INTRODUCTION TO ACTIVE-HDL

TUTORIAL #3 – CREATING VERILOG MODULES AND TEST FIXTURES

After finishing Tutorial #1 and Tutorial #2, you now know all the basics to creating, debugging, testing, and simulating with schematic designs. Additionally, you have seen a test fixture and seen Verilog code as displayed in Active-HDL. This tutorial is not a tutorial on how to code in Verilog. However, we will discuss coding in Verilog and creating test fixtures in the Active-HDL environment. For help with the syntax and semantics of Verilog, refer to the class textbook, "Fundamentals of Digital Logic with Verilog Design" by Brown.

Upon completing this tutorial you will know:

- How to create a Verilog module in Active-HDL;
- How to place your modules into your design; and
- How to create a test fixture to test your design.

We recommend you do this tutorial in conjunction with or after reading the Verilog portion of the textbook and learning how to do basic Verilog coding. This tutorial deals only with Verilog as it pertains to Active-HDL and may not be applicable in other applications or Verilog coding environments.

Creating Verilog Modules In Active-HDL

Recall from Tutorial #1 how you set the top-level. You did this by expanding the list under the schematic you wanted to set as your top-level. In this list you may have noticed that there was a file by the same name as your design with a .v extension. This is the Verilog code produced by Active-HDL, and under that file is the Verilog module Active-HDL created for your design library. The point is that almost everything you have done in these tutorials has boiled down into Verilog, whether you knew it or not. Verilog is one of the languages we use to instantiate hardware. Active-HDL provides an environment that allows you to do this in either a schematic, in Verilog, or both. To illustrate how to create a Verilog module in Active-HDL, we will implement a simple 2-input AND-gate.

1. Start Active-HDL, and open an existing workspace or create a new workspace.
2. In the Design Browser, double-click "Add New File".
3. There are two ways to create a Verilog file, just like creating a new schematic. You can use the wizard, or create an empty file. The wizard generates some of the code for you based on what you enter at creation. It will set up the module statement, inputs, outputs, argument types, and place an endmodule at the end of the file for you. For this tutorial we will use the wizard because it can save time typing the obvious and is more convenient. However, as you become more comfortable with Verilog, you should feel free to start from scratch. This will become apparent as you use designs where you may want some arguments to be registers instead of wires (Active-HDL makes them wires).
4. Once you have started the wizard, make sure the "Add generated file to design" box is checked. Then click next.

5. Type the name of the module in the “Type the name of the source file to create:” field; we used my_and2. Leave the optional field blank. By default, the name of the file and the module are the same, which is a good naming convention. Click Next.
6. This window is just like the window you saw when adding ports to your schematic diagrams in Tutorial #1 and #2. It also works the same. All input and output ports added here will be placed in the Verilog source code by Active-HDL. We need three single bit ports: in1, in2, and out. Add these ports as you have done in previous tutorials, and click Finish.

Figure 1 shows the Verilog source file generated by Active-HDL. The comments provide information about the file, and the timescale is there for simulation (i.e.; #10 in the code below would be a 10 ps delay, more on this later).

```
//------------------------------------------------------------
//
// Title        : my_and2
// Design       : full_adder
// Author       : CSE
// Company      : UW
//
//------------------------------------------------------------
//
// File         : my_and2.v
// Generated    : Thu Mar 20 14:51:14 2003
// From         : interface description file
// By           : Itf2Vhdl ver. 1.20
//
//------------------------------------------------------------
//
// Description :
//
//------------------------------------------------------------
`timescale 1ps / 1ps

//{{ Section below this comment is automatically maintained
//   and may be overwritten
//{module {my_and2}}
module my_and2 ( out ,in1 ,in2 );

input in1 ;
wire in1 ;
input in2 ;
wire in2 ;

output out ;
wire out ;

//}} End of automatically maintained section

// -- Enter your statements here -- //

endmodule
```

Figure 1

Note: What Active-HDL means by "automatically maintained" in the comments is that it will modify the code to reflect any changes you make to the inputs and outputs in a schematic diagram. Active-HDL will only update the portion of your code that describes the terminal types and argument types. Therefore, if you edit a part created in Verilog in a schematic design, you do not need to manually alter the code in this section to reflect these changes. Editing your code and symbol will be discussed further later in this tutorial.

7. Now enter in the code as shown in Figure 2.

```
module my_and2 ( out ,in1 ,in2 );

input in1 ;
wire in1 ;
input in2 ;
wire in2 ;

output out ;
wire out ;

//}} End of automatically maintained section

// -- Enter your statements here -- //
assign out = in1 & in2;

endmodule
```

Figure 2

8. Save and compile the file.

When coding larger modules, you may find that Active-HDL does not provide very helpful error messages. However, it provides some tools that you may find useful for debugging. Figure 3 shows the tool bar that contains the following buttons (from left to right): Comment (use this button to highlight and comment out portions of code), Uncomment (uncomments highlighted portions of code), Create group (makes the highlighted portion of code an expandable and retractable list), Remove group, Generate structure, and Format text (the last two buttons both help organize your code and automatically create groups).

