CPE166 Advanced Logic Design

Lab Number 2

Anthony Chavez

Table of Contents

Introduction

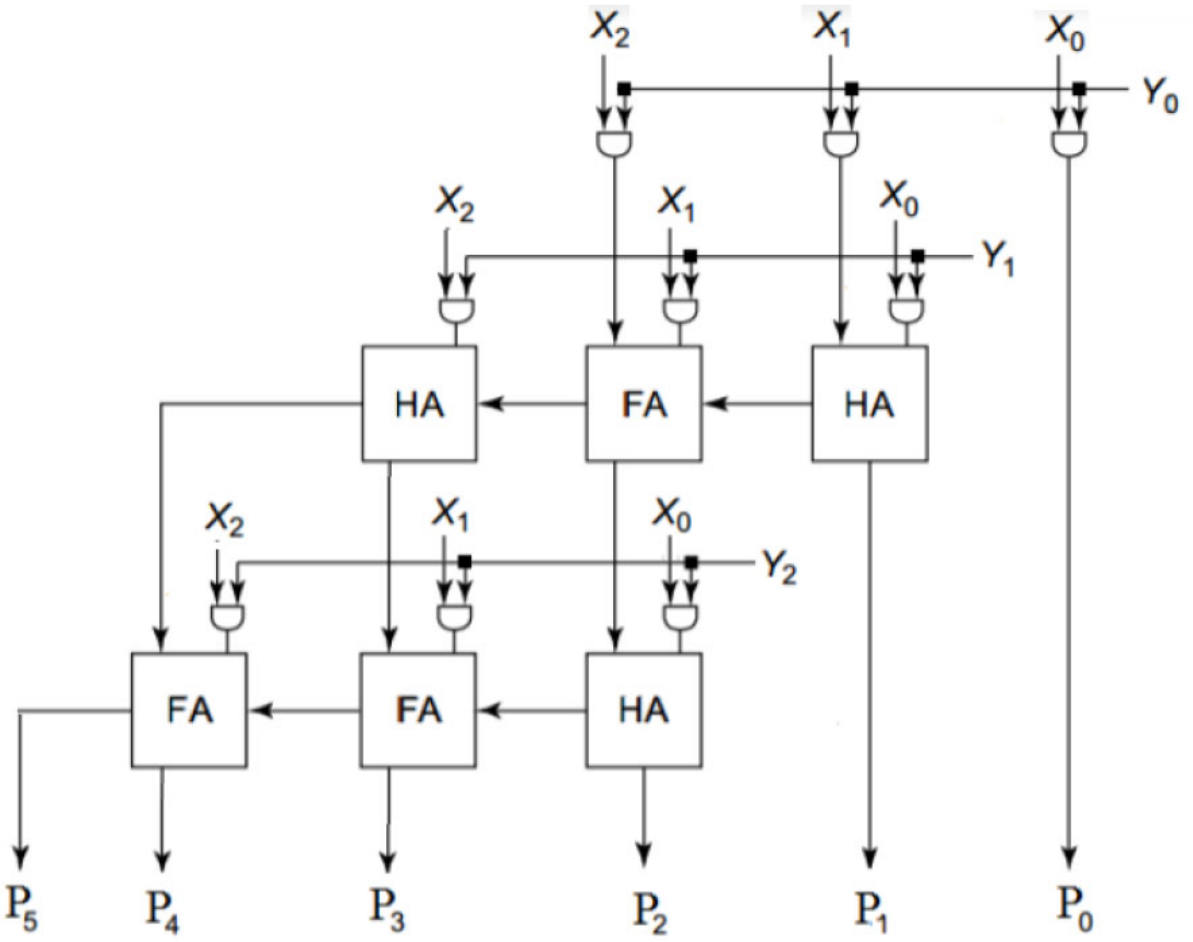
Lab 2 was a four-part lab that served as an introduction to circuit designing in Verilog. The four parts of the lab were a 3 by 3 binary combinational array multiplier, an 8-bit carry select adder, a two-speed BCD counter, and an automatic beverage vending machine. All code and test bench simulations were written and tested in the Xilinx Vivado IDE.

Laboratory Report

Part 1 – 3 By 3 Binary Combinational Array Multiplier

Design Purpose:

Two 3-bit unsigned binary numbers will be multiplied together. For example, “010” which is ‘2’ multiplied with “011” which is ‘3’ would result in ‘6’ being the product of the two binary numbers. To achieve this result, we will create a Half Adder and Full Adder circuit, then configure the circuits according to the provided diagram below. The Half Adder takes in two 1-bit binary numbers (a, b) and outputs two 1-bit binary numbers (cout, sum). The Full Adder is similar in design, but instead of 2 1-bit binary number inputs, there are 3 1-bit binary number inputs (a, b, cin). The product of two 3-bit unsigned binary numbers should be a 6-bit unsigned binary number (P).



Verilog Design:

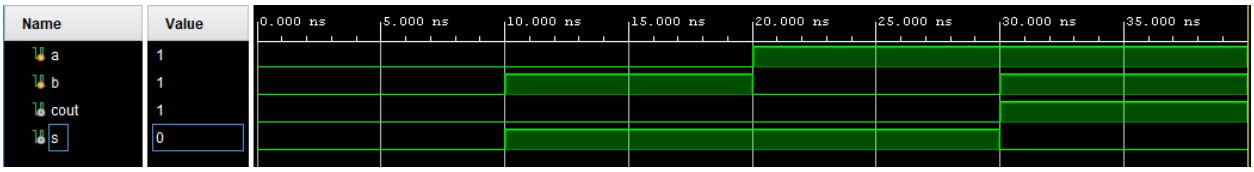
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/5/20
// Module Name: halfAdder
// Description: half adder
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////


module halfAdder(a,b,cout,s);
   input a,b;
   output cout,s;

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/5/20
// Module Name: fullAdder
// Description: full adder with carry out and sum
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////


