EEE 248/ CNG 232
Logic Design

Project 4

Yıldız Alara Köymen

2453389

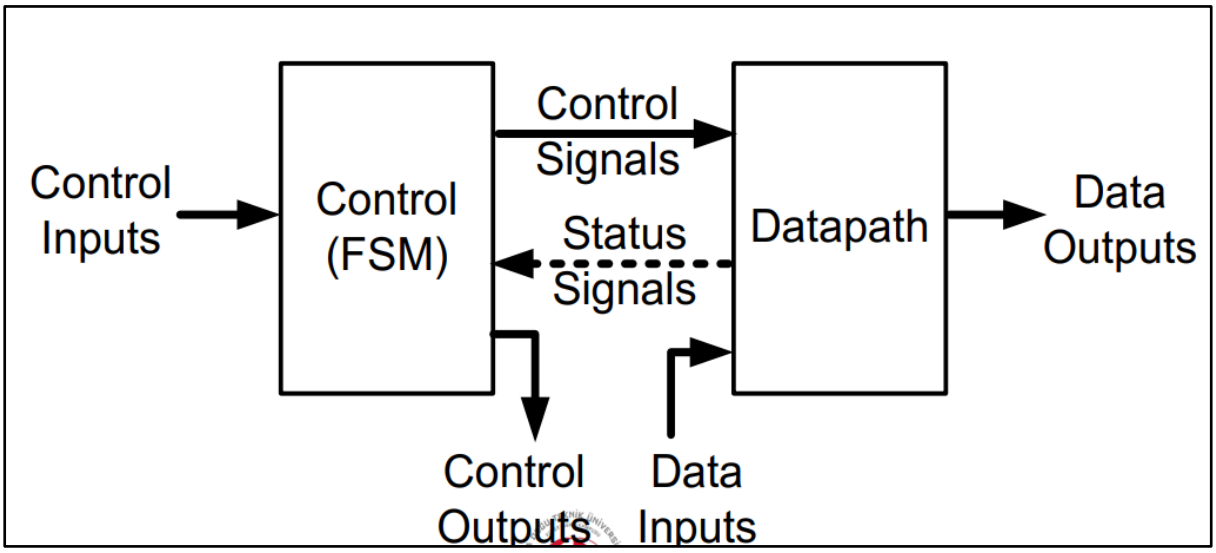
TABLE OF CONTENTS

4.1: Introduction:

We are required to design an implement a Datapath and a Controller FSM for a 4-bit Fibonacci Series Calculator. For project 4 we will focus on the Datapath design and implementation

What is a datapath?

Datapath is a collection of units that perform data processing operations. Data flows through the datapath through simple register to register movements, referred to as register transfers.



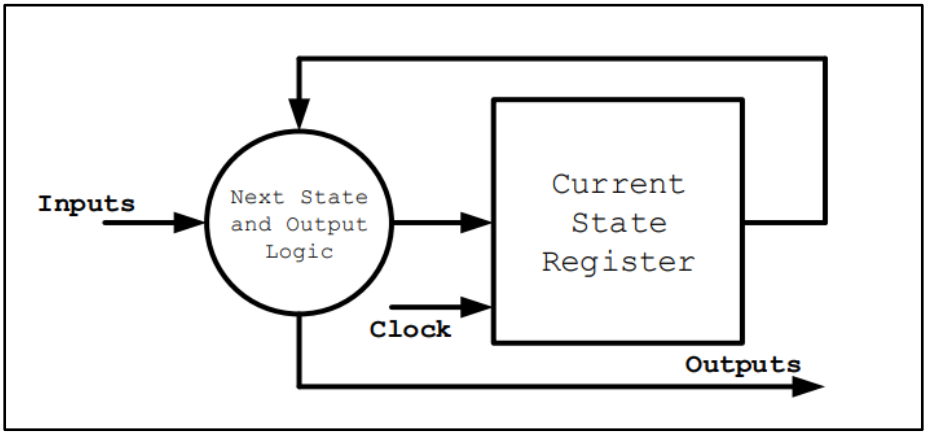
4.2 Preliminary Work:

4.2.1: Objectives:

After completing this lab, we will be able to:

- Model Mealy FSMs
- Model Moore FSMs

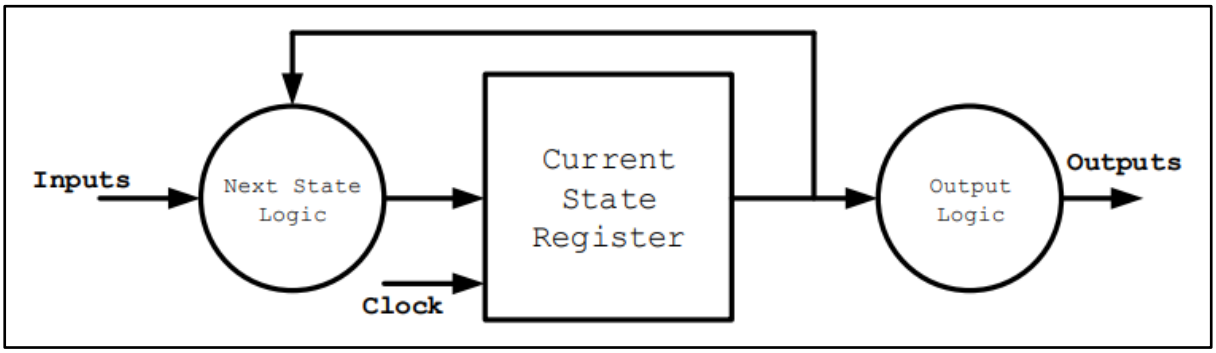
4.2.2: Mealy FSM:



In a Mealy machine, the output depends on both the present (current) state and the present (current) inputs.

4.2.3: Moore FSM:

In Moore machine, the output depends only on the present state.



4.3: Datapath Design (Week 13)

To achieve a 4-bit Fibonacci Series F(1)=1, F(2)=1 and F(N)=F(N-2)+F(N-1), the Datapath requires certain logic operations including, 2-to-4 Line decoder, 4-bit AND gate, five 4-bit 2-to-1 Multiplexers, five 4-bit registers, two 4-bit 4-to-1 Multiplexers, and 4-bit ALU as depicted in Figure

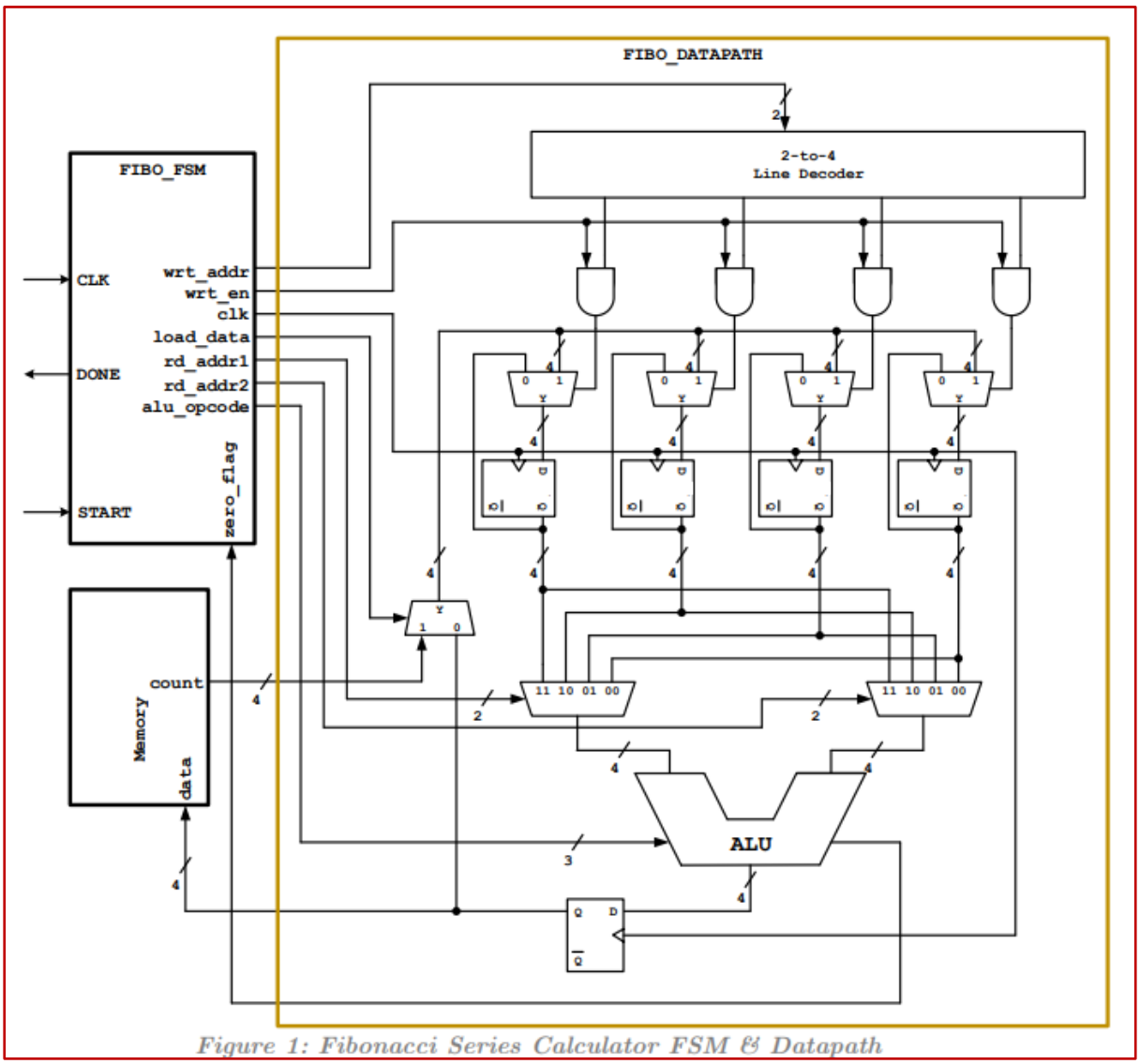


Figure 1: Fibonacci Series Calculator FSM & Datapath

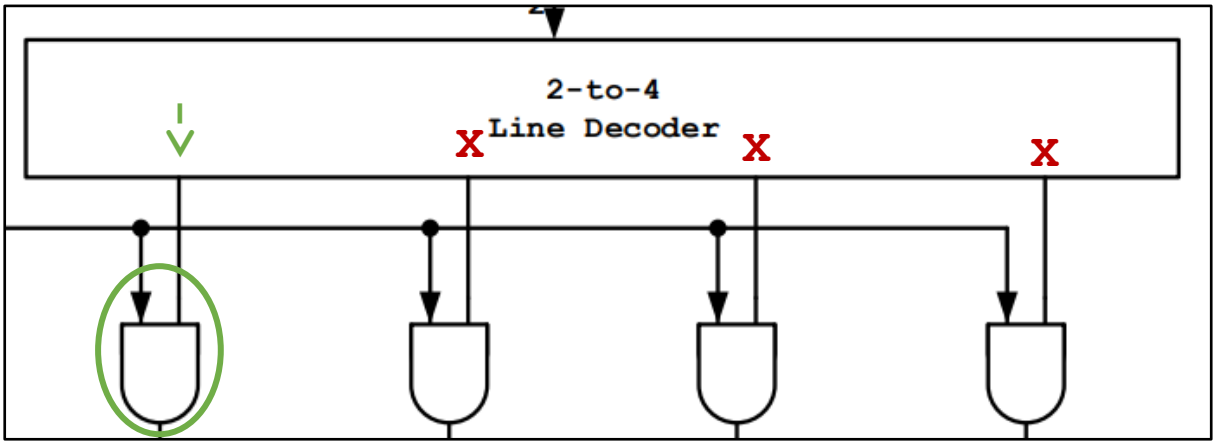
4.3.1: 2-to-4 Line decoder:

