Project 1: ModelSim Tutorial and Verilog Basics

ENEE 359a: Digital VLSI Circuits, Spring 2008
Assigned: Thursday, Feb 7; Due: Tuesday, Feb 19

This project will give you a basic understanding of ModelSim and the Verilog hardware description language (HDL). ModelSim is an IDE for hardware design which provides behavioral simulation of a number of languages, i.e., Verilog, VHDL, and SystemC. The Verilog HDL is an industry standard language used to create analog, digital, and mixed-signal circuits. HDL's are languages which are used to describe the functionality of a piece of hardware as opposed to the execution of sequential instructions like that in a regular software application. Both of these tools are used extensively in industry, so knowing how to use them can be beneficial later in your career.

ModelSim Tutorial

Luckily, there is a free, student version of ModelSim that can be downloaded from the following location (**NOTE:** If you do not have access to a Windows based computer, ModelSim is installed in the Windows labs of A.V. Williams):

- http://www.model.com/resources/student_edition/download.asp

Follow the instructions on the page to install the program and obtain a student license, which they will send to you via e-mail.

Once you have received the license and everything has been properly installed, ModelSim should execute without issue. Navigate to the **Help->PDF Documentation** pull-down menu and select **Tutorial** from the list. Thankfully, ModelSim has provided a simple explanation on the basic use of the application.

Read through and follow along sections 1-4 and 6 (Using Verilog)

- **Note:** The ModelSim tutorial will not instruct you on the syntax/use of Verilog. Just use their files for now and explanations will follow later in this project.

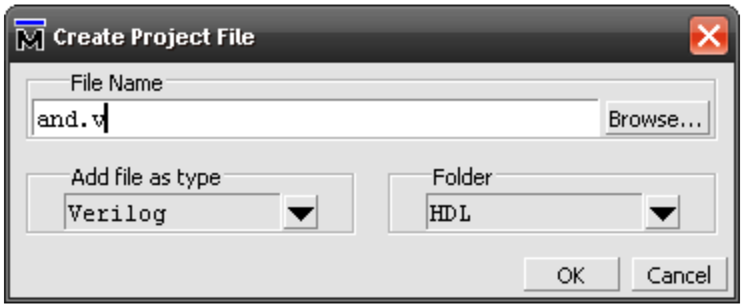
Verilog Basics

Now that you have a basic understanding of ModelSim, the following will give you some idea of how the Verilog language works. It is important to remember that the language is meant to model the functionality of physical hardware; thus, the language does not run as a sequential program like you are used to, where each step in a sequence of steps executes after the previous step has finished. In Verilog, as in hardware, all logic executes simultaneously. This is understandably confusing at first, but with practice it will become more intuitive.

AND Gate

The first step will be to create a module, the fundamental building block in Verilog. A module represents the fundamental building block of hardware: a piece of combinatorial or sequential logic.

1. Start by selecting the **Project** tab in ModelSim. Right-click on the **HDL** folder you created during the ModelSim tutorial and select **Add To Project -> New File**. Select the appropriate fields so that the dialog looks like this:



File Name: and.v
Add file as type: Verilog
Folder: HDL

After you click OK, **double-click** on the file to open it in the ModelSim editor.

2. Add the following text to the file :

```
module and2_1bit (a,b,c);
    input a;
    input b;
    output c;

    assign c = a&b;
endmodule
```

Save the file, select and **right-click** on it in the **Project** tab, and choose **Compile -> Compile Selected**. You should see a success message printed in the **Transcript** window at the bottom. Take a moment to look at the structure and syntax of the code you compiled, which describes a 1-bit, 2-input **AND** gate. Things to note:

- **Semicolon** after module declaration
- All signals in the **Port List** must be declared as either an input or output before they are used
- **endmodule** is one word and is not followed by a semicolon
- A naming convention for your modules will make your life infinitely easier – my convention above lists the number of inputs after the gate type followed by the number of bits of each input. You can have your own convention – just stick with it.