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

   halfAdder g1(a,b,n,m);
   halfAdder g2(cin,m,p,sum);
   or g3(cout,p,n);
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/5/20
// Module Name: multiplier
// Description: multiply two 3 bit numbers together
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////


module multiplier(x,y,p);
   input[2:0] x,y;
   output[5:0] p;
   wire[7:0] andNums;
   wire[2:0] fullSum,fullCarry;
   wire[2:0] halfSum,halfCarry;

   assign p[0] = x[0]&y[0];
   assign andNums[0] = x[1]&y[0];
   assign andNums[1] = x[2]&y[0];
   assign andNums[2] = x[0]&y[1];
   assign andNums[3] = x[1]&y[1];
   assign andNums[4] = x[2]&y[1];
   assign andNums[5] = x[0]&y[2];
   assign andNums[6] = x[1]&y[2];
   assign andNums[7] = x[2]&y[2];

   halfAdder g1(andNums[0],andNums[2],halfCarry[0],halfSum[0]);
   assign p[1] = halfSum[0];
   fullAdder g2(andNums[1],andNums[3],halfCarry[0],fullCarry[0],fullSum[0]);
   halfAdder g3(fullSum[0],andNums[5],halfCarry[2],halfSum[2]);
   assign p[2] = halfSum[2];
   halfAdder g4(andNums[4],fullCarry[0],halfCarry[1],halfSum[1]);
   fullAdder g5(andNums[6],halfSum[1],halfCarry[2],fullCarry[1],fullSum[1]);
   assign p[3] = fullSum[1];
   fullAdder g6(halfCarry[1],andNums[7],fullCarry[1],fullCarry[2],fullSum[2]);
   assign p[4] = fullSum[2];
   assign p[5] = fullCarry[2];
endmodule
```

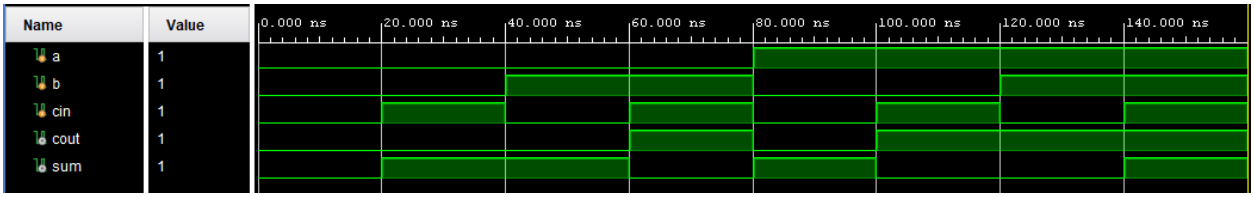
Verilog Testbench Design and Simulation Waveforms:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/5/20
// Module Name: halfAdder_tb
// Description: half adder testbench
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////

module halfAdder_tb;
   reg a,b;
   wire cout,s;

   halfAdder g1(.a(a),.b(b),.cout(cout),.s(s));
   initial begin
      a=0; b=0;
      #10 a=0; b=1;
      #10 a=1; b=0;
      #10 a=1; b=1;
      #10 $stop;
   end

endmodule
```



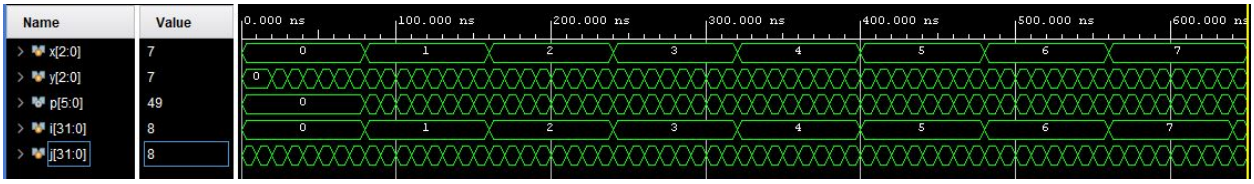
Here we have the four possible outputs of a half adder. When a=0 and b=0, we expect to see carry-out and sum to be 0 which is shown at 0 ns to 10 ns. When a=0 and b=1 or a=1 and b=0, we expect to see carry-out to be 0 and sum to be 1. Now if both a and b equal 1, we will get carry-out=1 and sum=1.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/5/20
// Module Name: fullAdder_tb
// Description: full adder test bench
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////


module fullAdder_tb;
   reg a,b,cin;
   wire cout,sum;

   fullAdder uut(a,b,cin,cout,sum);
   initial begin
      a=1'b0; b=1'b0; cin=1'b0;
      #20 a=1'b0; b=1'b0; cin=1'b1;
      #20 a=1'b0; b=1'b1; cin=1'b0;
      #20 a=1'b0; b=1'b1; cin=1'b1;
      #20 a=1'b1; b=1'b0; cin=1'b0;
      #20 a=1'b1; b=1'b0; cin=1'b1;
      #20 a=1'b1; b=1'b1; cin=1'b0;
      #20 a=1'b1; b=1'b1; cin=1'b1;
      #20 $stop;
   end
endmodule
```



For a full adder, there are eight possible combinations which can be seen in the simulation. If sum exceeds one bit, the additional bit is taken by the carry-out. For example, if a=1, b=1, cin=1, then sum=11 in binary form. However, the sum can only store one bit, so the carry-bit takes the most significant bit. The full adder is similar to the half adder with the exception of having an additional input, carry-in (cin).

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/14/20
// Module Name: multiplier_tb
// Description: multiplier test bench
// Project Part Number: Lab 2, Part 1
//////////////////////////////////////////////////////////////////////////////////


module multiplier_tb;
   reg[2:0] x,y;
   wire[5:0] p;
   integer i,j;

   multiplier uut(x,y,p);
   initial begin
      x=3'b000; y=3'b000;
      for(i=0; i<8; i=i+1) begin
         x=i;
         for(j=0; j<8; j=j+1) begin
            #10 y=j;
         end
      end
      #10 $stop;
   end
endmodule
```

