CPE 166 Advance Logic Design

Lab Section 2

Shammah Thao

10/8/20

Contents

Introduction

This lab was a four-part lab served as an introduction to hierarchical design in Verilog. We were tasked to design 3 by 3 binary array multiplier, 8-bit carry select adder, Two-speed BCD counter, and an Automatic Beverage Vending Machine. We had to use hierarchical design for each part of these lab. We had to build up with smaller Verilog coding then at the end we had to combine them and make them interact with each other to get the result. We had to learn hierarchical design strategies using VHDL along with creating adequate testbenches to test our source codes. 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.

Part 1: 3 by 3 Combinational Array Multiplier

Design Purpose:

By the end of this part of the lab, we are supposed to be able to get a 3 by 3 multiplier. To do so we would need to first create need two smaller Verilog coding design. One is the Half Adder Design and the other is Full Adder Design. The Half adder and Fuller Adder are the main components into making the final Verilog coding works since the three by three multiplier is just a combination of the Half Adder and Full Adder together.

Verilog Design:

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			

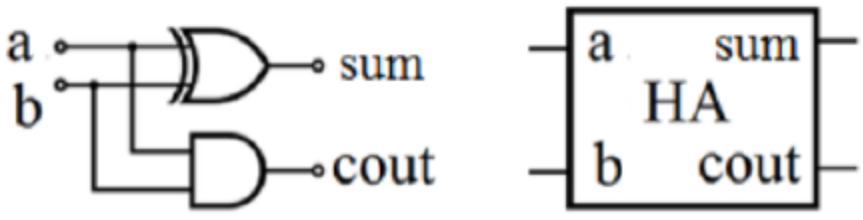


Figure 1. Half Adder truth table, schematic and logic symbol

In figure 1, It shows the truth table of the half adder, which shows it adding the two 1-bit input and generated a two 1-bit output which includes a 1-bit carry out.

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:

$sum = a \oplus b \oplus cin$

$cout = (a \oplus b) cin + a b$

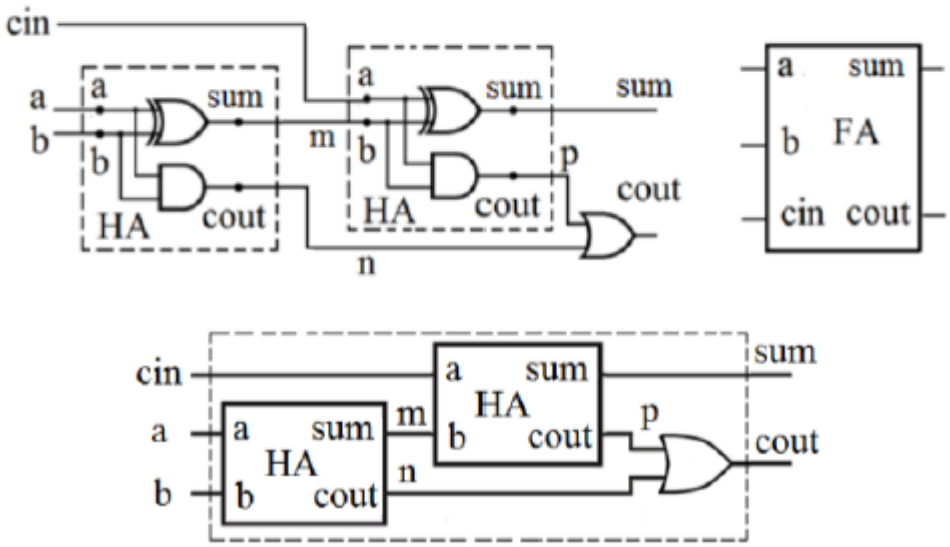


Figure 2. Full Adder truth table, schematic and logic symbol

Figure 2 features the truth table of the full adder, which shows it performing an addition operation with a three-bit number. It produces a sum of the three inputs and carry values.

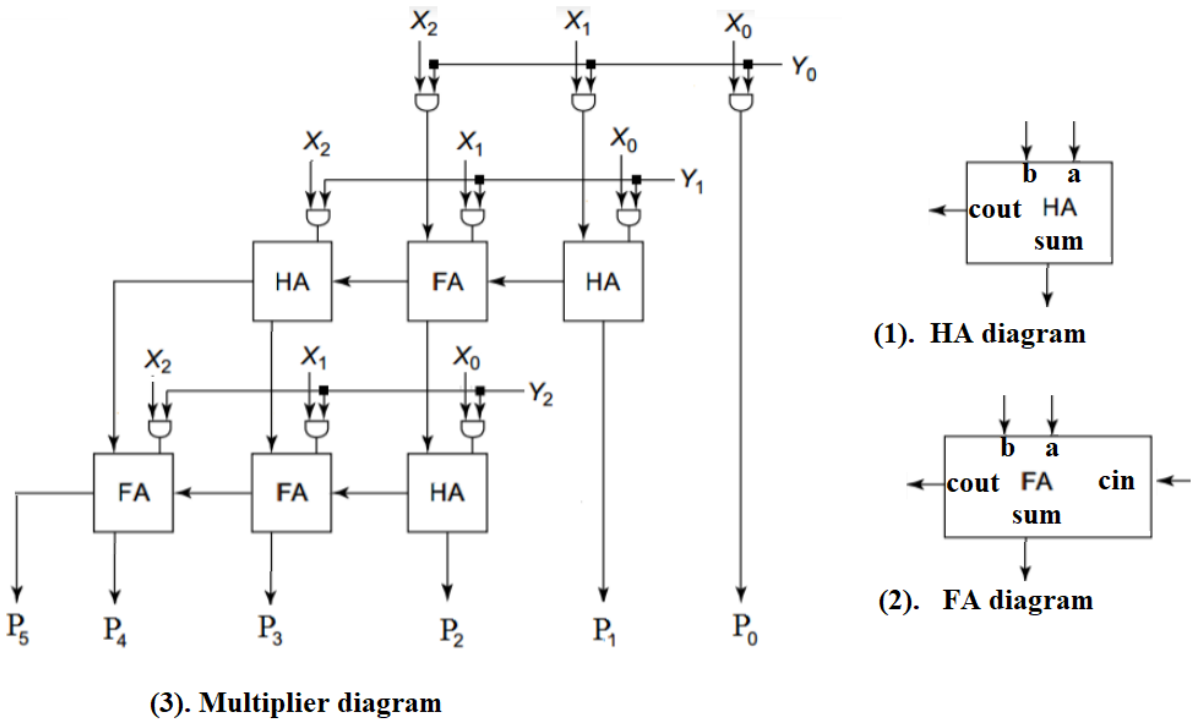


Figure 3. three by three combination array multiplier schematic

Figure 3 shows, the schematic of the multiplier. It has 6 inputs with 5 outputs, using a combination of the half adder and full adder. It goes though right to left and get a solution that is pushed through the next one until they get to the last one.

