CPEN 230L: Introduction to Digital Logic Laboratory
Lab #6: Verilog and ModelSim
Gabe DiMartino
October 17, 2023

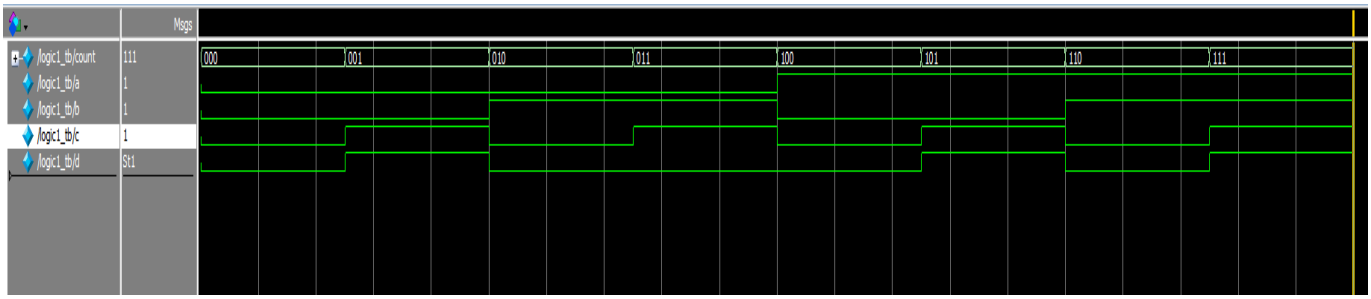
Part 1: Combinational Logic Design using Verilog and ModelSim.

Prelab work: The truth table below demonstrates the Prelab work for the Combinational Logic design. Each input and output were extracted from the starter code, with the theoretical solutions matching the tested output.

X	Y	Z	W	✓
0	0	0	0	✓
0	0	1	1	✓
0	1	0	0	✓
0	1	1	0	✓
1	0	0	0	✓
1	0	1	1	✓
1	1	0	0	✓
1	1	1	1	✓

Procedure: Following the provided starter code and procedure, each step was followed to produce the waveform simulation and transcribed truth table.

```
VSIM 3> run
# time   a b c  d
#    0   0 0 0  0
#    5   0 0 1  1
#   10   0 1 0  0
#   15   0 1 1  0
#   20   1 0 0  0
#   25   1 0 1  1
#   30   1 1 0  0
#   35   1 1 1  1
# ** Note: $finish    : D:/CPEN230L/Lab6/Lab6a/logic1_tb.v(16)
#    Time: 40 ps  Iteration: 0  Instance: /logic1_tb
```



```
// CPEN 230L Lab 6a, Combinational Logic
// Gabe DiMartino, 10/17/2023

module logic1(W, X, Y, Z);
output W;
input X, Y, Z;
wire P, Q;

// structural model
not(P, Y);
or(Q, X, P);
and(W, Q, Z);

endmodule

/* commented-out alternate implementation

module logic1(W, X, Y, Z);
output W;
input X, Y, Z;

// behavioral model
assign W = (X || !Y) && Z;
endmodule
*/


// CPEN 230L Lab 6a, testbench for module logic
// Gabriel DiMartino, 10/17/2023

module logic1_tb; // test bench, so no inputs/outputs
reg a, b, c;      // inputs to the DUT
wire d;           // output from the DUT
reg [2:0] count;  // 3-bit value to help generate DUT inputs

logic1 DUT(d, a, b, c); // instantiate the DUT

initial begin
      count = 3'b0;   // initialize count to 000 binary
      $display("time  a b c  d");     // truth table header
      $monitor("%4d %2d %1d %1d %2d", // output formatting
      $time, a, b, c, d);           // signals to be output
      #40 $finish;  // time goes 0 to 40 in steps of 5, then ends
end

always begin
      {a, b,c} = count;  // 3-bit count goes to 1-bit a, b, c
      #5 count = count+1; // increment count every 5-time ticks
end
endmodule
```

Results: Upon testing both the structural and behavioral models of the code, each simulation returned the same waveform graph and truth table. These configurations matched with theory and provided a thorough explanation on how the ModelSim software is used. Note that both the waveform image and transcript have been cropped to demonstrate the values spread across 5ps with a total of 40ps. Every change in the waveform corresponds to a change from 0 to 1 and vice versa. Finally, the 2 different files are used to run logic (logic1) and visualize the logic in the format of a waveform or truth table (logic1_tb).

Discussion: This first part of Lab 6 is critical to the understanding of both Verilog and the Intel/Quartus suite. Using ModelSim allowed for an easy demonstration of logic circuits such as the one above. Similarly, changing between the behavioral and structural models helped demonstrate the seamless interchange between using standard functions such as “And” and their bitwise operator “&”.