Truth table for 2-to-4 decoder:

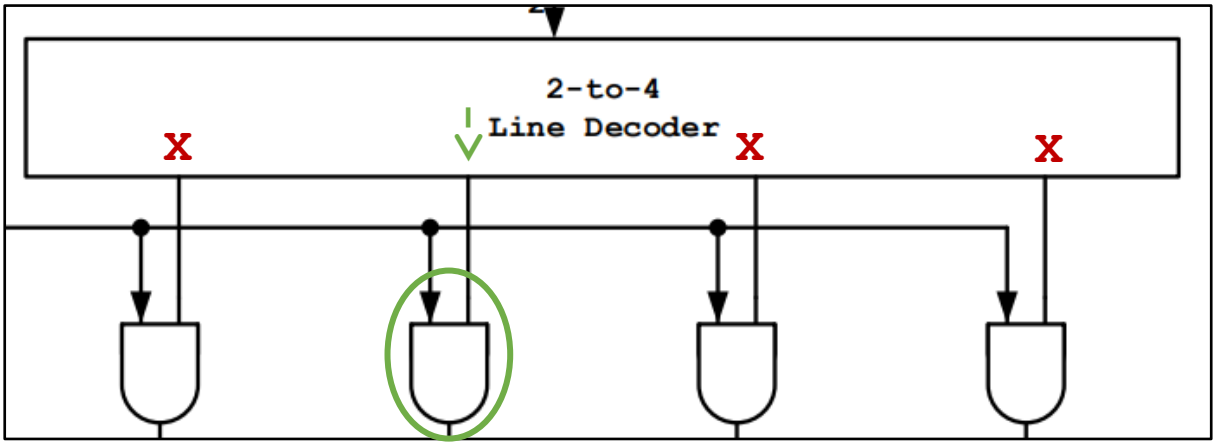
A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	

2 bit input would turn into a 2^2 = 4 outputs.

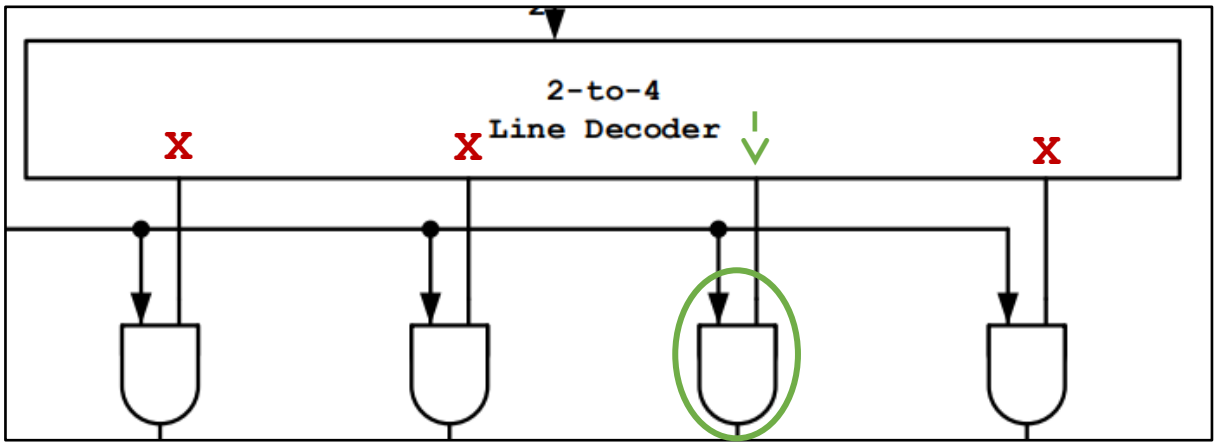
For input 00:



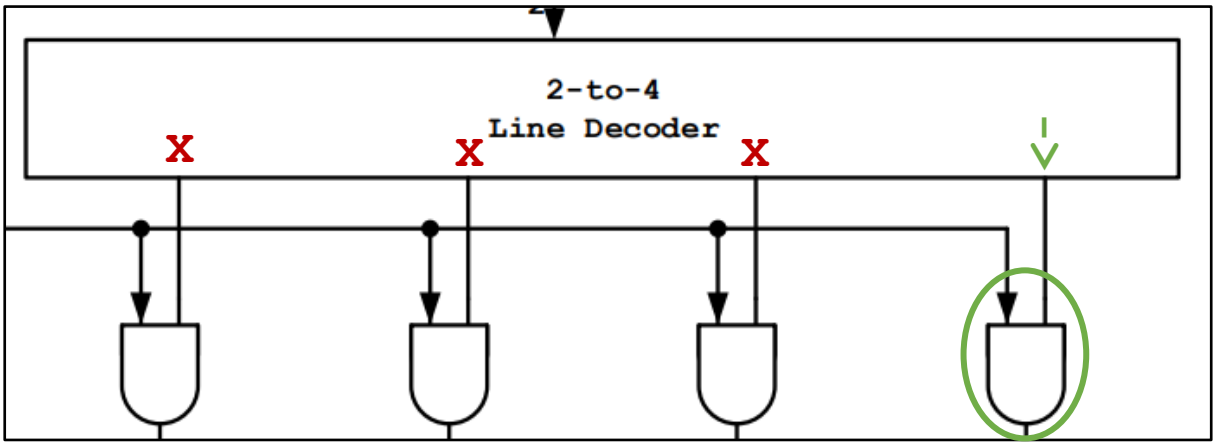
For input 01:

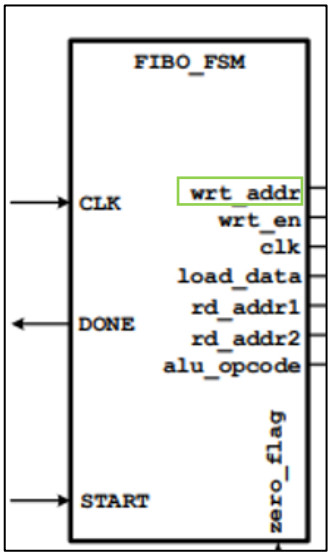


For input 10:



For input 11:





This way we extend 2 bit inputs into four outputs. 2 bits would extend to four different operations. We can choose the operation via wrt_addr which is the input for decoder that will come from the FSM.

Verilog Code:

```
2   module two_to_four_decoder(in, out);
3
4   input [1:0]in;
5   output [3:0]out;
6
7
8      assign out[0] = (~in[1] & ~in[0]);
9      assign out[1] = (~in[1] &  in[0]);
10     assign out[2] = ( in[1] & ~in[0]);
11     assign out[3] = ( in[1] &  in[0]);
12
13
14  endmodule
```

Code was implemented via continuous assignment.

in_1	in_0	out_3	out_2	out_1	out_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

out[0] = (~in[1] & ~in[0]), out[1], out[2], out[3] = 0

out[0] = 1

out[1] = (~in[1] & in[0]), out[0], out[2], out[3] = 0

out[1] = 1

out[2] = (in[1] & ~in[0]), out[0], out[1], out[3] = 0

= 1

out[3] = (in[1] & in[0]), out[0], out[1], out[2] = 0

= 1

When input is 00:

Out[0] = 1 & 1 = 1;

Out[1] = 1 & 0 = 0;

Out[2] = 0 & 1 = 0;

Out[3] = 0 & 0 = 0;

Out = 0001

When input is 01:

Out[0] = 1 & 0 = 0;

Out[1] = 1 & 1 = 1;

Out[2] = 0 & 0 = 0;

Out[3] = 0 & 1 = 0;

Out = 0010

When input is 10:

Out[0] = 0 & 1 = 0;

Out[1] = 0 & 0 = 0;

Out[2] = 1 & 1 = 1;

Out[3] = 1 & 0 = 0;

Out = 0100

When input is 11:

Out[0] = 0 & 0 = 0;

Out[1] = 0 & 1 = 0;

Out[2] = 1 & 0 = 0;

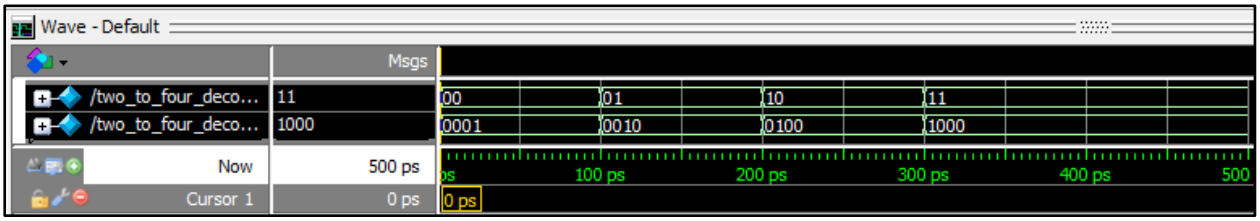
Out[3] = 1 & 1 = 1;

Out = 1000

Testbench Code:

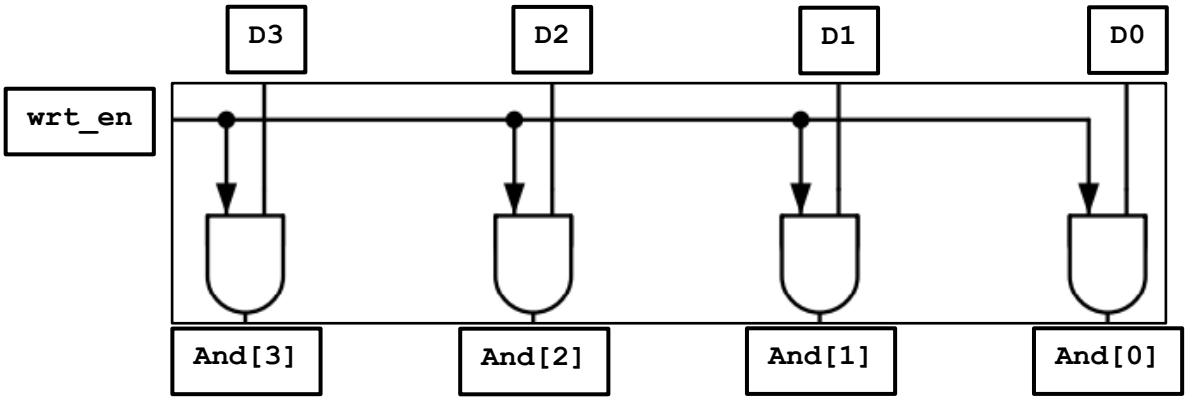
```
2   module two_to_four_decoder_testbench();
3
4   reg [1:0]in;
5   wire [3:0]out;
6
7
8   two_to_four_decoder DUT(in, out);
9
10     initial
11        begin
12           in[1] = 1'b0; in[0] = 1'b0; #100;
13           in[1] = 1'b0; in[0] = 1'b1; #100;
14           in[1] = 1'b1; in[0] = 1'b0; #100;
15           in[1] = 1'b1; in[0] = 1'b1; #100;
16        end
17
18  endmodule
```

Register inputs would be 00, 01, 10, 11 like a truth table.



We can see that the implementation of the code is correct. 00 is 0001, 01 is 0010, 10 is 0100 and 11 is 1000.

4.3.2: 4-bit AND gate:



D	And	
	wrt_en = 0	wrt_en = 1
0001	0000	0001
0010	0000	0010
0100	0000	0100
1000	0000	1000

We can see that wrt_en will control if an operation will happen or not.

When wrt_en = 0 regardless of D output will be 0.

When wrt_en = 1 the output will be D.

Verilog Code:

```
2  module four_bit_and_gate(out, in1, in0);
3
4  parameter size = 4;
5
6  input [size-1:0]in1, in0;
7  output [size-1:0]out;
8
9  genvar i;
10
11    generate
12       for (i=0; i < size; i = i + 1)begin: andoperation
13             and(out[i] , in1[i] , in0[i]);
14          end
15    endgenerate
16
17 endmodule
```

This code would and 4 bit in1 and in0 inputs. It would work for any four bit inputs. How will it for work D and wrt_en inputs?

If wrt_en is 0:

in1 = 0000

in0 = D3D2D1D0

out[0] = 0 & D[0] = 0

out[1] = 0 & D[1] = 0

out[2] = 0 & D[2] = 0

out[3] = 0 & D[3] = 0

out = 0000

Regardless of D, if wrt_en is 0, out will be 0.

If wrt_en is 1:

in1 = 1111

in0 = D3D2D1D0

out[0] = 1 & D[0] = D[0]

out[1] = 1 & D[1] = D[1]

out[2] = 1 & D[2] = D[2]

out[3] = 1 & D[3] = D[3]

out = D3D2D1D0

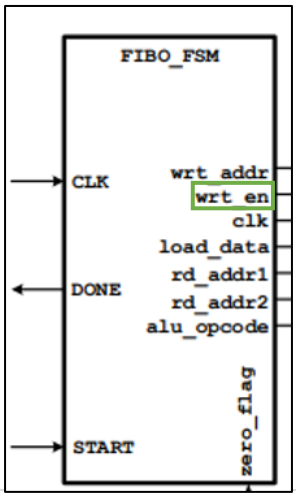
if D = 0001, out = 0001

if D = 0010, out = 0010

if D = 0100, out = 0100

if D = 1000, out = 1000

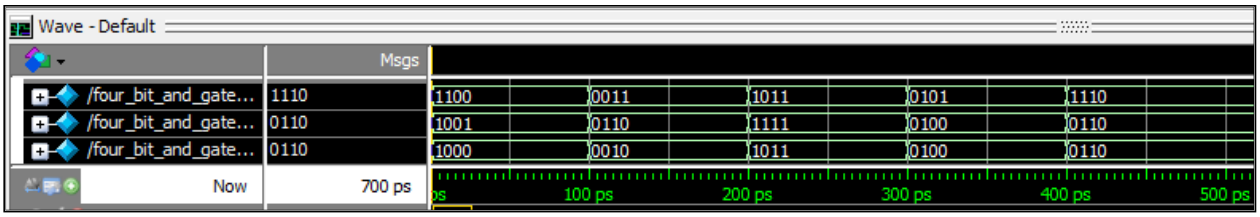
Wrt_en comes from FSM:



Testbench Code:

```
module four_bit_and_gate_testbench();

reg [3:0]in1, in0;
wire [3:0]out;

four_bit_and_gate DUT(out, in1, in0);

   initial
      begin
         in1[3] = 1'b1; in1[2] = 1'b1; in1[1] = 1'b0; in1[0] = 1'b0; /**/ in0[3] = 1'b1; in0[2] = 1'b0; in0[1] = 1'b0; in0[0] = 1'b1; #100;
         in1[3] = 1'b0; in1[2] = 1'b0; in1[1] = 1'b1; in1[0] = 1'b1; /**/ in0[3] = 1'b0; in0[2] = 1'b1; in0[1] = 1'b1; in0[0] = 1'b0; #100;
         in1[3] = 1'b1; in1[2] = 1'b0; in1[1] = 1'b1; in1[0] = 1'b1; /**/ in0[3] = 1'b1; in0[2] = 1'b1; in0[1] = 1'b1; in0[0] = 1'b1; #100;
         in1[3] = 1'b0; in1[2] = 1'b1; in1[1] = 1'b0; in1[0] = 1'b1; /**/ in0[3] = 1'b0; in0[2] = 1'b1; in0[1] = 1'b0; in0[0] = 1'b0; #100;
         in1[3] = 1'b1; in1[2] = 1'b1; in1[1] = 1'b1; in1[0] = 1'b0; /**/ in0[3] = 1'b0; in0[2] = 1'b1; in0[1] = 1'b1; in0[0] = 1'b0; #100;

      end

endmodule
```

Register inputs are random for in1 and in0:



The code works for any input as we can see. Let's check:

For in0 = 1100, in1 = 1001:

1 & 1 = 1

1 & 0 = 0

0 & 0 = 0

0 & 1 = 0

Out = 1000

For in0 = 0011, in1 = 0110:

0 & 0 = 0

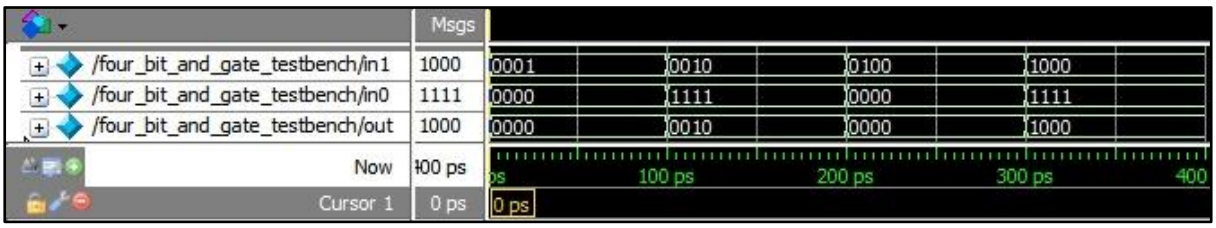
0 & 1 = 0

1 & 1 = 1

1 & 0 = 0

Out = 0010

When we change the testbench inputs we can see how the andgates would work in the datapath from the simulation.



Wrt_en = 0 -> in0 = 0000

in1 = D3D2D1D0 ->in instance 0-100 ps: in1 = 0001

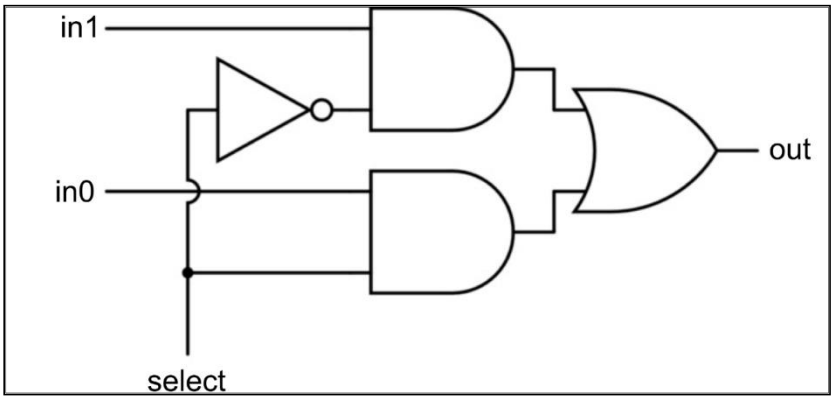
out = 0000

Wrt_en = 1 -> in0 = 1111

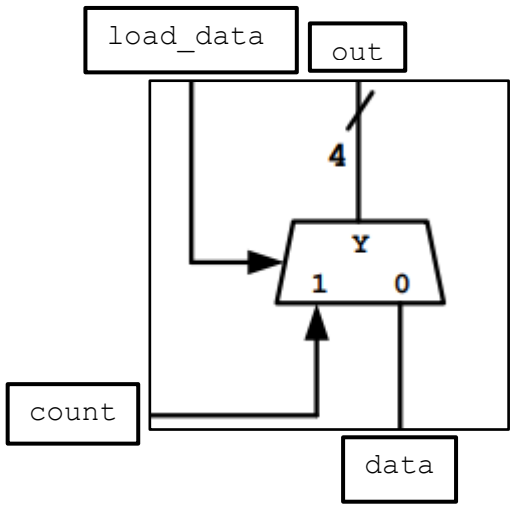
in1 = D3D2D1D0 ->in instance 100-200 ps: in1 = 0010

out = 0010

4.3.3: 4 bit 2-to-1 Multiplexers:

