CSE370 Tutorial 3 - Introduction to Using Verilog in Active-HDL

Objectives

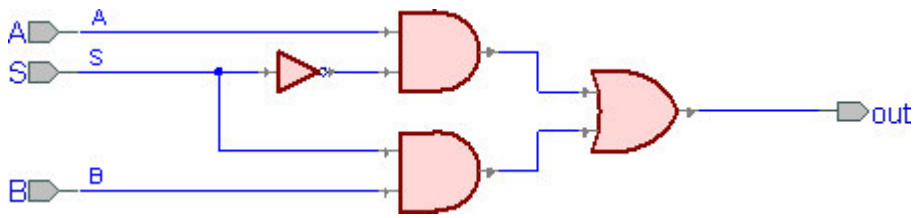
In this tutorial, you will learn how to write an alternate version of the full adder using Verilog, a hardware description language. This may be your first experience with Verilog, which allows you to describe designs using text rather than schematics. This tutorial will only cover very simple Verilog combinational logic modules that use only assign statements. Later you will learn more Verilog, but the way you use Verilog modules in Active-HDL will remain the same.
You will find that there are many tools and options that have been left out of this tutorial for the sake of simplicity. You may want to experiment with Active-HDL on your own. This will make you more proficient with Active-HDL, and you may find different methods that better suit your style, while still achieving the same design goals.

A Quick Introduction to Verilog

This is a very simple introduction to Verilog that will allow you to define simple combinational blocks using Verilog. Figure 1 show a simple schematic description of a 2-1 multiplexor along with the corresponding Verilog description of the multiplexor.

A Verilog module is defined by the module()/endmodule statements. A module definition is similar to a procedure definition in an ordinary programming language in that it defines a set of inputs and outputs and the functionality of the module. It is different since modules are not called like procedures; rather, they are "instantiated" or used in higher-level modules. Each input and output is listed in the module header, and then defined as an input or output by the input/output declarations. In this example, the inputs are A, B and S and the output is out.

Assign statements are used to define signal values as Boolean expressions. In this example, out is defined by the function **AS' + BS**, but must be written in Verilog using the AND operator ("&"), OR operator ("|"), the XOR operator ("^") and the NOT operator ("~"). It is important to remember that an assignment statement is identical to the corresponding schematic with gates wired to the inputs and outputs to define the Boolean function. In fact, assign statements are known as "continuous assignments" because, unlike assignment statements in a regular programming language, they are executed continuously, just like the corresponding gates in a schematic.



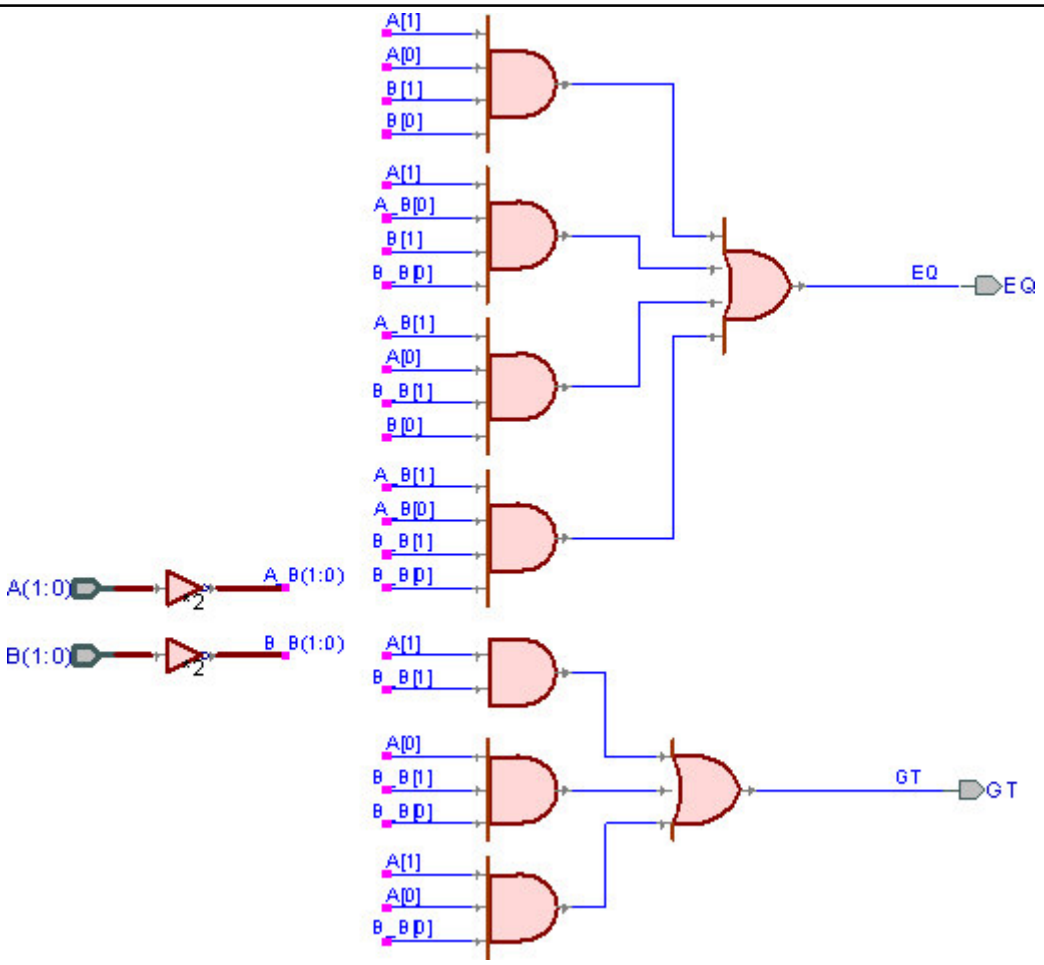

```
     // This is a simple Verilog implementation of the multiplexor
function
     module mux2 (A, B, S, out);
       input A, B, S;
       output out;
       assign out = (A & ~S) | (B & S);
```

```
    endmodule
```

