CPE 166 ADVANCED LOGIC DESIGN LAB

Lab 2. Carry Select Adder Design, Sequential Multiplier Design & Multiplexed Seven Segment Displays

Mariya Shtevnina

CPE 166 – 02 (Wednesdays at 1400)
Instructor: Carmi, Eric David

Table of Contents

Introduction

The lab is spilt into three parts: the carry select adder, the sequential multiplier, and the segment display. The purpose of the lab is to learn hierarchical design strategy using Verilog, carry select adder design, sequential shift/add multiplication algorithm, how to use multiplexed seven segment display, how to simulate a Verilog program, and how to download an application program into the FPGA board. Most importantly, I learn how to write code in Verilog for practical purposes and use the FPGA in conjunction with Xilinx.

In this second lab, I hope to learn about the program process, such as designing the program using schematics and logic equations, writing the code, testing the code, and finally downloading the code unto hardware. The lab starts with a simple concept of adding, multiplying, and displaying, yet the real challenge is getting familiar with the Verilog language and the application program, Xilinx.

Important concepts this lab requires is the d flip-flop, a multiplexer, combinational logic, sequential logic, and finite state machine. The d flip-flop is essential for holding data, the multiplexer for choosing data, and the finite state machine for outputting data. Without all these tools, the three parts would be very difficult to implement.

Part I – 8-Bit Carry Select Adder Design

1.1 Design Purpose: The purpose of part I is to create a Verilog based program that adds two 4-bit numbers into one 8-bit number using carry ripple adders and multiplexers. The two ripple carry adders use the half adder module and the full adder module. After, the multiplexer chooses which ripple carry adder to use. In this part, I use the method of creating code for small functions and using the small function code for a larger function code; For example, using the half adder for the full adder.

The design creation goes as follows: half adder, full adder, 4-bit ripple carry adder, multiplexer, and finally the 8-bit carry select adder.

1.2 Engineering Data: First, I must create the design of the half adder by constructing a truth table, deriving the logic equation, and drawing out the schematic. Table 1 contains the truth table and the logic equation, while Figure 1 is the schematic and symbol of the half adder.

Table 1

Inputs		Outputs	
a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
Logic Equation: Cout = a b Sum = a XOR b			

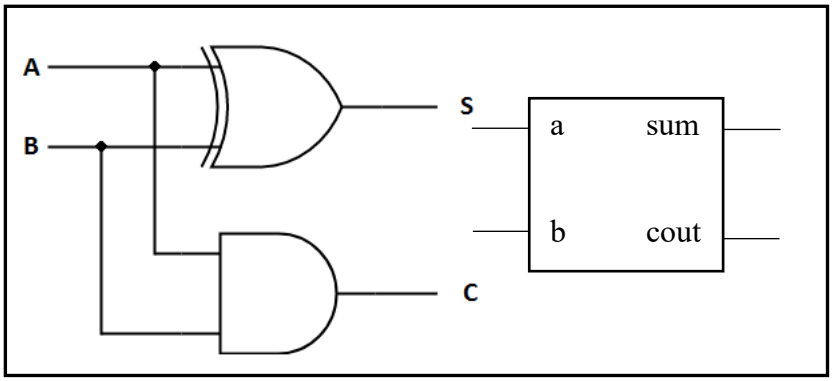


Figure 1

Next, from the half adder, I create the full adder. The following table, table 2, is the truth table and the logic equations for the full adder.

Inputs			Outputs	
a	b	cin	cout	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 xor b xor cin cout = (a xor b)				

Table 2

The fuller adder is essentially two half adders with a cin and an OR gate. Therefore, for the schematic of the full adder, I use the two half adders I already created. Figure 2 shows two schematics on how the two half adders are used in the full adder.

Figure 3

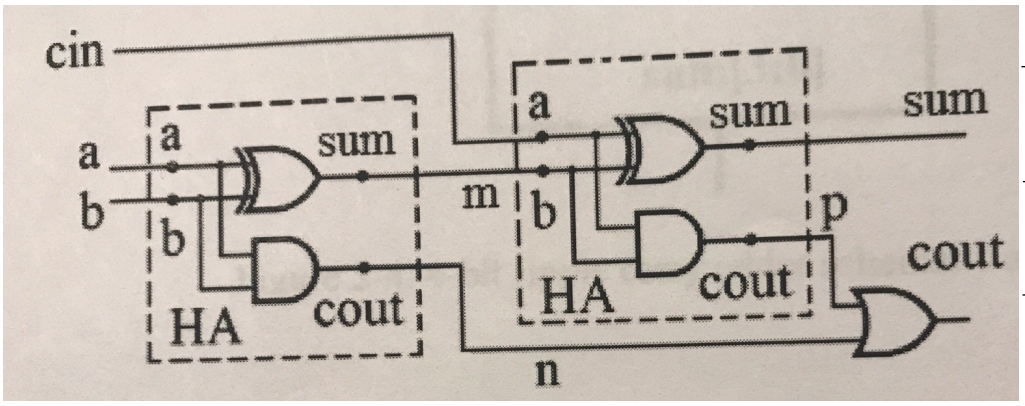
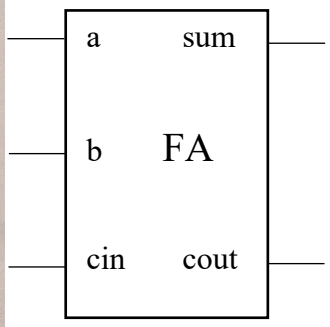
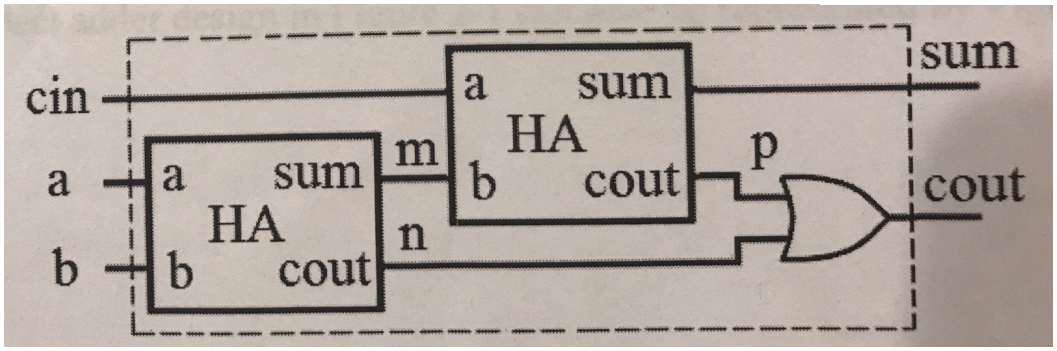


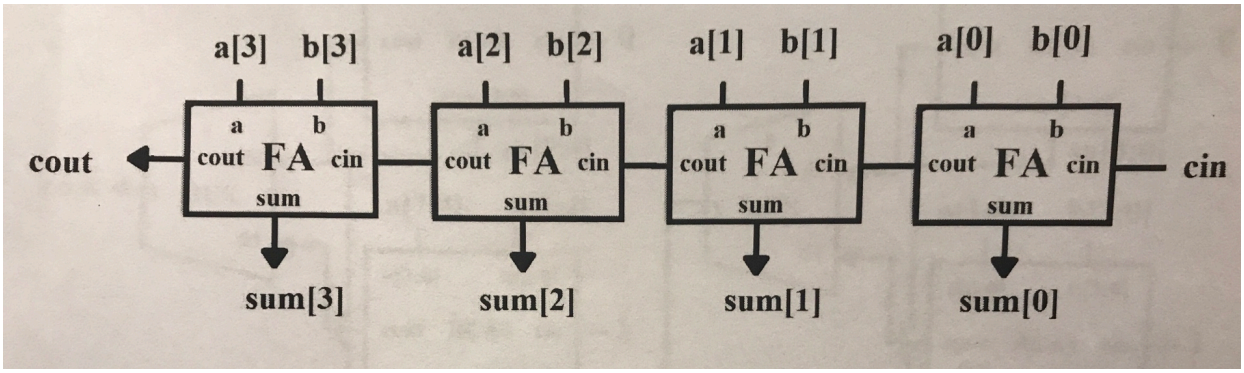
Figure 2, where there is three inputs and two outputs.

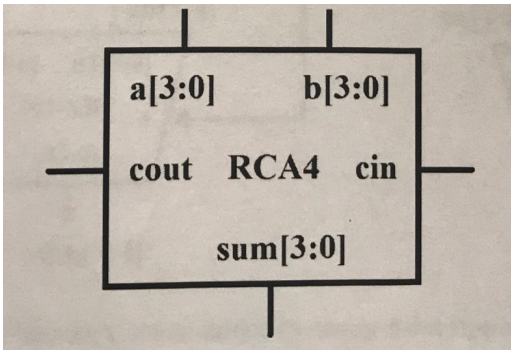




The full adder is then used to create the ripple carry adder. Since the full adder can only add two one-bit numbers together, I need to use four full adders to add two 4-bit numbers together. As shown in figure 4, the schematic of the RCA4 is four instances of the full adder.

Figure 4





One important tool that I use for the final project is a multiplexer, as shown in figure 5. The multiplexer receives two different pieces of data yet only outputs one, the one that the input, s, selects. The inputs and outputs a, b, and y, can more than one bit, for example in the 8-bit adder, I use the multiplexer for four-bit inputs and a four-bit output. Table 3 and 4 are truth tables for a multiplexer with 4-bits and single bit multiplexer.