Verilog Coding:

Step 1: Half adder

Source Code	Testbench

module Hadder(a,b,cout,sum);
input a;
input b;
output cout;
output sum;
 	
assign cout = a&b;
assign sum = a^b;
endmodule
	module Hadder_tb;
	reg a,b;
	wire cout,sum;
	Hadder g1(.a(a),.b(b),.cout(cout),.sum(sum));
	
	initial
	begin
	a=0;
	b=0;
	
	#10;
	a=1;
	b=0;

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

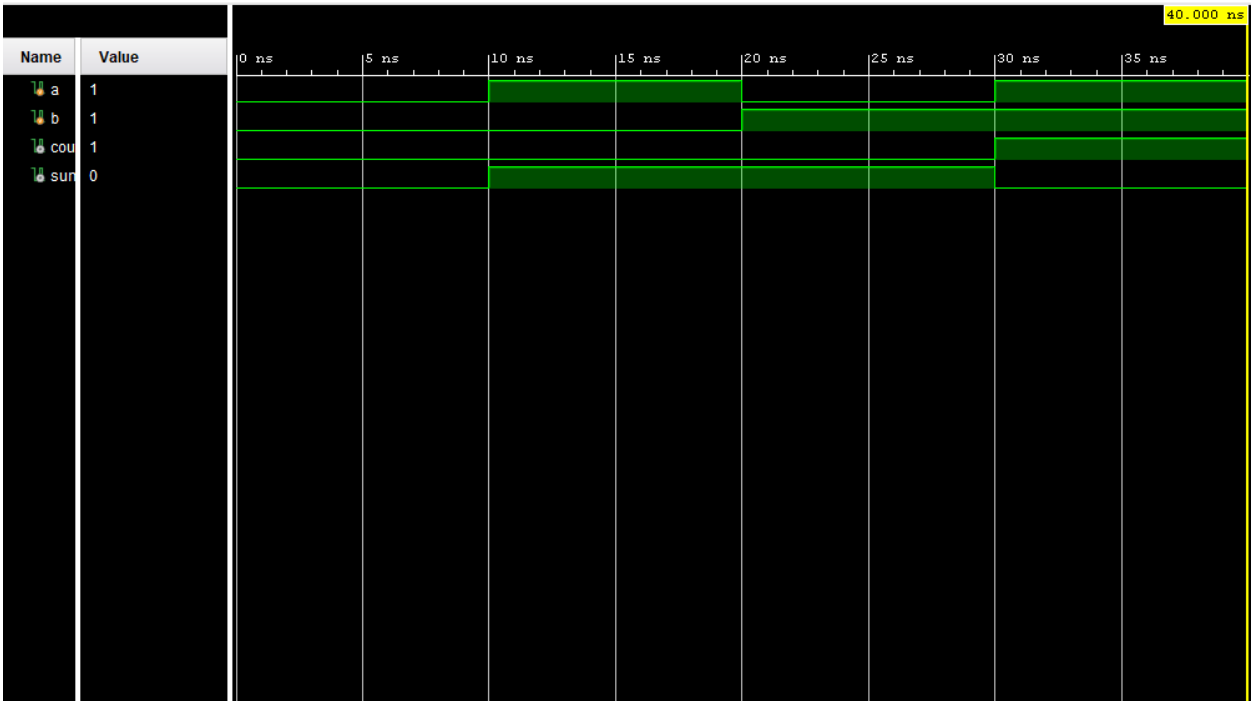


Figure 4. Half Adder

Step2: Full Adder

Source Code	Testbench
module FullerAdder(a,b,cin,cout,sum); input a,b,cin; output cout,sum; wire m,n,p; Hadder g1(.a(a),.b(b),.cout(n),.sum(m)); Hadder g2(.a(cin),.b(m),.cout(p),.sum(sum));	module FullerAdder_tb; reg a,b,cin; wire cout,sum; FullerAdder u1(.a(a),.b(b),.cin(cin),.cout(cout),.sum(sum)); initial begin

```
assign cout = p|n;

endmodule
```

```
a= 0;
b=0;
cin=0;
#10;
a=0;
b=0;
cin=1;
#10;
a=0;
b=1;
cin=0;
#10;
a=0;
b=1;
cin=1;
#10;
a=1;
b=0;
cin=0;
#10;
a=1;
b=0;
cin=1;
#10;
a=1;
b=1;
cin=0;
#10;
a=1;
b=1;
cin=1;

#10 $stop;

end

endmodule
```

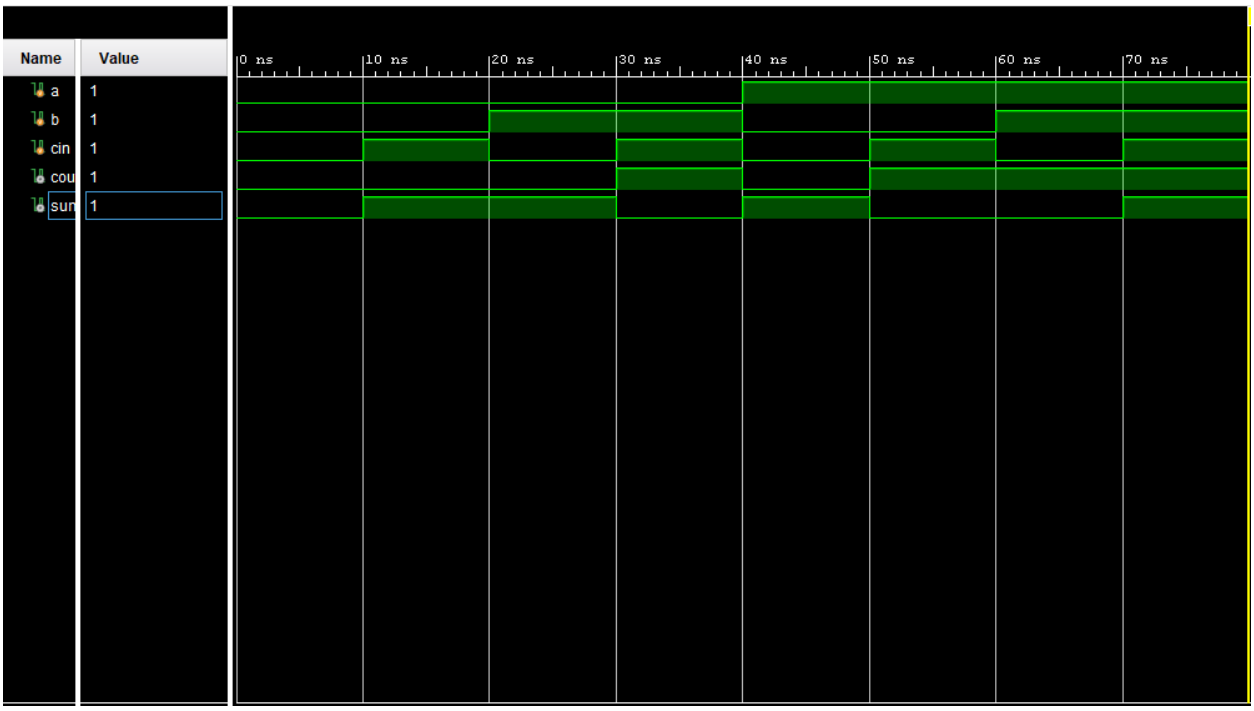


Figure 5. Full Adder

Part3: Final Combination multiplies design

Source Code	Testbench

module part1(x,y,p);
input [2:0]x,y;
output [5:0]p;
 	
wire[5:0]cin;
wire[1:0]sum;
wire[5:0]p;
wire[8:0]q;
 	
assign q[0]=x[0]&y[0];
assign q[1]=x[1]&y[0];
assign q[2]=x[2]&y[0];
assign q[3]=x[0]&y[1];
assign q[4]=x[1]&y[1];
assign q[5]=x[2]&y[1];
assign q[6]=x[0]&y[2];
assign q[7]=x[1]&y[2];
assign q[8]=x[2]&y[2];
 	
assign p[0]=q[0];
Hadder ha1(.a(q[3]),.b(q[1]), .cout(cin[0]), .sum(p[1]));
	module part1_tb;
	reg [2:0]x,y;
	wire [5:0]p;
	
	integer i,k;
	
	part1 u1(.x(x),.y(y),.p(p));
	
	initial begin
	x=3'b000;
	y=3'b000;
	for(i=0;i<8;i=i+1)
	begin
	x=i;
	#10;
	for(k=0;k<8;k=k+1)
	begin
	y=k;
	#10;
	end
	end