Part 2: Majority Gate Design using Verilog and ModelSim.

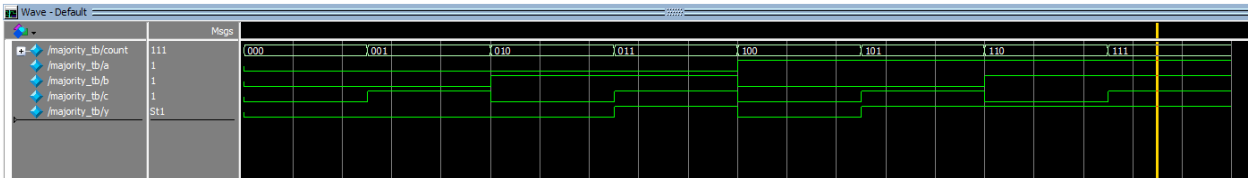
Prelab work: The truth table below demonstrates the Prelab work for a majority gate design. Each input and output were initialized alongside the creation of the majority gate code to ensure the Verilog program matched with the theoretical output.

A	B	C	Y	✓
0	0	0	0	✓
0	0	1	0	✓
0	1	0	0	✓
0	1	1	1	✓
1	0	0	0	✓
1	0	1	1	✓
1	1	0	1	✓
1	1	1	1	✓

```
//CPEN 230L Lab 6b majority logic
//Gabe DiMartino 10/17/2023

// Define the module
module majority (
    //Three "switch" inputs
    input a,
    input b,
    input c,
    output y //LED output
);

// Define three 2-input AND gates
and gate1 (and1, a, b);
and gate2 (and2, a, c);
and gate3 (and3, b, c);

// Define two 3-input OR gates
or or_gate1 (or1, and1, and2);
or or_gate2 (y, or1, and3);


endmodule

//CPEN 230L Lab 6b testbench for module majority
//Gabe DiMartino, 10/17/2023

module majority_tb; //no inputs or outputs

  reg a, b, c; // Inputs to the DUT
  wire y;  // Output from the DUT
  reg [2:0] count; // 3-bit value to help generate DUT inputs

  // Instantiate the majority module
  majority DUT(.a(a),   // Input 'a'
    .b(b),   // Input 'b'
    .c(c),   // Input 'c'
    .y(y)    // Output 'y'
  );

  // Initial block
  initial begin
    count = 3'b0; // Initialize count to 000 binary
    $display("time  a b c  y"); // Truth table header
    $monitor("%4d %2d %1d %1d %1d", $time, a, b, c, y); // Output formatting

    // Run the simulation for 40 time units
    #40 $finish;
  end

  // Always block to generate inputs
  always begin
    {a, b, c} = count; // Assign 3-bit count to 1-bit a, b, c
    #5 count = count + 1; // Increment count every 5 time units
  end

endmodule
```

Procedure: No procedure changes were made, and all instructions were followed.

```
/SIM 6> run
# time  a b c  y
#    0  0 0 0 0
#    5  0 0 1 0
#   10  0 1 0 0
#   15  0 1 1 1
#   20  1 0 0 0
#   25  1 0 1 1
#   30  1 1 0 1
#   35  1 1 1 1
# ** Note: $finish    : D:/CPEN230L/Lab6/Lab6b/majority_tb.v(25)
#    Time: 40 ps  Iteration: 0  Instance: /majority_tb
```



Results: Upon testing the structural model of the code, the simulation returned a valid waveform and truth table that matches with theory. The waveform image and transcript have been cropped to demonstrate the values spread across 5ps with a total of 40ps. Interpreting the image, during intervals 000 and 011, the output is 0 as expected, then changes when both inputs B and C are activated. This matches with the theory of the majority gate and is supported by the transcribed truth table.

Discussion: Part 2 provided a simple and useful practice with Verilog and ModelSim simulation. Unlike the previous simulation, the code was derived during the prelab, allowing me to practice programming in a structural style. All code is well documented to explain how the logic works. An important notation is the DUT variable when calling modules. This stands for Device Under Testing and ensures both the user and the compiler know the test bench is testing the provided logic. Similarly, the test bench module is commented with the modifications to the original file provided in Part 1 and makes use of the dot operator to connect 2 signals ".a to (a)".

Part 3: Full Adder Design using Verilog and ModelSim.

