

LAB 6 – Testing the ALU

Goals

- Learn how to write testbenches in Verilog to verify the functionality of the design.
- Learn to find and resolve problems (bugs) in the design.

To Do

- You will write a Verilog testbench that will verify that your ALU from Lab 5 works correctly.
- We will provide an ALU code that contains bugs. Using the same testbench you used for testing the ALU from Lab 5, you will simulate the buggy ALU to find the bugs and correct them.
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- To complete the lab, you have to show your work to an assistant before the deadline. The required tasks are clearly marked with gray background throughout this document.
- You will have an additional exercise in the report.

Introduction

We have seen that it is impossible in practice to verify the functionality of the ALU designed in Lab 5 using direct observation like we did in the earlier lab exercises. What we need is a method that automatically checks whether the circuit works correctly. This method is usually called functional verification.

The basic idea of software-based functional verification is not fundamentally different than what we did so far to check the correctness of a circuit (i.e., manually checking the output for different inputs). In this lab, we will perform functional verification in an automatic way. We will implement a Verilog module, i.e., testbench, that would 1) apply arbitrary inputs to the circuit that we are testing (unit-under-test (UUT)) and 2) check the correctness of the output. The inputs to test with could be either generated inside the testbench module using behavioral modeling in Verilog or loaded from a file as test-vectors, which is a set of input data and the expected result. If the output of the UUT matches the expected output for all test cases (e.g., all possible inputs), we could say that the implementation is correct. Making the functional verification automatic enables more test cases to be checked in a short amount of time compared to the manual effort. It is especially important when the possible input space is too large.

There are additional advantages in performing functional verification in software. On an FPGA, we can only observe the outputs of the top module. In contrast, when we use a software tool that *simulates our circuit*, we also have access to all internal modules. It makes tracing the bug much easier by reducing the testing time significantly.

So far, we have used Verilog to describe the actual circuit. As you have learned in class, we can also use Verilog to describe testbench modules that:

- instantiate the UUT, i.e., the module that you would like to test
- define inputs to test the UUT with
- collect the outputs
- compare the outputs with the expected results

In this exercise, we will implement a testbench in Verilog and simulate our ALU circuit from Lab 5. If there are mistakes in the circuit, you can find what went wrong and correct the mistakes.

Preparation¹

In this lab exercise, we will continue using the Verilog description of the ALU from Lab 5. You are expected to finish that exercise before starting with the testbench.

Download the .zip file using the link below. The file includes a template/example for a testbench file, the template for the test-vectors, and a Verilog description of an ALU, which contains some bugs.

https://safari.ethz.ch/digitaltechnik/spring2019/lib/exe/fetch.php?media=lab6_files.zip

Part 1 - Expected Results

Before we can start with writing our testbench, we need to prepare a set of inputs that we know the expected results for. Typically, for large projects, we could use a golden-model to find the expected result for a given input. A golden model could be any implementation that we know is correct. For example, it could be another previously-tested Verilog implementation of the circuit or a program written in a high-level language (e.g., Java, C++, Matlab, Python). We could also use a systematic way of generating several inputs so that the circuit is verified thoroughly².

In this exercise, you will be given a set of inputs for the ALU designed in Lab 5. Determine the correct ‘result’ for each of these inputs and, using a text editor, enter them to the file ‘testvectors_hex.txt’ that we provide. This file contains inputs for the ALU and your task is filling in the expected value of ‘result’ in hexadecimal notation.

Note that our ALU has an additional single-bit output, called ‘zero’. The expected value of that output can be easily determined based on the ‘result’. Thus, we can directly set its expected value within the testbench. If the expected value of ‘result’ is all zeros, the expected value of the signal ‘zero’ should be one.

