Lab6 Report

Talal Shafei

2542371

Data Path

FIBO_DATA_PATH Verilog

```
module
FIBO_DATAPATH(wrt_adder,wrt_en,Clk,load_data,rd_addr1,rd_addr2,alu_opcode,count,data_out,zero_flag);


parameter size = 4;

input wrt_en, Clk, load_data;
input [1:0]wrt_adder, rd_addr1,rd_addr2;
input [size-1:0] count;
input [2:0]alu_opcode;

output zero_flag;
output [size-1:0]data_out;


wire [size-1:0]decod;
wire [size-1:0]a;
wire [size-1:0]d0,d1,d2,d3,d4;
wire [size-1:0]Q0,Q1,Q2,Q3;
wire [size-1:0]w,A,B;



two_to_four_line_decoder D(wrt_adder,decod);

and an0(a[0],decod[0],wrt_en);
and an1(a[1],decod[1],wrt_en);
and an2(a[2],decod[2],wrt_en);
and an3(a[3],decod[3],wrt_en);

two_to_one_mux4bit mux20(Q0,w,a[0],d0);
two_to_one_mux4bit mux21(Q1,w,a[1],d1);
two_to_one_mux4bit mux22(Q2,w,a[2],d2);
two_to_one_mux4bit mux23(Q3,w,a[3],d3);


four_bit_reg REG0(d0,Clk,Q0);
four_bit_reg REG1(d1,Clk,Q1);
four_bit_reg REG2(d2,Clk,Q2);
four_bit_reg REG3(d3,Clk,Q3);


four_to_one_mux4bit mux40(Q0,Q1,Q2,Q3,rd_addr1,A);
four_to_one_mux4bit mux41(Q0,Q1,Q2,Q3,rd_addr2,B);

ALU A_unit(alu_opcode,A,B,d4,zero_flag);

wire negClk;
not n(negClk,Clk);
four_bit_reg REG4(d4,negClk,data_out);
```

```
two_to_one_mux4bit mux24(data_out,count,load_data,w);




endmodule
```

comments : the code is representation to the figure in the manual plus the inverter to make a negative edge clock for the register after the ALU

