Sam Lee

CPE 166 – 03

Advanced Logic Design Lab

Wednesday: 5PM - 7:50 PM

Lecture Professor: Jing Pang

Lab Professor: Eric Carmi

Lab #2 Report

Contents

[Introduction 3](#_gjdgxs)

[Part 1: 8-Bit Carry Select Adder Design 3](#_30j0zll)

[Design Purpose 3](#_1fob9te)

[Engineering Data: 3](#_3znysh7)

[Source Code](#_2et92p0) 8

[Simulation Waveforms](#_tyjcwt) 13

[Results Discussion](#_3dy6vkm) 15

[Part 2 : 4 By 4 Binary Sequential Multiplier Design](#_1t3h5sf) 16

[Design Purpose 16](#_4d34og8)

[Engineering Data](#_2s8eyo1) 16

[Source Code](#_17dp8vu) 20

[User Constraint File](#_26in1rg) 28

[Simulation Waveforms](#_35nkun2) 29

[Results Discussion](#_lnxbz9) 31

[Part 3: Character Displays On 8-Digit Multiplexed Seven Segment Displays](#_1ksv4uv) 32

[Design Purpose](#_44sinio) 32

[Engineering Data](#_2jxsxqh) 33

[Source Code](#_z337ya) 33

[User Constraint File](#_3j2qqm3) 35

[Results Discussion](#_1y810tw) 36

[Conclusion](#_4i7ojhp) 37

# Introduction

This lab was a three part lab that served as an introduction to hierarchical design in Verilog. We were tasked with designing an 8-bit carry select adder, 4x4 binary sequential multiplier design, using multiplexed seven segment displays to display out initials and CPE 166 on the NEXYS4 DDR board. In this lab we learned hierarchical design strategy using VHDL, carry select adder designed, sequential shift/add multiplication algorithm, how to use multiplexed seven segment display, simulate a Verilog deign, and learn to use a NEXYS4 DDR FPGA board. We are doing this lab because while Lab 1 introduced us to Verilog and VIVADO, we did not explore how to design FPGAs so in Lab 2 we explore how to do these via the concepts in this lab, which is a carry select adder, learn to create a sequential shift binary multiplier and a multiplexed seven segment display. These concepts are important because they explore many elements of circuit design and how to create a hierarchical design like combinational and sequential logic, creating a finite state machine, etc.

# Part 1: 8-Bit Carry Select Adder Design

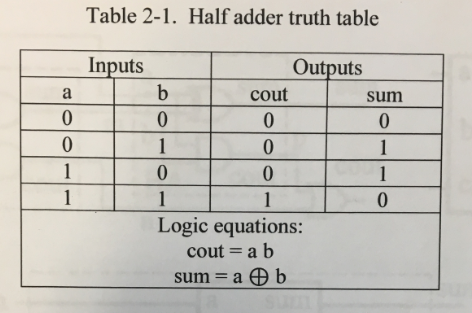
## Design Purpose:

In order to create an 8 bit adder, I first built 2 half adders that consist of an XOR and an AND gate. Half adders can add two 1-bit together. Then, by calling on 2 half adders, I was able to create a full adder which can add two 2 bits variable. Next, I built a 4 bit ripple carry adder using 4 full adders in order to add two variable of 4-bits. Lastly, Lastly, I moved on to build a multiplexer which is essentially a switch that output 1 of two input depending on an extra input. Finally, I built the 8-bit ripple carry adder with the modules connected.

## Engineering Data:

Figure 1: Half Adder Truth Table, Schematic, and Symbol

A half adder adds two 1-bit binary inputs and generates two 1-bit outputs, in which includes 1-bit carry out.



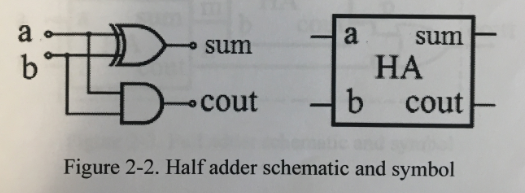
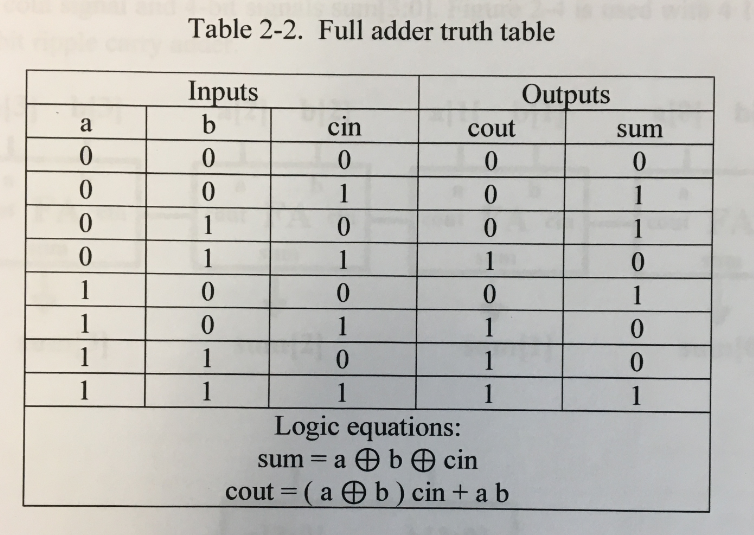


Figure 3: Full Adder Truth Table, Schematic, and Symbol

A full adder is a logical circuit that performs an addition operation on three one-bit binary numbers. The full adder produces a sum of the three inputs and carry value.



