CPE166 Advanced Logic Design

Lab 2 Report

Student Name: Andy Cha

Date: 3-06-2020

California State University, Sacramento

TABLE OF CONTENTS

Introduction

In this lab we will work with Carry select adder, sequential multiplier, and multiplexed seven segment displays. The purpose of this lab is to give us practice in hierarchical design and design strategy using Verilog. We must be familiar with how carry select adder works, sequential shift/add multiplication algorithm and multiplexed seven segment display. Once we understand how those 3 work, we can simulate the design with waveforms and determine if it is working. Then we will apply our Verilog code into the FPGA setting up the pins allocated to each input and output.

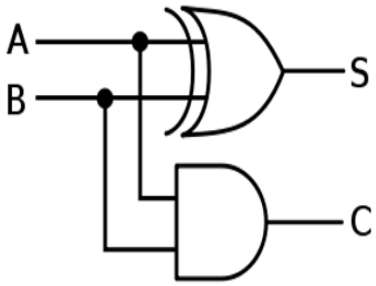
Part 1: 8-bit Carry Select Adder Design

Part 1.1

The first part of this lab we will need to create a half adder first. The truth table is given to us along with the design. From this we can create our Verilog code along with the testbench.

Truth Table

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



Source Code:

half_adder - Notepad

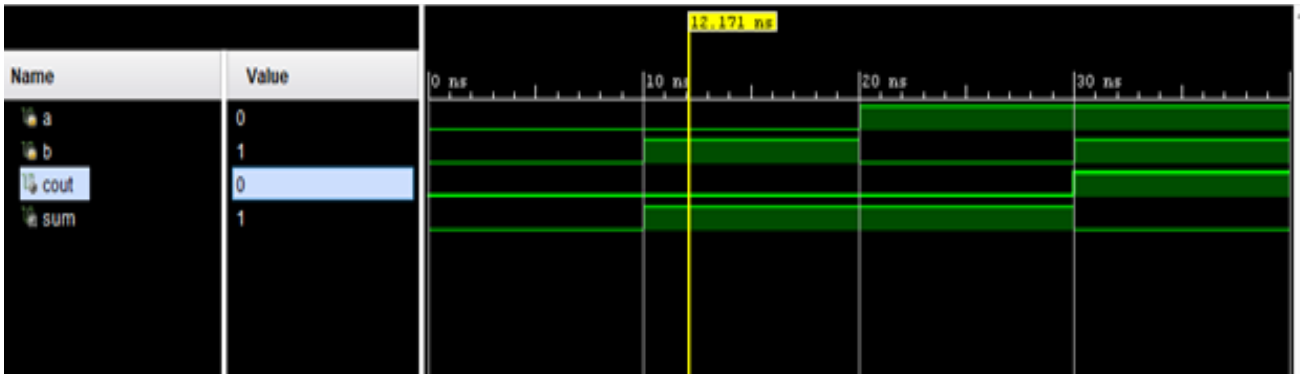
File Edit Format View Help

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

half_adder_tb - Notepad

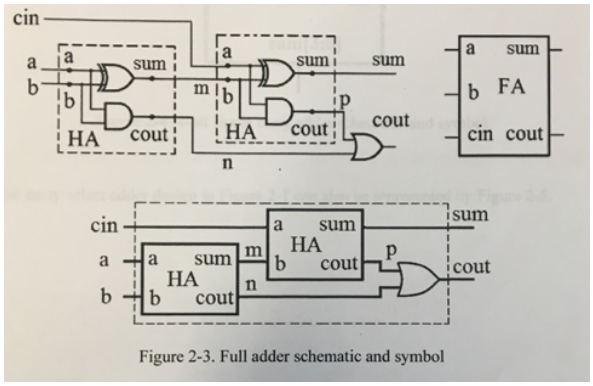
File Edit Format View Help

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

half_adder u1(.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
```



Part 1.2

Once we have created the half adder we can use it to create a full adder. A full adder consists of 2 half adders and one OR gate.



Truth Table

Inputs			Outputs	
A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Logic Equations:

$Sum = A \oplus B \oplus C_{in}$

$C_{out} = (A \oplus B) C_{in} + A \& B$

