CPE166 Advanced Logic Design

Lab 2 Report

Mario Palacios

03.11.2020

California State University, Sacramento

TABLE OF CONTENTS

Section **Page**

1) Introduction:

This lab serves as an introduction to hierarchical design in Verilog, a review on Finite State Machines (FSM), while also touching on the topics of combinational and sequential designs. We are tasked to complete an 8-Bit Carry Select Adder, 4 By 4 Sequential Multiplier, and display characters on an 8-Digit Multiplexed Seven Segment Displays. All of these Verilog designs would be created on Vivado and then implemented on the NEXYS4 DDR FPGA board. The lab serves a way for us to gain experience using the FPGA boards and to various Verilog techniques. These concepts are important because it will allow us to gain a solid foundation in order to explore future complex circuit designs.

2) 8-Bit Carry Select Adder Design – Project 1

2.1) 8-bit Carry Select Adder Schematic and Design Purpose

Schematic:

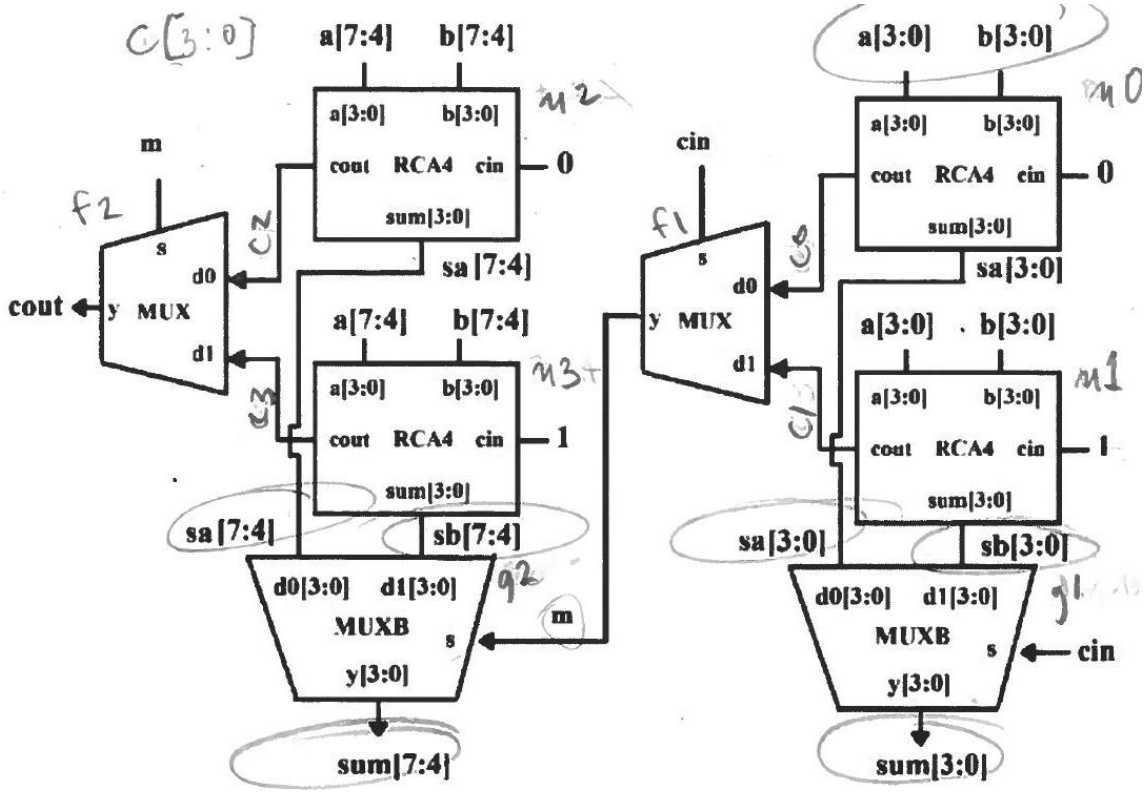


Figure 1. 8 – Bit Carry Select Adder by Using 4 – Bit Ripple Carry Adders, two 2to1 Multiplexers and two 4 – Bit 2to1 Multiplexers

Design Purpose:

The objective of this part of the lab is to learn how a hierarchical design strategy can be used in Verilog Hardware Description Language. In order to learn this we will be creating a carry select adder. This design is consisted of half adders, full adders, 4-bit ripple carry adders, a 2 to 1

multiplexer, and a multiplexer B that take two 4-bit inputs; all of these designs will be connected using a hierarchical design strategy.

2.2) Design Experiments

2.2.1) Half Adder Source Code, Testbench, Waveform and Result Discussion

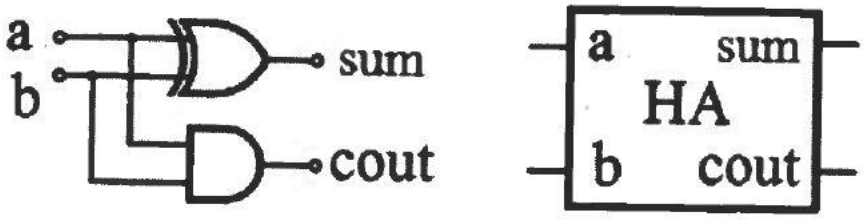


Figure 2. Schematic of Half Adder with Inputs A,B and Outputs of Sum, Cout

A half adder add two 1-bit inputs and b, then uses them to generate two outputs a 1-bit sum and a 10bit carry out (cout). The layout of a half adder can be seen in figure 2, right above, and in figure 3 right below, is the truth table of what our outputs should be. Figure 4 is the source code along with its testbench, while figure 5 displays the waveform created from the testbench.

Inputs		Outputs	
a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
Logic equations: cout = a b sum = a ⊕ b			

Figure 3. Half Adder Truth Table

//Source Code

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

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

//Testbench

```
`timescale 1ns / 1ps
module myHA_tb();
        reg a,b;
        wire cout,sum;

        myHA utt(.a(a),.b(b),.cout(cout),.sum(sum));

        initial
        begin
                a = 0; b = 0;
            #10 a = 0; b = 1;
```

```
                #10 a = 1; b = 0;
                #10 a = 1; b = 1;
                #20 $stop;
            end
endmodule
```

Figure 4. Half Adder Source Code and Testbench

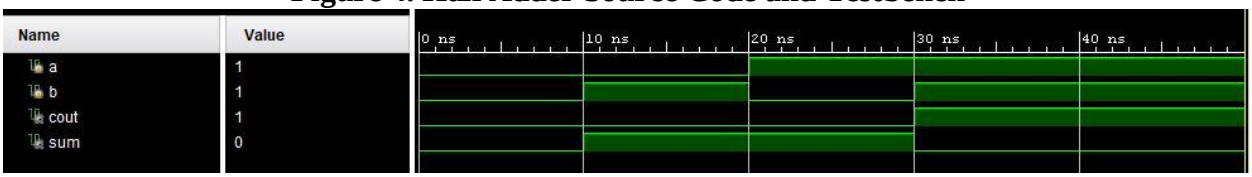


Figure 5. Half Adder Waveform Displaying Sum of A,B and the Cout

Results Discussion:
When creating the half adder, it was fairly simple to complete. In the waveform it is seen that when adding 01(1) + 01(1) it equal 10 (2), and as a result uses 0 as the sum but will have cout of 1, because the sum can only be a 1-bit number.

