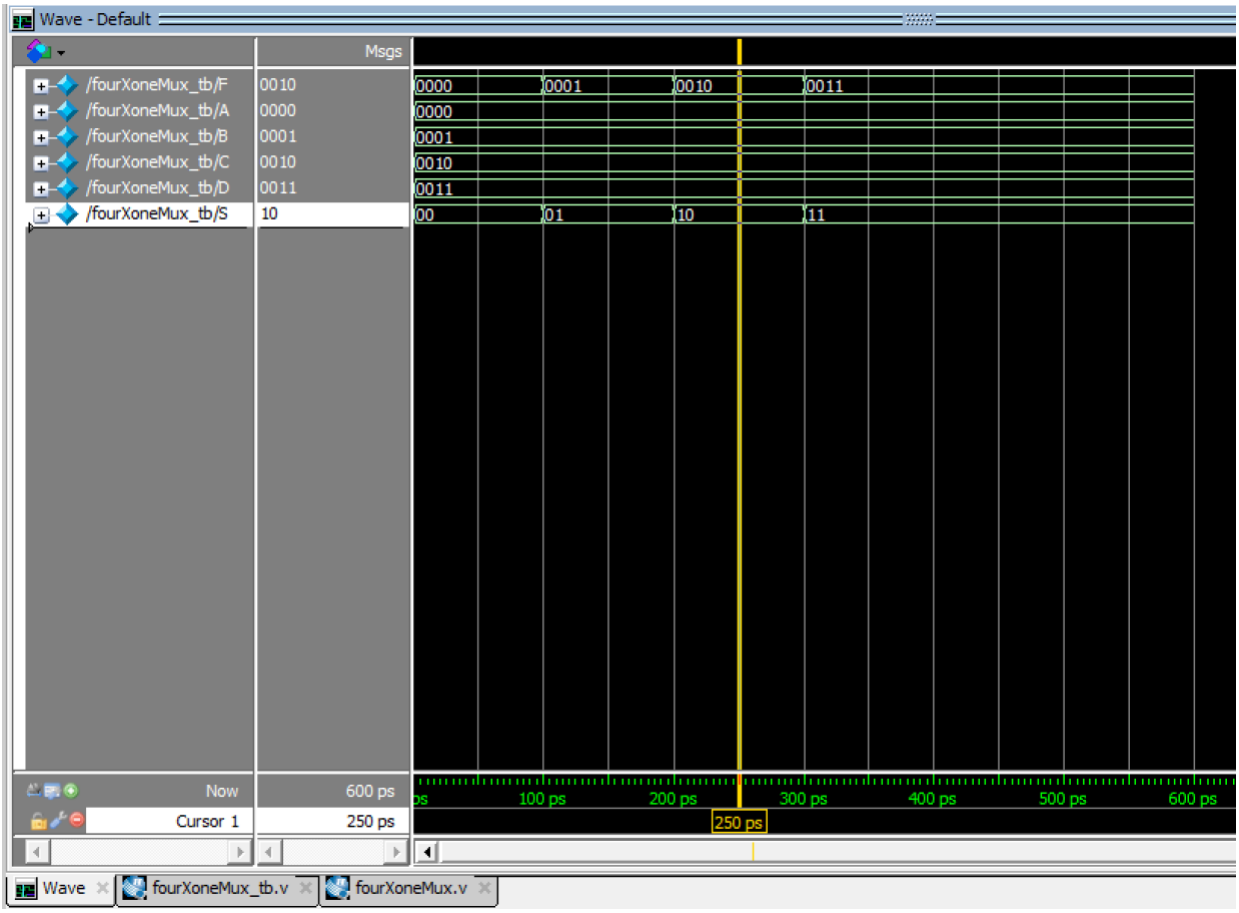

Figure 15: Simulation Results of 4x1 Multiplexer

Comments:

- I have implemented the 4x1 multiplexer by using behavioral approach.
- I have written a testbench to be ensure that my results are correct, then I simulated it.
- As we can see in the Figure 15, when S is 00, it selects A,; when S is 01, it selects B; when S = 10, it selects C; when S = 11, it selects D. Therefore, the results are correct.
- I will use two 4x1 multiplexer in my top-level design. They will decide the inputs of ALU, according to the signal coming from rd_addr1 and rd_addr2. Also, their inputs will be the output of 4-bit registers.

