ELEC-4200

Digital System Design

FROM: Jacob Howard

TO: Prof. Ujjwal Guin

DUE DATE: 2/11/21

Lab 3

Introduction

The goal of this lab was to familiarize ourselves with encoders, decoders, and memory. The main objectives of this lab were to design multi-output decoder circuits, encoders, and use read-only memories (ROM) for tasks.

Task 1

In Task 1, we were asked to design a 3-to-8 line decoder using dataflow modeling constructs. Once we completed the design, we verified the code with the testbench file provided to us. After running a successful testbench, we programed our board to run and test the code. My code performed as intended and can be shown in *Code 1* below. Simulation for part 1 is also shown below in *Simulation 1*.

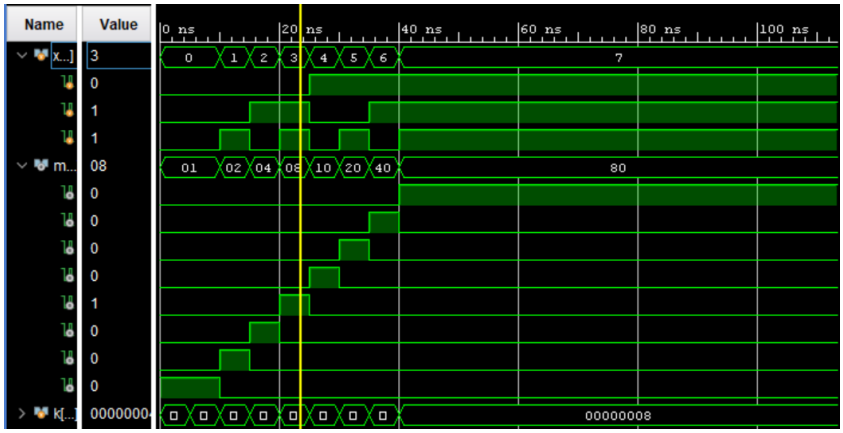


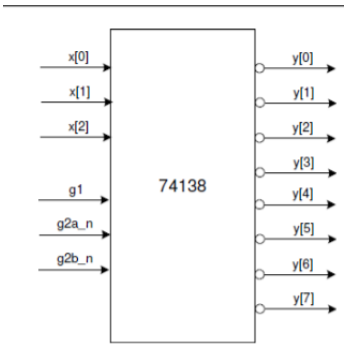
Figure 1

```
module Task1(
    input [2:0] x,//3 bit input
    output [7:0] m //8 bit output
    );

    assign m[0] = (~(x[2] | x[1] | x[0])); //m[0] is 1
when all bits of x are == 0
    assign m[1] = (~(x[2] | x[1]) & x[0]); //~a & ~ b &
c = m[1]
    assign m[2] = (~(x[2] | x[0]) & x[1]); //~a & b & ~c
    assign m[3] = (~x[2] & x[1] & x[0]); //~a & b & c
    assign m[4] = x[2] & ~(x[1] | x[0]); //a & ~b & ~c
    assign m[5] = x[2] & ~x[1] & x[0]; //a & ~b & c
    assign m[6] = x[2] & x[1] & ~x[0]; // a & b & ~c
    assign m[7] = x[2] & x[1] & x[0]; //a & b & c

endmodule
```

Code 1

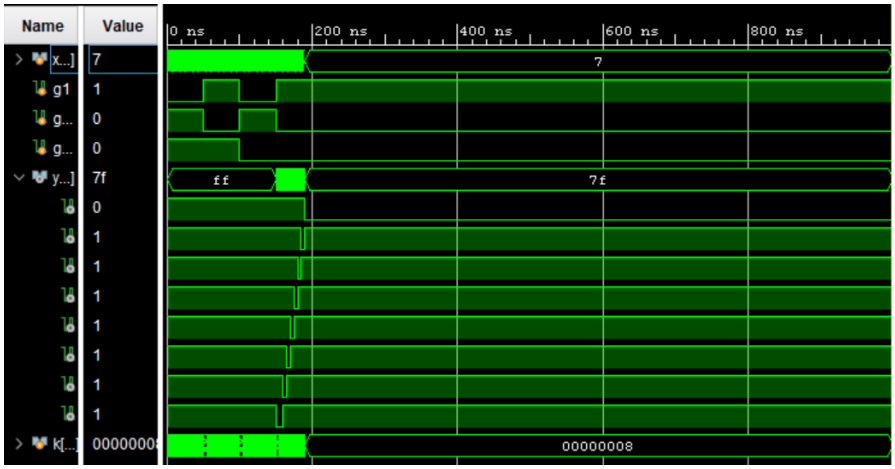
Task 2

In Task 2, we were asked to design an IC, 74138 using dataflow modeling and the decoder you used in Task 1. We were given a truth table to help us with the design. The truth table can be seen below in *Figure 1*. Once we were finished building the design, we were asked to verify it with the provided testbench. The simulation can be seen below in *Simulation 2*. After simulating the design, we downloaded the code to the board and verified it on the board. Everything functioned as intended. The code can be seen below in *Code 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 1



