Gabriel Emerson
ELEC 4200 – Lab 2
09/02/21

Lab 2: Numbering Systems

Goals:

The goal for this lab is to get familiar with number systems and doing different systems of math in both decimal and binary terms. This lab starts off just using a 7 segment decoder to display numbers, then gets more into the number systems themselves, making a ripple carry adder, and lastly a look-ahead adder.

Design Process:

Getting started for this lab took some research into 7 segment decoders and how they are implemented since they are used for all 6 tasks in this lab. This became much easier after viewing the diagram given to use at the end of lab 1. This can be seen in Figure 1. After this, I began reading over the types of adders used in Verilog and how they are created, this led to a lot of helpful suggestions when starting on each task.

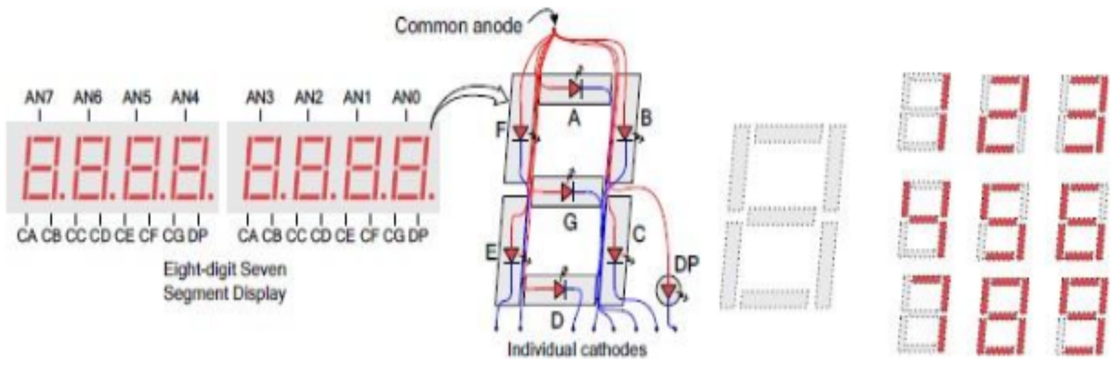


Figure 1

Task 2 took the first task and added a little more depth. For this task we are to take a 4 bit input and convert to be a 2 but decimal output. This means that anything more than 10 will set z = 1, and will display the rest on the 7 segment decoder. The output should be equal to the table shown in Figure 2.

v[3:0]	z	m[3:0]
0000	0	0000
0001	0	0001
0010	0	0010
0011	0	0011
0100	0	0100
0101	0	0101
0110	0	0110
0111	0	0111
1000	0	1000
1001	0	1001
1010	1	0000
1011	1	0001
1100	1	0010
1101	1	0011
1110	1	0100
1111	1	0101

Figure 2

Task 3 was very similar to that of task 2, however, it differed by converting a 4 bit input into a 2-out-of-5 code. This is done by simply finding the correct decimal input, to convert each bit into a 1. Knowing this, we can set each digit equal at different inputs. The output should match the table shown in Figure 3.

Decimal Digits	BCD (8-4-2-1)	6-3-1-1	Excess-3	2-out-of-5	Gray code
0	0000	0000	0011	00011	0000
1	0001	0001	0100	00101	0001
2	0010	0011	0101	00110	0011
3	0011	0100	0110	01001	0010
4	0100	0101	0111	01010	0110
5	0101	0111	1000	01100	1110
6	0110	1000	1001	10001	1010
7	0111	1001	1010	10010	1011
8	1000	1011	1011	10100	1001
9	1001	1100	1100	11000	1000