ALU Design

C:/Users/mertc/Desktop/ALU/ALU.v (/ALU_tb/DUT) - Default

```
module ALU(Rxx, Ryy, opcode, data, zero_flag);

output [3:0] data;
output zero_flag;

input [3:0] Rxx, Ryy;
input [2:0] opcode;

reg [3:0] data;
reg zero_flag;

always@(*)
begin
case(opcode)
3'b000: data <= 4'b0000; //DON'T CARE
3'b001: data <= 4'b0001; //set Rxx to 1
3'b010: data <= Rxx + 1'b1; //Rxx + 1
3'b011: data <= Rxx - 1'b1; //Rxx - 1
3'b100: data <= Rxx; //it doesn't do anything, mux will choose 4-bit count
3'b101: data <= Rxx; //it just pass through
3'b110: data <= Rxx + Ryy;//add
3'b111: data <= Ryy;//copy
endcase
end

always@(data)
begin
if(data == 4'b0000) zero_flag <= 1;
else zero_flag <= 0;
end

endmodule
```

Figure 16: Verilog Code of ALU

```
C:/Users/mertc/Desktop/ALU/ALU_tb.v (/ALU_tb) - Default
Ln#
1   module ALU_tb();
2
3   wire [3:0] data;
4   wire zero_flag;
5
6   reg [3:0] Rxx, Ryy;
7   reg [2:0] opcode;
8
9   ALU DUT(Rxx, Ryy, opcode, data, zero_flag);
10
11  always
12  begin
13    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b000; #50;
14    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b001; #50;
15    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b010; #50;
16    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b011; #50;
17    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b100; #50;
18    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b101; #50;
19    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b110; #50;
20    Rxx = 4'b0100; Ryy = 4'b0010; opcode = 3'b111; #50;
21  end
22  endmodule
```