Source Code:

full_adder - Notepad

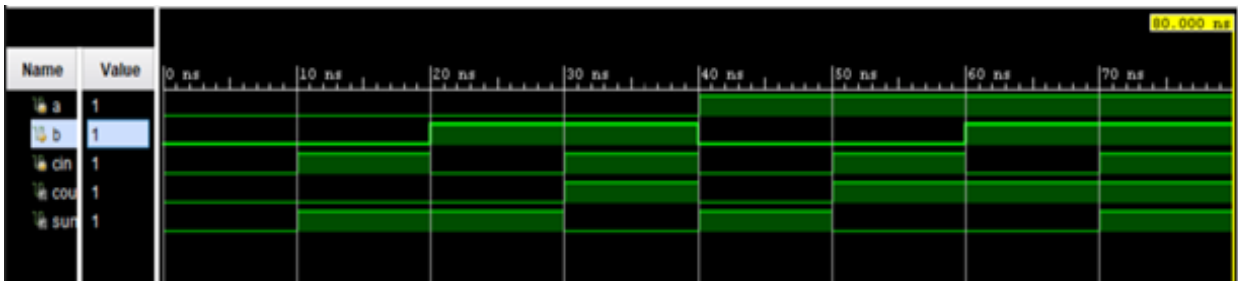
File Edit Format View Help

```
module full_adder(a, b, cin, cout, sum);
input a, b, cin;
output cout, sum;
wire m, n, p;

half_adder g1(.cout(n), .sum(m),.a(a), .b(b));
half_adder g2(.cout(p), .sum(sum), .a(cin), .b(m));
or g3(cout, p, n);

endmodule
```

full_adder_tb - Notepad

File Edit Format View Help

```
module full_adder_tb;
reg a, b, cin;
wire cout, sum;
full_adder u1( .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;
#10 {a, b, cin} = 5;
#10 {a, b, cin} = 6;
#10 {a, b, cin} = 7;
#10 $stop;
end
```



Part 1.3

Once we have the full adders completed we have to create a 4-bit ripple carry adder(RCA4). We connect 4 full adders together and move the cout to the next full adder's cin. We also have to change inputs to [3:0] a, b and create a name for the wires in the middle which will be [3:0] m.

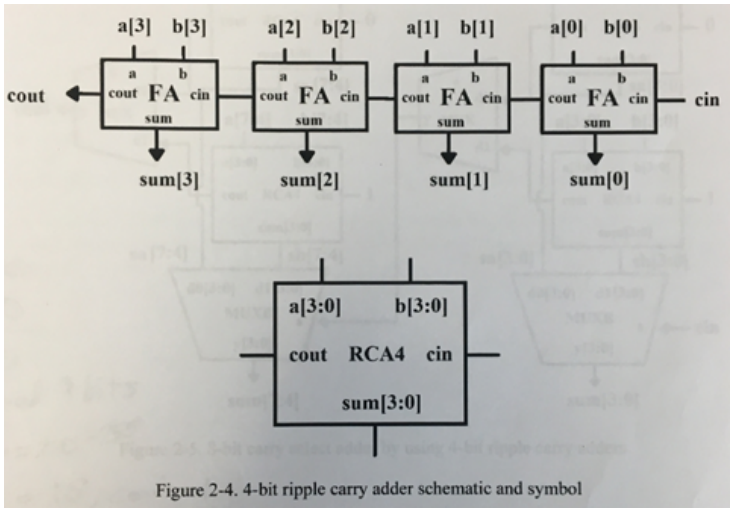


Figure 2-4. 4-bit ripple carry adder schematic and symbol

Source Code:

fa_ripplecarry - Notepad

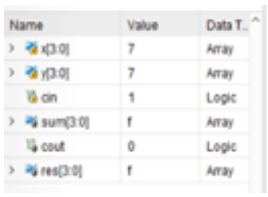
File Edit Format View Help

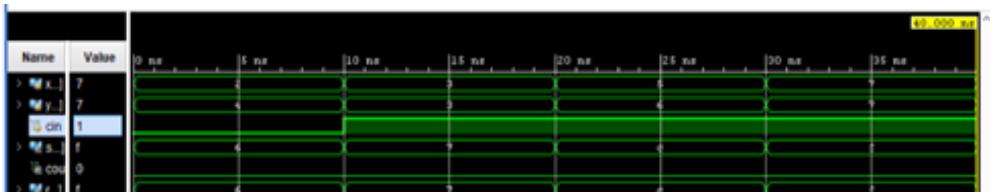
```
module fa_ripplecarry(a, b, cin, cout, sum);
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;

wire [3:0] m;

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

endmodule
```

fa_ripplecarry_tb - Notepad

File Edit Format View Help

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

assign res = { cout, sum };

fa_ripplecarry u1( a, b, cin, cout, sum );
initial
begin
      a = 2; b = 4; cin = 0;
 #10 a = 3; b = 3; cin = 1;
 #10 a = 5; b = 6; cin = 1;
 #10 a = 7; b = 7; cin = 1;
 #10 $stop;

end

endmodule
```





Part 1.4

Here we will use the multiplexer that was created in Lab 1.

