Lab 2: Binary Combinational Array Multiplier Design, Sequential Multiplier Design

CPE 166/EEE 270 Advanced Logic Design Lab

Lab Session: Monday 2:00 – 4:40

Section: 3

Table of Contents

Introduction

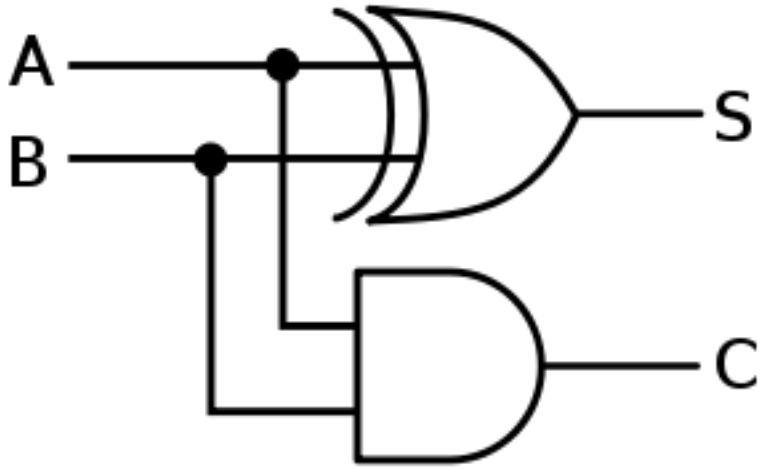
This lab explores two different ways to create a 4 by 4 binary multiplier in Verilog. Broken into two parts, Part 1 will use combinational logic while Part 2 will use sequential logic to create the multiplier. A time-independent digital circuit implemented using logic gates whose output depends solely on its inputs may be considered to be using combinational logic. A digital circuit using sequential logic will have an output that is dependent on the inputs as well as what state the circuit is in. Both programs can take two 4 bit binary numbers and return an 8-bit number that is the result of the two numbers multiplied together. A hierarchical design will be used to implement both programs. A top-level method will be made up of various lower level methods to create the multipliers. Every method will be created and tested individually, where they will eventually be linked together by a higher-level module. Using a hierarchical design, the top-level module will be neat and easy understand; given two 4-bit inputs, the 8-bit output will be the two inputs multiplied.

The lab serves as an introduction for writing modules in Verilog as well as creating test benches in Verilog to verify the modules are working through simulation. The waveforms from the simulations will be analyzed to show the module is working as intended.

Part 1: Combinational Logic Multiplier

Half Adder

Schematic/Logic Diagram:



Truth Table

Inputs		Outputs	
A	B	C_{out}	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
Logic Equations: C_{out} = A & B Sum = A $\oplus$ B			

Source Code

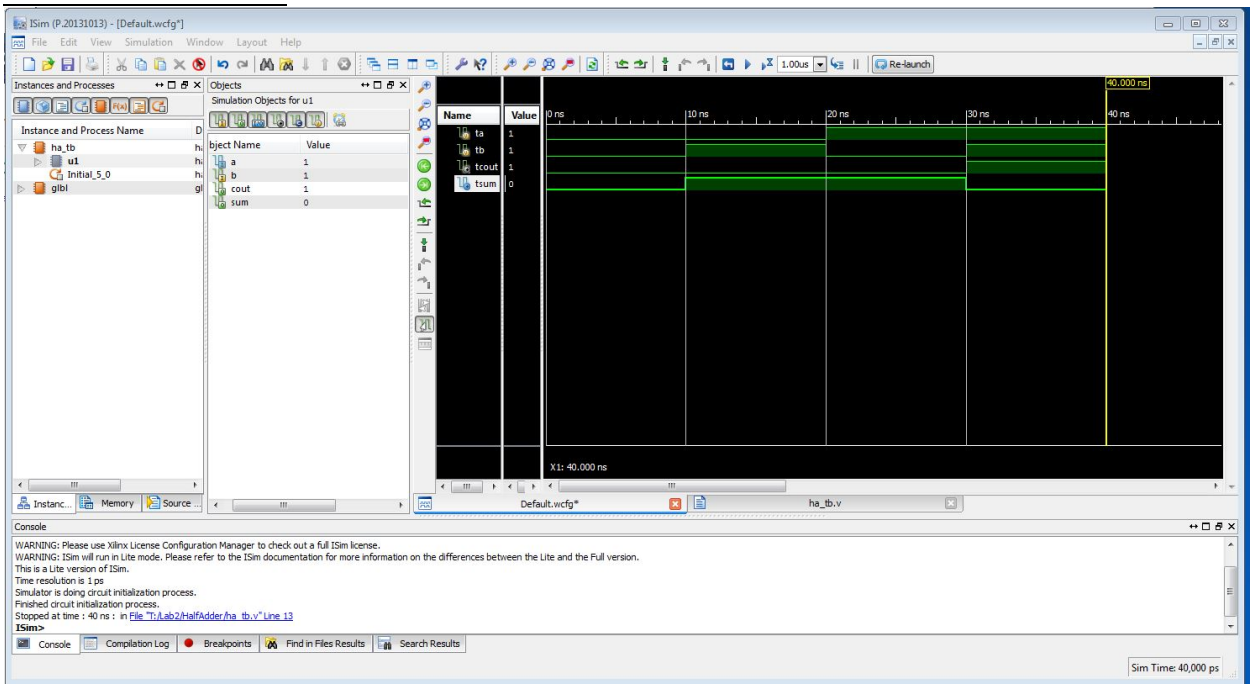
```
--Auth
--Date:
--File Name: halfadder.v
--Purpose of Code: Show the half adder logic.
--Project Part Number: Part 1 – A
```

```
module halfadder(sout,cout,a,b);
output sout,cout;
input a,b;
assign sout=a^b;
assign cout=(a&b);
endmodule
```

```
--Auth
--Date:
--File Name: halfadder_tb.v
--Purpose of Code: Test the half adder logic.
--Project Part Number: Part 1 – A
```

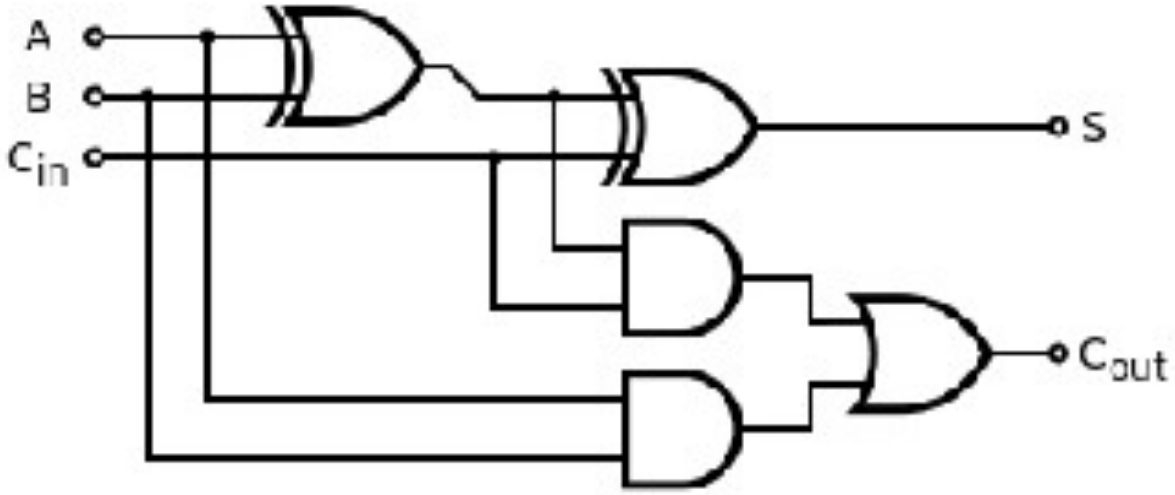
```
module halfadder_tb; //test batch
reg ta, tb;
wire tcout, tsum;
halfadder u1 (.a(ta), .b(tb), .cout(tcout), .sum(tsum)); //name association
initial
begin
        ta = 0; tb = 0;
        #10; ta = 0; tb = 1;
        #10;
        ta = 1; tb = 0;
        #10;
        ta = 1; tb = 1;
        #10 $stop;
end
endmodule
```

