Gabriel Emerson
ELEC 4200 – Lab 1
08/26/21

Lab 1: Modeling Concepts

Goals:

The goal for this first lab, was to mainly get started with some of the main features of Verilog and see how this Verilog IDE works. The beginning of the lab just talks through what some of the certain concepts are and how they are used in the software. In detail, this lab works through the different models including Gate-Level, Dataflow, Behavioral, and even how to start mixing these models to obtain better results.

Design Process:

The process of getting started for this lab starts with going over the beginning of the lab and reading through the different types of models. Then using these concepts, start to begin on the Verilog code and how it works. Since this is Lab 1, most of my prelab was getting used to Verilog, and how to use it to obtain the circuits I wanted.

The next step was going through the tasks to begin thinking how to create these circuits in Verilog. Task 1 was easy to think through, knowing it is only AND and OR gates, I only needed to remember how these functions in Verilog work. Task 2 was similar to the first, however, I needed to change the circuit to be 2 bit wide instead of the original 1 bit wide. This took time to figure out however after a bit of research I found how to prepare the circuit for 2 bit wide input and output.

Task 3 was an upgrade in circuit modelling, from Gate-Level, to Dataflow model of the circuit. The circuit was to do the same thing as the previous model, however, in Dataflow modelling instead. This means instead of using the AND and OR functions, use the ‘assign’ function already set in Verilog. This task is based all around the assign function and what all it can do with simplifying the circuit, and even assigning a delay in the circuit.

Task 4 was another upgrade in the modelling, this time to a Behavioral circuit. This type of circuit is for more complicated circuits and allows the user to create more while putting less. The main thing that sets this apart is the use of begin loops. This makes long circuits easier to read and assign.

The final task was to create a 3 to 1 multiplexer using what has been given so far. There were no specifications given to which method to use, in this case, since I understood the Gate-Level modelling the best, I created my 3 to 1 mux using Gate-Level modelling. This was just adding more a few more inputs, with another AND and OR. After creating the truth table for this mux, seeing how it worked in Gate-Level made more sense then trying to work out the begin loop.

Detailed Design:

Task 1 was simply coding the 2 to 1 mux using Gate-Level modelling. This can be done by the code shown in figure 1.

```
module Lab1_gateLevel(A, B, S, Y);
    output Y;
    input A, B, S;
    wire BS, AnotS, notS;

    and (BS, B, S), (AnotS, A, notS);
    not (notS, S);
    or (Y, BS, AnotS);

endmodule
```

Figure 1

Each function in Verilog is called a 'module' and takes the form in this figure as "module Lab1_gateLevel(…);". Inside the parenthesis of the module declaration is all the variables to be used in the module. They are then declared underneath whether they will be an input, output, wire, register, etc. After this we see what makes this code Gate-Level, which is the 'and', 'not', and 'or' functions. Inside these functions we see the first variable (a type wire) which is the output of whichever function is being called, with the second and third variables called, to be the inputs. Then after the module is finished, it must be ended with the statement 'endmodule'.

Task 2 was similar to task 1, however, we changed the variables from being a standard 1 bit, to everything being 2 bit wide. The code for this task is shown in figure 2.

```
module Lab1_gateLevel_2wide(A, B, S, Y);
    output Y[1:0];
    input A[1:0], B[1:0], S;
    wire notA[1:0], notB[1:0], notS;

    and (notA[0], B[0], S), (notB[0], A[0], notS), (notA[1], B[1], S), (notB[1], A[1], notS);
    not (notS, S);
    or (Y[0], notA[0], notB[0]), (Y[1], notA[1], notB[1]);
endmodule
```

Figure 2

This code looks very similar to that of task 1, however there is one big difference of each variable (except the S "select" variable) is that it is a vector variable. This means it has 2 bits to control the variable. The module starts the same with declaring the module name and all variables associated. However, then the vector variables must be declared as such, this includes the type wire variables. Then in the 'and', 'not', and 'or' functions, we use [x] to denote which

part of the variable we are wanting to act on in the circuit, x being the bit we choose. This is very similar to task 1 but uses more data and complexity with a second bit.