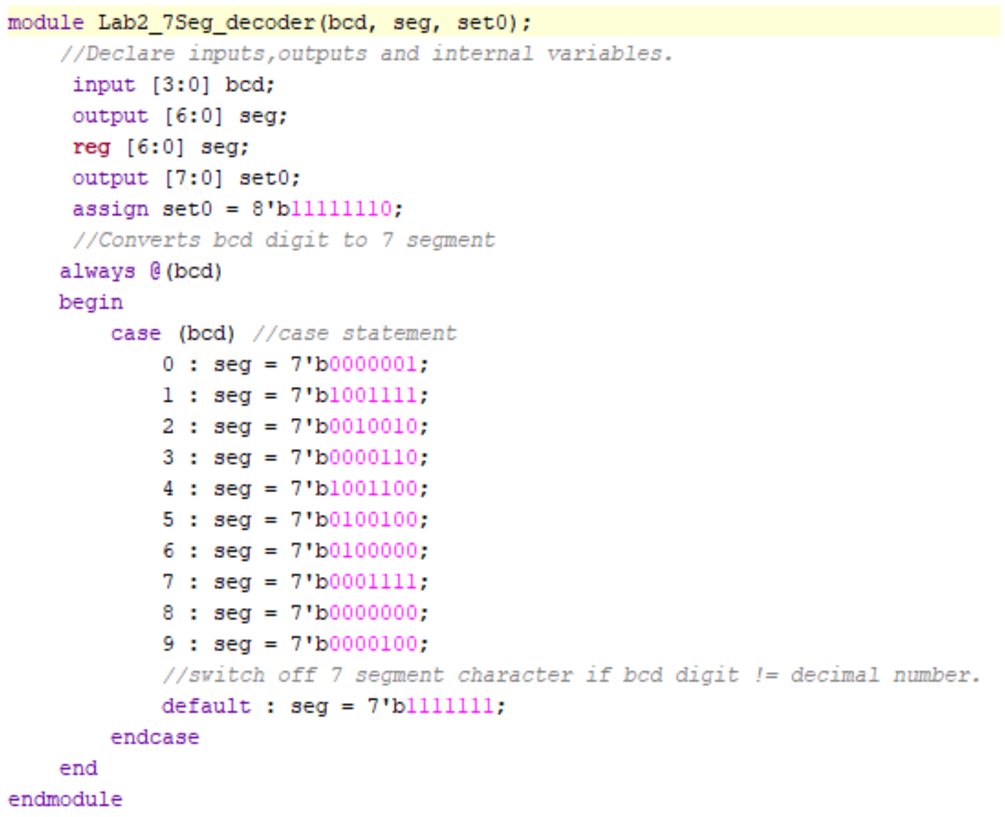
Figure 3

Task 4 is where the lab started to get more difficult, and also started to really get into the adders of the lab. This task was to create a 1 bit full adder first, then simulate it. After the simulation goes well, we are tasked to create a 4 bit ripple carry adder in the same task. This was difficult to build off of for me, so I commented out whichever part I was not currently using. This made the overall construction much easier since it essentially separated the two into two different parts.

Task 5 and 6 were quite similar in the fact that both were simply building off of the circuit in task 4. Task 5 asked to use the same setup as task 4, but insert the output into the 7 segment decoder to display the output. Then task 6 asked to build off task 4 and turn it into a look-ahead adder. This is done by incorporating the carry-look-ahead function.

Detailed Design:

Task 1 was asking to simply set up the 7 segment decoder function and be able to put an input and see it on the 7 segment decoder. The code for this task is shown in Figure 4.

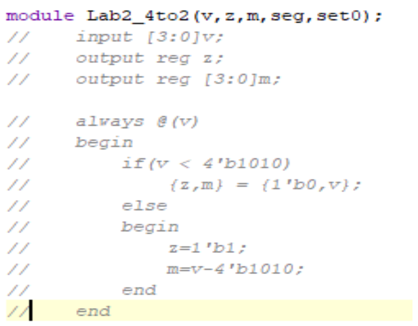


```
module Lab2_7Seg_decoder(bcd, seg, set0);
    //Declare inputs,outputs and internal variables.
     input [3:0] bcd;
     output [6:0] seg;
     reg [6:0] seg;
     output [7:0] set0;
     assign set0 = 8'b11111110;
     //Converts bcd digit to 7 segment
    always @(bcd)
    begin
        case (bcd) //case statement
            0 : seg = 7'b0000001;
            1 : seg = 7'b1001111;
            2 : seg = 7'b0010010;
            3 : seg = 7'b0000110;
            4 : seg = 7'b1001100;
            5 : seg = 7'b0100100;
            6 : seg = 7'b0100000;
            7 : seg = 7'b0001111;
            8 : seg = 7'b0000000;
            9 : seg = 7'b0000100;
            //switch off 7 segment character if bcd digit != decimal number.
            default : seg = 7'b1111111;
        endcase
    end
endmodule
```