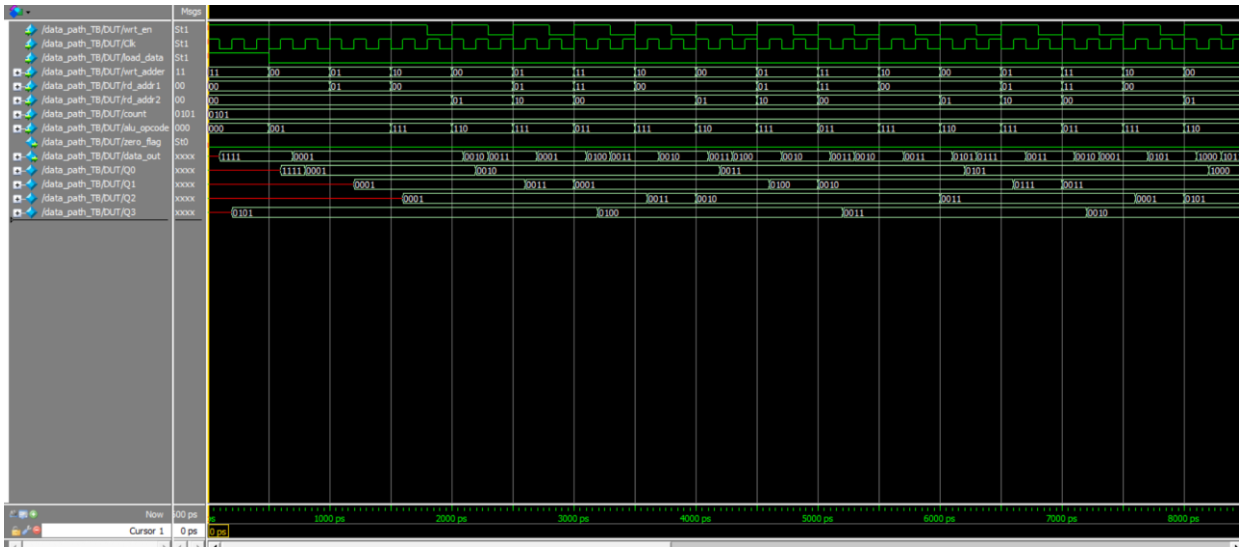
Test Bench

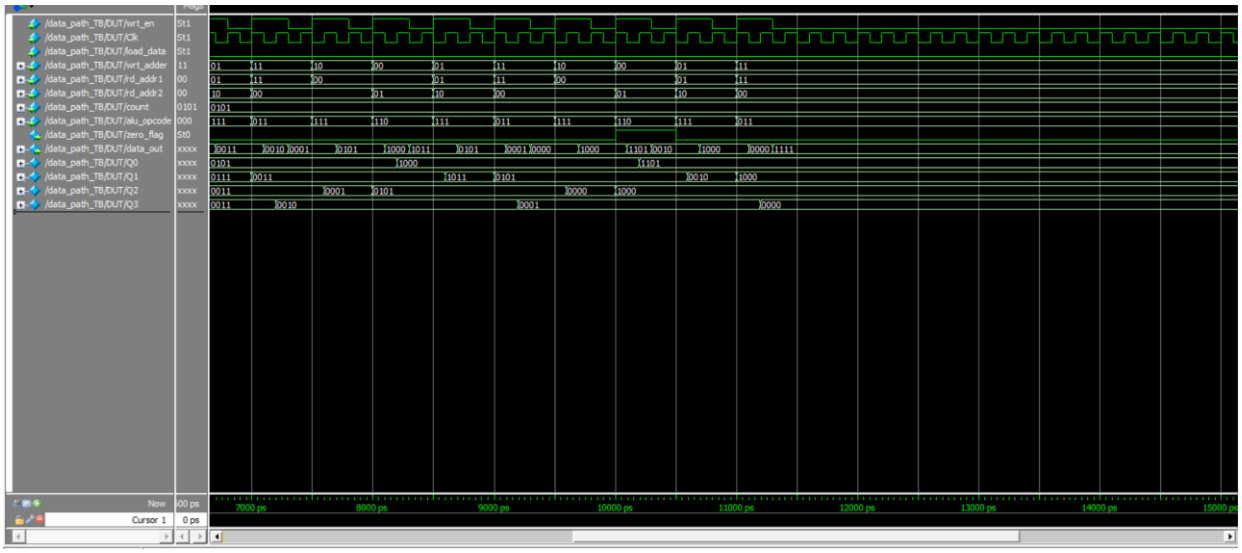
```
module data_path_TB();

reg wrt_en, Clk, load_data;
reg [1:0]wrt_adder, rd_addr1,rd_addr2;
reg [3:0] count;
reg [2:0]alu_opcode;

wire zero_flag;
wire [3:0]data_out;

FIBO_DATAPATH
DUT(wrt_adder,wrt_en,Clk,load_data,rd_addr1,rd_addr2,alu_opcode,count,data_out,zero_flag);

always #100 Clk = ~Clk;
// we are going to evaluate F(7) = 13 where count =5 (because the machine starts calculating after index 2)to test the data_path

integer i;
initial begin
Clk = 1;
count = 5;
// load count to R3 (R4 in manual)
                wrt_adder =3 ; wrt_en =1 ; load_data =1 ; rd_addr1 =0 ; rd_addr2 =0 ; alu_opcode= 3'b000;#500;

                // set R0 to num1 (R1 in the manual)
                wrt_adder =0 ; wrt_en =1 ; load_data =0 ; rd_addr1 =0 ; rd_addr2 =0 ; alu_opcode= 3'b001;#500;

                // set R1 to num1 (R2 in the manual)
                wrt_adder =1 ; wrt_en =1 ; load_data =0 ; rd_addr1 =1 ; rd_addr2 =0 ; alu_opcode= 3'b001;#500;




                        for(i=0;i<5;i=i+1)begin
                                // assign R2 = R0 (R2 is R3 manual)
                                wrt_adder =2 ; wrt_en =1 ; load_data =0 ; rd_addr1 =0 ; rd_addr2 =0 ; alu_opcode= 3'b111;#300;wrt_en=0;#200;

                                // assign R0 = R0 + R1
                                wrt_adder =0 ; wrt_en =1 ; load_data =0 ; rd_addr1 =0 ; rd_addr2 =1 ; alu_opcode= 3'b110;#300;wrt_en=0;#200;
```

```
                                    // assign R1 = R2
                                    wrt_adder =1 ; wrt_en =1 ; load_data =0 ; rd_addr1 =1 ; rd_addr2 =2 ;
alu_opcode= 3'b111;#300;wrt_en=0;#200;

                                    // decreament count that was stored in R3
                                    wrt_adder =3 ; wrt_en =1 ; load_data =0 ; rd_addr1 =3 ; rd_addr2 =0 ;
alu_opcode= 3'b011; #300;wrt_en=0;#200;
                                end




end

endmodule
```