Figure 3

You may find that using these options to organize your code will help you find mismatched begins and ends or missed semicolons.

As you can see, all of the work that goes into creating a Verilog module is found in the code, not in using Active-HDL. Coding in Verilog requires you to think in terms of hardware and avoid trying to simply code an "algorithm" that does what you want. Active-HDL is very forgiving, and poor semantics will compile and simulate in Active-HDL with no problems. However, when it comes time to synthesize, you may find out differently. Practicing coding properly now will save you time and energy in future hardware courses.

Placing Verilog Modules In Your Design

Now that you have compiled your module, it is ready to use in a schematic diagram. This is done exactly the same as when you added the test fixture parts in Tutorial #2.

1. Open the “Symbols Toolbox”.
2. Expand the list associated with your design and locate the “Units without symbols” list.
3. Expand this list and select the module you created (my_and2).
4. Drag and drop the part into the schematic diagram.

Active-HDL automatically generates a symbol for your module, but recall from Tutorial #2 you can edit this symbol by right clicking on it. You can change the size of the part, or the location of pins. Earlier we mentioned that Active-HDL would automatically maintain portions of your code. This is true, but it will only update the portion of your code that describes the terminal types and argument types. Therefore, any changes to the names of the pins, adding new pins, or removing the pins should be done in the Verilog source code. The following steps will show you how to edit your code and update the symbol generated by Active-HDL to reflect any changes.

1. Double-click on the symbol in the schematic diagram, or double-click the name of the source code file in the Design Browser to open the file.
2. Make changes to the code, save, and compile the file.
3. Go back to the schematic diagram that contains the symbol for your module, and right-click it.
4. Select the “Compare Symbol with Contents…” option. This will bring up the “Compare Interfaces” window (Figure 4).
5. Click the “Update Symbol” radio button as shown in Figure 4, and click OK.

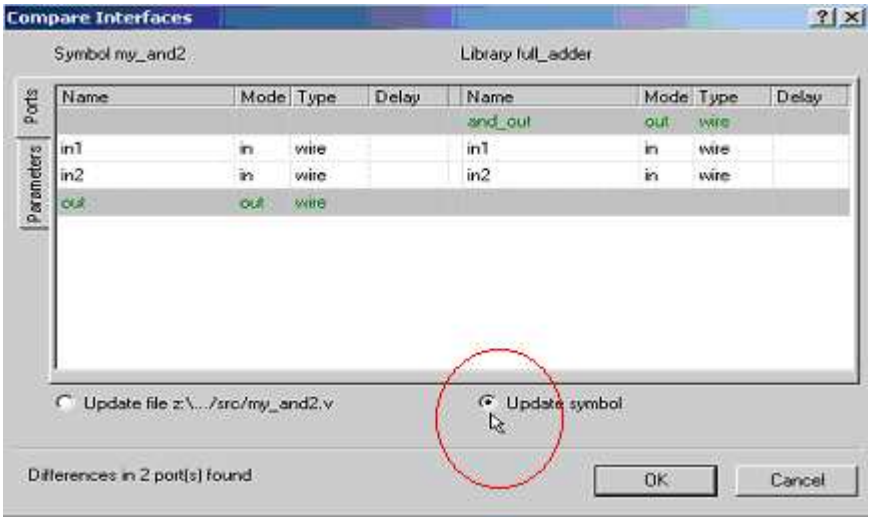

Figure 4

You need to save and compile your source code file after each change. Then, you should perform the above steps to ensure that your part reflects these recent changes. The case may be that small changes you make will not affect the operation of the symbol. If this is the case, then the “Update” radio buttons will be “grayed” out and all is well.

Creating Test Fixtures

Tutorial #2 described what test fixtures are used for and provided two test fixtures for you to use. This tutorial will provide more detailed information about test fixtures and how to use them to test your design and/or drive input signals for simulation. Even though this tutorial is not intended to teach Verilog, we think it is important to point out the coding techniques used to implement test fixtures.

Test fixtures are similar to other Verilog modules you have coded or will code. What separates a test fixture from other modules is the initial statement. The initial statement is used for simulation and will not synthesize. During simulation, the test fixture stimulates the inputs, compares the outputs, and prints the results to the screen (usually test fixtures print only the number of errors or a test passed message).

The steps below provide a general strategy for creating a test fixture for testing a module. Coding details and how to structure the tests are left for you to decide.