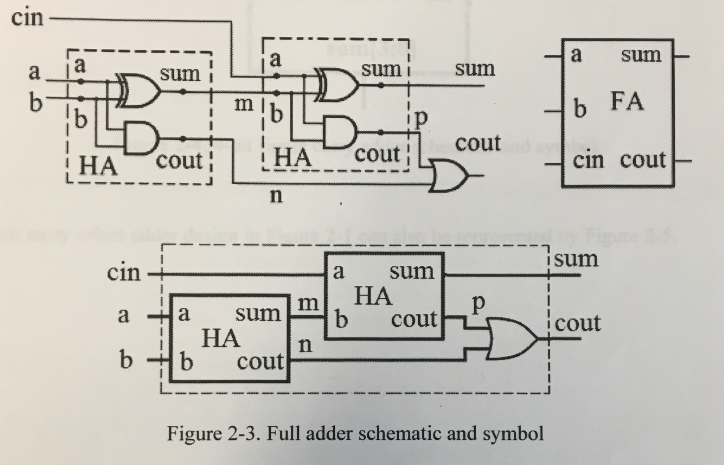


Figure 5: 4-Bit Ripple Carry Adder Schematic

In a ripple carry adder, the sum and carry out bits of any half adder stage is not valid until the carry in of that stage occurs. Propagation delays inside the logic circuitry is the reason behind this. Propagation delay is time elapsed between the application of an input and occurrence of the corresponding output. Consider a NOT gate, When the input is “0” the output will be “1” and vice versa. The time taken for the NOT gate’s output to become “0” after the application of logic “1” to the NOT gate’s input is the propagation delay here. Similarly, the carry propagation delay is the time elapsed between the application of the carry in signal and the occurrence of the carry out (Cout) signal. Circuit diagram of a 4-bit ripple carry adder is shown below.

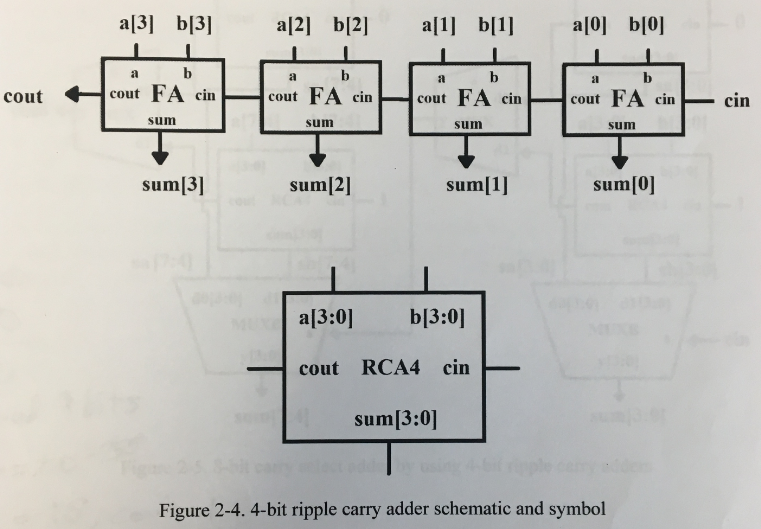
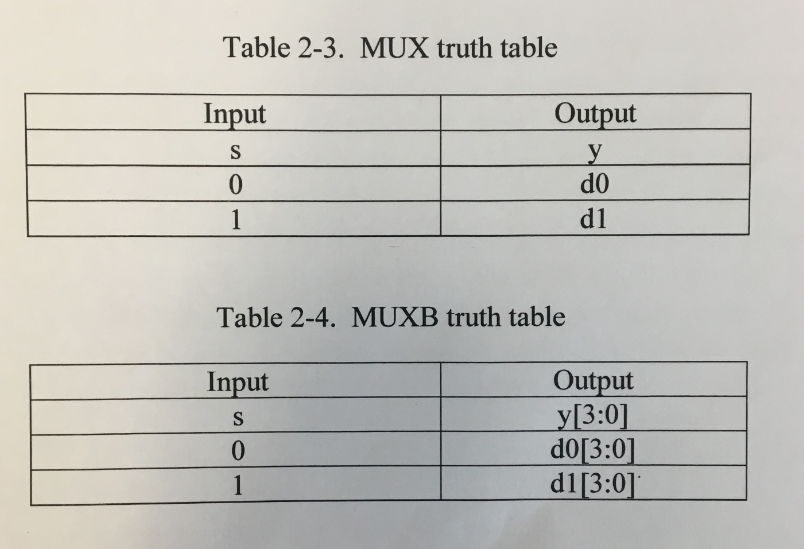


Figure 6: MUX and MUXB Truth Table and Schematic

A multiplexer (or mux) is a common digital circuit used to mix a lot of signals into just one. If you want multiple sources of data to share a single, common data line, you’d use a multiplexer to run them into that line. Multiplexers come in all sorts of shapes and sizes, but they’re all made out of logic gates. Every multiplexer has at least one select line, which is used to select which input signal gets relayed to the output. In a 2-to-1 multiplexer, there’s just one select line. More inputs means more select lines: a 4-to-1 multiplexer would have 2 select lines, an 8-to-1 has 3, and so on (2*n* inputs requires *n* select lines). Think of a mux as a "digital switch". The select line is the throw on the switch, it chooses which of the many inputs get to be the output.



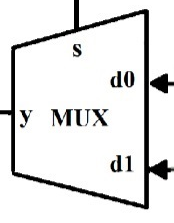
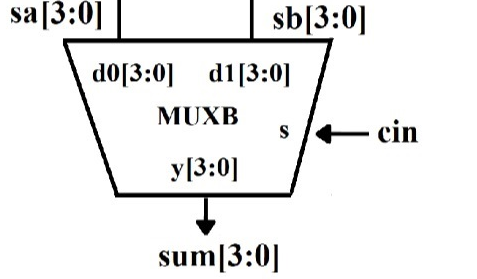
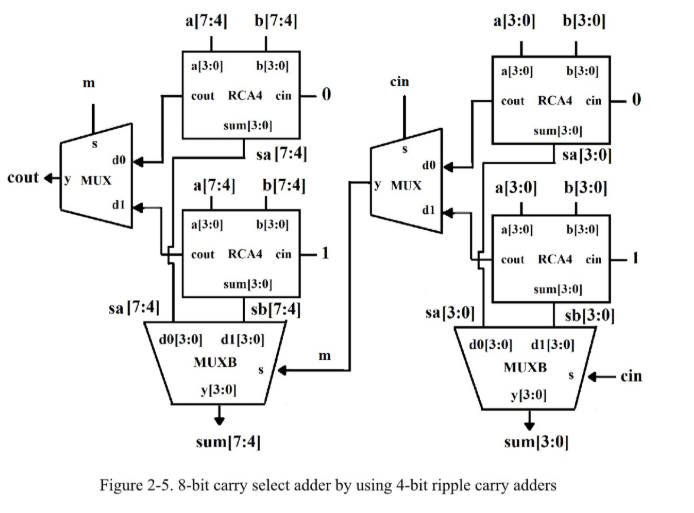
 

Figure 8: 8-Bit Carry Select Adder By Using 4-Bit Ripple Carry Adders

To wrap it all up, we will be combining all the previous designs into one (the CSA8). This design includes two of the 4-bit ripple carry adders and two of each multiplexer. Two RCA4 will assume the carry in is 1 and the other two will assume carry in is 0. The multiplexers’ role in this is to choose the right sum and carry out based on the carry in value. This method of addition is far faster than an 8-bit ripple carry adder.