Figure 4

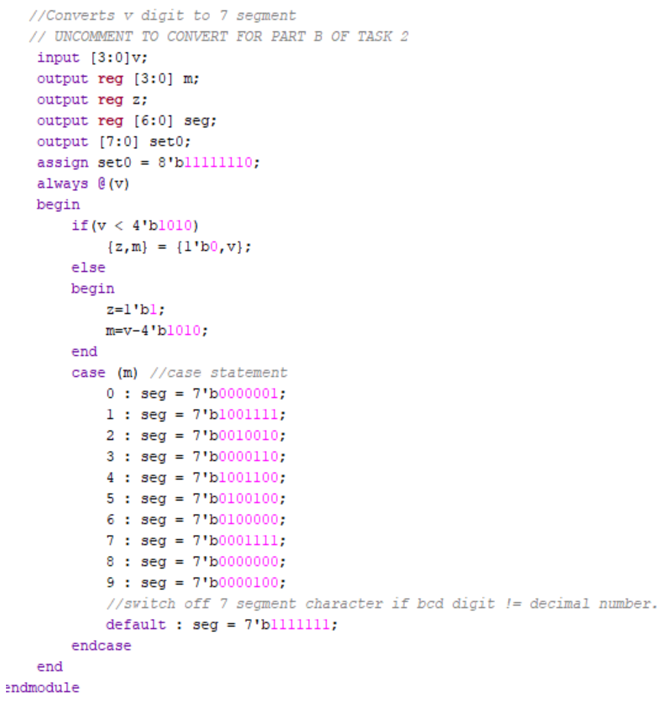
This function reads in whatever the input value is and goes through a switch case. Then depending on what the input is, assign a 7 bit code to the output register. These 7 bits are put together by studying the diagram shown in Figure 1, and making the LED's light up according to the diagram and what the current input is. It also states that the default bits are to turn off the display.

Task 2 was taking the input and displaying it as a 2 bit decimal number. The first part of this task is to run it through a simulation. This part is to just to check that the output is equal to z in the tens place and m in the ones. This is shown in Figure 5. Then the second part of this task is to take these outputs and put them into the decoder. Since the 7 segment display only shows one number, this is done by checking to see if the input is greater than 10, if it is, then set an LED on to show it in more than ten, and display the value in the ones place on the 7 segment decoder. This is shown in the code in Figure 6.



```
module Lab2_4to2(v,z,m,seg,set0);
//    input [3:0]v;
//    output reg z;
//    output reg [3:0]m;

//    always @(v)
//    begin
//        if(v < 4'b1010)
//            {z,m} = {1'b0,v};
//        else
//        begin
//            z=1'b1;
//            m=v-4'b1010;
//        end
//    end
```

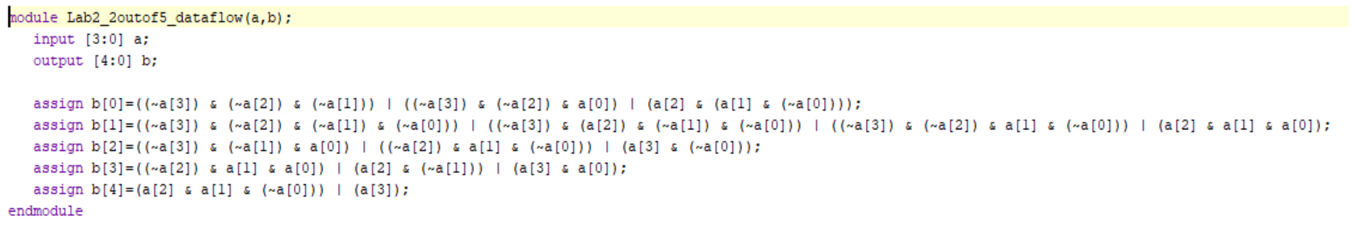
Figure 5



```
//Converts v digit to 7 segment
// UNCOMMENT TO CONVERT FOR PART B OF TASK 2
 input [3:0]v;
 output reg [3:0] m;
 output reg z;
 output reg [6:0] seg;
 output [7:0] set0;
 assign set0 = 8'b11111110;
 always @(v)
 begin
     if(v < 4'b1010)
         {z,m} = {1'b0,v};
     else
     begin
         z=1'b1;
         m=v-4'b1010;
     end
     case (m) //case statement
         0 : seg = 7'b0000001;
         1 : seg = 7'b1001111;
         2 : seg = 7'b0010010;
         3 : seg = 7'b0000110;
         4 : seg = 7'b1001100;
         5 : seg = 7'b0100100;
         6 : seg = 7'b0100000;
         7 : seg = 7'b0001111;
         8 : seg = 7'b0000000;
         9 : seg = 7'b0000100;
         //switch off 7 segment character if bcd digit != decimal number.
         default : seg = 7'b1111111;
     endcase
 end
endmodule
```

Figure 6

Task 3 was about taking an input and determining a very specific output. This is done by using the 2-out-of-5 binary code method. This simply takes a 4 bit binary decimal input, and converting it to get the 2-out-of-5 code. This is done easily by viewing the table given and setting each output to either 0 or 1 depending on what the current input is. This is shown in the code in Figure 7.