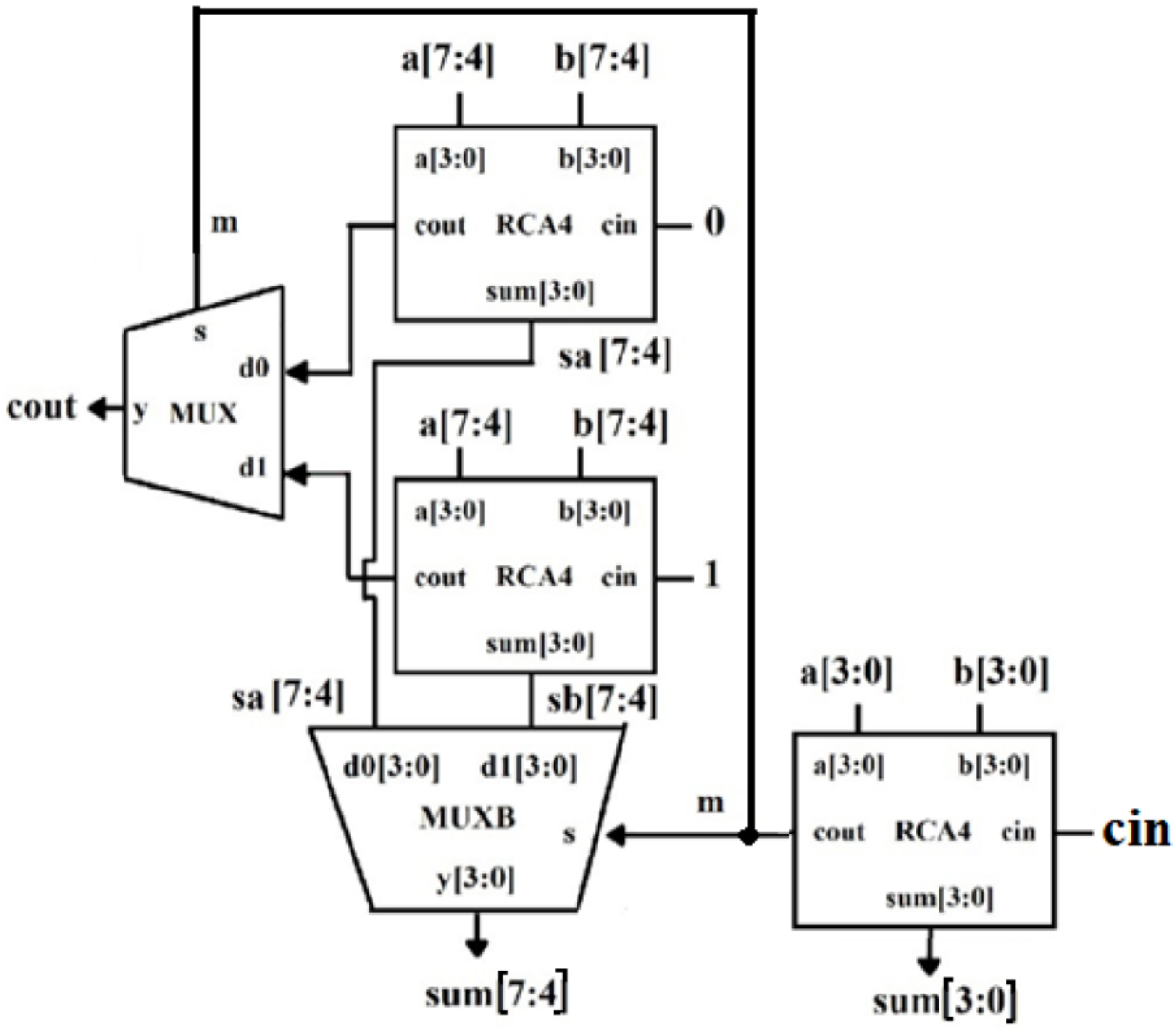


Here the simulation is used to ensure the circuit is properly handling all 64 possible cases. The circuit multiples two 3-bit binary numbers so we expect the product (p) to be calculated properly. Viewing the values in unsigned decimal form, if the x*y doesn't equal to expected product, there must be a logic error.

Result Discussion:
For this part, I was able to successfully demonstrate the 3 by 3 binary combinational array multiplier. While checking the simulation, I verified that all the possible combinations were computed correctly. One of the problems I ran into was writing the for loops in the last test bench to check every combination. I was a little confused on how the time function worked when using "#10" in certain spots. Eventually, I found the error and fixed the problem.

Part 2 – 8-Bit Carry Select Adder

Design Purpose:
To design the 8-bit carry select adder circuit provided (see figure below). The circuit consists of three 4-bit ripple carry adders (RCA4) and two multiplexers. One of the RCA4s assumes the carry-in to be zero and the other assumes the carry-in to be one. The multiplexers are used to select the correct sum and the carry-out based on the know carry-in value given by the first RCA4 at the far right.



