

CPE 166 / EEE270 ADVANCED LOGIC DESIGN LAB 2

Lab Session: Wednesday 2PM - 4:40PM

Student Name: Andrew Robertson

TABLE OF CONTENTS

Introduction.....	3
Part 1	4
Design Purpose:	4
Engineering Data:	4
Source code:	6
Simulation waveform(s):	7
Result discussion:	7
Part 2	8
Design Purpose:	8
Engineering Data:	8
Source code:	10
Simulation waveform(s):	15
Result discussion:	17
Part 3	18
Design Purpose:	18
Engineering Data:	18
Source code:	18
Simulation waveform(s):	19
Result discussion:	19

INTRODUCTION

This lab is a much more extensive lab than the first, and as such is allowed about 6 weeks total to complete. As outlined in the lab book, our goals for these exercises include:

- Hierarchical design
- Combinational multiplier design
- Sequential multiplier design
- Output to seven segment displays
- Simulations / Test benches
- Running programs on the development board

Please note these objectives sometimes parallel those in the lab book rather than keeping on track with them, this is because of different instructions given by Dr. Jing Pang – author of the manual being used. These new instructions are not mentioned in the manual but rather emailed to us directly likely due to the use of a brand-new development board instead of the board the manual was originally written for.

PART 1

DESIGN PURPOSE:

Design and test a four by four combinational multiplier. Four by four means the two multiplicands to be interpreted are four bits each, yielding an eight-bit result.

ENGINEERING DATA:

Figure F1 (Provided by the lab manual)

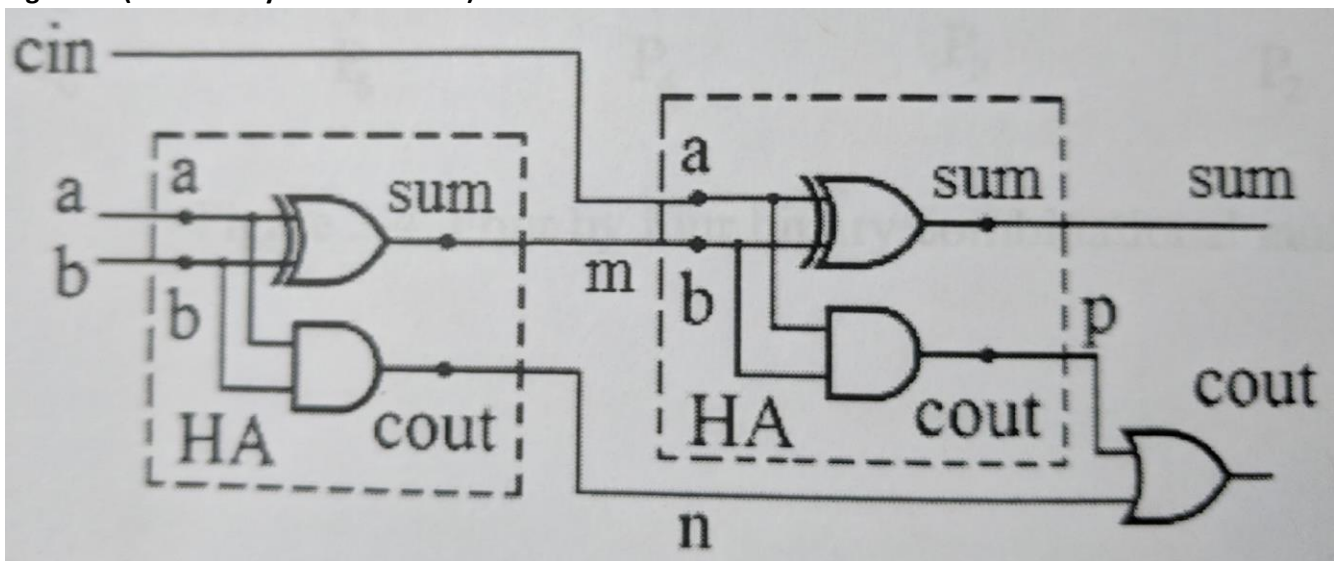
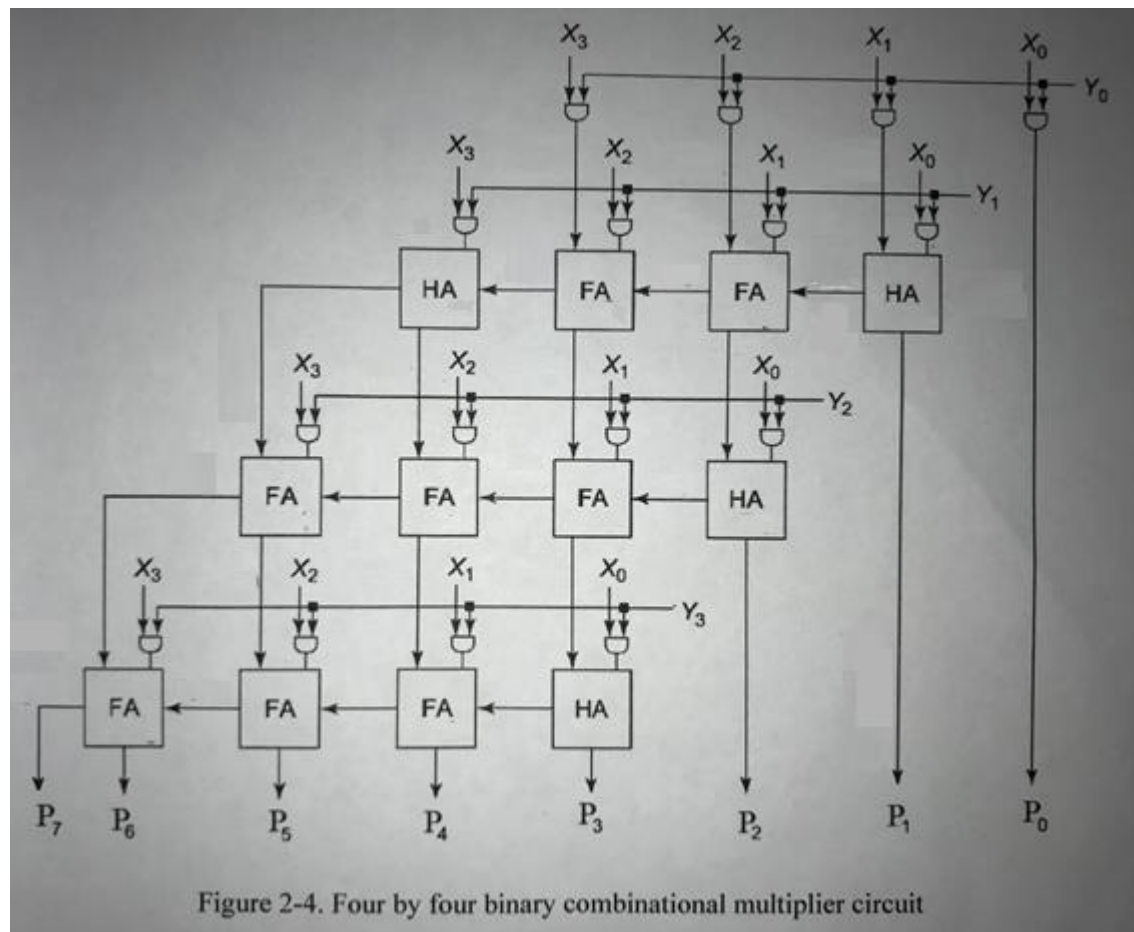


