LAB 6 – Testing the ALU

Goals

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

To Do

- You will write a Verilog testbench to 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 find and correct the bugs in the buggy ALU.
- 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 must show your work before the deadline, following the instructions provided by the assistant assigned to your group. There is nothing to hand in. The required tasks are clearly marked with gray background throughout this document.**

Introduction

In Lab 5 we learned that it is impractical to use direct observation, like we did in earlier labs, to verify the functionality of complex circuits. Instead, this requires a method that automatically verifies whether the circuit works correctly. This method is usually called ***functional verification***.

The basic idea of software-based functional verification is not different from what we have done so far to verify the correctness of the circuits we have designed (i.e., manually checking that the output is correct for multiple different inputs). However, in this lab, the functional verification will be performed *automatically*. To achieve this, we will implement a special kind of Verilog module, called a ***testbench***. The testbench will:

1) **Apply arbitrary inputs to the circuit under test.** We will call this circuit the *unit-under-test (UUT).*
2) **Check the correctness of the output.** The test inputs may either be 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 putting in the equivalent effort manually. This is especially important when the input space is large.

There are additional advantages associated with 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. This makes tracing bugs much easier and significantly reduces the testing time.

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 with which to test the UUT;
- 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.

Preparation

In this lab exercise, we will continue using the Verilog description of the ALU from Lab 5. **You are expected to finish Lab 5 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/spring2020/lib/exe/fetch.php?media=lab6_files.zip

Part 1 - Expected Results

Before we can start writing our testbench, we need to prepare a set of inputs that we know the expected results for. For large projects it is typically possible to use a ***golden model*** to find the expected output 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 such as Java, C++, MATLAB or Python. We could also use a systematic way of generating several inputs so that the circuit is verified thoroughly[1].

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. Your task is to fill in the expected value of 'result' in hexadecimal notation.

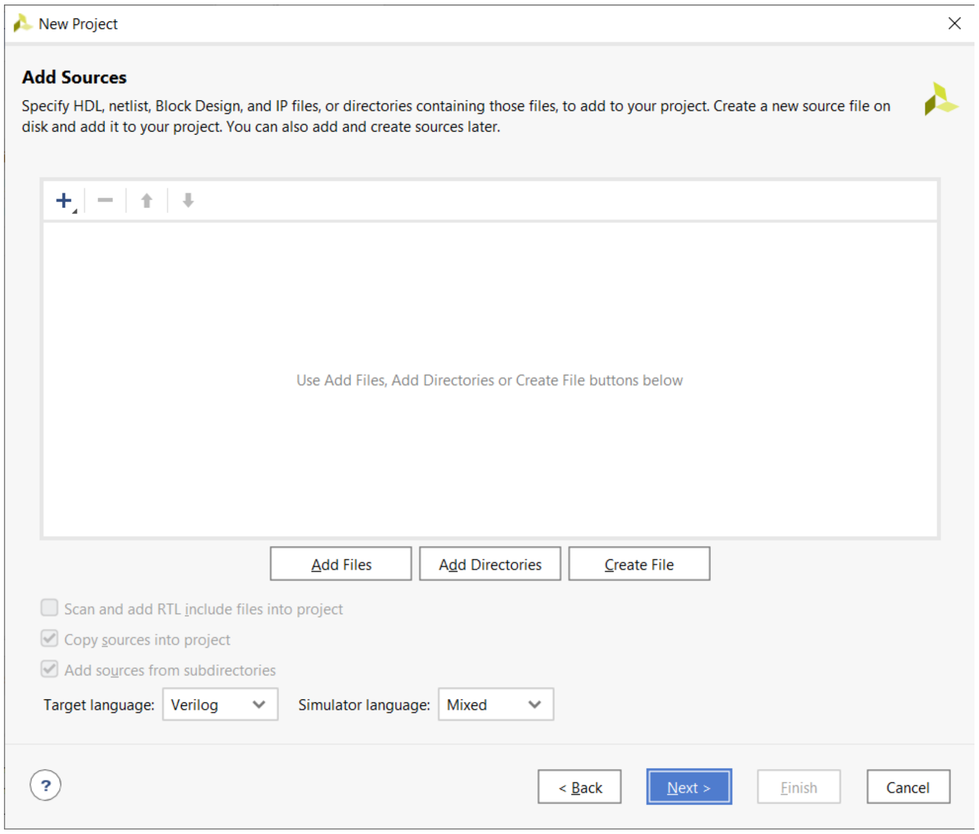
[1] Simulating each and every possible input is impractical. Therefore, it is important to find a subset of all inputs that verify *most* of the functionality. Inputs are chosen so that all functions of the circuit are tested, and all limits and special cases are covered.