Verilog Design:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: rca4
// Description: 4-bit Ripple Carry Adder
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


module rca4(a,b,cin,cout,sum);
   input[3:0] a,b;
   input cin;
   output[3:0] sum;
   output cout;
   wire[3:0] fullCarry;

   fullAdder g1(a[0],b[0],cin,fullCarry[0],sum[0]);
   fullAdder g2(a[1],b[1],fullCarry[0],fullCarry[1],sum[1]);
   fullAdder g3(a[2],b[2],fullCarry[1],fullCarry[2],sum[2]);
   fullAdder g4(a[3],b[3],fullCarry[2],fullCarry[3],sum[3]);
   assign cout = fullCarry[3];
endmodule
```

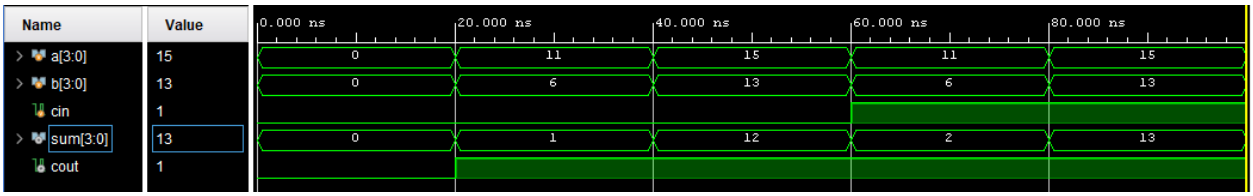
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: mux
// Description: multiplexer
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


module mux(s,d0,d1,y);
   input s,d0,d1;
   output y;
   reg y;

   always@(d0 or d1 or s) begin
      if(s==0)
         y=d0;
      else
         y=d1;
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: muxb
// Description: multiplexer with 2 4-bit inputs
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


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

   always@(d0 or d1 or s) begin
      if(s==0)
         y[3:0]=d0[3:0];
      else
         y[3:0]=d1[3:0];
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: csa8
// Description: 8-bit Carry Select Adder
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


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

   rca4 g1(a[3:0],b[3:0],cin,m,sum[3:0]);
   rca4 g2(a[7:4],b[7:4],0,n,sa[3:0]);
   rca4 g3(a[7:4],b[7:4],1,o,sb[3:0]);
   muxb g4(m,sa[3:0],sb[3:0],sum[7:4]);
   mux g5(m,n,o,cout);
endmodule
```

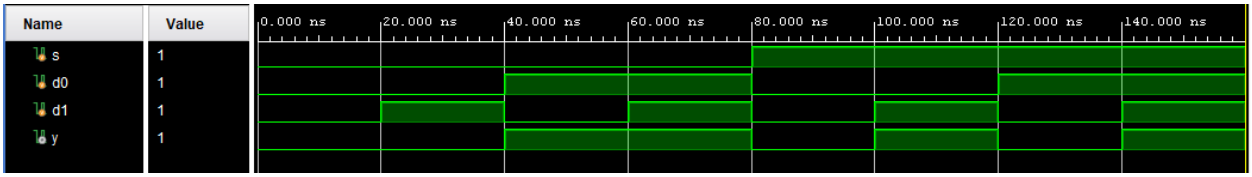
Verilog Testbench Design and Simulation Waveforms:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: rca4_tb
// Description: 4-bit Ripple Carry Adder Test Bench
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


module rca4_tb;
   reg[3:0] a,b;
   reg cin;
   wire[3:0] sum;

   rca4 uut(a,b,cin,cout,sum);
   initial begin
      a=4'b0000; b=4'b0000; cin=1'b0;
      #20 a=4'b1011; b=4'b0110; cin=1'b0;
      #20 a=4'b1111; b=4'b1101; cin=1'b0;
      #20 a=4'b1011; b=4'b0110; cin=1'b1;
      #20 a=4'b1111; b=4'b1101; cin=1'b1;
      #20 $stop;
   end
endmodule
```



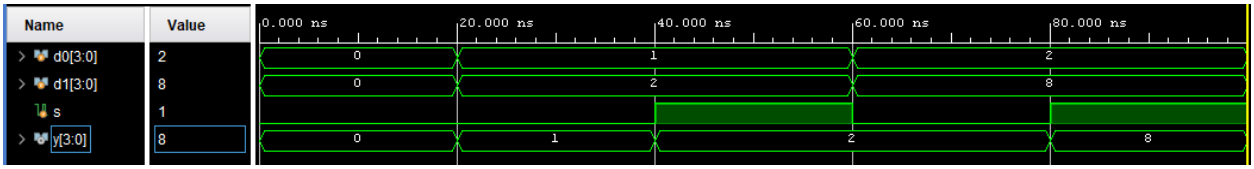
Here we verify the 4-bit ripple carry adder circuit is properly configured. We know that a 4-bit binary number has a maximum value of 15. Therefore, carry-out will take the extra bit at the most significant bit position. In addition, a carry-in (cin) bit must be considered, if cin=1 an additional ‘1’ must be added to the sum of the least significant bit position.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: mux_tb
// Description: multiplexer test bench
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


module mux_tb;
   reg s,d0,d1;
   wire y;

   mux uut(s,d0,d1,y);
   initial begin
      s=1'b0; d0=1'b0; d1=1'b0;
      #20 s=1'b0; d0=1'b0; d1=1'b1;
      #20 s=1'b0; d0=1'b1; d1=1'b0;
      #20 s=1'b0; d0=1'b1; d1=1'b1;
      #20 s=1'b1; d0=1'b0; d1=1'b0;
      #20 s=1'b1; d0=1'b0; d1=1'b1;
      #20 s=1'b1; d0=1'b1; d1=1'b0;
      #20 s=1'b1; d0=1'b1; d1=1'b1;
      #20 $stop;
   end
endmodule
```