Figure F1 above outlines the hierarchical design of a full adder. It is hierarchical because it's made up of two half adders that have their carry out signals run through an OR gate. Our first step to our end goal is to design the half adder so that we can use that design to create a full adder, and lastly use both the full adder and half adder designs for the combinational four-bit multiplier. Figure S1 under source code below shows the Verilog code used for our half adder design along with its resulting waveform W1. Once the half adder was designed, the full adder design was to simply create two instances of half adders and one OR gate, the only part of this design that may not be immediately intuitive is the wires created. These serve to tie together the outputs from one half adder into the inputs to either the other half adder or the OR gate. The design for the full adder is shown as figure S2 under source code along with its resulting waveform W2 below. Last was to apply the same sort of idea as the full adder, by way of instantiation of half adders and full adders to solve a larger problem. Figure F2 below helped a lot for keeping track of every adder module and their connection to one another. My approach to this problem was to physically label every wire in F2 that did not have a direct signal name as part of an array of wires named w. This way, every time I created another adder instance I can see the wire names I need to associate to the inputs and outputs clearly. All adder modules were also numbered just to make design even easier to follow. The source code for this approach is shown below as source S3 along with its resulting waveform W3.

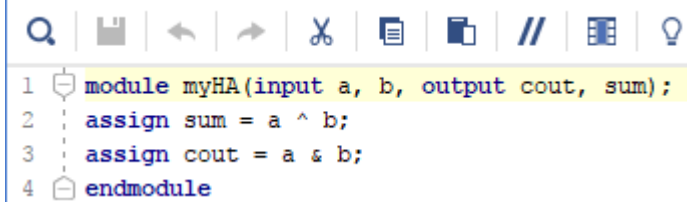
Figure F2 (Provided by Dr. Jing Pang)



Please note that in the design as described under source S3, I did not necessarily need to create wires for the small wires immediately following the and gates in figure F2 above. This was only done for neatness in code and to make the modules both easier to read and more uniform. For example, wire $w[0] = X3 \& Y0$, but I could have simply used $(X3 \& Y0)$ in place of $w[0]$.

SOURCE CODE:

Source S1

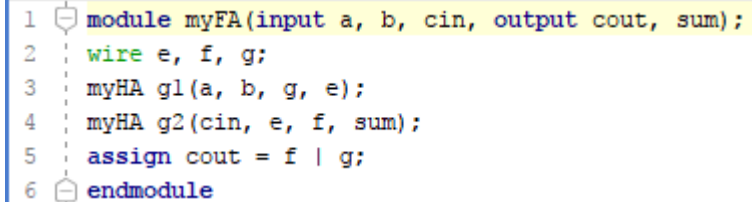


```

1 module myHA(input a, b, output cout, sum);
2   assign sum = a ^ b;
3   assign cout = a & b;
4 endmodule

```

Source S2

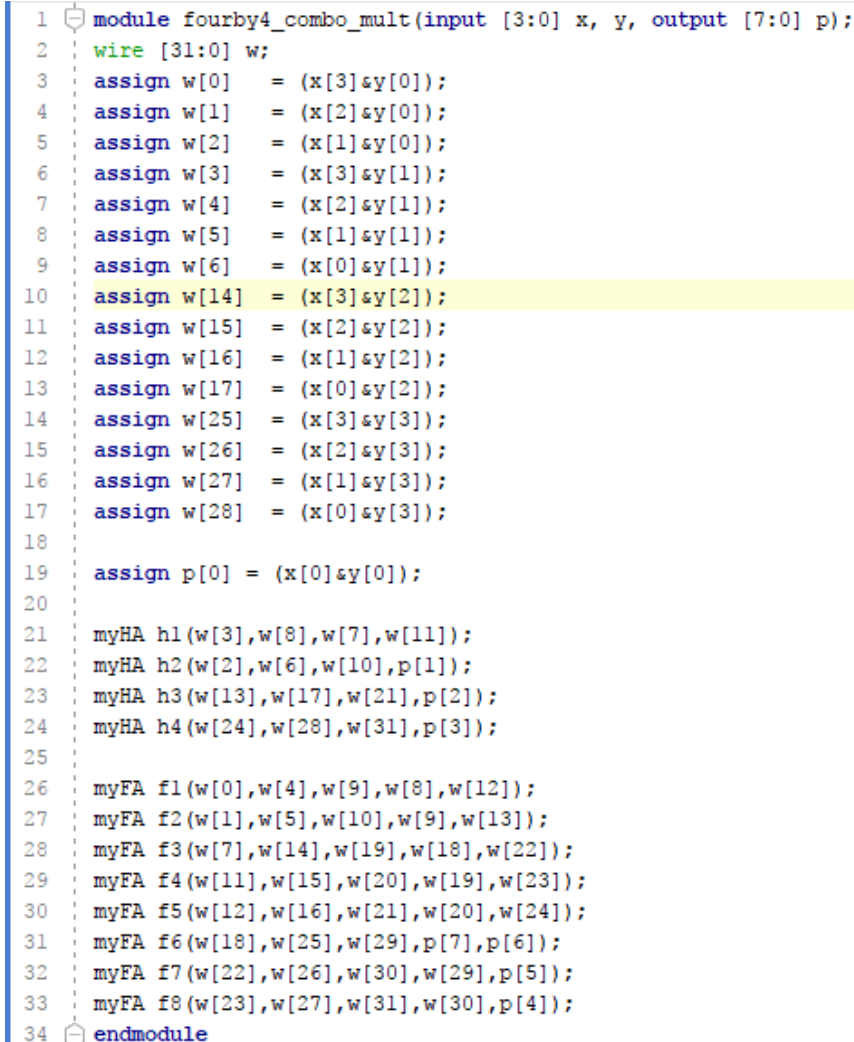


```

1 module myFA(input a, b, cin, output cout, sum);
2   wire e, f, g;
3   myHA g1(a, b, g, e);
4   myHA g2(cin, e, f, sum);
5   assign cout = f | g;
6 endmodule

