LAB: Testing with ModelSim in VHDL

# Software Required

Quartus Prime v16.1

ModelSim Altera Standard Edition

# Document Revision

V1.0 – 14/02/2018 – Initial version for ELEC241

## Important Note – please read as it refers to assessment

There is no specific lecture to accompany this. This lab is self-contained due to the practical nature of the topic.

This lab is very much a tutorial on using software tools. All tasks carry equal marks. The challenge at the end is worth 20%. You should only attempt this is you have completed all other tasks.

For your log book, you should keep a record that would enable you to use ModelSim in the future. As you follow the steps yourself, record screen shots of what you are doing (use Quartus version 16.1) .

**Build your own tutorial**

If you do this properly, you will not have to refer back to this lab again or scrub through any of the videos (which can be time consuming).

Furthermore, the videos in this section use an older version of the software. Although the steps should not have changed\*\*, your notes will be more current.

\*\*it is conceivable that some minor changes have occurred since these videos were recorded. We can address this in the lab sessions and amendments will be added to this document as appropriate.

# Introduction

In this tutorial, you will learn to use Quartus and ModelSim (starter edition) to test your VHDL designs. Quartus is the tool you have used so far to write and simulate all your VHDL. Quartus is focused more on writing synthesizable VHDL and testing on physical hardware. Synthesis and layout is complex, and therefore time consuming. However, while you are working on the behaviour of your VHDL, you are mostly interested in the syntax and correct logic of your VHDL code. This is where ModelSim comes in.

Intel / Altera ships a version of ModelSim specifically setup for Quartus, known as Altera ModelSim Standard Edition. This is a professional level tool for simulation that once mastered, can be highly productive for VHDL composition and simulation. VHDL has many power features that can be used for testing, but cannot be synthesized. As you learn to use ModelSim, you will now begin to look at some of these features.

In this practical **you will use Quartus to:**

* Create schematics and convert them to VHDL
* Write VHDL components
* Convert schematics to VHDL
* Export timing information to ModelSim for gate level simulation (with timing information)
* Create template test-bench code

**You will use ModelSim to:**

* Modify and develop test bench code
* Run both functional and timing analysis
* Perform automatic testing

# Task 01-01 Setting up a Quartus Project to use ModelSim

You can use ModelSim on its own, but to get the best of both worlds, it is advisable to start with Quartus. Before you can begin, you must create a project that is configured to use ModelSim for simulation and testing.

All the videos featured in this document also appear on your module site. The version you are using might differ visually from that shown in the video, but the method is the same.

**TASK** – Watch Video 01. If using Microsoft Word, you can double click the video and resize to full screen. Alternatively, you can use [this link](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=3bd1dd43-513f-4c49-9df3-a8790104e719).



Figure 1. Setting up Quartus with ModelSim.

Figure 2 (Video 01 ) - Setting up Quartus with ModelSim.

## TASKS

1. Follow the procedure in Video 01 to create your own project.
2. Create a XOR gate as shown in the video, or alternatively, create some alternative combinational logic where you know the defined behaviour.

NOTE - you need to know the folder where the ModelSim executable is located. This may vary, for example, between 32-bit and 64-bit operating systems. For the Windows version, right-click the icon for ModelSim and examine the properties. From there you should be able to derive the correct path.

# TestBenches in ModelSim

In this section, we will use ModelSim for testing. At this stage, we are not testing for timing errors (setup and hold time violations, propagation delay) but we use VHDL techniques to add delay to the simulation.

**Learning Outcomes**

1. Use a test bench to perform functional testing on a VHDL component using model sim
2. Use the wait instruction instead of a sensitivity list
3. Use wait for keywords to simulate delays in your VHDL

# Task 02-01 Creating and Running a Test Bench

Up to this point you have used a vector waveform file for testing. You may have found this simple to set up, but the creation of test benches and reading the results can be error prone and repetitive.

Test benches are VHDL code written to test other VHDL code.

