EE281 Logic Design Lab –Spring 2015- Ashley

Lab 3: Introduction to Verilog and the Spartan3/Basys2

Name: _____ Lab Partner: _____

Pre-lab

Answers to pre-lab questions and a design of any circuits you will build or experiments you will perform must be turned in at the beginning of your lab period. You may want to make a copy of the pre-lab for reference while you do the lab because you will not have the copy you turn in at the beginning. Make sure all the work you turn in is your own. I.e., do not copy other student's work (This applies to lab reports as well as pre-labs).

Hardware Description Languages

Schematic diagrams are an intuitive way to represent logic circuits. As the number of transistors on a chip increases to several hundred million on the high-end, low-level graphical formats, like schematics, become unmanageable. One way to improve the understandability of large digital circuits is to use a hardware description language to describe the circuit.

Like programming languages such as C++ and Java, a hardware description language (HDL) is a textual representation of how to solve a problem. However, a hardware description does not spell out what the circuit does procedurally. Rather it specifies what hardware is present, in a way that makes modular reuse easy. Though they could, in theory, be used to describe circuits built with discrete components, HDL's are usually used for *Programmable Logic Devices* (PLD's), chips that can have their functionality determined by a user, or *Application Specific Integrated Circuits* (ASIC's), chips custom built for a particular application.

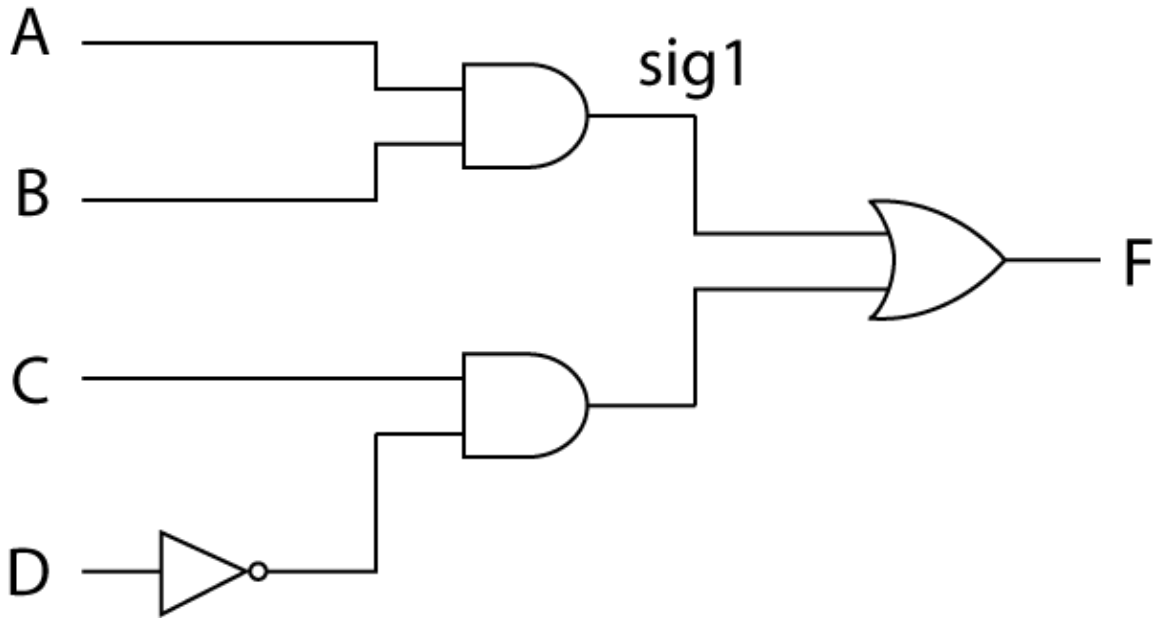
In EE 281, we will use an HDL called Verilog for Field Programmable Gate Arrays (FPGA's). We will talk about FPGA's in more detail in future labs. For now just know that FPGA's are chips we can reprogram (I.e., reconfigure) to implement any logic function we want, with some limitations that we will discuss in future labs.

A Brief Introduction to Verilog

Verilog is described at length throughout the EE 280 textbook, and by many other texts. For this lab, however, we only need a small subset of the language. In Verilog, a *module* is a sequence of HDL code that describes a particular logic function. Modules can *instantiate* other modules, much like functions in a procedural programming language can call other functions, except that each instantiated module corresponds with some circuitry.

In C and C++ a special function called `main` is where the program starts executing. Verilog has no such convention. The *top-level module* instantiates any other modules in the program, but it can be called anything. How it is designated depends on the software being used, but typically it must be the first module in the first file belonging to the project being implemented. The Xilinx Project Navigator allows the top-level module to be changed through the GUI.