## Source Code:

Half Adder Verilog and Test Bench Code:

`timescale 1ns / 1ps

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

`timescale 1ns / 1ps

module halfadder\_tb;

reg a, b;

wire cout, sum;

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

Full Adder Verilog and Test Bench Code:

`timescale 1ns / 1ps

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

module fulladder\_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'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

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

endmodule

4-Bit Ripple Carry Adder Verilog and Test Bench Code:

`timescale 1ns / 1ns

module rca (a, b, cin, cout, sum);

input [3:0] a,b;

input cin;

output [3:0] sum;

output cout;

wire [2:0] m;

fa h1(.cout(m[0]), .sum(sum[0]), .a(a[0]), .b(b[0]), .cin(cin));

fa h2(.cout(m[1]), .sum(sum[1]), .a(a[1]), .b(b[1]), .cin(m[0]));

fa h3(.cout(m[2]), .sum(sum[2]), .a(a[2]), .b(b[2]), .cin(m[1]));

fa h4(.cout(cout), .sum(sum[3]), .a(a[3]), .b(b[3]), .cin(m[2]));

endmodule

`timescale 1ns / 1ns

module rca\_tb;

reg [3:0] a, b;

reg cin;

wire [3:0] sum;

wire cout;

wire [3:0] res;

assign res = { cout, sum};

rca 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

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

endmodule

MUX Verilog and Test Bench Code:

`timescale 1ns / 1ns

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

`timescale 1ns / 1ns

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

MUXB Verilog and Test Bench Code:

`timescale 1ns / 1ns

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

input [3:0] d1, d0;

input 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

`timescale 1ns / 1ns

module muxb\_tb;

reg [3:0] d1,d0;

reg s;

wire [3:0] y;

muxb k1(d1, d0, s, y);

initial begin

d1 = 4'b0000; d0 = 4'b0000; s= 0;

#10 d1 = 4'b0001; d0 = 4'b0000; s= 0;

#10 d1 = 4'b0001; d0 = 4'b0010; s= 0;

#10 d1 = 4'b1001; d0 = 4'b0000; s= 1;

#10 d1 = 4'b1001; d0 = 4'b1000; s= 1;

#10 d1 = 4'b0001; d0 = 4'b1100; s= 1;

#10 $stop;

end

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

endmodule

8-Bit Carry Select Adder Verilog and Test Bench Code:

`timescale 1ns / 1ps

module CSA8(a , b, cin, cout, sa, sb, sum);

input [7:0] a, b;

input cin;

output [7:0] sa, sb, sum;

output cout;

wire c1, c2, c3, c4, m;

rca r1(.a(a[3:0]), .b(b[3:0]), .cin(0), .cout(c1), .sum(sa[3:0]));

rca r2(.a(a[3:0]), .b(b[3:0]), .cin(1), .cout(c2), .sum(sb[3:0]));

muxb mb1(.d1(sb[3:0]), .d0(sa[3:0]), .s(cin), .y(sum[3:0]));

mux2to1 m1(.d1(c2), .d0(c1), .s(cin), .y(m));

rca r3(.a(a[7:4]), .b(b[7:4]), .cin(0), .cout(c3), .sum(sa[7:4]));

rca r4(.a(a[7:4]), .b(b[7:4]), .cin(1), .cout(c4), .sum(sb[7:4]));

muxb mb2(.d1(sb[7:4]), .d0(sa[7:4]), .s(m), .y(sum[7:4]));

mux2to1 m2(.d1(c4), .d0(c3), .s(m), .y(cout));

endmodule

`timescale 1ns / 1ps

module CSA8\_tb;

reg cin;

reg [7:0] a,b;

wire cout;

wire [7:0] sum;

CSA8 u1(a, b, cin, cout, sa, sb, sum);

initial begin

cin = 1'b0; a = 8'b0000\_1000; b = 8'b1001\_1100;

#10;

cin = 1'b1; a = 8'b0000\_1001; b = 8'b1101\_1100;

#10;

cin = 1'b0; a = 8'b0000\_1001; b = 8'b1001\_1100;

#10;

cin = 1'b1; a = 8'b0100\_1000; b = 8'b1001\_1100;

#10;

cin = 1'b0; a = 8'b0010\_1000; b = 8'b1001\_1100;

#10;

cin = 1'b0; a = 8'b0100\_1000; b = 8'b1001\_1100;

#10;

cin = 1'b0; a = 8'b0100\_1000; b = 8'b1001\_1100;

#10 $stop;

end

endmodule

## Simulation Waveforms:

## 

## Figure 1: This figure above is the half adder simulation. A half adder adds the two values without CIN and this waveform shows that.

## 

Figure 2: This figure above is the full adder simulation. A full adder is a half adder with cin, this waveform shows this to be the case.

## 

Figure 3: This figure above is the 4-bit ripple carry adder simulation. RCA4 is full bit adders feeding into each other, allowing you to add 4 bit numbers. This waveform shows this to be the case

## 

Figure 4: This figure above is the MUX simulation. A multiplexer selects from the two, this waveform shows a multiplexer in action.

## 

Figure 5: This figure above is the MUXB simulation. MUXB is a 4 bit multiplexer, this shows that it selects properly.

## 

Figure 6: This figure above is the 8-bit carry select adder simulation. CSA8 is an 8 bit adder - this waveform shows that CSA8 adds properly.

## 

## Results Discussion:

The results of this part of the lab met the expectations, which was the CSA8 adds two 8 bit numbers properly and all the other Verilog files were correct. I had issues creating the CSA8, but by putting all files together but after some time I figured it out with the help of learning how to read the schematic diagram. The key findings of this part of the lab were how to use combinational logic via making a 4 bit RCA4, and putting it together with multiplexers to create an 8 bit carry adder and it went well.