Figure 5

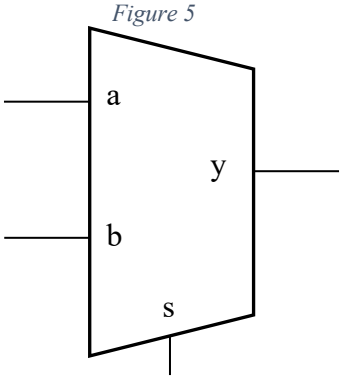


Table 3

Input	Output
s	y[3:0]
0	a[3:0]
1	b[3:0]

Table 4

Input	Output
s	y
0	a
1	b

Finally, I use the 4-bit ripple carry adder and a multiplexer to create the 8-bit carry select adder. The final product using four RCA4's, and four multiplexers. Each wire is shown in the schematic, figure 6, is important to add to the Verilog code. Any duplicate of a chip is using the same code, but each chip is called an instance.

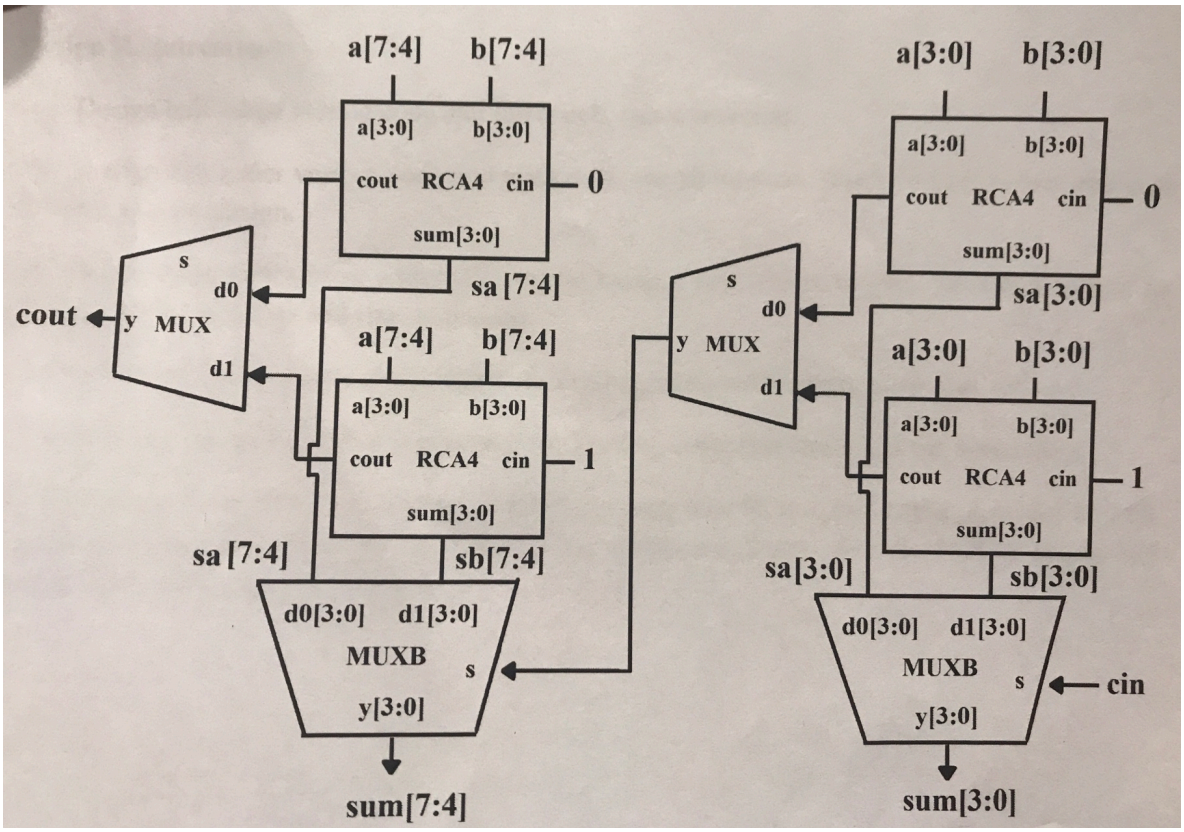


Figure 6

1.3 Source Code:

```
module ha(a, b, cout, sum);
input  a, b;
output cout, sum;

assign cout = a & b;
assign sum = a ^ b;

 xor   g1 (sum, a, b);
 and   g2 (cout, a, b);
endmodule
```

Code 1, Half adder - Verilog.

```
module ha_tb;
reg    a,  b;
wire   cout, sum;


ha  u1 ( a, b, cout, sum );



initial begin
      {a, b} = 2'b00;
  #10 {a, b} = 2'b01;
  #10 {a, b} = 2'b10;
  #10 {a, b} = 2'b11;
  #10 $stop;
end

initial $monitor($time, "ns, a=%b, b=%b, cout=%b, sum=%b", a, b, cout, sum);

endmodule
```

Code 2, Half adder testbench - Verilog

```
module fa(a, b, cin, cout, sum);
input    a, b, cin;
output  cout, sum;

wire  m, n, p;


ha    g1 (.cout(n), .sum(m), .a(a), .b(b) );
ha    g2 (.cout(p), .sum(sum), .a(cin), .b(m) );

assign cout = p | n;

endmodule
```

Code 3, Full adder - Verilog

```
module fa_tb;
reg       a,  b, cin;
wire     cout, sum;


fa   u1 ( a, b, cin, cout, sum );



initial begin
      {a, b, cin} = 3'b000;
  #10 {a, b, cin} = 3'b001;
  #10 {a, b, cin} = 3'b01_0;
  #10 {a, b, cin} = 3'b011;
  #10 {a, b, cin} = 4;
  #10 {a, b, cin} = 5;
  #10 {a, b, cin} = 6;
  #10 {a, b, cin} = 7;
  #10 $stop;
end

initial $monitor($time, "ns, a=%b, b=%b, cin = %b, cout=%b, sum=%b", a, b, cin, cout, sum);

endmodule
```

Code 4, Full adder testbench – Verilog

```
module rca4(a, b, cin, cout, sum);
input [3:0]   a, b;
input         cin;

output  [3:0]  sum;
output         cout;

wire     [2:0]  m;


fa    g1 (.cout(m[0]), .sum(sum[0]), .a(a[0]), .b(b[0]), .cin(cin));
fa    g2 (.cout(m[1]), .sum(sum[1]), .a(a[1]), .b(b[1]), .cin(m[0]));
fa    g3 (.cout(m[2]), .sum(sum[2]), .a(a[2]), .b(b[2]), .cin(m[1]));
fa    g4 (.cout(cout), .sum(sum[3]), .a(a[3]), .b(b[3]), .cin(m[2]));
endmodule
```

Code 5, Ripple Carry Adder 4-bit - Verilog

```
module rca4_tb;
reg  [3:0]  a,b;
reg         cin;
wire [3:0]  sum;
wire        cout;
wire [4:0]  res;

assign res = { cout, sum };

rca4   uu ( a, b, cin, cout, sum );



initial begin
      a = 2;  b= 4;  cin = 0;
  #10 a = 4;  b = 4; cin = 1;
  #10 a=  5;  b =6;  cin = 1;
  #10 a=  7;  b = 7; cin = 1;

  #10 $stop;
end

initial $monitor($time, "ns, a=%d, b=%d, cin = %d, addition result = %d", a, b, cin, res);
endmodule
```

Code 6, Ripple Carry Adder 4-bit Testbench - Verilog

```
module mux2to1(d1, d0, s, y);
input    d1, d0, s;
output   y;
reg      y;

always@(d1 or d0 or s)
begin
   if (s)   y = d1;
   else     y = d0;
end

endmodule
```

Code 7, 1-bit multiplexer - Verilog

```
module mux2to1_tb;
reg       d1, d0, s;
wire      y;

mux2to1   u1 (d1, d0, s, y);




initial begin
       d1 = 0;  d0 = 0;  s = 0;
  #10  d1 = 0;  d0 = 0;  s = 1;
  #10  d1 = 0;  d0 = 1;  s = 0;
  #10  d1 = 0;  d0 = 1;  s = 1;
  #10  d1 = 1;  d0 = 0;  s = 0;
  #10  d1 = 1;  d0 = 0;  s = 1;
  #10  d1 = 1;  d0 = 1;  s = 0;
  #10  d1 = 1;  d0 = 1;  s = 1;
  #10 $stop;
end

initial $monitor($time, "ns, d1=%b, d0=%b, s = %b,  y=%b", d1, d0, s, y);
endmodule
```

Code 8, 1-bit multiplexer testbench - Verilog

```
module muxb(d1, d0, s, y);
 input [3:0]    d1, d0, s;
 output[3:0]    y;
 reg   [3:0]    y;

always@(d1 or d0 or s)
begin
    if (s)  y = d1;
    else    y = d0;
end
endmodule
```

Code 9, 4-bit multiplexer - Verilog

```
module muxb_tb;
reg   [3:0]    d1, d0, s;
wire  [3:0]    y;

muxb   u1 (d1, d0, s, y);

initial begin
       d1 = 0;  d0 = 0;  s = 0;
  #10  d1 = 0;  d0 = 0;  s = 1;
  #10  d1 = 0;  d0 = 1;  s = 0;
  #10  d1 = 0;  d0 = 1;  s = 1;
  #10  d1 = 1;  d0 = 0;  s = 0;
  #10  d1 = 1;  d0 = 0;  s = 1;
  #10  d1 = 1;  d0 = 1;  s = 0;
  #10  d1 = 1;  d0 = 1;  s = 1;
  #10 $stop;
end

initial $monitor($time, "ns, d1=%b, d0=%b, s = %b,  y=%b", d1, d0, s, y);
endmodule
```

