EECS 470 Lab 2 Assignment

Note:

- The lab should be completed individually
- The lab check off is due by **Friday, September 12th, 2025**

1 Introduction

In this lab assignment, you will practice testing, debugging, and analyzing modules. Specifically, you will:

- Write a comprehensive testbench to debug the 1-bit full-adder.
- Learn to use a waveform viewer (Verdi).
- Build a 64-bit full-adder by combining 1-bit full-adders.
- Write a non-comprehensive testbench for the 64-bit full-adder.
- Synthesize the 64-bit adder and analyze the resulting files.

2 1-Bit Adder: Testbench and Waveforms

For this section, we'll explore writing a comprehensive testbench for a 1-bit full-adder module and simulating it on a waveform viewer.

2.1 A Comprehensive Testbench

We've supplied you with a slightly obfuscated full-adder (`full_adder_1bit.sv`) and an incomplete stub testbench (`full_adder_1bit_test.sv`). The testbench already computes the correct sum and fails the module if an incorrect value is output. To finish the testbench you need to do two things:

1. Wire up the module – Open the testbench and finish wiring the signals into the module ports.
2. Write the comprehensive tests – Add assignments to the `initial` block and add eight tests to handle every combination of inputs for the module.

Once you've done this, run the testbench with `make sim` and identify/fix the bug in `full_adder_1bit.sv`.

2.2 Verdi Tutorial

To help debug complex bugs, Synopsys provides a debugging tool, called Verdi, for hardware description designs. This debugger looks much more akin to the debugger you may have used in a digital design course (ModelSim in EECS 270) than in a programming course.

To start the debugger, use the command: `make verdi`. Run that command now. This should open a window, which we will now refer to as the Verdi window. (Note: Verdi can often be painfully slow to start, just be patient).

We will now walk through the default panes of the Verdi window, but you can add more panes yourself through the Window » Window Manager... menu.

