Experiment 2 - Debugging Combinational Designs

Objectives

The purpose of the second laboratory work is to introduce you to the concept of debugging, specifically debugging combinational designs. Throughout this preliminary work, you will learn basic debugging methods for Verilog HDL designs using Quartus and cocotb. These include, but are not limited to, checking synthesizer errors/warnings, checking the RTL schematics, and functional debugging using testbenches. As our test tool, we will use cocotb, which is widely accepted by both academia and industry as a powerful option for testing HDL designs. In the laboratory session, you will be given a ready design with several different types of errors that you will debug in the given time and demonstrate to your proctors.

Preliminary work is designed as a tutorial and teaches different levels and steps of debugging a design. Debugging is a supremely important concept in any engineering branch and one of the most useful skills, if not the most useful. Having a very well-designed idea laid out with groundbreaking innovations may be meaningless if you cannot fix design/implementation errors and get it up and running. Please take your time and familiarize yourself with everything present in this preliminary work, as what is taught here will likely be used throughout the rest of your career.

Although the preliminary work grade is not included in the overall grade of the experiment,

1. **You need to get at least 50% from the report.**
2. **The codes you debug should pass at least half of the testbenches (2/3 for this experiment).**

Both are required to be eligible to attend the experiment session. We will run testbenches automatically on the corrected codes you submit for the second requirement.

1 Preliminary Work

Let's start this laboratory work with two simple questions. First, why are we testing designs? There might be a plethora of answers to this question, but the most important reasons are:

- Verification of Correct Functionality: To confirm the design **functions** as expected under various use cases.
- Early Error Detection: To see the errors in the design before moving forward, as errors will likely propagate.

After you have seen the errors in a design/implementation, you need to **debug** the design. In general, debugging follows these steps after you find errors:

- Diagnosis: Finding exactly what is causing the issue
- Correction: Once you know the cause, you should change the relevant parts to fix the error.

After debugging, you go back to testing and repeat these steps until the design is working as intended. In this preliminary work, you will walk through how to test and debug Verilog HDL designs, so all the steps will be further divided into smaller elements and explained in that context.

You should submit the design codes you correct separately from the report for each item in this part.

To fulfill the requirements of this laboratory work, the following tasks should be performed. **Note: You should only include important parts of the code in the pdf report. The full codes should be submitted separately as a .zip file.**

Every design you correct here should be present in your submission as a ".v" file. The testbench you write should also be present as a ".py" file.

1.1 Reading Assignment

The Laboratory Manual, where the regulations and other useful information exist, is available on the ODTUClass course page. **Read the Laboratory Manual thoroughly**. Refer to the combinational Verilog lecture notes Part 1-3 as they contain everything about Verilog you need **for this experiment**. We also assume you are familiar with Chapters 1-4_2 of the EE348 course, as this experiment is complementary to those.

1.2 Debugging Assignment

For this preliminary work, you will not be designing a circuit; instead, you will try to debug the faulty Verilog modules we have uploaded to ODTUClass. Do not forget to open a Quartus project via System-Builder and set the Top-Level Entity appropriately for each part. For more details on creating projects and Top-Level Entity, refer to the Experiment 0 manual.

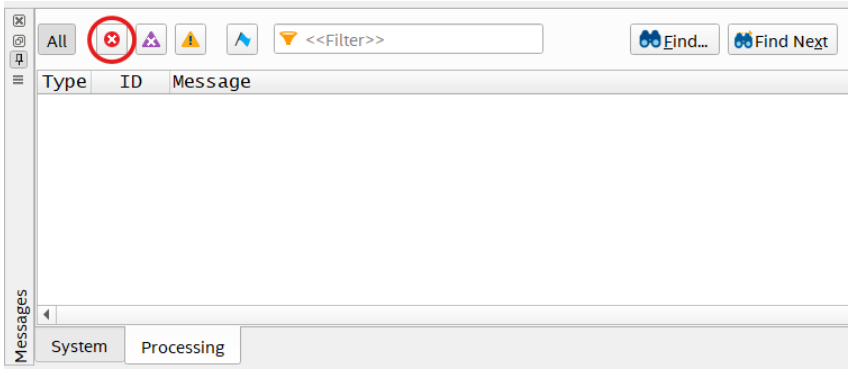
1.2.1 Debugging Synthesizer Errors (15% Credits)

One of the major advantages of using a software suite like Quartus is its robust error reporting system, which gives you an immediate head start in debugging Verilog designs. When you compile a design, Quartus automatically analyzes the code and flags any issues it finds, ranging from simple syntax errors to more complex misconfigurations. These error messages are presented with information, such as the error type, the specific line number in the code, and often even suggestions on how to resolve the problem.

Because these messages clearly indicate where the error lies, they are typically the easiest errors to address, allowing you to quickly debug the design without much hassle.

1. Open the given "OurDecoder" file, which contains the "OurDecoder" module in Quartus.
2. Start Analysis & Synthesis without modifying anything.