Code 10, 4-bit multiplexer testbench - Verilog

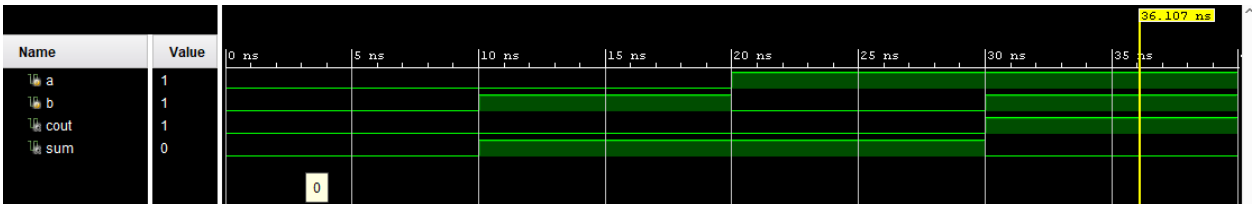
```
module csa8(a, b, cin, sum, cout);
input        cin;
input  [7:0] a, b;
output [7:0] sum;
output       cout;

wire [7:0] sa;
wire [7:0] sb;
wire d11, d01, d12, d02, ytos;

rca4    u1 (.cout(d11), .sum(sb[3:0]), .cin(1), .b(b[3:0]), .a(a[3:0]));
rca4    u2 (.cout(d01), .sum(sa[3:0]), .cin(0), .b(b[3:0]), .a(a[3:0]));
mux2to1 h1 (.y(ytos), .d0(d01), .d1(d11), .s(cin));
muxb    g1 (.y(sum[3:0]), .s(cin), .d0(sa[3:0]), .d1(sa[3:0]));
rca4    u3 (.cout(d02), .sum(sa[7:4]), .a(a[7:4]), .b(b[7:4]), .cin(0));
rca4    u4 (.cout(d12), .sum(sb[7:4]), .a(a[7:4]), .b(b[7:4]), .cin(1));
mux2to1 h2 (.y(cout), .d0(d02), .d1(d12), .s(cin));
muxb    g2 (.y(sum[7:4]), .s(ytos), .d0(sa[7:4]), .d1(sb[7:4]));



endmodule
```

Code 11, 8-bit carry select adder - Verilog

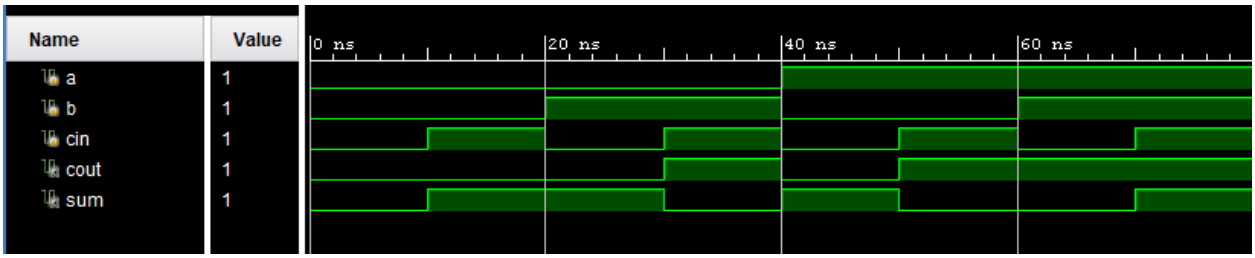
```
module csa8_tb;
reg  [7:0]  a,b;
reg         cin;
wire [7:0]  sum;
wire        cout;
wire [7:0]  res;

assign res = { cin, sum };

csa8   uu1 (a, b, cin, sum, cout);


initial begin
       a = 100;  b= 100;  cin = 0;
  #10  a = 105;  b = 95; cin = 0;
  #10  a=  32;  b =8;  cin = 0;
  #10  a=  7;  b = 7; cin = 0;

  #10 $stop;
end

initial $monitor($time, "ns, a=%d, b=%d, cin = %d, addition result = %d, carryout = %d", a, b, cin, res, cout);

endmodule
```

Code 12, 8-bit carry select adder testbench – Verilog

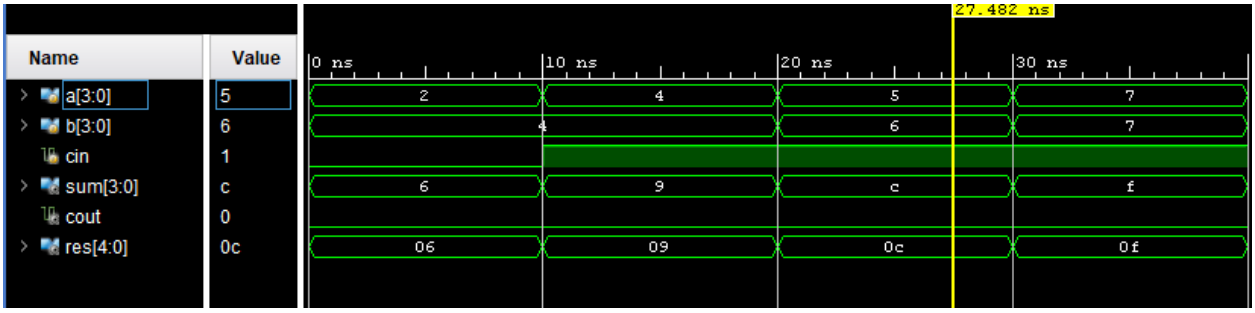
1.4 Simulation Waveform:



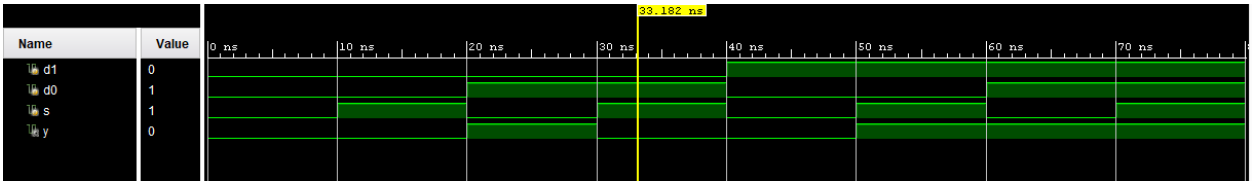
Waveform 1: Half Adder. The two input bits, A and B, are first 0, so the sum is 0. Then only one input is high to the sum is 1. Finally, if both are high then the sum is 0, yet the cout is 1, which means the output is 10, which is correct. Any change occurs at a rising clock edge.



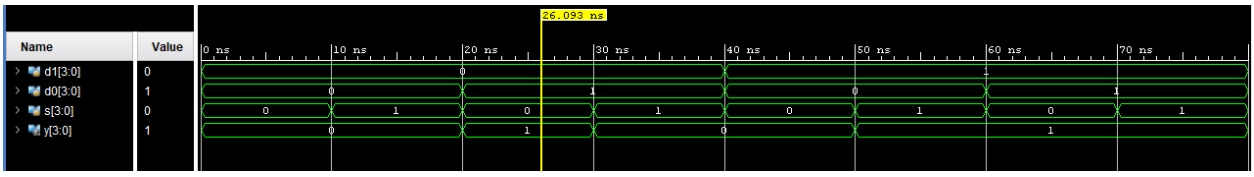
Waveform 2: Full Adder. The full adder adds the cin, a, and b inputs. First all are zero, so the output is zero. Then one of the inputs is high so the least significant bit is high to equal 1, in this case, the sum. If as shown in the end, all inputs are added, then the result should be 11 in binary, or 3 in decimal, where the first 1 is cout and the second is sum. Any change occurs at a rising clock edge.



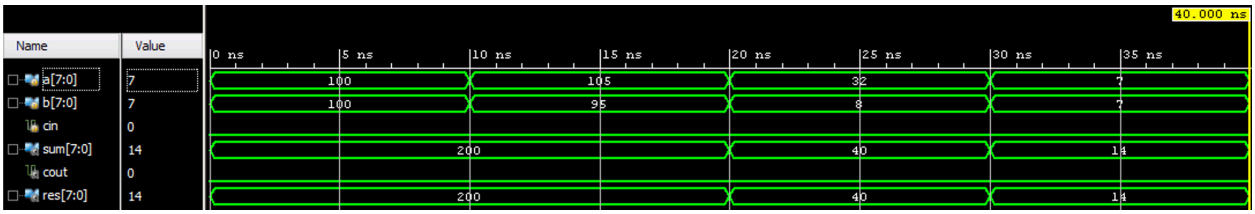
Waveform 3: 4-bit Ripple Carry Adder. Res is the output while a, b, and cin are the inputs. 0c is in hexadecimal and translates to 12 in decimal, which is correct since 5 + 6 + 1 aka cin = 12. The same goes for 7 + 7 + 1 = 0f which is 15 in decimal. Any change occurs at a rising clock edge.



Waveform 4: 1-bit Multiplexer. As shown, if s is 1 then d1 is the output, y, and if s is 0 then the d0 is the output y. Any change occurs at a rising clock edge.



Waveform 5: 4-bit Multiplexer. If s is 1 then d1 is the output, y, and if s is 0 then the d0 is the output y. In this case, the inputs and outputs are 4-bits. Any change occurs at a rising clock edge.