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.

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:


New Project

Add Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create sources later.

Use Add Files, Add Directories or Create File buttons below

- [ ] Scan and add RTL include files into project
- [x] Copy sources into project
- [x] Add sources from subdirectories

Target language: Verilog   Simulator language: Mixed

Since we want to test the ALU you implemented in Lab 5, you need to import your ALU module as a source. *We recommend that you check the option “Copy sources into project” when doing so.*

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 with 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.

You must make a few modifications in the testbench to simulate the ALU properly.

Open the testbench file ALU_test.v and make the following modifications. These modifications are also indicated as comments 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 must 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 for documentation on how to use it). 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 its file name.
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 to see. Now it is time to simulate it and see the results.

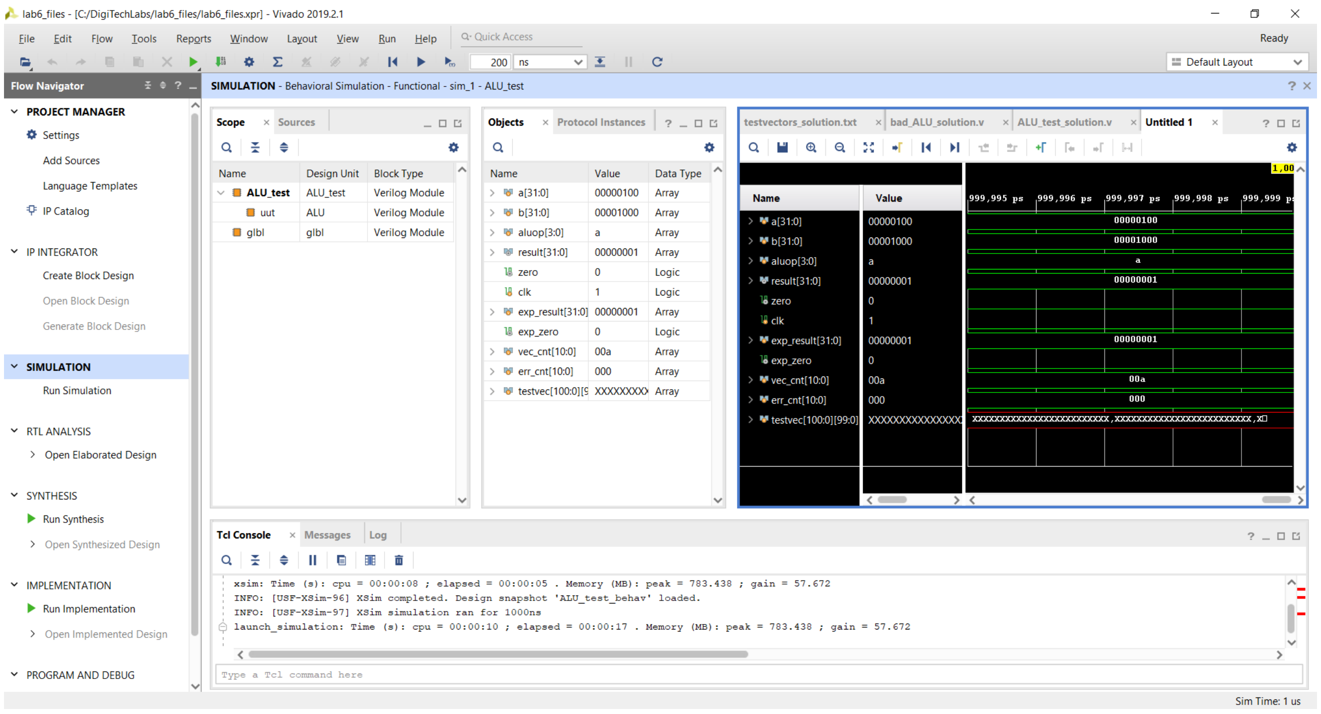
Part 3 - Simulating the ALU

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, look for 'Simulation'. 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':