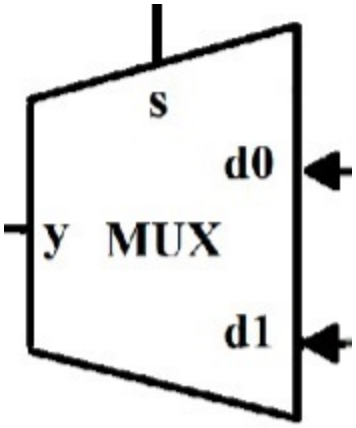


Table 2-3. MUX truth table

Input	Output
s	y
0	d0
1	d1

Source Code:

mux - Notepad

File Edit Format View Help

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

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

endmodule
```

mux_tb - Notepad

File Edit Format View Help

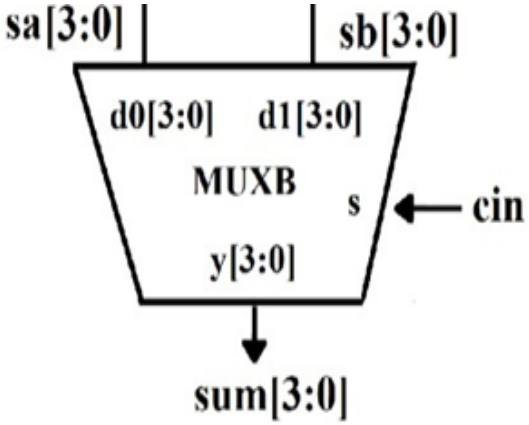
```
module mux_tb;
reg[1:0]d;
reg s;
wire y;
mux2to1 u1(.d(d), .s(s), .y(y));
initial
begin

        d=2'b10; s=0;
        #10;
        d=2'b01; s=0;
        #10;
        d=2'b10; s=1;
        #10
        d=2'b01; s=1;
        #20 $stop;

end
endmodule
```

Part 1.5

Here We will use the multiplexer that was created from Lab 1. We have to modify it to take 4 bits instead of one bit. We can use the same Testbench as the mux but we need to change the instance name to “muxb” and change the inputs to 4 bits. Then we also need to include a cin.

Table 2-4. MUXB truth table

Input	Output
s	y[3:0]
0	d0[3:0]
1	d1[3:0]

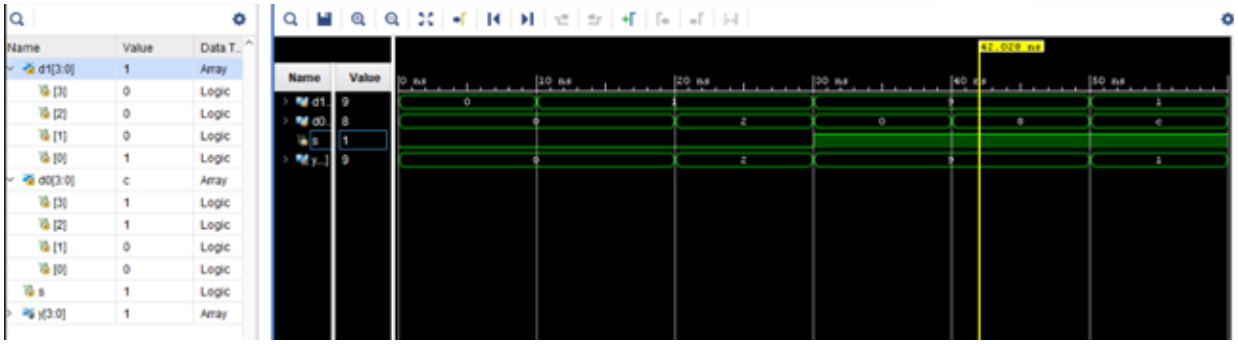


Source Code:

muxb - Notepad

File Edit Format View Help