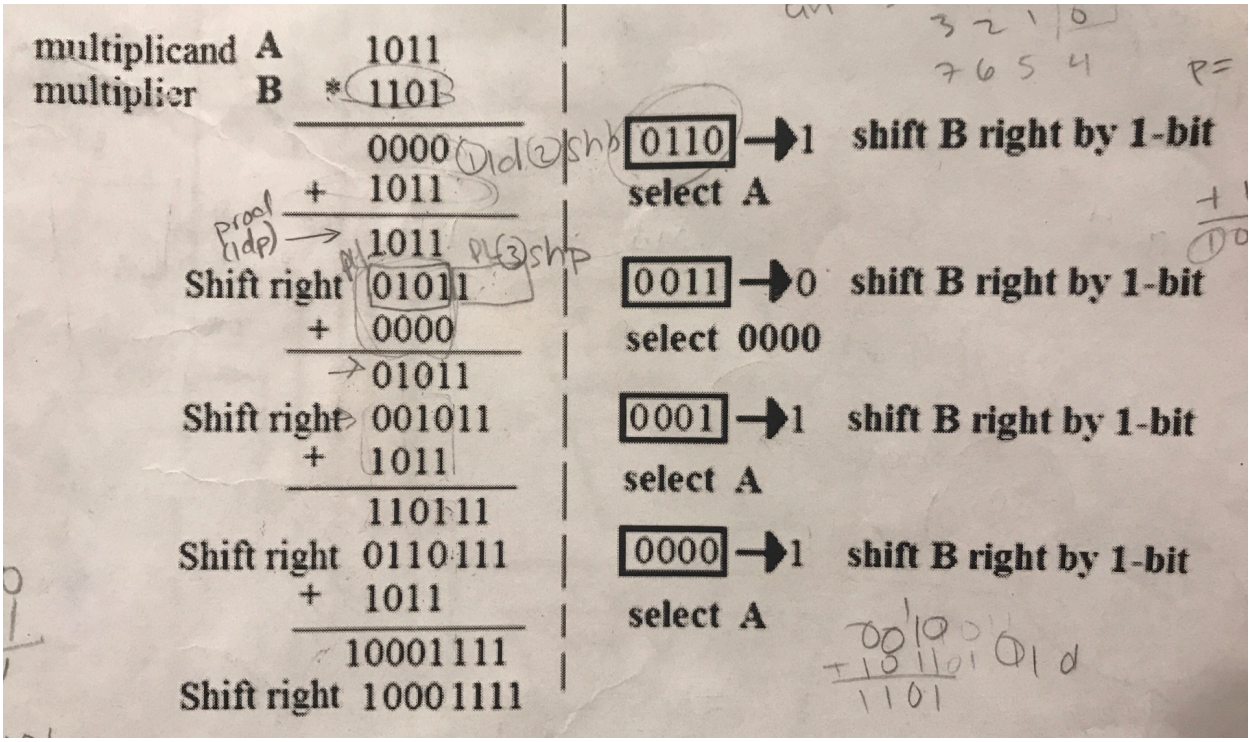


Waveform 6: 8-bit Carry Select Adder. As shown, the inputs add correctly. Res is the sum and a and b are the two 8-bit numbers to add together. Any change occurs at a rising clock edge.

1.5 Result Discussion: The 8-bit Carry Select adder was not too difficult to create. The problem did take a while to solve and most of the time I was getting familiar with the program. The design schematics given by the lab manual was very useful and I would definitely need to create my own for future designs. Sometimes, my code for the ripple carry adder would not work on the simulation but found that in the end simple mistakes such as mistyping a letter in the main code, can create huge issues. Being more detail orientated and fully understanding the problem and my solution helped solve the problems that I faced. Overall, the end code works perfectly, and the problem is solved.

Part II – 4 by 4 Binary Sequential Multiplier Design

2.1 Design Purpose: The purpose of this part is to create a Verilog code that will multiply two 4-bit numbers by long multiplication. The following picture shows how the long multiplication will work.



First, the two 4-bit numbers will load, then one of the inputs would be shifting because the least significant bit is used as a signal for the multiplexer. The multiplexer uses that signal to determine if to add the other input or simply zeros. The sum is shifted to the right and then is used as the new input to add together with the non-shifting input.

The methods used in the program would be a multiplexer, finite state machine, and d-flip-flop.

2.2 Engineering Data: Figure 7 shows the schematic of the whole program. As shown, the finite state machine controls what goes into the mult.

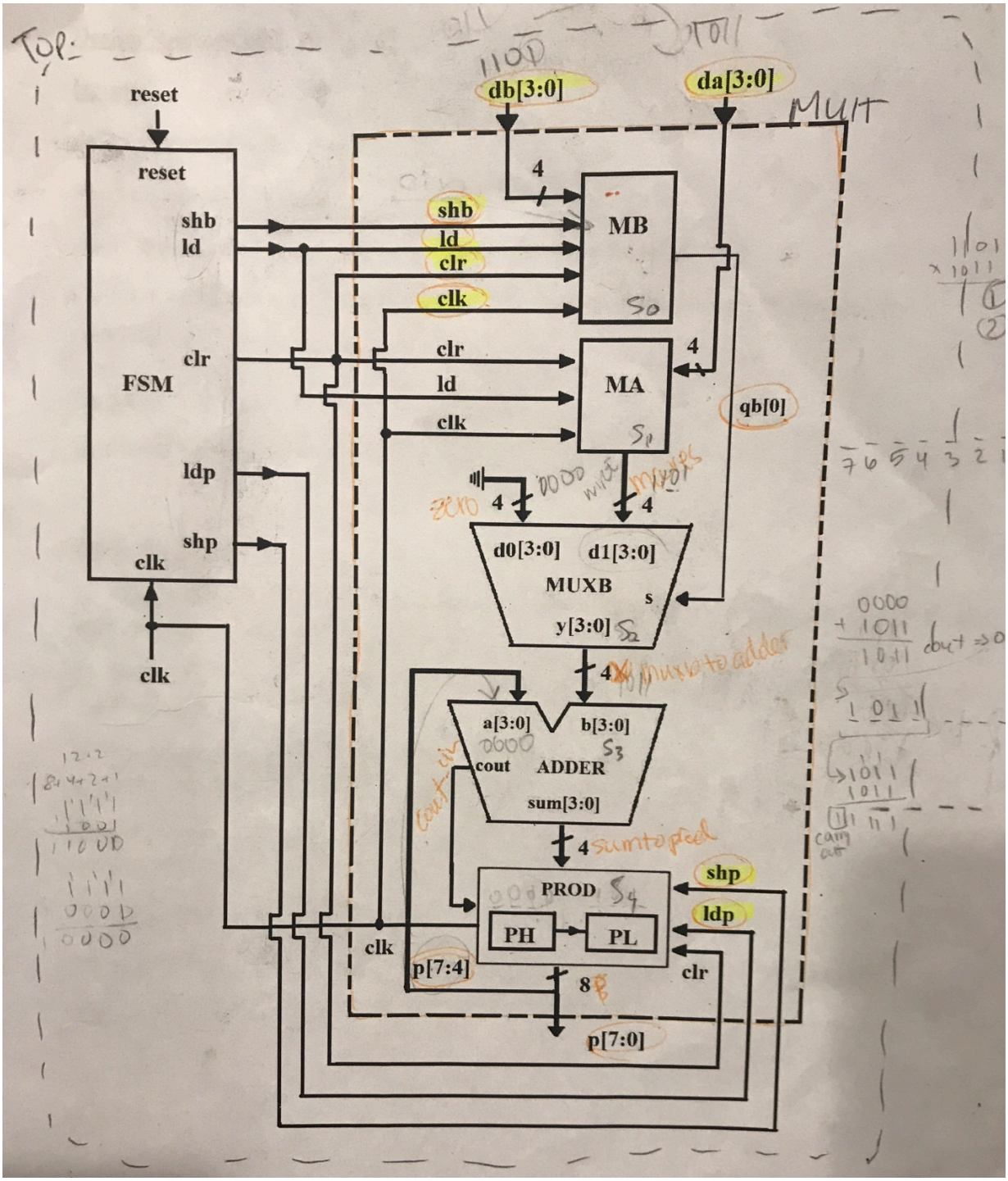
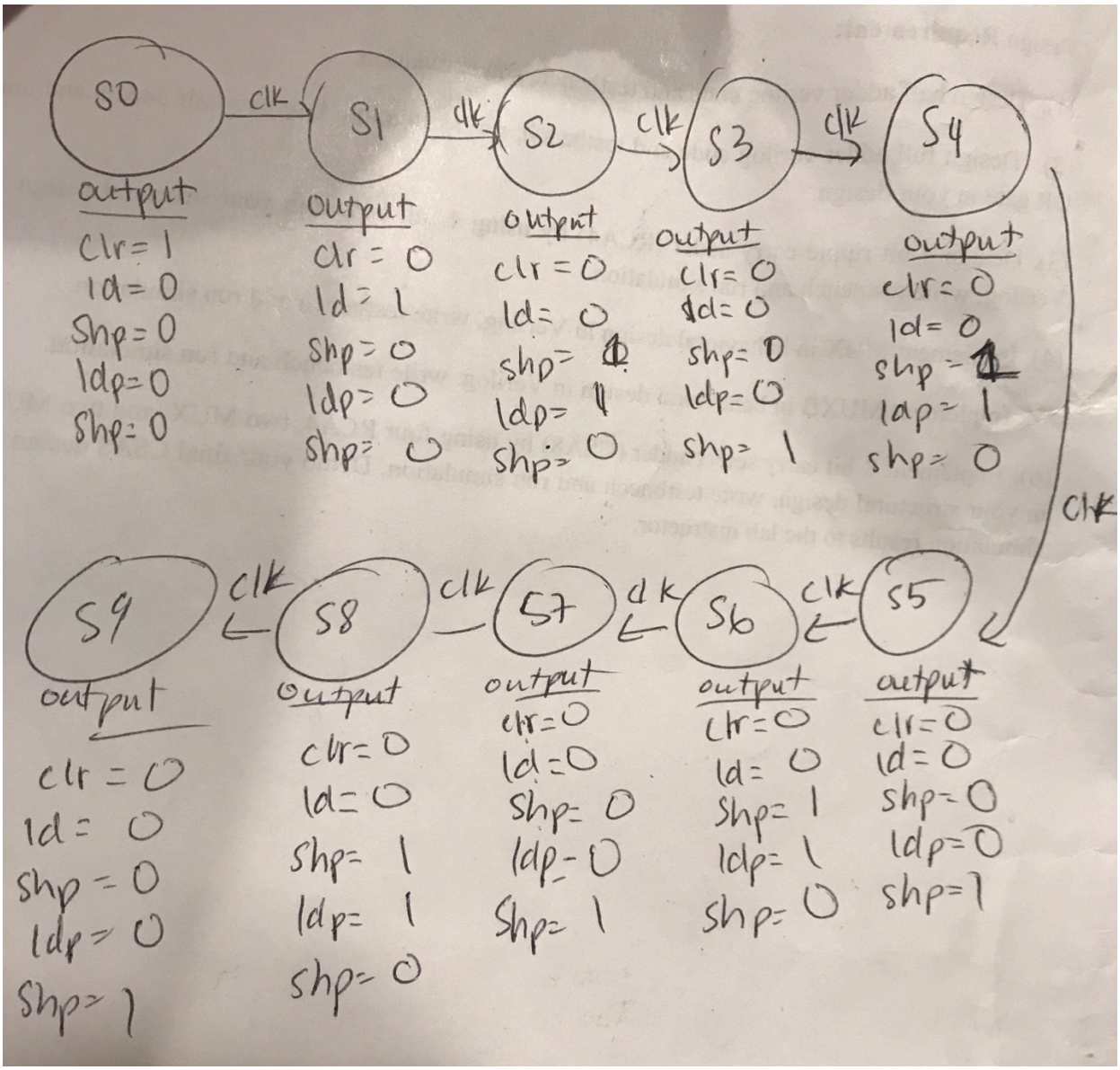


Figure 7



The photo above is my finite state machine diagram.