You can use the power of the complete VHDL language to generate test stimuli and observe the outputs. In this section, although we focus on functional testing, timing is introduced. Some important new keywords are introduced specifically related to timing:

* **wait** - will simply stop the simulation of a VHDL script.
* **wait for** - will create a delay or wait for an event

Test bench code uses structural VHDL instead of a schematic. Each test bench is typically an entity with no inputs or outputs. Any output is logged to the console or Wave view in ModelSim.

Although harder to write initially, this approach is much more portable and leverages many powerful features of VHDL. As we will soon discover, it can also be used to automate testing. This is a very powerful feature which allows us to rapidly check our changes have not broken any logic or timing constraints.

You will also note that the process blocks may have no sensitivity list. Instead, we use the **wait** statement. This gives us some insight into what sensitivity lists are in a simulation context. These are also explained in the video. Note that you cannot use both sensitivity lists and wait.

## TASKS

1. Watch [Video 02A](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=516e8f25-8ec2-425a-b9fa-a8790104e76f) and follow the procedure on your own computer. Use a for-loop to generate the inputs



Figure 3. (Video 02A) shows how to create a template test bench, and how to modify this template to create some test signals.

1. Now proceed to [Video 02B](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=5ce630a2-b61b-4b33-adc3-a8790104e754). This shows you how to run your test bench code.
2. Follow the procedure shown to use ModelSim produce a graphical timing diagram. Use this to verify your VHDL is correct.



Figure 4. (Video 02B) Perform some functional testing with ModelSim. This includes using a for-loop to automatically create test stimuli.

So far, we’ve only used a graphical means to analyse out logical outputs. Next, we turn our attention to the all-important **assert** command.

# Task 02-02 Automatic Testing

Using timing diagrams are certainly useful when you want to closely examine the outputs of some tests, or to provide final evidence that your design is correct. However, it is time consuming and error prone to read them. Checking a timing diagram every time you make a small change is both repetitive and error prone. If you consider a large system with multiple inputs and outputs, and critical timing characteristics, then visual inspection usually does not scale. As the design becomes more complex, so the time to visually verify grows disproportionately[[1]](#footnote-2).

A complimentary and very powerful technique is to perform automatic testing. Although slow to set up, this is a much more cerebral and non-repetitive task than visual inspection. For this we are going to use the **assert** command.

## TASKS

1. Watch the video [03 - Automatic error checking - PART A - The Assert Command](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=09ee18de-0cb3-4378-9097-a8790104e738) and follow the procedure on your own computer
2. Use an array to store the ‘expected outputs’
3. Try changing your VHDL (the object under test, not the test bench) so that the tests fail. Verify that you get an error message then you run your test bench.



Figure 5. (Video 03A) shows you how to use the assert command to automatically check your logic output is correct, and to flag errors when you run your test bench.

# Task 01-02 Using NativeLink

Native link is facility that allows your VHDL test bench(es) to run automatically every time you launch ModelSim from within Quartus. This is good practise where you want to run a set of tests on your VHDL every time you make a change (to check you’ve not broken any functionality!).

## TASKS

1. Watch [Video 03 - Automatic error checking - PART B - Native Link](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=d35bbbba-d8db-4660-b09e-a8790104e98f)
2. Follow the procedure in Video 03B to add automatic running of your test bench using NativeLink
3. Verify that the timing diagram is correct and that **assert** commands are detecting any errors



Figure 6. In this video, you see how to set up NativeLink inside Quartus. Follow these steps very carefully as it’s easy to make a mistake.

Now proceed to the next section where you will examine timing

# Gate Level Testing

So far, all the testing you have performed is functional testing. **Functional tests** are very important to verify the **logic** of your design, and to check that your approach is at least logically correct. They ignore timing however.

Functional testing is also relatively fast as propagation delay effects, setup times and hold times are not considered.

**Gate-level testing** requires electrical characteristics of the hardware device when you synthesize your design, as well as other factors such as the path length. This information is held by the device vendor (Altera in this case) and produced by Quartus when you perform a full build.

Gate level testing is important as a circuit may be perfectly designed in a functional sense, but in reality, it may fail due to issues related purely to timing.

