Gabriel Emerson
ELEC 4200 – Lab 3
09/09/21

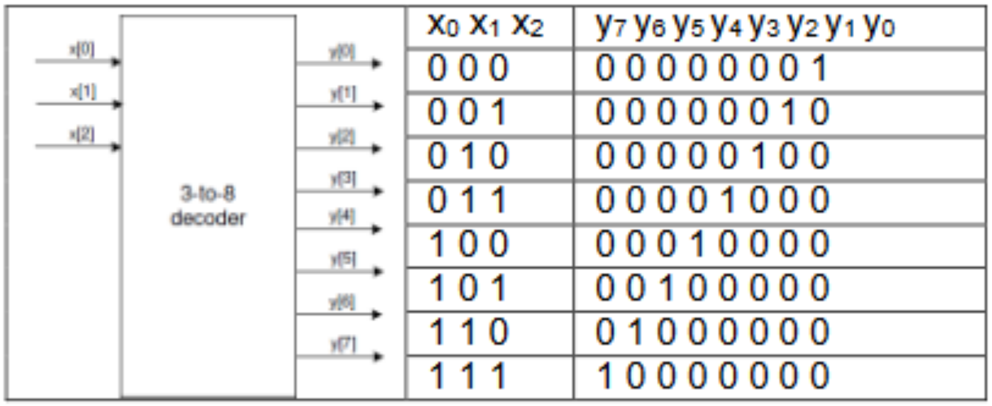
Lab 3: Multi-Output Circuits: Encoders, Decoders, and Memories

Goals:

The goal for this lab is to become familiar with multi output circuits. This includes Encoders, Decoders, and the use of memory files. This lab begins by using standard multi output encoders, decoders, and works to the creation of ROM files and how they are used to get multiple outputs.

Design Process:

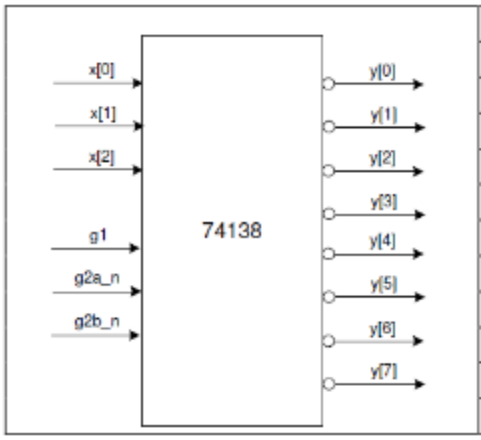
Getting started for this lab starts with creating a truth table for the task 1 circuit, a 3 to 8 line decoder. The first task only asks for the creation of this decoder, which can be easily implemented using dataflow structure in Verilog. The truth table for task 1 is shown in Figure 1.



x_0 x_1 x_2	y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0
0 0 0	0 0 0 0 0 0 0 1
0 0 1	0 0 0 0 0 0 1 0
0 1 0	0 0 0 0 0 1 0 0
0 1 1	0 0 0 0 1 0 0 0
1 0 0	0 0 0 1 0 0 0 0
1 0 1	0 0 1 0 0 0 0 0
1 1 0	0 1 0 0 0 0 0 0
1 1 1	1 0 0 0 0 0 0 0

Figure 1

Task 2 is similar to the creation of the 3 to 8 decoder in the way there are multiple inputs and multiple outputs. The main difference between the creation of this IC and a standard decoder, is that there are more inputs for the IC and the logic of this IC defines certain inputs to ‘x’ or ‘don’t care’. This makes the creation of the IC slightly more difficult, but still is done in a similar way to the decoder. The IC should be made according to the truth table given. This table is shown below in Figure 2.



g_1 g_{2a_n} g_{2b_n}	x_0 x_1 x_2	y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
0 x x	x x x	1 1 1 1 1 1 1 1
x 1 x	x x x	1 1 1 1 1 1 1 1
x x 1	x x x	1 1 1 1 1 1 1 1
1 0 0	0 0 0	0 1 1 1 1 1 1 1
1 0 0	0 0 1	1 0 1 1 1 1 1 1
1 0 0	0 1 0	1 1 0 1 1 1 1 1
1 0 0	0 1 1	1 1 1 0 1 1 1 1
1 0 0	1 0 0	1 1 1 1 0 1 1 1
1 0 0	1 0 1	1 1 1 1 1 0 1 1
1 0 0	1 1 0	1 1 1 1 1 1 0 1
1 0 0	1 1 1	1 1 1 1 1 1 1 0

Figure 2

Task 3 is still similar to task 1 and 2 in that it is a creation of multiple inputs and outputs. This task could be considered as the opposite of task 1, since this is the creation of the 8 to 3 encoder. While a standard 8 to 3 encoder could be considered opposite, we are created something a little more complex called the 'priority' 8 to 3 encoder. This form of encoder introduces more inputs and outputs to add a little more complexity to the logic of the circuit. The circuit should match the truth table given, which is shown in Figure 3.