Figure 17: TestBench of ALU

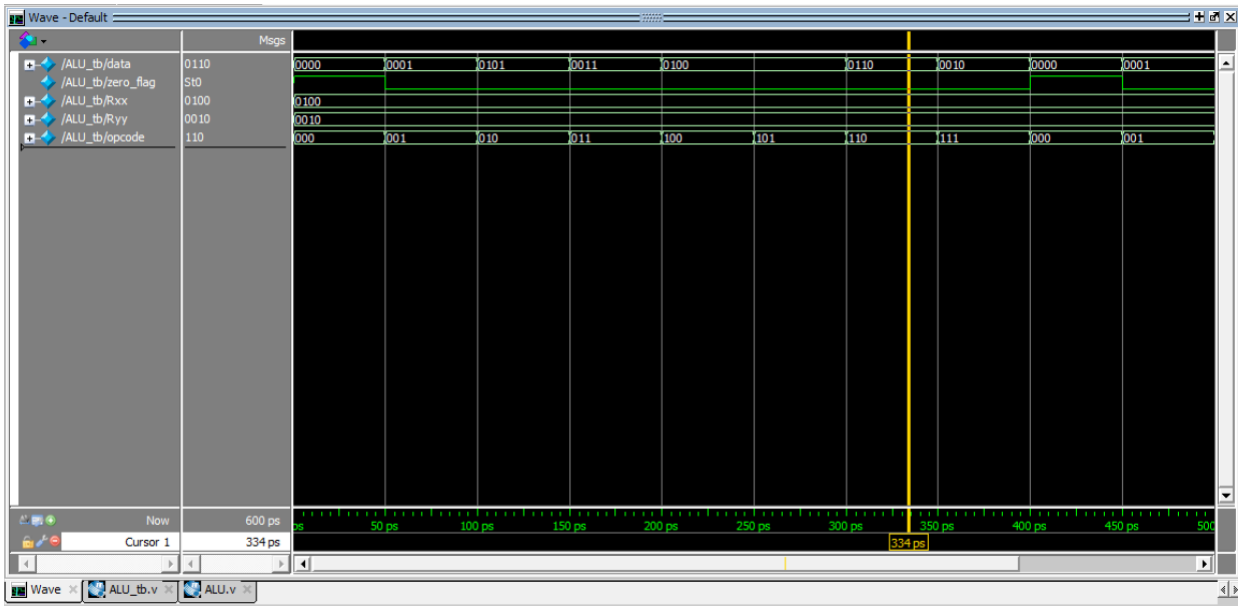

Figure 18: Simulation Results of ALU

Comments:

- I have implemented the ALU by using behavioral approach.
- I have written a testbench to be ensure that my results are correct, then I simulated it.
- ALU performs the operations according to opcode input, the instructions is specified in Table 2 in the manual.
- For example, between 300ps and 400ps the opcode is 110, so it performs addition of Rxx and Ryy. Rxx is 0100 and Ryy is 0010, and the sum is 0110. Moreover, we can see an output zero_flag. It is '1' only when the output of ALU is 0000 as we can see it in 0ps and 400ps.
- Therefore, my results are correct.
- I connected the output of ALU to negedge 4-bit register to avoid delay.

TOP-LEVEL DESIGN (FIBO_DATAPATH)

```
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 19: Verilog Code of Top-Level Design

```
module TB_FIBO_DATAPATH();

  reg [3:0] count;
  reg [1:0] wrt_addr,rd_addr1,rd_addr2;
  reg [2:0] alu_op;
  reg clk,wrt_en,load_data;

  wire [3:0] data;
  wire zero_flag;
  FIBO_DATAPATH DUT (wrt_addr,wrt_en,clk,load_data,rd_addr1,rd_addr2,alu_op,count,zero_flag,data);

  always begin

    wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b1; rd_addr1=2'b01; rd_addr2=2'b10; alu_op= 3'b010; count= 4'b1101; #100;
    wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b1; rd_addr1=2'b10; rd_addr2=2'b10; alu_op= 3'b011; count= 4'b0011; #100;
    wrt_addr= 2'b11; wrt_en=1'b0; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b00; alu_op= 3'b110; count= 4'b1001; #100;
    wrt_addr= 2'b00; wrt_en=1'b1; load_data= 1'b1; rd_addr1=2'b10; rd_addr2=2'b11; alu_op= 3'b010; count= 4'b1010; #100;
    wrt_addr= 2'b01; wrt_en=1'b0; load_data= 1'b0; rd_addr1=2'b00; rd_addr2=2'b00; alu_op= 3'b111; count= 4'b1010; #100;
    wrt_addr= 2'b01; wrt_en=1'b1; load_data= 1'b1; rd_addr1=2'b10; rd_addr2=2'b01; alu_op= 3'b101; count= 4'b0100; #100;
    wrt_addr= 2'b00; wrt_en=1'b0; load_data= 1'b1; rd_addr1=2'b00; rd_addr2=2'b10; alu_op= 3'b111; count= 4'b1001; #100;
    wrt_addr= 2'b10; wrt_en=1'b1; load_data= 1'b0; rd_addr1=2'b01; rd_addr2=2'b11; alu_op= 3'b010; count= 4'b1000; #100;
    wrt_addr= 2'b11; wrt_en=1'b0; load_data= 1'b0; rd_addr1=2'b11; rd_addr2=2'b01; alu_op= 3'b011; count= 4'b0101; #100;