2.2.1 Module Instance Pane

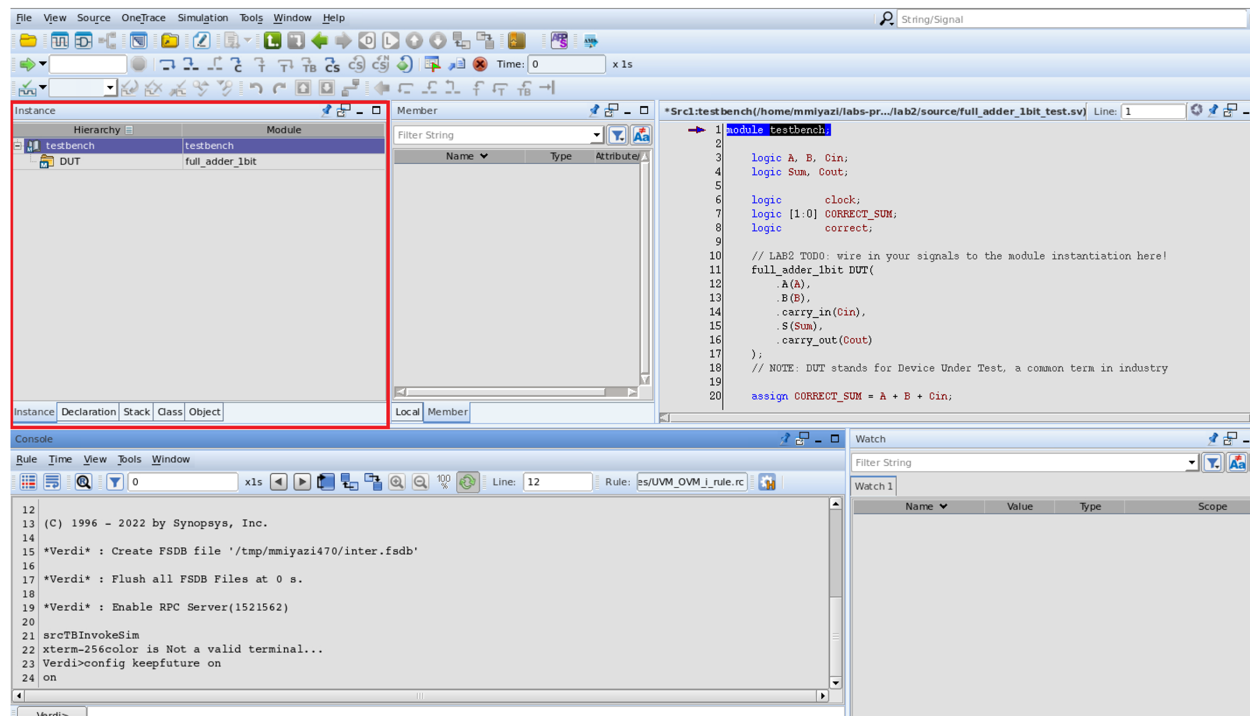

Figure 1: The Verdi Window with the Module Instance Pane highlighted

In fig. 1, the pane on the far left of the Verdi window is the Module Instance Pane. It contains the names of the modules you might be interested in, grouped hierarchically. You can use this list to select the modules you want to look at in other panes and views.

In our case, we have a testbench that instantiates a `full_adder_1bit` module. Expand the testbench (click the plus icon on the left) in the pane to see the submodule. You'll notice that the submodule is called `DUT`, which is not the name of the `full_adder_1bit` module, but rather the name of the instantiation in the testbench. Make sure to use meaningful names for instantiations when you implement your own designs.

2.2.2 Signal List Pane

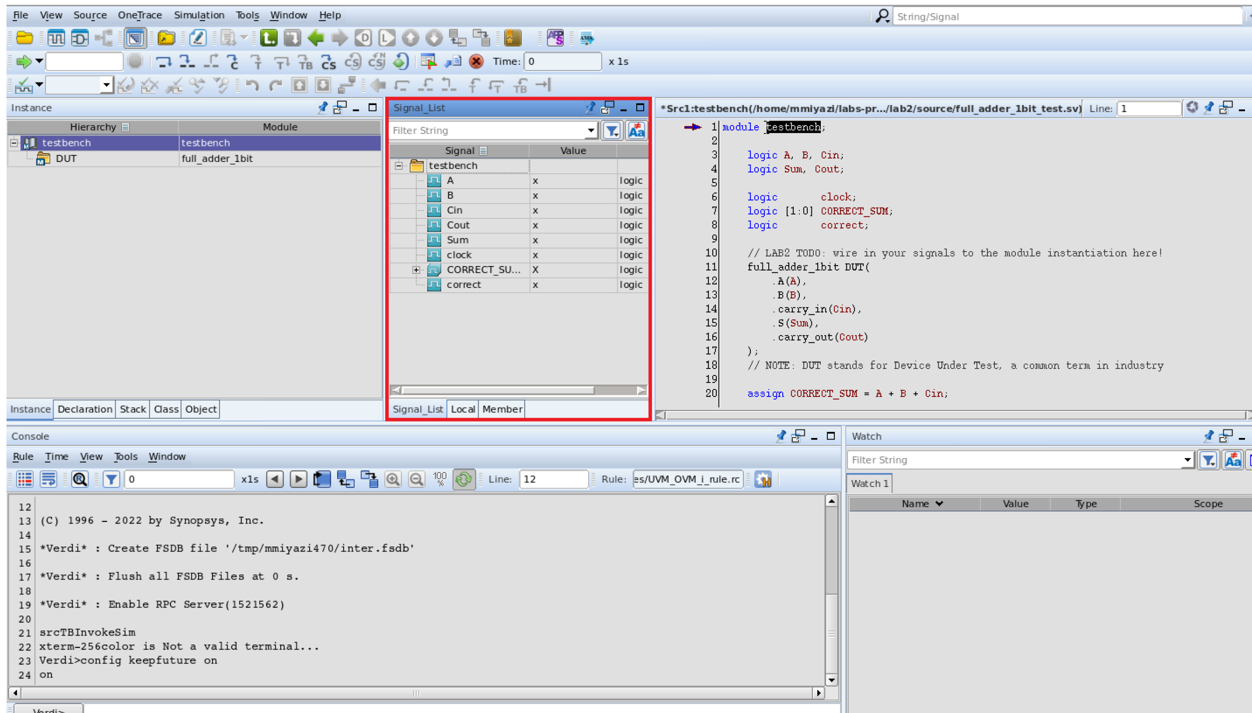

Figure 2: The Verdi Window with the Signal List highlighted

In fig. 2, the Signal List Pane is highlighted. You can add this pane through the Window › Window Manager... › Signal_List › Activate › Close menu.