Once the file has been successfully compiled, select the **Library** tab and expand your **Work** library. You should now see **and2_1bit** listed as a module. We now have a 1-bit, 2-input AND gate to use in our designs. Too bad they aren't very useful. Let's make something a little more worthwhile.

3. Return to the editor and add the following text to the end of the file we were just working on :

```
module and2_32bit(a,b,c);
    input [31:0] a;
    input [31:0] b;
    output [31:0] c;

    assign c = a&b;
endmodule
```

This module performs a bit-wise **AND** on 2, 32-bit inputs, as you probably have guessed. Note how multiple bit-width inputs are declared. Similar to arrays in C, using the square bracket indicates that the signal is more than one bit. The **most-significant-bit** is listed first, and the **least-significant-bit** is second. Compile the file again and check your **Work** library. You will now see both your 1-bit and 32-bit AND gates listed.

- **Note:** Since you are allowed to have multiple modules per file, it is a good idea to keep similar modules in the same file to avoid clutter

Additional Gates: OR and XOR

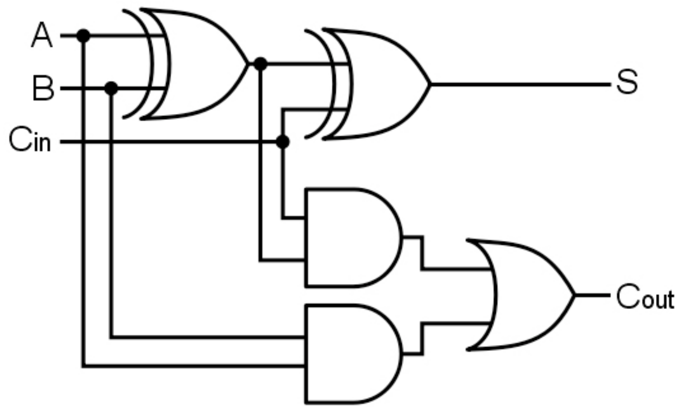
1. Create and compile similar files for an OR and XOR gate of various bit-widths (two modules for each logical function: OR and XOR gates of 1 bit and 32 bits).

Full Adder

Now that we have the basic building blocks of a digital system, we will create something useful: an adder. According to wikipedia "A **full adder** is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits."

The truth table for a full adder can be seen below. From the truth table we can reason that the block diagram can be created from the following logic (also from wikipedia):

A	B	Cin	S	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



You are now going to implement this in Verilog using the modules you have already created.

1. Under the **Project** tab, add a new Verilog file to your **HDL** directory called **fullAdder.v** and fill it with the following text:

```
module fullAdder(a,b,cin,
                 s,cout);
   input a;
   input b;
   input cin;
   output s;
   output cout;

   wire aXORb;
   wire cANDaXORb;
   wire aANDb;

   xor2_1bit XORgate1 (a,b,aXORb);
   xor2_1bit XORgate2 (aXORb,cin,s);
   and2_1bit ANDgate1 (cin,aXORb,cANDaXORb);
   and2_1bit ANDgate2 (a,b,aANDb);
   or2_1bit ORgate1 (cANDaXORb, aANDb, cout);
endmodule
```

   As you can see, there is far more to this module than our previous designs. After the input and output declarations, we are now declaring several **wire** objects. Think of these exactly as how they sound: a wire which will carry a signal from a source to a destination. We will use these wires to connect our gates together in order to create this **full-adder**.

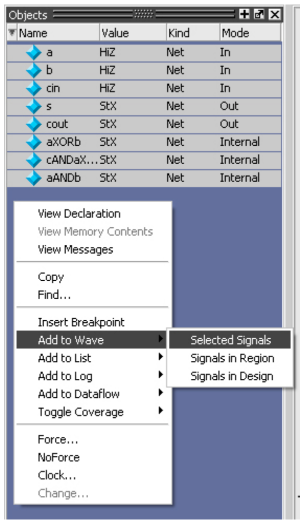
   After the **wire** declarations there are module declarations. Here, we instantiate the gates we have made and connect the appropriate wires to their respective ports. With the block diagram image above, try and reason about what was done in the code.