2.2.2) Full Adder Source Code, Testbench, Waveform and Result Discussion

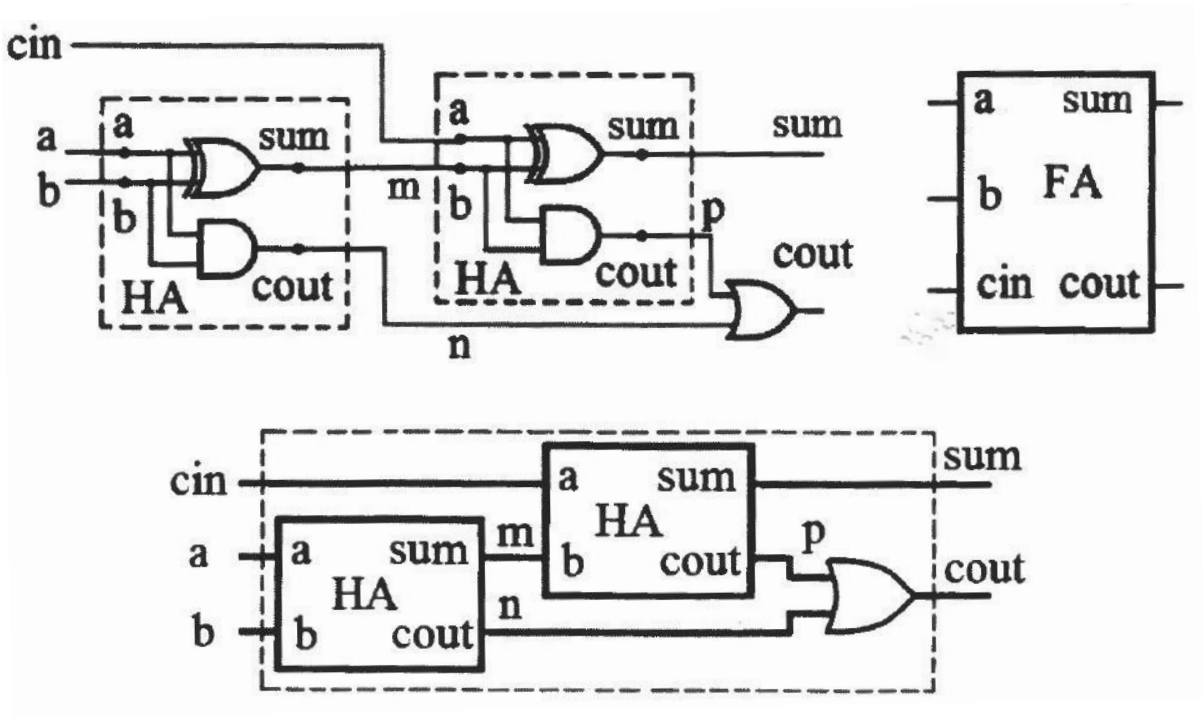


Figure 6. Schematic of Full Adder Created by Using Two Half Adders

A full adder adds three 1-bit binary inputs a, b and cin, which in result generates two outputs a 1=bit sum and 1-bit carry out (cout). The layout of a full adder can be seen in figure 6, right above, and in figure 7 right below, is the truth table of what our outputs should be. Figure 8 is the source code along with its testbench, while figure 9 displays the waveform created from the testbench.

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 ⊕ b ⊕ cin cout = (a ⊕ b) cin + a b				

Figure 7. Full Adder Truth Table

//Source Code

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

module myFA (a,b,cin,cout,sum);
                                    input a,b,cin;
                                    output cout,sum;
                                    wire m,n,p;
                                    myHA g1(.cout(n),.sum(m),.a(a),.b(b));
                                    myHA g2(.cout(p),.sum(sum),.a(cin),.b(m));

                                    or g3(cout,p,n);
endmodule
```

//Testbench

```
`timescale 1ns / 1ps
module myFA_tb();
                                    reg a,b,cin;
                                    wire cout,sum;

                                    myFA utt(.a(a),.b(b),.cin(cin),.cout(cout),.sum(sum));

                                    initial begin
                                       {a,b,cin} = 3'b000;
```

```
                    #10 {a,b,cin} = 3'b001;
                    #10 {a,b,cin} = 3'b010;
                    #10 {a,b,cin} = 3'b011;
                    #10 {a,b,cin} = 4;          //3'b100;
                    #10 {a,b,cin} = 5;          //3'b101;
                    #10 {a,b,cin} = 6;          //3'b110;
                    #10 {a,b,cin} = 7;          //3'b111;
                    #20 $stop;
                end
endmodule
```

Figure 8. Full Adder Source Code and Testbench

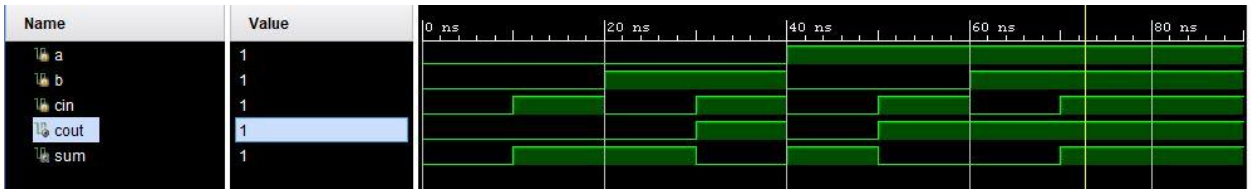


Figure 9. Full Adder Waveform Displaying Inputs and Outputs of Sum and Cout

Results Discussion:
When creating the full adder, it was the first time we got to practice hierarchical design. We created wires to help us pass information from one half adder to another. In the end the design was a success and was able to follow through with the truth table in figure 7.

2.2.3) Ripple Carry Adder Source Code, Testbench, Waveform and Result Discussion

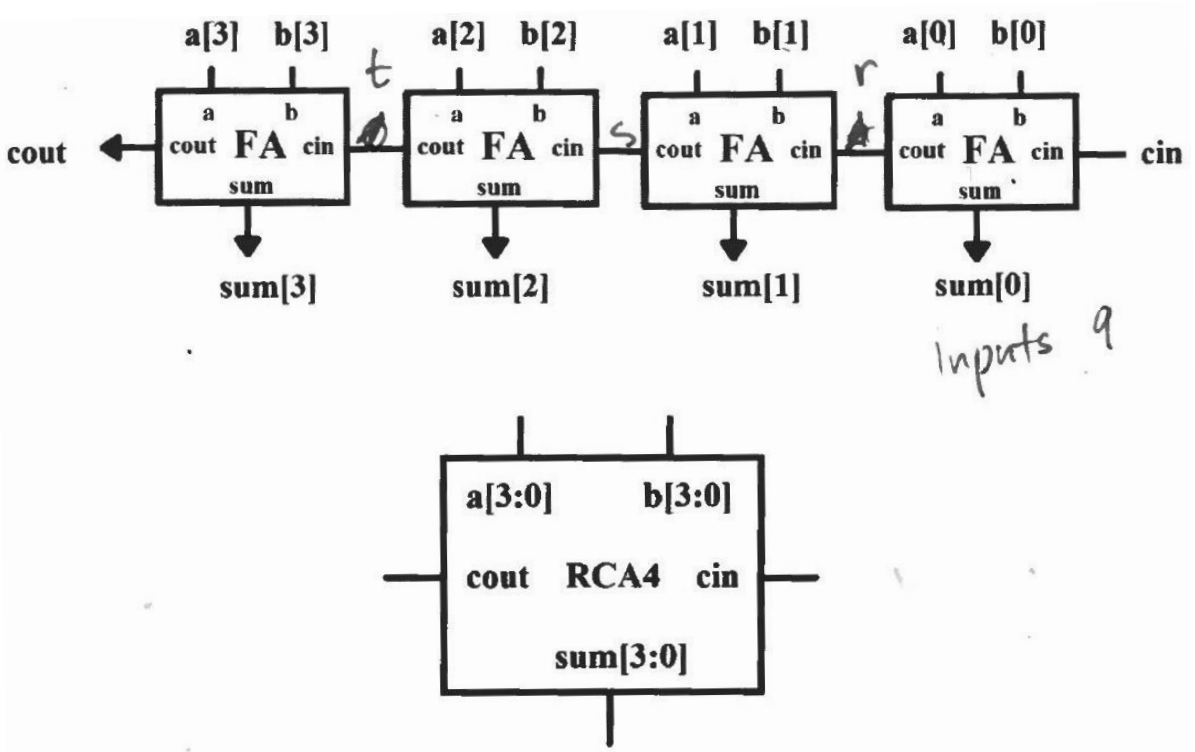


Figure 10. Schematic for RCA4 (Ripple Carry Adders) Composed of Four Full Adders

A 4-bit ripple carry adder does usmmations of 4-bit ninary inputs a[3:0], b[3:0], and 1-bit Cin. The result will generate a 1-bit carry out (cout) signal and 4-bit signals that signifies its sum[3:0]. In figure 10 the schematic for a ripple carry adder is provided. The source code and testbench can be seen in figure 11, and the waveform simulation created with the testbenc can be seen in figure 12.