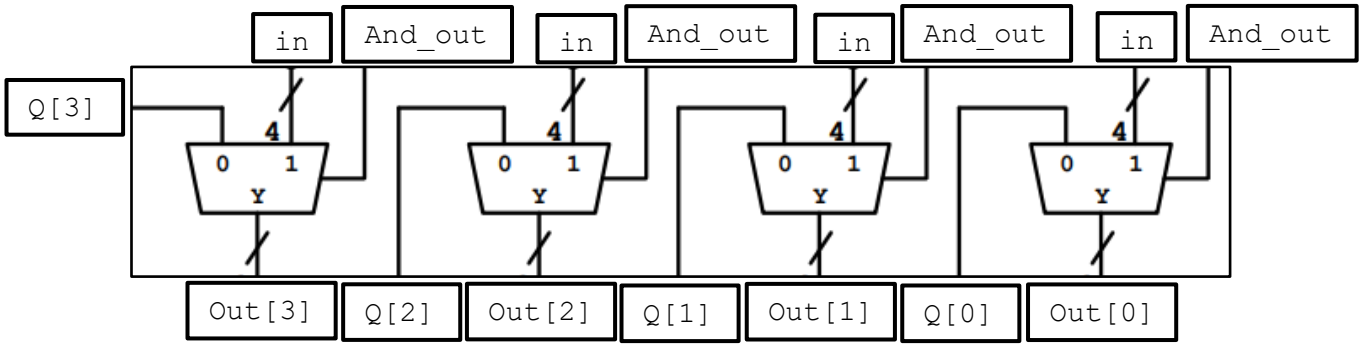


out = (in1 & ~select)+(in0 & select)



load_data = 0: out = data

load_data = 1: out = count



in0	in1	And_out	out
Q	data	0	Q
Q	data	1	data
Q	count	0	Q
Q	count	1	count

And_out = 0: out = Retain state

And_out = 1: out = in1

if load_data = 0: out = data

if load_data = 1: out = count

Verilog code:

```
module four_bit_two_to_one_mux(in1, in0, select, out);
parameter size = 4;
input [size-1:0]in1, in0;
input select;
output [size-1:0]out;
wire [size-1:0]wire1,wire2,wire3;




genvar i;

   generate
      for(i=0; i < size; i=i+1)begin: mux
         not(wire1[i], select);
         and(wire2[i], in0[i], select);
         and(wire3[i], in1[i], wire1[i]);
         or(out[i], wire2[i], wire3[i]);
      end

   endgenerate

endmodule
```

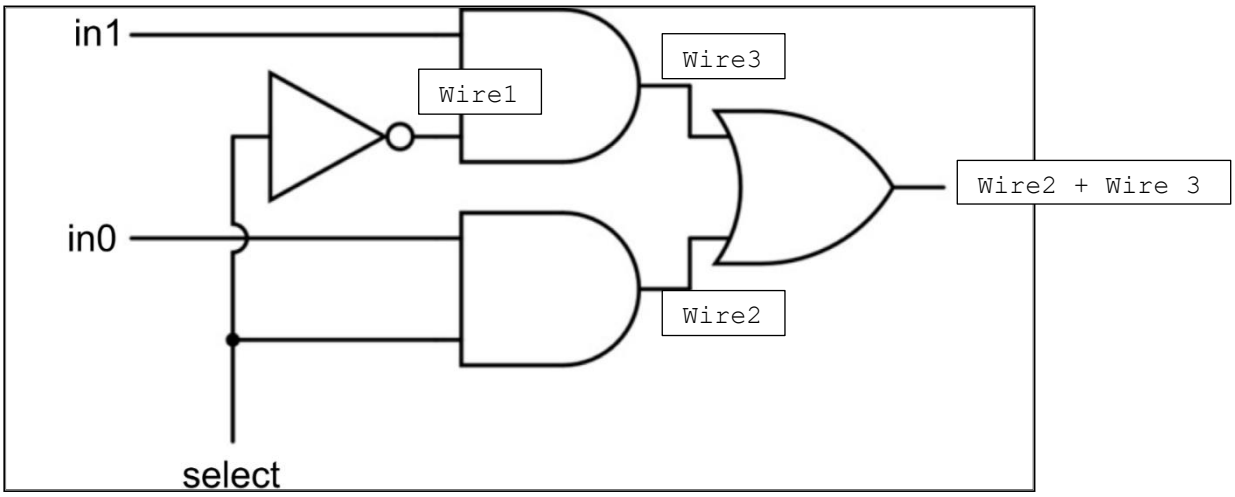
Generate loop:

Wire1[i] = ~select;

Wire2[i] = in0[i] & select;

Wire3[i] = in1[i] & Wire1[i];

Out[i] = Wire2[i] + Wire3[i];



Let’s say in1 = AAAA

Let’s say in0 = BBBB

Let’s say select = 0

Generate loop:

Wire1[i] = 1;

Wire2[i] = B & 0 = 0;

Wire3[i] = A & 1 = A;

Out[i] = 0 + A = A;

Out = AAAA;

Let’s say select = 1

Generate loop:

Wire1[i] = 1;

Wire2[i] = B & 1 = B;

Wire3[i] = A & 0 = 0;

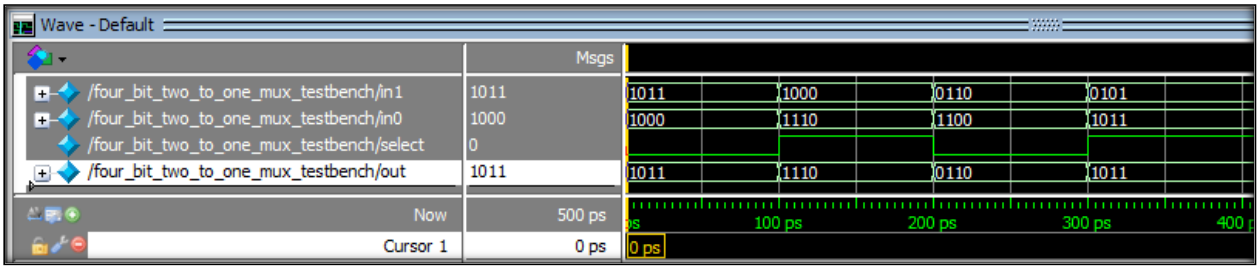
Out[i] = B + 0 = B;

Out = BBBB;

Testbench code:

```
module four_bit_two_to_one_mux_testbench();

reg [3:0]in1,in0;
reg select;
wire [3:0]out;


four_bit_two_to_one_mux DUT(in1, in0, select, out);


   initial
      begin
         in1[3] = 1'b1; in1[2] = 1'b0; in1[1] = 1'b1; in1[0] = 1'b1; /**/ in0[3] = 1'b1; in0[2] = 1'b0; in0[1] = 1'b0; in0[0] = 1'b0; select = 1'b0; #100;
         in1[3] = 1'b1; in1[2] = 1'b0; in1[1] = 1'b0; in1[0] = 1'b0; /**/ in0[3] = 1'b1; in0[2] = 1'b1; in0[1] = 1'b1; in0[0] = 1'b0; select = 1'b1; #100;
         in1[3] = 1'b0; in1[2] = 1'b1; in1[1] = 1'b1; in1[0] = 1'b0; /**/ in0[3] = 1'b1; in0[2] = 1'b1; in0[1] = 1'b0; in0[0] = 1'b0; select = 1'b0; #100;
         in1[3] = 1'b0; in1[2] = 1'b1; in1[1] = 1'b0; in1[0] = 1'b1; /**/ in0[3] = 1'b1; in0[2] = 1'b0; in0[1] = 1'b1; in0[0] = 1'b1; select = 1'b1; #100;
      end

endmodule
```

Random inputs are given for in1 and in0.



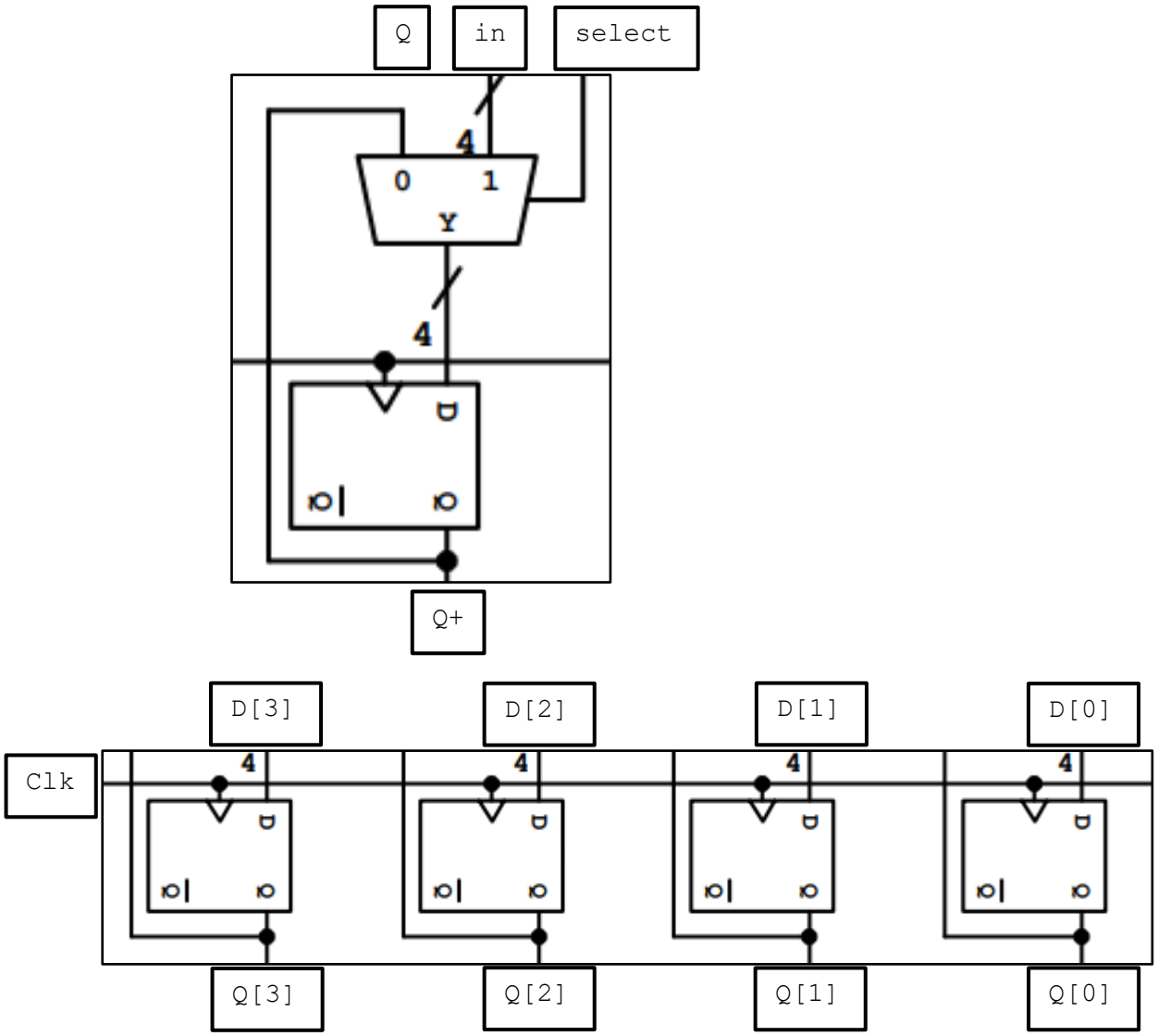
When select is 0 the output should be in1.