Inputs									Outputs				
E1	0	1	2	3	4	5	6	7	y2	y1	y0	GS	E0
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	X	X	X	X	X	X	X	0	0	0	0	0	1
0	X	X	X	X	X	X	0	1	0	0	1	0	1
0	X	X	X	X	X	0	1	1	0	1	0	0	1
0	X	X	X	X	0	1	1	1	0	1	1	0	1
0	X	X	X	0	1	1	1	1	1	0	0	0	1
0	X	X	0	1	1	1	1	1	1	0	1	0	1
0	X	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1

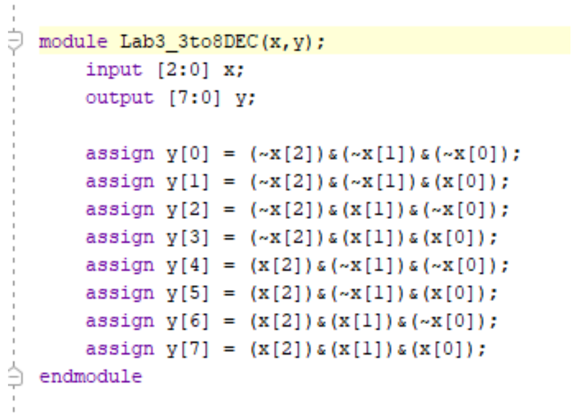
Figure 3

Task 4 is different from the previous 3 tasks. Where previous tasks are to create an output that is generated from the results of the current input, task 4 is not like this. Instead, we are tasked with generating the output results in a ROM memory file, then the input will point to a specific spot in this file and that spot will contain the output. For this task, we create a comparator circuit and store three bits and the one that is set high represents whether the two input bits are: greater than, less than, or equal to. This is in form {Lt, Gt, Eq}.

Task 5 is similar to task 4 and also uses a previously made ROM memory file that stores the results of the circuit, and the input points to which output is used. This task instead of making a comparator, creates a decimal multiplication circuit. This means that the circuit will take two 2-bit inputs, then use those two inputs to point to a specific spot in the ROM file, then output whatever is currently in that spot.

Detailed Design:

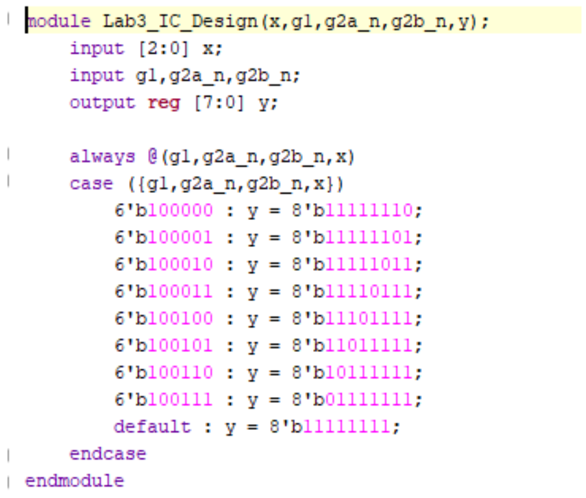
Task 1 was asking to set up a 3 to 8 decoder using dataflow structuring. This is done by simply setting up the values for each bit and setting when each bit is set to 1. This is shown in the code in Figure 4.



```
module Lab3_3to8DEC(x,y);
    input [2:0] x;
    output [7:0] y;

    assign y[0] = (~x[2])&(~x[1])&(~x[0]);
    assign y[1] = (~x[2])&(~x[1])&(x[0]);
    assign y[2] = (~x[2])&(x[1])&(~x[0]);
    assign y[3] = (~x[2])&(x[1])&(x[0]);
    assign y[4] = (x[2])&(~x[1])&(~x[0]);
    assign y[5] = (x[2])&(~x[1])&(x[0]);
    assign y[6] = (x[2])&(x[1])&(~x[0]);
    assign y[7] = (x[2])&(x[1])&(x[0]);
endmodule
```