comments: Test bench to test the code above by calculating the first 7 sequences in the Fibonacci series
note: count must be 5 because we are calculating starting from index 2
also the result should be stored in R0 after the execution of the code

Simulation





Comments: we can see that fist thing we aloading the count so Q3 become 0011 then we are setting Q0 and Q1 to 1 and then we start the algorithm in the manual to calculate The Fibonacci Series

We can see that Q0 is changing from 1 to 2 to 3 to 5 to 8 to 13
And 13 is the 7^{th} sequence

Also we can see that Q3 (the count is decreasing until it become 0)

Also the zero flag become 1 (before sending the ALU output to Q3) once the output from the ALU was zero that indicates that count is now zero

Data path parts

ALU Verilog

```
module ALU(alu_opcode,A,B,O,zero_flag);
parameter size = 4;

input [size-1:0]A;
input [size-1:0]B;
input [2:0]alu_opcode;

output reg [size-1:0]O;
output reg zero_flag;



always@(alu_opcode)begin

	zero_flag = 0;

	case(alu_opcode)
		3'b001: assign O = 1;     // set
		3'b010: assign O = A + 1; // increment
		3'b011: assign O = A - 1; // Decrement
		3'b101: assign O = A;     // Load
		3'b110: assign O = A + B; // add
		3'b111: assign O = B;     // copy
		default : O = 4'b1111;   // for 000 and 100 the AlU won't do
			// anything and we don't want to put it to 0 because
                                                                        // we don't want to activate the
zero_flag
	endcase

	if (O == 0)
		zero_flag = 1;

end

endmodule
```