3. Check the Processing tab of the Messages window for Error Messages (icon). By default, the window should be located at the bottom of your screen. Use the keyboard shortcut Alt + 3 if it is not open. Alternatively, you can use View → Utility Windows to choose which windows are displayed.

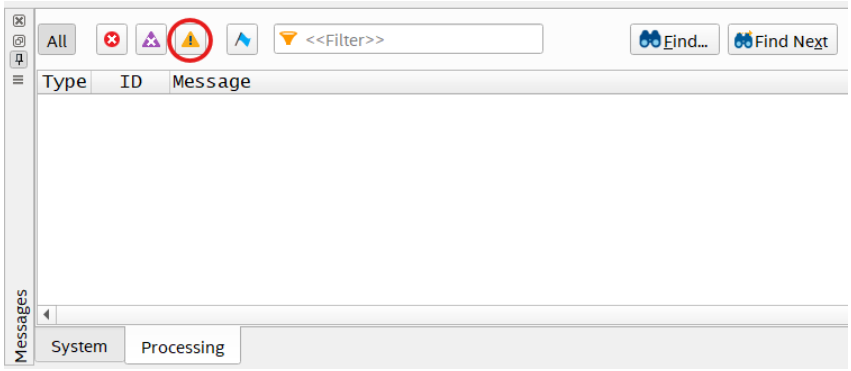


4. Describe and comment on the errors you get in one sentence. **(7% Credits)**
5. Fix the errors by making the necessary changes to the given code. Describe them in your report in one sentence. **(8% Credits)**

1.2.2 Debugging Synthesizer Warnings (15% Credits)

Quartus will also provide warning messages separate from the error messages. The warnings highlight unusual or unexpected aspects of a design that may not necessarily be errors but could indicate unintended behavior. You should check all warning messages to see if there is anything unintended present in the design. **Not all warning messages are errors** and some can be safely ignored.

1. For the "OurDecoder" module, start Analysis & Synthesis again.
2. Check the Processing tab of the Messages window for Warning Messages (icon).
3. Ignore warnings that have nothing to do with the given design files. Such as "Warning (18236): Number of processors has not been specified which may cause overloading on shared machines. Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance".



4. Describe and comment on the warnings you are getting in one sentence. **(7% Credits)**
5. Fix the warnings by making the necessary changes to the code. Describe them in your report in one sentence. **(8% Credits)**

1.2.3 Debugging RTL Errors with Naked Eye (15% Credits)

After addressing all Quartus messages, inspect the design's RTL schematic. Some errors can dramatically simplify the schematic—for instance, you might see only a few logic elements instead of the thousands you expect. Reviewing the RTL schematic "with naked eyes" helps you verify if there are any major errors, such as checking if every input and output is properly connected. If any signal is stuck (permanently grounded or held high), it indicates a design error. Unconnected signals are usually simple to fix by updating the code, while running testbenches can help identify any stuck inputs or outputs. Beware that only some of the bits of vector signals might be unconnected or stuck.

1. Open the given "OurEncoder" file, which contains the "OurEncoder" module in Quartus.
2. Start Analysis & Synthesis without modifying anything.
3. Open the RTL Viewer from Tools → Netlist Viewers → RTL Viewer. Put a screenshot of it into your report. Comment on the problem you are seeing in one sentence. **(8% Credits)**
4. Fix the problem by making the necessary changes to the given code. Describe them in your report in one sentence. **(7% Credits)**

1.2.4 Debugging Functional Errors with Testbench (20% Credits)

After addressing errors, warnings, and reviewing the RTL schematic, we verify functional correctness using the cocotb testbenches. The testbench includes a Python-based model of the Design Under Test (DUT) and generates test cases that are applied to both the DUT and the Python model, then compares their outputs. If the DUT is functionally correct, it will output the same values as the Python model; if not, the testbench will give errors as to which cases the design differs from the model. Since simply spotting mismatches isn't enough to pinpoint the issue, we also provide a helper function that prints all internal signals of the DUT after each failed test case so that you can check which of the internal signals is causing the error.

1. Test the "OurBCDConverter" module, which is inside the given "OurBCDConverter.v" file without modifying anything, using the testbench supplied on ODTUClass.
2. Describe the design problems shown by the testbench results in one sentence. **(6% Credits)**
3. Fix the problems by making the necessary changes to the given code. Describe them in your report with one sentence. **(7% Credits)**
4. Test the "OurBCDConverter" module once again using the testbench supplied on ODTUClass and add the screenshot of a successfully passed test to your report. **(7% Credits)**

1.2.5 Debugging Combined Designs with Testbench (35% Credits)

In this part, you will see why it is important to test our modules before combining them into larger ones. We have a module called "OurCodedConverter" which consists of a sequential combination of an encoder, a binary to grey code converter, and a decoder in that order.