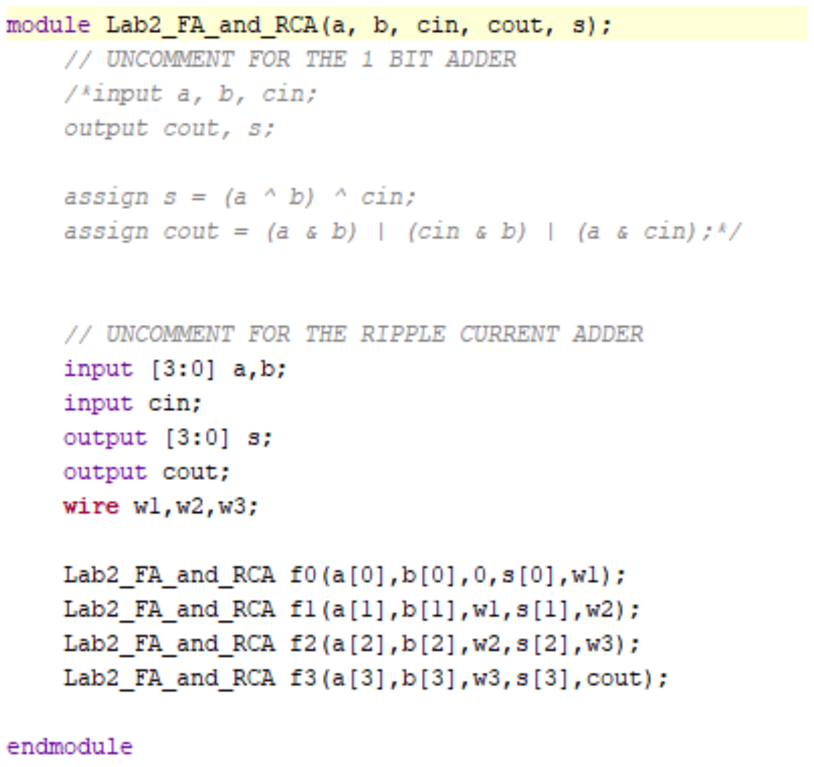


```
module Lab2_2outof5_dataflow(a,b);
   input [3:0] a;
   output [4:0] b;

   assign b[0]=((~a[3]) & (~a[2]) & (~a[1])) | ((~a[3]) & (~a[2]) & a[0]) | (a[2] & (a[1] & (~a[0])));
   assign b[1]=((~a[3]) & (~a[2]) & (~a[1]) & (~a[0])) | ((~a[3]) & (a[2]) & (~a[1]) & (~a[0])) | ((~a[3]) & (~a[2]) & a[1] & (~a[0])) | (a[2] & a[1] & a[0]);
   assign b[2]=((~a[3]) & (~a[1]) & a[0]) | ((~a[2]) & a[1] & (~a[0])) | (a[3] & (~a[0]));
   assign b[3]=((~a[2]) & a[1] & a[0]) | (a[2] & (~a[1])) | (a[3] & a[0]);
   assign b[4]=(a[2] & a[1] & (~a[0])) | (a[3]);
endmodule
```

Figure 7

This task was not too difficult and only takes time. By viewing the truth table given in the Lab specs, we can go through each of the 5 bits and set them to one only when the value is equal to that in the truth table. We then assign each bit to the corresponding decimal numbers.

Task 4 began to use adders in this lab and starts off with a simple full carry adder in the first part, and then a ripple carry adder in the second part. The code for both parts are shown in Figure 8.