Figure 4

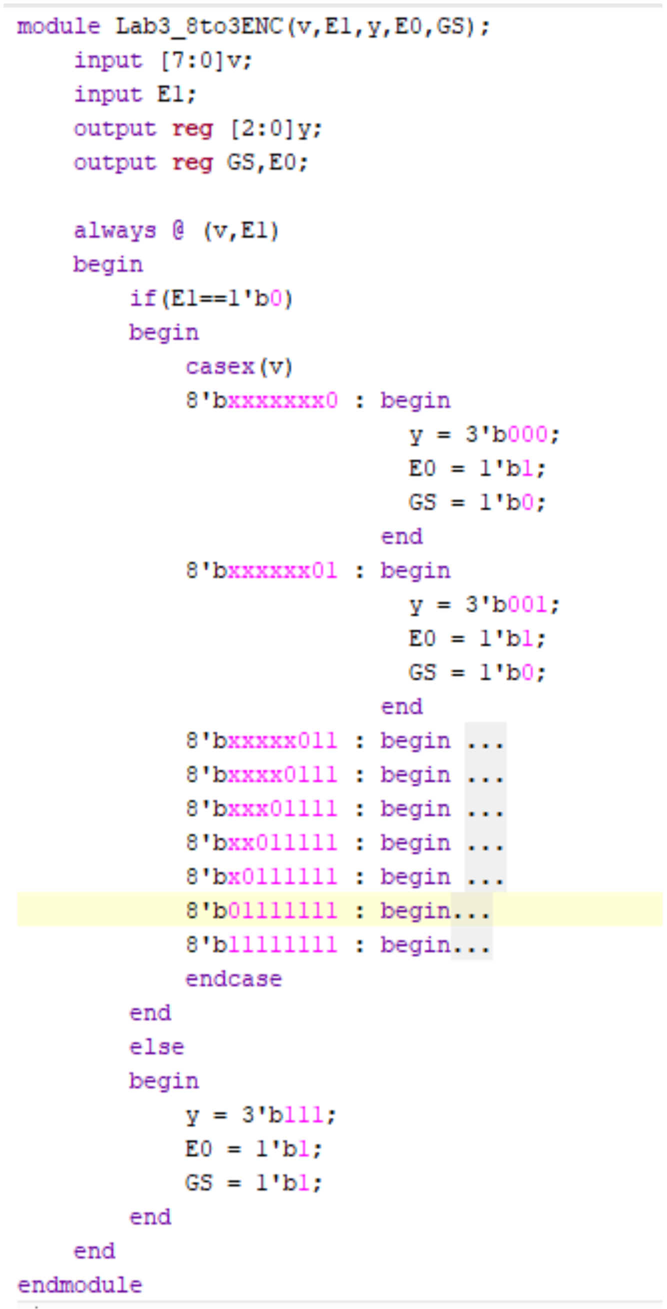
Task 2 was about designing your own version of the IC74138 chip using dataflow structuring. This is similar to Task 1 in just setting up the bits one by one and setting when they will be equal to 0. However, the amount of inputs used in this task is more than the previous, and setting the output bits is a lot different from a decoder. The code for this task is shown in Figure 5.



```
module Lab3_IC_Design(x,g1,g2a_n,g2b_n,y);
    input [2:0] x;
    input g1,g2a_n,g2b_n;
    output reg [7:0] y;

    always @(g1,g2a_n,g2b_n,x)
    case ({g1,g2a_n,g2b_n,x})
        6'b100000 : y = 8'b11111110;
        6'b100001 : y = 8'b11111101;
        6'b100010 : y = 8'b11111011;
        6'b100011 : y = 8'b11110111;
        6'b100100 : y = 8'b11101111;
        6'b100101 : y = 8'b11011111;
        6'b100110 : y = 8'b10111111;
        6'b100111 : y = 8'b01111111;
        default : y = 8'b11111111;
    endcase
endmodule
```