Double-click on a module in the Module Instance Pane to list the signals in that module. From the signal list pane, select the signals whose waveforms you would like to view. You can select multiple signals by holding down ⇧ Shift and clicking to select a group or holding down Ctrl and clicking to select multiple individual signals. Select all the signals available in the testbench and right-click on the highlighted group, then go to Add to Waveform › New Waveform to open the Waveform Viewer. If, at some point later on, you need to add more signals to the Waveform Viewer, you can do so by selecting them here and selecting Add to Waveform › Add to [wave name].

2.2.3 Waveform Viewer

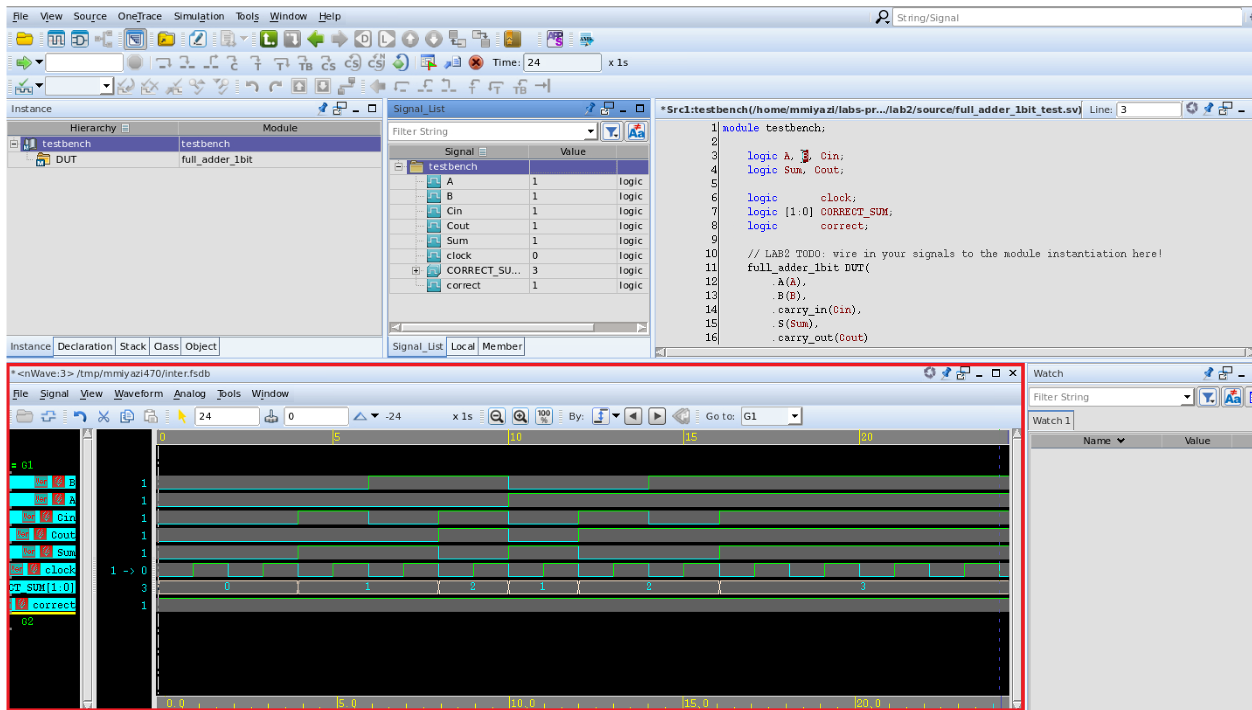

Figure 3: The Verdi Window with the Waveform Viewer highlighted

You will see a new window that shows waveforms, as in fig. 3. You need to go to Simulation » Run/continue or press F5 to start the simulation and populate the Waveform Viewer. At this point, if you added additional signals to this view, they would have no waveform, until you reran the simulator. This can be done by Simulation » Restart then Simulation » Run/continue or by pressing F5 twice.