```
module muxb(d0, d1, s, y);
input [3:0] d0;
input [3:0] d1;
input s;
output [3:0] y;
reg [3:0] y;
always@(s)
        begin
        if(s > 0)
                y <= d1;
        else
                y <= d0;
        end
endmodule
```



Part 1.6

Now we have all the components we need, we can design the carry select adder(CSA8) by connecting 4 RCA4, 2 MUXB, and 2 MUX. In Verilog we can make 8 instances and connect each one with the correct wires.

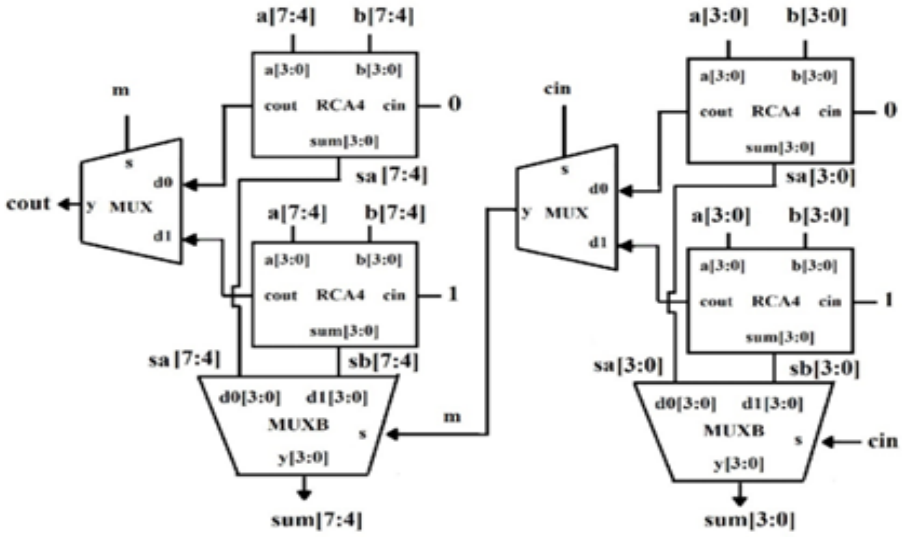


Figure 2-5. 8-bit carry select adder by using 4-bit ripple carry adders

Source Code:

cs8.demo - Notepad

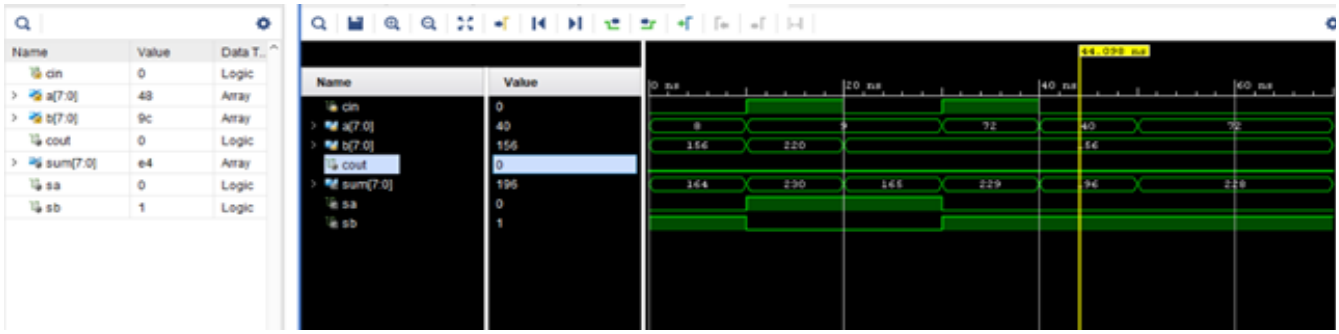
File Edit Format View Help

```
module csa8(a, b, cin, cout, sum);
input [7:0] a, b;
input cin;
output [7:0] sum;
output cout;
wire [3:0] z ;
wire m ;
wire [7:0] sa;
wire [7:0] sb;

fa_ripplecarry g1(.cout(z[0]), .sum(sa[3:0]), .cin(0), .a(a[3:0]), .b(b[3:0]) );
fa_ripplecarry g2(.cout(z[1]), .sum(sb[3:0]), .cin(1), .a(a[3:0]), .b(b[3:0]) );
muxb           g3(.d0(sa[3:0]), .d1(sb[3:0]), .s(cin), .y(sum[3:0]) ) ;

mux            g4(.d1(z[1]), .d0(z[0]), .s(cin), .y(m) );

muxb           g5(.d0(sa[7:4]), .d1(sb[7:4]), .s(m), .y(sum[7:4]) ) ;
fa_ripplecarry g6(.cout(z[2]), .sum(sa[7:4]), .cin(0), .a(a[7:4]), .b(b[7:4]) );
fa_ripplecarry g7(.cout(z[3]), .sum(sb[7:4]), .cin(1), .a(a[7:4]), .b(b[7:4]) );
mux            g8(.d1(z[3]), .d0(z[2]), .s(m), .y(cout) );

endmodule
```

csa8_tb - Notepad

File Edit Format View Help

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

csa8 u1( a, b, cin, cout, sum);

initial
begin
     a = 8'b0000_0001; b = 8'b1000_0011; cin = 1'b0;
     #10
     a = 8'b0100_0101; b = 8'b1100_0001; cin = 1'b1;
     #10
     a = 8'b1010_1010; b = 8'b0101_0101; cin = 1'b0;
     #10
     a = 8'b0000_1111; b = 8'b1111_0000; cin = 1'b1;
     #10 $stop;

end

endmodule
```



Result Discussion:

The result of this part was that the CSA8 adds two 8-bit numbers properly. It took a lot of time designing all the Verilog files and creating testbenches for each one to determine that there were no issues with it. It was better to check each piece to see if it was right than to test the final product and guess where the error was out of all the files. Some issues I came across were creating the CSA8. Putting all the files together took some time and had to overlook the diagram over and over. The goal of this part was how to use combinational logic and putting it together to create an 8 bit carry adder.