```

Source S3



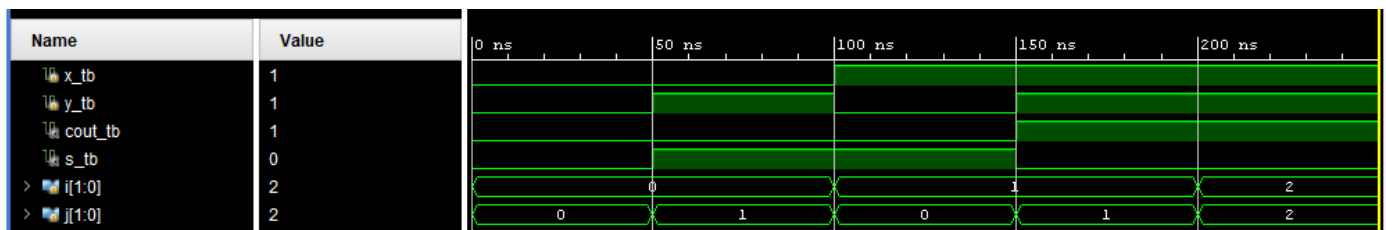
```

1 module fourby4_combo_mult(input [3:0] x, y, output [7:0] p);
2   wire [31:0] w;
3   assign w[0]   = (x[3]&y[0]);
4   assign w[1]   = (x[2]&y[0]);
5   assign w[2]   = (x[1]&y[0]);
6   assign w[3]   = (x[3]&y[1]);
7   assign w[4]   = (x[2]&y[1]);
8   assign w[5]   = (x[1]&y[1]);
9   assign w[6]   = (x[0]&y[1]);
10  assign w[14]  = (x[3]&y[2]);
11  assign w[15]  = (x[2]&y[2]);
12  assign w[16]  = (x[1]&y[2]);
13  assign w[17]  = (x[0]&y[2]);
14  assign w[25]  = (x[3]&y[3]);
15  assign w[26]  = (x[2]&y[3]);
16  assign w[27]  = (x[1]&y[3]);
17  assign w[28]  = (x[0]&y[3]);
18
19  assign p[0] = (x[0]&y[0]);
20
21  myHA h1(w[3],w[8],w[7],w[11]);
22  myHA h2(w[2],w[6],w[10],p[1]);
23  myHA h3(w[13],w[17],w[21],p[2]);
24  myHA h4(w[24],w[28],w[31],p[3]);
25
26  myFA f1(w[0],w[4],w[9],w[8],w[12]);
27  myFA f2(w[1],w[5],w[10],w[9],w[13]);
28  myFA f3(w[7],w[14],w[19],w[18],w[22]);
29  myFA f4(w[11],w[15],w[20],w[19],w[23]);
30  myFA f5(w[12],w[16],w[21],w[20],w[24]);
31  myFA f6(w[18],w[25],w[29],p[7],p[6]);
32  myFA f7(w[22],w[26],w[30],w[29],p[5]);
33  myFA f8(w[23],w[27],w[31],w[30],p[4]);
34 endmodule

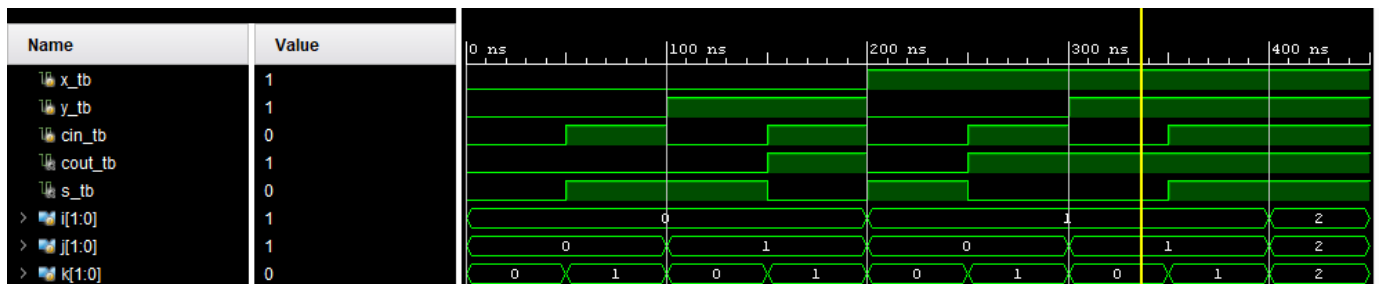
```

SIMULATION WAVEFORM(S):

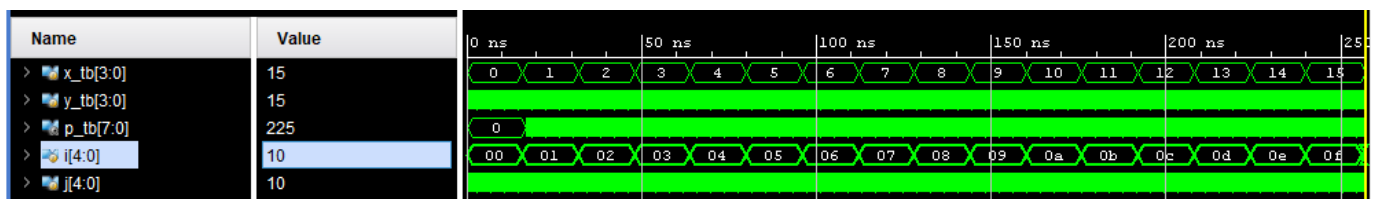
Wave W1



Wave W2



Wave W3



RESULT DISCUSSION:

Part 1 of lab 2 was an interesting experience for me because this marked a certain first for me. I previously always coded in an IDE, and of course the IDE makes you aware of any typological errors or syntax faults generally as you go along. In this case, I had a couple hours to spare before the lab session and wanted to give it a shot on my own. I used the basic windows notepad and did not need to change a thing when arriving to lab. The only part that was left was writing test benches, and to my surprise everything worked as intended right off the bat. Part 2 is not a similar story however.

