Lab Exercises in TDT4255 Computer Design

Computer Architecture and Design Group
Department of Computer and Information Science

August 31, 2016

Contents

List of Figures

Abbreviations

CPU Central Processing Unit

RTL Register Transfer Level

FPGA Field Programmable Gate Array

HDL Hardware Description Language

VLSI Very-Large-Scale Integration

VHDL Very-Large-Scale Integration (VLSI) Hardware Description Language (HDL)

ASIC Application-Specific Integrated Circuit

FSM Finite-State Machine

ALU Arithmetic Logic Unit

UART Universal Asynchronous Receiver/Transmitter

DUT Design Under Test

1. Introduction

This compendium is an accompaniment for the lab assignments in the course TDT4255 Computer Design. Aside from information on administrative practicalities and the description of each lab assignment, it contains a body of information that covers the basics of hardware design and prototyping on Field Programmable Gate Arrays (FPGAs).

We would like to remind you that the lab assignments are graded and these grades constitute part of the final grade in the course. Therefore, it is in your best interest to carefully read this compendium and understand its contents.

1.1. Goals and Outcomes

Central to the operation of any computer system is the Central Processing Unit (CPU). While the CPU itself consists of multiple components, the *processor core* is the most vital for computing. The primary goal of the TDT4255 lab assignments is to give you practical, hands-on experience in processor core architecture by allowing you to design and implement your own.

Throughout the lab assignments, you will design and implement processors of increasing complexity. You will begin with a warm-up exercise in which you will design a simple stack-based processor. In the graded exercises, you will first design a processor with a pipelined architecture, which you in the second exercise will expand with the ability to issue two instructions per cycle. You will implement your designs in a hardware description language, and run your implementation on an FPGA. Finally, you will answer a quiz which asks you to analyse the qualities of your design. Solving the exercises will be made possible by combining the theoretical knowledge you will receive from the course lectures with the practical skills in hardware design from the assignments and the compendium.

The learning outcomes of the laboratory exercises are as follows:

- knowledge of the processor core architecture
- hardware design in VHDL
- development workflow for hardware design
- prototyping hardware designs on FPGAs

1.2. Suggested Workflow for Assignments

There is a well-established workflow for digital design with Hardware Description Languages (HDLs), which describes the steps necessary to go from specification to functional hardware. Here we present a simplified version of this workflow, which you should follow for the lab assignments:

1. **Read and understand the specifications/requirements.** As with any engineering practice, the first step is to develop a general understanding of the problem and the requirements.
2. **Identify the inputs and outputs.** The inputs and outputs may or may not be explicitly specified as part of the requirements. Specifying them will help you structure your thinking for constructing a solution that fulfills the requirements.
3. **Sketch RTL with basic blocks.** Before you start implementing anything in VHDL, we highly recommend that you make a pen-and-paper sketch of the solution using RTL building blocks like registers, multiplexers and logic/arithmetic operators (covered in Appendix B).
4. **Implement in VHDL.** Once you have a register-level solution in mind, you can implement this in VHDL. It is important to keep in mind that *HDLs are not software programming languages*; thinking of HDL code as a static structure specification like HTML is actually more appropriate. Writing VHDL like writing Java can result in designs that appear to work at first, but have serious problems and may not run at all on an FPGA. This is why we emphasize envisioning your design in terms of basic blocks (step 3) before starting to code.
5. **Simulate and verify with testbenches.** The easy way of checking if your hardware design works without actually making hardware is simulation, which allows you to see how your design responds to different inputs. You can use VHDL to create *testbenches* that test your design. If you find that the solution does not work as intended, you should revisit the previous steps to fix it before moving to the next step.
6. **Synthesize and configure FPGA.** After verifying that your design works as intended in simulation, you can run it through the FPGA toolchain from Xilinx and configure the FPGA with your design. The toolchain may reveal additional problems with your design that is not detectable during the previous phases. It is very important to investigate any warnings issued by the toolchain, especially concerning the unintended creation of latches (see Appendix C).
7. **Test on FPGA.** Once you have configured the FPGA with your design, you can test it with the intended real-world inputs (software programs for TDT4255 since we are building a processor).

Additionally, we highly encourage you to apply modular design principles when designing complex top-level units, i.e breaking it down into a hierarchy of smaller submodules and following the steps above for each submodule. While working on small submodules it may be more practical to skip the FPGA testing stage until integration of all submodules is complete.

1.3. Practical Information

Some practical information is presented here in order to make it easier for you to prepare and deliver your assignments, but also to prevent misunderstandings regarding the content and grading of your deliveries.

1.3.1. General

- **Individual Deliveries:** In contrast to previous iterations of the course, the exercises are to be completed individually. Discussion with other students about how one may solve the exercises is encouraged, but collaboration on the implementation of such solutions is prohibited. In other words: you may share ideas, but you have to write all the code yourself.
- **Lab location:** The lab premises for TDT4255 are located on the fourth floor of the IT-west building, room number 458 (map: `http://s.mazemap.com/1oQFavT`).
- **Compendium updates:** Make sure you have obtained the latest version of this document before starting a new exercise. New compendium releases will be announced through the e-learning portal.
- **Questions and errata:** Please contact the teaching assistants if you have questions regarding the compendium content, or if you detect any errors.