```
//Source Code
module myHA (input a,b, output cout,sum);
        assign sum = a^b;
        assign cout = a&b;
endmodule

module myFA (a,b,cin,cout,sum);
        input a,b,cin;
        output cout,sum;
        wire m,n,p;
        myHA g1(.cout(n),.sum(m),.a(a),.b(b));
   myHA g2(.cout(p),.sum(sum),.a(cin),.b(m));
        or g3(cout,p,n);
```

```
endmodule

module myRCA4 (a,b,cin,cout,sum);
   input  [3:0] a,b;
   input        cin;
   output [3:0] sum;
   output       cout;
   wire         r,s,t;

   myFA u1(.a(a[0]),.b(b[0]),.cin(cin),.cout(r),.sum(sum[0]));
   myFA u2(.a(a[1]),.b(b[1]),.cin(r),.cout(s),.sum(sum[1]));
   myFA u3(.a(a[2]),.b(b[2]),.cin(s),.cout(t),.sum(sum[2]));
   myFA u4(.a(a[3]),.b(b[3]),.cin(t),.cout(cout),.sum(sum[3]));
endmodule
```

//Testbench

```
`timescale 1ns / 1ps
module myRCA4_tb();
   reg  [3:0] a,b;
   reg        cin;
   wire [3:0] sum;
   wire       cout;

   myRCA4 uut(a,b,cin,cout,sum);

   initial
   begin
      {a,b,cin} = 9'b0;
      for(integer k = 0; k < 512; k = k + 1) begin
         #10 {a,b,cin} = k;
      end
      #20 $stop;
   end
endmodule
```

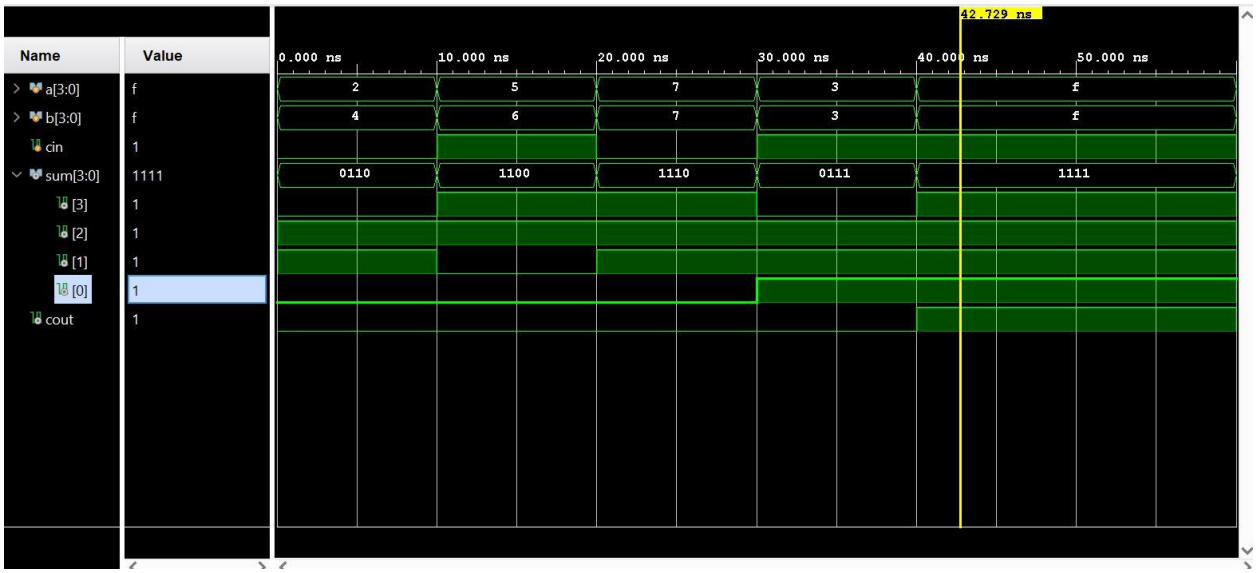


Figure 11. Ripple Carry Adder (RCA4) Simulation Waveform

Results Discussion:
The RCA4 was a simple build because it composed of four Full-Adders, where each Full-Adder take in two one bit inputs, and are connected through a wire that starts as its ccarry in and ends with a carry out. The waveform in Figure 11, shows how when you add the in puts, you have the sum of a four bit number and, when it has a carryout it's wave form is set to high.

2.2.5) 8-Bit Carry Select Adder Code and Testbench

```
//Source Code
module myHA (input a,b, output cout,sum);
        assign sum = a^b;
        assign cout = a&b;
endmodule

module myFA (a,b,cin,cout,sum);
        input a,b,cin;
        output cout,sum;
        wire m,n,p;
        myHA g1(.cout(n),.sum(m),.a(a),.b(b));
   myHA g2(.cout(p),.sum(sum),.a(cin),.b(m));
        or g3(cout,p,n);
endmodule

module myRCA4 (a,b,cin,cout,sum);
   input  [3:0] a,b;
   input       cin;
   output [3:0] sum;
   output      cout;
```