To illustrate how to use Verilog to represent circuits, we will use the following logic circuit:



The Verilog code begins with:

```
module ex1(A, B, C, D, F);
      input  A, B, C;
      input  D;
      output F;
      wire sig1;
```

A module begins with the `module` keyword, which is followed by the module name, `ex1` in this case, and the parameter list. Following the module declaration comes a list of signals used in the module. These include inputs and outputs declared in the module parameter list, and intermediate wire signals.

Following the signal declarations the module continues:

```
      assign sig1 = A & B;
      assign F = sig1 | C & ~D;
endmodule
```

The `assign` statement assigns the value of an expression on the right hand side of the equal sign to a signal on the left hand side of the equal sign. In Verilog the '&' operator is used for AND, the '|' operator is used for OR, and the '~' operator is used for NOT. The precedence rules are the same as in boolean algebra, with NOT having highest precedence, followed be AND, then OR. Parentheses can be used to override the order of operations.

It is important to note that the `assign` statement is declaring how signals are connected with gates, rather than a sequence of actions to be taken. Order is not important. Reversing the two assign statements above has no affect on the circuit's behavior.

The `endmodule` keyword signifies the end of the module. More than one module can be placed in the same file, but typically only the first module in a file can be referenced by modules outside the file in which the module is declared.

Often when we have several related signals it is easier to group them together into a bus. Verilog provides facilities for doing this easily. For example the following module represents a circuit with three inputs X_2, X_1, and X_0, and a single output F.

```
module ex2(X, F)
       input [2:0] X;
       output F;
```

The input declaration for X says that X is a bus with inputs from 2 down to 0. In general, the top and bottom limits can be any numbers (within reason). The signals within the bus can be operated on as a group or individually.

```
       assign F = X[1] & X[0];
```

AND's together X_1 and X_0 and assigns the result to F.

```
       assign G = ~X;
```

Inverts all the bits of X and assigns them to G (assuming G has been declared previously as an bus of three wires or outputs.) In general the AND, OR, NOT, and most other operators all work on buses. Verilog also has several operators for accessing and concatenating slices of signals, all of which can save you much typing.

In some cases you will want to enter a constant value. Numbers are assumed to be decimal unless otherwise specified. To specify a number in binary, Verilog starts with the number of bits, followed by a single quote, the letter 'b', followed by the binary digits. For example, a 4-bit representation of the number two would be 4'b0010; likewise, the 3-bit value of seven is 3'b111.

In addition to describing how modules are connected and simple logic functions, Verilog provides the ability to write higher-level descriptions of circuit behavior. Such code is

called *behavioral* because it describes what the circuit does, rather than how it is implemented. For this lab, you may want to use the `case` statement, which essentially allows you to specify a truth table.

Assume we want to implement a function of three variables X_2, X_1, and X_0 with output Z specified by the truth table below:

$X_2X_1X_0$	Z
000	0
001	1
010	0
011	0
100	1
101	1
110	0
111	1

Behavioral code is contained with in an `always` block, which begins

```
always @(X or Z)
begin
```

The variables in parentheses following the '@' sign are known as the sensitivity list. In this case, we are saying that the circuit represented by this `always` block should change value any time X or Z change. For combinational circuits, any signal that appears on the right hand side of an assignment should appear in the sensitivity list. Missing signals in the sensitivity list will lead to the Verilog compiler inferring a latch, which is usually not what you expected.

The `case` statement looks like this, assuming X is a 3-bit signal:

```
case(X)
       3'b000:  Z = 0;
       3'b001:  Z = 1;
       3'b010:  Z = 0;
       3'b011:  Z = 0;
       3'b100:  Z = 1;
       3'b101:  Z = 1;
       3'b110:  Z = 0;
       3'b111:  Z = 1;
       default: Z = 0;
endcase
```

Here the Z is assigned a value based on the 3-bit value of X. Even though it appears that all cases are covered without it, the `default` statement is important to include. In addition to 1 and 0, binary Verilog signals can take on other values like 'z' and 'x' for high-impedance or unknown states respectively. Adding a `default` case removes the possibility that there is and uncovered case. If any case is ever left uncovered, then the case statment implies that the value does not change in that case, which will cause a latch to be generated (If you are taking EE 280 right now, don't worry about what a latch is yet, just know that you should always have a default when using the case statement.)

A final end statement closes out the always block.

This brief introduction has provided the minimum amount of Verilog needed to do this week's lab. Future lab sessions will deal with Verilog in more detail, including the difference between structural and behavioral code, and cost of generated code.

Pre-lab Questions

1. Write a Verilog module representing the equation:

   f(W,X,Y,Z) = X'Y'Z' + WYZ + WX'Y

   The module should have four inputs, W, X, Y, and Z, and one output, f.