1.3.2. Delivery Contents

The delivery for each graded assignment should contain the following items:

1. VHDL files for the implementation.
2. Answers to the quiz for each graded assignment.

1.3.3. Evaluation

Assignment deliveries are evaluated based on your answers to the quiz and the correctness of your implementation. The latter is checked using an automated set of tests, which

will check that your design conforms to the exercise requirements both in simulation and on the FPGA. The grade weights are as follows:

- Design correctness in simulation: 30 %.
- Design correctness on the FPGA: 20 %.
- Answers to the quiz: 50 %.

2. A Hands-On Introduction

In this section, we will cover a simple digital design example to provide a practical introduction to the lab assignments. This will include sketching a design that meets the specification, implementing this design in VHDL and testing it with a simulated testbench, and finally using the Xilinx tools to prototype the implementation on a real FPGA.

2.1. Digital Design with HDLs

Digital design is the process of designing digital electronic circuits, which are central to the operation of modern computers. This is typically carried out with the help of a HDL, which specifies the digital circuit in a manner similar to how HTML is used to specify the structure of a web page. A design specified in an HDL can be implemented as a digital circuit in several different ways, including as hand-wired components, as an integrated circuit made from semiconductors, or using an FPGA.

For the TDT4255 lab, we will focus on the abstract logic functionality of the digital circuit in terms of how it is designed and realized with an HDL instead of discussing how it is physically implemented. We will be working at an abstraction level called Register Transfer Level (RTL) to describe the flow of data between registers, and the logic operations between them. We will also cover how these designs can be implemented on an FPGA, which offers a powerful implementation platform for digital design combined with HDLs.

2.2. Part 1: Warmup with VHDL and Xilinx Vivado

For the lab assignments, you will be working with a language called VHDL. For convenience, you will be using the Xilinx Vivado as a development environment[1]. This section will walk you through the basics of creating a new project using Xilinx Vivado, making a simple VHDL module that implements some combinational logic and verifying this in simulation, and finally testing your design on a real FPGA.

[1] In the handouts for assignments, there is also a Makefile-based project you can use if you want to write your VHDL code in another text editor. This may be a better option if you work remotely.

Throughout this section, you will be presented with two different perspectives on VHDL. While working on the lab assignments, it may be beneficial to keep this distinction in mind:

1. **VHDL for hardware:** This type of VHDL (also called *synthesizable VHDL*) describes actual hardware, and will be introduced in Section 2.2.2. The hardware that you create as part of the assignments in TDT4255 must be written in synthesizable VHDL. When writing this kind of VHDL, you should think of the language as something which specifies a static structure similar to what HTML does.
2. **VHDL for testbenches:** This type of VHDL is useful for checking how your hardware behaves in simulation, and includes useful functions like file and console I/O in addition to the features of synthesizable VHDL. It will be covered in Section 2.3.4. You will be using this type of VHDL for testing your hardware designs. When writing this kind of VHDL, you can think of the language as something similar to a normal programming language.

2.2.1. Step 1: Creating a New Vivado Project

We will start by creating a new project for the tutorial:

1. Launch Xilinx Vivado, which is installed on all lab computers. You can use the Unity launcher, or execute `vivado` in a terminal.
2. Create a new project named `tutorial`. The project location is up to you. The project type should be *RTL Project*. Uncheck "Do not specify sources at this time".
3. In the next window, set "Target language" and "Simulator language" to be VHDL. Do not add any files: click "Next" to pass "Add Sources", "Add Existing IP" and "Add Constraints".
4. Select the part named `xc7a35tcsg324-1`. This cryptic name can be interpreted as follows:
   - Family: Artix-7
   - Device: XC7A35T (colloquially, "an Artix-7 35T FPGA").
   - Package: CSG324
   - Speed: -1 [2]

Take a moment to familiarize yourself with the general layout of the IDE. Figure 2.1 highlights the main areas of interest. A brief explanation for each marked area is provided below:

[2] The speed grade of a particular FPGA depends on the chip fabrication process. Typically, the higher the number after the dash, the better.