1. Fill the template for "OurCodedConverter" inside the given "OurCodedConverter.v" file, by adding the fixed "OurEncoder" and "OurDecoder" modules and combining them with the given "OurBinaryToGrayConverter" module as described in the paragraph above.
2. Run the given testbench for the overall module "OurCodedConverter". Briefly comment on the testbench output. What problems do you see in the logged signal values(dut.binary, dut.outdecoder, etc.)? **(7% Credits)**

For this case, we know that the problem resides within the given "OurBinaryToGrayConverter" module. But in real cases, you have to find the faulty module by thoroughly testing and investigating each one. To avoid things coming to this point, you should test the modules at every step of the design process. That is why the ability to design testbenches is an important skill for an engineer.

1. To begin debugging the 4-bit "OurBinaryToGrayConverter" module, first you need to write a cocotb testbench. Your testbench should be exhaustive (check every input case). The basic idea of a testbench is to give a test input and compare the acquired output with the expected true output. You should read subsection 1.3 thoroughly to get an idea of how to write your testbenches as a beginner. The already given testbenches should also be a good reference point. You should include and utilize the "Log_Design" function as it is done in the given testbenches. This function allows us to print the values of the signals in our design. Note that the "Log_Design" function requires the "to_hex" function as well. Describe the working principles of your testbench. Put your testbench code(without the function declarations for "Log_Design" and "to_hex") in the appendix of your report. **(12% Credits)**
2. Now that you have your testbench, you can investigate the "OurBinaryToGrayConverter" module and debug the problems you are seeing.
3. After you have fixed the module, verify it by running your own testbench and add the screenshot of a successfully passed test to your report. **(8% Credits)**
4. Once again, run the given testbench for the overall module "OurCodedConverter" and add the screenshot of a successfully passed test to your report. **(8% Credits)**

1.3 What Is Cocotb and How to Use It

In the most basic sense, cocotb is an interface between Python and an HDL simulator. Cocotb transfers every signal in a Verilog code to the Python script as the variables of an object representing the Verilog design. It executes a Python code until no thread has anything left to do, after which it switches to the HDL simulator, changes the signal values you changed in the Python, and advances the simulation time. You can check Figure 1 for a simple flowchart of cocotb's operation.

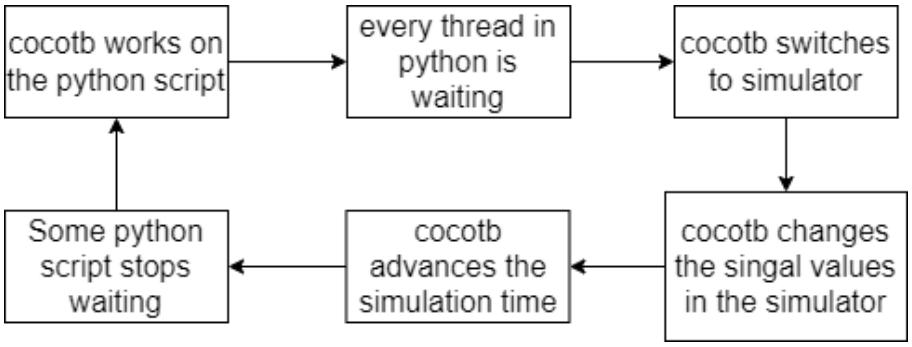


Figure 1: Simplified Workflow of cocotb

You should first read cocotb quickstart guide if you have not already. All of the code will be written in Python. Cocotb modules will be imported just like any other library modules and will be called in the code. Cocotb will call the functions marked as tests and give a design as an argument to the function. Thus, all the signals from the Verilog design can be accessed just like variables of an object.

You can either verify the design by manually changing the signals and checking the outputs, or fully automate the process. Anything doable with Python is possible, so the testbench need not take any particular form.

One thing to note is the usage of 'makefile'. As explained in cocotb quickstart guide the makefile is simple text that will contain the name of the Verilog source codes you want to include, the top-level module for the Verilog design, the Simulator that will be used, and the Python test file that contains the test-benches.

For reference, please check the testbenches supplied in Odtuclass. You can also use them as templates for your testbenches. Before writing your testbenches, read cocotb quickstart guide and Writing Testbenches part of the how-to guides.

2 Experimental Work

In the experiment session, you will have two parts: a preliminary work demonstration and a debug task.

2.1 Preliminary Work Demonstration (20% Credits)

Run the cocotb testbench for the "OurCodedConverter" module once again and show a successful run to your laboratory instructor.

2.2 Experiment Session Debug Task (80% Credits)

In the experiment session, you will be given buggy modules (same/similar modules with different errors in each session) and will be required to debug specific module combinations. These modules will include the ones from the preliminary work as well as brand new ones. The exact details will only be given when the experiment sessions start.

Because the purpose of the laboratory is debugging, the instructor assistant will not help you in any capacity or answer any of your questions about the problems with the designs.

3 Parts List

`DE1-SoC Board`