The Waveform Viewer is intended to facilitate debugging by showing you how several signals change in relation to one another over time. The pane on the left-hand side of this window lists the names of the signals currently being displayed. The right-hand pane shows the signals themselves. At the bottom of this window, there is a scroll bar that moves through time. The Waveform Viewer toolbar has buttons to zoom in, zoom out, and scale the waveform to fit in the viewer.

Signals in the Waveform Viewer can be displayed several different ways, largely related to the 4 state logic used in Verilog. The "good" signals `0` and `1` are shown as green signal lines with transitions between the two values as appropriate. The "bad" signals `X` and `Z` show up as a red block where the signal should be and a yellow line half way between `0` and `1`, respectively. Verdi is a great tool for pinpointing "bad" signals in a complex design.

Signals can be dragged around using the middle button of your mouse. You can also remove signals by highlighting them and then pressing the delete key. You can also add custom signals that are a logical operation of other signals by going to Signal » Logical Operation....

For the lab check-off, you must show an instructor the populated Waveform Viewer. To capture and save the waveform, go to File » Capture Window » Save as... in the Waveform Viewer.

2.2.4 Saving and Restoring Configurations

We want to save the Waveform Viewer configuration so we can easily reload them when we run the Waveform Viewer again. It may not seem to be a big deal right now, but eventually when you're working with hundreds of signals, only a few of which are related to a problem you're trying to debug, it can be annoying to try to

find and place them on the waveform viewer every time. To do this, on the Waveform Viewer go to File > Save Signal.... Type in a filename, like `signal.rc` or, in the future, any name you want. It will be useful to keep the names descriptive as there may be different sets of signals to look at for different parts of a module you're testing. Now exit Verdi.

To re-launch Verdi, first run `make verdi`. Now, in the Verdi Window, go to Tools > New Waveform... > File > Restore Signal. Select your `signal.rc` file and press OK. The Waveform Viewer, complete with the original signals, should appear. Run the simulation again. Here we can see, that everything is back to how it was before exiting Verdi.

2.2.5 Conclusion

This concludes our *very* brief introduction to Verdi, but there are many of other useful features such as:

- **Adding breakpoints and markers:** Verdi allows you to set breakpoints in your simulation, pausing execution at specific points to inspect the state of your design. Markers can be used to highlight important events or times within the waveform for easy navigation and reference during debugging.
- **Stepping forward and backward through code:** In addition to breakpoints, Verdi provides the capability to step through your code line by line, either forward or backward. This feature is invaluable for closely examining how your design behaves at each stage of execution, allowing you to pinpoint the exact moment when a bug occurs.
- **Viewing hardware schematics post-synthesis:** Verdi offers the ability to visualize your design's RTL schematics. This is especially useful after synthesis when you need to understand the physical implementation of your design. The schematic viewer helps you trace signals, understand the data flow, and verify that the synthesized hardware matches your intended design.

3 64-Bit Adder: A Harder Testbench and Synthesis

Using the 1-bit adder, you will now build a 64-bit adder. The file `full_adder_64bit.sv` contains a module stub: `full_adder_64bit`.

Use an array of 1-bit adders to complete the module as described in the lab presentation. (This should take no more than 5 minutes; feel free to get help from a neighbor!)

Then, update the Makefile to compile this new module (change "`1bit`" to "`64bit`" in the `TESTBENCH`, `SOURCES`, and `SYNTH_FILES` Makefile variables).

3.1 Where Comprehensive Tests Fail

We have provided you with a slightly more complete testbench in `full_adder_64bit_test.sv`. It initially implements a comprehensive testbench similar to what you wrote for the 1-bit adder, but you're going to need to change it. To understand why, first just run the testbench:

Run the testbench with `make sim`.

Hint: Use `Ctrl-\` to force-quit an unresponsive simulation.