2.3 Source Code:

```
module ma(da, ld, clr, clk, qa);

input  [3:0] da;
input  ld, clr, clk;
output [3:0] qa;

reg [3:0] qa;

always @ ( posedge clk or posedge clr)
    begin
        if (clr)
            qa <= 0;
        else if(ld)
            qa <= da;
    end

endmodule
```

Code 13: MA, the nonshifting input - Verilog

```
module ma_tb;

reg  [3:0] da;
reg  ld, clr, clk;
wire [3:0] qa;

ma   g1 (.da(da), .ld(ld), .clr(clr), .clk(clk), .qa(qa));

initial clk = 0;
always #10 clk = ~ clk;

initial
begin
    clr = 1;  ld = 0; da = 4'b1011;
    #22 ld = 1;
    #20 clr = 0;
    #60 $stop;
end

endmodule
```

Code 14: MA testbench - Verilog

```verilog
module mb(db, shb, ld, clr, clk, qb);

input  [3:0] db;
input  shb, ld, clr, clk;
reg    [3:0] qb;
output [3:0] qb;

always@(posedge clr or  posedge clk)
begin
    if(clr) qb <= 0;
    else if (ld)
       qb <= db;
    else if (shb)
       qb <= { 1'b0,  qb[3:1] };
end
endmodule
```

Code 15: MB, the shifting input - Verilog

```verilog
module mb_tb;
reg   clk, clr, ld, shb;
reg   [3:0]  db;
wire  [3:0]  qb;

mb uut (db, shb, ld, clr, clk, qb);

initial clk = 0;
always #10 clk = ~ clk;

initial
begin
    clr = 1;  ld = 0;  shb = 0;  db = 4'b1011;
    #10 ld = 1; clr = 0;
    #10 shb = 1; ld = 0;
    //#10   ld = 1; clr = 0;
    #10 shb = 1; ld = 0;
    #10 shb = 1; ld = 0;
    #10 shb = 1; ld = 0;
    #60 $stop;
end
endmodule
```

Code 16: MB testbench - Verilog

```
module muxb(d1, d0, s, y);
input [3:0]    d1, d0, s;
output[3:0]    y;
reg   [3:0]    y;

always@(d1 or d0 or s)
begin
   if (s)  y = d1;
   else    y = d0;
end
endmodule
```

Code 17: 4-bit Multiplexer - Verilog

```
module muxb_tb;
reg   [3:0]    d1, d0, s;
wire  [3:0]    y;

muxb   u1 (d1, d0, s, y);


initial begin
       d1 = 0;  d0 = 0;  s = 0;
  #10  d1 = 0;  d0 = 0;  s = 1;
  #10  d1 = 0;  d0 = 1;  s = 0;
  #10  d1 = 0;  d0 = 1;  s = 1;
  #10  d1 = 1;  d0 = 0;  s = 0;
  #10  d1 = 1;  d0 = 0;  s = 1;
  #10  d1 = 1;  d0 = 1;  s = 0;
  #10  d1 = 1;  d0 = 1;  s = 1;
  #10 $stop;
end

initial $monitor($time, "ns, d1=%b, d0=%b, s = %b,  y=%b", d1, d0, s, y);
endmodule
```

Code 18: 4-bit Multiplexer testbench - Verilog

```
module adder(a, b, cout, sum);

input  [3:0] a, b;
output cout;
output [3:0] sum;

assign { cout, sum[3:0] } = a[3:0] + b[3:0];

endmodule
```

Code 19: 4-bit Adder - Verilog

```
module adder_tb;
reg  [3:0] a, b;
wire cout;
wire [3:0] sum;

adder tt(a, b, cout, sum);

initial begin
     a = 0; b = 0;
     #10 a = 5; b = 5;
     #10 a = 10; b = 2;
     #10 a = 3; b = 4;
     #10 a = 15; b = 1;
     #10 $stop;
end

initial $monitor($time, "ns, a=%b, b=%b, cout = %b,  sum=%b", a, b, cout, sum);

endmodule
```

Code 20: 4-bit Adder testbench - Verilog

```
module prod(shp, ldp, clr, sum, cin, p, clk);

input  shp, ldp, clr, cin, clk;
input  [3:0] sum;
reg    [7:0] p;
reg    [3:0] ph, pl;
output [7:0] p;



always@(posedge clk or posedge clr)
begin
    if (clr) begin
        pl <= 0;
        ph <= 0;
        p <= 0;
    end
    else if (shp) begin
        ph = sum;
        pl = pl >> 1;
        pl[3] = ph[0];
        ph = ph >> 1;
        ph[3] = cin;
        p <= { ph, pl };
     end
    else if (ldp) begin
        p <= { ph, pl };
    end
end
endmodule
```

Code 21: Prod, where if loads data then shifts it - Verilog

```
module prod_tb;
reg   shp, ldp, cin, clk, clr;
reg   [3:0]  sum;
wire  [7:0]  p;

prod uut (shp, ldp, clr, sum, cin, p, clk);

initial clk = 0;
always #10 clk = ~ clk;

initial
begin
    clr = 1;  ldp = 0;  shp = 0;  sum = 4'b1011;
    #20 ldp = 0; clr = 0; cin = 1'b0;
    #20 shp = 1; cin = 1'b1; ldp = 1;
    #20 shp = 1; cin = 1'b1; ldp = 1;
    #20 shp = 1; cin = 1'b1; ldp = 1;
    #20 shp = 1; cin = 1'b1; ldp = 1;
    #20 $stop;
end
initial $monitor($time, "ns, shp=%d, cin=%d, clr=%d, ldp=%d, p=%d", shp, cin, clr, ldp, p);
endmodule
```

Code 22: Prod testbench – Verilog

```
module mult(shb, ld, clr, clk, db, da, shp, ldp, p);

input   shb, ld, clr, clk, shp, ldp;
input   [3:0] db, da;
wire    cout_cin;
wire    [7:0] p;
wire    [3:0] muxb_adder, ma_res, sum_prod, qb;
//reg     [3:0] zero;
output  [7:0] p;


mb      g1 (.qb(qb), .shb(shb), .ld(ld), .clr(clr), .clk(clk), .db(db));
ma      g2 (.qa(ma_res), .ld(ld), .clr(clr), .clk(clk), .da(da));
muxb    g3 (.y(muxb_adder), .d0(4'b0000), .s(qb[0]), .d1(ma_res));
adder   g4 (.sum(sum_prod), .cout(cout_cin), .a(p[7:4]), .b(muxb_adder));
prod    g5 (.p(p), .shp(shp), .ldp(ldp), .clr(clr), .sum(sum_prod), .cin(cout_cin), .clk(clk));

endmodule
```

Code 23: Mult, which connects the chips to have a final product – Verilog

```
module mult_tb;
reg        shb, ld, clr, clk, shp, ldp;
reg  [3:0] db, da;
wire [7:0] p;

mult uut (shb, ld, clr, clk, db, da, shp, ldp, p);


initial clk = 0;
always #10 clk = ~ clk;

initial
begin
      clr = 1; ld = 0; db = 4'b1101; da = 4'b1011; shp = 0; ldp = 0; shb = 0;
              #20 ld = 1; clr = 0;
              #20 ldp = 1; shp = 1; shb = 0; ld = 0; //db = 0110
              #20 ldp = 0; shp = 0; shb = 1;  //db = 0011
              #20 ldp = 1; shp = 1; shb = 0;
              #20 ldp = 0; shp = 0; shb = 1;
              #20 ldp = 1; shp = 1; shb = 0;
              #20 ldp = 0; shp = 0; shb = 1;
              #20 ldp = 1; shp = 1; shb = 0;
      #20 $stop;
end
initial $monitor($time, "ns, shp=%d, ld=%d, clr=%d, ldp=%d, p=%d, shb=%d, db=%d, da=%d", shp, ld, clr, ldp, p, shb, db, da);
endmodule
```