Simulation 2

```
module Task2(
    input [2:0] x,
    input g1,
    input g2a_n,
    input g2b_n,
    output [7:0] y
    );

    assign y[0] = !((g1 & ~g2a_n & ~g2b_n) & (~(x[2] |
x[1] | x[0])));
    assign y[1] = !((g1 & ~g2a_n & ~g2b_n) & (~(x[2] |
x[1]) & x[0]));
    assign y[2] = !((g1 & ~g2a_n & ~g2b_n) & (~(x[2] |
x[0]) & x[1]));
    assign y[3] = !((g1 & ~g2a_n & ~g2b_n) & (~x[2] &
x[1] & x[0]));
    assign y[4] = !((g1 & ~g2a_n & ~g2b_n) & (x[2] &
~(x[1] | x[0])));
    assign y[5] = !((g1 & ~g2a_n & ~g2b_n) & (x[2] &
~x[1] & x[0]));
    assign y[6] = !((g1 & ~g2a_n & ~g2b_n) & (x[2] & x[1]
& ~x[0]));
    assign y[7] = !((g1 & ~g2a_n & ~g2b_n) & (x[2] & x[1]
& x[0]));


endmodule
```

Code 2

Task 3

In Task 3, we were asked to design an 8-to-3 priority encoder using behavioral modeling. There were no test benches for the rest of the tasks, so once our code was written, we simulated it on the board to verify functionality. A truth table was given to help us and is shown below in *Figure 2*. The code functioned as expected and can be seen in *Code 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 2

```
module Task3(
   input wire [7:0] v,
   input wire E1,
   output reg E0,
   output reg GS,
   output reg [2:0] y
   );
   wire [9:0] S;
   assign S = {E1, v[7:0]};
   //Checks if switch E1 is on
      //If on, all LEDs light up
  always @(S)
   begin
   casex(S)
   9'b1xxxxxxxx : begin y = 3'b111; GS = 1'b1; E0 = 1'b1; end
   9'b011111111 : begin y = 3'b111; GS = 1'b1; E0 = 1'b0; end
   9'b0xxxxxxx0 : begin y = 3'b000; GS = 1'b0; E0 = 1'b1; end
   9'b0xxxxxx01 : begin y = 3'b001; GS = 1'b0; E0 = 1'b1; end
   9'b0xxxxx011 : begin y = 3'b010; GS = 1'b0; E0 = 1'b1; end
   9'b0xxxx0111 : begin y = 3'b011; GS = 1'b0; E0 = 1'b1; end
   9'b0xxx01111 : begin y = 3'b100; GS = 1'b0; E0 = 1'b1; end
   9'b0xx011111 : begin y = 3'b101; GS = 1'b0; E0 = 1'b1; end
   9'b0x0111111 : begin y = 3'b110; GS = 1'b0; E0 = 1'b1; end
   9'b001111111 : begin y = 3'b111; GS = 1'b0; E0 = 1'b1; end
   endcase
   end

endmodule
```

Code 3

Task 4

In Task, we were asked to design a 2-bit comparator that compares two 2-bit numbers and asserts outputs indicating whether the decimal equivalent of word A is less than, greater than, or

equal to that of word B. Then we were asked to create and add a memory file with the desired outputs. The objective of this was for our code to read the memory file from the desired input location and output the correct output with no logic required. Once I figured out how to set up the memory file correctly, the code performed as intended and can be seen in *Code 4* below. Also, the memory file can be seen in *Memory 1* below.

```
module Task4(
    input [1:0] a,
    input [1:0] b,
    output [2:0] lt_gt_or_eq
    );

    wire [3:0] ROM_addr;
    reg [2:0] ROM [15:0];

   assign ROM_addr = {a[1:0], b[1:0]};
   assign lt_gt_or_eq = ROM[ROM_addr];

   begin
   initial $readmemb("memory.mem", ROM, 0, 15);
   end
endmodule
```

Code 4

```
001   //these are the desired outputs
100
100
100
010
001
100
100
010
010
001
100
010
010
010
001
```

Memory 1

Task 5

In Task 5, we were asked to implement a 2-bit by 2-bit multiplier with two 2-bit inputs (a, b) and a 4-bit product output using a ROM. This was a simple task as no logic was required in the code. We just needed to derive/know the desired outputs and put them in a .mem file. The code performed as intended and can be seen in *Code 5* below along with the memory file in *Memory 2*.

```
module Task5(
    input [1:0] a,
    input [1:0] b,
    output [3:0] c
    );

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

     assign ROM_addr = {a[1:0], b[1:0]};
     assign c = ROM[ROM_addr];

     begin
     initial $readmemb("Multiply.mem", ROM, 0, 15);
     end
endmodule
```

Code 6

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

Memory 2

Conclusion

In conclusion, this lab was very helpful for us to understand different types of decoders and learning about ROM. I would say that this lab was not too difficult after a bit of understanding with the decoders and memory, and the provided truth tables helped us with writing the code. Overall, this lab was a good learning experience and not too difficult in performing the required tasks.