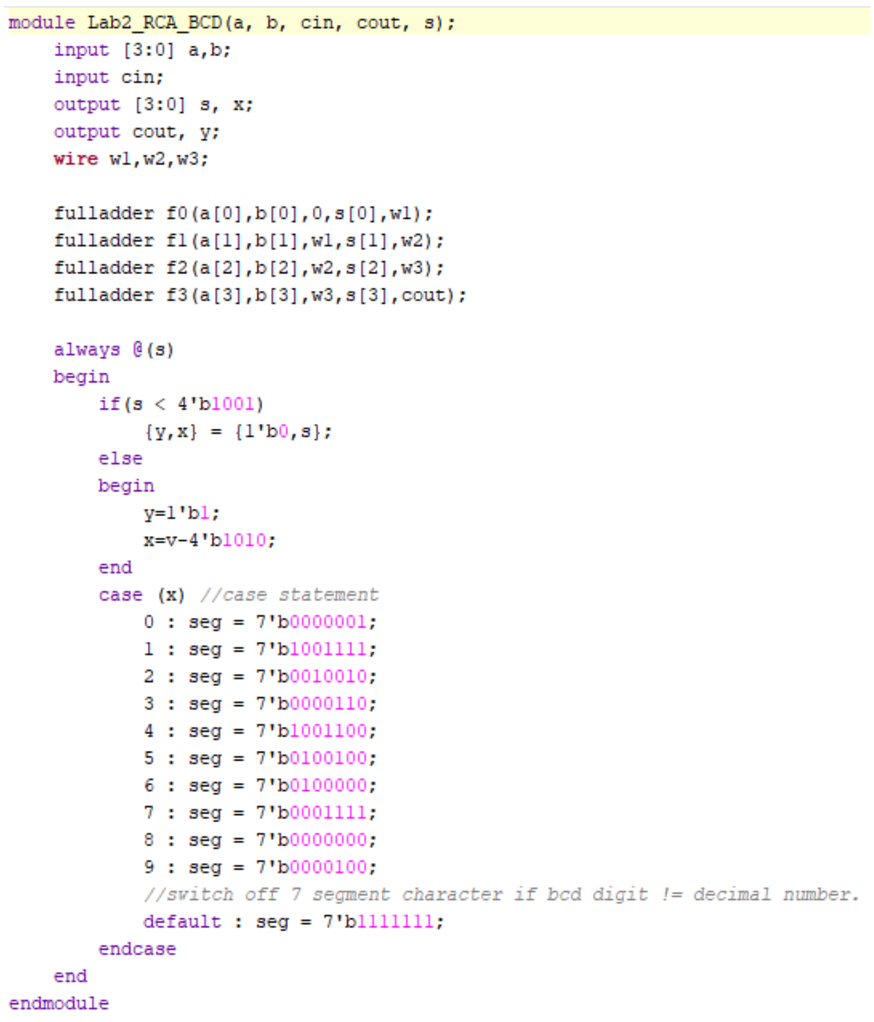


```
module Lab2_FA_and_RCA(a, b, cin, cout, s);
    // UNCOMMENT FOR THE 1 BIT ADDER
    /*input a, b, cin;
    output cout, s;

    assign s = (a ^ b) ^ cin;
    assign cout = (a & b) | (cin & b) | (a & cin);*/


    // UNCOMMENT FOR THE RIPPLE CURRENT ADDER
    input [3:0] a,b;
    input cin;
    output [3:0] s;
    output cout;
    wire w1,w2,w3;

    Lab2_FA_and_RCA f0(a[0],b[0],0,s[0],w1);
    Lab2_FA_and_RCA f1(a[1],b[1],w1,s[1],w2);
    Lab2_FA_and_RCA f2(a[2],b[2],w2,s[2],w3);
    Lab2_FA_and_RCA f3(a[3],b[3],w3,s[3],cout);

endmodule
```

Figure 8

The code for the first part of this lab is shown commented out, this is because when trying to separate the two parts of this task I was getting very confused, and decided the best way to complete it would be to try to separate the two parts almost like they are two separate tasks. The first part (the commented out section) simply takes the input and cin, and adds them together to get the output and cout. The second part creates the 4 bits instances and adds them together by using itself on different bits.

Task 5 builds off of task 4, but adds the 7 segment decoder. This is useful for a more visual component to be able to see what exactly is going on with the adder. The code is shown in Figure 9.