Scope Name	Design Unit	Block Type
ALU_test	ALU_test	Verilog Module
uut	ALU	Verilog Module
glbl	glbl	Verilog Module

Name	Value	Data Type
a[31:0]	00000100	Array
b[31:0]	00001000	Array
aluop[3:0]	a	Array
result[31:0]	00000001	Array
zero	0	Logic
clk	1	Logic
exp_result[31:0]	00000001	Array
exp_zero	0	Logic
vec_cnt[10:0]	00a	Array
err_cnt[10:0]	000	Array
testvec[100:0][9	XXXXXXXXX	Array

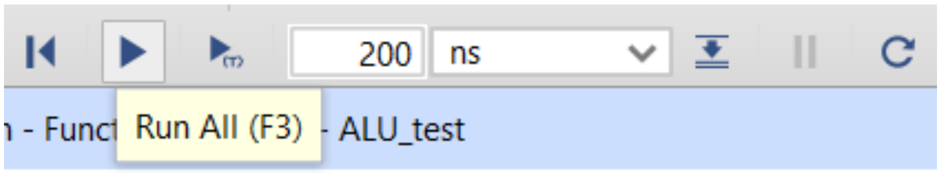
```
xsim: Time (s): cpu = 00:00:08 ; elapsed = 00:00:05 . Memory (MB): peak = 783.438 ; gain = 57.672
INFO: [USF-XSim-96] XSim completed. Design snapshot 'ALU_test_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:10 ; elapsed = 00:00:17 . Memory (MB): peak = 783.438 ; gain = 57.672
```

Figure 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 window. If the simulator did not start, it is most likely because it encountered an error while compiling your Verilog code. The most common cause for this is a syntax error in your testbench. If this happens, scroll up in the Tcl Console and look for ***"ERROR:"*** messages to help you troubleshoot the problem(s). Once you know what it is, 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 *$finish* statement, the simulator will automatically stop once *$finish* is reached.[2]

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**). The simulator will then continue.



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

[2] This happens when all the test-vectors have been exhausted.

Part 4 - Debugging Problems

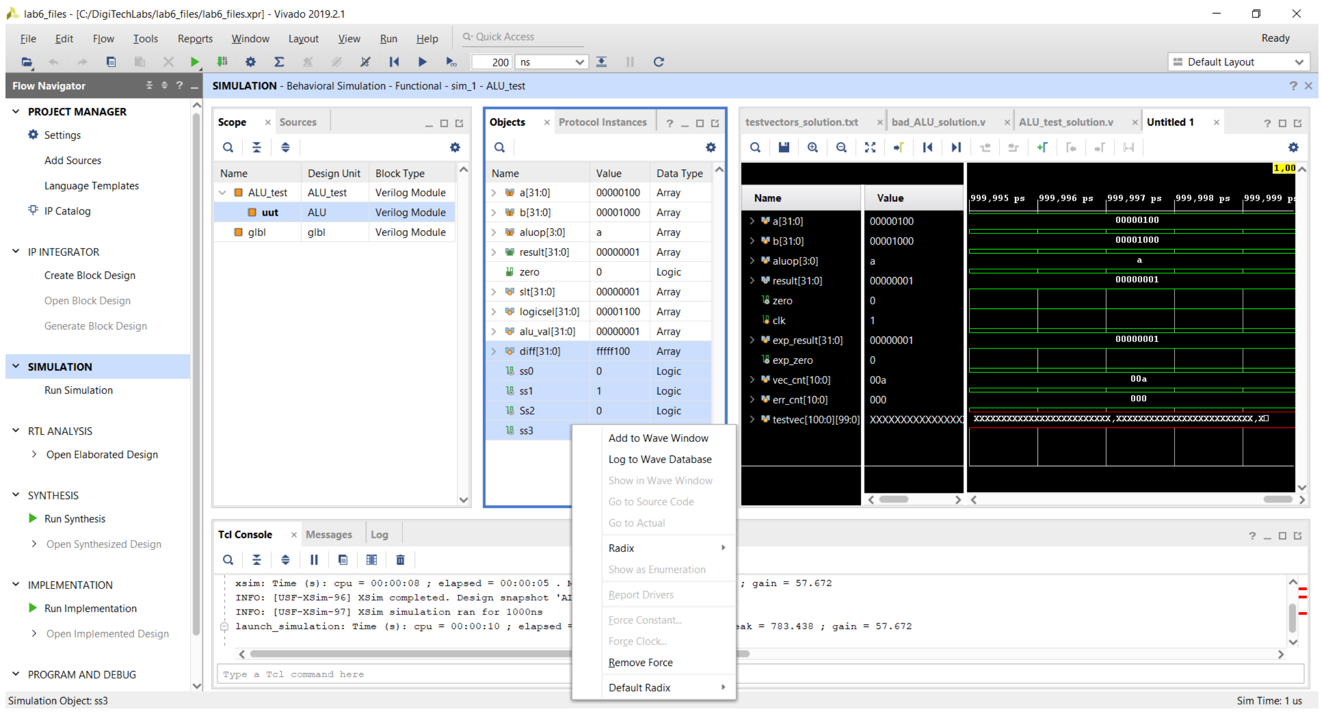
Using a simulator can help you locate the problems in your circuits. You can not only observe the outputs but also the state of all *internal* variables.