PART 2

DESIGN PURPOSE:

Design and test a four by four sequential multiplier. Four by four means the two multiplicands to be interpreted are four bits each, yielding an eight-bit result.

ENGINEERING DATA:

The two figures I will be referring to often are both from the lab manual, and are shown immediately below:

Figure F3

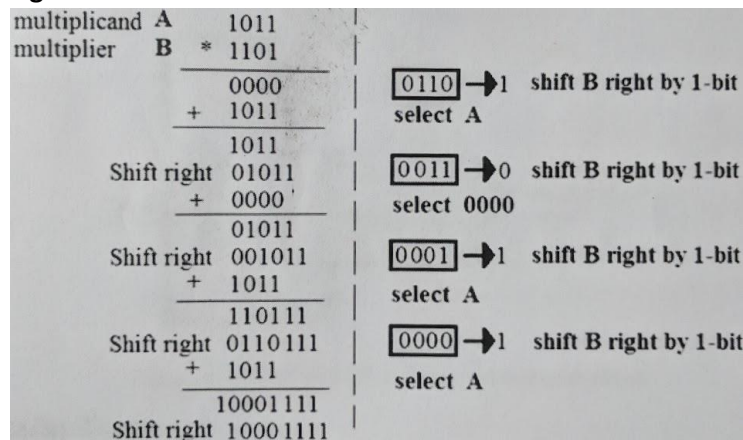
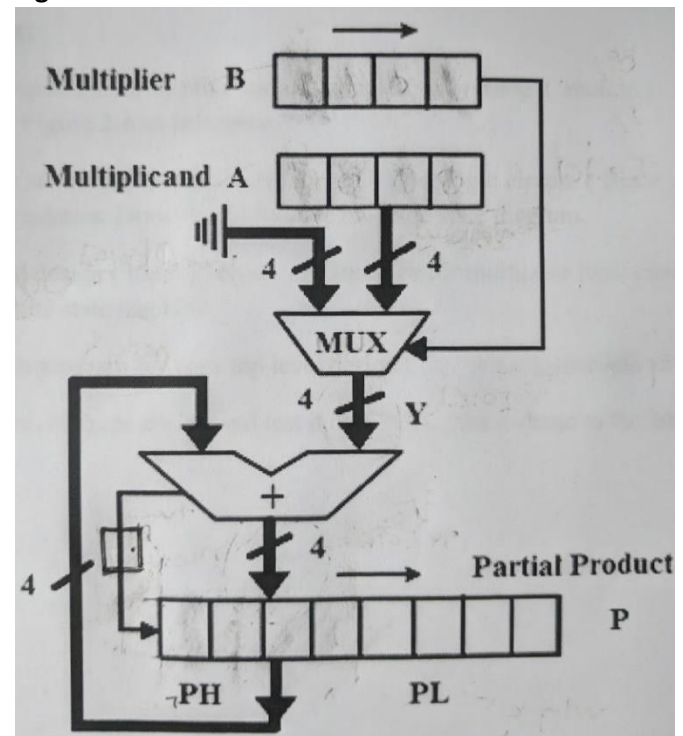