FullerAdder
fa1(.a(q[4]),.b(q[2]),.cin(cin[0]),.cout(cin[1]),.sum(sum[0]));
Hadder ha2(.a(q[5]),.b(cin[1]), .cout(cin[2]), .sum(sum[1]));
Hadder ha3(.a(q[6]),.b(sum[0]), .cout(cin[3]), .sum(p[2]));
FullerAdder
fa2(.a(sum[1]),.b(q[7]),.cin(cin[3]),.cout(cin[4]),.sum(p[3]));
FullerAdder
fa3(.a(cin[2]),.b(q[8]),.cin(cin[4]),.cout(p[5]),.sum(p[4]));
 	
endmodule
	#10 $stop;
	end
	endmodule

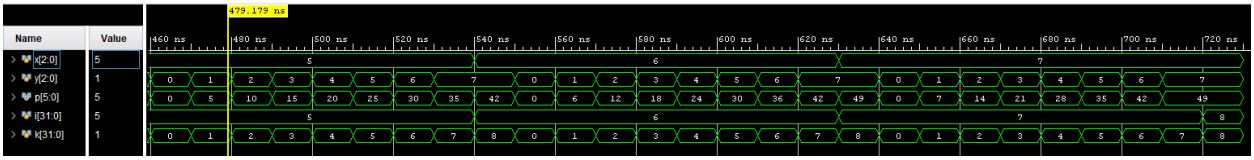


Figure 6. 3 by 3 multiplier

Results

The result that was achieve what exactly that was wanted. The half was quite straight forward as it was just using the equation given to us and put it into Verilog. The issues were kind the full adder as going into it I did not know that it was supposed to be a hierarchical Verilog coding, using the half adder. I assumed it was just its own part using the equation that was given. Later, I was able to catch on was able to work out the logic and Verilog coding for the 3 b 3 just fine.

Part 2: 8-bit Carry Select Adder

Design Purpose:

The purpose of this part of the lab was to create a 8-bit carry select adder circuit. It consists of three 4-bit ripple carry adder and two multiplexers. One of 4-bit ripple carry adders assumes the carry-in to be zero, and the other assumes the carry-in to be one. The multiplexers select the correct sum and the carry-out based on the known carry-in value.

Verilog Design:

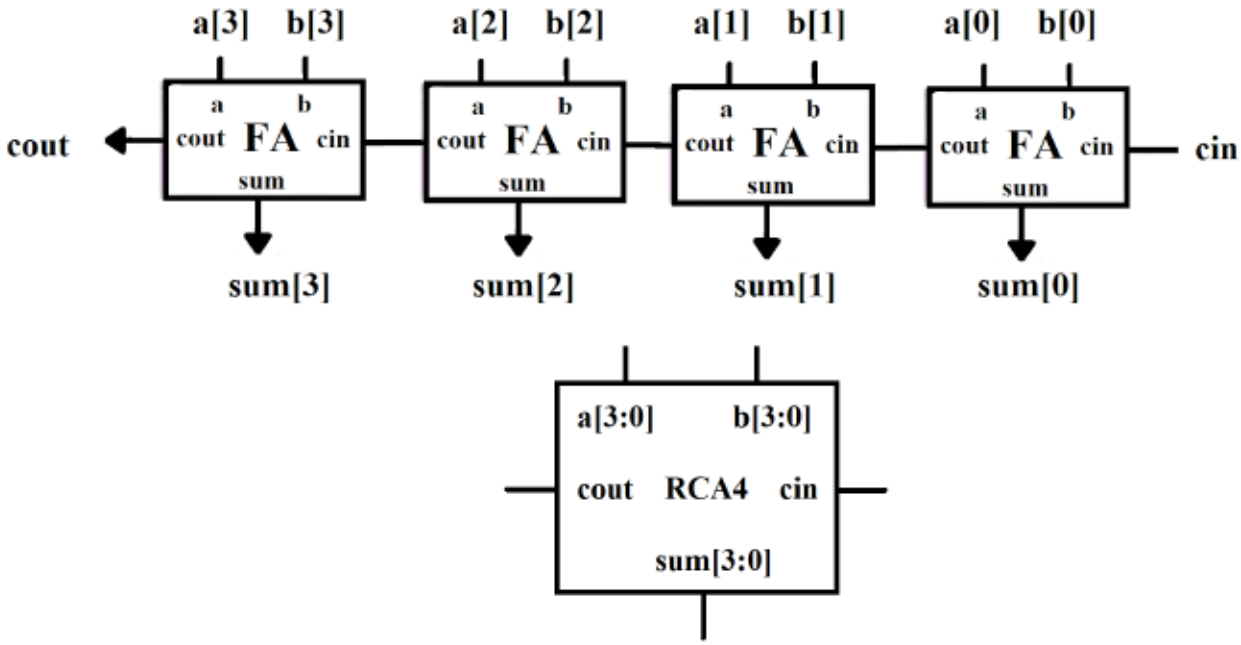


Figure 7. 4-bit ripple carry adder circuit and block diagram

Figure 7 was the 4-bit ripple carry adder that we were suppose to create using coding from lab 2 part 1. We remade or copied code from our Full adder and half add and combine its to remake a 4-bit ripple carry adder.

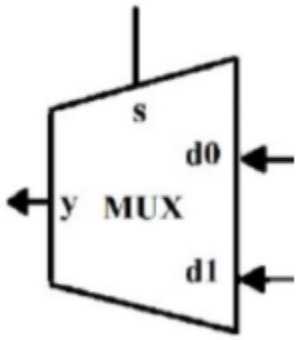


Figure 8. 2 to 1 multiplexer diagram

Figure 8 features another Verilog coding that we had to do, a 2 to 1 multiplexer. It was created using if statements to execute when certain conditions are met

s	y
0	d0
1	d1

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

Figure 9. MUXB design

Figure 9 was about MUXB that consists of two 4-bit input D0 and D1, one 1-bit selection input S and one 4-bit output Y. According to the logic value of the selection signal S, D0 or D1 will be passed to the output.