Code 24: Mult testbench - Verilog

```
module fsm(clk, rst, shb, ld, clr, ldp, shp);

input  clk, rst;
output ld, clr, shb, ldp, shp;

reg [3:0]   cs, ns;
reg    ld, clr, shb, ldp, shp;

parameter s0 = 4'b0000, s1=4'b0001, s2=4'b0010, s3=4'b0011, s4=4'b0100, s5=4'b0101, s6=4'b0110, s7 = 4'b0111, s8 = 4'b1000, s9 = 4'b1001;

always @ ( posedge clk or posedge rst)
begin
  if(rst)
       cs <= s0;
  else
       cs <= ns;
end

always @( clk or cs )
begin
   case(cs)
   s0:   ns = s1;

   s1:   ns = s2;

   s2:   ns = s3;

   s3:   ns = s4;

   s4:   ns = s5;

   s5:   ns = s6;

   s6:   ns = s7;

   s7:   ns = s8;

   s8:   ns = s9;

   s9:   ns = s9;

   default: ns = s0;

   endcase
end
```

```

always @( cs )
begin
   case(cs)
   s0:   begin  clr = 1; ld = 0; shp = 0; ldp = 0; shb = 0; end
   s1:   begin  clr = 0; ld = 1; shp = 0; ldp = 0; shb = 0; end
   s2:   begin  clr = 0; ld = 0; shp = 1; ldp = 1; shb = 0; end
   s3:   begin  clr = 0; ld = 0; shp = 0; ldp = 0; shb = 1; end
   s4:   begin  clr = 0; ld = 0; shp = 1; ldp = 1; shb = 0; end
   s5:   begin  clr = 0; ld = 0; shp = 0; ldp = 0; shb = 1; end
   s6:   begin  clr = 0; ld = 0; shp = 1; ldp = 1; shb = 0; end
   s7:   begin  clr = 0; ld = 0; shp = 0; ldp = 0; shb = 1; end
   s8:   begin  clr = 0; ld = 0; shp = 1; ldp = 1; shb = 0; end
   s9:   begin  clr = 0; ld = 0; shp = 0; ldp = 0; shb = 0; end
   default: begin  clr = 0; ld = 0; shp = 0; ldp = 0; shb = 0; end

   endcase
end
endmodule

```

Code 25: The Finite State Machine - Verilog

```
module top(clk, rst, da, db, y);
input        clk, rst;
input [3:0]  da, db;
output [7:0] y;
wire         sp, l, sb, clear, lp;

fsm     u1(.shp(sp), .ld(l), .shb(sb), .clr(clear), .ldp(lp), .clk(clk), .rst(rst));
mult    u2(.p(y), .shp(sp), .ld(l), .shb(sb), .clr(clear), .clk(clk), .db(db), .da(da), .ldp(lp));

endmodule
```

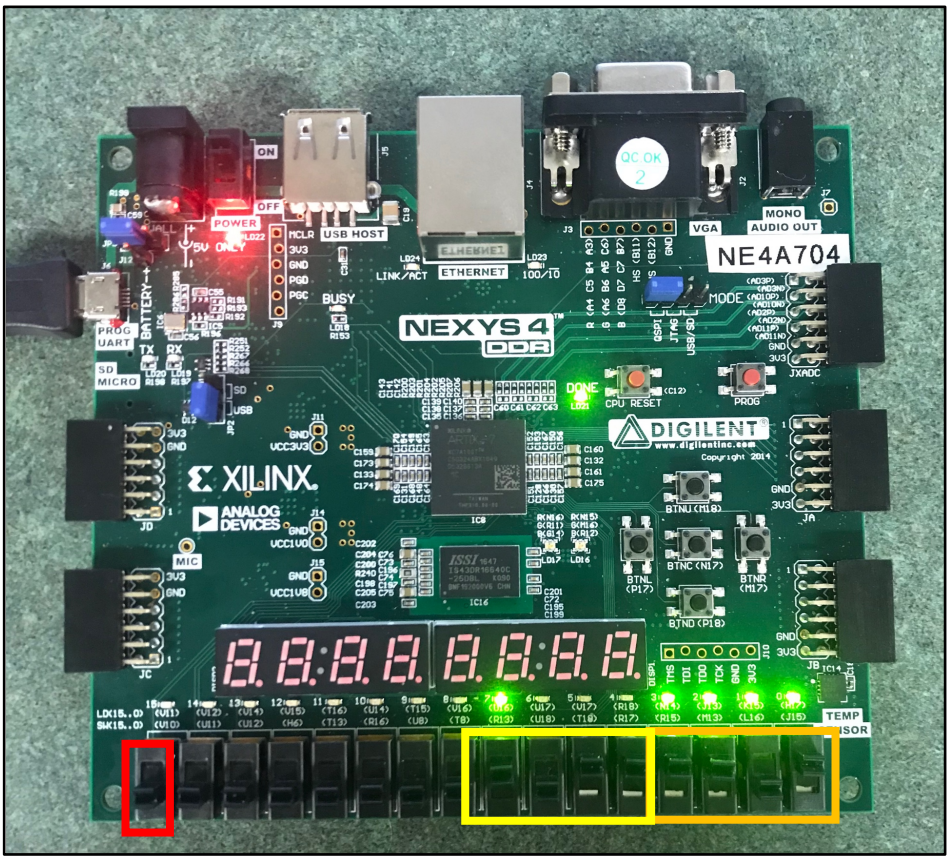
Code 26: The Top part which connects everything, the fsm to mult - Verilog

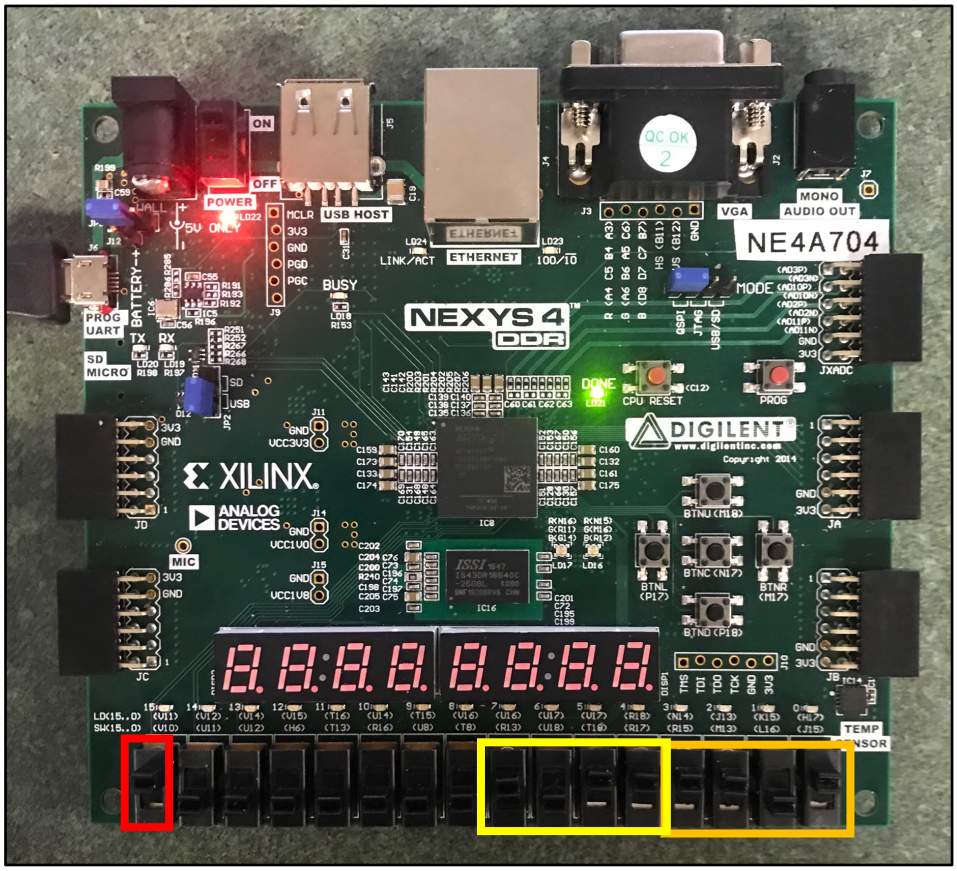
```
module top_test;
reg         clk, rst;
reg  [3:0] db, da;
wire [7:0] y;

top uut (clk, rst, da, db, y);

initial clk = 0;
always #10 clk = ~ clk;

initial
begin
     rst = 1;
     #20 rst = 0; db = 4'b1101; da = 4'b1011;
     #250 rst = 0;
     #20 rst = 0; db = 4'b0010; da = 4'b0100;
     #250 $stop;
end
initial $monitor($time, "ns, rst=%d, y=%d, db=%d, da=%d", rst, y, db, da);
endmodule
```

Code 27: Top testbench - Verilog

2.4 Constraint File:

```
1
2  ## Clock signal
3  set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
4  create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];
5
6
7  ##Switches
8
9  set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { db[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
10 set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { db[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
11 set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { db[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
12 set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { db[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
13 set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { da[0] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
14 set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { da[1] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
15 set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { da[2] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
16 set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { da[3] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
17 #set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
18 #set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
19 #set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
20 #set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
21 #set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
22 #set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
23 #set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
24 set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { rst }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
25
26
27 ## LEDs
28
29 set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { y[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
30 set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { y[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
31 set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { y[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
32 set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { y[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
33 set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { y[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
34 set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { y[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
35 set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { y[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
36 set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { y[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
```

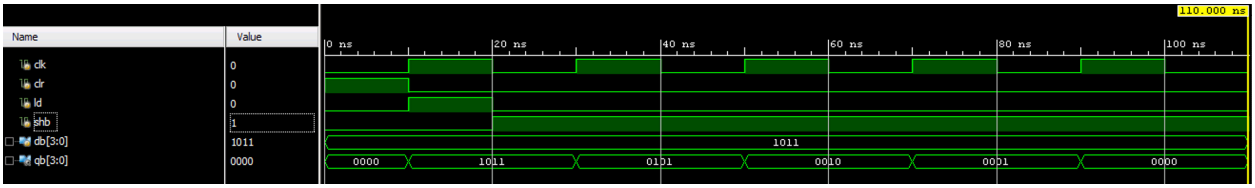




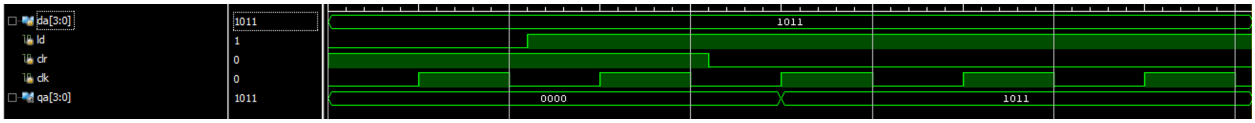
As shown in the pictures to the left, the inputs are 1101 and 1011. The first should the correct output and the second photo shows what happens when the reset switch (far left) is high, even if the inputs are high.