  end

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

Figure 20: TestBench of Top-Level Design

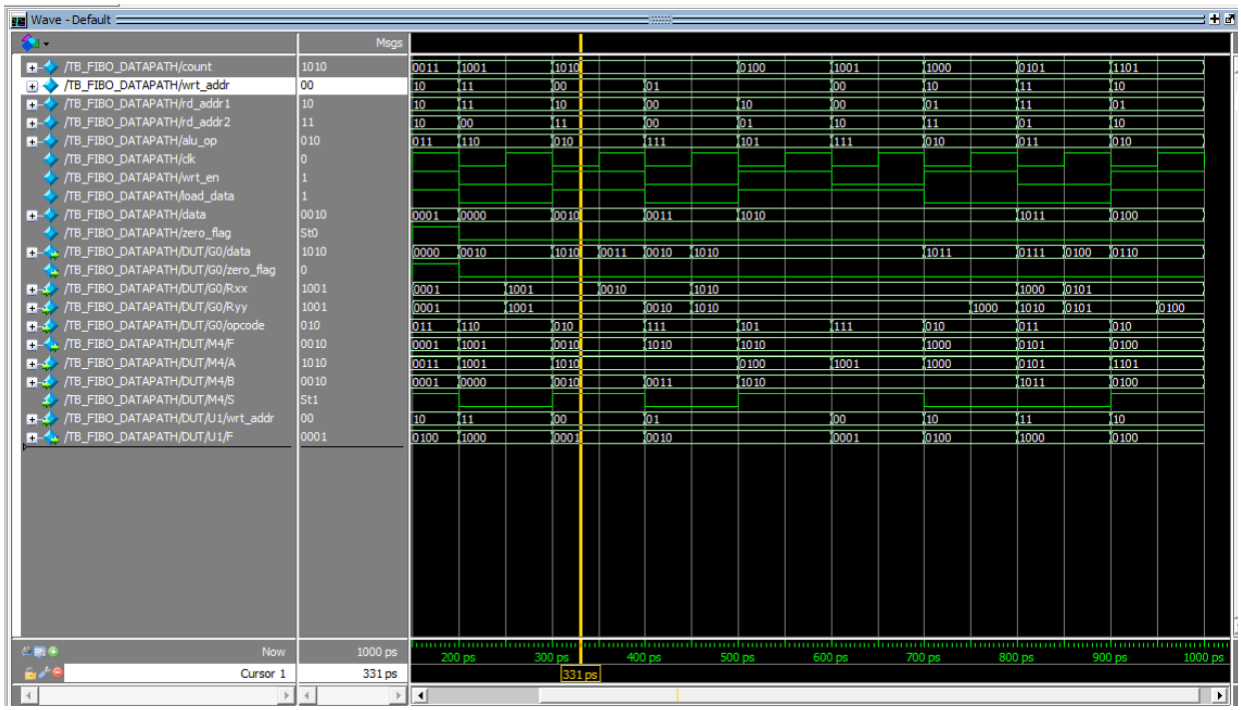

Figure 21: Simulation Results of Top-Level Design

Comments:

- I have implemented the top-level design by using hierarchical design approach.
- I have written a testbench to be ensure that my results are correct, then I simulated it.
- There are eight inputs in the top-level design which are wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode and count. Also, there is two output which are data and zero_flag.
- wrt_addr works with wrt_en, they decide which register's value is updated.
- If load_data is '1', register will be updated according to the value of count input.
- According to rd_add1 and rd_addr2, 4x1 multiplexers send their outputs to ALU.
- alu_opcode decides which operation is performed in ALU.
- The data is updated according to negedges, but the registers are updated according to posedge.
- For instance, we can trace the simulation result between 300ps and 400ps. wrt_addr is 00, so output of the decoder will be 0001, in 350ps the wrt_en is 1, so the register R1 is updated at that time. R2, R3, R4 will take the value of count in 250ps because load_data is '1'. Now, we need to look at the outputs of 4x1 multiplexers. rd_addr1 is 00, and rd_add2 is 11, so Rxx will be R1 and Ryy will be R4. In this situation, R1 and R2 are 1001 because load_data is '1'. The opcode is "010", so the incrementation will be performed. "1001" + "0001" is "1010", as we can see in the Figure 21.
- Therefore, the results are correct.
- I will use this datapath to design a FSM in lab 7.