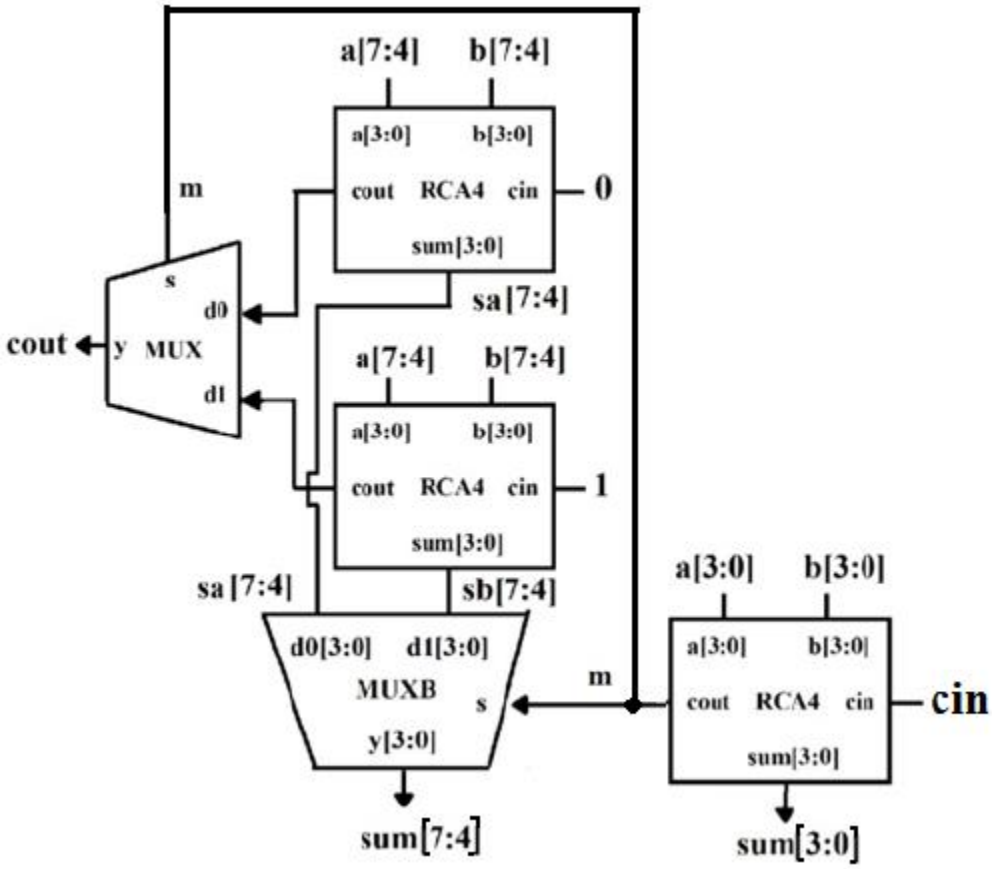


Figure 10. 8-bit carry select adder circuit

Figure 10 was a combination of all of part 2. It includes 8-bit carry-select adder (CSA8) circuit above by using three 4-bit ripple carry adders (RCA4), one MUX and one MUXB

Verilog Coding:

Half Adder

Source Code	Testbench

`timescale 1ns / 1ps
module HA(a, b, cout, sum);
input a, b;
output cout, sum;
assign cout = a & b;
assign sum = a ^ b;
	`timescale 1ns / 1ps
	module HA_tb;
	reg a, b;
	wire cout, sum;
	HA u1 (.a(a), .b(b), .cout(cout), .sum(sum));
	initial begin

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

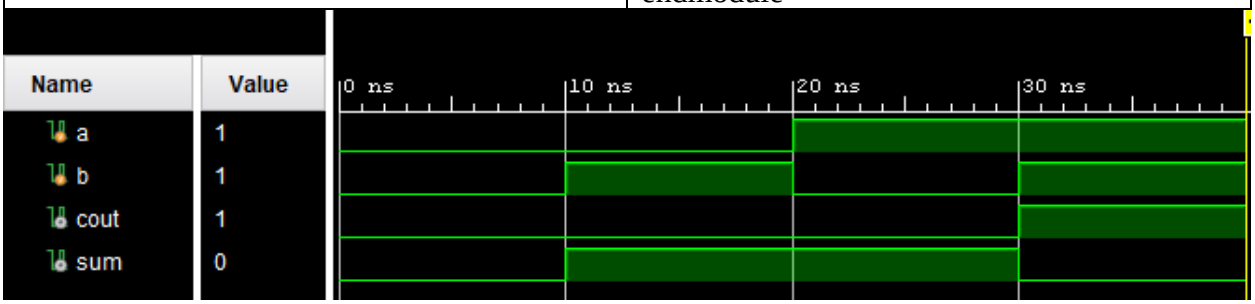


Figure 11. Half Adder

Full adder

Source Code	Testbench

`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 FA_tb;
	reg a, b, cin;
	wire cout, sum;
	FA u1 (a, b, cin, cout, sum);
	initial begin
	{a, b, cin} = 3'b000;
	#10 {a, b, cin} = 3'b001;
	#10 {a, b, cin} = 3'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

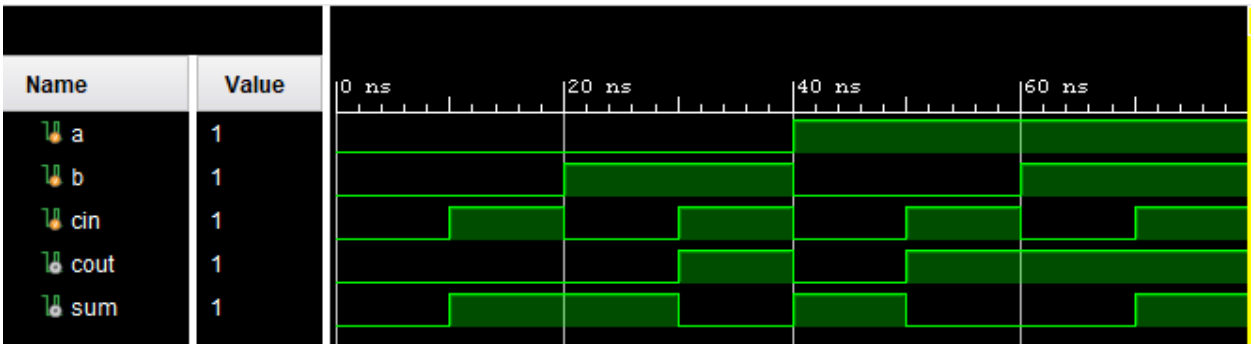


Figure 12. Full Adder

4-bit ripple carry adder design

Source Code	Testbench

`timescale 1ns / 1ns
module RCA4 (a, b, cin, cout, sum);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;
wire [2:0] m;
FA 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 RCA4_tb;
	reg [3:0] a, b;
	reg cin;
	wire [3:0] sum;
	wire cout;
	wire [3:0] res;
	assign res = { cout, sum};
	RCA4 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