   - **Note:** Make sure the order of the wires in your port list is correct for your particular modules. You may have ordered them differently than in my code.
   - **Note:** Be as verbose as possible with your variable and module names. It will make your/my life much easier.

2. Assuming your file compiled successfully, you will now simulate the **full-adder** you just created. Under the **Library** tab, expand your **Work** library and **double-click** on the module labeled **fullAdder**. The ModelSim windows should rearrange themselves and some new text will appear in the **Transcript** tab.

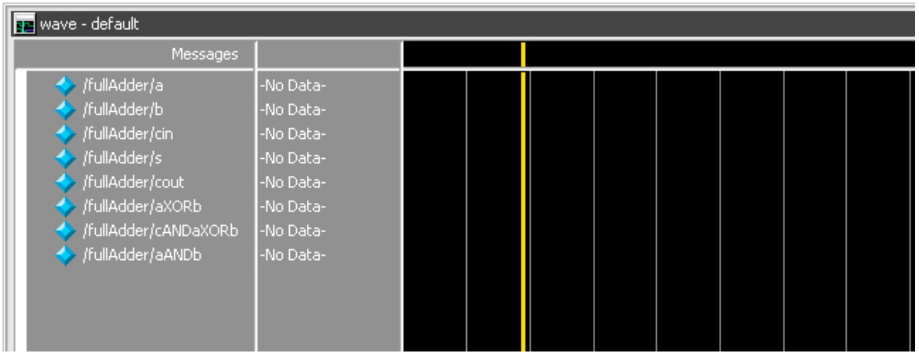
   - **Note:** Any action you perform with the user interface also has a corresponding command which can be executed from the command line. For example, loading a module for simulation (which you just did by double-clicking) can by done by executing the command `vsim work.and2_1bit` assuming your AND gate is named as such.

3. In the **Objects** window, select all of the items listed using the standard **shift-click**. Once they are all selected, **right-click** and select **Add To Wave -> Selected Signals**.



Name	Value	Kind	Mode
a	HiZ	Net	In
b	HiZ	Net	In
cin	HiZ	Net	In
s	StX	Net	Out
cout	StX	Net	Out
aXORb	StX	Net	Internal
cANDaX...	StX	Net	Internal
aANDb	StX	Net	Internal

This should open up the **Wave** tab with all of the signals from the **fullAdder** module.



Messages	
/fullAdder/a	-No Data-
/fullAdder/b	-No Data-
/fullAdder/cin	-No Data-
/fullAdder/s	-No Data-
/fullAdder/cout	-No Data-
/fullAdder/aXORb	-No Data-
/fullAdder/cANDaXORb	-No Data-
/fullAdder/aANDb	-No Data-

4. At the command line in the **Transcript** tab, type run 100 and hit enter. The waveform should display a bunch of red and blue lines representing undefined or high-impedance signals. This is because there are no values on the input. Let's change this.
5. At the command line, type the following:

```
force a 0; force b 1; force cin 1; run 100
```

   This time, the waveform should display green lines for each signal. Using the truth table above, make sure your outputs s and cout are the correct values. Using the syntax above, change the values of the inputs to observe the module's behavior. Hopefully, for all combinations of inputs, the correct output is produced.

4-bit Adder, Built from 1-bit Full Adders

1. Now, create a module for a 4-bit, 2-input adder using the full-adder module just created. Use the internets to find a block diagram. Turn in all of your HDL source code and a screenshot of the waveform after you have added together a couple of numbers (In decimal, explained below).

Tips:

- You can splice off individual wires from a multi-bit width signal by using the brackets, just like in C. If you need the **least-significant-bit** of signal output, address it with output[0].
- **Right-click** on a signal in the waveform window to select the radix. When turning in the screenshot of your results, and for your own use, use **Decimal**.
- When forcing signals via the command line you can specify the radix of the value you are forcing. Here is an example :

  ```
  force a ‘d4
  force b ‘d11
  ```

  The d stands for **Decimal**. h works for hex, b for binary, etc.