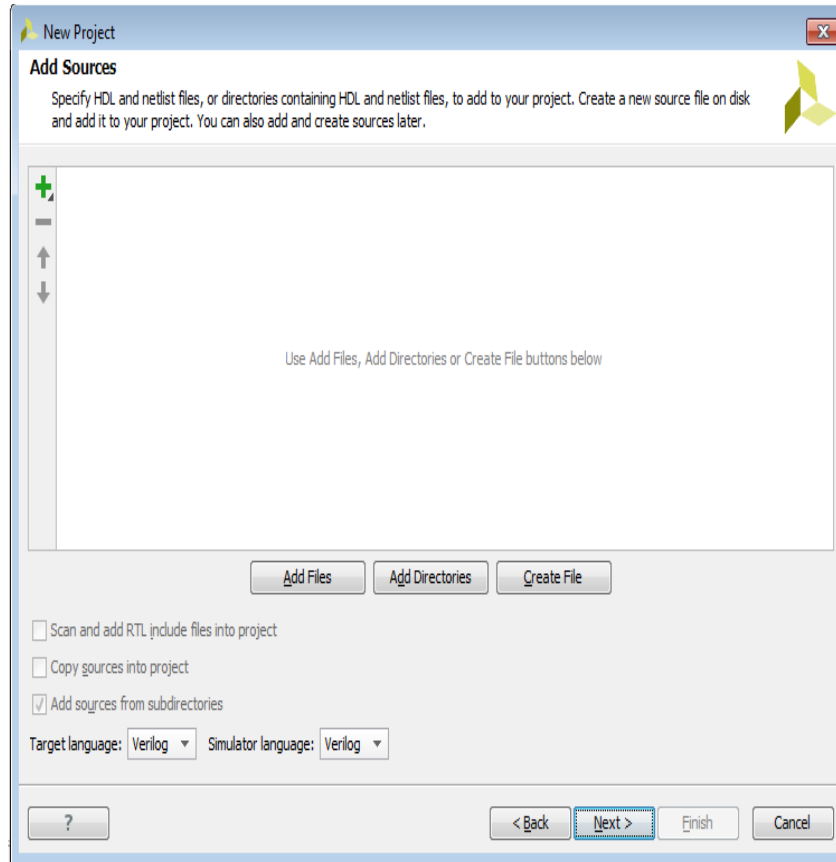
¹ If Vivado does not find the constraints file, you have to check “set as target constraints file” on the desired .xdc.

² As mentioned earlier, simulating each and every possible input is not practical. Therefore, it is important to find a subset of all inputs that verify most of the functionality. Inputs are chosen so that all functions of the circuit are tested, all limits and special cases are covered.

Part 2 - Preparing the Testbench

In this part, we will create a testbench module and implement our testing logic.

Create a new RTL project and make sure Verilog is selected as the simulator language:



Since you want to test the ALU you implemented in Lab 5, you need to import your ALU module as a source. Do that within the dialog you see above. Then, keep clicking “Next” and complete project creation by clicking “Finish”.

After creating the project, you can start implementing the testbench. For this exercise, we provide you a testbench template that is almost complete.

To use the provided file, you first have to add the file to the project. Click “Add Sources” on the left-hand side and select “Add or create simulation sources”. Then browse to *ALU_test.v* that you downloaded.

The new file will not be immediately visible in the project hierarchy. This is because, by default, the hierarchy only shows the “Design Sources”. To view “Simulation Sources”, expand the corresponding folder. After adding the ALU from Lab 5 as a source, you should see both the ALU and the testbench in the project hierarchy.

There are a few modifications that you need to make in the testbench in order to simulate the ALU properly.

Open the testbench file *ALU_test.v* and make the following modifications. These modifications are also indicated as comment lines in the testbench file.

1. Declare an array that is big enough to hold all our test cases and can store the inputs *aluop*, *a*, *b* and the expected value of the output *exp_result*. Declaring arrays in Verilog is not different from declaring multi-bit signals. For example, to declare a 4-element array of 5-bit regs, you need to write:

```
reg[4:0] example_array [0:3];
```

2. Add a statement that reads the contents of the ‘testvectors_hex.txt’ file into the array declared above. Hint: You can use the *readmemh* function (check online documentation for how to use). When reading the file, you can either specify the full path to the file or add the file to the Vivado project and directly use the file name. Note that the last line of the file is “X”, which is needed to specify the end of the file for the testbench.
3. Generate the value of *exp_zero* from the *exp_result*. A simple assign statement should be sufficient to do this.
4. In the testbench, instantiate your ALU from Lab 5. Ensure that you connect the test signals correctly to the module you instantiated.

So far, we developed a testbench that will apply the vectors in the “testvectors_hex.txt” file and check the actual outputs of our ALU against what we expect. Now, it is time to simulate and see the results.

Part 3 - Simulating the ALU

There are many commercial software-based simulators that can be used to simulate Verilog circuits. In this exercise, we use the built-in simulator from Vivado.

Make sure that the *ALU_test* module on the hierarchy is selected as “top simulation module”. On the *Flow Navigator* window, there will be a ‘Simulation’ section. Expand ‘Run Simulation’ and select ‘Run Behavioral Simulation’.