Part 2: 4 by 4 Binary Sequential Multiplier Design

Part 2.1:

This part of the lab uses a shift and add algorithm. The picture below is an example of a 4 by 4 binary sequential multiplication example. Every clock cycle, the multiplier is shifted to the right by one bit and its value is tested.

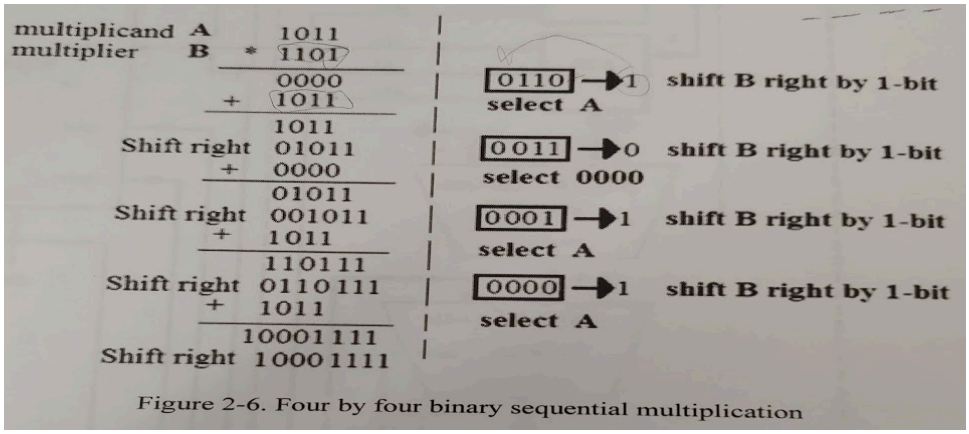


Figure 2-6. Four by four binary sequential multiplication

To create this in Verilog we need to design the circuit below. We need a FSM, dffa, dffb, muxb, adder, and a prod circuit. Once we created each one we can wire them accordingly. We will need to create a mult file where we do the circuits within the dotted line. Then we need to create a top level that takes everything and puts it together.

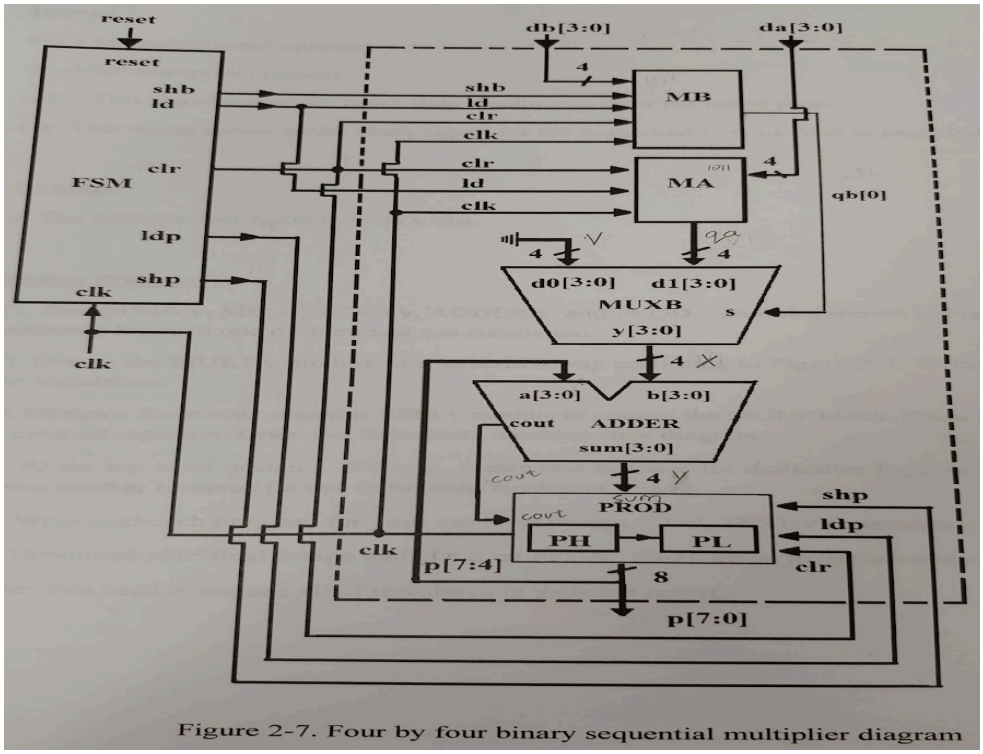


Figure 2-7. Four by four binary sequential multiplier diagram

Part 2.2:

Source Code:

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

assign {cout,sum} = a + b;

endmodule
```

```
module  adder_tb;
reg     [3:0]   a;
reg     [3:0]   b;
reg     clk;
reg     add;
wire    cout;
wire    [3:0]sum;
adder   u1(.a(a), .b(b), .cout(cout), .sum(sum), .clk(clk), .add(add));

initial clk = 1;
always  #10 clk = ~clk;
initial
begin
add = 0;
a = 4'b0000; b = 4'b0000;
#15;
add=1; a = 4'b1101; b = 4'b1110;
#20;
add=0; a = 4'b0001; b = 4'b0000;
#20;
add = 1;
#20;
add = 0;a = 4'b1000; b = 4'b1011;
#20;
add=1;
#20 $stop;
end

endmodule
```

dffa - Notepad

File Edit Format View Help

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

reg [3:0] qa;

always@(posedge clk or posedge clr)
begin
if(clr)
        qa <= 4'b0000;
else if(ld)
        qa <= da;
end

endmodule
```

dffa_tb - Notepad

File Edit Format View Help

```
module dffa_tb;
reg clr, clk, load;
reg [3:0] da;
wire [3:0] qa;

dffa u1(.clr(clr), .clk(clk), .load(load), .da(da), .qa(qa));

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

initial
begin
clr = 1; load = 0; da = 4'b1010;
#10 clr =0;
load =1;
#10;
load =0;
#10;
clr =1;
#10;
$stop;

end

endmodule
```

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

reg [3:0] qb;

always@(posedge clk or posedge clr)
begin
if(clr)
        qb <= 4'b0000;
else if(ld)
        qb <= db;
else if (shb)
        qb <= { 1'b0, qb[3:1] };
end

endmodule
```

```
module  dffb_tb;
reg clk, clr, load, shift;
reg [3:0] db;
wire [3:0] qb;

dffb u1(.clr(clr), .clk(clk), .load(load), .shift(shift), .db(db), .qb(qb));

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

initial
begin
clr = 1; load = 0; shift = 0; db = 4'b1010;
#10 clr =0;
load =1;
#10;
load =0; shift =1;
#50;
clr =1;
#10;
$stop;

end


endmodule
```

```
module  fsm(reset, clk, clr, shb, ld, ldp, shp);
input reset, clk;
output clr, shb, ld, ldp, shp;

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

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, s10 = 4'b1010, s11 = 4'b1011, s12 = 4'b1100;

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

always @ (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 = s10;
                s10:            ns = s11;
                s11:            ns = s12;
                s12:            ns = s12;
                default:        ns = s0;
        endcase
end
```

```
always @ (cs)
begin
        case(cs)
                s0: begin
                        clr = 1; ld = 0; shp = 0; shb = 0; ldp = 0;
                        end
                s1: begin
                        clr = 0; ld = 1; shp = 0; shb = 0; ldp = 0;
                        end
                s2: begin
                        clr = 0; ld = 0; shp = 0; shb = 0; ldp = 1;
                        end
                s3: begin
                        clr = 0; ld = 0; shp = 1; shb = 0; ldp = 0;
                        end
                s4: begin
                        clr = 0; ld = 0; shp = 0; shb = 1; ldp = 0;
                        end
                s5: begin
                        clr = 0; ld = 0; shp = 0; shb = 0; ldp = 1;
                        end
                s6: begin
                        clr = 0; ld = 0; shp = 1; shb = 0; ldp = 0;
                        end
                s7: begin
                        clr = 0; ld = 0; shp = 0; shb = 1; ldp = 0;
                        end
                s8: begin
                        clr = 0; ld = 0; shp = 0; shb = 0; ldp = 1;
                        end
                s9: begin
                        clr = 0; ld = 0; shp = 1; shb = 0; ldp = 0;
                        end
               s10: begin
                        clr = 0; ld = 0; shp = 0; shb = 1; ldp = 0;
                        end
               s11: begin
                        clr = 0; ld = 0; shp = 0; shb = 0; ldp = 1;
                        end


               s12: begin
                        clr = 0; ld = 0; shp = 0; shb = 0; ldp = 0;
                        end