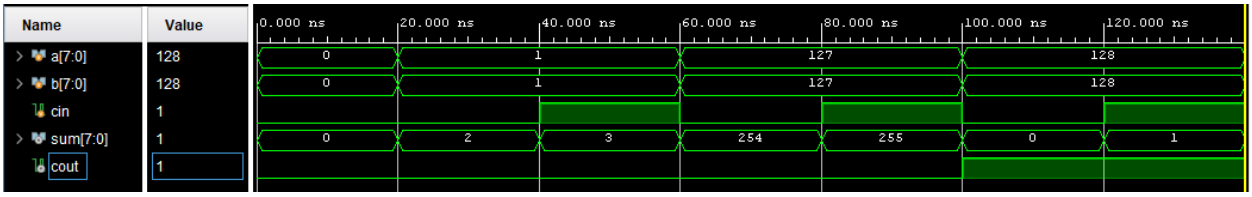
All eight possible combinations were checked in the simulation to ensure the multiplexer functions properly. If the multiplexer chooses to output the wrong input, the 8-bit carry select adder would have the incorrect carry-out. If the input s=0, y=d0 is expected and looking at cases 0 ns to 80 ns holds true. Same goes for when input s=1, y=d1 is expected and looking at cases 80 ns to 160 ns holds true as well.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: muxb_tb
// Description: multiplexer with two 4-bit inputs test bench
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


module muxb_tb;
   reg[3:0] d0,d1;
   reg s;
   wire[3:0] y;

   muxb uut(s,d0,d1,y);
   initial begin
      s=1'b0; d0=4'b0000; d1=4'b0000;
      #20 s=1'b0; d0=4'b0001; d1=4'b0010;
      #20 s=1'b1; d0=4'b0001; d1=4'b0010;
      #20 s=1'b0; d0=4'b0010; d1=4'b1000;
      #20 s=1'b1; d0=4'b0010; d1=4'b1000;
      #20 $stop;
   end
endmodule
```



This multiplexer is like the previous multiplexer, however, this one takes two 4-bit binary numbers for d0 and d1 instead of two 1-bit binary numbers. Here we can see if s=0, then we expect y=d0 which can be seen at 20 ns to 40 ns. Now if s=1, then we expect y=d1 which an be seen at 40 ns to 60 ns.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/6/20
// Module Name: csa8_tb
// Description: 8-bit carry select adder test bench
// Project Part Number: Lab 2, Part 2
//////////////////////////////////////////////////////////////////////////////////


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

   csa8 uut(a,b,cin,cout,sum);
   initial begin
      a=8'b00000000; b=8'b00000000; cin=1'b0;
      #20 a=8'b00000001; b=8'b00000001; cin=1'b0;
      #20 a=8'b00000001; b=8'b00000001; cin=1'b1;
      #20 a=8'b01111111; b=8'b01111111; cin=1'b0;
      #20 a=8'b01111111; b=8'b01111111; cin=1'b1;
      #20 a=8'b10000000; b=8'b10000000; cin=1'b0;
      #20 a=8'b10000000; b=8'b10000000; cin=1'b1;
      #20 $stop;
   end
endmodule
```

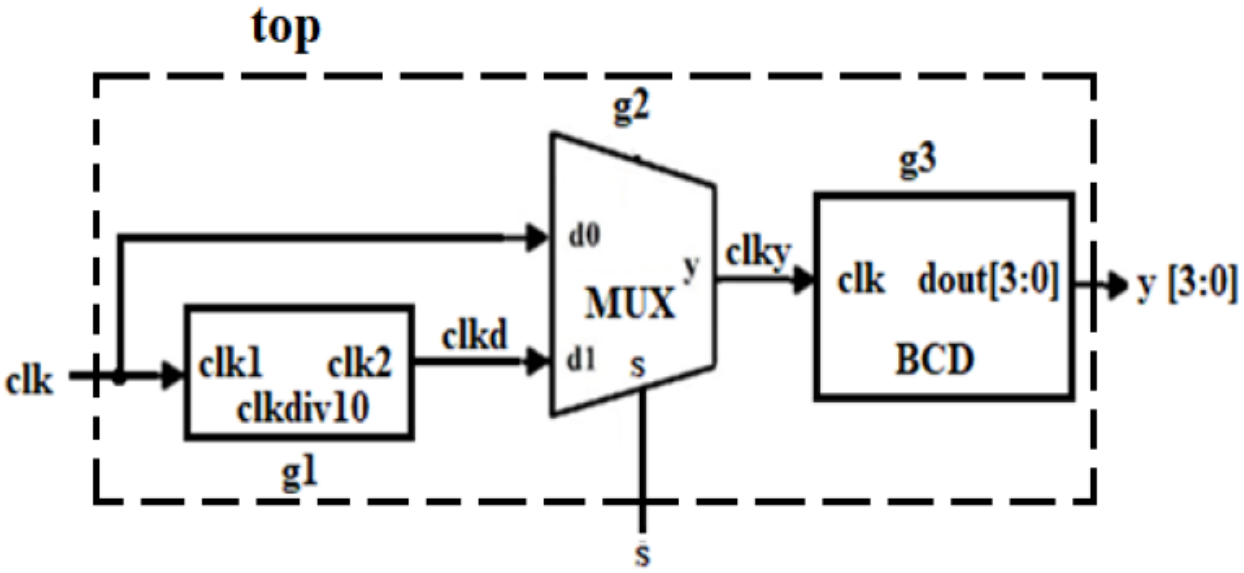


This simulation is to verify that the 8-bit carry select adder circuit functions properly. For example, 1+1=2 but with a carry-in we would have 1+1+1=3 which we get for both. However, an 8-bit binary number can only represent a decimal number up to 255. Therefore, 128+128=256 would result in 0 and 128+128+1=1, but both sums will require a carry-out which can be seen in the simulation.

Result Discussion:
I was successful in creating the 8-bit carry select adder circuit. I found it easier to calculate a few test cases by hand and verify simulation result matched than simply typing them into a calculator. This allowed me to understand how the bits move throughout the circuit.

Part 3 – Two-speed BCD Counter

Design Purpose:
To design a binary coded decimal (BCD) counter with two different speeds. The first speed setting will be a standard clock speed of 2 ns logic high and 2 ns logic low. The second speed will have a clock that is 10 times slower than the speed of the first clock setting. The circuit simply will count from 0 to 9 repeatedly with the selected clock speed.