# Part 2: 4 By 4 Binary Sequential Multiplier Design

## Design Purpose:

The purpose of this lab is to create a 4 bit binary shift multiplier with an FSM controlling it. This part of the lab consists of 7 modules - 2 four bit shift registers with one acting as a loader and one actually shifting, a multiplexer which emulates binary emulation, a 4 bit adder, a 8 bit shift register which stores the current product, a mult module which contains the elements that the FSM controls, the FSM itself and a TOP module which includes everything.

## Engineering Data:

## Overall Design:

A 4 by 4 sequential multiplier uses shift and add algorithm. At every clock cycle, the multiplier is shifted right by one bit and its value is tested. If the value is zero, a zero value is added to the accumulator, and the result is shifted right by one bit. If the value is one, the multiplicand value is added to the accumulator and the result is shifted right by one bit. The result is obtained when all the values have been tested.

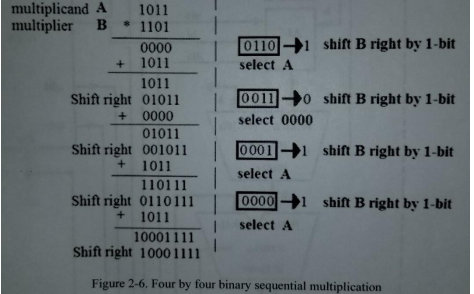
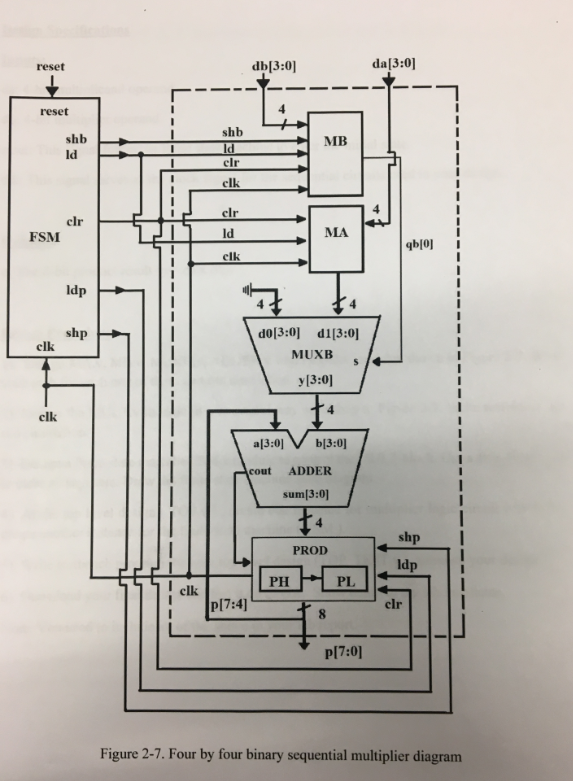
  


Figure 1: The figure below is the MA and MB. MA is a shift register that doesn’t shift (takes data in) and MB shifts

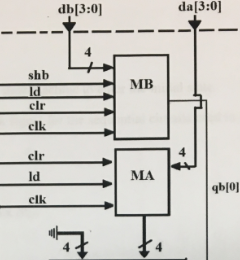
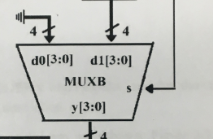


Figure 2: The figure below is the MUXB. MUXB is a 4 bit multiplexer.



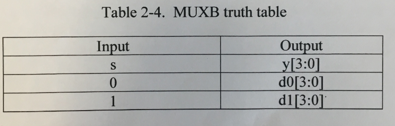


Figure 3: The figure below is the ADDER. The adder adds 2 4-bit numbers together.

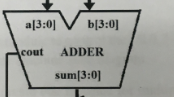


Figure 4: The figure below is the PROD. PROD is an 8 bit shift register split in half - PH is the first half and loads Cin and SUM, PL gets from PH.

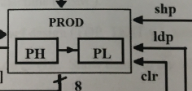


Figure 5: This figure below is the MULT.

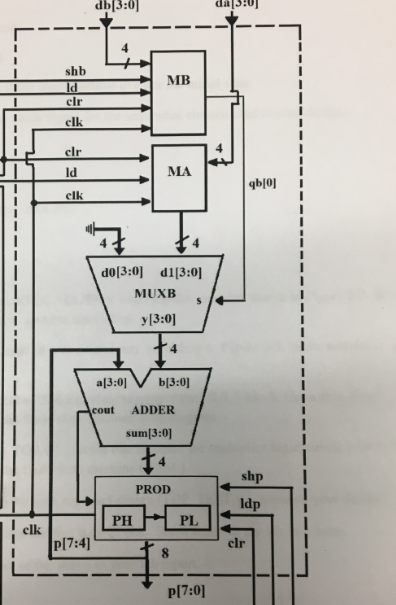
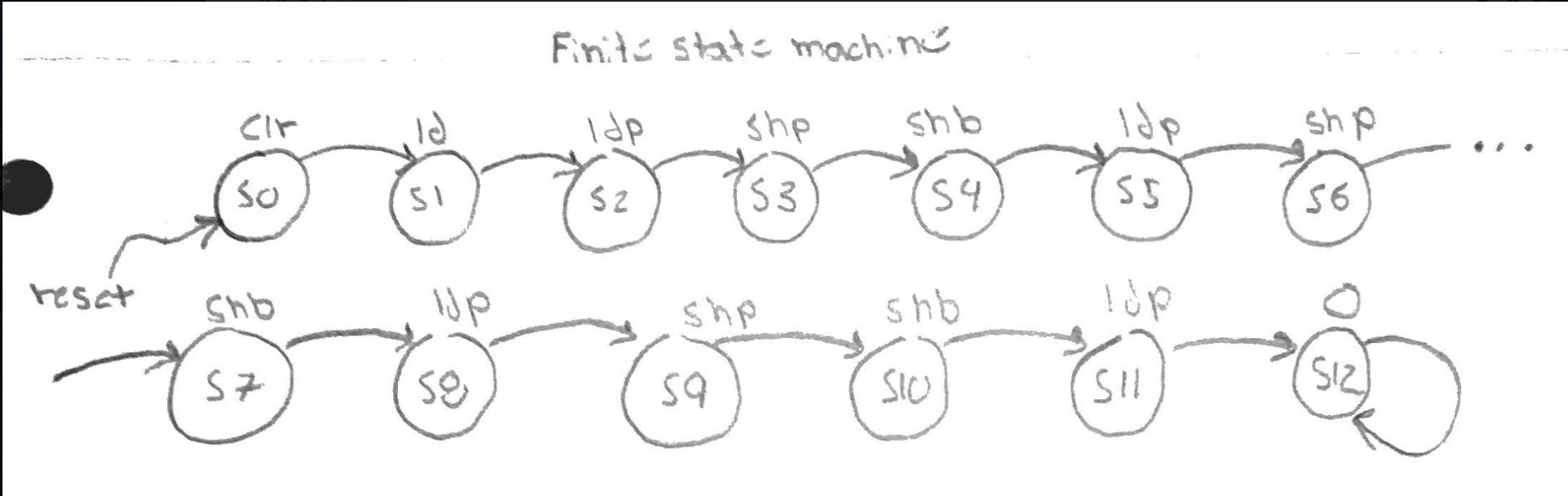


Figure 6: The figure is below is the FSM.