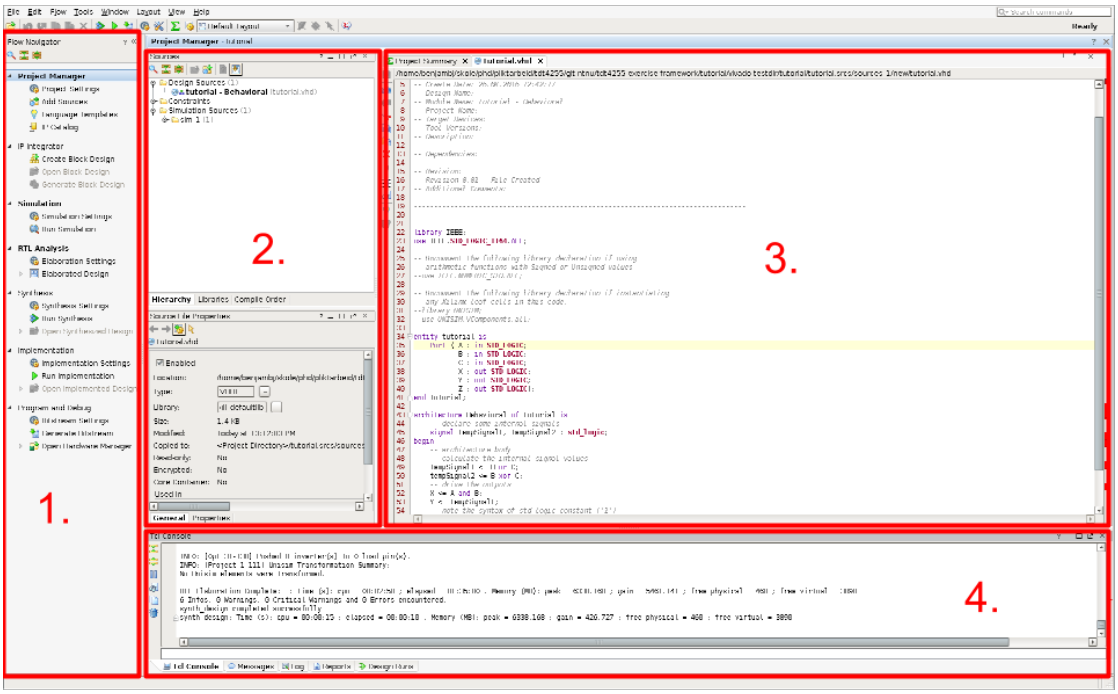


Figure 2.1.: Screenshot of Vivado with main areas of interest enumerated.

1. **Flow Navigator:** Controls the actions performed by Vivado as well as the active primary view (the ones labelled 2 and 3 in our screenshot). You may initiate certain actions by clicking on items below the headings in this pane. Note that some actions depend on the results from others, so if you start an action further down the list the dependency actions will be launched first. If you have multiple flows active, you can select between active views by clicking on the corresponding flow heading (active flows are indicated by bold header fonts).

2. **Sources:** Shows the current modules and submodules in the design. This is visible when the "Project Manager" flow is selected. You can use this to navigate between different VHDL files in the design. You can right-click modules here to change to top-level module (thereby controlling what subset of the design the actions initiated from the Flow Navigator will apply to.)

3. **Code Edit:** Used for viewing and editing VHDL code. This is visible when the "Project Manager" flow is selected.

4. **Messages/Console:** Feedback (such as error messages) will be shown in this window for the current actions.

2.2.2. Step 2: Your First VHDL Design

To get a hands-on introduction to the basic concepts and syntax of VHDL, we will be developing a simple module that implements several logic operators. The specification for the module will be as follows:

- The module will have three binary *input ports*, A, B and C.
- The module will have three binary *output ports*, X, Y and Z.
- The relation of the outputs to the inputs shall be as follows:
  - X = A AND B
  - Y = B OR C
  - Z = (B OR C) when A = 1, (B XOR C) otherwise

Start by clicking "Add Sources" (hotkey Alt+a) and make a new VHDL file with the name `tutorial` by selecting "Add or create design sources" and clicking "Create File" on the next wizard screen. We recommend that you create the source files in a subdirectory (for example `vhdl` or `src`) as the Vivado project directory itself will get cluttered with many generated files. When you click "finish" you will be presented with a new wizard screen, where you will have the option to specify details regarding the inputs and outputs of the module you create. This wizard is as such convenient for generating the standard minimum code for new VHDL modules. Create three ports named A, B, C with the Direction parameter set to "in" (for input), and three ports X, Y, Z with the direction "out". Leave the Bus box unchecked, and do not enter anything into the MSB or LSB fields. We will use these fields to create multi-bit inputs and outputs later on. You can leave the Architecture Name as "Behavioral".

The wizard will now complete and generate a `tutorial.vhd`, which you can now find under "Design Sources". Take a moment to examine the generated code, which we will be going through briefly to highlight some of the basic concepts and syntax issues in VHDL. We also provide a "cheatsheet" in Appendix A for your convenience.

Basic Syntax

VHDL is case insensitive, but using consistent capitalization is good practice. Comments start with `--` and are single-line[3]. Most statements end with a semicolon `;`, though there are exceptions. Looking at example code is a good idea until you get used to the quirks of the VHDL syntax. The code is structured as different kinds of blocks as we will see further on, whose boundaries are marked with `begin/end`.

[3] You can select multiple lines of text, right-click the selection and select "Toggle line comments" for easy multi-line comments.

Library Imports

Like many software programming languages, VHDL code starts with a list of statements that describe which pre-existing libraries should be imported. The pre-generated code should already include an import for `ieee.std_logic_1164.all`. Now add another import for `ieee.numeric_std.all`, as shown in Listing 2.1. The appropriate line already exists as a comment in the generated file, and can be included by uncommenting it. These two libraries (`std_logic_1164` and `numeric_std`) are common in many VHDL designs.