Figure 5

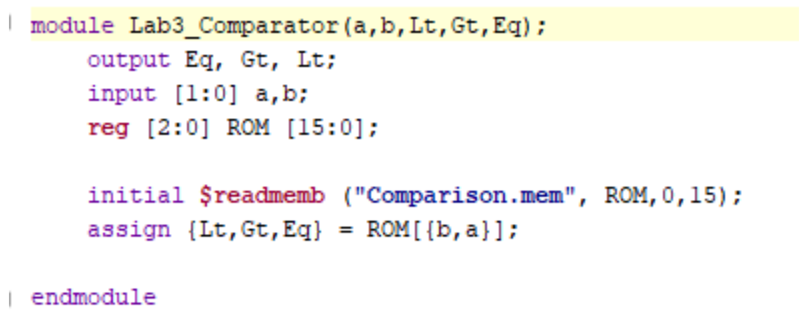
Task 3 is almost the opposite implementation of task 1, but instead of a standard 8 to 3 encoder we add a little complexity and create a priority encoder. This is done by adding an input, and a couple new outputs. This is done using behavioral structures which makes the coding of this step quite simple. However, the code for this is quite long, so it has been collapsed in the switch cases to fit in a screenshot. The first couple switch cases are shown in full to show how each case is done inside of the collapsed part. The code for task 3 is shown in Figure 6.



```
module Lab3_8to3ENC(v,E1,y,E0,GS);
    input [7:0]v;
    input E1;
    output reg [2:0]y;
    output reg GS,E0;

    always @ (v,E1)
    begin
        if(E1==1'b0)
        begin
            casex(v)
            8'bxxxxxxx0 : begin
                              y = 3'b000;
                              E0 = 1'b1;
                              GS = 1'b0;
                          end
            8'bxxxxxx01 : begin
                              y = 3'b001;
                              E0 = 1'b1;
                              GS = 1'b0;
                          end
            8'bxxxxx011 : begin ...
            8'bxxxx0111 : begin ...
            8'bxxx01111 : begin ...
            8'bxx011111 : begin ...
            8'bx0111111 : begin ...
            8'b01111111 : begin...
            8'b11111111 : begin...
            endcase
        end
        else
        begin
            y = 3'b111;
            E0 = 1'b1;
            GS = 1'b1;
        end
    end
endmodule
```

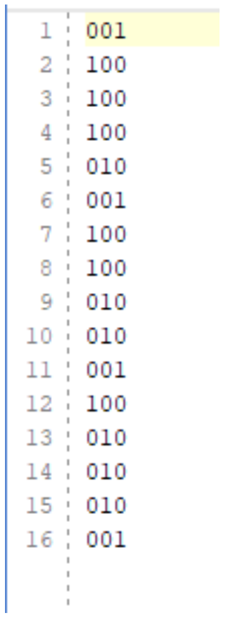
Figure 6

Task 4 and 5 are done in very similar ways. Task 4 is to create a circuit, with a ROM memory file, that acts as a comparator, taking in a and b as inputs, then displaying on an LED if it is Greater than, Less than, or Equal to. This is done by importing the memory file into the program, then using the input as an address for the file. Then the value in that address is the resulting output. The code and the memory file can be seen in Figures 7 & 8 respectively. Task 5 was essentially the same, except the values in the ROM file change since it is changed to be a multiplication circuit. This means we load the memory file with the outputs of all different combinations of a*b. The code and the memory file for task 5 is shown in Figures 9 & 10.



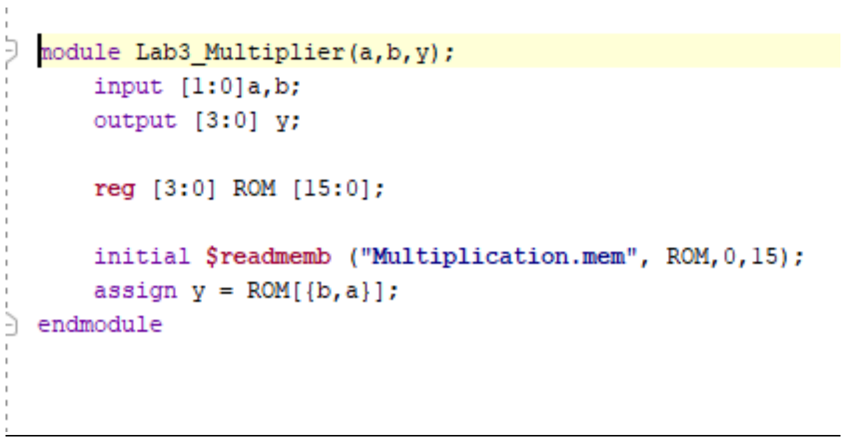
```
module Lab3_Comparator(a,b,Lt,Gt,Eq);
    output Eq, Gt, Lt;
    input [1:0] a,b;
    reg [2:0] ROM [15:0];

    initial $readmemb ("Comparison.mem", ROM,0,15);
    assign {Lt,Gt,Eq} = ROM[{b,a}];

endmodule
```