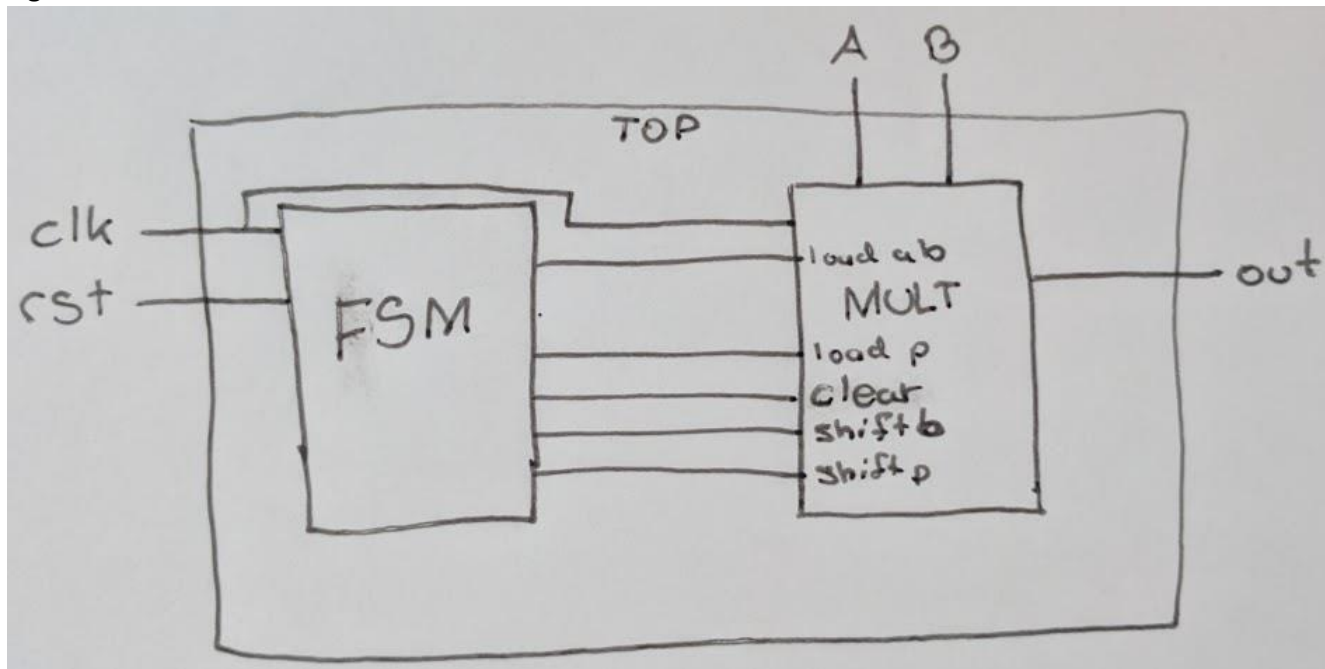
Figure F4



With this problem, as with any other layered solution, it is important to understand what is happening and come up with a way to break the problem up into smaller problems. Luckily for us, Dr. Jing Pang gave a huge helping hand in making this work. First, two lecture sessions were dedicated to the understanding of this circuit. Second, Verilog modules for D flip flops, the MUX, and the half adder were all provided and were tailored specifically for use in this design. The flip flops created for the partial products upper four bits, lower four bits, and in our case the carry were all simply modified versions of Dr. Pang's code. The most important thing I learned from this project is the large difference between combinational circuit design and sequential circuit design. In part 1, we worked entirely with combinational circuits and did not try to meld the two approaches together. In this part, we meld both approaches together. I realize now that all troubles that came from this process all stemmed from a misunderstanding of this idea. The design recommendation from Dr. Pang in graphical form is below, figure F5. Figure F5 is where all began to make sense. The idea here is to first create a combinational circuit with sequential elements as indicated by the chip named MULT on the right of figure F5, this chip directly corresponds to the circuit elements outlined in figure F4. Second, a single path finite state machine is created to drive the MULT chip created. Third, an encapsulating layer is created to synchronize the two and simplify development board design. Figure F3 is important because

it explains the sequential nature of this approach. This is very close to the natural way we do multiplication by hand. By following this we can tell that first we calculate the partial product, we then add this to the last partial product, shift the result right and then repeat this process for the next value of the second multiplicand. Essentially, figure F3 tells us how our finite state machine should work.

Figure F5



In Figure F4, I drew one more element in to the recommended design that is an additional D flip flop with the purpose of storing the carry value from the half adder. The reason I decided to add this came about when reviewing the test bench of the multiplier with known values and outcomes. After staring at the waveform for quite some time trying to see what went wrong I noticed that the value for the half adders carry out would sometimes change immediately after loading the half adders sum into the product register. This was bad because it happened so fast that shifting our product right would then pull in the new carry value instead of the one that is desired. The carry D flip flop was instantiated to load at the same time the partial product is loaded to preserve that value and use it when shifting the partial product instead.

SOURCE CODE:

Source S4

```
module dffa (clk, clr, load, da, qa);
input      clk, clr, load;
input [3:0] da;
output [3:0] qa;

reg      [3:0] qa;

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

endmodule
```

Source S5

```
module dffb (clk, clr, load, sft, db, qb);
input      clk, clr, load, sft;
input [3:0] db;
output [3:0] qb;

reg      [3:0] qb;

always@(posedge clr or posedge clk)
begin
    if(clr) qb <= 0;
    else if (load)
        qb <= db;
    else if (sft)
        qb <= { 1'b0, qb[3:1] };

    // qb[3] <= 1'b0;
    // qb[2] <= qb[3];
    // qb[1] <= qb[2];
    // qb[0] <= qb[1];

end

endmodule
```

Source S6

```

1 module MUX2_1(d0, d1, s, y);
2   input [3:0] d1, d0;
3   input      s;
4   output [3:0] y;
5
6   reg [3:0] y;
7
8   always@( s or d1 or d0 )
9   begin
10    if (s)
11        y = d1;
12    else
13        y = d0;
14  end
15
16 endmodule

```

Source S7

```

1 module adder( a, b, cout, s );
2   input [3:0] a, b;
3   output      cout;
4   output [3:0] s;
5
6   assign { cout, s } = a + b;
7
8   endmodule

```

Source S8

```

1 module dffcarry (clk, clr, load, da, qa);
2   input      clk, clr, load;
3   input da;
4   output qa;
5
6   reg qa;
7
8   always@(posedge clr or posedge clk)
9   begin
10    if(clr) qa <= 0;
11    else if (load)
12        qa <= da;
13  end
14
15 endmodule

```

Source S9

```

1 module ph (clk, clr, load, sft, carry , dph, qph);
2   input      clk, clr, load, sft, carry;
3   input [3:0] dph;
4   output [3:0] qph;
5
6   reg      [3:0] qph;
7
8   always@(posedge clr or  posedge clk)
9   begin
10      if(clr) qph <= 0;
11      else if (load)
12          qph <= dph;
13      else if (sft)
14          qph <= { carry,  qph[3:1] };
15
16          // qb[3] <= carry;
17          // qb[2] <= qb[3];
18          // qb[1] <= qb[2];
19          // qb[0] <= qb[1];
20
21      end
22
23 endmodule

```

Source S10

```

1 module pl (clk, clr, load, sft, ph_low , dpl, qpl);
2   input      clk, clr, load, sft, ph_low;
3   input [3:0] dpl;
4   output [3:0] qpl;
5
6   reg      [3:0] qpl;
7
8   always@(posedge clr or  posedge clk)
9   begin
10      if(clr) qpl <= 0;
11      else if (load)
12          qpl <= dpl;
13      else if (sft)
14          qpl <= { ph_low,  qpl[3:1] };
15
16          // qb[3] <= ph_low;
17          // qb[2] <= qb[3];
18          // qb[1] <= qb[2];
19          // qb[0] <= qb[1];
20
21      end
22
23 endmodule

```

Source S11

```

1  module multiplier(da, db, load_ab, load_p, shftb, shftp, clk, clr, out);
2
3  input clr, clk, shftb, shftp, load_ab, load_p;
4  input [3:0] da, db;
5  output [7:0] out;
6
7  wire [7:0] out;
8  wire [3:0] y, phw, plw, pp, dl, qa, qb, d0;
9  wire c_outl,cbuff;
10
11  assign d0 = 4'b0000;
12
13  ph phl(clk, clr, load_p, shftp, cbuff, pp, phw);
14
15  pl pll(clk,clr,0,shftp,phw[0],0,plw);
16
17  dffa dffal(clk,clr,load_ab,da,qa);
18
19  dffb dffb1(clk,clr,load_ab,shftb,db,qb);
20
21  dffcarry dffcarryl(clk, clr, load_p, c_outl,cbuff );
22
23  MUX2_1 muxl(d0,qa,qb[0],y);
24
25  adder addl(phw,y,c_outl,pp);
26
27  assign out = {phw, plw};
28
29  endmodule