Listing 2.1: Suggested VHDL library imports.

```
1 library ieee;                    -- from the library called ieee:
2 use ieee.std_logic_1164.ALL;     -- IEEE standard logic definitions
3 use ieee.numeric_std.ALL;        -- basic operations on numbers
```

Entity and Architecture Declarations

You may be familiar with the concept of separating the *interface* from the *implementation* – for example, the separation of declaration and definition of a class in .h and .cpp files in C++. VHDL embraces this concept for digital design: each module to be implemented must have an *entity* declaration for the interface of the module, and one or more *architecture* declarations that describe the internal implementation of the module. In object-oriented design terms, an architecture declaration can be thought of as an implementation of an abstract class (which is the entity declaration).

Since we already specified the inputs and outputs, the entity declaration for the module has already been created by the wizard, and should look as in Listing 2.2. Here you can see how the named input/output ports are described in VHDL syntax, all of which have the type `std_logic`, which describes a binary value[4]. There may be an additional section in the entity declaration that allows parametrization, which we will cover later.

Listing 2.2: Entity declaration for the module.

```
1 entity tutorial is
2     Port ( A : in STD_LOGIC;
3            B : in STD_LOGIC;
4            C : in STD_LOGIC;
5            X : out STD_LOGIC;
6            Y : out STD_LOGIC;
7            Z : out STD_LOGIC);
8 end tutorial;
```

[4] Actually `std_logic` allows values other than 0 and 1, but for this course assume only 0 and 1. You may also encounter the value 'U' (uninitialized) and 'X' (indeterminable) when debugging.

Signals and Logic Operators

In VHDL, *signals* are named objects that carry a value. Despite how it sounds, this is **not** the same as a variable in software programming languages. A VHDL signal always has the value of its *driver*, which may be a constant value, an input port of the module, or another signal (which needs a driver itself). Usually, the term "signal" refers to an internal signal of the architecture. All input/output ports in the entity declaration are also signals, but with some restrictions: from inside the module, input signals cannot be driven (read-only) and output signals cannot be read (write-only).

The *signal assignment* operator is used to connect driving and driven signals, as in `drivenSignal <= drivingSignal`, and can be directly used inside the architecture body[5]. Note that this not a one-off assignment but is **always active**; so whenever the driving signal changes value, the driven signal will also change value.

Logic operators can be applied on the right-hand side (driving) signals during the assignment. Listing 2.3 shows how signals can be declared and assigned using logic operators. This architecture should now exhibit the desired behavior according to the specifications.

Listing 2.3: Signal declaration, assignment and logic operators.

```
1  architecture Behavioral of tutorial is
2      -- declare some internal signals
3      signal tempSignal1, tempSignal2 : std_logic;
4  begin
5      -- architecture body
6      -- calculate the internal signal values
7      tempSignal1 <= B or C;
8      tempSignal2 <= B xor C;
9      -- drive the outputs
10     X <= A and B;
11     Y <= tempSignal1;
12     -- note the syntax of std_logic constant ('1')
13     -- and use of the 'when' keyword to create a multiplexer
14     Z <= tempSignal1 when A = '1' else tempSignal2;
15 end Behavioral;
```

2.2.3. Step 3: Simple Simulation

Designing hardware is hard, but checking what you designed is perhaps even more difficult. Digital designers typically use simulations to check if the design is behaving as intended before turning it into real hardware. HDL designs are often *reactive*, in the sense that the circuit reacts to the input. Therefore, it is desirable to stimulate the input signals to the design to see how it reacts. Creating the stimuli can be done in a number

[5]Note that only "concurrent statements" can be written outside `process` statements like we're doing here. Processes will be introduced later on.