The red switch is the reset.
The orange switches are the db.
The yellow switches are the da.

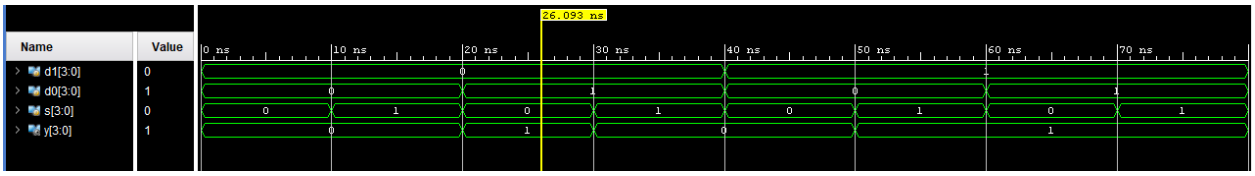
2.5 Simulation Waveform:



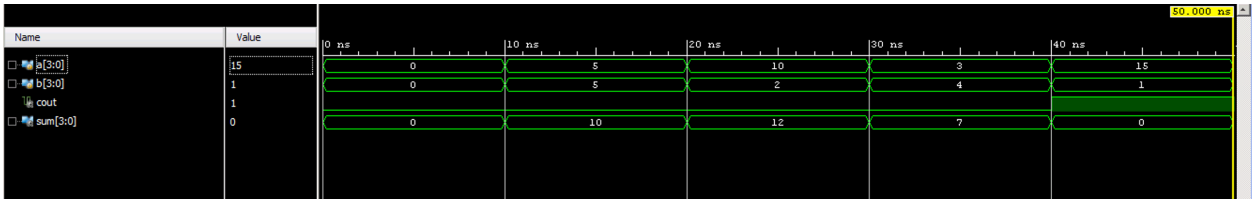
Waveform 7: The shifting input, MB, where db is the input. If ld is high, then the input is load and the output is given. Otherwise, if shb is high than the output shifts right by 1 bit, as shown Any change occurs at a rising clock edge.



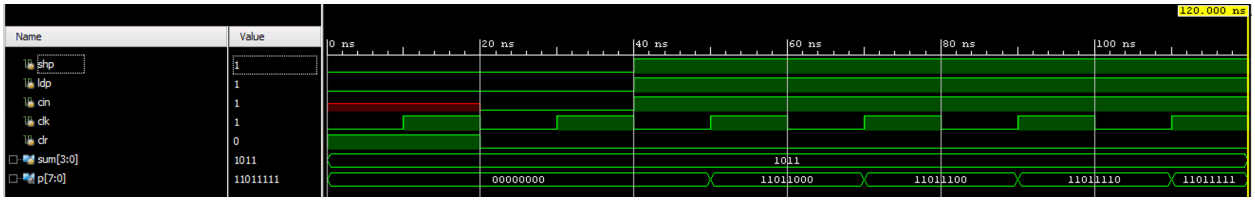
Waveform 8: The non-shifting input, MA, is a d flip-flop and only holds data. Any change occurs at a rising clock edge.



Waveform 9: 4-bit Multiplexer. If s is 1 then d1 is the output, y, and if s is 0 then the output is d1 Any change occurs at a rising clock edge.

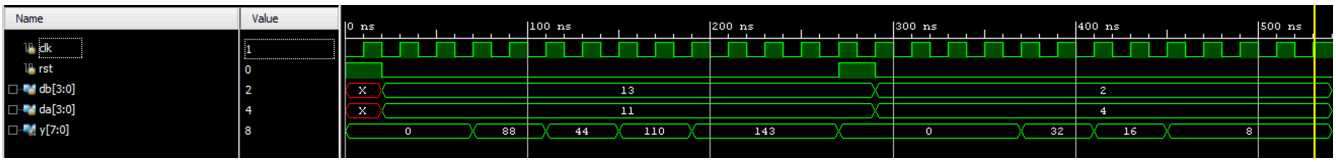


Waveform 10: 4-bit Adder. Sum is the output, and as shown, works correctly. The last test, 15 + 1 = 16, shows how the cout works by carrying out the extra bit. Any change occurs at a rising clock edge



Waveform 11: The Prod. Works perfectly by loading the input, labeled sum, into the prod. ONLY the last four bits shift, as it should, when shp is high. Ldp loads the internal process into the output. Any change occurs at a rising clock edge.

Waveform 12: The Mult. This is where we see the final product of the two input numbers. As shown, 11 x 13 = 143, which is correct. When shb is high the input in MB is shifted. When ld is high, MB and MA are loaded. When shp and ldp are high the output in prod is updated. Any change occurs at a rising clock edge.



Waveform 13: The Top. Reveals that the finite state machine works because the testbench is the clock, reset, and two input values, and yet the result labeled y is correct. I tested two produces 13 & 11 and 2 & 4. Both are correct. Any change occurs at a rising clock edge.

2.6 Result Discussion: Part II of lab 2 took a long time and required much work. I had the most difficulty with the prod. First, I had to understand the importance of blocking and nonblocking assignments. Second, although the simulation is correct, the code might not synthesis. And fixing that might make the FPGA unable to download the code. For example, in the prod I had only if statements with no else if, but the code would only synthesis with if and else if. However, my output would be incorrect, so I spilt the code to have two always blocks. The simulation worked, yet the FPGA did not like having two always blocks. Eventually, I found a way that made the program work. Other than that, the other issues were small such as still being a little unfamiliar with the program or mistyping. In this lab, I learned more about finite state machine, d flip-flops, and multiplexers. Overall, with the amount of time that I poured into this problem, I am very proud of my 4 by 4-bit multiplier.

Part III – Character Displays on 8-Digit Multiplexed Seven Segment Displays

3.1 Design Purpose: For this part, I experiment with the segment displays on the FPGA. The professor provides at lot of the code beforehand. But understanding the logic and schematic behind the code is very important. For part three, my assignment is to display “CPE166MS” on the FPGA.

3.2 Engineering Data:

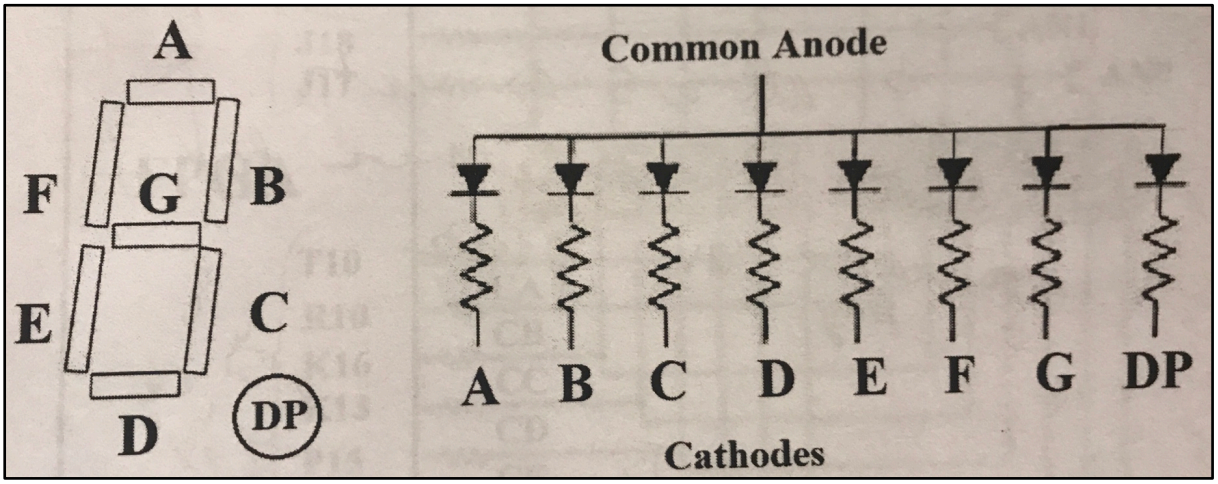


Figure 8: The internal schematic of a common anode segment display.