Simulation Waveform



Full Adder

Schematic/Logic Diagram:



Truth Table

Inputs			Outputs	
A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1
Logic Equations: Sum = $A \oplus B \oplus C_{in}$ $C_{out} = (A \oplus B)\, C_{in} + A \,\&\, B$				

Source Code

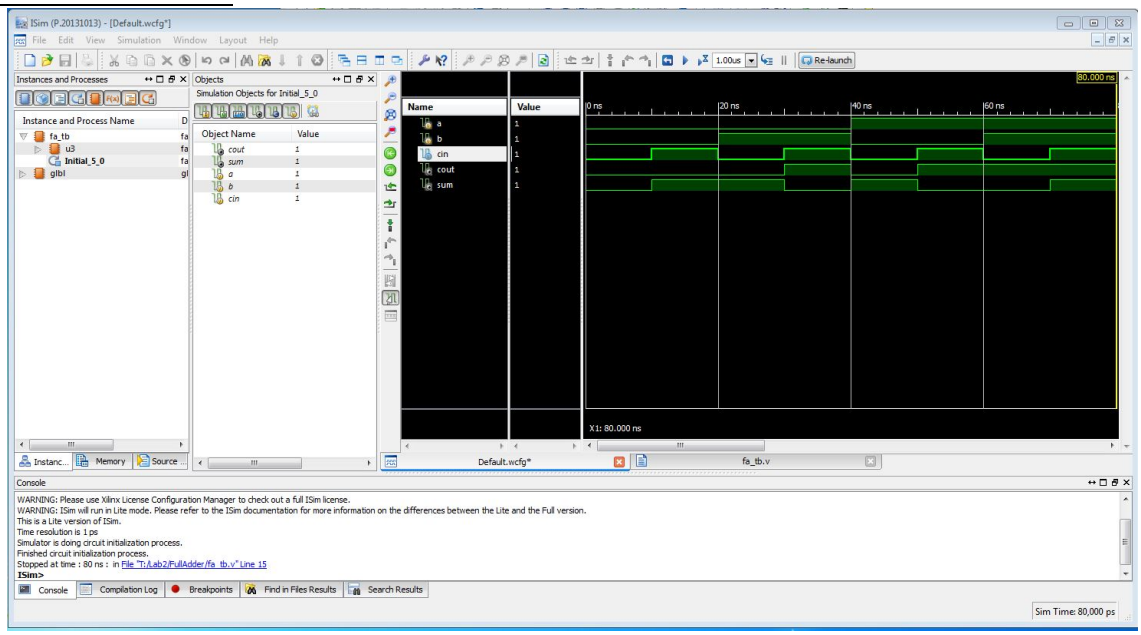
```
--Auth
--Date
--File Name: fulladder.v
--Purpose of Code: Show the full adder logic.
--Project Part Number: Part 1 – B
module fulladder(sout,cout,a,b,cin);
output sout,cout;
input a,b,cin;
assign sout=(a^b^cin);
```

```
assign cout=((a&b)|(a&cin)|(b&cin));
endmodule
```

```
--Auth
--Date
--File Name: fulladder_tb.v
--Purpose of Code: Test the full adder logic.
--Project Part Number: Part 1 –B
```

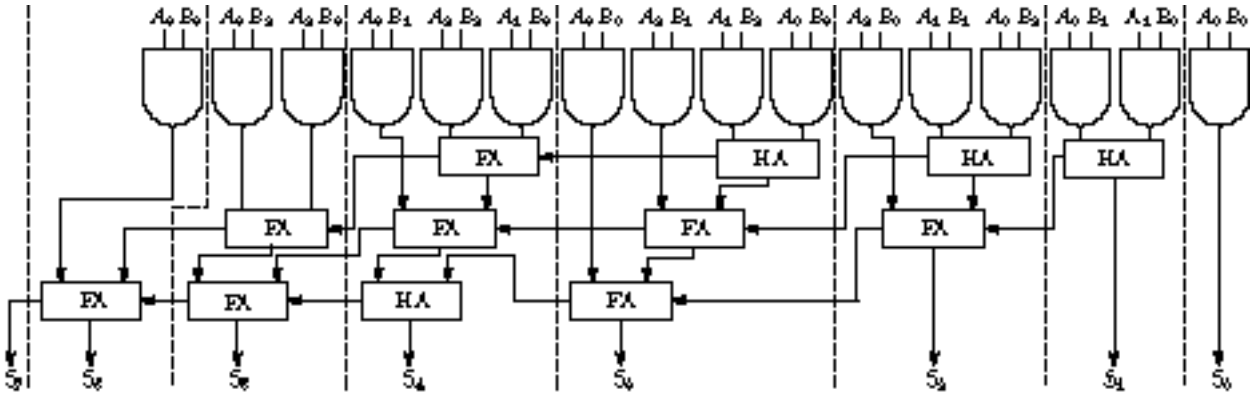
```
module fulladder_tb;
reg a, b, cin;
wire cout, sum;
fulladder u3(.a(a), .b(b), .cin(cin), .cout(cout), .sum(sum));
initial
begin
        a = 0; b = 0; cin = 0;
        #10 a = 0; b = 0; cin = 1;
        #10; a = 0; b = 1; cin = 0;
        #10; a = 0; b = 1; cin = 1;
        #10; a = 1; b = 0; cin = 0;
        #10; a = 1; b = 0; cin = 1;
        #10; a = 1; b = 1; cin = 0;
        #10; a = 1; b = 1; cin = 1;
        #10 $stop;
end
endmodule
```

Simulation Waveform



Four by Four Multiplier

Schematic/Logic Diagram:



Truth Table

Inputs		Outputs
A[3:0]	B[3:0]	Pro[7:0]
0000	0000	00000000
0001	0001	00000001
0010	0010	00000100
0011	0011	00001001
0100	0100	00010000
0101	0101	00011001
0110	0110	00110001
0111	0111	00110001
1000	1000	01000000
1001	1001	01010001
1010	1010	01100100
1011	1011	01111001
1100	1100	10010000
1101	1101	10101001
1110	1110	11000100
1111	1111	11100001

Source Code

--Authc
--Date:
--File Name: multiply4bits.v

```
--Purpose of Code: Show the four by four combinational multiplexer logic.
--Project Part Number: Part 1 – C
```