Verilog Design:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: clkdiv10
// Description: Makes frequency of clk 10 times slower.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module clkdiv10(clk1,clk2);
   input clk1;
   output clk2;
   reg clk2;
   reg[3:0] cnt;

   initial cnt = 0;
   initial clk2 = 0;

   always@(posedge clk1) begin
      if(cnt == 9) begin
         cnt <= 0;
         clk2 <= 1;
      end
      else if (cnt < 4) begin
         cnt <= cnt + 1;
         clk2 <= 1;
      end
      else begin
         cnt <= cnt + 1;
         clk2 <= 0;
      end
   end
endmodule
```

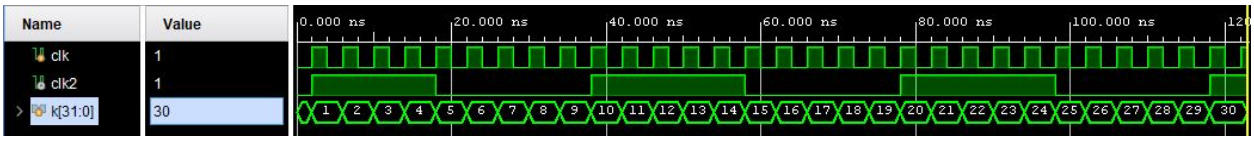
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: mux
// Description: multiplexer
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module mux(s,d0,d1,y);
   input s,d0,d1;
   output y;
   reg y;

   always@(d0 or d1 or s) begin
      if(s==0)
         y=d0;
      else
         y=d1;
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: BCD
// Description: BCD counter outputs a value from 0 to 9 on each rising edge of the
//              clk signal and then repeats the execution.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module BCD(clk, dout);
   input clk;
   output[3:0] dout;
   reg[3:0] dout;

   initial dout = 4'b0000;

   always@(posedge clk) begin
      if(dout == 9) dout <= 4'b0000;
      else dout <= dout + 1;
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: top
// Description: Two-speed BCD counter circuit.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module top(clk,s,y);
   input clk,s;
   output[3:0] y;
   wire clkd, clky;

   clkdiv10 g1(clk, clkd);
   mux g2(s, clk, clkd, clky);
   BCD g3(clky,y[3:0]);
endmodule
```

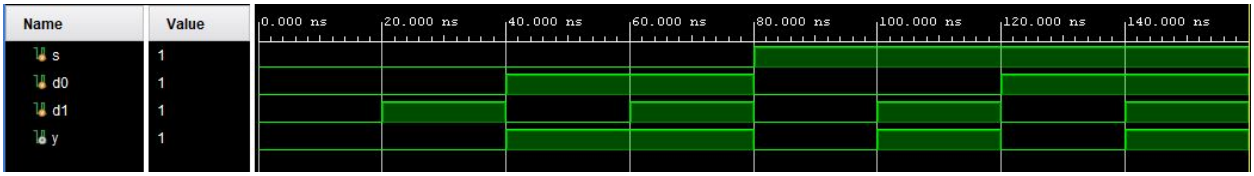
Verilog Testbench Design and Simulation Waveforms:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: clkdiv10_tb
// Description: clkdiv10 test bench.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module clkdiv10_tb;
   reg clk;
   wire clk2;
   integer k;

   clkdiv10 uut(clk,clk2);

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

   initial begin
      k = 0;
      while(k != 30) begin
         @(posedge clk);
         $display($time, "ns, k=%d, clk2=%b", k, clk2);
         k = k + 1;
      end
      #5 $stop;
   end
endmodule
```



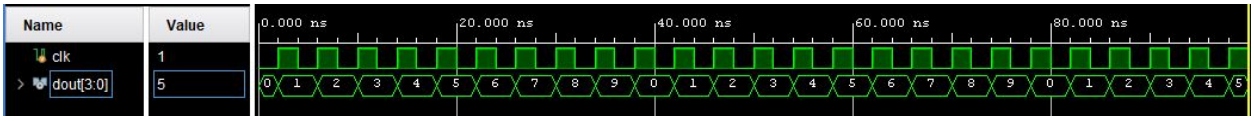
This testbench verifies that the input clock speed (clk) is divided by ten, meaning the output clock speed (clk2) is ten times slower than the original clock speed. As shown in the waveform, five clk cycles make up for one clk2 cycle when clk2 is logic high as well as when clk2 is logic low.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: mux_tb
// Description: multiplexer test bench
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module mux_tb;
   reg s,d0,d1;
   wire y;

   mux uut(s,d0,d1,y);
   initial begin
      s=1'b0; d0=1'b0; d1=1'b0;
      #20 s=1'b0; d0=1'b0; d1=1'b1;
      #20 s=1'b0; d0=1'b1; d1=1'b0;
      #20 s=1'b0; d0=1'b1; d1=1'b1;
      #20 s=1'b1; d0=1'b0; d1=1'b0;
      #20 s=1'b1; d0=1'b0; d1=1'b1;
      #20 s=1'b1; d0=1'b1; d1=1'b0;
      #20 s=1'b1; d0=1'b1; d1=1'b1;
      #20 $stop;
   end
endmodule
```



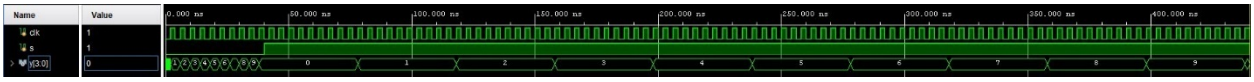
All eight possible combinations were checked in the simulation to ensure the multiplexer functions properly. If the multiplexer chooses to output the wrong input, the 8-bit carry select adder would have the incorrect carry-out. If the input s=0, y=d0 is expected and looking at cases 0 ns to 80 ns holds true. Same goes for when input s=1, y=d1 is expected and looking at cases 80 ns to 160 ns holds true as well.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: BCD_tb
// Description: BCD test bench.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module BCD_tb;
   reg clk;
   wire[3:0] dout;