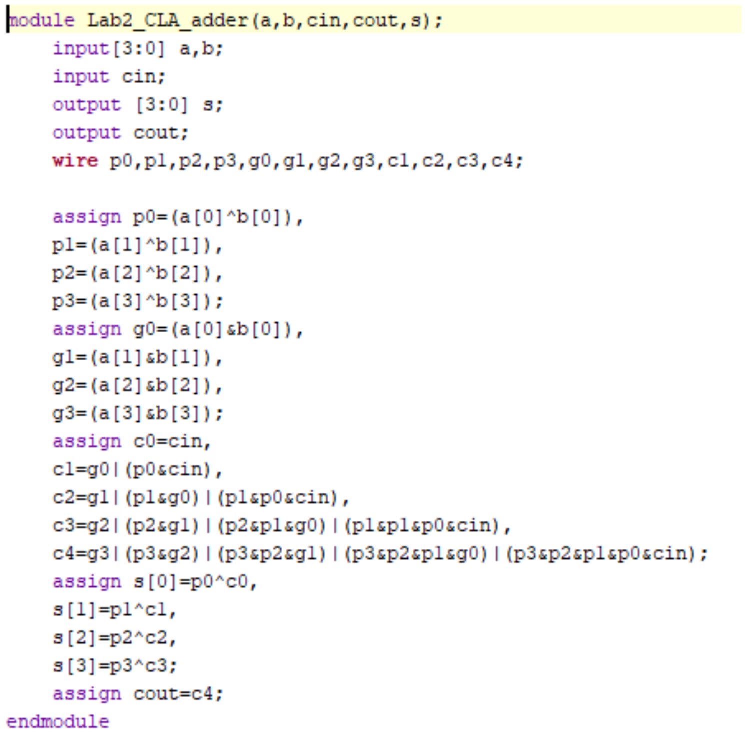


```
module Lab2_RCA_BCD(a, b, cin, cout, s);
    input [3:0] a,b;
    input cin;
    output [3:0] s, x;
    output cout, y;
    wire w1,w2,w3;

    fulladder f0(a[0],b[0],0,s[0],w1);
    fulladder f1(a[1],b[1],w1,s[1],w2);
    fulladder f2(a[2],b[2],w2,s[2],w3);
    fulladder f3(a[3],b[3],w3,s[3],cout);

    always @(s)
    begin
        if(s < 4'b1001)
            {y,x} = {1'b0,s};
        else
        begin
            y=1'b1;
            x=v-4'b1010;
        end
        case (x) //case statement
            0 : seg = 7'b0000001;
            1 : seg = 7'b1001111;
            2 : seg = 7'b0010010;
            3 : seg = 7'b0000110;
            4 : seg = 7'b1001100;
            5 : seg = 7'b0100100;
            6 : seg = 7'b0100000;
            7 : seg = 7'b0001111;
            8 : seg = 7'b0000000;
            9 : seg = 7'b0000100;
            //switch off 7 segment character if bcd digit != decimal number.
            default : seg = 7'b1111111;
        endcase
    end
endmodule
```

Figure 9

This code looks very familiar as it is nothing but combining tasks 4 and 1. In the top there is task 4 adder being used to get the output, then it gets moved to the bottom part which is task 1, just taking the output of the top as the input, and displaying it on the 7 segment decoder.

Task 6 also has the same idea as task 5, which is to build off task 4 adders. Instead of displaying the output differently, build off the adder to make a carry-look-ahead adder. The code for this is shown below in Figure 10.



```
module Lab2_CLA_adder(a,b,cin,cout,s);
    input[3:0] a,b;
    input cin;
    output [3:0] s;
    output cout;
    wire p0,p1,p2,p3,g0,g1,g2,g3,c1,c2,c3,c4;

    assign p0=(a[0]^b[0]),
    p1=(a[1]^b[1]),
    p2=(a[2]^b[2]),
    p3=(a[3]^b[3]);
    assign g0=(a[0]&b[0]),
    g1=(a[1]&b[1]),
    g2=(a[2]&b[2]),
    g3=(a[3]&b[3]);
    assign c0=cin,
    c1=g0|(p0&cin),
    c2=g1|(p1&g0)|(p1&p0&cin),
    c3=g2|(p2&g1)|(p2&p1&g0)|(p1&p1&p0&cin),
    c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&cin);
    assign s[0]=p0^c0,
    s[1]=p1^c1,
    s[2]=p2^c2,
    s[3]=p3^c3;
    assign cout=c4;
endmodule
```

Figure 10

The look ahead adder is different then the ripple carry mostly because the input should only affect that cin before the computation begins. This is different as the ripple carry adder than cin is pushed through like a normal input and then used in the adder. In task 6 we assign values directly to the cin to make whether or not they are a cin or not before the computation of the adder begins.

Verification:

Each task is verified differently. The first task is verified by looking at the decoder display on the board. By looking at the display we see that the input bits directly match the number shown on the display, thus task 1 is correct.

Task 2 is verified by checking the simulation results ran in the first part of the task. We should see in the results that the four bits should be equal to the ones place of the decimal digit shown, and turn on the tens place only when greater than 9. The results are shown in Figure 11.