**Learning Outcomes**

1. Use a test bench to perform gate level testing on a VHDL component using model sim
2. Use the assert comment to test for timing errors

# Task 03-01 Gate Level Simulation with Quartus and ModelSim

Quartus can provide the relevant information that allows ModelSim to perform gate level simulation, and thus you can begin to test for timing errors. It is a very large topic, so we can only begin to explore it here.

## TASKS

1. Watch [Video 03 - Automatic error checking - PART C - Gate Level Simulation](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=557edf1a-18bf-4275-88b1-a8790104ee2e)
2. Follow the procedure in Video 03C to enable testing at gate level
3. Show the timing diagram showing propagation delay effects
4. Modify your timing requirements such that the assert command detects timing errors (as demonstrated in the video)



Figure 7. In this video, you are shown how to simulate and verify timing for a particular FGPA (supported by Quartus). Quartus exports the model information to ModelSim so that it can perform an accurate gate-level simulation that incorporates propagation delay effects.

**Challenge (20% mark)**: In the previous lab, you built a state machine that worked as a simple timer (256 counts).

Try to build a testbench to test that component. You should not use graphical output for testing, instead use assert

Start with purely functional testing. Once you have your tests setup, try a gate level simulation. Does it still pass?.

Can you find out how to single step through your VHDL?

**Tips**

* Start in Quartus. Make your timer the top level entity and generate a testbench (to save you having to do it)
* Check everything checks out with a vector waveform file
* Now start the simulation in ModelSim. Don’t forget to compile the test bench
* Start with a process block for the clock

# Timing

Our focus so far has been on functional testing. In this section we take a closer look at methods to detect timing faults, and in particular setup and hold time violations.