               default: begin
                           clr = 0; ld = 0; shp = 0; shb = 0; ldp = 0;
                           end
        endcase
end

endmodule
```

mux - Notepad

File Edit Format View Help

```
module  mux(s, a, b, y);

input [3:0] a, b;
input s;
output [3:0]y;


assign y = s? b : a;
endmodule
```

mux_tb - Notepad

File Edit Format View Help

```
module mux_tb;

reg[3:0] a, b;
reg s;
wire [3:0] y;
mux u1(.s(s), .a(a), .b(b), .y(y));
initial
begin
        a = 4'b1100; b = 4'b0011; s = 0;
        #10;
        a = 4'b1100; b = 4'b0011; s = 1;
        #20;
        $stop;
end
endmodule
```

preg - Notepad

File Edit Format View Help

```
module  preg(sum, shp, ldp, clr, p, clk, cout);
input clk, clr, ldp, shp, cout;
input [3:0] sum;
output [7:0] p;
reg [7:0] p;

always @ (posedge clk or posedge clr)
begin
if (clr)
        p <= 8'b0000;
else if (ldp)
        p[7:3] <= {cout,sum};
else if (shp)
        begin
             p[7:3] <= {1'b0,p[7:4]};
             p[2:0] <= {p[3],p[2:1]};
        end

end

endmodule
```

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

wire [3:0] gnd, a, qa, mux_out, add_out, qb;
wire cout;

assign gnd = 4'b0000;   //connected to ground

dffb u1(.clr(clr), .clk(clk), .ld(ld), .shb(shb), .db(db),.qb(qb));
dffa u2(.clr(clr), .clk(clk), .ld(ld), .da(da), .qa(qa));
mux u3(.s(qb[0]), .a(gnd), .b(qa), .y(mux_out));
adder u4(.b(mux_out), .a(p[6:3]), .cout(cout), .sum(add_out));
preg u5(.cout(cout), .sum(add_out), .shp(shp), .ldp(ldp), .clr(clr), .p(p), .clk(clk));

endmodule
```

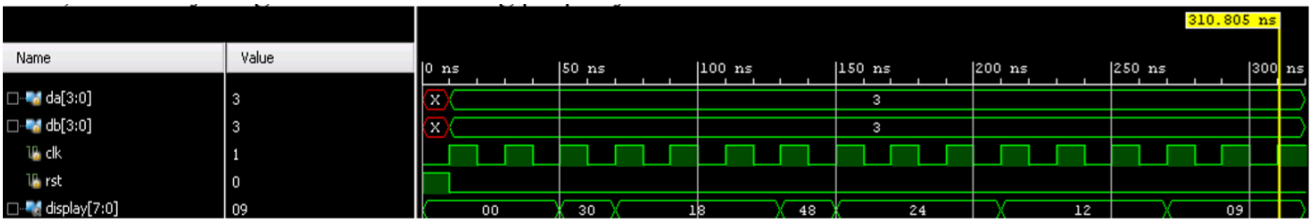
```
module  top(da, db, p, clk, reset);
input [3:0] da, db;
input clk, reset;
output [7:0] p;

wire shb, ld, clr, ldp, shp;

mult u2(.da(da), .db(db), .p(p), .ld(ld), .shb(shb), .clr(clr), .clk(clk), .shp(shp), .ldp(ldp));
fsm u1(.reset(reset), .clk(clk), .shb(shb), .ld(ld),.clr(clr), .ldp(ldp), .shp(shp));

endmodule
```

```
module  top_tb;
reg [3:0] da, db;
reg clk, reset;
wire [7:0] p;

top uut(.da(da), .db(db), .p(p), .clk(clk), .reset(reset));

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

initial
begin
da = 4'b1011; db = 4'b1101; reset = 1;
#20
reset = 0;
#400 $stop;
end

endmodule
```

Pins Part 2_2 - Notepad

File Edit Format View Help

```
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 V10   IOSTANDARD LVCMOS33 } [get_ports { da[3] }];
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { da[2] }];
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { da[1] }];
set_property -dict { PACKAGE_PIN H6   IOSTANDARD LVCMOS33 } [get_ports { da[0] }];
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { db[3] }];
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { db[2] }];
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports { db[1] }];
set_property -dict { PACKAGE_PIN T8   IOSTANDARD LVCMOS33 } [get_ports { db[0] }];
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { reset }];

set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { p[7] }];
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { p[6] }];
set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 } [get_ports { p[5] }];
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { p[4] }];
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { p[3] }];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { p[2] }];
set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 } [get_ports { p[1] }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { p[0] }];
```



Result Discussion:

It took me a long time to get the 4 binary segmental multiplier multiplied values properly. I had issues mainly with mult and top. I also messed up the wiring a lot. The key findings of this part of the lab was how to put these components together and use sequential logic with the FSM to control all of these components. In this part of the lab, we figured out how to create shift registers and FSM and got complex circuits to work together. We had a multiplier on the FPGA that produces the correct values. The lab taught us a lot about how Verilog, Vivado and the NEXYS board works by setting pins and LEDs.