Now your task is to perform 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* (also called *base*) that depends 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 to 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.



Name	Design Unit	Block Type
ALU_test	ALU_test	Verilog Module
uut	ALU	Verilog Module
glbl	glbl	Verilog Module

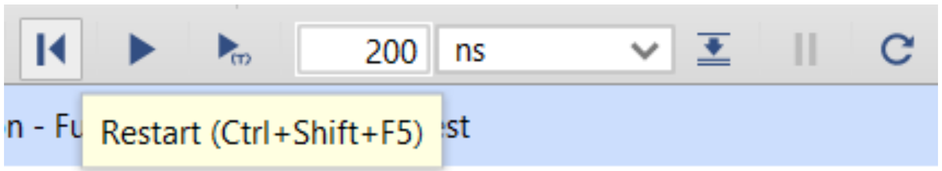
Name	Value	Data Type
a[31:0]	00000100	Array
b[31:0]	00001000	Array
aluop[3:0]	a	Array
result[31:0]	00000001	Array
zero	0	Logic
slt[31:0]	00000001	Array
logicsel[31:0]	00001100	Array
alu_val[31:0]	00000001	Array
diff[31:0]	fffff100	Array
ss0	0	Logic
ss1	1	Logic
Ss2	0	Logic
ss3		

Name	Value
a[31:0]	00000100
b[31:0]	00001000
aluop[3:0]	a
result[31:0]	00000001
zero	0
clk	1
exp_result[31:0]	00000001
exp_zero	0
vec_cnt[10:0]	00a
err_cnt[10:0]	000
testvec[100:0][99:0]	XXXXXXXXXXXXXX

```
xsim: Time (s): cpu = 00:00:08 ; elapsed = 00:00:05 . M                 ; gain = 57.672
INFO: [USF-XSim-96] XSim completed. Design snapshot 'AI
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:10 ; elapsed =                 eak = 783.438 ; gain = 57.672
```

Figure 2. *uut* is highlighted in "Scope"; the *bad_ALU's* internal signals 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 by pressing the shortcut **Ctrl+Shift+F5**):



After restarting the simulation, you will need to run the simulation again. Once you have done so 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 corrections[3]. Remember to show the working circuit to your assistant.

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 you to look inside the circuit to monitor internal signals 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 their corresponding expected responses. In practical designs, developing testbenches and automated verification flows are an important part of the digital design process. Typically, much more than half of the design time is spent in verification.

[3] As the name implies, the code is *bad*! Do not take this code as an example of how to write Verilog.