When select is 1 the output should be in0.

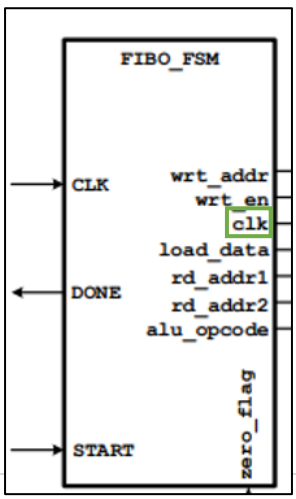
Our code works since output is expected for each select input.

4.3.4: 4 bit Register:

4 bit registers previously used in Project 3 have been used. The only difference is that asynchronous reset and load functions don't exist.



Clk comes from FSM:



1 bit Register Verilog Code:

```
2   module register(d, clk, q, qnot);
3
4
5
6   input d, clk;
7   output reg q, qnot;
8
9      initial
10        begin
11           q = 1'b0;
12           qnot = 1'b1;
13        end
14
15     always @(posedge clk)
16        begin
17           q = d;
18           qnot = ~q;
19        end
20
21  endmodule
```

Q+ = D;

4 bit Register Verilog Code:

```
2   module four_bit_register(d, clk, q, qnot);
3
4   parameter size = 4;
5
6
7   input [size-1:0]d;
8   input clk;
9   output [size-1:0]q, qnot;
10
11
12  genvar i;
13
14     generate
15        for(i = 0; i < size; i = i + 1) begin: fourbitregister
16           register U1(d[i], clk, q[i], qnot[i]);
17        end
18     endgenerate
19
20  endmodule
21
```

Register have been parametrised.

Generate loop:

Q[i] = d[i];

Qnot[i] = ~d[i];

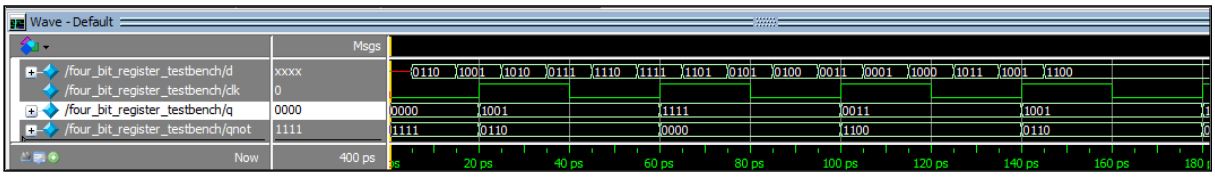
Q+ = D:

Testbench Code:

```
2   module four_bit_register_testbench();
3
4   reg [3:0]d;
5   reg clk;
6   wire [3:0]q, qnot;
7
8   four_bit_register DUT(d, clk, q, qnot);
9
10     always
11        begin
12           clk = 1'b0; #20;
13           clk = 1'b1; #20;
14        end
15
16     initial
17        fork: testbench
18           #5   d = 4'b0110;
19           #15  d = 4'b1001;
20           #25  d = 4'b1010;
21           #35  d = 4'b0111;
22           #45  d = 4'b1110;
23           #55  d = 4'b1111;
24           #65  d = 4'b1101;
25           #75  d = 4'b0101;
26           #85  d = 4'b0100;
27           #95  d = 4'b0011;
28           #105 d = 4'b0001;
29           #115 d = 4'b1000;
30           #125 d = 4'b1011;
31           #135 d = 4'b1001;
32           #145 d = 4'b1100;
33
34        join
35
36  endmodule
```

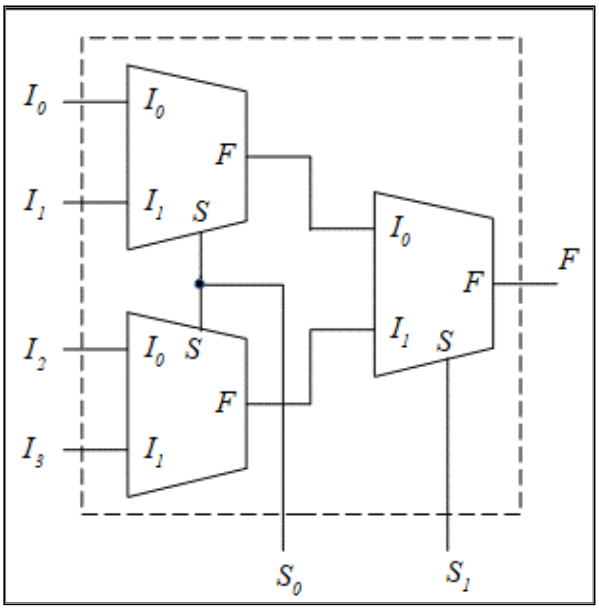
Random values have been given for d values at different times.

Starting from 5 pico seconds, there are different d values at 10 pico second intervals.

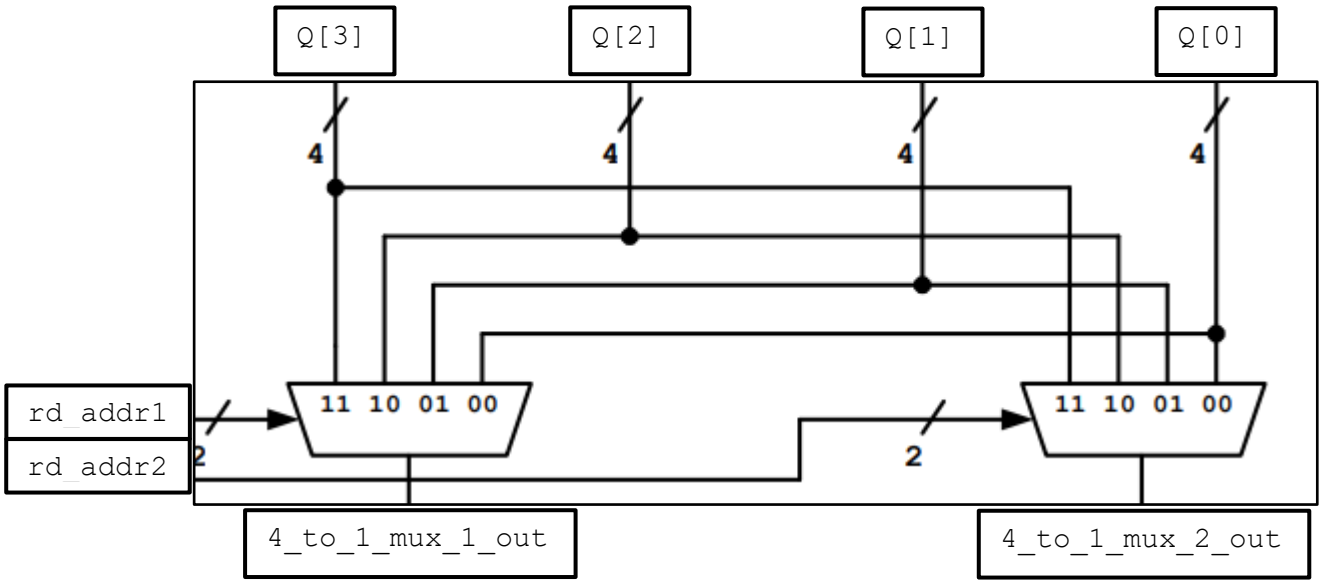


When clk is at positive edge, d values are updated. At positive edge of clk at 20 ps we can see that 1010 value had just been registered. We can see that the 4 bit register gives out a desired result.

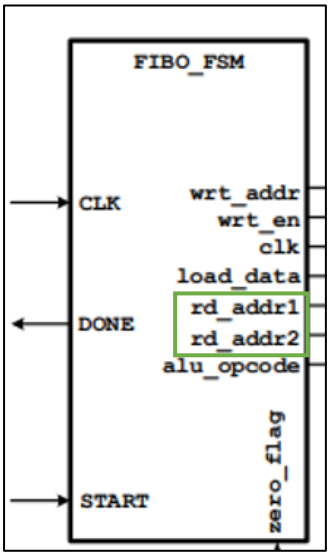
4.3.5: 4 to 1 multiplexer:



Using 2 to 1 multiplexer we can create a 4 to 1 multiplexer.



rd_addr comes from the FSM:



Verilog Code:

```
1   module four_bit_four_to_one_mux(in3, in2, in1, in0, select, out);
2
3   parameter size = 4;
4
5
6   input [size-1:0]in3, in2, in1, in0;
7   input [1:0]select;
8   output [size-1:0]out;
9   wire [size-1:0]w1,w2;
10
11
12          four_bit_two_to_one_mux M1(in3, in2, select[0], w1);
13          four_bit_two_to_one_mux M2(in1, in0, select[0], w2);
14          four_bit_two_to_one_mux M3(w1, w2, select[1], out);
15
16  endmodule
17
```

4-Bit 4-to-1 mux is implemented using 4-bit 2-to-1 muxs in the arrangement in above schematic.