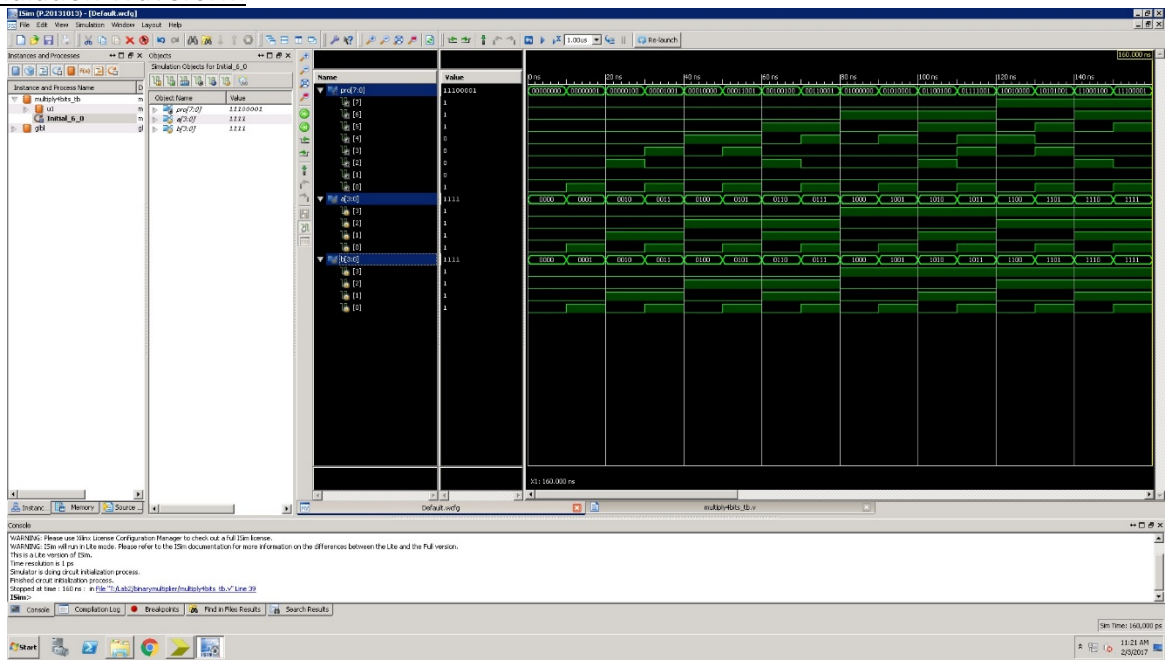
```
module multiply4bits(pro,a,b);
output [7:0]pro;
input [3:0]a;
input [3:0]b;
assign pro[0]=(a[0] & b[0]);
wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17;
halfadder halfadder1(pro[1],x1,(a[1] & b[0]),(a[0] & b[1]));
fulladder fulladder1(x2,x3,a[1]&b[1],(a[0]&b[2]),x1);
fulladder fulladder2(x4,x5,(a[1]&b[2]),(a[0]&b[3]),x3);
halfadder halfadder2(x6,x7,(a[1]&b[3]),x5);
halfadder halfadder3(pro[2],x15,x2,(a[2]&b[0]));
fulladder fulladder5(x14,x16,x4,(a[2]&b[1]),x15);
fulladder fulladder4(x13,x17,x6,(a[2]&b[2]),x16);
fulladder fulladder3(x9,x8,x7,(a[2]&b[3]),x17);
halfadder halfadder4(pro[3],x12,x14,(a[3]&b[0]));
fulladder fulladder8(pro[4],x11,x13,(a[3]&b[1]),x12);
fulladder fulladder7(pro[5],x10,x9,(a[3]&b[2]),x11);
fulladder fulladder6(pro[6],pro[7],x8,(a[3]&b[3]),x10);
endmodule
```

```
--Auth
--Date:
--File Name: multiply4bits_tb.v
--Purpose of Code: test the four by four combinational multiplexer logic.
--Project Part Number: Part 1 – C
```

```
module multiply4bits_tb();
reg [3:0]a;
reg [3:0]b;
wire [7:0]pro;
multiply4bits u1(.a(a), .b(b), .pro(pro));
initial
begin
        a = 4'b0000; b = 4'b0000;
        #10;
        a = 4'b0001; b = 4'b0001;
        #10;
        a = 4'b0010; b = 4'b0010;
        #10;
        a = 4'b0011; b = 4'b0011;
        #10;
```

```
        a = 4'b0100; b = 4'b0100;
        #10;
        a = 4'b0101; b = 4'b0101;
        #10;
        a = 4'b0110; b = 4'b0110;
        #10;
        a = 4'b0111; b = 4'b0111;
        #10;
        a = 4'b1000; b = 4'b1000;
        #10;
        a = 4'b1001; b = 4'b1001;
        #10;
        a = 4'b1010; b = 4'b1010;
        #10;
        a = 4'b1011; b = 4'b1011;
        #10;
        a = 4'b1100; b = 4'b1100;
        #10;
        a = 4'b1101; b = 4'b1101;
        #10;
        a = 4'b1110; b = 4'b1110;
        #10;
        a = 4'b1111; b = 4'b1111;
        #10; $stop;
end
endmodule
```

Simulation Waveform



Part 2: Sequential Multiplier

Adder

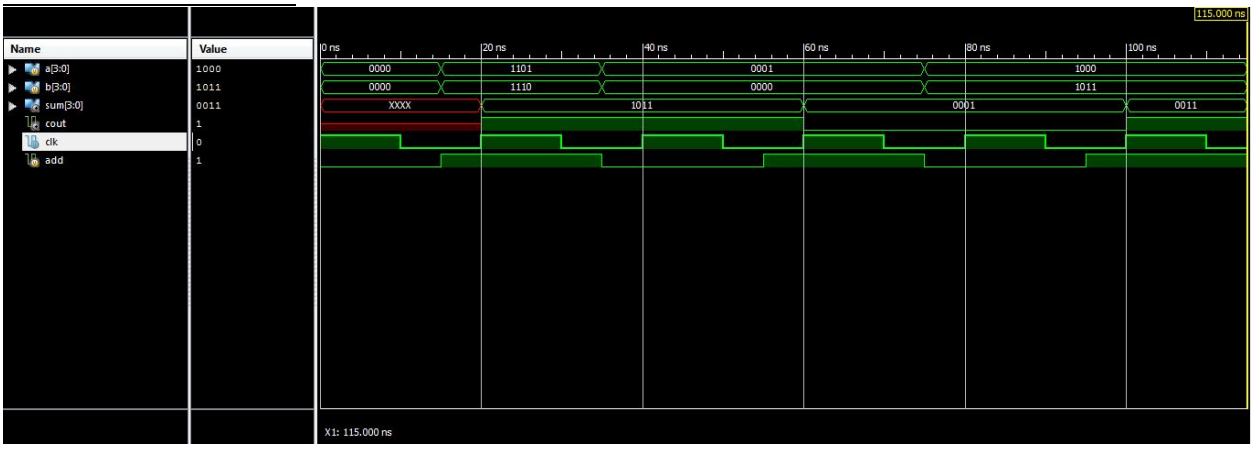
Source Code:

```
/*
Autho
Date:
File Name: adder.v
Purpose of Code: adds registers a and b together
Project Part Number: Part 2a
*/
module adder (a, b, cout, sum, clk, add);
input clk, add;
input [3:0]a, b;
output [3:0]sum;
output cout;
reg cout;
reg [3:0]sum;
always@(posedge clk)
begin
if (add)
        {cout, sum} <= a + b;
end
endmodule

/*
Auth
Date
File Name: adder_tb.v
Purpose of Code: test adder.v
Project Part Number: Part 2a
*/
`timescale 1ns / 1ps
module adder_tb;
reg [3:0] a;
reg [3:0] b;
reg clk;
reg add;
wire cout;
wire [3:0]sum;
adder u1 (.a(a), .b(b), .cout(cout), .sum(sum), .clk(clk), .add(add));
initial clk = 1;
always #10 clk = ~clk;
```

```
initial
begin
        add = 0;
        a = 4'b0000; b = 4'b0000;
        #15;
        add = 1;
        a = 4'b 1101; b = 4'b 1110;
        #20;
        add = 0;
        a = 4'b 0001; b = 4'b 0000;
        #20;
        add = 1;
        #20;
        add = 0;
        a = 4'b1000; b = 4'b1011;
        #20;
        add = 1;
        #20 $stop;
end
endmodule
```