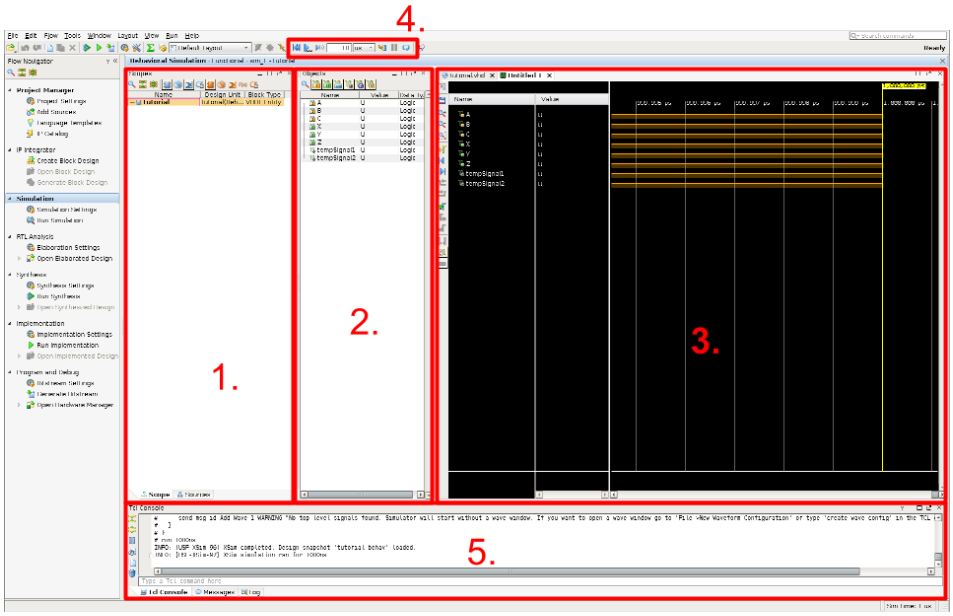


Figure 2.2.: ISim, Vivado's built-in simulator.

of ways: for the simplest of designs – like the one we have just made – it is possible to use a GUI tool inside the development environment to set the input values over time and observe how it reacts.

To initiate a simulation, click "Run Simulation" in the Flow Navigator pane and "Run Behavioral Simulation" from the resulting menu. This will launch the ISim simulator as shown in Figure 2.2. The enumerated parts of the window are described below:

1. **Scopes:** Similar to the Hierarchy panel in Vivado, can be used to inspect the modules and submodules in the current design.

2. **Objects:** Lists the signals of the module currently selected in "Scopes". You can right-click a signal to perform different operations on or with it.

3. **Waveform:** Shows the values and status[6] of signals over time. You may zoom in and out by holding the Ctrl-key while using the scroll wheel on your mouse, and you may scroll left and right by holding the Shift-key while using the scroll wheel. You may right-click signals to operate on them. Useful tools are to select a different radix; or to group signals by first making a selection (hold Ctrl or Shift while left-clicking signals), right-clicking the selection and clicking "New Group".

[6]The color of each signal indicates its drive status. Un-driven signals are orange, "normal" signals are green. If you have red (multiple-driver) signals, this indicates problems with the design.

4. **Simulation Control:** The simulation can be run further or re-started from the control panel toolbar above the waveform.
5. **Console:** Feedback messages from the simulator. Text command input is also available for assigning values to signals or controlling the simulation.

The simulator assumes input stimuli have already been created when first launched, and you will notice that the simulation has already run for 1 microsecond. Since we have not specified how the inputs should be driven, all signals in the waveform display are orange (non-driven/"floating"). We will now create input stimuli that allows us to observe how the outputs behave when the inputs change. Since this is a small design with only three inputs and no memory, we can easily test all possible input combinations and observe the outputs. To do this, we will create three oscillating square waves with periods T, $2T$ and $4T$ that will drive the inputs A, B and C. There are two alternatives to create the input stimuli:

Using the ISim GUI

Right click the desired signal and choose "Force Clock" to create a square wave driver. Set the parameters as follows for a square wave with $T = 10$ ps:

- Value Radix: Binary
- Leading Edge Value: 0
- Trailing Edge Value: 1
- Starting at Time Offset: 0 ps
- Cancel after Time Offset: (leave blank)
- Duty Cycle (%): 50
- Period: 10 ps

Use the ISim command console

Enter the following commands in the console to create square waves for A, B and C:

```
1 add_force {/tutorial/A} -radix bin {0 0ns} {1 5ps} -repeat_every 10ps
2 add_force {/tutorial/B} -radix bin {0 0ns} {1 10ps} -repeat_every 20ps
3 add_force {/tutorial/C} -radix bin {0 0ns} {1 20ps} -repeat_every 40ps
```

When the stimuli are created, run the simulation for $4T = 40$ ps using the control toolbar or entering `run 40 ps` in the command console. Examine the generated waveform, which should be the same as shown in Figure 2.3. Verify that the outputs X, Y and Z are appropriately set for the changing input values of A, B and C. Use the restart button on

Figure 2.3.: Expected waveform for the tutorial module.

the simulation control toolbar (or enter the `restart` command in the console) to restart the simulation from the beginning, but note that this will remove the signal drivers.

2.2.4. Step 4: Testing on the FPGA

An FPGA can be used to make *reconfigurable hardware*. More specifically, an FPGA is an integrated circuit which can be reconfigured to perform any logic function, contrary to Application-Specific Integrated Circuits (ASICs) whose functionality is fixed. Now that we are sure of the tutorial module design behaves as expected, we can deploy it on an FPGA and observe its operation on real hardware[7].

To bridge the gap between the abstract HDL design and the physical FPGA chip, some additional effort is needed. The *synthesis tool* handles this process and produces a *bitfile* that can be used to configure the FPGA, not unlike how a compiler turns code into programs. These actions are launched in the Flow Navigator Vivado (panel 1 in Figure 2.1). The first stage converts the HDL description into a logic gate description by pattern-matching HDL fragments and translating them into basic blocks, as described in Appendix B. You can initiate this action and view the RTL schematic by clicking the "Open Elaborated Design" item under the "RTL Analysis" heading in the Flow Navigator pane. This will open a schematic as seen in Figure 2.4. When working with more complicated designs, checking that this schematic corresponds with the one you sketched before writing the VHDL code is a good way of checking that your description is correct. It is also a useful tool for understanding how the synthesis tool translates your code into hardware. The second stage implements this circuit using FPGA resources (look-up tables and flip-flops). This can be run by clicking the "Run Synthesis" item under the "Synthesis" heading in Flow Navigator. You may find it interesting to compare this schematic ("Synthesized Design" → "Schematic") with the RTL schematic.