```
    wire        r,s,t;

    myFA u1(.a(a[0]),.b(b[0]),.cin(cin),.cout(r),.sum(sum[0]));
    myFA u2(.a(a[1]),.b(b[1]),.cin(r),.cout(s),.sum(sum[1]));
    myFA u3(.a(a[2]),.b(b[2]),.cin(s),.cout(t),.sum(sum[2]));
    myFA u4(.a(a[3]),.b(b[3]),.cin(t),.cout(cout),.sum(sum[3]));
endmodule

module myMUX(s,y,d0,d1);
    input     d0,d1;
    input     s;
    output    y;
    wire      m,n;

    assign m = (~s)&d0;
    assign n = s&d1;
    assign y = m|n;
endmodule

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

    always@(d0 or d1 or s)
    begin
        if (s == 1)
            y = d1;
        else
            y = d0;
    end
endmodule

module myCSA8(a,b,cin,sum,cout);
    input  [7:0] a,b;
    input        cin;
    output [7:0] sum;
    output       cout;
    wire   [7:0] sa,sb;
    wire   [3:0] c;
    wire         m;

    myRCA4 u0(a[3:0],b[3:0],1'b0,c[0],sa[3:0]);
    myRCA4 u1(a[3:0],b[3:0],1'b1,c[1],sb[3:0]);
    myRCA4 u2(a[7:4],b[7:4],1'b0,c[2],sa[7:4]);
```

```
    myRCA4 u3(a[7:4],b[7:4],1'b1,c[3],sb[7:4]);

    myMUXB g1(sa[3:0],sb[3:0],sum[3:0],cin);
    myMUXB g2(sa[7:4],sb[7:4],sum[7:4],m);

    myMUX f1(cin,m,c[0],c[1]);
    myMUX f2(m,cout,c[2],c[3]);
endmodule
```

```
//Testbench
`timescale 1ns / 1ps
module myCSA8_tb();
    reg  [7:0] a,b;
    reg        cin;
    wire [7:0] sum;
    wire       cout;
    wire [7:0] result;

    assign result = {cout,sum};

    myCSA8 uut(a,b,cin,sum,cout);

    initial begin
            a = 2; b = 4; cin = 0;
        #10 a = 3; b = 3; cin = 1;
        #10 a = 10; b = 2; cin = 0;
        #10 a = 25; b = 35; cin = 1;
        #10 a = 125; b = 125; cin = 1;
        #20 $stop;
    end
endmodule
```

2.2.6) 8-Bit Carry Select Adder Simulation Waveform and Result Discussion

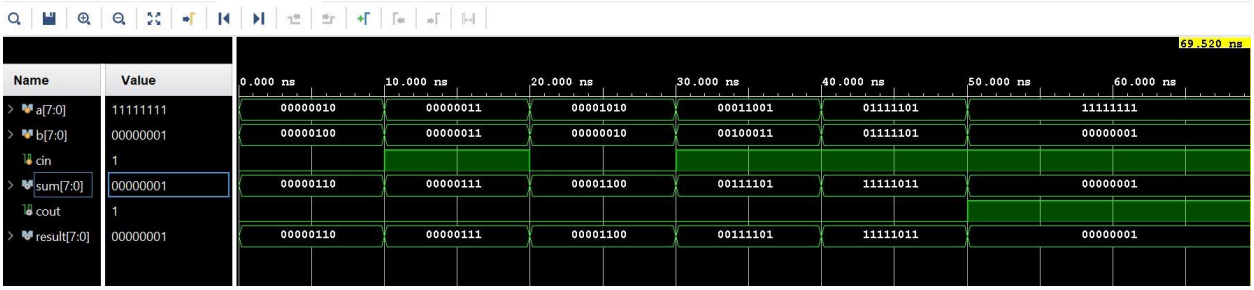


Figure 12. 8-bit Carry Select Adder (CSA8) Simulation Waveform

Results Discussion:
The results of this part meet the expectations, which is that the CSA8 adds two 8-bit numbers properly and all the other Verilog files assited in making this possible. The main focus of this Project was to use combinational logic and hierarchial deisgn. The design was possible through the use of 4-bit RCA's and putting them togeth with multiplexers.

3) 4 by 4 Binary Sequential Multiplier – Project 2

3.1) 4 by 4 Binary Sequential Multiplier Project Schematic and Design Purpose

Schematic:

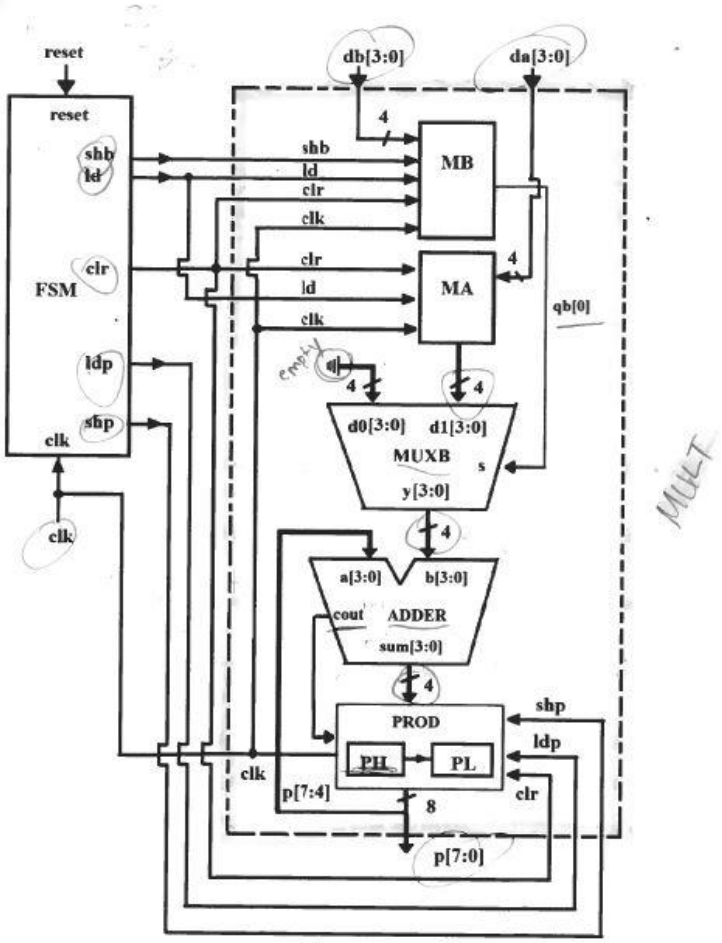


Figure 13. 4 by 4 Binary Sequential Multiplier

Design Purpose:
The purpose of this lab is to create a 4-bit binary shit multiplier with an FSM helping it do the calculations. This part of the lab consist of 7 modules; 2 four-bit registers MA and MB, a multiplexer, 4-bit adder,an 8-bit shfit register that stores the curent product, an FSM, and a module that contains the elements of an FSM. The register MA acts as the loader and the register MB shifts the bits to the right.

3.2) Register MA Source Code, Testbench, Simulation Waveform and Result Discussion