```

Source S12

```

1 module myfsm(clk, rst, clr, shftb, shftp, loadab, loadp);
2
3     input clk, rst;
4     output clr, shftb, shftp, loadab, loadp;
5     reg clr, shftb, shftp, loadab, loadp;
6
7     parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4, s5 = 5, s6 = 6,
8               s7 = 7, s8 = 8, s9 = 9, s10 = 10, s11 = 11, s12 = 12, s13 = 13;
9
10    reg [3:0] cs, ns;
11
12    always @ (posedge rst or posedge clk)
13    begin
14        if(rst) cs <= s0;
15        else    cs <= ns;
16    end
17
18    always @ (cs)
19    begin
20        case(cs)
21            s0:    ns = s1;
22            s1:    ns = s2;
23            s2:    ns = s3;
24            s3:    ns = s4;
25            s4:    ns = s5;
26            s5:    ns = s6;
27            s6:    ns = s7;
28            s7:    ns = s8;
29            s8:    ns = s9;
30            s9:    ns = s10;
31            s10:   ns = s11;
32            s11:   ns = s12;
33            s12:   ns = s13;
34            s13:   ns = s13;
35            default: ns = s0;
36        endcase
37    end
38
39    always @ (cs)
40    begin
41        case(cs)
42            s0:    {clr, shftb, shftp, loadab, loadp} = 5'b10000;
43            s1:    {clr, shftb, shftp, loadab, loadp} = 5'b00010;
44            s2:    {clr, shftb, shftp, loadab, loadp} = 5'b00001;
45            s3:    {clr, shftb, shftp, loadab, loadp} = 5'b00100;
46            s4:    {clr, shftb, shftp, loadab, loadp} = 5'b01000;
47            s5:    {clr, shftb, shftp, loadab, loadp} = 5'b00001;
48            s6:    {clr, shftb, shftp, loadab, loadp} = 5'b00100;
49            s7:    {clr, shftb, shftp, loadab, loadp} = 5'b01000;
50            s8:    {clr, shftb, shftp, loadab, loadp} = 5'b00001;
51            s9:    {clr, shftb, shftp, loadab, loadp} = 5'b00100;
52            s10:   {clr, shftb, shftp, loadab, loadp} = 5'b01000;
53            s11:   {clr, shftb, shftp, loadab, loadp} = 5'b00001;
54            s12:   {clr, shftb, shftp, loadab, loadp} = 5'b00100;
55            s13:   {clr, shftb, shftp, loadab, loadp} = 5'b00000;
56            default: {clr, shftb, shftp, loadab, loadp} = 5'b10000;
57        endcase
58    end
59 endmodule

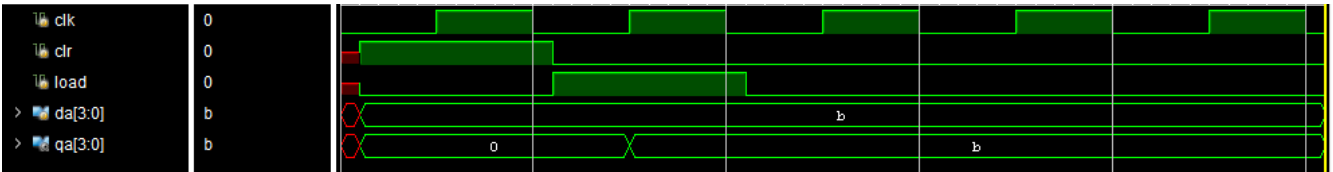
```

Source S13

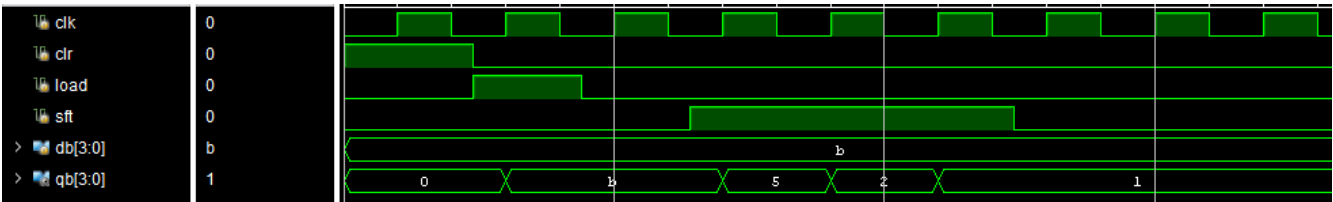
```
1 module top(topclock, topreset, multa, multb, topout);
2
3   input topclock, topreset;
4   input [3:0] multa, multb;
5   output [7:0] topout;
6
7   wire [7:0] topout;
8   wire loadmults, loadout, shiftmult, shiftout, clear;
9
10  myfsm myfsm1(topclock, topreset, clear, shiftmult, shiftout, loadmults, loadout);
11
12  multiplier multiplier1(multa, multb, loadmults, loadout, shiftmult, shiftout, topclock, clear, topout);
13
14 endmodule
```

SIMULATION WAVEFORM(S):

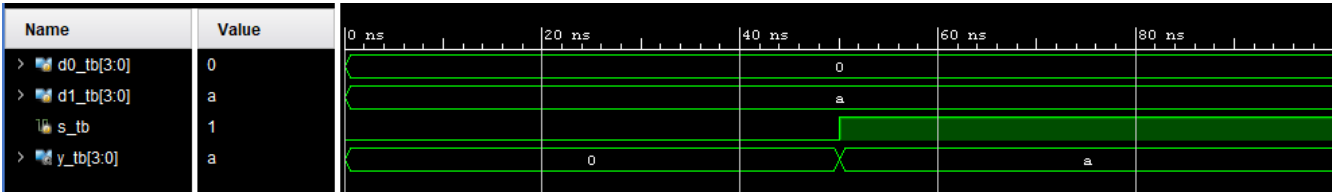
Wave W4 (DFFA)



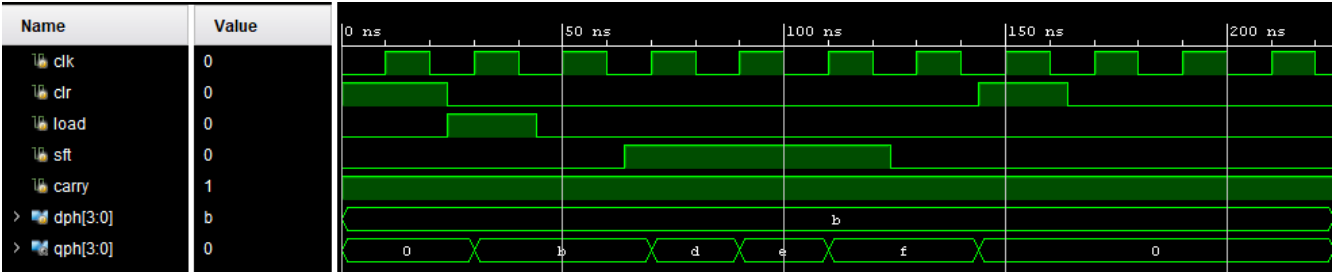
Wave W5 (DFFB)