## Source Code:

## MA and MB Verilog and Test Bench Code:

## `timescale 1ns / 1ps

## module MA(clk, clr, ld, da, qa);

## input clk, clr, ld;

## 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 (ld)

## qa <= da;

## end

## endmodule

## `timescale 1ns / 1ps

## module MA\_tb;

## reg clk, clr, ld;

## reg [3:0] da;

## wire [3:0] qa;

## MA uut(.clk(clk), .clr(clr), .ld(ld), .da(da), .qa(qa));

## initial clk = 0;

## always #10 clk = ~clk;

## initial

## begin

## clr =1;

## da = 4'b1011;

## 

## #24 clr = 0; ld = 1;

## #24 ld = 0;

## 

## #60 $stop;

## end

## endmodule

## `timescale 1ns / 1ps

## module MB(clk, clr, ld, shb, db, qb);

## input clk, clr, ld, shb;

## 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 (ld)

## qb <= db;

## else if (shb)

## qb <= { 1'b0, qb[3:1] };

## // qb[3] <= 1'b0;

## // qb[2] <= qb[3];

## // qb[1] <= qb[2];

## // qb[0] <= qb[1];

## end

## endmodule

## `timescale 1ns / 1ps

## module MB\_tb;

## reg clk, clr, ld, shb;

## reg [3:0] db;

## wire [3:0] qb;

## MB uut (clk, clr, ld, shb, db, qb);

## initial clk = 0;

## always #10 clk = ~ clk;

## initial

## begin

## clr = 1; ld = 0; shb = 0; db = 4'b1011;

## #22 ld = 1;

## #20 shb = 1;

## #60 $stop;

## end

## endmodule

## ADDER Verilog and Test Bench Code:

## `timescale 1ns / 1ps

## module ADDER(a, b, sum, cout);

## input [3:0] a, b;

## 

## output [3:0] sum;

## output cout;

## 

## assign {cout,sum} = a + b;

## endmodule

## `timescale 1ns / 1ps

## module ADDER\_tb;

## reg [3:0] a,b;

## wire [3:0] sum;

## wire cout;

## ADDER g1(.a(a), .b(b), .sum(sum), .cout(cout));

## initial begin

## a = 4'b0000; b = 4'b0001;

## #10

## a = 4'b1000; b = 4'b0101;

## #10

## a = 4'b0011; b = 4'b0101;

## #10

## a = 4'b1010; b = 4'b0101;

## #10 $stop;

## end

## endmodule

## PROD Verilog and Test Bench Code:

## `timescale 1ns / 1ps

## module PROD(sum, shp, ldp, clr, clk, cout, p);

## input [3:0] sum;

## input shp, ldp, clr, clk, cout;

## output [7:0] p;

## reg [7:0] p;

## always@(posedge clr or posedge clk)

## begin

## if(clr) p <= 0;

## 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

## `timescale 1ns / 1ps

## module PROD\_tb;

## reg cin, shp, ldp, clr, clk;

## reg [3:0] sum;

## wire [7:0] p;

## PROD uu1PROD(sum, shp, ldp, clr, clk, p, cin);

## initial clk =0;

## always #10 clk = ~clk;

## initial

## begin

## sum = 4'b1011; cin = 1; shp = 0; ldp = 0; clr = 1;

## #10

## clr = 0;

## #10;

## shp = 1;

## #10

## ldp = 1;

## #10

## ldp = 0;

## #60 $stop;

## end

## endmodule

## MULT Verilog and Test Bench Code:

## `timescale 1ns / 1ns

## module MULT(da, db, p, ld, shb, clr, clk, shp, ldp);

## input [3:0] db , da;

## input ld, ldp, shb, shp, clk, clr;

## output [7:0] p;

## wire [3:0] null, a, qa, mux\_out, add\_out, qb;

## wire cout;

## assign null = 4'b0000;

## MB g1 (.clk(clk), .clr(clr), .ld(ld), .shb(shb), .db(db), .qb(qb));

## MA g2 (.clk(clk), .clr(clr), .ld(ld), .da(da), .qa(qa));

## MUXB g3 (.d1(qa), .d0(null), .s(qb[0]), .y(mux\_out));

## ADDER g4 (.a(p[6:3]), .b(mux\_out), .sum(add\_out), .cout(cout));

## PROD g5 (.sum(add\_out), .cout(cout), .shp(shp), .ldp(ldp), .clr(clr), .p(p), .clk(clk));

## endmodule

## `timescale 1ns / 1ns

## module MULT\_tb;

## reg [3:0] da, db;

## reg ld, shb, clr, clk, shp, ldp;

## wire [7:0] p;

## 

## MULT g1(da, db, p, ld, shb, clr, clk, shp, ldp);

## 

## initial clk = 0;

## always #10 clk = ~clk;

## 

## initial begin

## da = 4'b1011; db = 4'b1101; ld = 0; shb = 0; clr = 1; shp = 0; ldp = 0;

## #20 clr = 0;

## #20 ld = 1;

## #20 ld = 0;

## 

## #20 ldp = 1;

## #20 ldp = 0;

## 

## #20 shb = 1;

## #20 shb = 0;

## 

## #20 shp = 1;

## #20 shp = 0;

## 

## #20 ldp = 1;

## #20 ldp = 0;

## 

## #20 shb = 1;

## #20 shb = 0;

## 

## #20 shp = 1;

## #20 shp = 0;

## 

## #20 ldp = 1;

## #20 ldp = 0;

## 

## #20 shb = 1;

## #20 shb = 0;

## 

## #20 shp = 1;

## #20 shp = 0;

## 

## #20 ldp = 1;

## #20 ldp = 0;

## 

## 

## #20 $stop;

## end

## endmodule

## FSM Verilog Code:

## `timescale 1ns / 1ns

## module