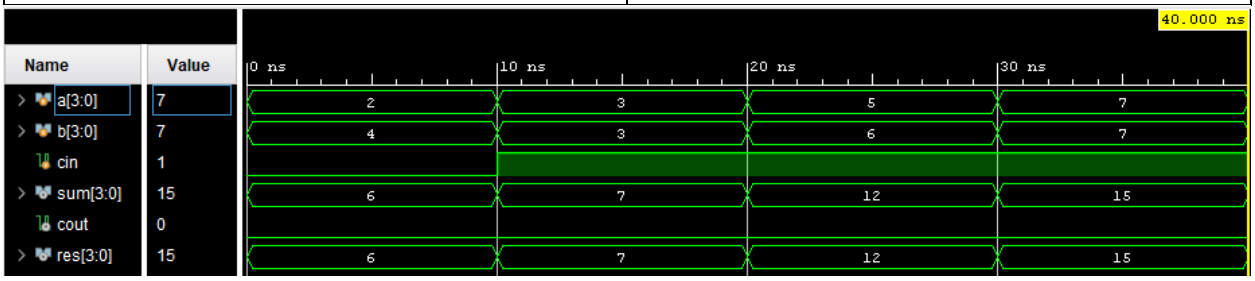


Figure 13. 4-bit ripple carry adder

2 to 1 Multiplexer

Source Code	Testbench
module mux2to1(d1, d0, s, y);	`timescale 1ns / 1ns

input d1, d0, s;
output y;
reg y;
always@(d1 or d0 or s)
begin
if (s) y = d1;
else y = d0;
end
endmodule
	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

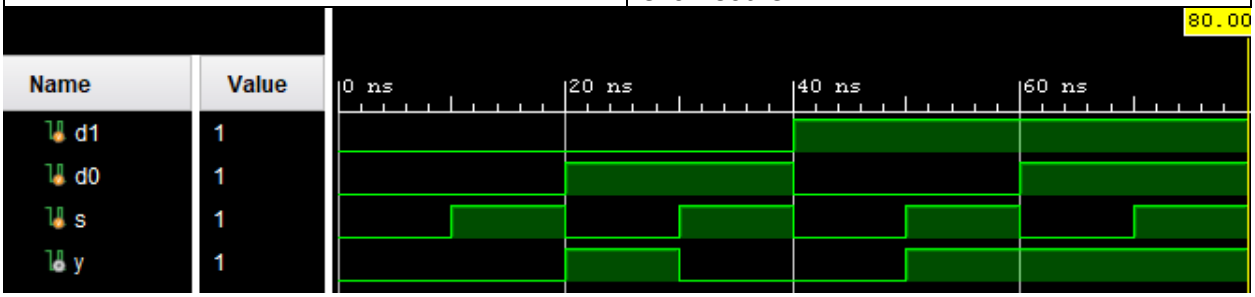


Figure 14. 2 to 1 Multiplexer

Muxb design

Source Code	Testbench

`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

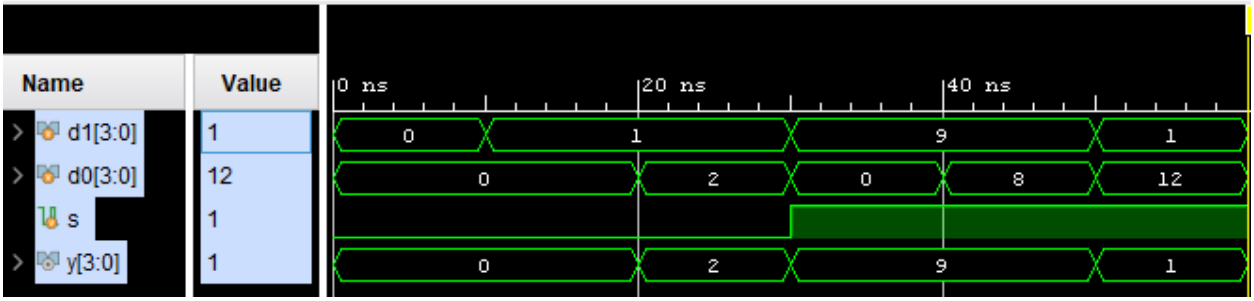


Figure 15. MUXB

8-bit carry select adder design

Source Code	Testbench

`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;
RCA4 r1(.a(a[3:0]), .b(b[3:0]), .cin(0),
.cout(c1), .sum(sa[3:0]));
RCA4 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));
RCA4 r3(.a(a[7:4]), .b(b[7:4]), .cin(0),
.cout(c3), .sum(sa[7:4]));
RCA4 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 = 0; a = 8; b = 42;
	#10;
	cin = 1; a = 12; b = 42;
	#10;
	cin = 0; a = 8; b = 42;
	#10;
	cin = 1; a = 9; b = 42;
	#10;
	cin = 0; a = 15; b = 42;
	#10;
	cin = 0; a = 16; b = 42;
	#10;
	cin = 0; a = 17; b = 42;
	#10 $stop;
	end
	endmodule

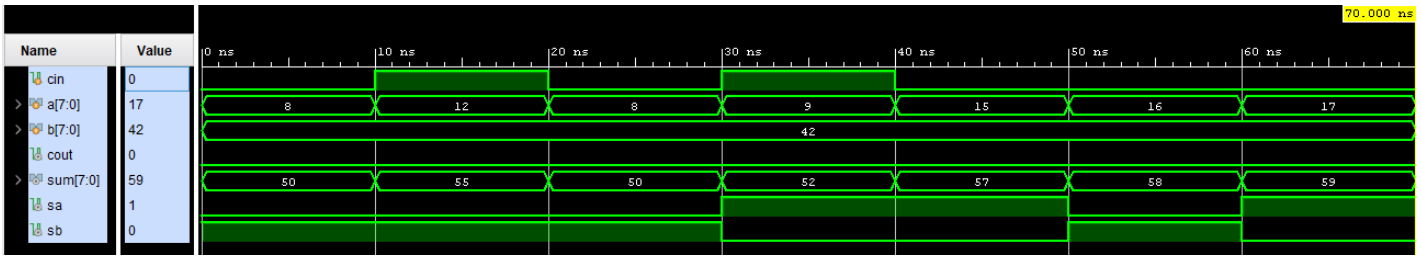


Figure 16. 8-bit Carry Select Adder

Result Discussion:

The result came out as expected. The only issues that I encounter with this was the logic that came along with it. It took some time looking at the diagram and going back and forth changing the coding. This part of the lab was like part 1 of the lab so there was not as much difficultly. One thing that I also found helpful during the demo was that apparently, we could just use number on the testbench instead of just binary. This will make future lab easier.

Part3: Two-Speed BCD Counter

Design Purpose:

We are to make a binary coded decimal counter that has a digital counter that counts to 0 to 9 and then repeats. Creating this BCD counter, we would be able to choose from two different speed at which it can run. This part of the lab would focus and help us learn more about CLK Verilog coding.

Verilog Design:

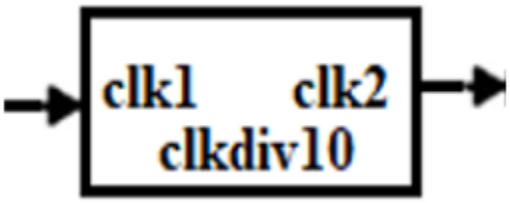


Figure 17. clkdiv10 module block diagram

Figure 17 is a clkdiv10 design that has one input and one output. It was design to be 10 times slower than the frequency of the input.

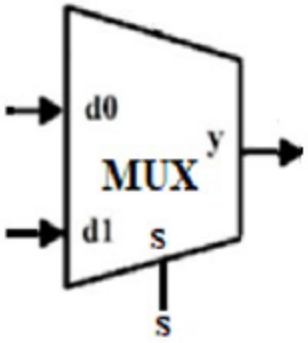


Figure 18. 2 to 1 multiplexer

Figure 18 is a 2 to 1 multiplexer like what was done in the previous part of the lab. It has two input, one selection input and one output.