Now read through the testbench and try to understand what went wrong. It has a lot of comments, but you don't need to read them all. You will find a pair of for-loops that implement the comprehensive testing. Read through them and understand why they won't work for a testbench written by humans with finite lifetimes. Feel free to discuss with a neighbor.

Comment out the for-loops in the testbench.

Read the "`Test Specific Edge Cases`" section, and add tests there. Try to test edge cases or interesting values. The most interesting values in any test are 0, 1, -1, and infinity (aka MAX). Add a few tests for combinations of those and for some other simple numbers.

You also need to fix the `compare_correct_sum` task in the testbench. Look at the 1 bit testbench and try to see what's missing in the 64-bit test.

Run the testbench and make sure your module passes.

3.2 Synthesis

3.2.1 The Synthesis Tool: `dc_shell`

In this class we require that your hardware designs really represent hardware and we judge your final project on the overall speed of your design. To accomplish this you will need to synthesize your design. The synthesis tool attempts to create an actual circuit level implementation of your Verilog design description. This circuit level design is actually a Verilog file itself, but it is structural Verilog and uses a library of standard cells that another tool can lay out on a real chip. One benefit of the output being a Verilog file is that we can simulate the circuit level design in the same way we simulate your behavioral design.

Thus we can test your synthesized design and if it does not behave properly your design is not considered to be synthesizable and is therefore incorrect (though we do give partial credit).The clock speed of the circuit that it generates will be the clock speed we use for your design.

We interact with the synthesis tool, Synopsys Design Compiler, through a script of options and commands written in the Tool Command Language, which is abbreviated Tcl (`*.tcl`) and pronounced either T-C-L or like the word "tickle." We have provided you with a synthesis script called `470synth.tcl`.

The clock period is set in the Makefile and passed to the tcl script, Ctrl+F for `CLOCK_PERIOD` in the Makefile to change it. A higher clock period will yield faster synthesis times, while a lower one will take longer because it requires more effort on the part of the synthesis tool. A clock period set too small may not even be possible (hint for the next section).

Run `make syn` to synthesize and simulate the synthesized design. This will produce multiple files. Please open them and read them.

1. `full_adder_64bit.vg` – This file contains the structural Verilog that the synthesis tool generated for your design and design constraints.
2. `full_adder_64bit.chk` – This file contains errors if something went wrong. Many of the warnings are ignorable, but you should review them when investigating synthesis bugs.
3. `full_adder_64bit.rep` – This file contains the report for your design. The `470synth.tcl` file has been configured to provide a report on area consumption and timing. The timing report is how you will identify the critical path and whether your design is able to run at a certain clock period.
4. `full_adder_64bit.ddc` – (Don't open this one) This file contains the proprietary Synopsys representation of the synthesized design. It can be included like a Verilog design file in synthesis, either to be optimized again or as a black box.

3.2.2 Fixing Violated Slack

Once you've run synthesis (`make syn`), run `make slack` to grep the generated .rep file. This will find the reported "slack" for the module. It's been violated! For the check-off you should be able to answer what this means and how you can fix it. Fix it, and be able to show that slack is no longer violated.

Ensure that your 64-bit testbench also gives a pass to the synthesized module.

4 Submission

For lab 2, be ready to show your 1-bit adder Verdi waveform, your 64-bit testbench, the output of `make slack` on the 64-bit module without violation, and your answers to the following questions in a `.txt` file.

1. How can you write an exhaustive testbench with a for loop? Answer in at most two sentences.

2. What was wrong with the 64-bit testbench? Answer in at most two sentences.
3. What are your new test cases for your 64-bit testbench? Answer in a bulleted list.
4. What is the clock period of the 64-bit adder?
5. Between what input(s) and output(s) is the critical path? Look at `full_adder_64bit.rep` file for more details on timing.
6. Pick 3 different types of gates from the synthesized netlist in `full_adder_64bit.vg` and describe what they are in at most one sentence each.

Place yourself on the help queue during lab or office hours once you're confident you've completed the lab satisfactorily. Turn in your check-off to Gradescope by the end of the day of next week's lab.