Task 3 starts to be different by using a new type of model in Verilog. The main difference that makes this code a Dataflow model, is the use of the 'assign' function. This can be viewed in figure 3.

```
module Lab1_dataFlow(x, y, s, m);
    output [1:0] m;
    input [1:0] x, y;
    input s;
    wire notA[1:0], notB[1:0], notS;

    // The '?' means output Y will equal B if line S is true, or equal A if line S is false
    assign #3 m[0] = (s) ? y[0]: x[0];
    assign #3 m[1] = (s) ? y[1]: x[1];

endmodule
```

Figure 3

Like the previous tasks, this code begins with the declaration of module, with module name and declared variables. Since this task also required 2 bits, the variables are declared as such, and are considered vector variables. The main difference in this task is the use of the 'assign' statement. This statement can (reading left to right) first take a '#x' to denote a delay amount of time in nano seconds. This statement then states what the output is and sets it equal to → read the value of 's' and if True (or 1) output = y, if 's' is False (or 0) then output = x.

Task 4 we are given a new model type and begin use of ‘always’ loops. This makes complex circuits easier to write and read. This code is shown in figure 4.

```
module Lab1_Behavioral(A, B, S, Y);
    input [1:0] A, B;
    input S;
    output reg [1:0] Y;

    always @ (A or B or S)
    begin
        if (S)
        Y = B;
        else
        Y = A;
    end
endmodule
```

Figure 4

This module looks pretty similar to all the other before, and abide by the same rules. First state the module name, variables, set them to inputs/outputs, and then begin construction. Behavioral modelling is usually denoted by the use of ‘always’ loops. These are used by first stating all variables to be used in the loop, then use a ‘begin’ and ‘end’ statement to show what all is in the loop. We also use ‘if’ and ‘else’ statements that are used just like any standard programming language. This makes our task of creating a 2 to 1 mux really simple and only needing a few lines of code.

Task 5 is the only task that requires a new type of circuit. We are now creating a 3 to 1 mux by adding in a few more variables, and using whichever model we choose. The code for task 5 is shown in figure 5.

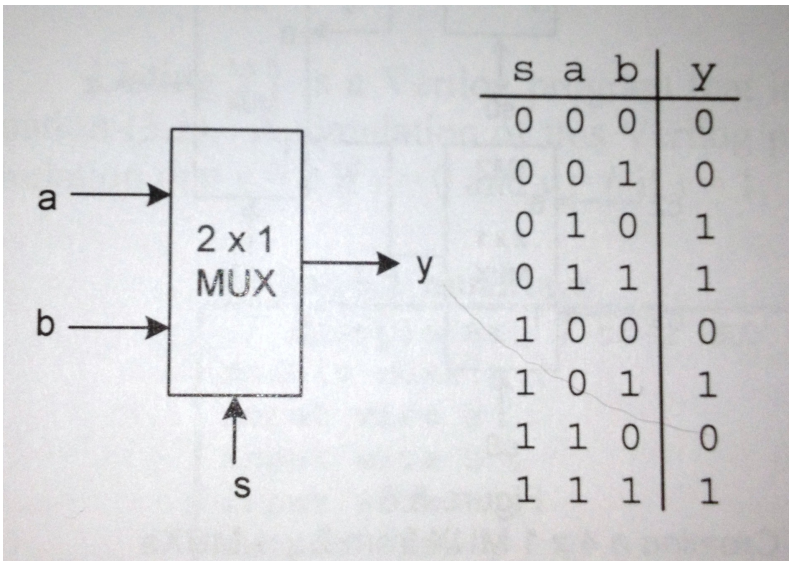
```
module Lab1_3to1MUX(A, B, C, S1, S2, Y);
    output Y;
    input A, B, C, S1, S2;
    wire D, DBS, BS, AnotS, notS1, notS2, CnotS2;

    and (BS, B, S1), (AnotS, A, notS1);
    not (notS1, S1);
    or (D, BS, AnotS);

    and (DBS, D, S2), (CnotS2, C, notS2);
    not (notS2, S2);
    or (Y, DBS, CnotS2);
endmodule
```

Figure 5

In this code it is easy to tell I used Gate-Level modelling. I did this because it is the easier to understand and modify on this very low level model. The main difference in this task and task 1, is the new variables. We now have 5 inputs instead of 3, which contain 3 main variables and 2 selects. We then use the 'and', 'not', and 'or' functions to create the low level diagram of a 3 to 1 mux and this completes our task 5.

Verification:

Task 1, 2, and 4, were all verified in the same way. The Verilog file was uploaded to the Nexys4 DDR board and the user manually flipped the switches set in the constraints file to view the output on an onboard LED. All 3 of these tasks were successful in this experiment. All of these outputs matched that of the truth table shown in figure 6.



s	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 6

Task 3 was verified a little different from the other tasks. This task came with a testbed file given in the lab description. After importing this file into the project files, the user could then run a simulation of the file to see what the values in output would be. Figure 7 (on next page) shows the output simulation for task 3.