[7]Strictly speaking, the FPGA is only a highly parallel computing platform that simulates the HDL design, but the same argument can also be made for transistors.

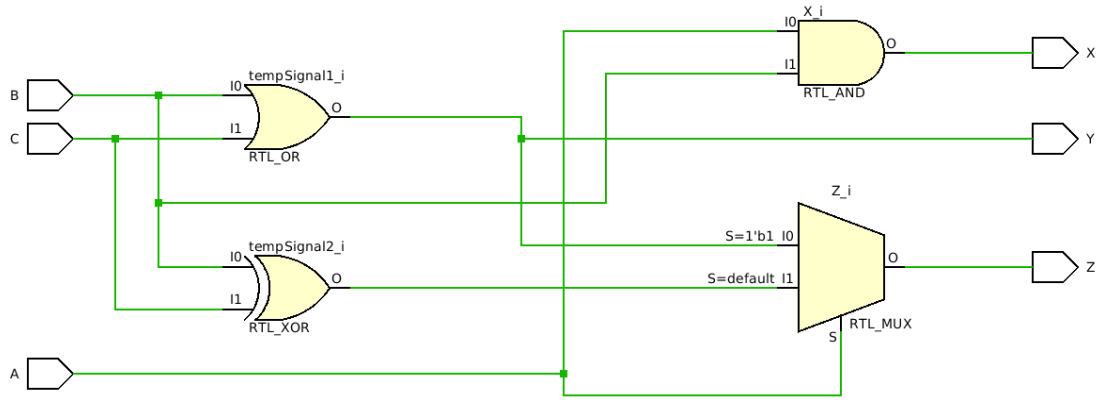


Figure 2.4.: RTL diagram of the tutorial module.

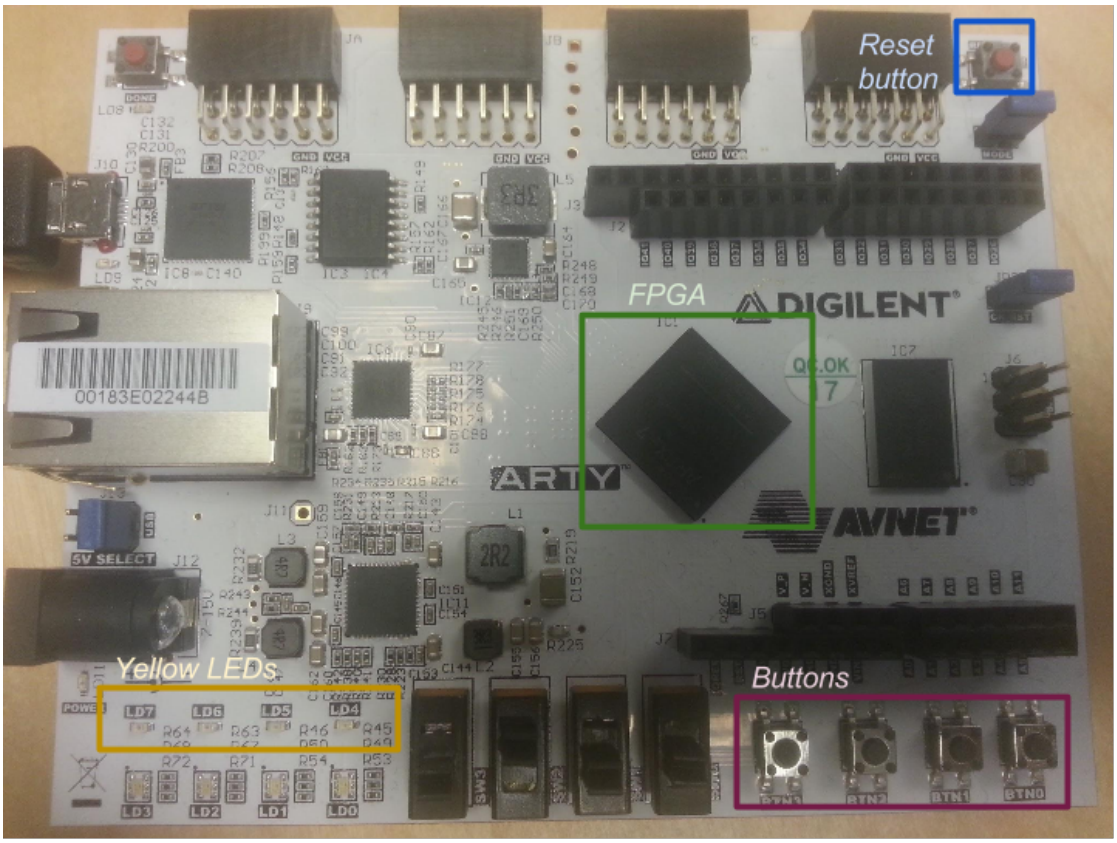


Figure 2.5.: The Artix-7 35T Arty FPGA Evaluation Kit. The FPGA, active-low reset button, yellow LEDs and active-high push-buttons are highlighted.