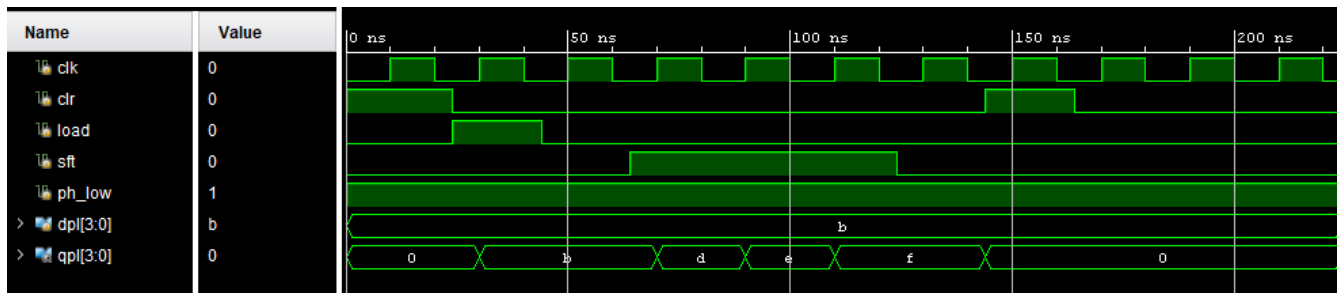
Wave W6 (MUX)



Wave W7 (DFFPH)



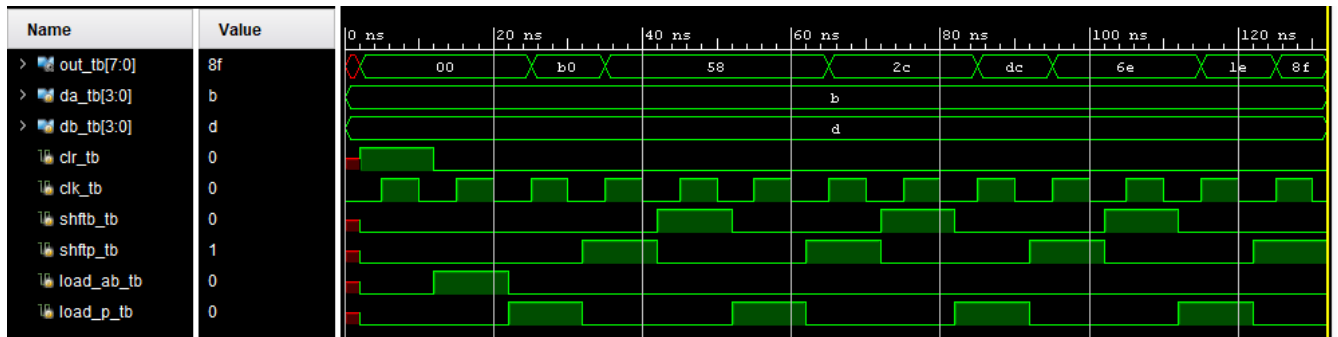
Wave W8(DFFPL)



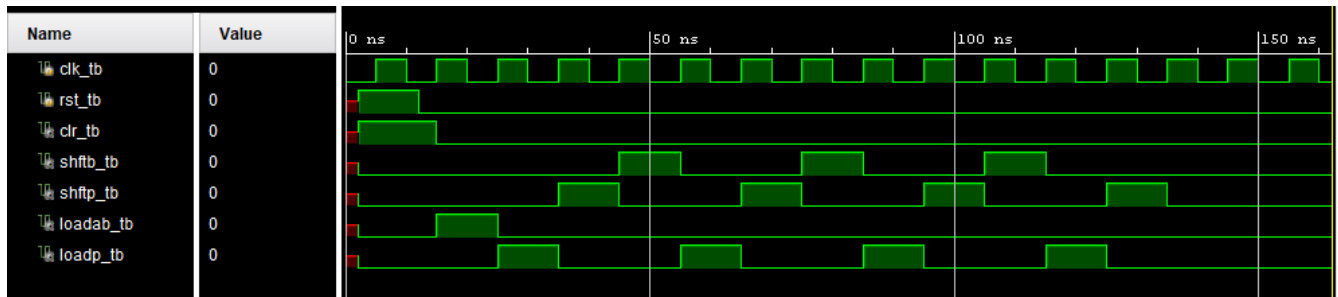
Wave W9(ADDER)



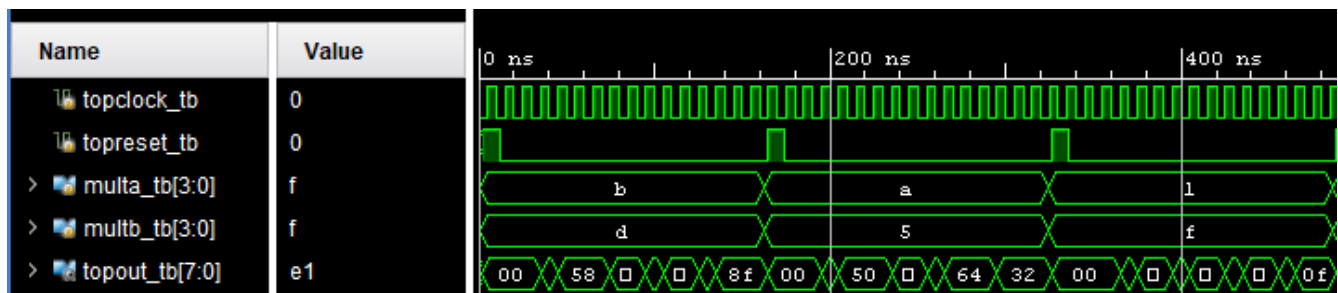
Wave W10(MULT)



Wave W11(FSM)



Wave W12(TOP)



RESULT DISCUSSION:

I learned way more from this lab than I expected too, most of what I learned I spoke about above but there is another way that I benefitted from this. Since I completed this lab quicker than most, I had the opportunity to help others with their versions of this same lab. From that experience I gained an even deeper knowledge of the workings of this lab through their approaches. Two ways I found I could improve on my code involved the registers PH, PL, and carry. First, PH and PL could be combined into 1 large register. This would make shifting more straight forward and would eliminate some wires. Second, the carry register could be integrated into the large P register as well – further eliminating wires but also ridding myself of another module. The only thing one would need to remember is that the result would be the lower eight bits of P and not its entirety. Well, that and the 4-bit wide wire coming from P to the half adder would be the four top bits just below the MSB since that is the carry.