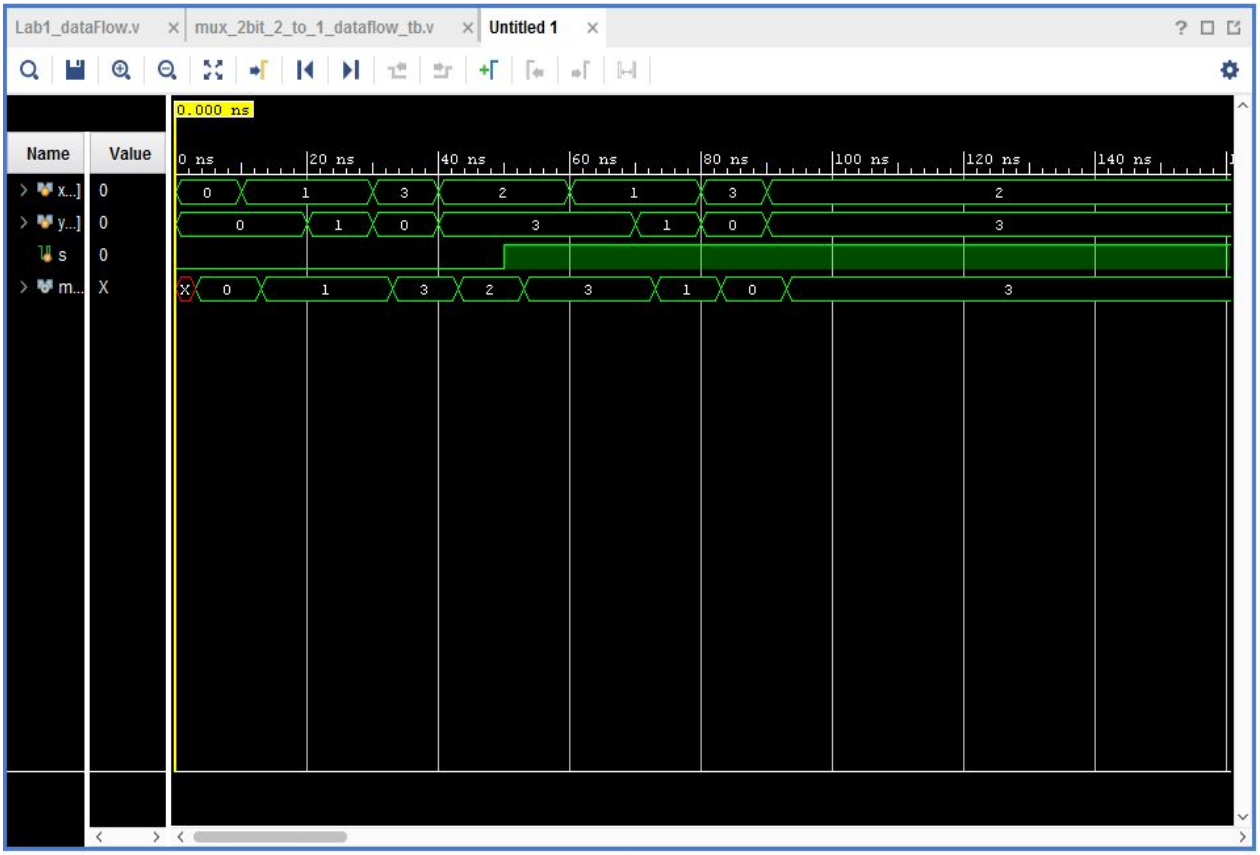


Figure 7

The final task was creating a 3 to 1 multiplexer. This was verified by uploading to the Nexys4 DDR board and the user manually flipped the switches set in the constraints file to view the output on an onboard LED. After modifying the constraints file to accommodate having 3 inputs and two select, the task LED matched with the chart given in figure 8.

SEL	OUT
00	IN0
01	IN1
10	IN2
11	IN2

Figure 8

Conclusion:

What I learned in this lab is first that the functions are there to help. If you do not know what these given functions do, then the code becomes hard to read, and even harder to implement. I also learned that making the same circuit in each model level makes the construction of the overall circuit much easier. This is nice because it is easier for the user to conceptually get the idea of Gate-Level, but it is much easier to implement in the Behavioral model. Lastly, I learned that the simulation, and other tools, are there to help the user in creating the circuit they want. This lab went well for me since I ended up being able to get all 5 tasks to work before the end of lab. If I would have done it all again, I wish I would have really known what the Gate-Level model was doing before moving on. After figuring out how each model can relate to another and why each is important, the construction of the overall circuit is more straightforward.