There is one more thing that needs to be done before we can test our design on the FPGA. We will be using an FPGA evaluation kit called Arty (depicted in figure 2.5), and we wish to connect our module's inputs and outputs to buttons and LEDs on this board, respectively. The LEDs and buttons we will be using are connected to specific FPGA pins. To ensure that our tutorial module's inputs and outputs correspond to the correct pins, we provide the toolchain with a *constraints file* which specifies how signals map to pins. The constraints file can also contain other information about physical characteristics of the FPGA system; in our case, we need to set a configuration voltage setting. Click "Add Sources", select "Add or create constraints", click "Create File" in the next wizard window and call the file `constraints`. This will create an empty `constraints.xdc` file, which you can find by first first clicking on "Project Manager" in the Flow Navigator, and then expanding the "Constraints/constrs_1" folder you can find in the "Sources" pane (pane number 2 in Figure 2.1). Open the `constraints.xdc` file by double-clicking it, and fill it with the following content[8]:

```
1  #### Constraints for the LEDs
2  set_property -dict {PACKAGE_PIN H5 IOSTANDARD LVCMOS33} [get_ports {X}];
3  set_property -dict {PACKAGE_PIN J5 IOSTANDARD LVCMOS33} [get_ports {Y}];
4  set_property -dict {PACKAGE_PIN T9 IOSTANDARD LVCMOS33} [get_ports {Z}];
5
6  #### Constraints for the push-buttons (input)
7  set_property -dict {PACKAGE_PIN D9 IOSTANDARD LVCMOS33} [get_ports {A}];
8  set_property -dict {PACKAGE_PIN C9 IOSTANDARD LVCMOS33} [get_ports {B}];
9  set_property -dict {PACKAGE_PIN B9 IOSTANDARD LVCMOS33} [get_ports {C}];
10
11 #### Configuration voltage settings specific to the Arty board.
12 set_property CONFIG_VOLTAGE 3.3 [current_design]
13 set_property CFGBVS VCCO [current_design]
```

After you have added the constraints file, you can simply click the "Generate Bitstream" action under the "Program and Debug" heading in Flow Navigator to automatically go through the rest of the steps. After a while, this should result in a file `tutorial.bit` in the project sub-folder `tutorial.runs/impl_1`.

Ensure that the board is connected to the lab machine. You can then launch the `hostcomm` utility through the Unity launcher or by typing `hostcomm` in a terminal, and upload the `tutorial.bit` bitfile to the FPGA. You should now be able to use the three right-most push buttons on the FPGA card to provide inputs to the module, and observe the outputs on the three right-most LEDs in the lower left corner. Remember that FPGAs store configuration in volatile memory: the FPGA must be reconfigured each time after the board is powered off and on[9].

[8] You do not have to learn the constraint syntax, these files will be provided for you in the lab.

[9] Special-purpose flash memories exist to circumvent this issue, although we won't be using them in this course.

2.3. Part 2: Sequential Logic and Testbenches

We will now extend the design from Part 1 with a blinking LED, and learn about *sequential logic* and how it is described in VHDL in the process. The updated specification for the module will be as follows:

- The module will have three binary (or boolean) input ports, A, B and C.
- The module will have three binary output ports, X, Y and Z.
- The module will have a *clock input*, clk, that will be connected to a 100 MHz clock source.
- The relation of the outputs to the inputs shall be as follows:
  - Y = B OR C
  - Z = (B OR C) when A = 1, (B XOR C) otherwise
- The output X will be used to blink an LED. Its value should take the form of a square wave with a period of two seconds: 1 for one second, and 0 for one second.

2.3.1. Step 1: Thinking in RTL

When designing hardware to solve a problem, the most important step is not writing the HDL code itself, but rather envisioning what the hardware will look like. So how does one envision hardware from a given description? The key to digital hardware design is understanding how data is stored, transformed and moved over time. In this lab, we encourage you to develop your design as a sketch on paper by thinking in terms of *RTL Building Blocks* described in Appendix B. Representing a hardware design as a sketch that describes how data is stored, transformed, and carried around makes it easy to write synthesizable VHDL afterwards.

Designing with "Black Boxes"

Often, the design specifications will not be simple enough to identify all the necessary building blocks and how to connect them all at once. Modeling some parts of the design as a "black box" can be helpful when starting a sketch. In this case, we already know (from Part 1 of the tutorial) how to use logic operators and a multiplexer for the outputs Y and Z, but the "blinking" output X may not be as easy. Therefore, as can be observed in Figure 2.6, we model it as separate module called `blinky` while sketching the top-level schematic, whose contents we do not know yet. We can, however, reason about the inputs and outputs of the module itself: the output of the box must be connected to the output X, and since the behavior of the box itself only depends on time and no other inputs, the input of the box must be connected to the clock input.