Figure 7



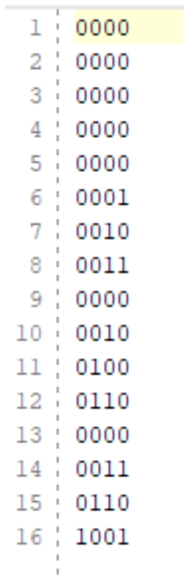
```
001
100
100
100
010
001
100
100
010
010
001
100
010
010
010
001
```

Figure 8



```
module Lab3_Multiplier(a,b,y);
    input [1:0]a,b;
    output [3:0] y;

    reg [3:0] ROM [15:0];

    initial $readmemb ("Multiplication.mem", ROM,0,15);
    assign y = ROM[{b,a}];
endmodule
```

Figure 9



```
0000
0000
0000
0000
0000
0001
0010
0011
0000
0010
0100
0110
0000
0011
0110
1001
```

Figure 10

Verification:

Each task is verified differently. The first task had two separate parts. Part one of the first task involves adding a testbench simulation file to the project and running a simulation for the decoder. Then by viewing the results and expanding outputs we can see the staircase of bits formed as shown in the truth table in Figure 1. The simulation output can be seen in Figure 11. The second part of task 1 involves uploading to the Nexys4 DDR board and visually verifying the results. Since my results matched the truth table shown in Figure 1, this verifies the functionality of task 1.

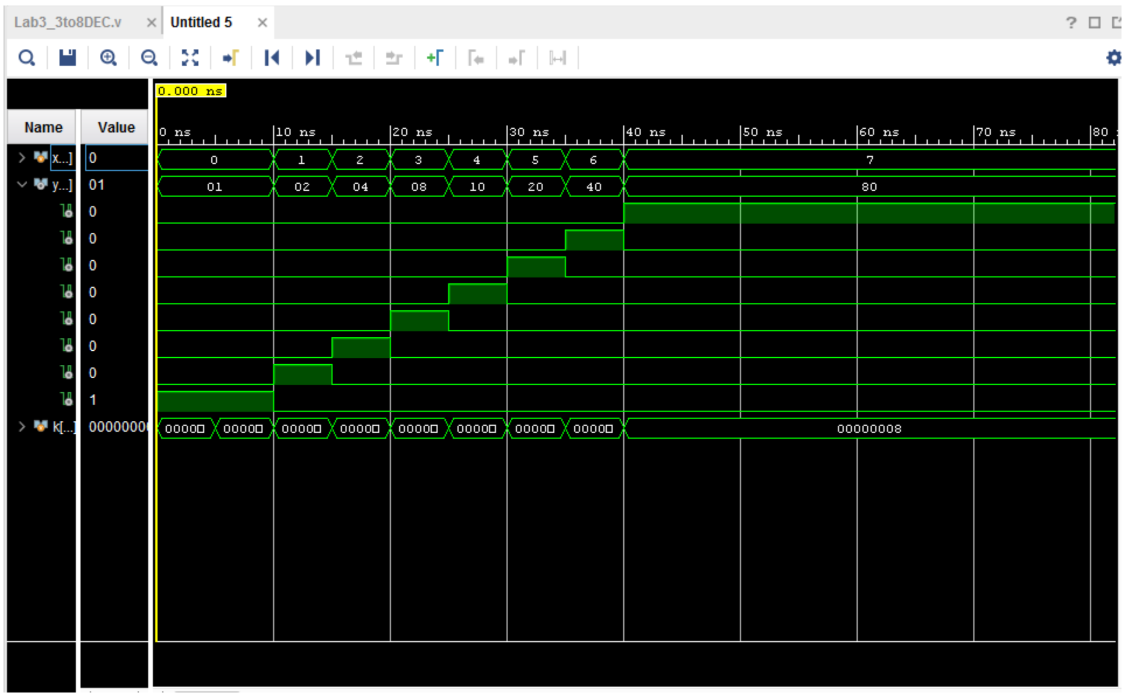


Figure 11

Task 2 has the same verification methods as task 1 did. For the first part of task 2 we use the given testbench to simulate the program of the IC. This results in a strange inverted staircase, which is shown in the simulation results in Figure 12. The second part of this task involves uploading to the Nexys4 DDR board and visually verifying the results. Then we check, and since the output matched the output of the truth table in Figure 2, this verifies the results of task 2.