Part 3: Character Display on 8-Digit Multiplexed Seven Segment Displays

Part 3.1:

Here the anodes of the 7 segments are tied together as AN7, AN6, AN5, AN4, AN3, AN2, AN1, and AN0. For the cathodes they remain separated as CA, CB, CC, CD, CE, CF, CG, and DP. The FPGA drives the anode signals and corresponding cathode patterns of each digit in continuous fast-paced signals.

The schematic of the internal structure of one common anode seven segment display device is shown in Figure 2-8.

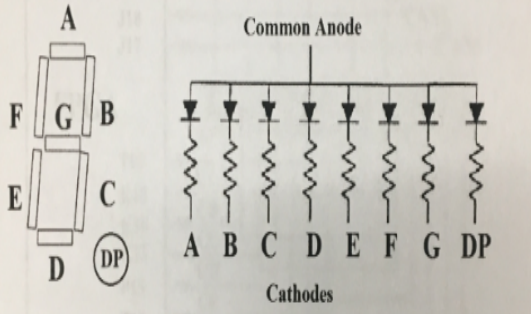


Figure 2-8. Internal schematic of one common anode seven segment display device

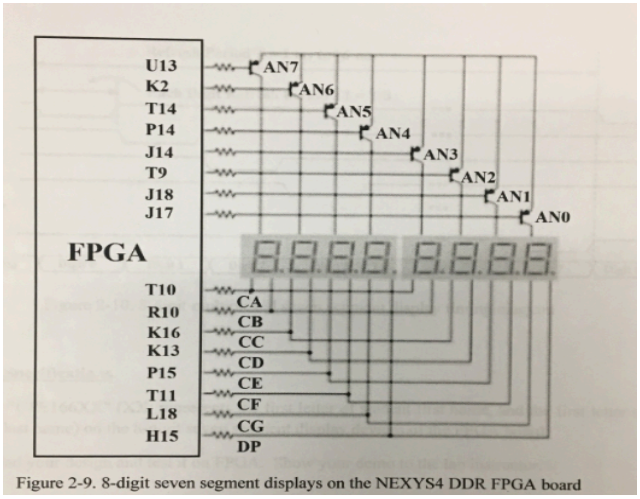


Figure 2-9. 8-digit seven segment displays on the NEXYS4 DDR FPGA board

Part 3.2:

Source Code:

```
module fpga(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 = b'b10111111;
                        end
          3'b111:
                        begin
                                dd = 4'd0;
                                an = 8'b01111111;
                        end
        end
        endcase
assign dig = an;

always @ (dd)
begin
        seg[7] = 1'b1;
        case(dd)
                4'd0: seg[6:0] = 7'b1000110     //display C
                4'd1: seg[6:0] = 7'b0001100     //display P
                4'd2: seg[6:0] = 7'b0000110     //display E
                4'd3: seg[6:0] = 7'b1111001     //display 1
                4'd4: seg[6:0] = 7'b0000010     //display 6
                4'd5: seg[6:0] = 7'b0000010     //display 6
                4'd6: seg[6:0] = 7'b0001000     //display A
                4'd7: seg[6:0] = 7'b1000110     //display C

                default: seg[6:0] = 7'1111111;  //blank
        endcase
        end
endmodule
```

```
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]
```

Result Discussion:

I managed to manipulate the 7-segment display. This lab was pretty easy with no issues. The code was already given to us online. I just had to understand how to configure the pins and manipulate the bytes so that the LED can light my initials. The goal of this part was how to write constraint files as well as how the anodes and diodes work with the seven-segment display to show values. It also helped us get familiar with the Vivado program and FPGA.

Conclusion

In conclusion, this lab helped me learn more about Verilog. It taught me how to code with hierarchy design and writing testbenches. I also got a better understanding of how waveforms are read and how to use it to check if our code was correct. The first part was simple. Create each circuit design and connect them at the end. Part 2 then introduced us to sequential design with the FSM. This part was the hardest for me. I struggled getting the right output. Part 3 taught us how to use a multiplexed seven segment display. Overall, This lab really gave me a better understanding of how coding in Verilog works and the steps to creating a circuit. It also helped to use the FPGA and seven segment display to see how the software and hardware interact with each other.