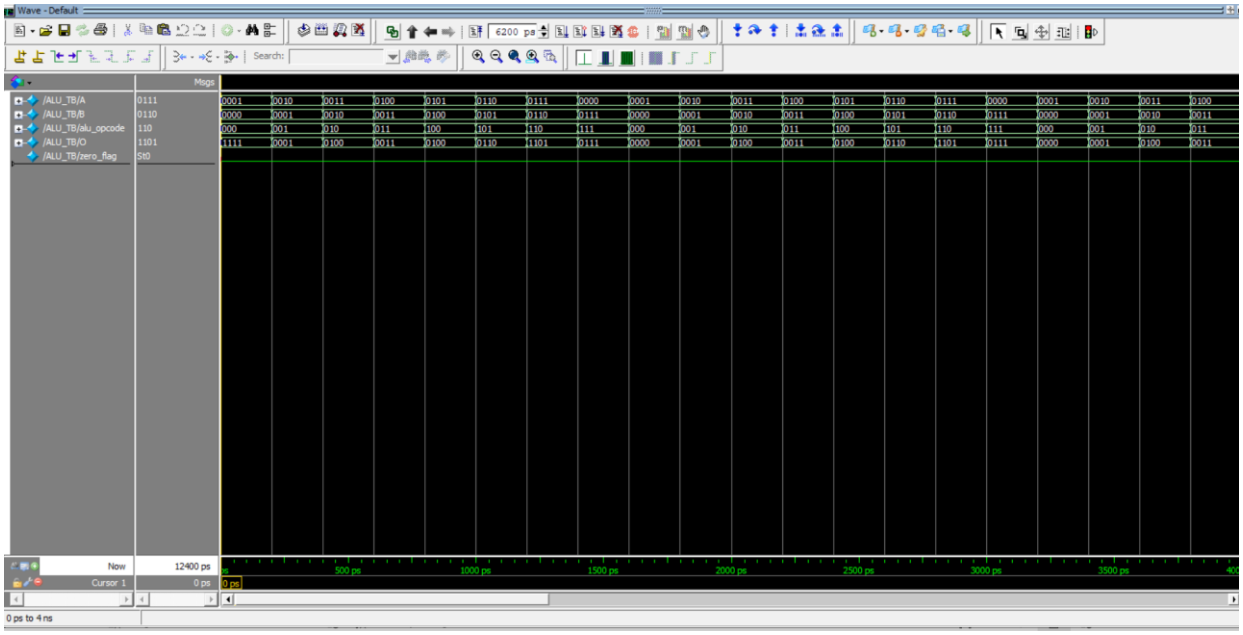
comments: Behavioral code as allowed in the manual the alu_opcode decide the operation of the block

Test Bench

```
module ALU_TB();

reg [3:0]A;
reg [3:0]B;
reg [2:0]alu_opcode;

wire [3:0]O;
wire zero_flag;

ALU DUT(alu_opcode,A,B,O,zero_flag);


always begin
alu_opcode = 3'b000; A= 4'b0001;B=4'b0000; #200;
alu_opcode = 3'b001; A= 4'b0010;B=4'b0001; #200;
alu_opcode = 3'b010; A= 4'b0011;B=4'b0010; #200;
alu_opcode = 3'b011; A= 4'b0100;B=4'b0011; #200;
alu_opcode = 3'b100; A= 4'b0101;B=4'b0100; #200;
alu_opcode = 3'b101; A= 4'b0110;B=4'b0101; #200;
alu_opcode = 3'b110; A= 4'b0111;B=4'b0110; #200;
alu_opcode = 3'b111; A= 4'b0000;B=4'b0111; #200;



end

endmodule
```

comments: Test Bench to test the code above using all the alu_opcode possible

Simulation



Comments: when the alu_opcode is 000 the output is 1111 to avoid making it 0000 so we don't activate the zero_flag
When the alu_opcode is 001 we are setting output to 1
When the alu_opcode is 010 we are setting output to A +1 (increament) so 0011+1 = 0100
When the alu_opcode is 011 we are setting output to A -1 (decreament) so 0100-1 = 0011
And so on according to the table in the manual

D-Flip Flop Verilog

```
module DFF(input D, input Clk, output reg Q);

always @ (posedge Clk)
        Q<=D;



endmodule
```

comments: code for simple D-FF

4-bit Register using D-FF's

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

parameter size = 4;

input [size - 1:0] D;
input Clk;

output [size-1:0]Q;

genvar i;
```

```
generate
        for(i=0;i<size;i=i+1)begin:top
                DFF dff(D[i],Clk,Q[i]);
        end