   BCD uut(clk, dout);

   initial begin
      clk = 0;
      #100 $stop;
   end

   always #2 clk = ~clk;

   initial $monitor($time, " ns, dout=%d", dout);
endmodule
```

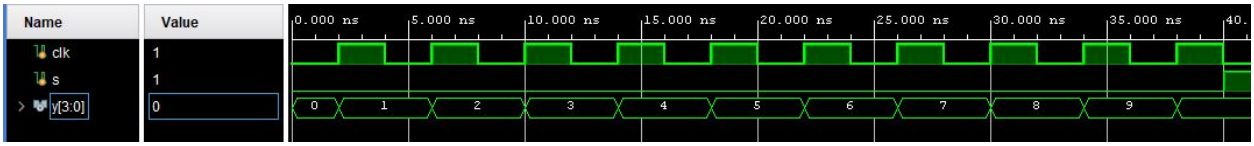


This testbench verifies that the BCD circuit properly counts from 0 to 9 and then repeats. Every positive edge of the clock should trigger dout to output the respectable number.

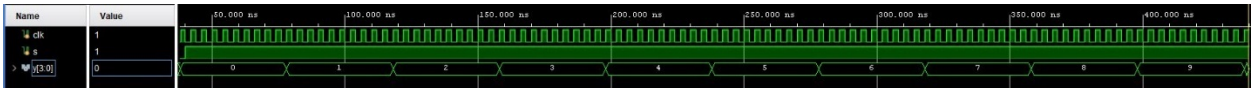
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/20/20
// Module Name: top_tb
// Description: Two-speed BCD counter circuit test bench.
// Project Part Number: Lab 2, Part 3
//////////////////////////////////////////////////////////////////////////////////


module top_tb;
   reg clk, s;
   wire[3:0] y;

   top uut(clk, s, y);

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

   initial begin
      s = 1'b0;
      #40 s = 1'b1;
      #400 $stop;
   end
endmodule
```



This testbench shows the final two speed BCD counter.



When s=0 the user defined clock speed is unchanged. NOTE: clk changes every 2 ns.



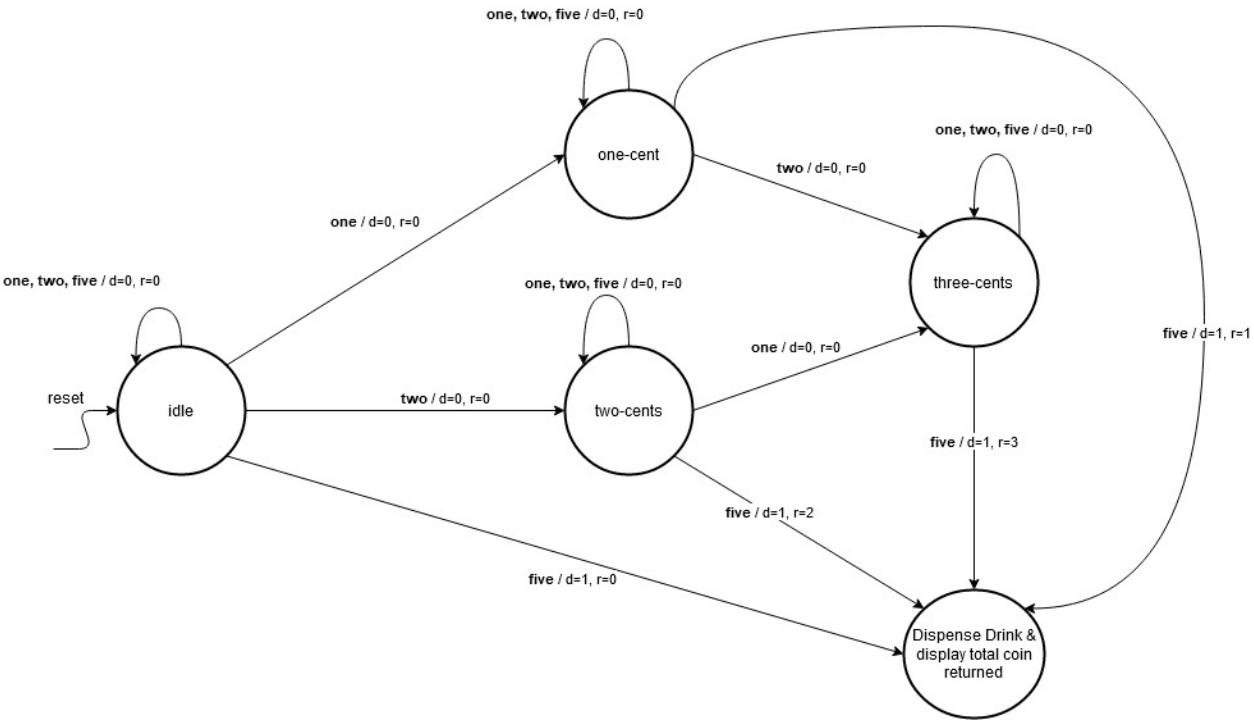
When s=1 the clock speed is ten times slower than the user defined clock speed.

Result Discussion:
I successfully implemented the Two-speed BCD Counter circuit. This part of the lab helped me understand how to properly configure a clock as well as how to test and analyze the waveforms.

Part 4 - Automatic Beverage Vending Machine

Design Purpose:
To design an automatic beverage vending machine. The token price of a beverage is 5 cents, and the machine only accepts tokens of 1 cent, 2 cents, and 5 cents. First, we will design a finite state machine (FSM) for the machine, then implement the project in Verilog. This lab shall be used as an introduction to case statements in Verilog.