Select = 00 -> out = in3;

Select = 01 -> out = in2;

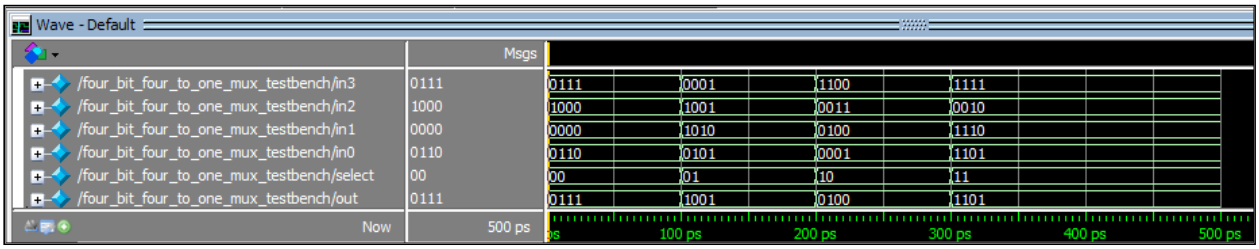
Select = 10 -> out = in1;

Select = 11 -> out = in0;

Testbench Code:

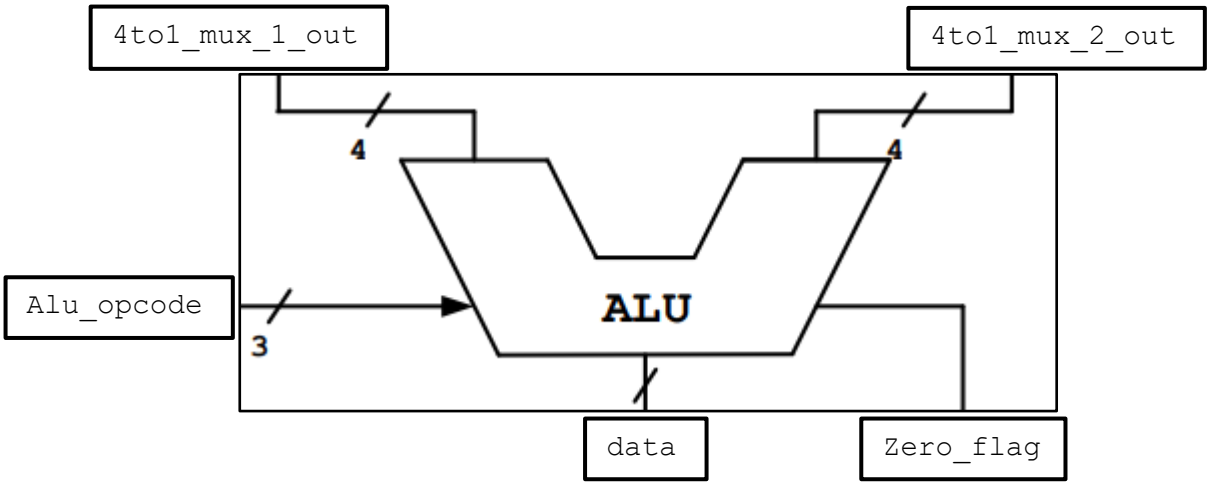
```
2   module four_bit_four_to_one_mux_testbench();
3
4   reg [3:0]in3, in2, in1, in0;
5   reg [1:0]select;
6   wire [3:0]out;
7
8   four_bit_four_to_one_mux DUT(in3, in2, in1, in0, select, out);
9
10     initial
11        begin
12           select = 2'b00;
13           in3 = 4'b0111;
14           in2 = 4'b1000;
15           in1 = 4'b0000;
16           in0 = 4'b0110;
17           #100;
18
19           select = 2'b01;
20           in3 = 4'b0001;
21           in2 = 4'b1001;
22           in1 = 4'b1010;
23           in0 = 4'b0101;
24           #100;
25
26           select = 2'b10;
27           in3 = 4'b1100;
28           in2 = 4'b0011;
29           in1 = 4'b0100;
30           in0 = 4'b0001;
31           #100;
32
33           select = 2'b11;
34           in3 = 4'b1111;
35           in2 = 4'b0010;
36           in1 = 4'b1110;
37           in0 = 4'b1101;
38           #100;
39
40           #100;
41        end
42  endmodule
```

Random values are given for in3, in2, in1 and in0 inputs. Values change very 100 pico seconds.



The simulation shows the outputs as we expected.

4.3.6: 4 bit ALU:



alu_opcode	operation
000	no operation
001	set
010	increment
011	decrement
100	load
101	store
110	add
111	copy

Operations in ALU would be done with spesific alu_opcode inputs.

Verilog code:

```
2   module ALU(opr1, opr2, opcode, out, zero_flag);
3
4   parameter size = 4;
5
6   input [size-1:0]opr1;
7   input [size-1:0]opr2;
8   input [2:0]opcode;
9   output reg [size-1:0]out;
10  output reg zero_flag;
11
12     initial
13        begin
14           zero_flag = 1'b0;
15           out = 4'xxxx;
16        end
17
18     always @(opcode) begin
19        case (opcode)
20           3'b001: out = 1;
21           3'b010: out = opr1 + 1;
22           3'b011: begin
23                    out = opr1 - 1;
24                       if(out == 4'b0000)
25                             zero_flag = 1'b1;
26                    end
27           3'b100: ;
28           3'b101: out = opr1;
29           3'b110: out = (opr1 + opr2);
30           3'b111: out = opr2;
31           default: ;
32        endcase
33     end
34  endmodule
```

Opr1 is for 4to1_mux_1_out

Opr2 is for 4to1_mux_2_out

When zero_flag is 1 the operation is finished. This case would happen when out(data) would equal to 0000.

Otherwise zero_flag doesn't change out.

alu_opcode	Opr 1	Opr 2	Op	Instruction	
000	--	--	noop	--	--
001	xx	--	set	R[xx] =	1
010	xx	--	increment	R[xx] =	R[xx] + 1
011	xx	--	decrement	R[xx] =	R[xx] - 1
100	xx	--	load	R[xx] =	Data_in
101	xx	--	store	Data_out =	R[xx]
110	xx	yy	add	R[xx] =	R[xx] + R[yy]
111	xx	yy	copy	R[xx] =	R[yy]

Opcode = 001 -> out = 1;

Opcode = 010 -> out = opr1 + 1;

Opcode = 011 -> out = opr1 – 1;

Opcode = 100 -> out = Data_in (load data);

Opcode = 101 -> out = opr1 (store);

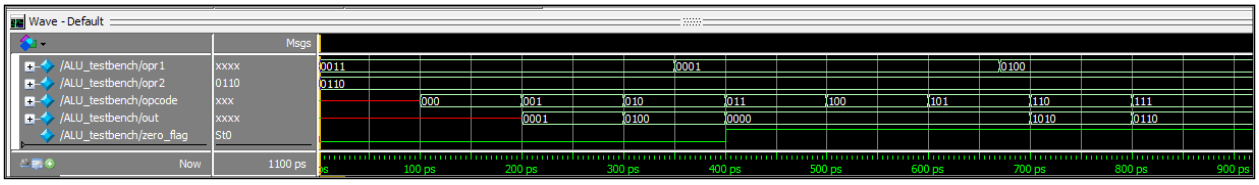
Opcode = 110 -> out = opr1 + opr2;

Opcode = 111 -> out = opr2;

Testbench Code:

```
2   module ALU_testbench();
3
4   reg [3:0]opr1, opr2;
5   reg [2:0]opcode;
6   wire [3:0]out;
7   wire zero_flag;
8
9   ALU DUT(opr1, opr2, opcode, out, zero_flag);
10
11     initial
12        fork: ALU
13           #1 opr1 = 4'b0011; opr2 = 4'b0110;
14           #100 opcode = 3'b000;
15           #200 opcode = 3'b001;
16           #300 opcode = 3'b010;
17           #350 opr1  = 4'b0001;
18           #400 opcode = 3'b011;
19           #500 opcode = 3'b100;
20           #600 opcode = 3'b101;
21           #670 opr1  = 4'b0100;
22           #700 opcode = 3'b110;
23           #800 opcode = 3'b111;
24        join
25
26  endmodule
```

Opr1 and opr2 values are randomly given.



Simulation revealed the desired behavior.

Simulation starts with zero_flag = 0, out = xxxx, opcode xxx as expected since they are not given any value initially.

100-200ps: opcode is 000 which is noop operation. Therefore, we can see that nothing changes in this interval.

200-300ps: opcode is 001, which is set operation. We can see that output is 0001.

300-400ps: opcode is 010, which is increment operation. Output is 0100 (4 in decimal), input opr1 is 0011 (3 in decimal). Therefore, output is 1 + opr1. Also, on 350ps opr1 is set to be 0001 in order to illustrate the functionality of zero_flag in the upcoming interval 400-500ps.

400-500ps: opcode is 011, which is decrement operation. We can clearly see that input opr1 is decremented by 1 (0001 - 0001 = 0000) and output is 0000. We can see that decrement operation works but also, since we reached output = 0000, zero_flag is 1. (Setting zero flag back to 0 is not written in the code

since zero_flag = 1 means the end of the fibo_calculator operation)

500-700ps: in this interval, as explained before, ALU does nothing. And opr1 is set to be 0100 in order to demonstrate addition next.

700-800ps: opcode is 110, which is addition operation. opr1 and opr2 are 0100 and 0110 in this interval respectively. And output is 1010. It is clear that

0100 + 0110 = 1010.

800-900ps: opcode is 111, which is copy operation. We can see that output is 0110, which the value of opr2.

4.3.6: Fibo Datapath:

To create the datapath we need to connect all the units we have created. This includes:

One 2-to-4 Line Decoder

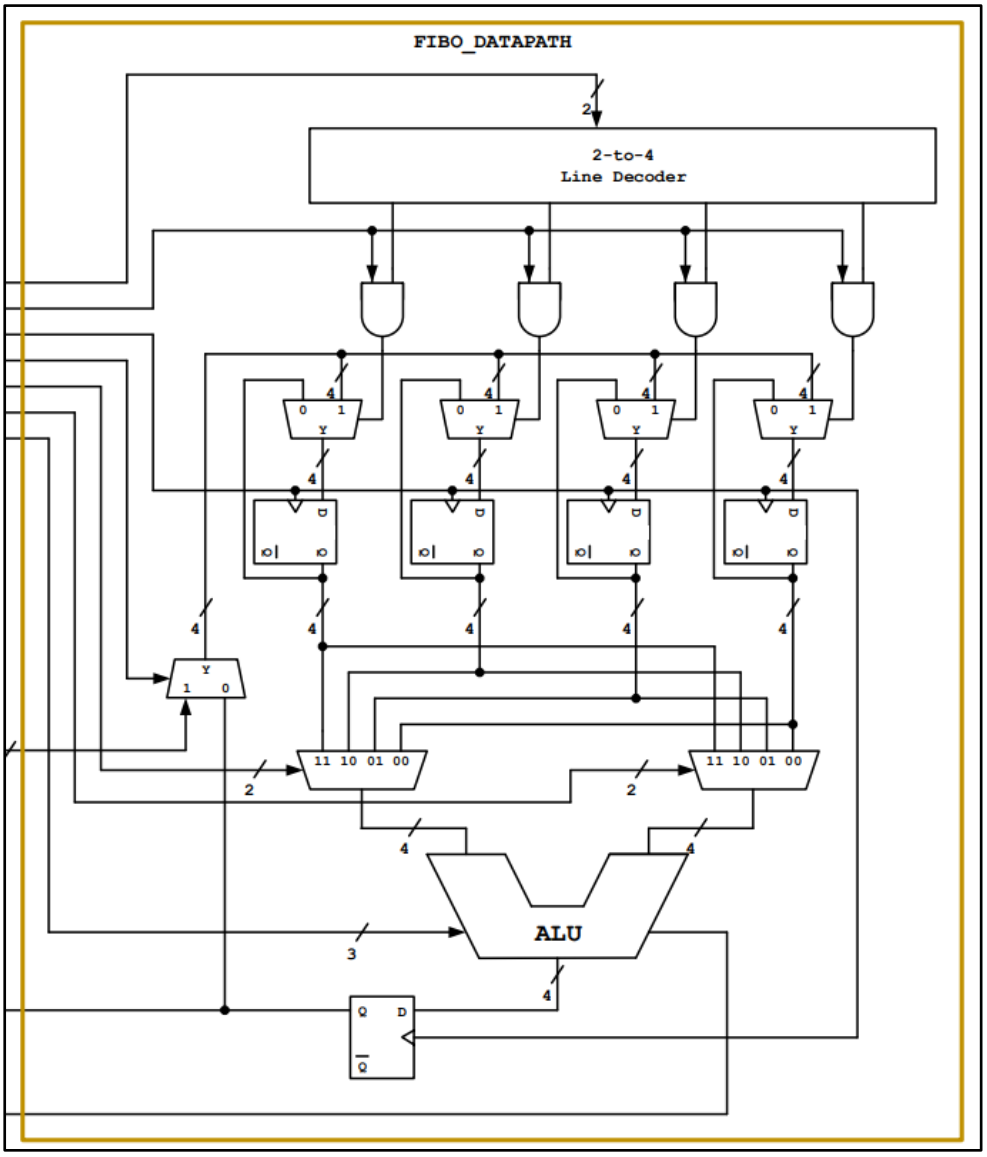
Four 4-bit AND gates

Five 2-to-1 Multiplexers

Five 4-bit Registers

Two 4-to-1 Multiplexers

One 4-bit ALU



Verilog Code:

```
module fibo_datapath(wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode, count, data, zero_flag,regouttest3,regouttest2,regouttest1,regouttest0, aluOutTest);

parameter size = 4;

input wrt_en, clk, load_data;
input [1:0] wrt_addr, rd_addr1, rd_addr2;
input [2:0] alu_opcode;
input [size-1:0] count;
output [size-1:0] data;
output zero_flag;


output [3:0]regouttest3,regouttest2,regouttest1,regouttest0;
output [3:0] aluOutTest;


wire [3:0]decoderWire;
wire [3:0]andWire;
wire [3:0]wrt_en_wire;
wire [size-1:0]qnotWire3,qnotWire2,qnotWire1,qnotWire0,qnotWireOut;

wire [size-1:0]ttoMuxWire3, ttoMuxWire2, ttoMuxWire1, ttoMuxWire0;
wire [size-1:0]ftoMuxWire1, ftoMuxWire0;
wire [size-1:0]regWire3, regWire2, regWire1, regWire0;

wire [size-1:0]ALU_Wire;

wire [size-1:0]dataMuxWire;

one_to_four_bit OTF(wrt_en, wrt_en_wire);  //extending 1-bit wrt_en to 4-bit wrt_en_wire


two_to_four_decoder DEC(wrt_addr, decoderWire);
four_bit_and_gate AND(andWire, wrt_en_wire, decoderWire);

four_bit_two_to_one_mux TTO3(regWire3, dataMuxWire , andWire[3], ttoMuxWire3);
four_bit_two_to_one_mux TTO2(regWire2, dataMuxWire , andWire[2], ttoMuxWire2);
four_bit_two_to_one_mux TTO1(regWire1, dataMuxWire , andWire[1], ttoMuxWire1);
four_bit_two_to_one_mux TTO0(regWire0, dataMuxWire , andWire[0], ttoMuxWire0);

four_bit_register reg3(ttoMuxWire3, clk, regWire3, qnotWire3);
four_bit_register reg2(ttoMuxWire2, clk, regWire2, qnotWire2);
four_bit_register reg1(ttoMuxWire1, clk, regWire1, qnotWire1);
four_bit_register reg0(ttoMuxWire0, clk, regWire0, qnotWire0);

four_bit_four_to_one_mux FTO1(regWire0, regWire1, regWire2, regWire3, rd_addr1, ftoMuxWire1);
four_bit_four_to_one_mux FTO0(regWire0, regWire1, regWire2, regWire3, rd_addr2, ftoMuxWire0);

ALU Arit(ftoMuxWire1, ftoMuxWire0, alu_opcode, aluOutTest, zero_flag);

four_bit_register regout(aluOutTest, clk, data, qnotWireOut);

four_bit_two_to_one_mux M16(data, count, load_data, dataMuxWire);

not N1(regouttest3[3],qnotWire3[3]);
not N2(regouttest3[2],qnotWire3[2]);
not N3(regouttest3[1],qnotWire3[1]);
not N4(regouttest3[0],qnotWire3[0]);

not N5(regouttest2[3],qnotWire2[3]);
not N6(regouttest2[2],qnotWire2[2]);
not N7(regouttest2[1],qnotWire2[1]);
not N8(regouttest2[0],qnotWire2[0]);

not N10(regouttest1[3],qnotWire1[3]);
not N11(regouttest1[2],qnotWire1[2]);
not N12(regouttest1[1],qnotWire1[1]);
not N13(regouttest1[0],qnotWire1[0]);

not N14(regouttest0[3],qnotWire0[3]);
not N15(regouttest0[2],qnotWire0[2]);
not N16(regouttest0[1],qnotWire0[1]);
not N17(regouttest0[0],qnotWire0[0]);
endmodule
```

All the blocks are connected using appropriately named wires.

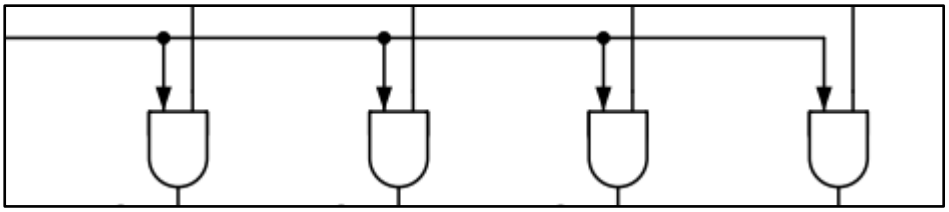
Additional outputs, regouttest, aluOutTest are added in order to explain testbench better and will not be present in the next weeks fibo calculator project.

A function called one_to_four_bit is used in order to connect the 1-bit wrt_en into the 4-bit input of the 4-bit and gate. Which will be explained below.

```
2   module one_to_four_bit(in, out);
3
4   input in;
5   output [3:0]out;
6
7   assign out[3] = in;
8   assign out[2] = in;
9   assign out[1] = in;
10  assign out[0] = in;
11
12  endmodule
```

In this code, 1-bit input is directed into each bit of a 4-bit output

This is for wrt_en.



Wrt_en is 1 bit in the FSM, it was extended to 4 bits to feed into the and gates.

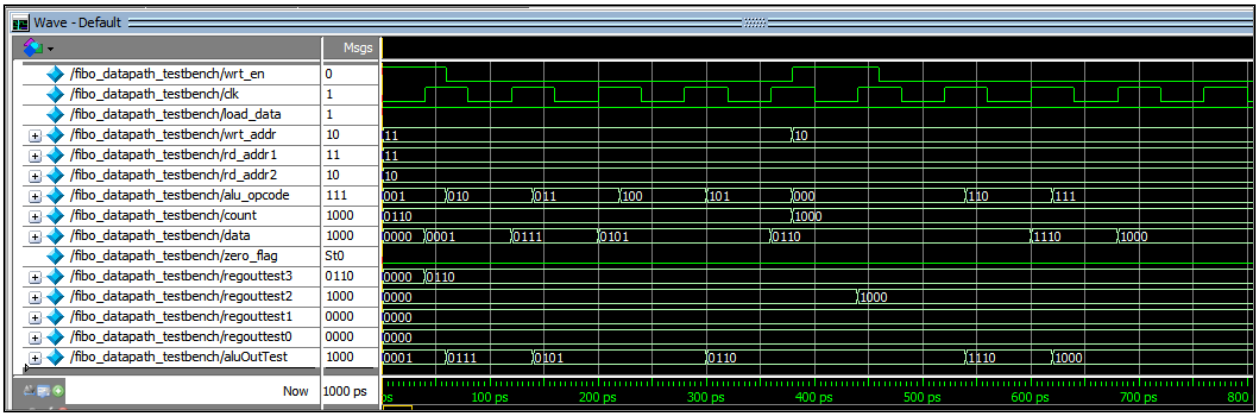
Testbench Code:

```
module fibo_datapath_testbench();

reg wrt_en, clk, load_data;
reg [1:0] wrt_addr, rd_addr1, rd_addr2;
reg [2:0] alu_opcode;
reg [3:0] count;
wire [3:0] data;
wire zero_flag;
wire [3:0]regouttest3,regouttest2,regouttest1,regouttest0;
wire [3:0] aluOutTest;


fibo_datapath DUT(wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode, count, data, zero_flag,regouttest3,regouttest2,regouttest1,regouttest0, aluOutTest);

   always
      begin
      clk = 1'b0; #40;
      clk = 1'b1; #40;
      end

   initial
      fork: test
            begin wrt_addr = 2'b11; wrt_en = 1'b1; load_data = 1'b1; rd_addr1 = 2'b11; rd_addr2 = 2'b10; alu_opcode = 3'b001; count = 4'b0110; end
         #60  begin                  wrt_en = 1'b0;                                                     alu_opcode = 3'b010;                  end
         #140 begin                                                                                     alu_opcode = 3'b011;                  end
         #220 begin                                                                                     alu_opcode = 3'b100;                  end
         #300 begin                                                                                     alu_opcode = 3'b101;                  end
         #380 begin wrt_addr = 2'b10; wrt_en = 1'b1;                                                    alu_opcode = 3'b000; count = 4'b1000; end
         #460 begin                  wrt_en = 1'b0;                                                                                       end
         #540 begin                                                                                     alu_opcode = 3'b110;                  end
         #620 begin                                                                                     alu_opcode = 3'b111;                  end

      join

endmodule
```

Testbench code is written in usual fashion.

