Introduction to Electronic System Design (DAT093)
Lab 3: Test benches and design synthesis

Sven Knutsson, Lars Svensson

Draft of Version 2.1, September 21, 2018

1 Introduction

The previous lab sessions have illustrated some of the fundamentals of HDL-based hardware design: description of combinational and sequential behavior; design hierarchy and parameterization; some fundamental arithmetic implementations; and the practical tools and tasks involved in development. This session will offer two additional perspectives:

1. For previous tasks, you have been provided with simple test benches for verification of your designs. Most probably, these have helped you to find bugs and oversights in your work. In an industrial design project without externally-provided verification infrastructure, *test bench development* is a necessary skill for the hardware designer: it will help you to arrive quickly at bug-free implementations.

2. So far, you have used QuestaSim simulations to check your designs for correctness; but simulation results are rarely the intended end product of the design effort. *Design synthesis* produces hardware that implements the behavior described by the code, either as configurations for reconfigurable hardware (such as the FPGAs used in this course), or as netlists of components for chip integration. The synthesis process is in some ways similar to the QuestaSim compilation stage, but there are important additional concerns.

2 Preparation

Read through this entire lab PM and work through all tasks marked **Preparation**.

3 Test bench design

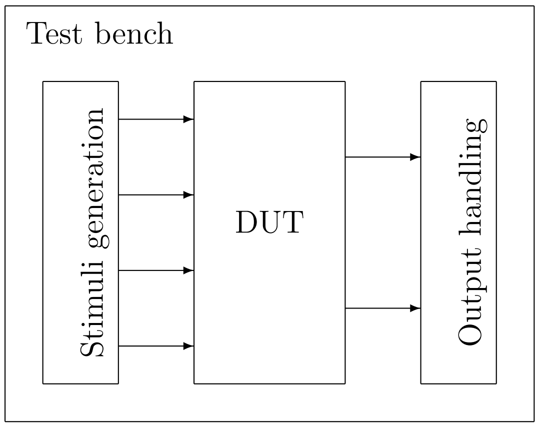

Figure 1: Test bench

Figure 1 illustrates a test bench of the kind you have used in previous labs[1]. The Design Under Test (DUT) is included in the test bench as a component, in the same way as it would be in a larger design. The test bench generates input stimuli for the DUT, records the DUT output signals, compares these outputs with the expected values, and generates warnings and error messages if appropriate. Especially for complex DUTs, test bench development may take as much effort as the development of the DUT itself.

It can be risky to let the designer of a component also design the test bench for it. When design errors are caused by misunderstandings of the specifications, the test bench may suffer from the same misunderstandings and therefore fail to reveal the erroneous behavior[2]. Common ways to reduce these risks include letting another designer (or group of designers) develop the test bench from specifications without detailed knowledge of the design to be tested, or at least to let the designer *first* develop the test bench before continuing on to the design.

In this lab, you will design, from specifications, a test bench for a counter and use

[1]Other kinds of test benches exist, but as usual, we stay simple in these labs.

[2]Similar observations are often made for software development (for more information, you may search for "test-driven development" and "test-first programming").

The DUT is a synchronous modulo-N up/down counter. It counts the cycles of the clock signal and wraps around after N cycles, that is, the value $N-1$ is followed by the value 0 when counting up, and 0 is followed by $N-1$ when counting down.

The interface comprises the following ports:

- A clock signal `clk`
- An asynchronous `reset` input signal that sets the count value to 0
- A synchronous `enable` input signal that enables the counting of clock cycles
- A synchronous `load` input signal which sets the counter to the value given on the `data` inputs
- An $\texttt{up}/\overline{\texttt{down}}$ input which selects the direction of counting
- An output signal `count` that shows the current value

The `reset` signal overrides `load` and `enable`. `load` overrides `enable`. Thus, reset and load operations work also when counting is stopped. All signals are active high. Triggering is on the rising edge of the clock.

Figure 2: DUT specification

it to test several "black-box" implementations. The specification for the DUT is shown in Figure 2; its VHDL entity is shown in Figure 3. As you can see, the counter design is generic with a parameter N for the count range. The parameter value is used to derive the signal widths. The way this is done requires the use of the `ieee.math_real` library. Assume $N = 20$ in your test bench.

Preparation: Design a test bench[3] of type 3 for the counter as specified in Figures 2 and 3. Feel free to use the test benches of the previous labs for inspiration. Make sure to test all combinations of the control signals, and also what happens at positive and negative wrap-around. Make an effort to think of every part of the behavior that may go wrong! Allow the test bench to run to completion even if an error was detected. Do not forget to design a *do* file for the test bench.

- Create a VHDL entity for the test bench, and an architecture in accordance

[3]See the document "Introduction to QuestaSim", downloadable from the course homepage.

```
ENTITY counter IS
  GENERIC(N:POSITIVE:=20);
  PORT(clk:IN STD_LOGIC;
       reset:IN STD_LOGIC;
       enable:IN STD_LOGIC;
       load:IN STD_LOGIC;
       up_down:IN STD_LOGIC;
       data:IN STD_LOGIC_VECTOR(INTEGER(CEIL(LOG2(REAL(N))))-1 DOWNTO 0);
       count:OUT STD_LOGIC_VECTOR(INTEGER(CEIL(LOG2(REAL(N))))-1 DOWNTO 0));
END counter;
```

Figure 3: DUT entity. The widths of the `data` and `count` signals are derived by computing the number of bits necessary to represent the value of `N`.

with your prepared design. Compile and correct any errors.

