CSC258H Lab 7: A Taste of Verilog

1 Reminder: Submit Your Last Lab!

If you didn't submit all of the ALU lab that occurred before reading week, please start by demonstrating it to your TA. Here were the items being checked: 1 mark for showing your left circular shift and ALU schematics to the TA and explaining how they work; 1 mark for completing and testing the left circular shift circuit *and demonstrating your understanding of the device* to the TA; and 1 mark for completing the ALU circuit.

2 Introduction

This week, we'll experiment with a hardware design language called Verilog. While the schematics we used before reading week suffice for specifying small circuits, they become increasingly cumbersome as the size of the circuits being designed increase. To build larger logic devices and state machines, we would need to switch to a design language.

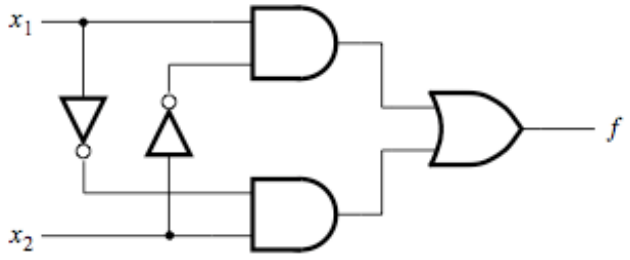
Unfortunately, Verilog (and the other major hardware design language, VHDL) can be difficult to use effectively because the problem it is solving is very difficult. Hardware design forces you to consider many issues that you don't normally encounter when programming. For example, should your assignment be "blocking" (a register) or "non-blocking" (a wire)? Do you care about the order in which your statements are executed, or can they be executed in parallel? We will explore these issues in this lab.

The easiest mistake to make in Verilog is to generate complex code. For example, avoid code with nested loops and many variables. As a general rule, you should always have a rough idea of the circuit diagram the compiler will create from your Verilog design. If you cannot envision the circuit that Verilog will build from your code, simplify it.

If you want to do further reading about Verilog or need a reference, the web has a few solid choices. Here are two favorites:

1. Verilog in One Day: http://www.asic-world.com/verilog/verilog_one_day.html
2. Verilog Quick Reference: http://www.asic-world.com/verilog/vqref.html

3 Creating a Verilog Project




```
module light(x1, x2, f);
    input x1, x2;
    output f;
    assign f = (x1 & ~x2) | (~x1 & x2);
endmodule
```

Begin by implementing a simple light controller circuit. The circuit and the verilog code that implements it is displayed above. Create a new Quartus project and add a new Verilog HDL file. Note that the Text Editor in Quartus provides a series of Verilog templates. For now, however, just enter the Verilog code from above into the file.

As you enter the file, note its structure. You're creating a module (a device) named *light* that has three wires. Two of those wires are declared to be inputs, and one is declared to be an output. The output line is assigned a logical value, with & standing for "AND", | standing for "OR", and ~ standing for "NOT".

The assign statement uncovers one key difference between Verilog and most traditional languages. If we had more than one assign statement, they would be considered to be in parallel, and the ordering of the statements would not matter. The assignments are said to be "continuous" or "concurrent". This reflects the parallel nature of many circuits. (Think about your 7-segment controller, for example. Each bit of the output could be computed in parallel.)

Compile the circuit and create a test vector to verify that the circuit is working as expected. **Show your TA the simulation result** before moving on.

4 Multi-bit Values and Procedural Statements

```
module mux4to1(In0, In1, In2, In3, S, f);
   input In0, In1, In2, In3;
   input [1:0] S;
   output reg f;

   always @(*)
   begin
      if (S == 2'b00)
         f = In0;
      else if (S == 2'b01)
         f = In1;
      else if (S == 2'b10)
         f = In2;
      else
         f = In3;
   end
endmodule
```

```
module mux4to1(In0, In1, In2, In3, S, f);
   input In0, In1, In2, In3;
   input [1:0] S;
   output reg f;

   always @(*)
   begin
      case (S)
         0: f = In0;
         1: f = In1;
         2: f = In2;
         3: f = In3;
      endcase
   end
endmodule
```

The code above contains two different implementations of a 4-to-1 mux. Both implementations use "always" blocks. The statements within an always block are executed procedurally (sequentially), rather than in parallel. The always block itself, however, is executed concurrently; if a module has multiple always blocks (or a combination of always blocks and assigns), they are executed in parallel.

The parameter list to an always block is called a "sensitivity list". The list should include all signals (wires) that affect the output generated by the always block. In both of the examples above, the wildcard "*" is used to indicate that all of the inputs to the module affect the output of the always block. If you can be more precise, you should be, as it will decrease simulation time.

The two implementations from above both contain a multi-bit input. The selector bits are designated by an array of bits named S. The array can be dereferenced as normal. For example, "S[0]" refers to the bit at position 0 in array S, and "S[1]" refers to the bit at position 1. The keyword "reg" in front of the output tells the compiler that the output should maintain its value until it is changed by the always block; the output value is being stored in a flip-flop (or "register").

The two implementations above differ in two ways. First, one uses if-else statements, and the other uses a case statement. Verilog also supports "for" and "while" loops within always blocks. Second, the example to the right compares S to a 2-bit binary value, and the example to the left uses decimal values.

Using the code from above as an example, create a Verilog module for a **7-bit wide, 8-to-1 mux**, i.e., the mux is choosing from eight inputs each of which is a 7-bit number. After compiling the file, create a test vector that verifies that the selector bits steer the correct input to the output and verify the results of your simulation. **Show it your TA.**

5 Blocking and Non-Blocking Assignments

```
module Dflipflops_v1(D, Clock, Q1, Q2);
   input D, Clock;
   output reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 = D;
      Q2 = Q1;
   end
endmodule
```

```
module Dflipflops_v2(D, Clock, Q1, Q2);
   input D, Clock;
   output reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end
endmodule
```

Verilog uses two kinds of assignments – blocking and nonblocking. "=" designates a blocking assignment and "<=" designates a nonblocking assignment. The two modules above are identical except that the one on the left uses blocking assignments and the one on the right uses nonblocking assignments. Both circuits contain only two D flip-flops. Your goal is to determine what the difference is between the two modules by creating a test vector and simulating both circuits. Use the results of simulation to infer the circuit diagram for each module.

Draw the circuit diagrams for these two implementations and show them and the results of your simulation to the TA.

6 Summary of TODOs

Below is a short summary of the steps to be completed for this lab:

1. Submit the ALU lab, if you did not do so before reading week.
2. Implement the light controller circuit, simulate it, and show it to your TA.
3. Implement the 7-bit wide 8-to-1 mux and show it to your TA.

4. Simulate the blocking and nonblocking assignments and draw their circuit diagrams. Be prepared to explain the difference between the two implementations to your TA.

Evaluation (3 marks in total): 1 mark for the simulation result of the light controller circuit, 1 mark for the simulation result of the 7-bit wide 8-to-1 mux, and 1 mark for the circuit diagram and the simulation result for the blocking/non-blocking assignments.

Congratulations! We're now done with all the Quartus/DE-2 labs. Next week we will start working with a new tool – assembly!