1. First, you should write down on paper all of the cases you want to test on your module. The temptation is to test a small set of cases and be convinced that your module works. Your job is to determine how many cases you need in order to best verify the correctness of your design.
2. Follow the steps described earlier to create a new Verilog source code file. You should name this file <name of module to test>_tf.v (the naming convention suggested in Tutorial #2). The inputs for the test fixture will be the outputs for the module to be tested. The outputs for the test fixture will be the inputs for the module to be tested.
3. Use an initial statement along with delays (#), display statements ($display), and other Verilog statements to implement your test cases you designed in step 1. Below is a portion of an initial statement and other Verilog code used to test the full adder from Tutorial #1.

```
initial
begin
    A = 0; B = 0; Cin = 0;
    #10
    if((Sum != 0) && (Cout != 0))
        $display("Error: Sum and Cout are incorrect");
    else if(Sum != 0)
        $display("Error: Sum is incorrect");
    else if(Cout != 0)
        $display"Error: Cout is incorrect");
    else
        $display("Case 1 passed");
```

Figure 5

Figure 5 is an example of one test case. You can imagine using similar blocks of codes to test additional cases (the test fixture above is viewable in its entirety in the file FA_tf.v). This test fixture outputs (drives) the inputs (A and B) to the full adder module, and it takes the module's outputs (Sum and Cout) as inputs. The test fixture samples the outputs of the module and compares them to the module's inputs, which it is driving. Notice that there is a delay from the time that the test fixture begins to drive the inputs to the time it samples the outputs. This is necessary to allow the module being tested to take the inputs

in and produce outputs. You can specify how long to delay per unit by using the timescale statement in your Verilog source code. A common mistake is to not give a long enough delay between tests. Be aware of the fact that the gates you will be using have delays. You should know the critical path of your design and set your delays accordingly.

Note: Another useful technique is to use a for-loop and the $random statement. This will allow you to test hundreds or thousands of random cases; however, designing your own specific cases is more reliable (i.e., who knows what is being tested if everything is random).

4. Save, and compile your test fixture.
5. Create a new schematic diagram (.bde file) and place your test fixture and the module to be tested in the schematic (remember to use a good naming convention).
6. Connect the test fixture's outputs to the inputs of the module to be tested, and connect the module's outputs to the inputs of the test fixture.
7. Save, run the check diagram, and compile the schematic.
8. Set the schematic as the top-level.
9. If the Console window is not viewable, select View/Console in the menu bar.
10. Initialize a simulation and run the simulation long enough for the test fixture to finish its test. The same simulation debugging techniques learned in Tutorial #1 still apply. Feel free to open a new waveform and add signals to the waveform in conjunction with your test fixture. While running the simulation, you can double-click the module or the test fixture and mouse over signals to view their values. We left out the fact that you can mouse over signals in the Verilog code as well as the schematic from the previous tutorials to avoid confusion. So, if you have not done so before, now is a good time to start using this technique.
11. View the contents of the Console window to see messages your test fixture has printed, and debug your module as necessary.

The steps above lay down basic steps for creating a test fixture. However, you may find that some modules are either too complicated or too simple to make creating a test fixture sensible. Perhaps watching the signals in the waveform is preferable and/or you do not want to take the time to develop a test fixture to sample outputs from a module. We are not providing you an easy way out of developing test fixtures, and we recommend you use test fixtures as often as you can. However, the steps below will help you to use the same techniques used for developing test fixtures to drive signals for modules to be viewed in a waveform (a **driver**).

1. Create a new Verilog source code file.
2. Add an output for every input to the module you will be driving (drivers do not need inputs).
3. Use an initial statement, as you did above, and assign the outputs to the values you want to drive the module's inputs with.
4. Save, and compile the file.
5. Follow the steps outlined for creating a test fixture to place this new file and the module to be driven in a new schematic.

6. Set the schematic containing this new file and the module to be driven to the top-level.
7. Initialize a simulation and open a new waveform.
8. As described in Tutorial #1, add the signals you wish to observe to the waveform, and start the simulation.

This may seem trivial after you have created a test fixture or two. However, this technique is very useful when simulating modules with several multi-bit signals. Additionally, you will find creating a driver is necessary for more complicated, pipelined designs that you may come into contact with in future courses. Figure 6 is an example of a driver used in CSE 467 to drive the inputs for a pipelined video card design.

```
initial
    begin
        masterclock = 0;
        reset = 1;
        #0 dout = 0;
        #0 writeenable = 0;

        #20 reset = 0;    // 20 clock periods

        #2000 dout = 72'h000000ff0000000000;
        #0 writeenable = 1;
        #2 writeenable = 0;

        #2000 dout = 72'h0a0000000010000000;
        #0 writeenable = 1;
        #2 writeenable = 0;

        #2000 dout = 72'h070aa1f55ffa880000;
        #0 writeenable = 1;
        #2 writeenable = 0;

    end

always
    begin
        #1 masterclock = ~masterclock;  // 20 ns clock
        ground = 0;
    end
```

Figure 6

The driver above drove signals to a pipeline that took 10 bytes of data on the front end. Calculations and other manipulations of the data occurred in several other modules before the final outputs were produced. Setting stimulators for every bit and computing delays before changing each bit would have been a daunting task without this driver.

Conclusion

You should have a general idea of how to create Verilog modules of your own, use your own Verilog modules in conjunction with Active-HDL parts, and create your own test fixtures and drivers. Even though this tutorial explains how to work in the Active-HDL environment, the concepts of test fixtures and drivers can be applied in all areas of design. We recommend you read the textbook on how to code in Verilog to learn how to use Verilog properly. No effort has been made to teach you Verilog in this tutorial, but the examples in this tutorial and previous tutorials will provide you with some guidance in creating your own modules and test fixtures. We encourage you to practice these techniques and thoroughly test your modules and your designs.