Figure 1 – Verilog for a 2-1 multiplexor

Figure 2 shows a slightly more complex module that uses busses and wires. The inputs A and B are 2-bit busses and are declared as a vector of bits. Note well the declaration for busses in Verilog which differs from C or Java. Note especially the numbering from high-order bit to low-order bit.

Temporary signals, in this case the inverted inputs, which are used only inside the module are declared using the "wire" declaration. Inputs and outputs are just special wires; in fact, you will see redundant wire declarations in the Verilog files that Aldec generates.




```
// This is a simple Verilog implementation for a 2-bit comparator
     module compare2bit (A, B, EQ, GT);
       input [1:0] A, B;
       output EQ, GT;
       wire [1:0] A_B, B_B;        // Inverted signals for A and B
       assign A_B[0] = ~A[0];
       assign A_B[1] = ~A[1];
       assign B_B[0] = ~B[0];
       assign B_B[1] = ~B[1];
       assign EQ = (A[1] & A[0] & B[1] & B[0]) |
                   (A[1] & A_B[0] & B[1] & B_B[0]) |
                   (A_B[1] & A[0] & B_B[1] & B[0]) |
                   (A_B[1] & A_B[0] & B_B[1] & B_B[0]);
```

```
        assign GT = (A[1] & B_B[1]) | (A[0] & B_B[1] & B_B[0]) | (A[1] &
A[0] & B_B[0]);
      endmodule
```

Figure 2 – Verilog for a 2-bit comparator

Verilog is a lot more powerful than we have shown here. For example, there are easier ways to specify the comparator function, but we will wait until later to introduce these concepts. For now, think of Verilog as an easy alternative to drawing a schematic.

Creating a Verilog Module

We will now create a simple Verilog module for a full-adder. It will implement exactly the same functionality as the full-adder schematic, but do so using Boolean expressions rather than gates.

1. Add a new Verilog file to the design. To do this double-click on the "Add New File" in the Design Browser, then select the "Wizards" tab, and click on the "Verilog Source Code Wizard". Then, click OK.