Easily the most arduous part of this lab is making the multiplier chip function correctly. Once that was completed I was able to write and test the finite state machine, top, and their test benches in about an hour. In comparison, the multiplier chip was the rest of the 4-week period. At times I did not enjoy this project but in the end, I think it was extremely valuable and had a fun time with it.

PART 3

DESIGN PURPOSE:

Design a program that outputs a message that is visible all at once. This is to be done using multiple seven segment displays with only one display being active at any given time.

ENGINEERING DATA:

Part 3 of this lab took no more than 20 minutes to get working, and most of that time was figuring out the orientation of the development board vs the constraint file. Aside from that Dr. Pang provided sample code that pretty much already solved the design. All we needed to do was alter the bit mask for the seven-segment display at the very end of the code to display CPE166AR instead.

SOURCE CODE:

Source S14 (This is the code as provided by Dr. Pang)

```

1  module fpga_proj( clk, seg, dig);
2
3      input        clk;
4      output [7:0] seg;
5      output [7:0] dig;
6
7      parameter N = 18;
8
9      reg [N-1:0] count;
10     reg [3:0] dd;
11     reg [7:0] seg;
12     reg [7:0] an;
13
14     always @ (posedge clk)
15     begin
16         count <= count + 1;
17
18         case(count[N-1:N-3])
19             3'b000 :
20             begin
21                 dd = 4'd7;
22                 an = 8'b11111110;
23             end
24
25             3'b001:
26             begin
27                 dd = 4'd6;
28                 an = 8'b11111101;
29             end
30
31             3'b010:
32             begin
33                 dd = 4'd5;
34                 an = 8'b11111011;
35             end
36
37             3'b011:
38             begin
39                 dd = 4'd4;
40                 an = 8'b11110111;
41             end
42
43             3'b100 :
44             begin
45                 dd = 4'd3;
46                 an = 8'b11101111;
47             end
48
49             3'b101:
50             begin
51                 dd = 4'd2;
52                 an = 8'b11011111;
53             end
54
55             3'b110:
56             begin
57                 dd = 4'd1;
58                 an = 8'b10111111;
59             end
60
61             3'b111:
62             begin
63                 dd = 4'd0;
64                 an = 8'b01111111;
65             end
66         endcase
67     end
68     assign dig = an;
69
70
71     always @ (dd)
72     begin
73         seg[7] = 1'b1;
74         case(dd)
75             4'd0 : seg[6:0] = 7'b0001110; //to display F
76             4'd1 : seg[6:0] = 7'b0001100; //to display P
77             4'd2 : seg[6:0] = 7'b0000010; //to display G
78             4'd3 : seg[6:0] = 7'b0001000; //to display A
79             4'd4 : seg[6:0] = 7'b0111111; //to display -
80             4'd5 : seg[6:0] = 7'b0001110; //to display F
81             4'd6 : seg[6:0] = 7'b1100011; //to display u
82             4'd7 : seg[6:0] = 7'b0101011; //to display n
83             default : seg[6:0] = 7'b1111111; //blank
84         endcase
85     end
86 end
87
88 endmodule

```

Source S15 (The modified portion of Dr. Pang's code)

```

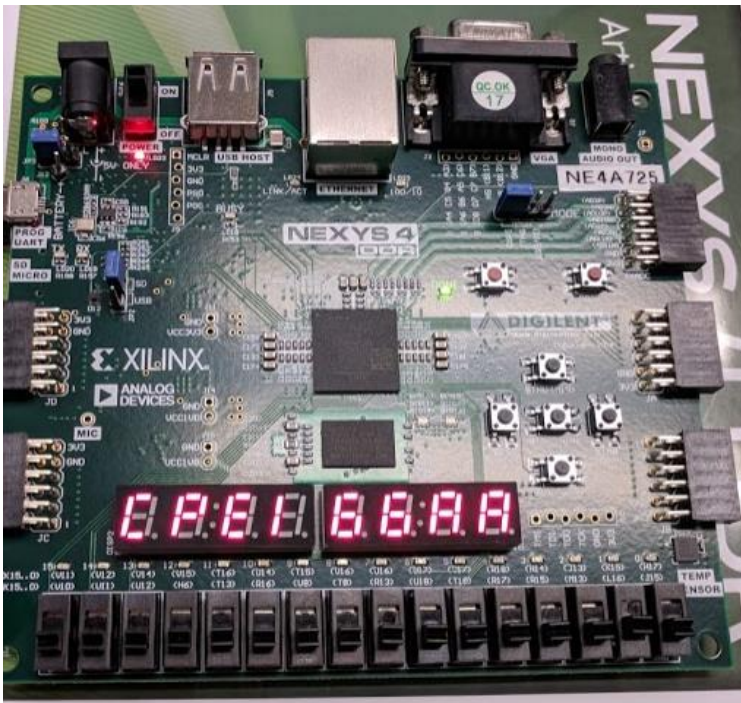
72  always @ (dd)
73  begin
74      seg[7] = 1'b1;
75      case(dd)
76          4'd0 : seg[6:0] = 7'b1000110; //to display C
77          4'd1 : seg[6:0] = 7'b0001100; //to display P
78          4'd2 : seg[6:0] = 7'b0000110; //to display E
79          4'd3 : seg[6:0] = 7'b1001111; //to display l
80          4'd4 : seg[6:0] = 7'b0000010; //to display 6
81          4'd5 : seg[6:0] = 7'b0000010; //to display 6
82          4'd6 : seg[6:0] = 7'b0001000; //to display A
83          4'd7 : seg[6:0] = 7'b0001000; //to display R
84          default : seg[6:0] = 7'b1111111; //blank
85      endcase
86  end
87
88  endmodule

```

SIMULATION WAVEFORM(S):

(We were told no simulation was needed for this part of lab 2 since working code was already provided)

RESULT DISCUSSION:



The result as seen to the human eye is one solid image. This is because the displays are cycling on and off so fast that we can't really see the time the displays are off. This seems like it would be a good idea for energy savings. Keep dialing down the cycles until just above the point we can tell.