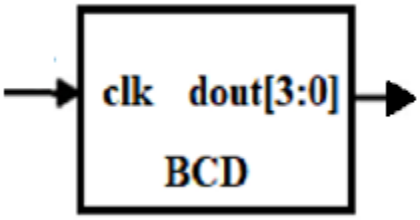
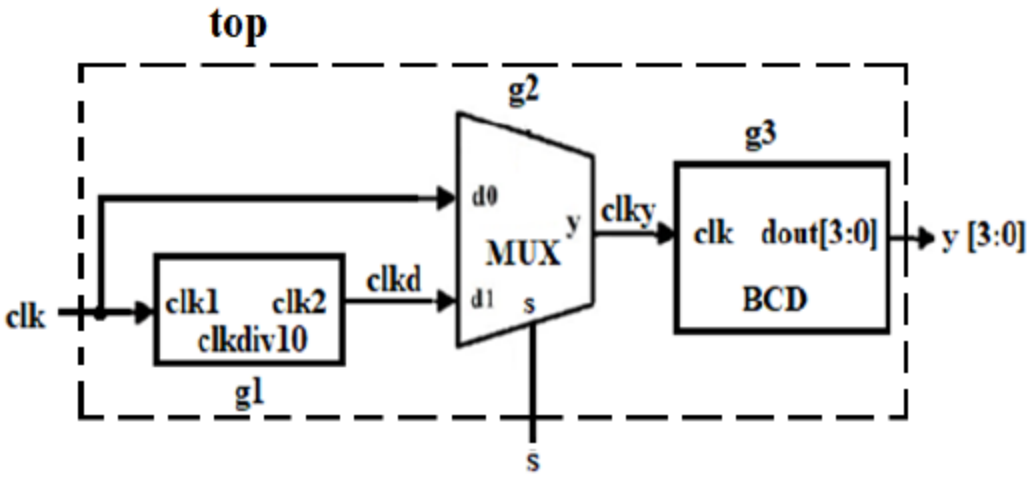


Figure 19. BCD counter Diagram

Figure 19 is a BCD counter circuit that has one input clk, and four-bit output signal dout. The dout signal counts from 0 to 9 and then repeats.



Design Name	Port Names	Port Direction	Port Size
top.v	clk	input	1 bit
	s	input	1 bit
	y	output	4 bits

Figure 20. Two-speed BCD counter diagram and interface information

Figure 20 is the final two-speed BCD counter circuit that requires one clkdiv10 instance, one MUX instances, and one BCD instance. When s is logic 0, the BCD counter will run at the input clock frequency. When s is logic 1, the BCD counter will run 10 times slower. The output y generates a value from 0 to 9, and then repeats.

Verilog Coding:

Clkdiv10:

Verilog	Testbench
`timescale 1ns / 1ps module clkdiv10(clkin,clkout); input clkin; output clkout; reg clkout; reg[2:0] cnt;	`timescale 1ns / 1ps module clkdiv10_tb; reg clkin; wire clkout; integer k; clkdiv10 uut (clkin, clkout);

```
initial cnt = 0;
initial clkout = 0;

always@(posedge clkin)
begin
   if(cnt==9)
   begin
      cnt<=0;
      clkout<=1;
   end
   else if(cnt<4)
   begin
      cnt<=cnt+1;
      clkout<=1;
   end
   else
   begin
      cnt<=cnt+1;
      clkout<=0;
   end
end
endmodule
```

```
initial clkin = 0;
always #2 clkin = ~ clkin;
initial begin
 k=0;
 while( k != 50 )
 begin
 @(clkin);
 $display($time, " ns, k=%d, clkout=%b", k,
clkout);
 k = k + 1;
 end
 #5 $stop;
end
endmodule
```

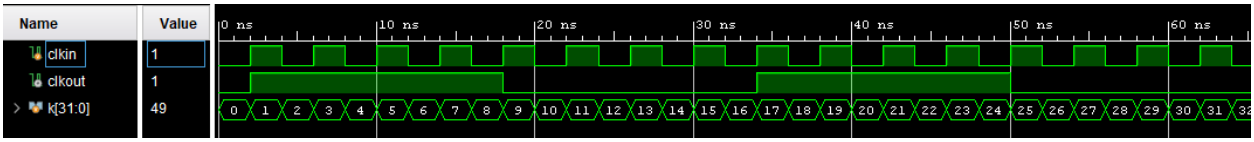


Figure 21. Clkdiv10

Mux2 to 1:

Verilog	Testbench

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
	module mux_tb;
	// Declaring Inputs
	reg d0;
	reg d1;
	reg s;
	
	// Declaring Outputs
	wire y;
	
	// Instantiate the Unit Under Test (UUT)
	mux2to1 uut(.d0(d0), .d1(d1), .s(s),.y(y));
	
	initial begin

	// Apply Inputs
	d0 = 0;
	d1 = 0;
	s = 0;
	// Wait 100 ns
	#100;
	
	//Similarly apply Inputs and wait for 100 ns
	d0 = 0; d1 = 0; s = 1; #100;
	d0 = 0; d1 = 1; s = 0; #100;
	d0 = 0; d1 = 1; s = 1; #100;
	d0 = 1; d1 = 0; s = 0; #100;
	d0 = 1; d1 = 0; s = 1; #100;
	d0 = 1; d1 = 1; s = 0; #100;
	d0 = 1; d1 = 1; s = 1; #100;
	end
	
	endmodule

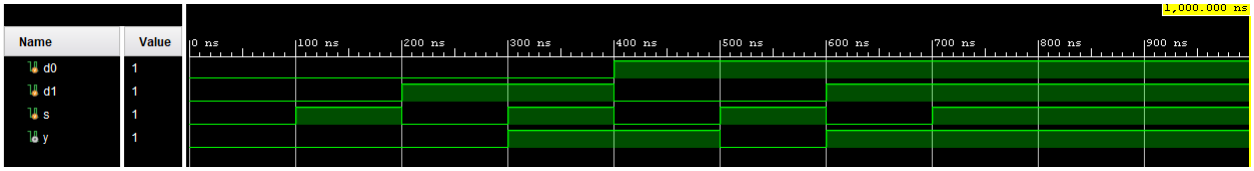


Figure 22. 2 to 1 multiplexer

BCD

Verilog	Testbench

`timescale 1ns/1ps
 	
module bcd(clk,dout);
 	
input clk;
output [3:0] dout;
 	
reg [3:0] dout,cnt;
 	
initial
dout=0;
initial
cnt=0;
always @(posedge clk)
begin
	`timescale 1ns/1ps
	module bcd_tb();
	reg clk;
	wire [3:0] dout;
	
	bcd uut(clk, dout);
	