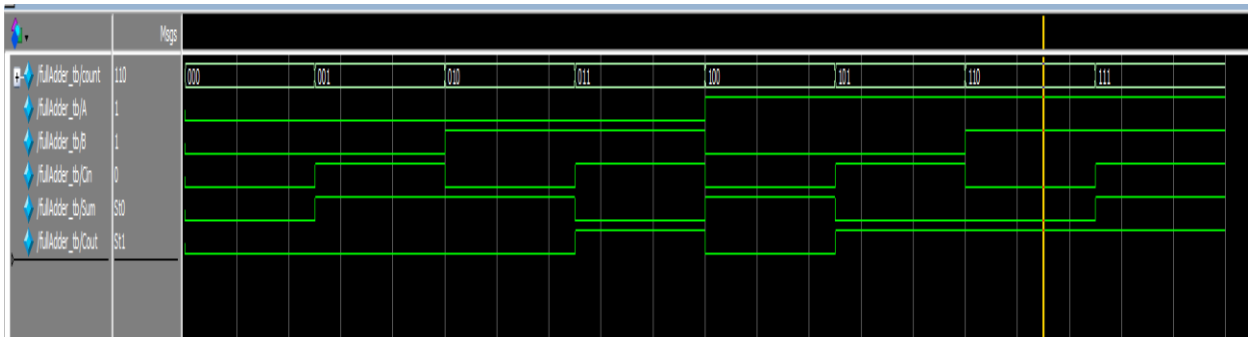
Prelab work: The truth table below demonstrates the Prelab work for a full adder design. Each input, carry, and output were initialized alongside the creation of the full adder code to ensure the Verilog program matched with the theoretical output.

A	B	Cin	Sum	Cout	✓
0	0	0	0	0	✓
0	0	1	1	0	✓
0	1	0	1	0	✓
0	1	1	0	1	✓
1	0	0	1	0	✓
1	0	1	0	1	✓
1	1	0	0	1	✓
1	1	1	1	1	✓

```
//CPEN 230L Lab 6b fullAdder logic
//Gabe DiMartino 10/17/2023

module fullAdder (input A, input B, input Cin, output Sum, output Cout);
    assign {Cout, Sum} = A + B + Cin;
endmodule

// CPEN 230L Lab 6b testbench for module fullAdder
// Gabe DiMartino, 10/16/2023

module majority_tb; // no inputs or outputs
  reg A, B, Cin; // Inputs to the DUT
  wire Sum, Cout; // Outputs from the DUT
  reg [2:0] count; // 3-bit value to help generate DUT inputs

  // Instantiate the fullAdder module
  fullAdder DUT (
    .A(A),
    .B(B),
    .Cin(Cin),
    .Sum(Sum),
    .Cout(Cout)
  );

  // Initial block
  initial begin
    count = 3'b0; // Initialize count to 000 binary
    $display("time  A B Cin Sum Cout"); // Truth table header
    $monitor("%4d %1b %1b %1b %1b %1b", $time, A, B, Cin, Sum, Cout); //
Output formatting
    // Run the simulation for 40 time units
    #40 $finish;
  end

  // Always block to generate inputs
  always begin
    {A, B, Cin} = count; // Assign 3-bit count to 1-bit A, B, Cin
    #5 count = count + 1; // Increment count every 5 time units
  end

endmodule
```

Procedure: No procedure changes were made, and all instructions were followed.

```
VSIM 3> run
# time  A B Cin Sum Cout
#    0 0 0 0 0 0
#    5 0 0 1 1 0
#   10 0 1 0 1 0
#   15 0 1 1 0 1
#   20 1 0 0 1 0
#   25 1 0 1 0 1
#   30 1 1 0 0 1
#   35 1 1 1 1 1
# ** Note: $finish    : D:/CPEN230L/Lab6/Lab6c/fullAdder_tb.v(26)
#    Time: 40 ps  Iteration: 0  Instance: /fullAdder_tb
```



Results: Upon testing the behavioral model of the code, the simulation returned a valid waveform and truth table that matches with theory. The waveform image and transcript have been cropped to demonstrate the values spread across 5ps with a total of 40ps. Interpreting the image, the bottom most wave refers to the carry output. As numbers increase, the carry becomes more relevant. The second line from the bottom represents the sum of two inputs A and B. This matches with the theory of the full adder and is supported by the transcribed truth table.

Discussion: Like Part 2, this second part was another great experience of practicing Verilog using the behavioral model. I personally found it easier to code compared to the structural model due to its similarity to other languages such as C++ which relies on bitwise operators. This representation of a full adder has a lot of potential as it can be looped with a parameter to create an N-bit adder or subtractor with minor tweaking.

Summary and Conclusion

Upon completion of this lab, I found it enjoyable to program gate logic using Verilog. Unlike Cedar, Quartus, and the physical wire connections, ModelSim was a simple alternative which allowed me to program several complex circuits in an easy to read and configure method. Moving forward into other labs and the final project, this lab will be critical to my understanding of Verilog and programming logical designs.