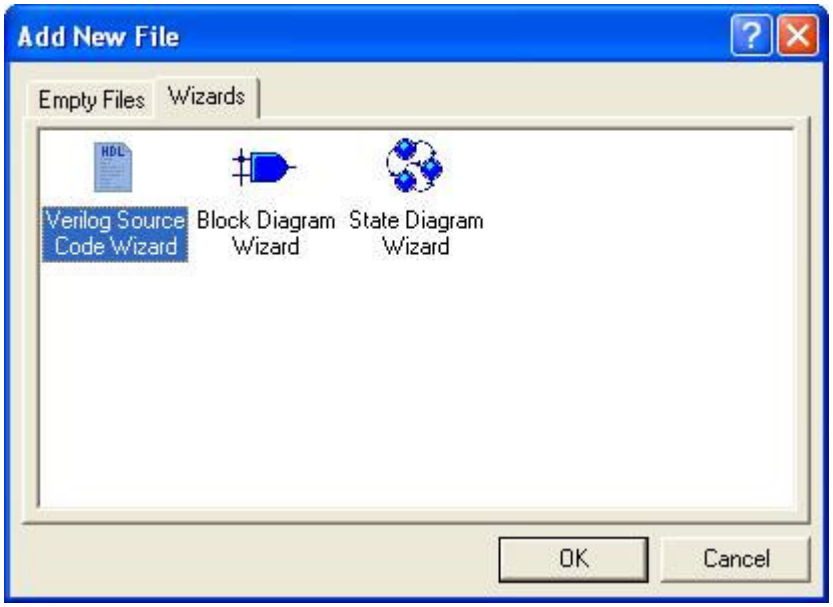
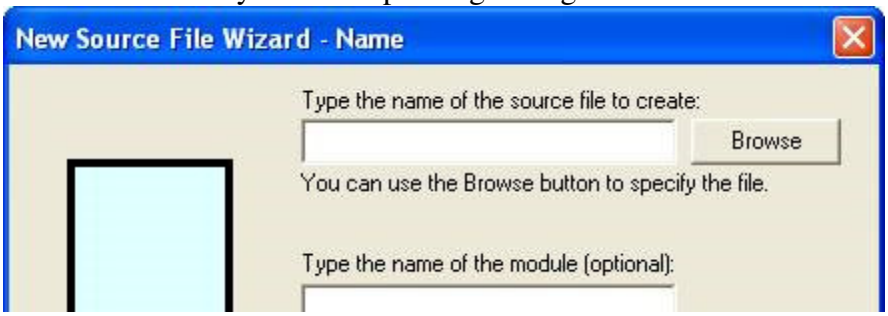


Figure 3

2. In the New Source File Wizard, make sure that the box is checked next to the "Add the generated file to the design" option. Click the Next button.

3. In the "Type the name of the source file to create" field, enter the name of your verilog file using a **descriptive** name. You must try to keep them distinct so as not to confuse both you and the tool. Do not use the same name that you named your design or workspace. Also, **do not fill in the name of the module**; leave this field blank. Active-HDL will use the source files name for the module by default. This just makes it easier for everyone to keeps things straight. Click the Next button (Figure 4).



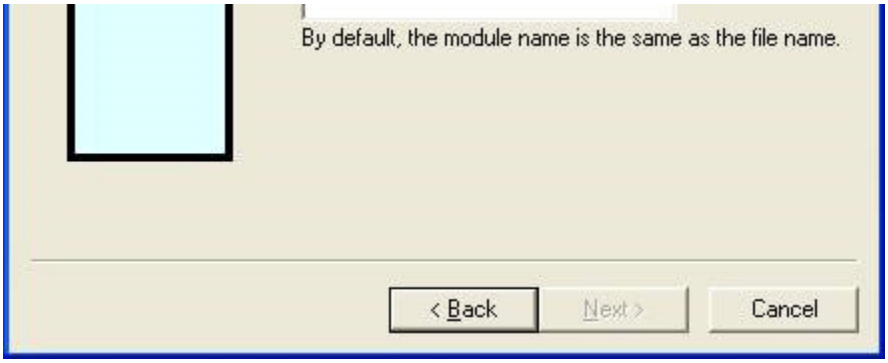


Figure 4

4. The next window that opens is where you add your input and output ports. These are the inputs and outputs of your module. To add ports, click the "New" button (see Figure 7). Choose whether you want an input or output port by clicking the desired radio button located under the "Port Direction" heading. In the "Name" field, type the name of the port. We will only be using 1-bit ports, so ignore the "Array Indexes" fields for now. Your screen should look like Figure 7. If so, then click the Finish button.