If everything works fine, you should see a window similar to the one below. In order to view the entire simulation period, you may have to right-click the waveforms window and select ‘Full View’:

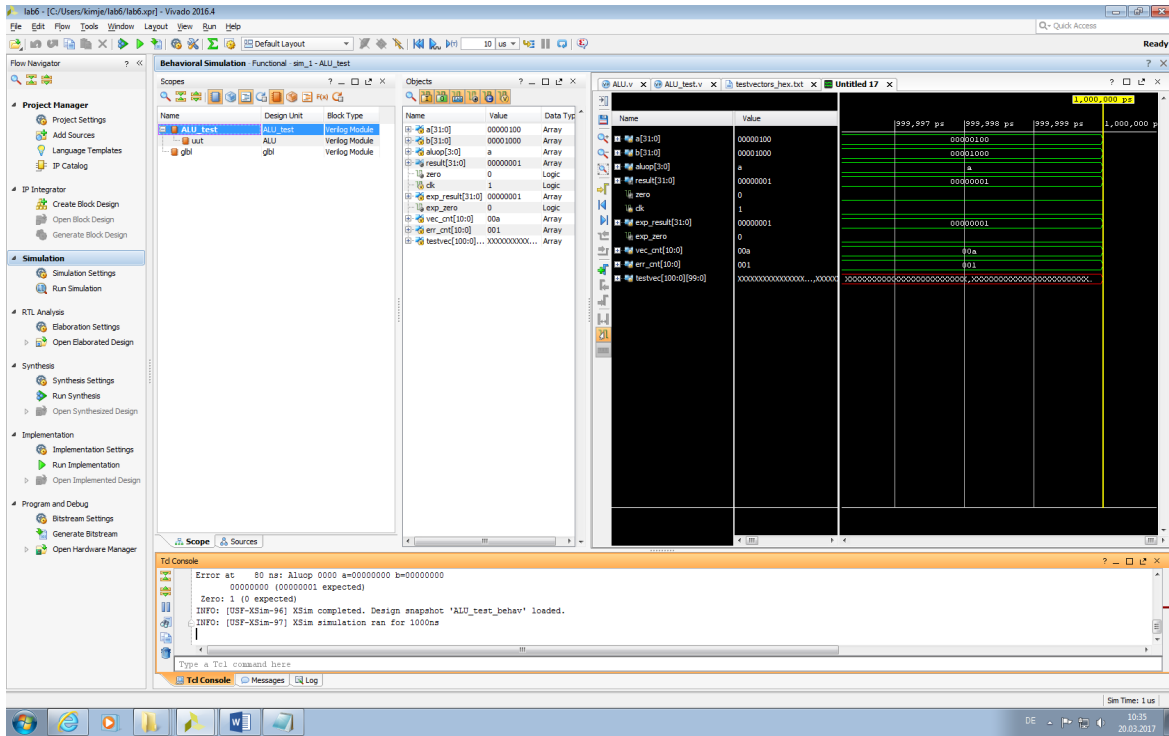


Fig.1 Vivado main window

The bottom part of Vivado contains the *Tcl console*. This is very important, as the *\$display* messages within the testbench will be displayed in this window³. If for some reason the simulator did not start, it most probably means that the simulator has encountered an error while compiling your Verilog code. The most common cause for this is a syntax error in your testbench. Scroll up on the console to find the reason why the code did not compile. Exit the simulator, correct the error in the editor and restart the simulator by selecting ‘Run Behavioral Simulation’.

Since in the testbench we have a *\$finish* statement, the simulator will automatically stop once the *\$finish* is reached⁴.

In case you want to simulate for a longer time, on the top part of the Vivado window you can click on ‘Run all’ or press **F3**



so that the simulator continues the simulating.

Examine the console, make sure that the circuit is working correctly.

³ If you look carefully at Fig.1 you will see that the simulation has completed with 1 error. You could scroll up and see the error messages displayed during the simulation that will help you identify the problems.

⁴ This happens when all the test-vectors have been exhausted.

Part 4 - Debugging Problems

Using a simulator can help you locate the problems in your circuits. You can not only observe the outputs but the state of all internal variables as well.

Now, your task is performing a simulation to find and correct bugs in a Verilog module. The ALU we provide, 'bad_ALU.v', is supposed to work the same way as your ALU, but it contains some intentional mistakes that result in incorrect behavior in some cases.

Add the source 'bad_ALU.v' into your project. Modify your testbench to instantiate bad_ALU instead of your ALU. Simulate the circuit and see the cases that result in errors.

There should be at least 7 errors (in 12 test-vectors). The console will display all cases that resulted in an error. In addition to the console, you can also use the waveform. By default, the waveform window will display all values with a particular radix that depend on the type of the signal. To change the radix, you can right-click on the signal and hover over *radix*. You can select multiple signals and quickly change the radix of all of them at the same time.

By selecting the name of your instance (typically called *uut*) in the hierarchy browser, you will be able to select internal signals of the *bad_ALU* in the objects window. This can be seen in the figure below.