2. Draw a schematic for the circuit represented by the Verilog module below:

```
1. module prelab2_2(V,W,X,Y,Z,OUT1,OUT2);
2. input   V;
3. input   W;
4. input   X;
5. input   Y, Z;
6. output  OUT1;
7. output  OUT2;
8. wire    wire1;
9.
10.         assign OUT1 = (V & ~W)|(Z & X);
11.         assign wire1 = Y & ~Z & ~V;
12.         assign OUT2 = wire1 & OUT1;
13. endmodule
```

   You need not minimize the logic, but be sure to label all signals named in the module.

Do not forget to turn in designs (handwritten is fine) for what you plan to do in lab.

Experiments

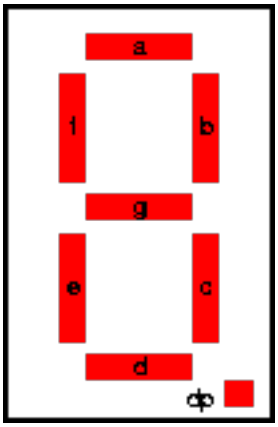
Part 1: Getting Started with Xilinx ISE and the Spartan 3

The first part of this lab will familiarize you with the design flow used for programming the Xilinx Spartan 3 starter kit boards we will be using for several of the experiments in EE 281. The resources available on them are described in the Spartan-3 User's Guide. Your first assignment for this lab will be to complete a tutorial designed to help you get started using the board and related design tools. We have also provided a simple tutorial (available on Bb under course content – ***EE281XilinxISimTutorial.pdf***) to help you get started. The detailed tutorial is also available in the course content area of Bb but it is 150 pages long. A copy of the Xilinx software (search Xilinx 13.4 Webpack – you will have to register) is available free of charge for non-commercial use from Xilinx's web site if you would like to run the software on your home computer or laptop. A sample file for the first part of the lab is also on Bb (Assignment tab - Lab3 – ***ee281tut.v***)

The circuit we will build will be the user input portion of a digital version of the old Rock, Paper, Scissors game, where two or more people count to three then make one of three shapes with their hand. A fist represents rock, two fingers represents scissors, and a flat hand represents paper. Rock beats scissors, scissors beat paper, and paper beats rock.

Our electronic version will only handle input from one player (we will expand it to two players in a later lab.) Two of the slide switches on the starter kit will select between rock (00), paper (01), and scissors (10). Two pushbuttons will enable the output display of which input is selected. The output will be displayed on a seven segment display.

A seven segment display is a collection of 7 Light Emitting Diodes (LED's) mounted in a configuration that looks like this:



The letter on each segment indicates which signal is used to turn on the segment. The Spartan-3User's Guide lists which pins on the chip are connected to each of the segments on the board. Each segment is connected to a pin on the FPGA using negative logic. Or when the pin outputs a zero the segment lights up, and when the pin outputs a one the

segment turns off. By turning on different combinations of segments we can get all the digits from 0-9. We can also get some, but not all of the letters of the alphabet. For this project we will turn on segments to make the letters 'r', 'P', and 'S'. (Don't worry the sample file will take care of this for you.)

The Spartan 3 board has four seven segment displays all connected to the same FPGA pins. Each display has an enable signal that turns it on. We can make it appear that there are different characters on the different seven segment displays be quickly turning each of them on one at a time. For this lab, though we will only use one of the displays.

To get started, follow the tutorial. When presented with the "New Source Wizard" enter the following instead of what is listed in the tutorial enter the following:

Port Name	Direction	Bus	MSB	LSB
HAND	input	checked	1	0
GO1	input	not checked		
GO2	input	not checked		
AN	output	checked	2	0
SSEG	output	checked	7	0
DISPLAY	output	not checked		

When editing the constraints file, use the Spartan-3User's Guide to determine which inputs and outputs should be connected to each pin. Be sure to simulate your circuit before trying it on the hardware (you will also need to include copies of your simulation with your lab report – this is required for all of our Verilog-focused labs).

Once it is working, demonstrate your circuit to your TA._____

Part 2: Displaying Numbers with 7 Segments

Using four of the slide switches as inputs; write a Verilog module to output 0-9 (if the input is <10 in decimal – less than 1010) and a “o” to represent an overflow if the input is greater than or equal to 10 in decimal. Display the result on the first digit of the 7 segment display when a binary number 0000 (0) to 1111 (15) are input on the four switches.

Simulate as described in the tutorial. When editing the constraints file, use the Spartan-3User's Guide to determine which inputs and outputs should be connected to each pin. Be sure to simulate your circuit before trying it on the hardware.

Demonstrate your circuit to your TA. _____