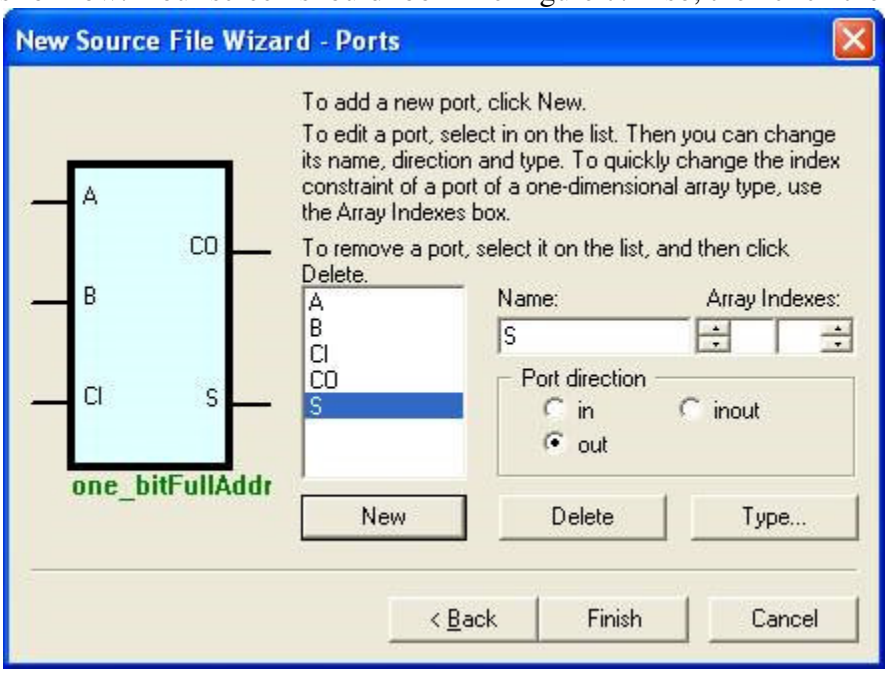


Figure 5

5. Aldec will open a Verilog file with the module, inputs, outputs, and wires already declared (see Figure 6). Locate the section of the file that contains the comment: " // -- Enter your statements here -- //". In this area you will insert the assign statements that define CO and S in terms of a Boolean function. Refer to the Verilog introduction above.

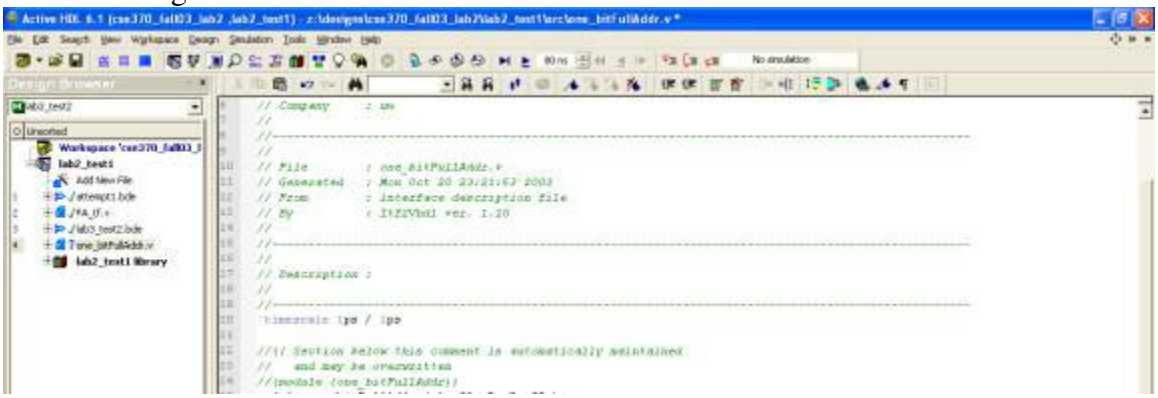


Figure 6

6. Save and compile your Verilog module. As in tutorial #1, a green checkmark should appear in Design Browser next to your Verilog module if it successfully compiles and a red "x" will appear if problems occur. When you compile a Verilog module, a symbol is created for it in your design library, which you can use in your higher-level schematics.

7. To verify your Verilog module follow the procedure described in Tutorial #2 for test fixtures but now replace the schematic version of the full adder with your new Verilog module. First, Open the Symbols Toolbox and find the name of your design to expand your design's part list. You should see a "Units without symbols" list. Expand it and find the name of your Verilog full adder. Select it, and notice that a block symbol appears in the bottom window of the Symbols Toolbox. Add this part to your test schematic by dragging the symbol from the bottom window. Notice that this component no longer appears in the "Units without symbols" list after it has been added to the schematic once. Now, simply disconnect the schematic-based full adder from the test fixture, and connect your new Verilog-based module. Test away.

Concluding Remarks

We have shown you how to make a Verilog module using the Wizard. You can also create an empty Verilog file and write the Verilog yourself, or copy an old Verilog file and edit it to be what you want. You can use whichever method is most convenient for you. Finally, you can edit the symbol that Aldec makes for you using the Symbol Editor. This allows you to move the pins around, change the size, and add text. For example, all the symbols in the lib370 library have been edited.

Comments to: cse370-webmaster@cs.washington.edu (Last Update:)