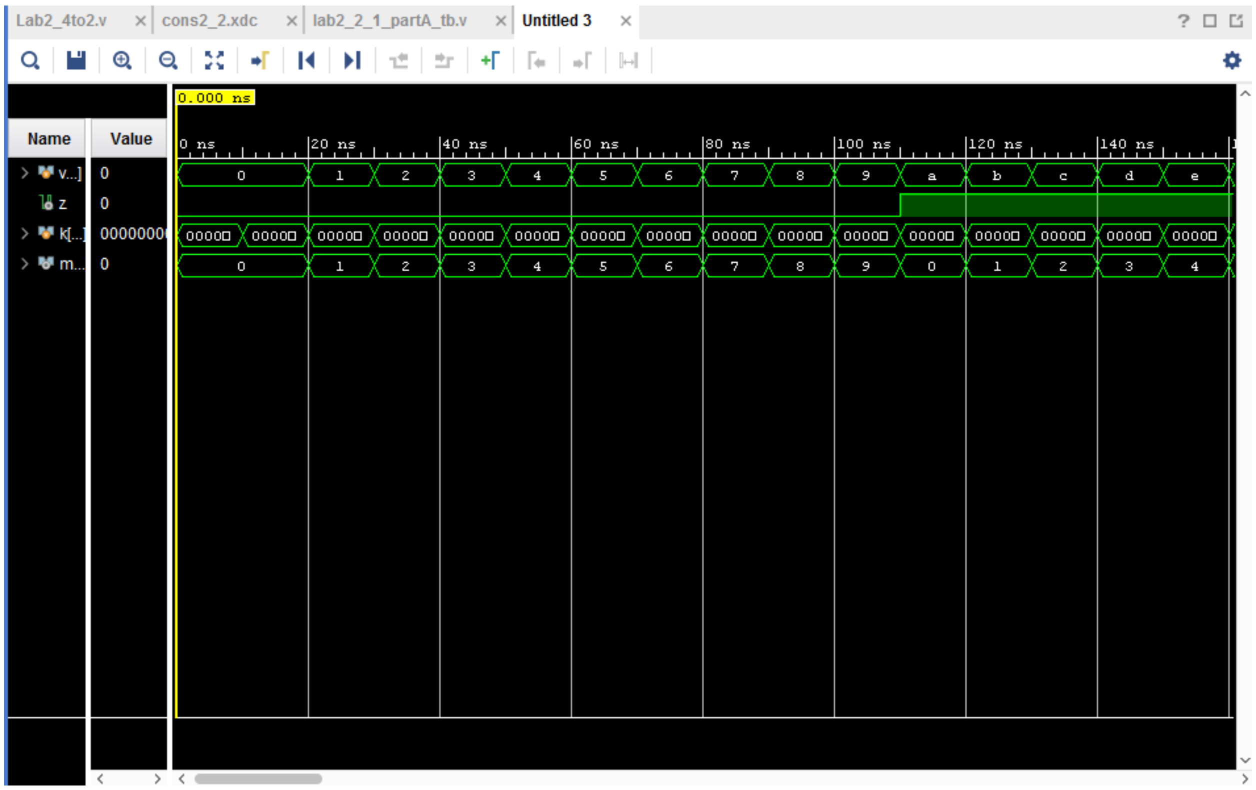


Figure 11

Task 3 is easy to test since the switches matched to the input bits should be set and view the output LED's to see if the program matches the truth table given. This is an easy way to test this task since the truth table was given in the Lab description. The truth table is shown in Figure 3 above.

Task 4 also used a simulation to test whether the code works or not. By checking the simulation we see that the input bits going through the adder should equal the output. This verifies that the code is able to add the inputs together through a simple adder. The simulation results are shown in Figure 12.