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



Verilog Design:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/23/20
// Module Name: ABVM
// Description: Automatic Beverage Vending Machine
// Project Part Number: Lab 2, Part 4
//////////////////////////////////////////////////////////////////////////////////


module ABVM(clk, reset, one, two, five, d, r);
   input clk, reset, one, two, five;
   wire three;
   output d;
   output[2:0] r;
   reg d;
   reg[2:0] r;
   reg[3:0] cs, ns; // current state, next state

   parameter S0=4'b0000, // idle state
             S1=4'b0001, // one-cent
             S2=4'b0010, // two-cent
             S3=4'b0100, // three-cent
             S4=4'b1000; // dispense

   always@(posedge clk or posedge reset) begin
      if(reset) cs <= S0;
      else cs <= ns;
   end

   always@(cs or one or two or five) begin // controls the next states based on input
      case(cs)
         S0: if(one) ns = S1;
            else if(two) ns = S2;
            else if(five) ns = S4;
            else ns = S0;

         S1: if(two) ns = S3;
            else if(five) ns = S4;
            else ns = S1;

         S2: if(one) ns = S3;
            else if(five) ns = S4;
            else ns = S2;
```

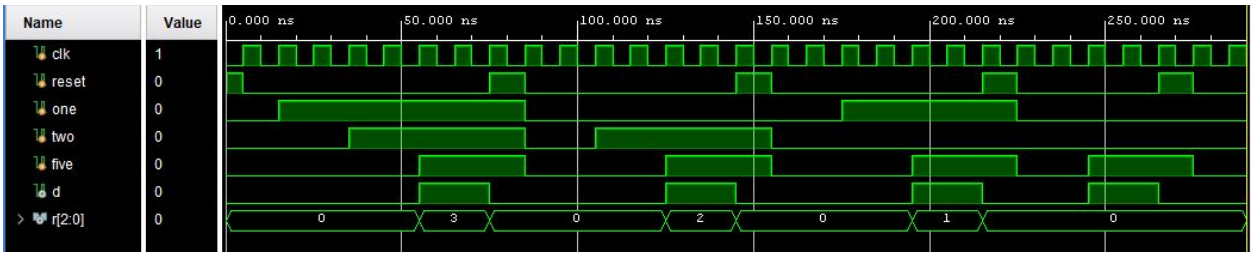
```
      S3: if(five) ns = S4;
         else ns = S3;

      S4: if(reset) ns = S0;
         else ns = S4;

      default: ns = S0;
   endcase
end

always@(cs or one or two or five) begin // controls outputs
   case(cs)
      S0: if(one) begin
            d = 0;
            r = 0;
         end
         else if(two) begin
            d = 0;
            r = 0;
         end
         else if(five) begin
            d = 0;
            r = 0;
         end
         else begin
            d = 0;
            r = 0;
         end

      S1: 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(five) begin
              d = 1;
              r = 2;
            end
            else begin
              d = 0;
              r = 0;
            end

        S3: if(five) begin
              d = 1;
              r = 3;
            end
            else begin
              d = 0;
              r = 0;
            end

        S4: if(reset) begin
              d = 0;
              r = 0;
            end
            else begin
              d = 1;
            end

        default: begin
                   d = 0;
                   r = 0;
                 end
      endcase
   end
endmodule
```

Verilog Testbench Design and Simulation Waveforms:

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Author: Anthony Chavez
// Date Last Revised: 9/23/20
// Module Name: ABVM_tb
// Description: Automatic Beverage Vending Machine
// Project Part Number: Lab 2, Part
4////////////////////////////////////////////////////////////////////////////////


module ABVM_tb;
   reg clk, reset, one, two, five;
   wire d;
   wire[2:0] r;

   ABVM uut(clk, reset, one, two, five, d, r);

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

   initial begin
      one = 0;
      two = 0;
      five = 0;
      reset = 1;
   end

   initial begin
      // Test One: 8-cents
      #5 reset = 0;
      #10 one = 1;
      #20 two = 1;
      #20 five = 1;

      // Reset
      #20 reset = 1;
      #10 reset = 0;
      one = 0; two = 0; five = 0;

      // Test Two: 7-cents
      #20 two = 1;
      #20 five = 1;
```

```
    // Reset
    #20 reset = 1;
    #10 reset = 0;
    one = 0; two = 0; five = 0;

    // Test Three: 6-cents
    #20 one = 1;
    #20 five = 1;

    // Reset
    #20 reset = 1;
    #10 reset = 0;
    one = 0; two = 0; five = 0;

    // Test Four: 5-cents
    #20 five = 1;

    // Reset
    #20 reset = 1;
    #10 reset = 0;
    one = 0; two = 0; five = 0;

    #15 $stop;
end
```



This testbench tests the various combinations of the tokens being inserted. For the first combination, tokens one, two, and five are inserted. A drink is dispensed when five-cents is received (d=1), but three-cents will be returned (r=3). The second combination, tokens two and five are inserted. A drink is dispensed (d=1), but two-cents is returned (r=2). The third combinations, tokens one and five are inserted. A drink is dispensed (d=1), but one-cent is returned (r=1). Finally, the last combination is simply the five-cent token. A drink is dispensed (d=1), but there is nothing to return so (r=0).

Result Discussion:
I successfully implemented the automatic beverage vending machine project. This part of the lab helped me understand how to configure case statements in Verilog. The only problem I came across during the lab was designing the finite state machine. Drawing the FSM in draw.io helped me to visualize all the combinations for the various inputs and outputs.

Conclusion

Before starting this class, I was a little worried about getting back into writing Verilog programs. During my Freshman year, in CPE 64, I was introduced to the language, but never fully understood how all the code worked together and how to utilize the waveforms. Since then, I have gained some programming experience in my computer science courses, so I have a better understanding of how to properly code. However, this lab has helped me to understand the Verilog language and how to analyze the waveforms better than I did before.

The first part of the lab helped me understand how to design gates and implement them into a whole working 3 by 3 Binary Combinational Array Multiplier. The second part of the lab helped to understand the use of vectors and how to utilize the waveforms to check if my complete circuit calculated the values properly. The third part of the lab helped me to understand how to configure a clock and see how it affects the output of the circuit. Finally the fourth part of the lab helped me to understand how case statements work in Verilog.