You will use your test bench to test four different counter implementations. Some of these do *not* adhere to the specifications as given above. Your test bench is supposed to identify the erroneous counters. The counter implementations are available as encrypted[4] VHDL files on the course home page.

- Download the encrypted counter implementations from the home page.
- Compile your test bench with one of the encrypted DUTs and run it. Make note of the errors flagged. Repeat for the other encrypted DUTs. Check with the lab assistant to verify that your test bench correctly identifies the erroneous behaviors and pinpoints the errors.

[4]Encrypted HDL descriptions are often used in industry to reduce the risk of plagiarism.

4 Design synthesis

Originally and fundamentally, VHDL was conceived to describe the behavior of digital hardware, and specifically to make it possible to *simulate* the description. Thus the language, by design, includes constructs which make sense in simulation but which cannot reasonably be synthesized. (One example is the `AFTER` construct, which states that a signal assignment is to be delayed by a certain time; but the exact time behavior of digital hardware depends on many factors such as voltage and temperature, and accurate delays therefore cannot be guaranteed. Another example is file-system access for test patterns, which is useful in simulation but not in embedded hardware.)

Any VHDL design intended to be synthesized[5] must therefore avoid using the non-synthesizable constructs, or in other words, must use only the *synthesizable subset* of the language. Experienced designers rarely find it difficult to avoid the problematic constructs.

Your next task in this lab session is to investigate your designs from Labs 1 and 2 to determine if they are synthesizable, and if necessary modify them to eliminate synthesis problems.

Preparation: Download the document "Introduction to Xilinx Vivado" from the homepage and study it. Make note of how to create an RTL project; how to add VHDL files to the project; how to select what FPGA hardware to synthesize for; how to invoke the QuestaSim simulator from inside Vivado; and how to synthesize the design for the selected FPGA. (You will not actually download the resulting configuration to an FPGA in this lab, so you may postpone reading the parts after the heading **Downloading to the FPGA**.)

The first design to synthesize is the ripple-carry adder/subtractor (RCAS) from Lab 1.

- Launch Vivado. Create a new project and include copies of the files needed by your Lab-1 RCAS (excluding the test bench which you will not want to synthesize!). Perform synthesis of your design. Replace any non-synthesizable constructs until synthesis succeeds; consult the lab assistants if you get stuck.
- Study the Vivado synthesis reports. Check for warnings and contemplate whether they are serious (again, consult the lab assistants if you have

[5]Test benches are examples of VHDL designs that *don't* need to be synthesizable.

doubts). *Save the synthesis logs;* you will need to submit them to pass the lab!

- Make note of size and speed reports after synthesis, for later comparisons.

Successful synthesis is like successful compilation of a software program: it is a necessary step that may uncover mistakes, but does not guarantee that the final design is free of bugs. It is possible to simulate the design also after the synthesis stage. To do this, you need some *do* files for the simulator.

- Download the Lab-3 *do* files from the course homepage (since the test bench files are not used here, the new *do* files are different from those used for the same designs in earlier labs).
- Use the appropriate *do* file to simulate the synthesized RCAS at the post-route stage (that is, when the code has been synthesized and mapped into logic of the selected FPGA family, and when signal routing of the design has been done).

  Note that without a separate test bench, it is necessary to manually check that the results are the expected ones. Inspect the *do* file to see what to look for.

  Be aware that the synthesis process may have eliminated some internal signals. If you need these signals for debugging, you can make them accessible by temporarily adding an extra port for them. Any such extra ports should be removed once the bugs have been fixed.
- Show the correct simulation result to your teaching assistant.

The same procedure can now be carried out also for the other designs from Labs 1 and 2.

- Repeat the synthesis and simulation stages above for the serial adder/subtractor from Lab 1, and for the direct and the serial FIR filters of Lab 2.

The parallel and serial versions of the adder and the filter should be functionally equivalent but differ in terms of cost and performance.

- Compare the speed and size ratios of the parallel and serial adders. Is the hardware expense of the parallel version reasonable in view of the improved performance?

- Repeat the comparison for the parallel and serial FIR filters.

Again, note that you will need to submit the synthesis logs for the RCAS and for the serial FIR filter to pass this lab! Submission is done via the course homepage. If you did not complete this task during the lab session, you may do it later; the submission deadline is given in the course PM.

Preparation: Consult the course PM to find out the submission deadline for the synthesis logs.

5 Wrap-up

After completing this lab session, you are expected to be able to carry out the following tasks:

- Develop a simple VHDL test bench for a sequential digital design, including the *do* files for QuestaSim
- Synthesize a simple VHDL design in Vivado
- Verify through simulation that the behavior of the synthesized design corresponds to that before synthesis
- Evaluate speed and area reports from a synthesis run

The following questions may be brought up in class later:

- Did you find the English-language specification in Figure 2 sufficiently detailed and accurate to make it easy to develop the test bench?
- Did you find any non-synthesizable constructs in your designs? What did you do about them?
- Did the speed and area comparisons for serial and parallel implementations agree with your expectations?