Clock is alternated inside an always block, input values are assigned inside an initial fork/join block. Every possible opcode value are assigned here with the appropriate wrt_addr, wrt_en, load_data, rd_addr1, rd_addr0, and count value combinations.

These are going to be explained in the ***Simulation Results*** part below.



0ps: Initially, wrt_en = 1 in order to write a data into a register.

Load_data = 1 in order to write the data in the count input into the register.

Write_addr = 11 in order to write the data in count input into the register with the address 11.

Rd_addr1 = 11, rd_addr2 = 10 in order to allow ALU to get the values from registers 11 and 10.

Register 10 will not be assigned a data until later.

Alu_opcode = 001 in order to demonstrate *set* operation. in fact the temporary output aluOutTest is 0001 at this instant, however data output waits the first posedge to update to 0001 at 40ps.

Count value is selected randomly as 0110.

40ps: Here we can see that ALU output is written into the output register, and we can see the value 0001 from the *set* operation in the data output. Also, the value in the count input is written into the register with the address 11, and we can see that in the temporary test output regouttest3.

60ps: Here we can see that the opcode is set to be 010, which is the increment operation.

We immediately see that temporary aluOutTest output is updated as 0111. This is because 0110+0001=0111. We see that data output also displays this value in the next clock posedge as expected.

140ps: opcode is set to be 011, which is the decrement operation. we immediately see that temporary aluOutTest output is 0101 because 0110-0001=0101. Operation is made with 0110 instead of 0111 because the value in the register is still

0110, we did not write the incremented value to the register. Although we could simply do so by load_data=0, wrt_en=1.

200ps: decremented value 101 is now displayed in data output since there is posedge here.

220ps: opcode is 100, which is load operation that loads the data to registers, since we don't need the ALU to do anything for this operation nothing changes here. We already established the ability to load data into registers.

300ps: here the opcode is set to be 101, which is store operation. the value in the register will be outputted. Sure enough we see that immediately the temporary output aluOutTest displays the value of register 11. Again, data output waits the posedge clock to display this value at 360ps.

380ps: we are going to write a value to second register with the address 10. So, we set the wrt_addr to 10, opcode to 000 (no operation state), count to 1000 (random value that is going to be assigned to register 10).

440ps: we see a posedge clock here, temporary output regouttest now displays 1000, which means we written the value in count input into the register 10.

540ps: here we set the opcode to 110 in order to perform the addition operation. and immediately we see that temporary output aluOutTest displays 1110, because 0110+1000=1110.

Again, data output waits the next clock posedge to display this result at 600ps.

620ps: here we set the opcode to 111 in order to demonstrate copy operation. temporary output aluOutTest immediately displays 1000, which is the value in register with the address 10. And then data output displays this value at the next posedge clock at 680ps.

4.3.7: Seven Segment Display:

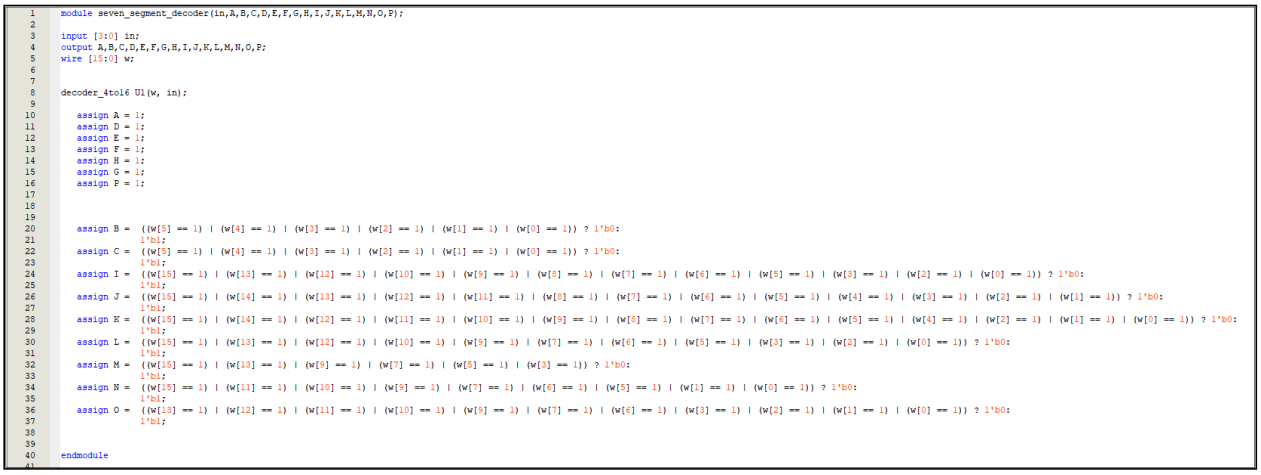
The decoder we used for project 3 has been used.

4-to-16 Decoder Verilog Code:

```
module decoder_4to16(output wire [15:0] F , input wire [3:0] ABCD);

   assign F[15] = ~ABCD[3] & ~ABCD[2] & ~ABCD[1] & ~ABCD[0];
   assign F[14] = ~ABCD[3] & ~ABCD[2] & ~ABCD[1] &  ABCD[0];
   assign F[13] = ~ABCD[3] & ~ABCD[2] &  ABCD[1] & ~ABCD[0];
   assign F[12] = ~ABCD[3] & ~ABCD[2] &  ABCD[1] &  ABCD[0];
   assign F[11] = ~ABCD[3] &  ABCD[2] & ~ABCD[1] & ~ABCD[0];
   assign F[10] = ~ABCD[3] &  ABCD[2] & ~ABCD[1] &  ABCD[0];
   assign F[9]  = ~ABCD[3] &  ABCD[2] &  ABCD[1] & ~ABCD[0];
   assign F[8]  = ~ABCD[3] &  ABCD[2] &  ABCD[1] &  ABCD[0];
   assign F[7]  =  ABCD[3] & ~ABCD[2] & ~ABCD[1] & ~ABCD[0];
   assign F[6]  =  ABCD[3] & ~ABCD[2] & ~ABCD[1] &  ABCD[0];
   assign F[5]  =  ABCD[3] & ~ABCD[2] &  ABCD[1] & ~ABCD[0];
   assign F[4]  =  ABCD[3] & ~ABCD[2] &  ABCD[1] &  ABCD[0];
   assign F[3]  =  ABCD[3] &  ABCD[2] & ~ABCD[1] & ~ABCD[0];
   assign F[2]  =  ABCD[3] &  ABCD[2] & ~ABCD[1] &  ABCD[0];
   assign F[1]  =  ABCD[3] &  ABCD[2] &  ABCD[1] & ~ABCD[0];
   assign F[0]  =  ABCD[3] &  ABCD[2] &  ABCD[1] &  ABCD[0];

endmodule
```

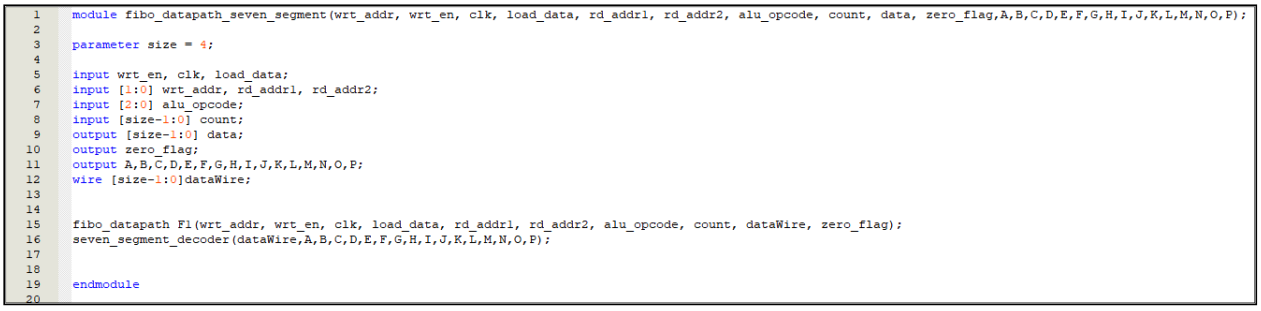
Enable	Inputs				outputs															
E	I_3	I_2	I_1	I_0	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Seven Segment Decoder Verilog Code:



```
module seven_segment_decoder(in,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);

input [3:0] in;
output A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;
wire [15:0] w;


decoder_4to16 U1(w, in);

   assign A = 1;
   assign D = 1;
   assign E = 1;
   assign F = 1;
   assign H = 1;
   assign G = 1;
   assign P = 1;



   assign B =  ((w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign C =  ((w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign I =  ((w[15] == 1) | (w[13] == 1) | (w[12] == 1) | (w[10] == 1) | (w[9] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[3] == 1) | (w[2] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign J =  ((w[15] == 1) | (w[14] == 1) | (w[13] == 1) | (w[12] == 1) | (w[11] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1)) ? 1'b0:
               1'b1;
   assign K =  ((w[15] == 1) | (w[14] == 1) | (w[12] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[4] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign L =  ((w[15] == 1) | (w[13] == 1) | (w[12] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[3] == 1) | (w[2] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign M =  ((w[15] == 1) | (w[13] == 1) | (w[9] == 1) | (w[7] == 1) | (w[5] == 1) | (w[3] == 1)) ? 1'b0:
               1'b1;
   assign N =  ((w[15] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign O =  ((w[13] == 1) | (w[12] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[6] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;


endmodule
```

Fibo Datapath Seven Segment Decoder Verilog Code



```
module fibo_datapath_seven_segment(wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode, count, data, zero_flag,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);

parameter size = 4;

input wrt_en, clk, load_data;
input [1:0] wrt_addr, rd_addr1, rd_addr2;
input [2:0] alu_opcode;
input [size-1:0] count;
output [size-1:0] data;
output zero_flag;
output A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;
wire [size-1:0]dataWire;


fibo_datapath F1(wrt_addr, wrt_en, clk, load_data, rd_addr1, rd_addr2, alu_opcode, count, dataWire, zero_flag);
seven_segment_decoder(dataWire,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);


endmodule
```

Seven segment decoder and fibo datapath codes are simply connected under a top-level code. Output of the fibo datapath is connected to input of the seven-segment decoder with a wire named dataWire.