Simulation Waveform:



D Flip-Flop A

Source Code:

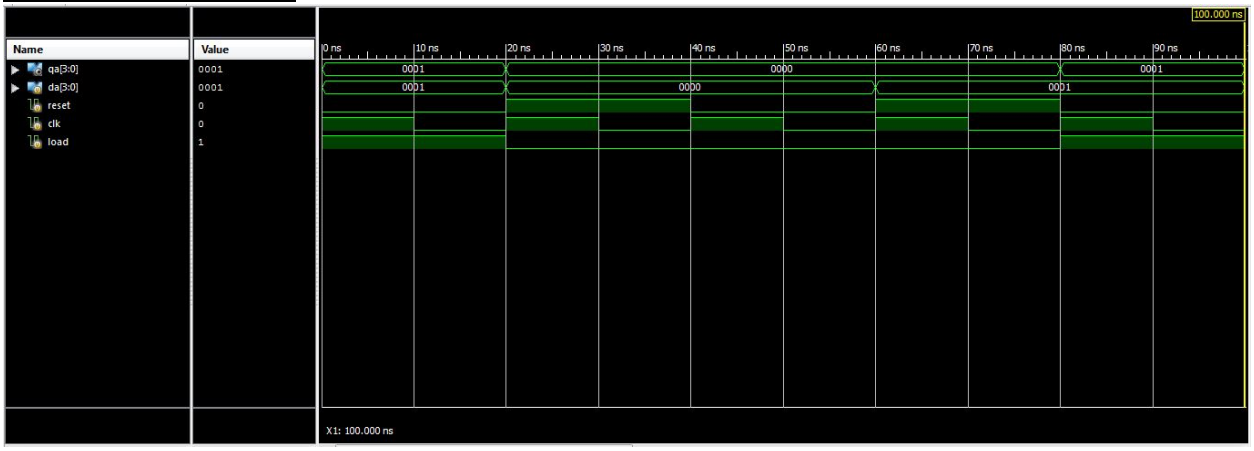
```
/*
Autho
Date:
File Name: dffa.v
Purpose of Code: load a number into the A register
Project Part Number: Part 2b
*/
```

```
module dffa ( reset, clk, load, da, qa);
input   reset, clk, load;
input [3:0]da;
output [3:0]qa;
reg [3:0]qa;
always@(posedge clk or posedge reset)
begin
    if (reset)
        qa <= 4'b0000;
    else if (load)
        qa <= da;
end
endmodule

/*
Auth
Date
File Name: dffa_tb.v
Purpose of Code: test flip flop
Project Part Number: Part 2b
*/
`timescale 1ns / 1ps
module dffa_tb;
reg reset;
reg clk;
reg load;
reg [3:0]da;
wire [3:0]qa;
dffa u1(.reset(reset), .clk(clk), .load(load), .da(da), .qa(qa));
always
begin
        clk = 1'b1;
        #10;
        clk = 1'b0;
        #10;
end
initial
begin
        load = 1'b1;
        da = 1'b1;
        reset = 1'b0;
        #20;
        load = 1'b0;
        da = 1'b0;
```

```
        reset = 1'b1;
        #20;
        load = 1'b0;
        da = 1'b0;
        reset = 1'b0;
        #20;
        load = 1'b0;
        da = 1'b1;
        reset = 1'b1;
        #20;
        load = 1'b1;
        da = 1'b1;
        reset = 1'b0;
        #20 $stop;
end
endmodule
```

Simulation Waveform:



D Flip-Flop B

Source Code:

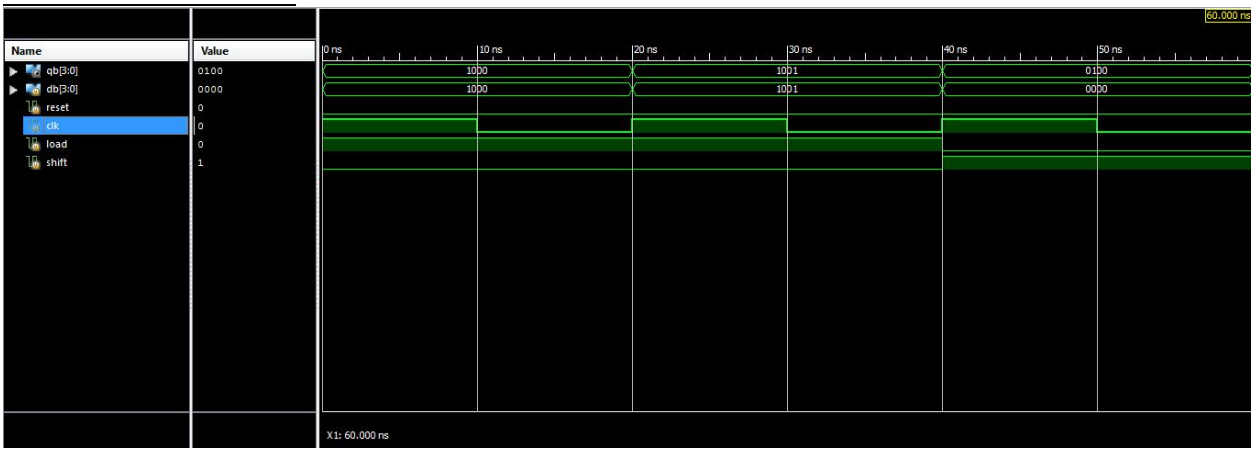
```
/*
Authc
Date:
File Name: dffb.v
Purpose of Code: acts as a d flip flop to shift register to the right one bit
Project Part Number: Part 2c
*/
module dffb(reset, clk, load, shift, db, qb);
input reset, clk, load, shift;
input [3:0]db;
output [3:0]qb;
```

```
reg [3:0]qb;
always@(posedge clk or posedge reset)
begin
    if (reset)
        qb <= 4'b0000;
    else if (load)
        qb <= db;
    else if (shift)
        qb <= { 1'b0, qb[3:1] };
end
endmodule

/*
Auth
Date
File Name: dffb_tb.v
Purpose of Code: test dffb.v
Project Part Number: Part 2c
*/
`timescale 1ns / 1ps
module dffb_tb;
reg clk;
reg reset;
reg load;
reg shift;
reg [3:0] db;
wire [3:0] qb;
dffb u1(.clk(clk), .reset(reset), .load(load), .shift(shift), .db(db), .qb(qb));
always
begin
        clk = 1'b1;
        #10;
        clk = 1'b0;
        #10;
end
initial
begin
        reset = 0;
        load = 1'b1;
        shift = 0;
        db = 4'b1000;
        #20;
        reset = 0;
        load = 1'b1;
```

```
        shift = 0;
        db = 4'b1001;
        #20;
        reset = 0;
        load = 0;
        shift = 1'b1;
        db = 0;
        #20     $stop;
        end
endmodule
```