FSM(reset, clk, shb, ld, clr, ldp, shp );

## input clk, reset;

## 

## output shb, ld, clr, ldp, shp;

## reg shb, ld, clr, ldp, shp;

## 

## reg [3:0] cs, ns;

## parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4, s5 = 5,

## s6 = 6, s7 = 7, s8 = 8, s9 = 9, s10 = 10, s11 = 11, s12 = 12,

## s13 = 13;

## 

## 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;

## //s13: ns <= s13;

## 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

## TOP Verilog Code and Test Bench Code:

## `timescale 1ns / 1ps

## module TOP(reset,clk,db,da,p);

## input reset,clk;

## input [3:0] db,da;

## output[7:0] p;

## wire shb,ld,clr,ldp,shp;

## MULT g1(.da(da), .db(db), .p(p), .ld(ld), .shb(shb), .clr(clr), .clk(clk), .shp(shp), .ldp(ldp));

## FSM g2(.reset(reset), .clk(clk), .shb(shb), .ld(ld),.clr(clr), .ldp(ldp), .shp(shp));

## endmodule

## `timescale 1ns / 1ps

## module TOP\_tb;

## reg [3:0] da,db;

## reg reset, clk;

## wire [7:0] p;

## TOP g1(reset,clk,db,da,p);

## initial clk =0;

## always #10 clk = ~clk;

## initial begin

## //da = 4'b1011; db = 4'b1101; reset = 1;

## //da = 4'b1010; db = 4'b1010; reset = 1;

## da = 4'b1011; db = 4'b1101; reset = 1;

## #20 reset = 0;

## #400 $stop;

## end

## endmodule

## User Constraints File:

## set\_property -dict {PACKAGE\_PIN E3 IOSTANDARD LVCMOS33} [get\_ports clk];

## create\_clock -period 10.000 -name sys\_clk\_pin -waveform {0 5} -add [get\_ports clk];

## set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports { da[0] }]; #IO\_L24N\_T3\_RS0\_15 Sch=sw[0]

## set\_property -dict { PACKAGE\_PIN L16 IOSTANDARD LVCMOS33 } [get\_ports { da[1] }]; #IO\_L3N\_T0\_DQS\_EMCCLK\_14 Sch=sw[1]

## set\_property -dict { PACKAGE\_PIN M13 IOSTANDARD LVCMOS33 } [get\_ports { da[2] }]; #IO\_L6N\_T0\_D08\_VREF\_14 Sch=sw[2]

## set\_property -dict { PACKAGE\_PIN R15 IOSTANDARD LVCMOS33 } [get\_ports { da[3] }]; #IO\_L13N\_T2\_MRCC\_14 Sch=sw[3]

## set\_property -dict { PACKAGE\_PIN R17 IOSTANDARD LVCMOS33 } [get\_ports { db[0] }]; #IO\_L12N\_T1\_MRCC\_14 Sch=sw[4]

## set\_property -dict { PACKAGE\_PIN T18 IOSTANDARD LVCMOS33 } [get\_ports { db[1] }]; #IO\_L7N\_T1\_D10\_14 Sch=sw[5]

## set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports { db[2] }]; #IO\_L17N\_T2\_A13\_D29\_14 Sch=sw[6]

## set\_property -dict { PACKAGE\_PIN R13 IOSTANDARD LVCMOS33 } [get\_ports { db[3] }]; #IO\_L5N\_T0\_D07\_14 Sch=sw[7]

## #set\_property -dict { PACKAGE\_PIN T8 IOSTANDARD LVCMOS18 } [get\_ports { SW[8] }]; #IO\_L24N\_T3\_34 Sch=sw[8]

## #set\_property -dict { PACKAGE\_PIN U8 IOSTANDARD LVCMOS18 } [get\_ports { SW[9] }]; #IO\_25\_34 Sch=sw[9]

## #set\_property -dict { PACKAGE\_PIN R16 IOSTANDARD LVCMOS33 } [get\_ports { SW[10] }]; #IO\_L15P\_T2\_DQS\_RDWR\_B\_14 Sch=sw[10]

## #set\_property -dict { PACKAGE\_PIN T13 IOSTANDARD LVCMOS33 } [get\_ports { SW[11] }]; #IO\_L23P\_T3\_A03\_D19\_14 Sch=sw[11]

## #set\_property -dict { PACKAGE\_PIN H6 IOSTANDARD LVCMOS33 } [get\_ports { SW[12] }]; #IO\_L24P\_T3\_35 Sch=sw[12]

## #set\_property -dict { PACKAGE\_PIN U12 IOSTANDARD LVCMOS33 } [get\_ports { SW[13] }]; #IO\_L20P\_T3\_A08\_D24\_14 Sch=sw[13]

## #set\_property -dict { PACKAGE\_PIN U11 IOSTANDARD LVCMOS33 } [get\_ports { SW[14] }]; #IO\_L19N\_T3\_A09\_D25\_VREF\_14 Sch=sw[14]

## set\_property -dict { PACKAGE\_PIN V10 IOSTANDARD LVCMOS33 } [get\_ports { reset }]; #IO\_L21P\_T3\_DQS\_14 Sch=sw[15]

## set\_property -dict { PACKAGE\_PIN H17 IOSTANDARD LVCMOS33 } [get\_ports { p[0] }]; #IO\_L18P\_T2\_A24\_15 Sch=led[0]

## set\_property -dict { PACKAGE\_PIN K15 IOSTANDARD LVCMOS33 } [get\_ports { p[1] }]; #IO\_L24P\_T3\_RS1\_15 Sch=led[1]

## set\_property -dict { PACKAGE\_PIN J13 IOSTANDARD LVCMOS33 } [get\_ports { p[2] }]; #IO\_L17N\_T2\_A25\_15 Sch=led[2]

## set\_property -dict { PACKAGE\_PIN N14 IOSTANDARD LVCMOS33 } [get\_ports { p[3] }]; #IO\_L8P\_T1\_D11\_14 Sch=led[3]

## set\_property -dict { PACKAGE\_PIN R18 IOSTANDARD LVCMOS33 } [get\_ports { p[4] }]; #IO\_L7P\_T1\_D09\_14 Sch=led[4]

## set\_property -dict { PACKAGE\_PIN V17 IOSTANDARD LVCMOS33 } [get\_ports { p[5] }]; #IO\_L18N\_T2\_A11\_D27\_14 Sch=led[5]