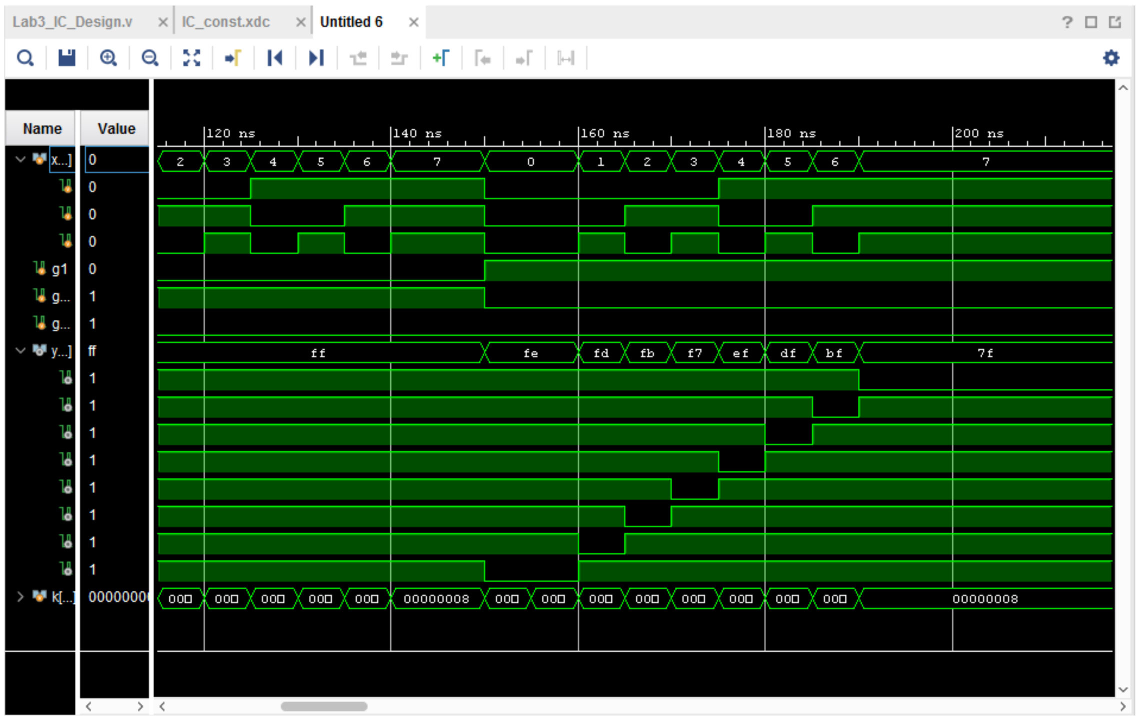


Figure 12

Task 3 does not require a simulation to verify its results. Instead, we simply upload our program to the Nexys4 DDR board and visually verifying the results. By matching our results with that of the truth table shown in Figure 3, we verify the functionality of task 3.

Task 4 and 5 both do not require a simulation testbench to test results. We instead just upload our program to the Nexys4 DDR board and visually verifying the results. The outputs of the ROM memory files should be easy to check by simply either comparing a and b (for task 4) or multiplying a and b (for task 5) on each spot and then checking if the result in the memory file is the same. Since the upload visually passes the tests given, this verifies the results for tasks 4 and 5.

Conclusion:

What I learned in this lab is building decoders/encoders or any other multi-input and output circuit can be easily done using dataflow structuring, or if need be by behavioral switch cases. I also learned the power and simplicity of using ROM memory files when creating different circuits. By doing the calculations before the implementation, and updating the results to a memory file, we can easily access the file and pull out the results.These tasks were very different from last lab, since last time each step of the lab built off each other, but this time each task was quite different except a few similarities. I did not have any great problems with these tasks, but did have a small error in task 1 where my outputs were not making a staircase but instead were all over the place. After a few minutes of looking it over I simply found my inputs bits were backwards which was making 001 look like 100. After swapping my values back to their correct place, task 1 worked well.