endgenerate

endmodule
```

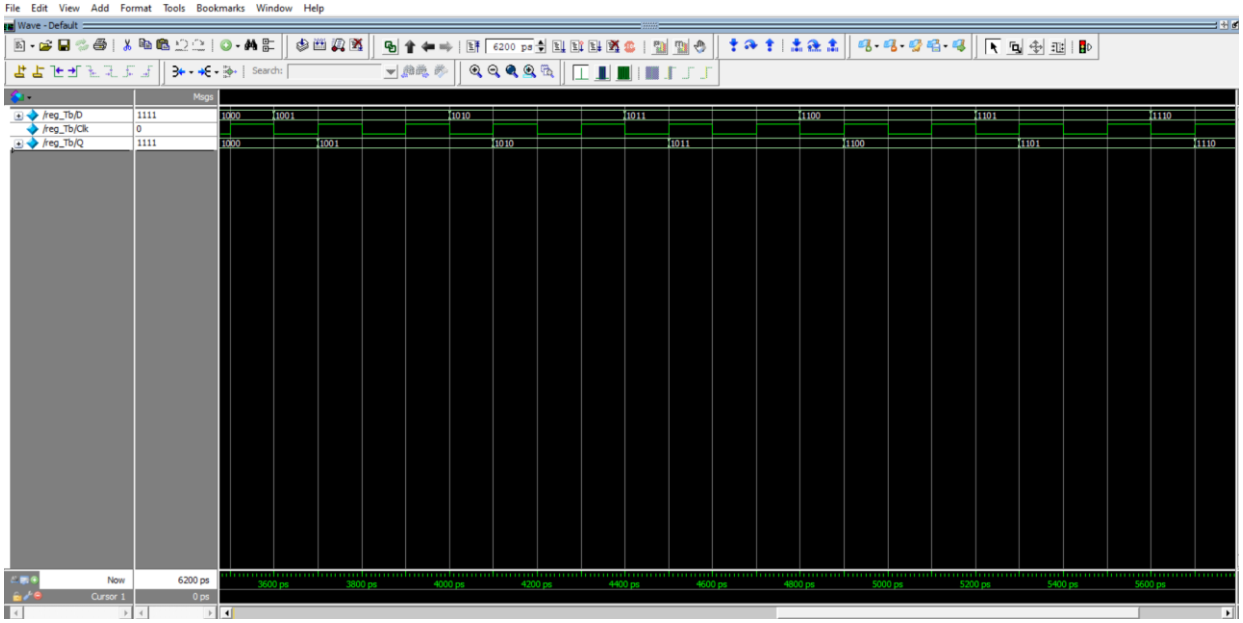
comments: simple register that loads the 4-bit data
note: parametrized so if we want to make the calculator calculate large Fibonacci's sequence we can adjust the size of the register

Test Bench

```
module reg_Tb();

reg [3:0] D;
reg Clk;

wire [3:0]Q;

four_bit_reg DUT(D,Clk,Q);

always #100 Clk= ~Clk;

initial begin
Clk=0;
D =0;#400;
D=1;#400;
D=2;#400;
D=3;#400;
D=4;#400;
D=5;#400;
D=6;#400;
D=7;#400;
D=8;#400;
D=9;#400;
D=10;#400;
D=11;#400;
D=12;#400;
D=13;#400;
D=14;#400;
D=15;#400;


end
endmodule
```

comments: testing the register on all the inputs that might take

Simulation



Comments: assigning Q=D when Clk is at positive edge

Two_to_four_line_decoder Verilog