```
//Source Code
module myMA (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
```

```
//Testbench
`timescale 1ns / 1ps
module myMA_tb();
        reg [3:0] da;
        reg clk, clr, load;
        wire [3:0] qa;

        myMA uut (clk, clr, load, da, qa);

        initial clk = 0;
        always #10 clk = ~ clk;
        initial
        begin
                clr = 1; load = 0; da = 4'b1011;
                #22 clr=0;
                load = 1;
                #20 load = 0;
                clr = 1; load = 0; da = 4'b1011;
                #22 clr=0;
                load = 1;
                #20 load = 0;
                #60 $stop;
        end
endmodule
```

Figure 14. Four Bit Load Register (MA) Simulation Waveform

Results Discussion:
We can see in Figure 14 that when a 4-bit value is passed into value da and the load is set to low qa, my load variable is zero. However when load is set to high qa loads the value in da, and when clr is set high it clears the value.

3.3) Register MB Source Code, Testbench, Simulation Waveform and Result Discussion

//Source Code:

```
module myMB (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] };        //This does the shifting
        end
endmodule
```

//Testbench:

```
`timescale 1ns / 1ps
module myMB_tb();
        reg [3:0] db;
        reg clk, clr, load;
        wire [3:0] qb;

        myMB uut (clk,clr,load,sft,db,qb);
```

```
initial clk = 0;
        always #10 clk = ~ clk;
        initial
        begin
                clr = 1; load = 0; db = 4'b1001;
                #22 clr=0;
                load = 1;
                #20 load = 0;
                #60 $stop;
        end
endmodule
```

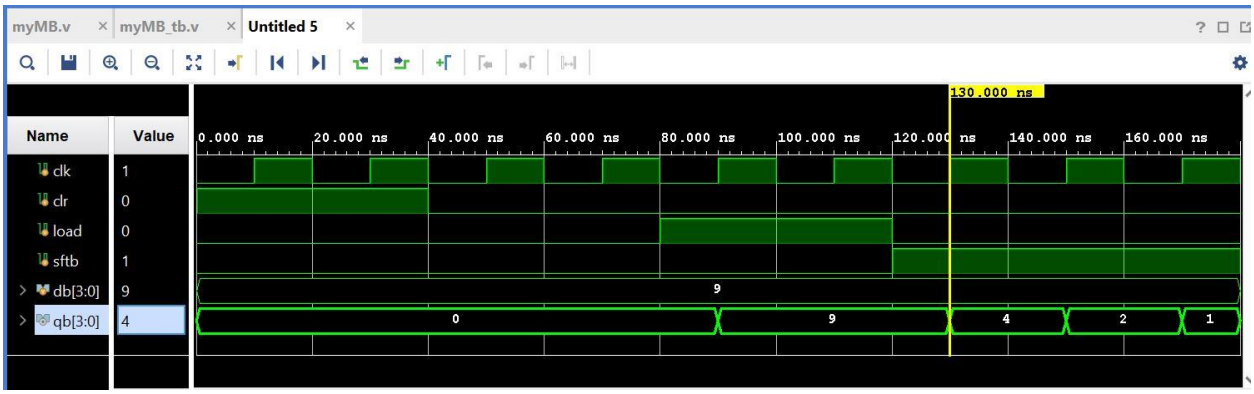


Figure 15. Four Bit Right Shift Register (MB) Simulation Waveform

Results Discussion:
We can see in Figure 15 that my MB module works, where an input db has a value of 9, and then is loaded when the load is set to high. It will shift the bits to the right everytime the clk cycle is high.

3.4) MUXB Source Code, Testbench, Simulation Waveform and Discussion

```
//Source Code:
module myMUXB(d0,d1,y,s);
   input [3:0]  d0,d1;
   input        s;
   output [3:0] y;
   reg    [3:0] y;

   always@(d0 or d1 or s)
   begin
      if (s == 1)
         y = d1;
      else
         y = d0;
   end
endmodule
```

```
//Testbench:
`timescale 1ns / 1ps
module myMUXB_tb();
   reg [3:0]  d0,d1;
   reg        s;
   wire [3:0] y;

   myMUXB utt(d0,d1,y,s);
   initial
   begin
                    d1 = 4'b0000; d0 = 4'b0000; s = 0;
               #10 d1 = 4'b0011; d0 = 4'b1000; s = 0;
               #10 d1 = 4'b0000; d0 = 4'b0001; s = 0;
               #10 d1 = 4'b0010; d0 = 4'b0101; s = 1;
               #10 d1 = 4'b0011; d0 = 4'b1000; s = 1;
               #10 d1 = 4'b1100; d0 = 4'b0011; s = 1;
               #10 $stop;
   end
endmodule
```

Figure 16. MUXB Simulation Waveform

Results Discussion:

The MUXB shows how a value can be selected depending on the select (s) variable, if it is low then it will choose d0 and if it is high then it chooses d1.

3.5) ADDER Source Code,Testbench, Simulation Waveform and Discussion

```
//Source Code:
module myADDER(a,b,cout,sum);
        input  [3:0] a,b;
        output [3:0] sum;
        output       cout;

        assign {cout,sum} = a + b;
endmodule
```

```
//Testbench:
`timescale 1ns / 1ps
module myADDER_tb();
        reg  [3:0] a,b;
    wire [3:0] sum;
        wire       cout;

    myADDER utt(a,b,cout,sum);

        initial begin
                    a = 4'b0000; b = 4'b0000;
                #10 a = 4'b1110; b = 4'b1001;
                #10 a = 4'b1010; b = 4'b0101;
                #10 a = 4'b1111; b = 4'b1111;
                #10 $stop;
        end
endmodule
```

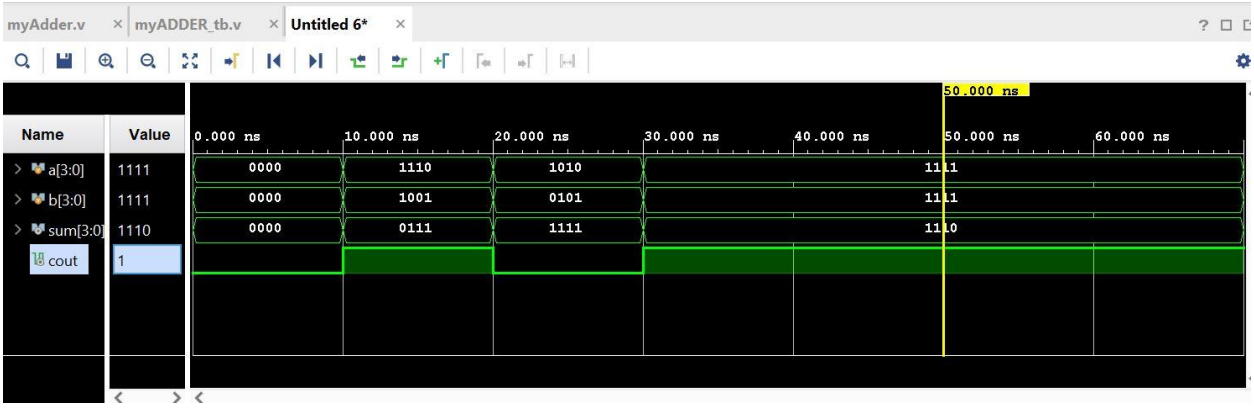


Figure 17. Adder Simulation Waveform

Restults Discussion:
This works the same as a Full Adder, and as seen from Figure 17, it adds binary numbers correctly.

3.6) Register PROD Source Code,Testbench, Simulation Waveform and Discussion

//Source Code:

```
module myPROD(sum,shiftp,loadp,clr,clk,cout,p);
        input      shiftp,loadp,clr,clk,cout;
        input [3:0] sum;
        output[7:0] p;
        reg   [7:0] p;

        always@(posedge clr or posedge clk)
        begin
                if(clr) p <= 0;
                else if (loadp) p[7:3] <= {cout,sum};
                else if (shiftp) begin
                        p[7:3] <= {1'b0, p[7:4];                //PH
                        p[2:0] <= {p[3], p[2:1]};               //PL
                end
        end
endmodule
```

//Testbench:

```
`timescale 1ns / 1ps

module myPROD_tb();
   reg clr,clk,cout,loadp,sftp;
   reg [3:0] sum;
   wire [8:0] p;

   myPROD uut(sum,loadp,sftp,clr,clk,cout,p);

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

   initial begin
      clr = 1; sftp = 0; loadp = 0; cout = 0;
         sum = 4'b0101;
      #40 clr = 0; loadp = 1;
      #60 sftp = 1; loadp = 0;
      #40 loadp = 1; sftp = 0;
         sum = 4'b1110;
      #60 sftp = 1; loadp = 0;
      #100 $stop;
   end
endmodule
```

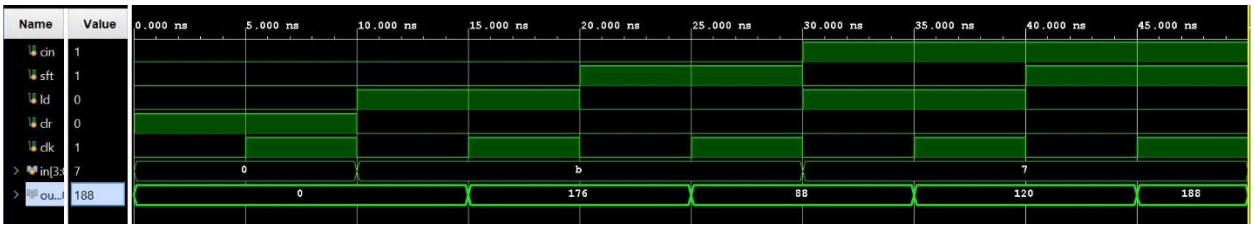


Figure 18. PROD Simulation Waveform

Results Discussion:
The PROD module takes the input from the adder and load it into the left 4 bits of the final product. In the same loading process, the carry in bit is placed into a register named C. This will be needed later when the product bits are shifted once to the right. The reason why the carry in is placed into register C is because once the left 4-bits of the product are changed, the carry in from the adder is changed as well.

3.7) Multiplier Data Path Design, Testbench, Simulation Waveform and Discussion

Data Path Design:

```
//Source Code:
module myMULT(clk,shp,ldp,clr,ld,shb,da,db,p);
   input clk,shp,ldp,clr,ld,shb;
   input [3:0] da,db;
   output [7:0] p;

   wire [3:0] qa,qb,amux,sadd;
   wire cadd;

   myMB g1(.clk(clk),.clr(clr),.load(ld), .sftb(shb),.db(db),.qb(qb));
   myMA g2(.clk(clk),.clr(clr),.load(ld),.da(da),.qa(qa));
   myMUX g3(.d1(qa),.d0(4'b0000),.s(qb[0]),.y(amux));
   myAdder g4(.a(p[7:4]),.b(amux),.cout(cadd),.sum(sadd));
   myPROD g5(.in(sadd),.cin(cadd),.sft(shp),.ld(ldp),.clr(clr),.clk(clk),.out(p));
endmodule
```

```
//Testbench:
`timescale 1ns / 1ps
module myMULT_tb();
   reg clk,shp,ldp,clr,ld,shb;
   reg [3:0] da,db;
   wire [7:0] p;

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

   initial clk = 0;
   always #5 clk = ~clk;
```

```
    initial begin
            da = 11; db = 13; shb = 0; ld = 1;
            ldp = 0; shp = 0; clr = 0;
        #10 clr = 0; ld = 0; ldp = 1;
        #10 ldp = 0; shp = 1;
        #10 shp = 0; shb = 1;
        #10 shb = 0; ldp = 1;
        #10 ldp = 0; shp = 1;
        #10 shp = 0; shb = 1;
        #10 shb = 0;
        #5 clr = 1;
        #5 clr = 0;
        #10 $stop;
    end
endmodule
```

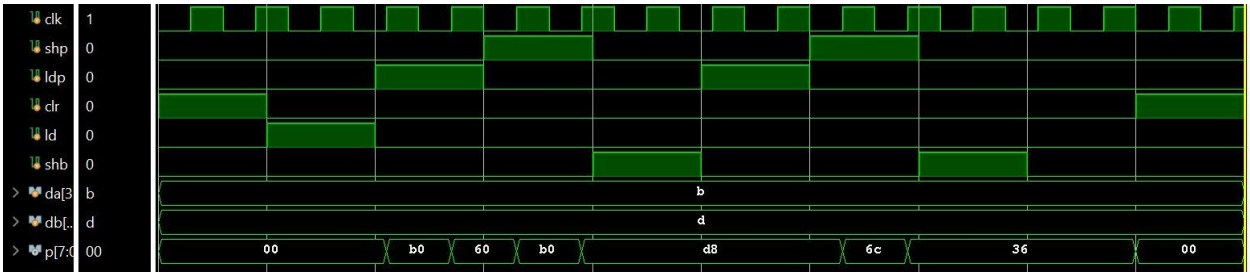
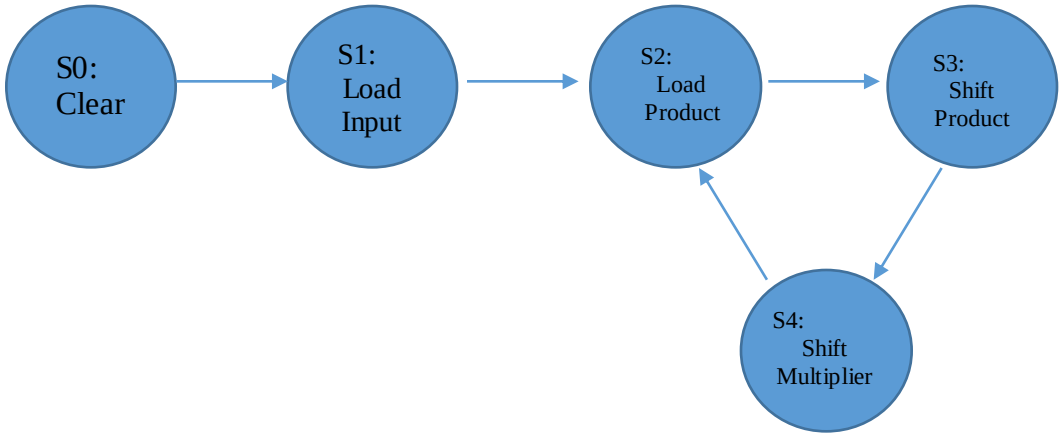


Figure 19. MULT Simulation Waveform

Results Discussion:
In the end this was the easiest module to build but the most crucial, due to the fact that this is where all the calculations will be taking place. As you can see in Figure 19, it was a success leading me to finish this project.

3.8) Finite State Machine State Diagram



3.9) Finite State Machine Design Code, Testbench, Simulation Waveform and Result Discussion

//Source Code:

```
module myFSM(rst,clk,shb,ld,clr,ldp,shp);
   input rst, clk;
   output clr, shb, ld, ldp, shp;
   reg shb, ld, shp, ldp, clr;
   reg [2:0] cs, ns;

   parameter s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
         s3 = 3'b011, s4 = 3'b100;

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

   always@(cs) begin
      case(cs)
         s0:
            begin
               shb <= 0; ld <= 0; ldp <= 0; clr <= 1; shp <= 0;
               ns <= s1;
            end
         s1:
            begin
               ld <= 1; clr <= 0;
               ns <= s2;
            end
         s2:
            begin
               shb <= 0; ld <= 0; ldp <= 1;
               ns <= s3;
            end
         s3:
            begin
               ldp <= 0; shp = 1;
               ns <= s4;
            end

         s4:
            begin
               shp <= 0; shb <= 1;
               ns <= s2;
            end
      endcase
   end
endmodule
```

```
//Testbench:
`timescale 1ns / 1ps
module myFSM_tb();
    reg rst, clk;
    wire clr, shb, ld, ldp, shp;

    myFSM uut (rst,clk,shb,ld,clr,ldp,shp);

    initial clk = 0;
    always #5 clk = ~clk;

    initial begin
            rst = 1;
        #5  rst = 0;
        #80 rst = 1;
        #5  rst = 1;
        #10 $stop;
    end
endmodule
```

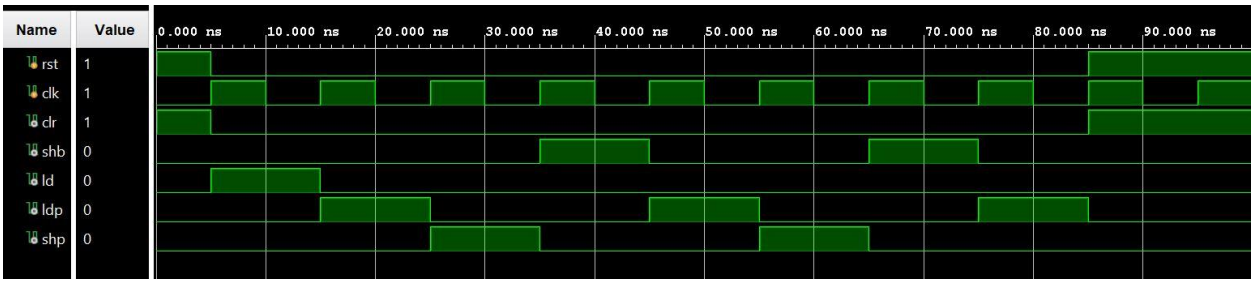


Figure 20. FSM Simulation Testbench

Results Discussion:

The FSM regulates the steps for the multipler (MULT) to work. First it initializes the circuti by clearing the inputs (this is done with the reset switch), then it loads the inputs into the multipler, loads the result of the adder into the product, shifts the product and adds the carry in, shift the multipler to the next bit, then starts over at the product load state.

3.10) Top Level Design of 4 By 4 Binary Sequential Multiplier Source Code and Testbench

```
//Source Code:
module TOP(rst,clk,da,db,p);
    input rst,clk;
    input [3:0] da,db;
    output [7:0] p;
    wire clr, shb, ld, ldp, shp;

    myMULT g1(.clk(clk),.shp(shp),.ldp(ldp),.clr(clr),.ld(ld),.shb(shb),.da(da),.db(db),.p(p));
    myFSM g2(.rst(rst),.clk(clk),.shb(shb),.ld(ld),.clr(clr),.ldp(ldp),.shp(shp));
endmodule
```

```
//Testbench:
`timescale 1ns / 1ps
module TOP_TEST();
    reg rst,clk;
    reg [3:0] da,db;
    wire [7:0] p;

    TOP u1(rst,clk,da,db,p);

    initial clk = 0;
    always #2 clk = ~clk;

    initial begin
          rst = 1;
        #2 rst = 0;
        #2 da = 11; db = 13;
        #50
        #2 $stop;
    end
endmodule
```

3.11) Top Level Design Simulation Waveform and Result Discussion

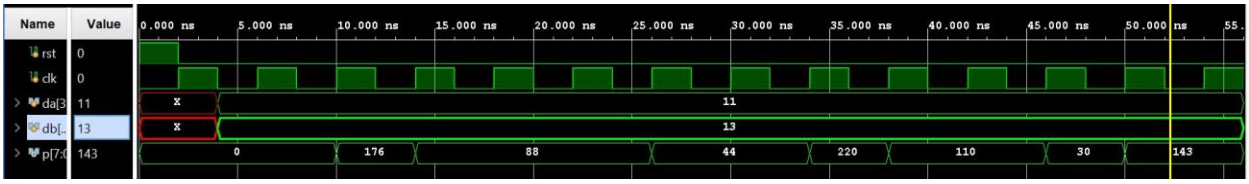


Figure 21. Top Level Design Simulation Waveform

Results Discussion:

The multipler works by looping through the steps of loading and shifting unitl the product of the two inputs is calculated. The use of the FSM means that the user does not have to manulaly load and shift the bits, instead this is done with a clock. As seen in Figure 21 we mulitpled inputs 11 and 13 and got a result of 143.

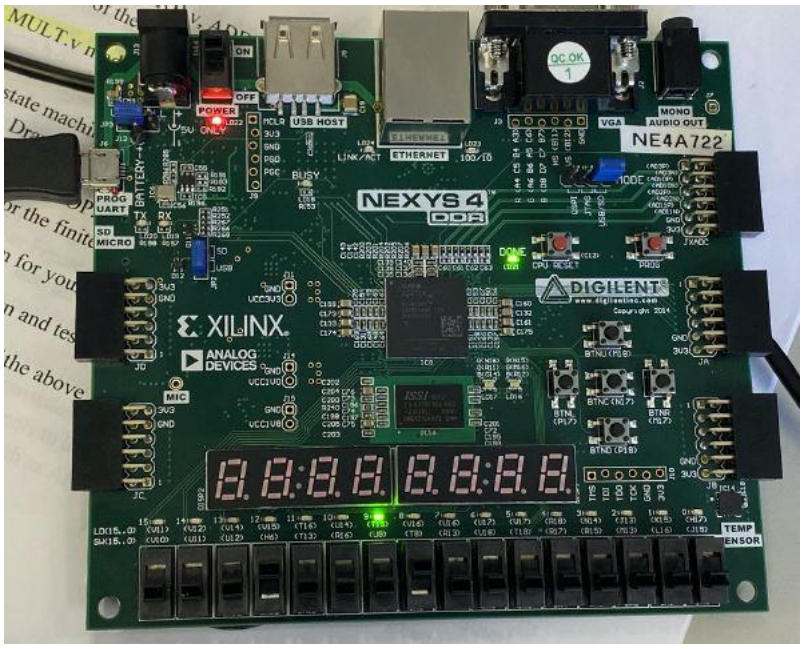
3.12) FPGA User Design Constraints and Picture of Implementation Result on FPGA Board