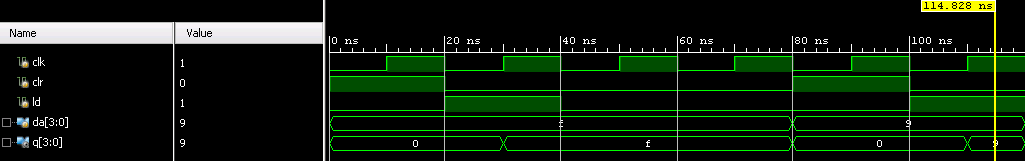
## set\_property -dict { PACKAGE\_PIN U17 IOSTANDARD LVCMOS33 } [get\_ports { p[6] }]; #IO\_L17P\_T2\_A14\_D30\_14 Sch=led[6]

## set\_property -dict { PACKAGE\_PIN U16 IOSTANDARD LVCMOS33 } [get\_ports { p[7] }]; #IO\_L18P\_T2\_A12\_D28\_14 Sch=led[7]

## Simulation Waveforms:

Figure 1: This figure below is the simulation of MA and MB. MA loads data in, this shows this to be true. MB is a shift register which outputs the carry, this waveform shows that MB is working properly.

MA:



MB:

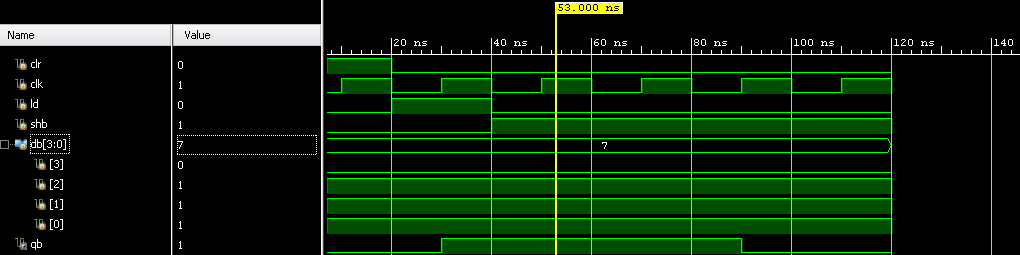


Figure 2: This figure below is the simulation of the MUXB. MUXB is a four bit multiplier, this waveform shows it is selecting properly.

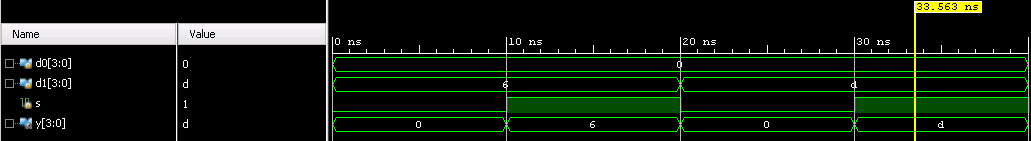


Figure 3: This figure below is the simulation of the ADDER. ADDER adds the two values together, this shows it is adding properly.

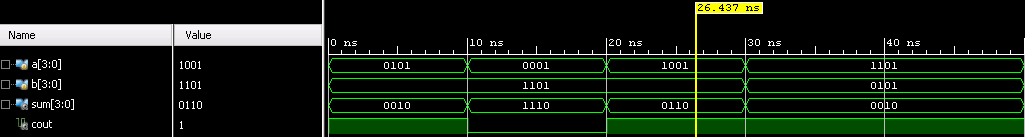


Figure 4: This figure below is the simulation of the PROD. The PROD is an 8 bit shift register, this waveform shows it loads values properly and shifts them over properly.

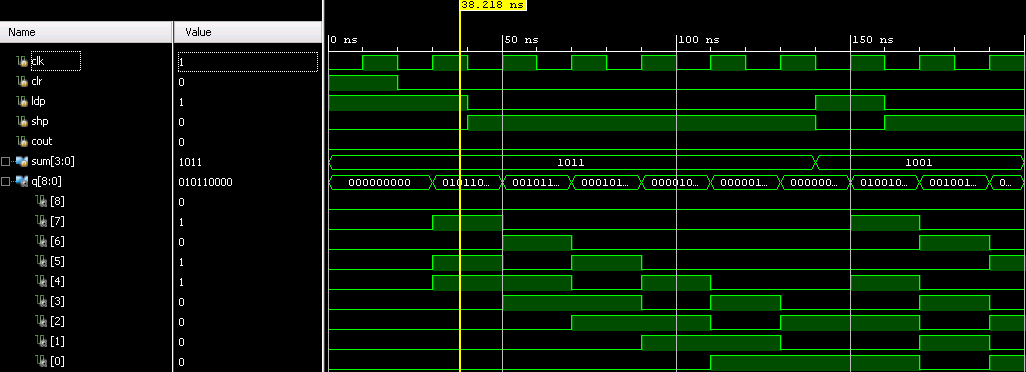


Figure 5: This figure below is the simulation of the MULT. MULT is a module that contains everything FSM controls. This test bench shows that each component works properly together in the way that it should. It loads, it does the operations, it shifts properly.

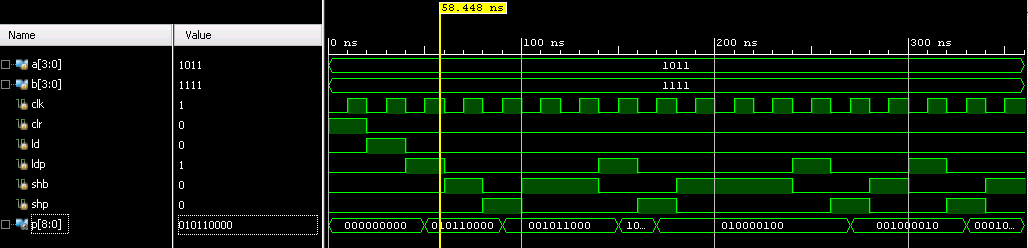
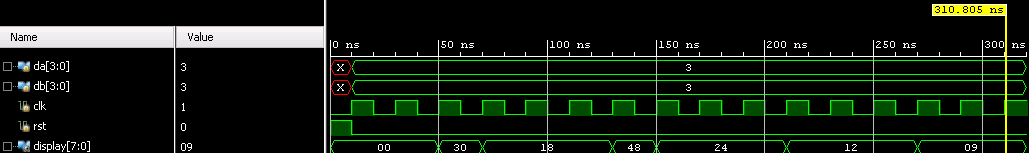


Figure 6: This figure below is the simulation for the TOP. TOP is the result of the multiplier, the values in display should be multiplied values of da and db at the end, this waveform shows that TOP, and everything in TOP is working properly.