```
module two_to_four_line_decoder(Y,D);


input [1: 0] Y;
output reg[3:0] D;


always@(Y)begin
D=0;
D[Y]=1;

end

endmodule
```

comments: simple behavioral code assigning for all D bits 0 and only for Y^{th} bit assigning 1
note no need to parametrized here since even if the calculator can store more bits it will still have 4 registers so we always need two_to_four line decoder

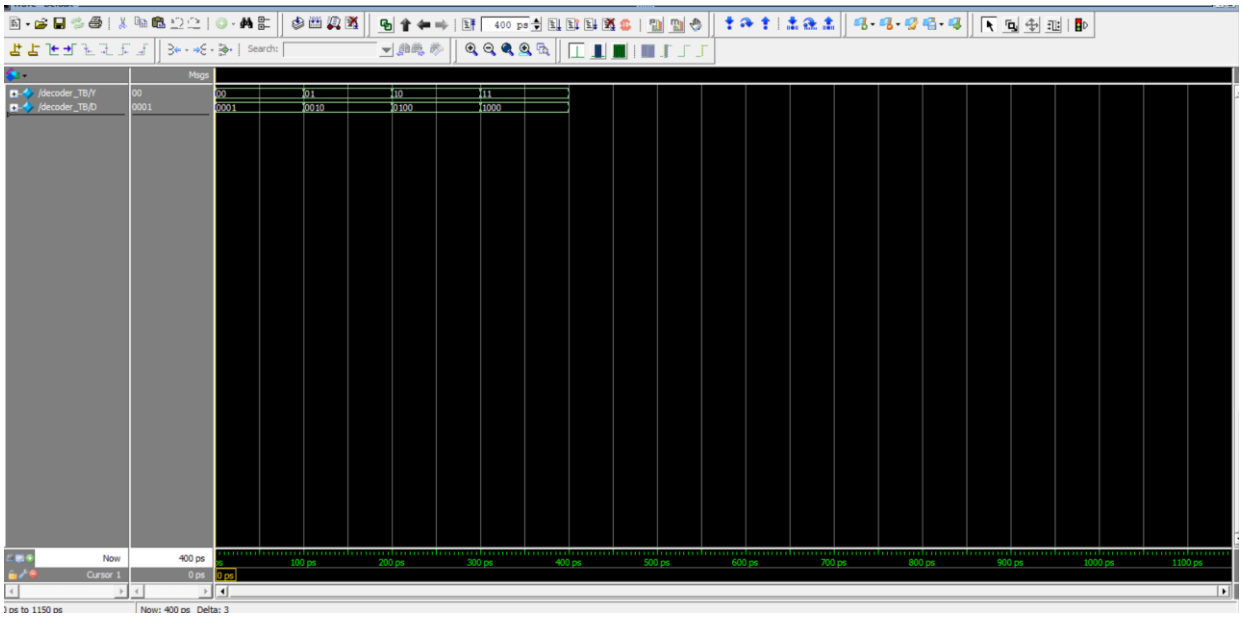
Test Bench

```
module decoder_TB ();

reg [1:0]Y;

wire [3:0]D;

two_to_four_line_decoder DUT(Y,D);

always begin

Y=0; #100;
```

```
Y=1; #100;
Y=2; #100;
Y=3; #100;

end

endmodule
```

comments: Testing all the possible outputs of the decoder

Simulation



Comments: output as expected similar to the table of a decoder

4-bit two_to_one mux Verilog

```
module two_to_one_mux4bit(A,B,S,O);

parameter size = 4;

input [size-1:0] A;
input [size-1:0]B;
input S;
output [size-1:0] O;


assign O = S ? B : A ;

endmodule
```

comments: if S is 1 assigning O=B else O=A
also the code is parametrized
No test bench since it's the same as lab3

4-bit four_to_one mux Verilog

```
module four_to_one_mux4bit(A,B,C,D,S,O);

parameter size = 4;
```

```
input [size-1:0]A;
input [size-1:0]B;
input [size-1:0]C;
input [size-1:0]D;

input [1:0]S;

output reg [size-1:0]O;

always@(S)begin

	case(S)
		2'b00: assign O = A;
		2'b01: assign O = B;
		2'b10: assign O = C;
		2'b11: assign O = D;

	endcase
end

endmodule
```

comments: will assign O to the input depends on the value of S
No test bench since it’s the same as lab3