Simulation Waveform:



Finite State Machine

Source Code:

```
/*
Auth
Date    ,   ,
File Name: fsm.v
Purpose of Code: shows the finite state machine and the steps that it will take in order
                        to load, shift, add, and clear registers for the sequential multiplier.
Project Part Number: Part 2d
*/
module fsm(reset, clk, clr, load_ab, load_p, shf_b, shf_p, en_add);
input   reset, clk;
output clr, load_ab, load_p, shf_b, shf_p, en_add;
reg             clr, load_ab, load_p, shf_b, shf_p, en_add;
reg[4:0]        cs, nexts;
parameter               s_clr = 5'b00000,
                        s_load_ab = 5'b00001,
                        s_en_add1 =             5'b00010,
                        s_load_p1 =             5'b00011,
```

```
                        s_shf_p1 =          5'b00100,
                        s_shf_b1 =          5'b00101,
                        s_en_add2 =         5'b00110,
                        s_load_p2 =         5'b00111,
                        s_shf_p2 =          5'b01000,
                        s_shf_b2 =          5'b01001,
                        s_en_add3 =         5'b01010,
                        s_load_p3 =         5'b01011,
                        s_shf_p3 =          5'b01100,
                        s_shf_b3 =          5'b01101,
                        s_en_add4 =         5'b01110,
                        s_load_p4 =         5'b01111,
                        s_shf_p4 =          5'b10000,
                        s_wait =            5'b10001;
always @ (posedge clk or posedge reset)
begin
        if(reset) cs <= s_clr;
        else cs <= nexts;
end
always @ (cs)
begin
        case(cs)
                s_clr    :                  nexts = s_load_ab;
                s_load_ab :                 nexts = s_en_add1;
                s_en_add1 :                 nexts = s_load_p1;
                s_load_p1 :                 nexts = s_shf_p1;
                s_shf_p1 :                  nexts = s_shf_b1;
                s_shf_b1 :                  nexts = s_en_add2;
                s_en_add2 :                 nexts = s_load_p2;
                s_load_p2 :                 nexts = s_shf_p2;
                s_shf_p2 :                  nexts = s_shf_b2;
                s_shf_b2 :                  nexts = s_en_add3;
                s_en_add3 :                 nexts = s_load_p3;
                s_load_p3 :                 nexts = s_shf_p3;
                s_shf_p3 :                  nexts = s_shf_b3;
                s_shf_b3 :                  nexts = s_en_add4;
                s_en_add4 :                 nexts = s_load_p4;
                s_load_p4 :                 nexts = s_shf_p4;
                s_shf_p4:                   nexts = s_wait;
                s_wait:                     nexts = s_wait;
                default :                   nexts = s_clr;
        endcase
end
always @ (cs)
```

```
begin
        case(cs)
                s_clr:
                begin
                        clr = 1;
                        load_ab = 0;
                        load_p = 0;
                        shf_b = 0;
                        shf_p = 0;
                        en_add = 0;
                end
                s_load_ab:
                begin
                        clr = 0;
                        load_ab = 1;
                        load_p = 0;
                        shf_b = 0;
                        shf_p = 0;
                        en_add = 0;
                end
                s_en_add1:
                begin
                        clr = 0;
                        load_ab = 0;
                        load_p = 0;
                        shf_b = 0;
                        shf_p = 0;
                        en_add = 1;
                end
                s_load_p1:
                begin
                        clr = 0;
                        load_ab = 0;
                        load_p = 1;
                        shf_b = 0;
                        shf_p = 0;
                        en_add = 0;
                end
                s_shf_p1:
                begin
                        clr = 0;
                        load_ab = 0;
                        load_p = 0;
                        shf_b = 0;
```

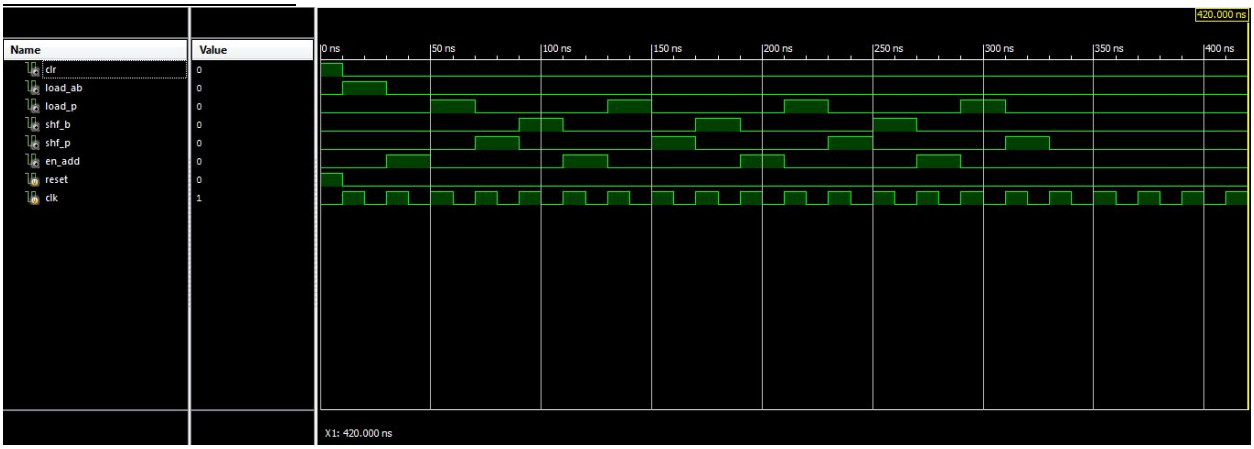
```
        shf_p = 1;
        en_add = 0;
end
s_shf_b1:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 1;
        shf_p = 0;
        en_add = 0;
end
s_en_add2:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 0;
        shf_p = 0;
        en_add = 1;
end
s_load_p2:
begin
        clr = 0;
        load_ab = 0;
        load_p = 1;
        shf_b = 0;
        shf_p = 0;
        en_add = 0;
end
s_shf_p2:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 0;
        shf_p = 1;
        en_add = 0;
end
s_shf_b2:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
```

```
        shf_b = 1;
        shf_p = 0;
        en_add = 0;
end
s_en_add3:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 0;
        shf_p = 0;
        en_add = 1;
end
s_load_p3:
begin
        clr = 0;
        load_ab = 0;
        load_p = 1;
        shf_b = 0;
        shf_p = 0;
        en_add = 0;
end
s_shf_p3:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 0;
        shf_p = 1;
        en_add = 0;
end
s_shf_b3:
begin
        clr = 0;
        load_ab = 0;
        load_p = 0;
        shf_b = 1;
        shf_p = 0;
        en_add = 0;
end
s_en_add4:
begin
        clr = 0;
        load_ab = 0;
```

```
                load_p = 0;
                shf_b = 0;
                shf_p = 0;
                en_add = 1;
            end
            s_load_p4:
            begin
                clr = 0;
                load_ab = 0;
                load_p = 1;
                shf_b = 0;
                shf_p = 0;
                en_add = 0;
            end
            s_shf_p4:
            begin
                clr = 0;
                load_ab = 0;
                load_p = 0;
                shf_b = 0;
                shf_p = 1;
                en_add = 0;
            end
            s_wait:
            begin
                clr = 0;
                load_ab = 0;
                load_p = 0;
                shf_b = 0;
                shf_p = 0;
                en_add = 0;
            end
            default:
            begin
                clr = 0;
                load_ab = 0;
                load_p = 0;
                shf_b = 0;
                shf_p = 0;
                en_add = 0;
            end
        endcase
end
endmodule
```

```
/*
Auth
Date:
File Name: fsm_tb.v
Purpose of Code: test fsm.v
Project Part Number: Part 2d
*/
`timescale 1ns / 1ps
module fsm_tb;
	reg reset, clk;
	wire clr, load_ab, load_p, shf_b, shf_p, en_add;
	fsm u1 (.reset(reset), .clk(clk), .clr(clr), .load_ab(load_ab), .load_p(load_p),
.shf_b(shf_b), .shf_p(shf_p), .en_add(en_add));
	initial clk = 0;
	always #10 clk = ~clk;
	initial
	begin
		reset = 1;
		#10;
		reset = 0;
		#10;
		#400 $stop;
	end
endmodule
```