## Results Discussion:

## The results of this part of the lab met the expectations - the 4 binary segmental multiplier multiplies values properly. I had issues mainly with PROD and MULT - I was trying to make prod with two 4-bit shift registers but was having issues loading it in, I resolved them by making PROD an 8 bit shift register, and MULT was having issues getting things loading properly and I fixed it with troubleshooting skills. The key findings of this part of the lab were how to put these components together and use sequential logic with the FSM to control all of these components, I had a lot of issues during the process of this lab but figured out the elements in the end. In this part of the lab we figured out how to create shift registers and FSM and got complex circuits to work together, in the end though we had a multiplier on the FPGA that produces correct values. This part of the lab taught us a lot about how Verilog, Vivado and the NEXYS board works and while it was challenging it was very rewarding in the end.

## Figure 1: This figure below is the result of multiplying 1011 and 1101 together.

## 

## Part 3: Character Displays On 8-Digit Multiplexed Seven Segment Displays

Design Purpose:

For this final part of the lab, I was expected to display “CPE 166” and my initials “SL”. This required me to understand how to access each cathode and each segment in each cathode in order to turn the segments on in a pattern that is recognizable as a number or letter. In order to do this, I must first understand the common Anode which allows you to understand which letter to turn high in order to activate a certain part of 1 segment display. By turning on the correct segments of each display, I was able to turn the segments into a recognizable letter or number. I also had to understand which pins to connect in order to access each cathode.

Engineering Data:

Figure 1: This figure is a common anode seven segment display device.

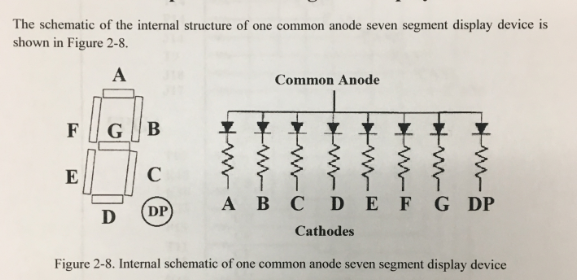
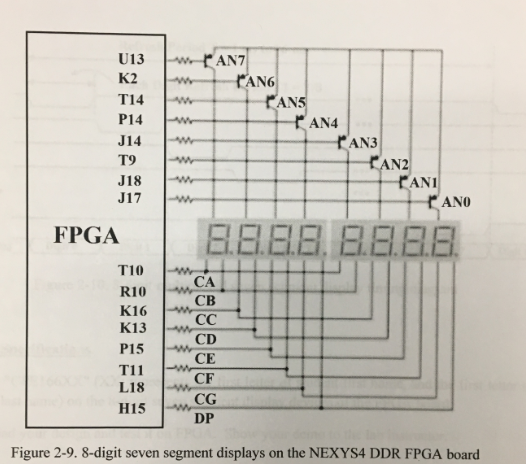


Figure 2: This figure below is the 8-digit seven segment displays on the NEXXYS4 DDR FPGA board.



Source Code:

`timescale 1ns / 1ps

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; //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'b0010010; //to display S

4'd7 : seg[6:0] = 7'b1000111; //to display L

default : seg[6:0] = 7'b1111111; //blank

endcase

end

endmodule

User Constraints 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]

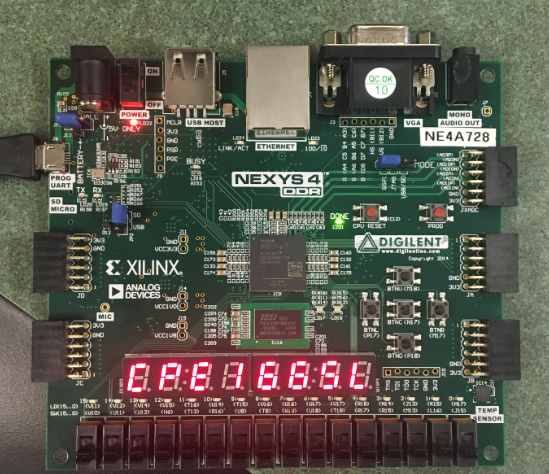
set\_property -dict { PACKAGE\_PIN K2 IOSTANDARD LVCMOS33 } [get\_ports { dig[6] }]; #IO\_L23P\_T3\_35 Sch=an[6]

set\_property -dict { PACKAGE\_PIN U13 IOSTANDARD LVCMOS33 } [get\_ports { dig[7] }]; #IO\_L23N\_T3\_A02\_D18\_14 Sch=an[7]

Results Discussion:

The results of this part of the lab met the expectations - we managed to manipulate the 7-segment display. This lab was pretty much smooth sailing with no major issues. The key findings of this part of the lab were how to use write constraint files as well as how the anodes and diodes work with the seven-segment display to show values, and things went smoothly. This part of the lab taught us a lot about how the seven-segment display works as well as constraint works, and it went smoothly.

Figure 1: This figure below is CPE166Sl on the board.



Conclusion:

Overall, this lab helped me learn more about Verilog, taught me how to use the NEXYS4 board/VIVADO and how combinational and sequential logic works. Part 1 introduced us to hierarchical design by breaking the design into several components. We then used these parts to learn how to create a carry select adder. Part 2 then introduced us to sequential design with the FSM and we had to create a sequential shift/add multiplication algorithm/circuit. Part 3 taught us how to use a multiplexed seven segment display. Throughout the process of both part 1 and 2 we learned how to write test benches and create waveform simulations, while part 2 and 3 showed us how to use constraint files to download a program into the NEXYS4 FPGA board. After this lab I have a better understanding of hierarchical, combinational and sequential design, as well as a better understanding of Verilog, VIVADO, and the NEXYS4 Board. Overall the lab was challenging but went well and I learned a lot from the lab.

## 