**Setup time** – the time an input must be in a steady state before the active clock edge

**Hold time** – the time an input must be in a steady state after the active click edge.

To be continued

# Appendix A – Nucleo and DE0-Nano Connections

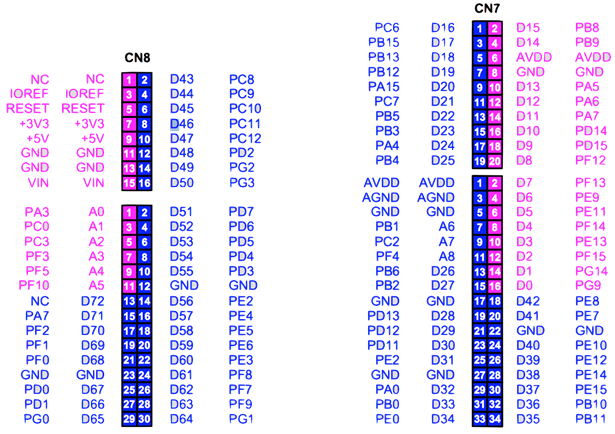


Figure 8 Nucleo Pinouts

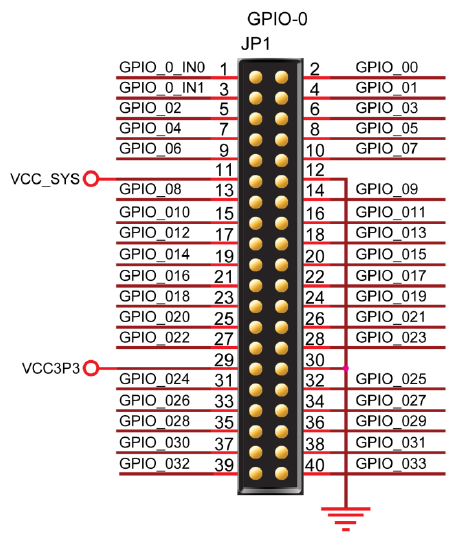


Figure 9. DE0-Nano Pinouts for JP1

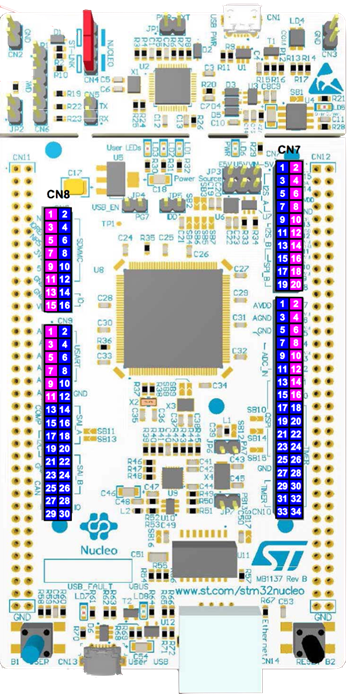


Figure 10. Nucleo FZ429ZI

A circuit board

Description generated with very high confidence

Pin 1

Figure 11. Showing the pin 1 on the FPGA DE-0 Nano Board

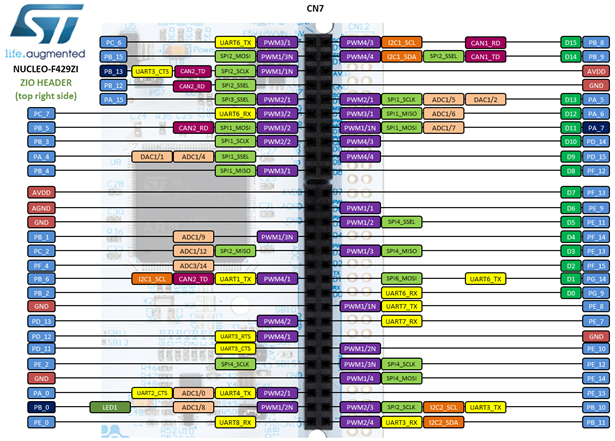
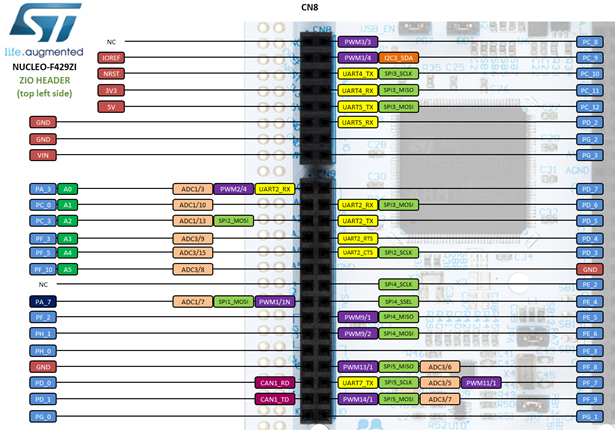


Figure 12. Showing the alternative functions of the pins on the Nucleo F429ZI CN7 connector



# Appendix B – ASM Charts

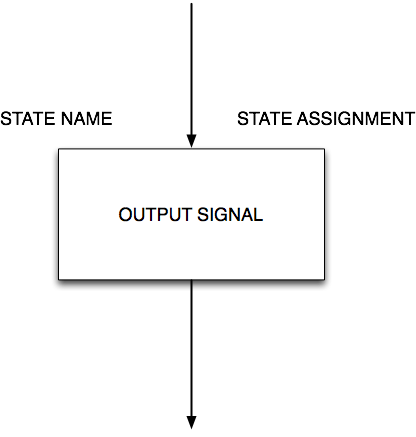


Figure 13 State with Moore Output Signal. The state name and assignment (optional) area all unique

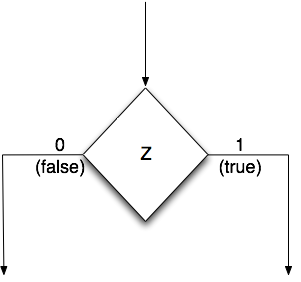


Figure 14 State Transition Decision. The variable Z is being tested. This must ultimately lead to a state box, possibly via another decision and/or Mealy Output(s)

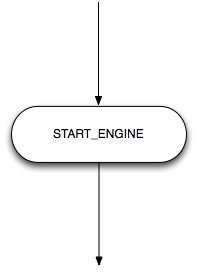


Figure 15. Mealy Output. The example here is the START\_ENGINE signal which is asserted high. It only makes sense to connect a decision symbol to the input of the Mealy output symbol.

1. Let’s not forget mental fatigue and errors [↑](#footnote-ref-2)