Simulation Waveform:



Multiplier

Source Code:

```
/*
Auth
Date:
File Name: mult.v
```

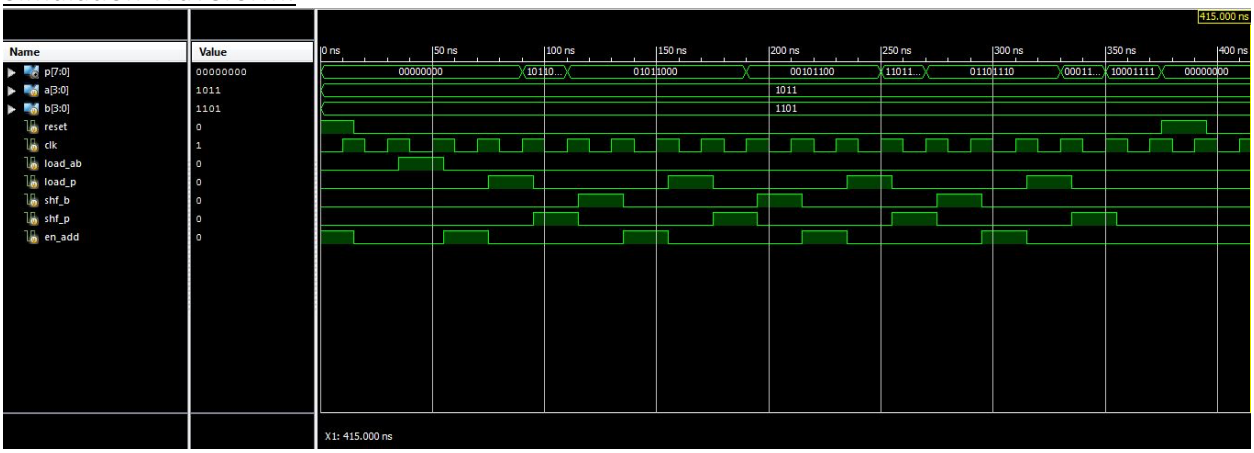
```
Purpose of Code: This design of the sequential multiplier connects dffa,dffb,mux,adder,and
pregister.
Project Part Number: Part 2e
*/
module mult(a, b, p, reset, clk, load_ab, load_p, shf_b, shf_p, en_add);
input    reset, clk, load_ab, load_p, shf_b, shf_p, en_add;
input[3:0]a, b;
output[7:0]p;
wire[3:0]z, x, y, m, n;
wire r;
dffa u1(.reset(reset), .clk(clk), .load(load_ab), .da(a), .qa(x));
dffb u2(.reset(reset), .clk(clk), .load(load_ab), .shift(shf_b), .db(b), .qb(z));
mux u3(.a(x), .b(z[0]), .y(y));
adder u4(.a(n), .b(y), .sum(m), .cout(r), .clk(clk), .add(en_add));
preg u5(.clk(clk), .reset(reset), .load(load_p), .shift(shf_p), .cin(r), .din_p(m), .p(p), .ph(n));
endmodule

/*
Auth
Date
File Name: mult_tb.v
Purpose of Code: This design of the sequential multiplier connects dffa,dffb,mux,adder,and
pregister.
Project Part Number: Part 2e
*/
`timescale 1ns / 1ps
module mult_tb;
reg [3:0]a;
reg [3:0]b;
reg reset;
reg clk;
reg load_ab;
reg load_p;
reg shf_b;
reg shf_p;
reg en_add;
wire [7:0] p;
mult u1 (.a(a), .b(b), .p(p), .reset(reset), .clk(clk), .load_ab(load_ab), .load_p(load_p),
.shf_b(shf_b), .shf_p(shf_p), .en_add(en_add));
initial clk = 1;
always #10 clk = ~clk;
initial
begin
        a = 4'B1011;
```

```
b = 4'b1101;
reset = 1;
clk = 0;
load_ab = 0;
load_p = 0;
shf_b = 0;
shf_p = 0;
en_add = 1;
#15;
reset = 0;
en_add = 0;
#20 load_ab = 1;
#20 load_ab = 0;
en_add = 1;
#20 en_add = 0;
load_p = 1;
#20 load_p = 0;
shf_p = 1;
#20 shf_p = 0;
shf_b = 1;
#20 shf_b = 0;
en_add = 1;
#20 en_add = 0;
load_p = 1;
#20 load_p = 0;
shf_p = 1;
#20 shf_p = 0;
shf_b = 1;
#20 shf_b = 0;
en_add = 1;
#20 en_add = 0;
load_p = 1;
#20 load_p = 0;
shf_p = 1;
#20 shf_p = 0;
shf_b = 1;
#20 shf_b = 0;
en_add = 1;
#20 en_add = 0;
load_p = 1;
#20 load_p = 0;
shf_p = 1;
#20 shf_p = 0;
#20 reset = 1;
```

```
        #20 reset = 0;
        #20 $stop;
        end
endmodule
```

Simulation Waveform:



Mux

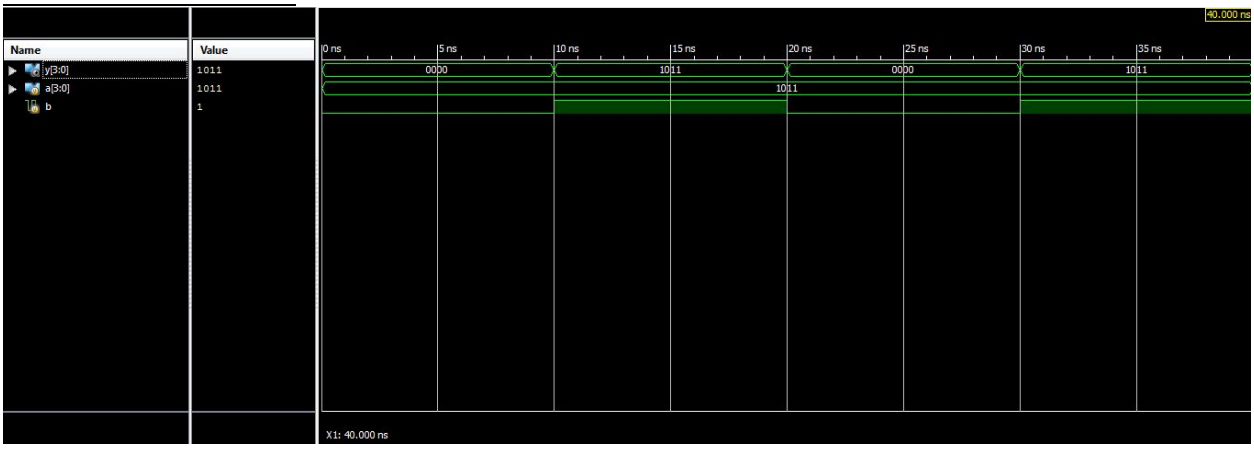
Source Code:

```
/*
Auth
Date:
File Name: mux.v
Purpose of Code: chooses the value for y depending on whether the value is 1 or a 0.
Project Part Number: Part 2f
*/
module mux(a, b, y);
input b;
input[3:0]a;
output[3:0]y;
reg[3:0]y;
wire[3:0]z;
assign z = 4'b0000;
always@ (a or b or z)
begin
        case(b)
                0 : y = z;
                1 : y = a;
        endcase
end
endmodule
```

```
/*


File Name: mux_tb.v
Purpose of Code: tests mux
Project Part Number: Part 2f
*/
`timescale 1ns / 1ps
module mux_tb;
reg [3:0]a;
reg b;
wire [3:0]y;
mux u1 (.a(a), .b(b), .y(y));
initial
begin
        a = 4'b1011; b = 0;
        #10;
        b = 1;
        #10;
        b = 0;
        #10;
        b = 1;
        #10 $stop;
end
endmodule
```

Waveform Simulation:



P-Register

Source Code:

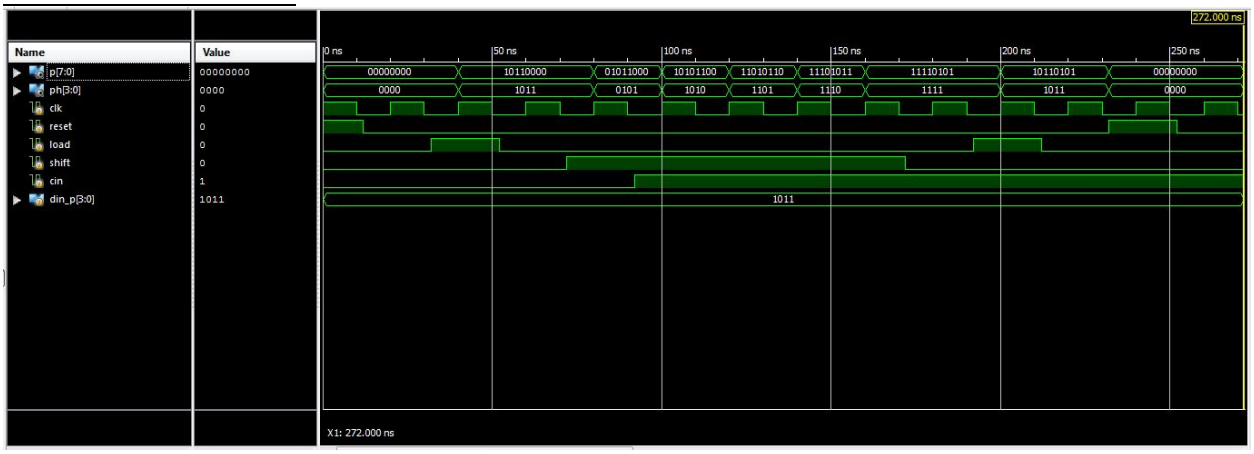
```
/*
```

```
File Name: preg.v
Purpose of Code: The p-register stores the value of the multiplier.
Project Part Number: Part 2g
*/
module preg(clk, reset, load, shift, cin, din_p, p, ph);
input clk, reset, load, shift, cin;
input[3:0]din_p;
output[7:0]p;
output[3:0]ph;
reg[3:0]ph, pl;
always@(posedge clk or posedge reset)
begin
	if (reset)
		ph <= 4'h0;
	else if (load)
		ph <= din_p;
	else if (shift)
		ph <= { cin , ph[3:1] };
end
always@(posedge clk or posedge reset)
begin
	if (reset)
		pl <= 4'h0;
	else if (shift)
		pl <= { ph[0] , pl[3:1] };
end
assign p = {ph, pl};
endmodule

/*
Auth
Date
File Name: preg_tb.v
Purpose of Code: tests preg.v
Project Part Number: Part 2g
*/
`timescale 1ns / 1ps
module preg_tb;
reg clk;
reg reset;
reg load;
reg shift;
reg cin;
reg [3:0] din_p;
```

```
wire [7:0] p;
wire [3:0] ph;
preg u1(.clk(clk), .reset(reset), .load(load), .shift(shift), .cin(cin), .din_p(din_p), .p(p), .ph(ph));
initial clk = 1;
always #10 clk = ~clk;
initial
begin
        load = 0;
        reset = 1;
        shift = 0;
        cin = 0;
        din_p = 4'b1011;
        #12;
        reset = 0;
        #20 load = 1;
        #20 load = 0;
        #20 shift = 1;
        #20 cin = 1;
        #80 shift = 0;
        #20 load = 1;
        #20 load = 0;
        #20 reset = 1;
        #20 reset = 0;
        #20; $stop;
end
endmodule
```

Waveform Simulation:



Top Module

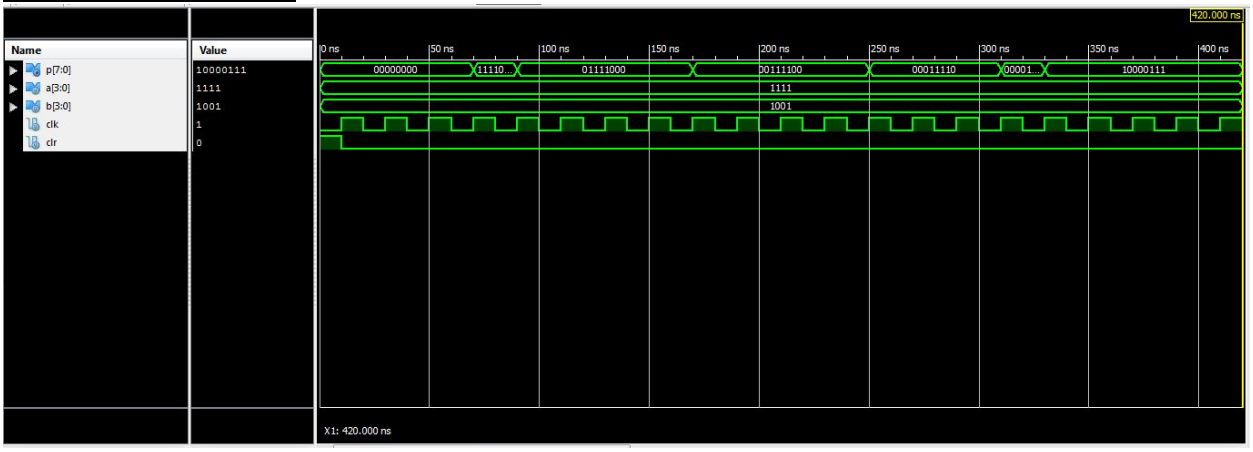
Source Code:

```
/*
Auth
Date
File Name: top.v
Purpose of Code: Combines all the previous Verilog codes into one top module.
Project Part Number: Part 2h
*/
module top(a, b, p, clk, clr);
input[3:0]      a, b;
input           clk, clr;
output[7:0]     p;
wire            j, k, m, n, o, q;
//create one instance for finite state machine
fsm f1(.reset(clr), .clk(clk), .clr(q), .load_ab(j), .load_p(k), .shf_b(m), .shf_p(n), .en_add(o));
//create one instance for multiplier logic circuit
mult m1(.a(a), .b(4'b1101), .p(p), .reset(q), .clk(clk), .load_ab(j), .load_p(k), .shf_b(m), .shf_p(n),
.en_add(o) );
endmodule

/*
Auth
Date
File Name: top_tb.v
Purpose of Code: test top.v
Project Part Number: Part 2h
*/
`timescale 1ns / 1ps
//testbench for top.v
module top_tb;
        // Inputs
        reg [3:0] a;
        reg [3:0] b;
        reg clk;
        reg clr;
        // Outputs
        wire [7:0] p;
        // Instantiate the Unit Under Test (UUT)
        top uut (
                .a(a),
                .b(b),
                .p(p),
                .clk(clk),
```

```
            .clr(clr)
            );
    initial clk = 0;
    always #10 clk = ~clk;
    initial
    begin
    a = 4'b1111;
    b = 4'b1001;
    clr = 1;
    #10;
    clr = 0;
    #10;
    #400;
    $stop;
    end
endmodule
```

Waveform Simulation:



Part 3: LCD

Source Code:

```
module lcd(clk, sf_ce0, lcd_rs, lcd_rw, lcd_e, lcd_4, lcd_5, lcd_6, lcd_7);
 parameter              k = 18;
 input            clk;      // synthesis attribute PERIOD clk "50 MHz"
 reg   [k+8-1:0]          count;
 output         sf_ce0;    // high for full LCD access
 reg            sf_ce0;

 output         lcd_e, lcd_rs, lcd_rw, lcd_7, lcd_6, lcd_5, lcd_4;
 reg            lcd_e, lcd_rs, lcd_rw, lcd_7, lcd_6, lcd_5, lcd_4;

 reg            lcd_busy;
 reg            lcd_stb;
 reg      [5:0] lcd_code;
 reg      [6:0] lcd_stuff;

 always @ (posedge clk) begin
  count     <= count + 1;
  lcd_busy <= 1'b1;
  sf_ce0    <= 1;             // StrataFlash Chip Enable
  case (count[k+7:k+2])
    0: lcd_code <= 6'h03;      // power-on initialization
    1: lcd_code <= 6'h03;
    2: lcd_code <= 6'h03;
    3: lcd_code <= 6'h02;
    4: lcd_code <= 6'h02;      // function set
    5: lcd_code <= 6'h08;
    6: lcd_code <= 6'h00;      // entry mode set
    7: lcd_code <= 6'h06;
    8: lcd_code <= 6'h00;      //  display on/off control
    9: lcd_code <= 6'h0C;
   10: lcd_code <= 6'h00;      // display clear
   11: lcd_code <= 6'h01;
   12: lcd_code <= 6'h24;      // C
   13: lcd_code <= 6'h23;
   14: lcd_code <= 6'h25;      // P
   15: lcd_code <= 6'h20;
   16: lcd_code <= 6'h24;      // E
   17: lcd_code <= 6'h25;
   18: lcd_code <= 6'h23;      // 1
   19: lcd_code <= 6'h21;
   20: lcd_code <= 6'h23;      // 6
```

```
    21: lcd_code <= 6'h26;
    22: lcd_code <= 6'h23;       // 6
    23: lcd_code <= 6'h26;

    default: lcd_code <= 6'h10;
  endcase
  lcd_stb <= ^count[k+1:k+0] & ~lcd_rw & lcd_busy;
  lcd_stuff <= {lcd_stb,lcd_code};
  {lcd_e,lcd_rs,lcd_rw,lcd_7,lcd_6,lcd_5,lcd_4} <= lcd_stuff;
 end
endmodule
```

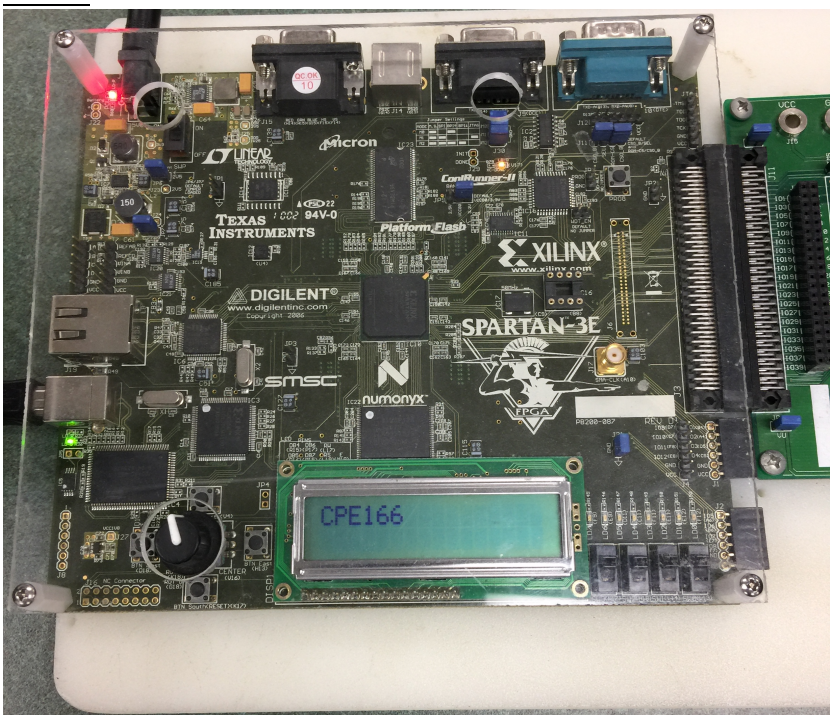
User Constraint File

```
NET "sf_ce0" LOC = "D16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "clk" LOC = "C9" | IOSTANDARD = LVCMOS33 ;

NET "lcd_e" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "lcd_rs" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "lcd_rw" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;

NET "lcd_4" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "lcd_5" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "lcd_6" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "lcd_7" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
```

Demo:



Conclusion

This lab overall served as an introduction to the Verilog programming language. The code required making logic gate assignments, creating time dependent modules such as d flip flops, linking modules together in hierarchical design and using case statements to create a finite state machine. Test benches for all the modules also needed to be designed in Verilog. Inputs and outputs needed to be declared a certain way, the tested module needed to be called in a certain fashion. For the sequential multiplier, a clock needed to be made for the simulation, as well as learning how to make the simulation wait certain periods before assigning more inputs. The concepts of combinational design versus sequential were also explored. The advantage of a combinational design is that is does not require a clock, but they can be messy to organize. The advantage of a sequential design is that while they require a clock, managing the connected modules is an easier task.