	initial begin
	clk=0;
	forever #5 clk=~clk;
	end
	
	endmodule

if(dout==9) //condition to end count
begin
dout<=0;
end
 	
else
begin
cnt<=cnt+1;
dout=dout+1;
end
end
endmodule	

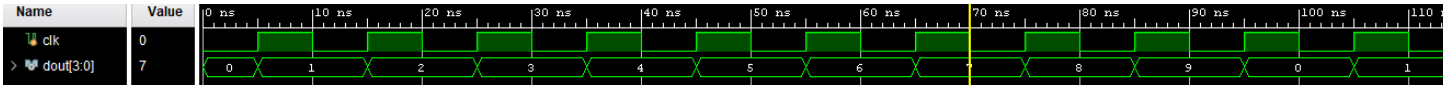


Figure 23. BCD

Final Two-Speed BCD Counter

Source Code	Testbench

module Fbcd(clk,s,y);
input clk,s;
output [2:0]y;
 	
wire [2:0]y;
 	
clkdiv10 g1(.clkin(clk),.clkout(y[0]));
mux2to1 g2(.d0(clk),.d1(y[0]),.s(s),.y(y[1]));
bcd g3(.clk(y[1]),.dout(y[2]));
 	
endmodule
	module Fbcd_tb;
	reg clk,s;
	wire [2:0]y;
	Fbcd uut(.clk(clk),.s(s),.y(y));
	
	always begin
	#10 clk = 1'b1;
	#10 clk = ~clk;
	#10 s = 1'b1;
	#10 s= ~s;
	end
	
	endmodule

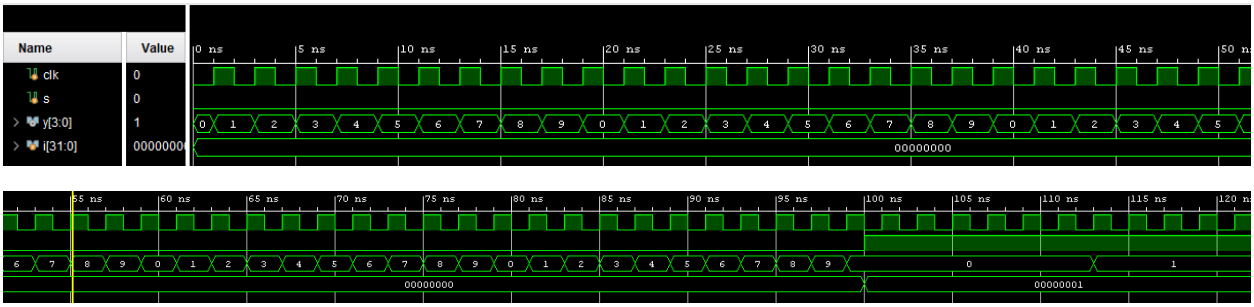


Figure 24. Final Two-Speed BCD Counter

Result Discussion:

The result was correct. The result does come out with the CLK lagging every 100ms, so it is 10 time slower than the beginning and after it goes pass the value 9, it repeats all over again. The difficult part of this lab was creating the BCD Verilog coding. It seems like the clkdiv10 coding from figure 21 but since it had to do with a 4-bit output it made it more confusing to code. I assumed it just require me adding arrays to the output, but it was not correct. I later was able to get some help that made me realize that I didnt including a counter in my initial coding which made my code run without going back to 0, making my number look weird. After plugging that part in, my coding ran fine since the Verilog coding from the final two-speed BCD counter was straightforward.

Part 4: Automatic Beverage Vending Machine

Design Purpose:

The Design of this was that it was be an Automatic Beverage Vending Machine. It only takes in one, two and five cents. A finite state machine should be designed to see the outcome of the vending machine. There should be 5 inputs that involves clock and reset. The vending machine should have two output with one having a 3-bit port size. Every time a coin is insert into the machine it should go to logic high indicating that it has received the coin. After reaching a certain amount it should release a drink and reset the number and add on the remaining coins.

Verilog Design:

Port Names	Port Direction	Port Size
clk	input	1 bit
reset	input	1 bit
one	input	1 bit
two	input	1 bit
five	input	1 bit
d	output	1 bit
r	output	3 bits

Figure 25. Automatic Beverage Vending machine Interface Signals

- When the input "reset" signal is logic high, it resets the machine to a starting state.
- When the input "one" signal is logic high, it indicates that the 1-cent token has been inserted into the vending machine.
- When the input "two" signal is logic high, it indicates that the 2-cent token has been inserted into the vending machine.
- When the input "five" signal is logic high, it indicates that the 5-cent token has been inserted into the vending machine.
- At any time, only one token can be inserted into the vending machine.
- When the output "d" signal is logic high, the vending machine returns a drink, and the output "r" signal displays the total coin token returned by the vending machine.
- When the output "d" signal is logic low, the output "r" signal should be zero.

Figure 26. Interface Signals Results

Figure 25 and 26 shows what the finite state machine and Verilog code should do. 25 is the interface signal while 26 tells you what it should be doing for each inputs and outputs.

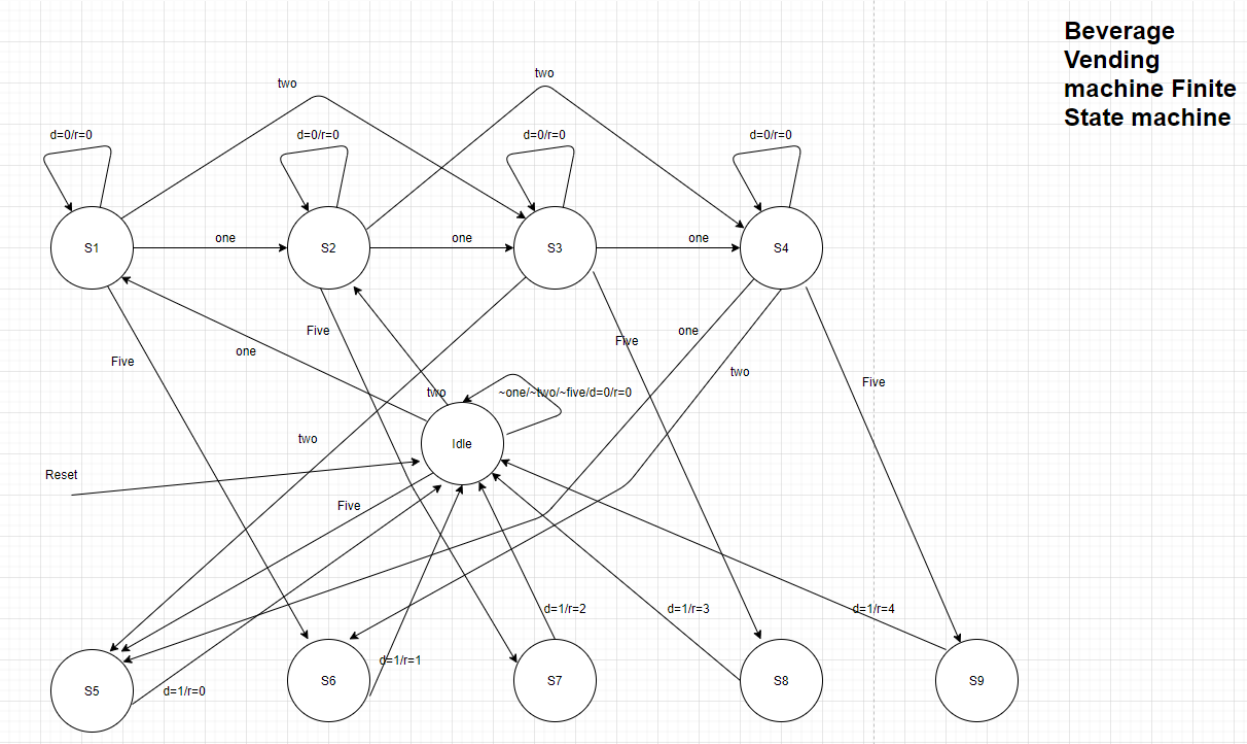


Figure 27. Beverage Vending Machine Finite State Machine

Figure 27 was the finite state machine that I create with the information given. The idle should be the initial state when there is nothing being done. As coins are being inserted into the machine number it should be shifting through the states until its through a certain amount, goes back to the idle state and start over again.