```
//Constraints:
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk } ];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk } ];


set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { da[0] } ];
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { da[1] } ];
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { da[2] } ];
```

```
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { da[3] } ];
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { db[0] } ];
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { db[1] } ];
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { db[2] } ];
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { db[3] } ];
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { rst } ];


set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { p[0] } ];
set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { p[1] } ];
set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { p[2] } ];
set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { p[3] } ];
set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { p[4] } ];
set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { p[5] } ];
set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { p[6] } ];
set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { p[7] } ];
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { p[8] } ];
```



4) Multiplexed Character Display - Project 3

4.1 Design Purpose

For this final part of the lab, we were expected to display “CPE 166” and our initials. This required us to understand how to access each Cathode of each segment in order to turn them on in this pattern. First we must understand the common Anode which will allow us to understand which letter to turn high in order to activae a certain part of the segment display as seen in Figure 22.

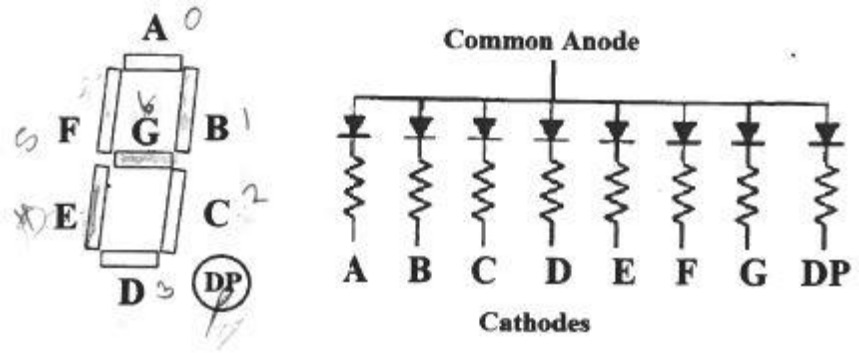


Figure 22. Schematic of One Common Anode on Seven Segment Display Device

4.2 Design Code

```
//Source Code:
module mySeg(clk,seg,dig);
   input       clk;
   output  [7:0] seg,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;    //C
        4'd1 : seg[6:0] = 7'b0001100;    //P
        4'd2 : seg[6:0] = 7'b0000110;    //E
        4'd3 : seg[6:0] = 7'b1111001;    //1
        4'd4 : seg[6:0] = 7'b0000010;    //6
        4'd5 : seg[6:0] = 7'b0000010;    //6
        4'd6 : seg[6:0] = 7'b0001001;    //M?
        4'd7 : seg[6:0] = 7'b0001100;    //P
        default: seg [6:0] = 7'b11111111; //blank
    endcase
  end
endmodule
```

4.3) FPGA User Design Constraints

//Constraint:

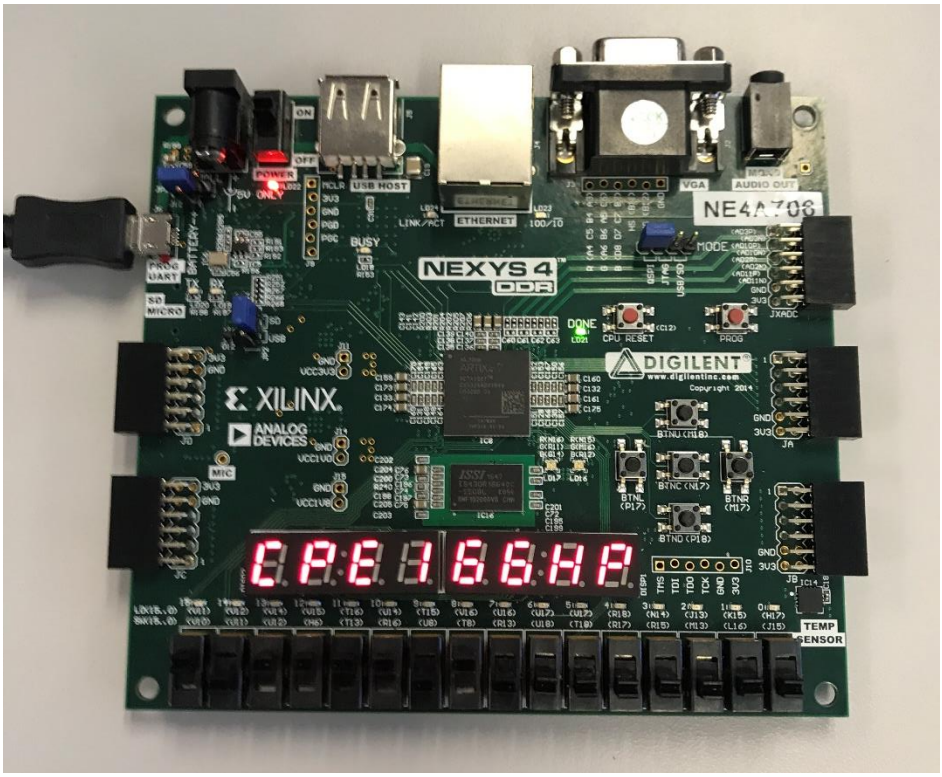
```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; # Sch=clk100mhz

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] }]; # Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { seg[1] }]; # Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { seg[2] }]; # Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { seg[3] }]; # Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { seg[4] }]; # Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { seg[5] }]; # Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { seg[6] }]; # Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { seg[7] }]; # Sch=dp

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

4.4) Picture of Implementation Result on FPGA Board and Result Discussion



Result Discussion:
I was successful in displaying CPE 166 and my initials, unfortunately I could not create an M but I tried my best. Some key concepts of this part of the lab was how to use the contraint files and how the anodes and diodes work.

Conclussion:

Overall, this lab helped me learn more about Verilog, how to use the NEXYS4 FPGA board, and the Vivado enviroment. During this lab we got to practice with combinational and sequential logic. In project 1 we learned how to create a Carry Select Adder by breaking it down through serveral components, and then using hierarchial design to combine all modules and create one working module. In project 2 we were able to proactice our requential designs with a FSM and also creat sequential shift, add and mulitplication circuits. Project 3 taught us how to use the seven segment display and the idea of a common anode. I now have a better understanding of constraint files as well, because we had to use some in order to program and run our codes on the NEXYS4 FGPA board. In the end this lab was challenging but it went really well and I look forward to the next projects.