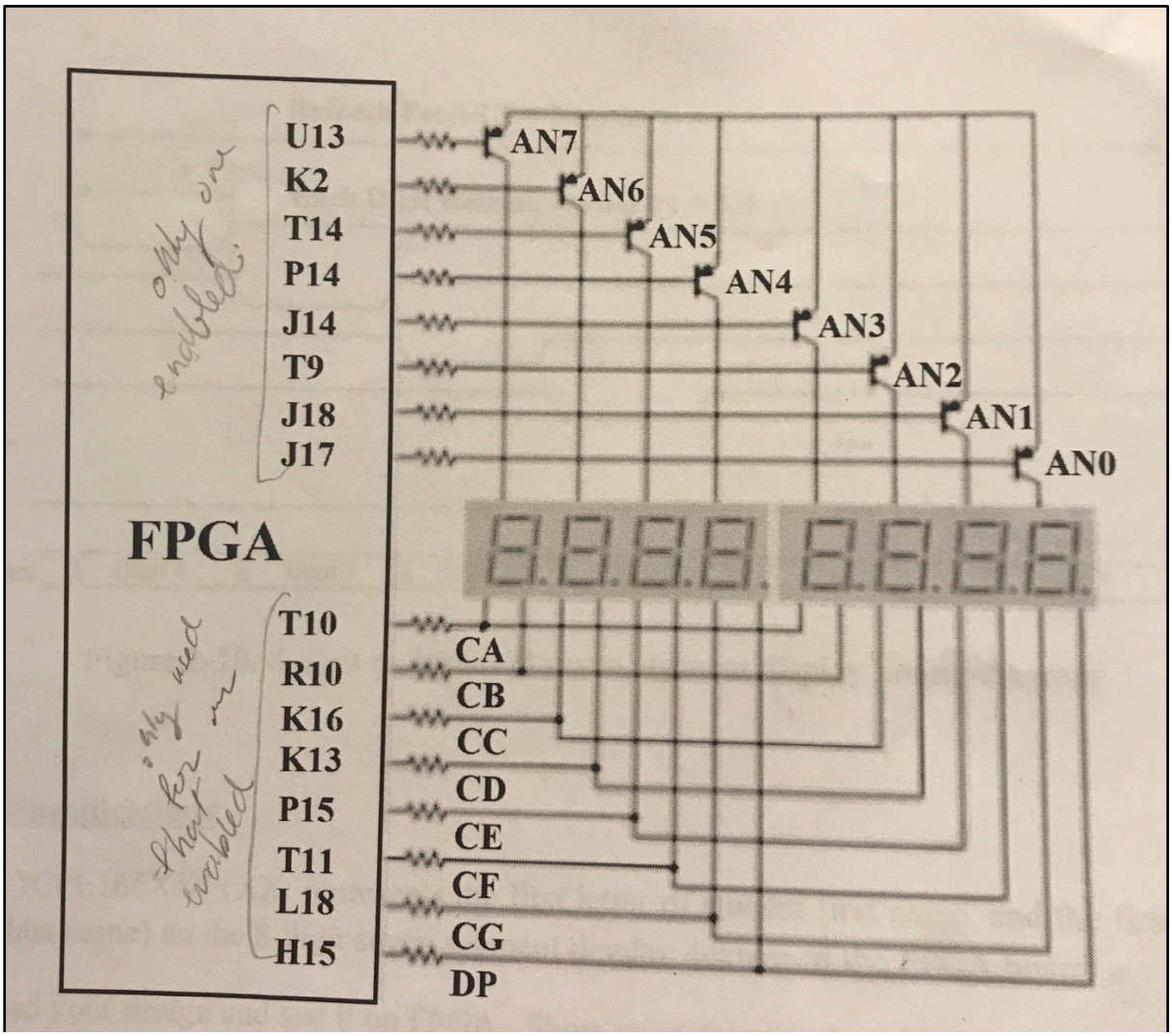


Figure 9: Since the FGPA can only take so many inputs, the anodes and common anodes need to share input. To turn on a segment the anode and the cathode show be low active.

3.3 Source Code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Mariya Shtevnina
//////////////////////////////////////////////////////////////////////////////////
module fpga_fun(clk, seg, dig);

input        clk;
output [7:0] seg;
output [7:0] dig;

parameter N = 18;

reg [N-1:0] count;
reg [3:0]   dd;
reg [7:0]   seg;
reg [7:0]   an;

always @ (posedge clk)
 begin
  count <= count + 1;

  case(count[N-1:N-3])
   3'b000 :
    begin
     dd = 4'd7;
     an = 8'b11111110;
    end

   3'b001:
    begin
     dd = 4'd6;
     an = 8'b11111101;
    end

   3'b010:
    begin
     dd = 4'd5;
     an = 8'b11111011;
    end

   3'b011:
    begin
     dd = 4'd4;
     an = 8'b11110111;
```

```
    end

    3'b100 :
        begin
         dd = 4'd3;
         an = 8'b11101111;
        end

       3'b101:
        begin
         dd = 4'd2;
         an = 8'b11011111;
        end

       3'b110:
        begin
         dd = 4'd1;
         an = 8'b10111111;
        end

       3'b111:
        begin
         dd = 4'd0;
         an = 8'b01111111;
        end
  endcase
 end
assign dig = an;

always @ (dd)
 begin
  seg[7] = 1'b1;
  case(dd)
   4'd0 : seg[6:0] = 7'b1000110; //G F E D C B A to display C
   4'd1 : seg[6:0] = 7'b0001100; //to display P
   4'd2 : seg[6:0] = 7'b0000110; //to display E
   4'd3 : seg[6:0] = 7'b1111001; //to display 1
   4'd4 : seg[6:0] = 7'b0000010; //to display 6
   4'd5 : seg[6:0] = 7'b0000010; //to display 6
   4'd6 : seg[6:0] = 7'b0001001; //to display (H) M;
   4'd7 : seg[6:0] = 7'b0010010; //to display S
   default : seg[6:0] = 7'b1111111; //blank
  endcase
 end
```

3.4 Constraint File:

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk }];



set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { seg[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { seg[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { seg[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { seg[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { seg[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { seg[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { seg[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { seg[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { dig[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { dig[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { dig[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { dig[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { dig[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { dig[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { dig[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { dig[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

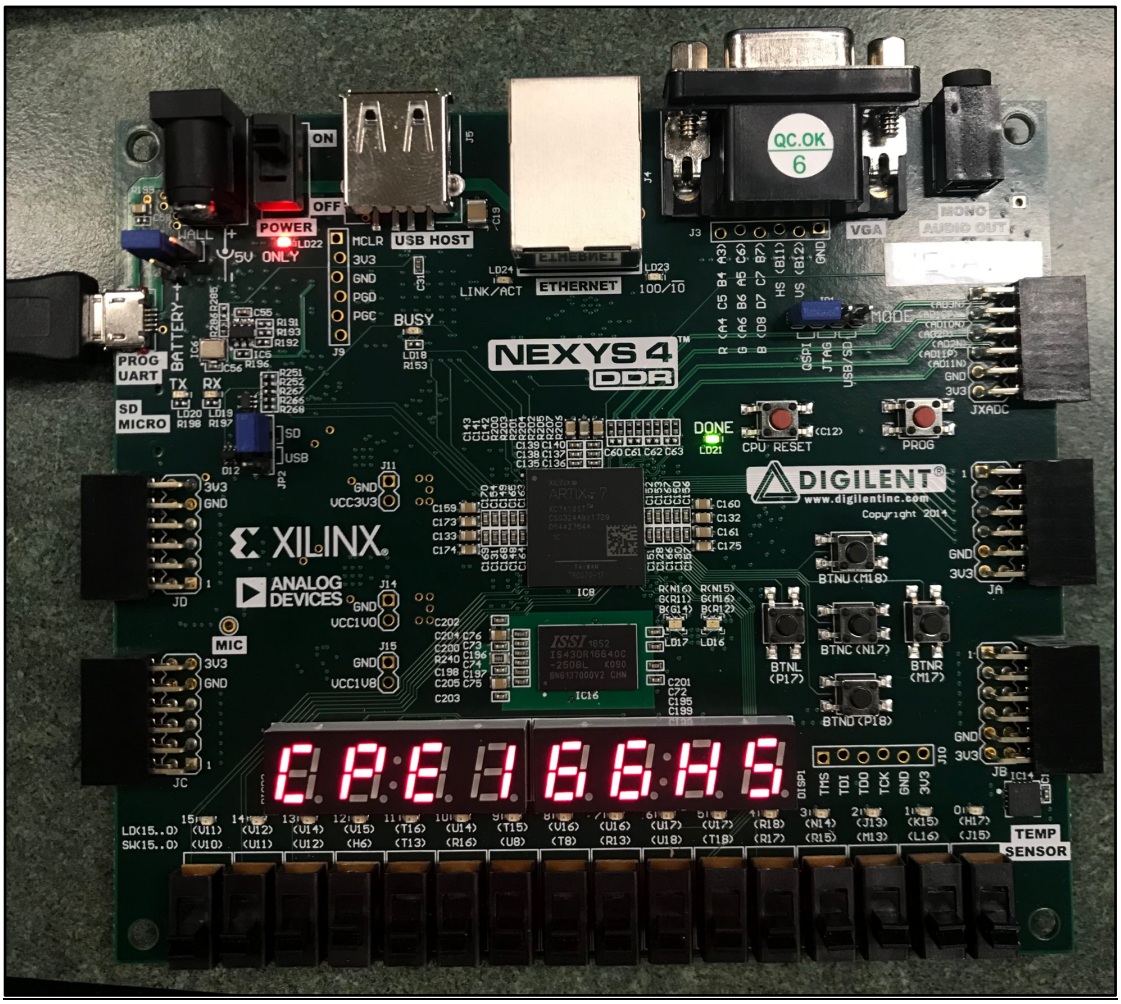


Figure 10, I could not spell the letter M on the FPGA, so I chose H instead.

3.5 Result Discussion: I found the third part fun and easy. I had trouble at first understanding the common anode and how the all the cathodes worked together to make the seven-segment display, but after the professor explained the code in class, I understood it much better. Overall, I enjoyed this part, especially after part two.

Conclusion

In lab 2 I learned how to write in Verilog, how to program on the FPGA, the importance of testbenches, how to solve problems, how to start designing solutions for proposed problems, and the hierarchical design strategy, especially in parts I and II. I also learned how to create a carry select adder, sequential 4 by 4 multiplier, d flip-flop, testbench, and multiplexer. The professor helped immensely by explaining the labs in class. I understood that labs require much patience and diligence, but eventually the problem gets solved. I enjoyed building a code off a schematic and seeing immediate results on either the simulation or the FPGA. I got a lot more familiar with Verilog, Xilinx, and the NEXYS4 FPGA through this lab. Overall, this lab taught me much about computer engineering, even though a lot of the lab was basics like adding and multiplying and displaying on the hardware.