Verilog Code:

BVM

Source Code	Testbench

module bvm(clk,reset,one,two,five,d,r);
input clk,reset,one,two,five;
output d;
output [2:0] r;
parameter
idleState=0,s1=1,s2=2,s3=3,s4=4,s5=5,s6=6,s7=7,s8=8,s9=9;
reg [3:0]currentState, nextState;
reg[2:0]r;
reg d;
 	
always@(posedge clk or posedge reset)
begin
  if(reset) currentState <= idleState;
  else currentState <= nextState;
end
 	
always@(currentState or one or two or five or posedge reset)
begin
  case(currentState)
  idleState:
  if(one) nextState =s1;
  else if(two) nextState = s2;
  else if(five) nextState =s5;
  else nextState = idleState;
  s1:
  if(one)nextState = s2;
  else if(two) nextState = s3;
  else if(five) nextState =s6;
  else nextState =s1;
  s2:
  if(one) nextState =s3;
  else if(two) nextState= s4;
  else if(five) nextState =s7;
  else nextState =s2;
  s3:
  if(one) nextState= s4;
  else if(two) nextState=s5;
  else if(two) nextState=s8;
  else nextState= s3;
  s4:
  if(one) nextState = s5;
  else if(two) nextState = s6;
	module bvm_tb();
	reg clk, reset, one, two, five;
	wire d;
	wire [2:0] r;
	abvm u1(.clk(clk),
	.reset(reset), .one(one),
	.two(two), .five(five), .d(d),
	.r(r));
	initial clk =0;
	always #2 clk = ~clk;
	initial begin
	 reset = 1;one =0;two = 0;five
	=0;
	
	  #5 reset = 0; one =1;two =0;
	five=0;
	  #5 one =0; two =0; five =1;
	  #5 reset =1; one =0; two =0;
	five =0;
	  #5 reset = 0; one =1; two=0;
	five=0;
	  #5 one =0; two =1; five =0;
	  #5 one =0; two = 0; five =
	1;
	  #5 $stop;
	 end
	endmodule

```
        else if(five) nextState = s9;
        else nextState = s4;
    s5:
        nextState = idleState;
    s6:
        nextState = idleState;
    s7:
        nextState = idleState;
    s8:
        nextState = idleState;
    s9:
        nextState = idleState;
    default: nextState = idleState;
 endcase
 end

always@(currentState or one or two or five)
begin
case(currentState)
idleState:
if(one)
begin
    d = 0;
    r = 0;
end
else if (two)
begin
    d = 0;
    r = 0;
end
else if (five)
begin
        d = 0; //0
        r = 0;
    end
    else
    begin
        d = 0;
        r = 0;
    end

s1: if (one)
begin
    d = 0;
    r = 0;
end
```

```
else if (two)
begin
   d = 0;
   r = 0;
end
else if (five)
begin
      d = 1;
      r = 1;
   end
   else
   begin
      d = 0;
      r = 0;
  end

s2: if (one)
begin
   d = 0;
   r = 0;
end
else if (two)
begin
   d = 0;
   r = 0;
end
else if (five)
begin
      d = 1;
      r = 2;
   end
   else
   begin
      d = 0;
      r = 0;
   end


s3: if (one)
begin
   d = 0;
   r = 0;
end
else if (two)
begin
   d = 0;
```

```
    r = 0;
end
else if (five)
begin
        d = 1;
        r = 3;
    end
    else
    begin
        d = 0;
        r = 0;
    end

s4: if (one)
begin
    d = 1;
    r = 0;
end
else if (two)
begin
    d = 1;
    r = 1;
end
else if (five)
begin
        d =1;
        r = 4;
    end
    else
    begin
        d = 0;
        r = 0;
    end

s5:

    begin
        d = 1;
        r = 0;
    end

s6:

    begin
        d = 1;
        r = 1;
```

```
    end

s7:

    begin
        d = 1;
        r = 2;
    end

s8:

    begin
        d = 1;
        r = 3;
    end

s9:

    begin
        d = 1;
        r = 4;
    end
endcase
end

endmodule
```

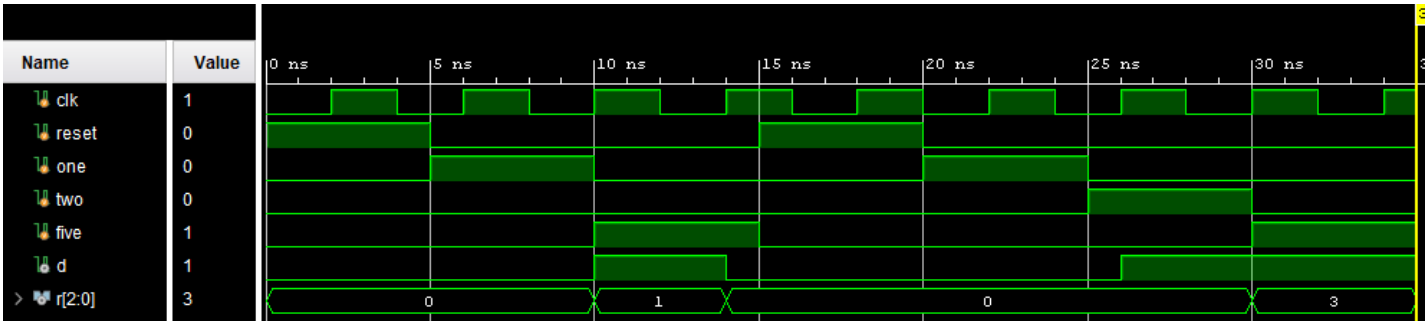


Figure 28. Automatic Beverage Machine Waveform

Result Discussion:

The result was good. The result that I got at the end matches my expectation of what was supposed to happen. My first issue was that I didn't really understand why it have 9 finite state machine, later on I was able to figure out that with 5 inputs they would need to go somewhere if a 5 coin was just insert in so 9 states was needed. The other issue I had was just some error with

my testbench which made my r number on the waveform funky because of me using too many resets at the beginning but it was later fixed and made simpler.

Conclusion:

Overall, this lab was very helpful in me learning more about hierarchical Verilog along with combinational and sequential logic work. It was not as complicated at the beginning but as we work on toward the last part of the lab, we had to think a bit more. Each part of the lab does contribute to each other making it more helpful as we just can reuse codes that had already been made. For the me, the most complicated part was making the testbench. Without a good testbench I would not be able to test out the Verilog coding to see if it is correct, but after trial and error I would eventually get it to work which is quite pleasing. Doing all these labs was challenging but it was quite helpful in learning more about Verilog and making adequate testbenches.