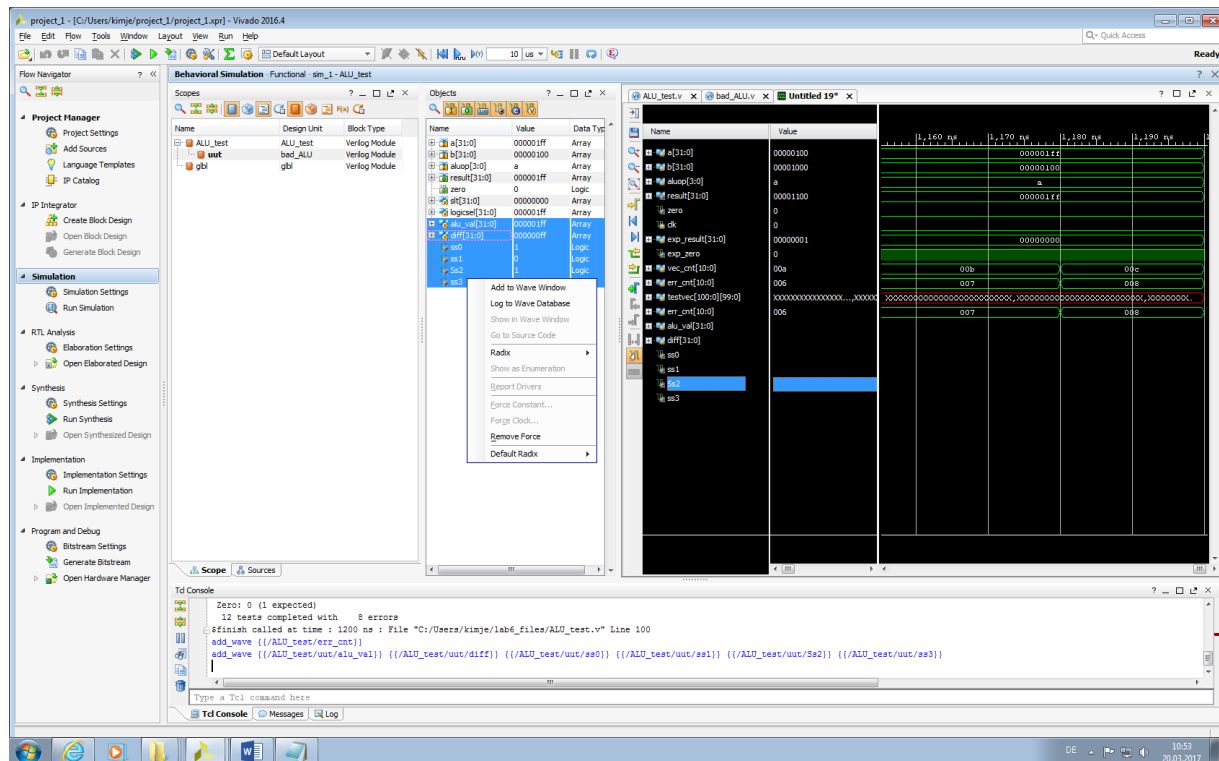


Fig.2 uut is highlighted in Vivado, and internal signals of the bad_ALU are selected

You can add any signal into the waveform by right-clicking on the signal and then clicking "Add to the Wave Window". The waveform viewer will not immediately display values for the signals you add. To see the values of the newly added signals in the waveform, you

should restart the simulation to let the simulator sample these new signals. You can restart the simulation by clicking the button we show below, which is located at the top of the window in Vivado, or pressing the shortcut **Ctrl+Shift+F5**:



After restarting the simulation, you need to run the simulation again. Now, you should be able to see all the waves.

Find and correct the errors. The code should be very close to working and should not require a major re-write, but just a few small corrections⁵. Show an assistant the working circuit.

Is there anything else wrong with the Verilog code of bad_ALU? Is it possible that the Verilog code is not suitable for synthesis even though it simulates correctly? (Hint: Look in the Design Summary to compare the performance and efficiency of the circuit with your ALU implementation in Lab 5.)

Last Words

You have just used a simulator to verify and correct a Verilog module. A simulator is able to run many thousands of test-vectors in a short period of time and is therefore much faster than verifying all outputs manually. Additionally, the simulator allows peeking inside the circuit allowing you to observe any internal nodes during operation.

However, the simulator alone is insufficient to test the circuit. You need a testbench that can automatically check if the outputs are correct, and most importantly, a way to generate meaningful test-vectors and corresponding expected responses. In practical designs, developing testbenches and automated verification flows are an important part of a digital design. Typically, more than half of the design time is spent in verification.

⁵ As the name implies the code is 'bad'. Do not take this code as an example of how to write Verilog.