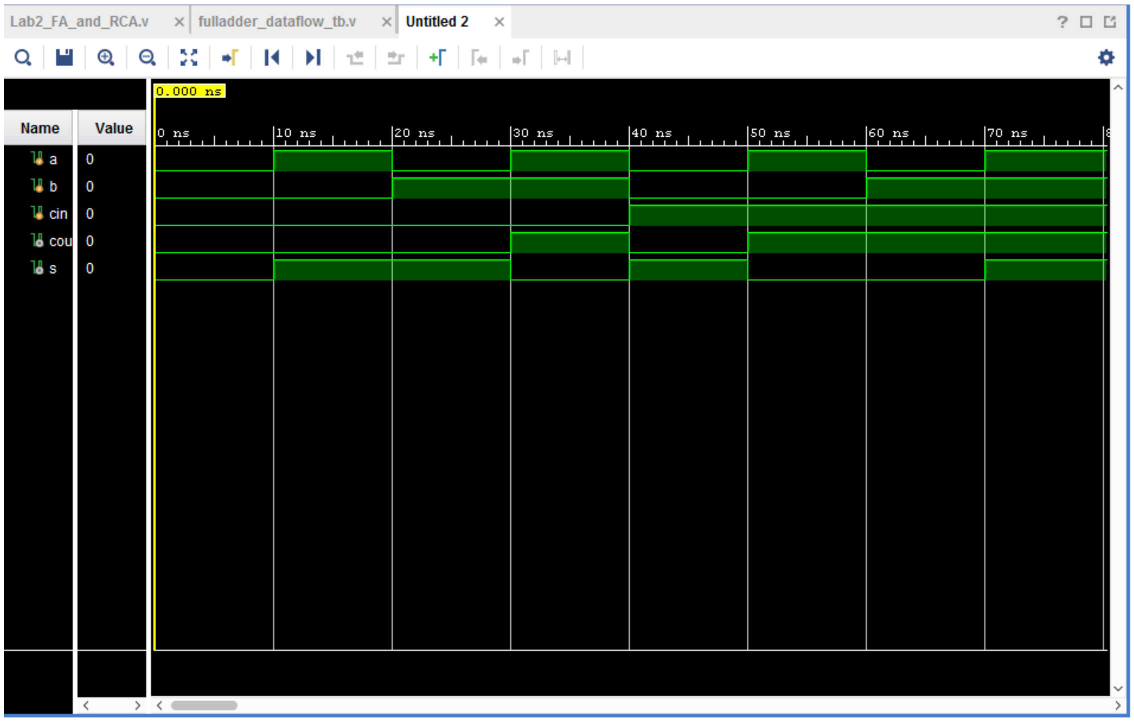
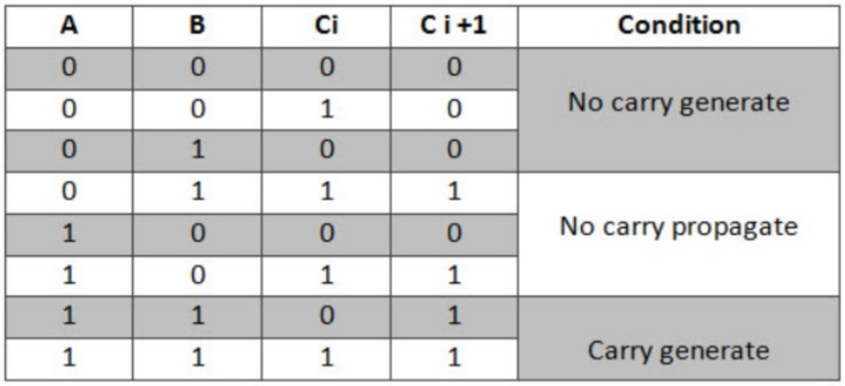


Figure 12

Task 5 is a simple to verify task since it is put visually on the board. After modifying the previous task to output to a BCD 7 segment decoder, all that we must do to verify is add simple bits through the adder and view the results on the display of the decoder. After viewing the display is correct, we verify that task 5 works.

Task 6 is harder to verify since there is no display except for the LED's on the board. However, another way to check the results is to compare the generate/propagate to a look ahead adder truth table. After viewing the results of the look-ahead adder and comparing the generate/propagate to the truth table, we can show that the final task has worked. The truth table is shown in Figure 13



A	B	Ci	C i +1	Condition
0	0	0	0	No carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

Figure 13

Conclusion:

What I learned in this lab is by building off previous tasks, it makes the overall buildup of codes easier. With having similar tasks in each section, we can quickly derive and add up the different functions throughout the lab. I also learned how to put together adders, ripple carry adders, and look-ahead adders. I also learned how to take a 3-bit number and translate that to show on the 7 segment decoder display. I did not get to finish all these tasks in lab mainly due to a small error later fixed in task 3 of the lab. However, after passing this I made it to the 4th task where I got the simulation done before running out of time. Next lab I will try to work faster and possibly find the small mistakes quicker to avoid having this problem of running out of time. Next lab will go better since I can be more prepared in each task and finish quickly.