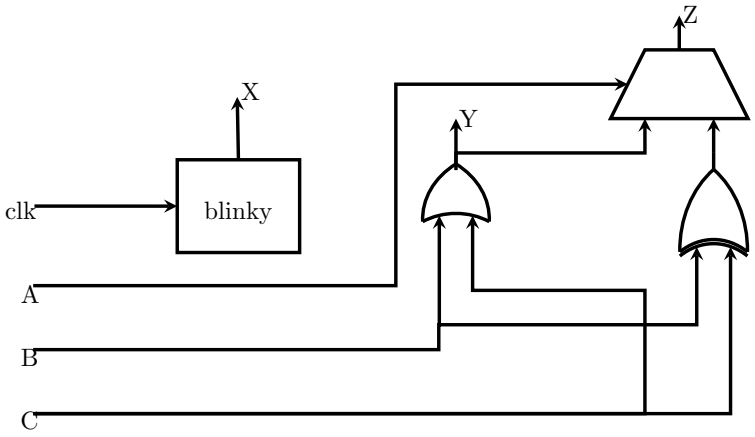


Figure 2.6.: RTL top-level sketch for the new specification.

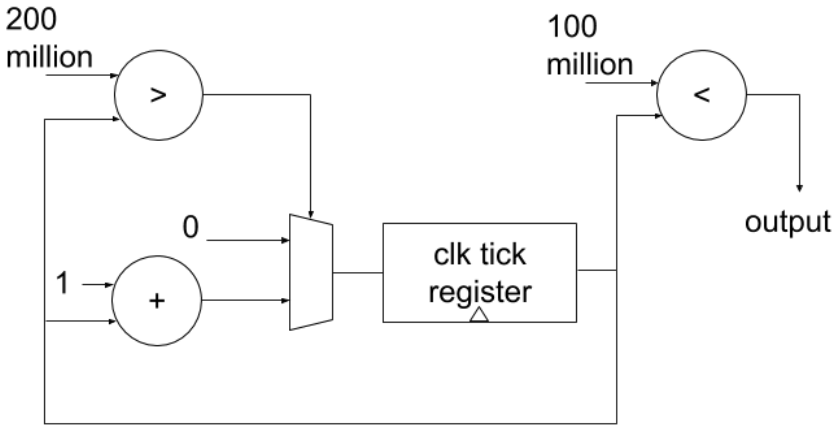


Figure 2.7.: RTL sketch for the `blinky` module.

RTL sketching with building blocks

We will now create an RTL sketch for the "blinky" module by piecing together a group of RTL building blocks. The requirement from this module is that it outputs 0 for one second, and afterwards outputs 1 for one second. We have a 100 MHz clock input that supplies us with 100 million ticks every second. By counting the clock ticks, we are able to measure how much time has passed. We therefore introduce a register that will count the clock ticks, through incrementing its value by one every clock cycle. We generate the output of the module by comparing the number of current ticks to 100 million: if the elapsed time is less than 1 second, the comparator will output a 1, otherwise it will output a 0. The periodic behavior is ensured by using another comparator and a multiplexer: when the number of ticks is greater than 200 million, the tick count register will be reset to zero. Figure 2.7 depicts the completed RTL sketch for the `blinky` module.

2.3.2. Step 2: More VHDL Background

Before we start coding VHDL for the module, we will first introduce several important concepts in VHDL required to implement sequential logic.

VHDL Processes

A central concept in VHDL is a *process*, which is a group of statements with a *sensitivity list*. The sensitivity list contains a list of signals to which the process is *sensitive*; whenever one of these signals changes value, the contents of the process will be re-evaluated. Recall how the signal assignment operator `<=` was "always active"; that is actually true only when the statement is placed outside a process. When the signal assignment statement is inside a process, the assignment will be triggered only when something in the sensitivity list changes. Consider the code example below, which is a re-write of the first part of the tutorial and does the same thing as Listing 2.3. Observe that `tempSignal1` and `tempSignal2` are now assigned inside a process named `CalcTempSignals`.

Listing 2.4: Combinational process.

```
1  architecture Behavioral of tutorial is
2      signal tempSignal1, tempSignal2 : std_logic;
3  begin
4      -- architecture body
5      CalcTempSignals: process(B, C)
6      begin
7          tempSignal1 <= B or C;
8          tempSignal2 <= B xor C;
9      end process;
10     -- ...rest of code is identical to Part 1
11 end Behavioral;
```

The process example above contains all the signals it reads (B, C) in the sensitivity list. This is called a *combinational process*. The benefits of using combinational processes instead of writing assignment statements directly inside the architecture body is twofold: it helps with the organization of the code (somewhat similar to functions in software programming) and allows using certain VHDL language constructs that are not allowed directly inside the architecture body. Using explicit processes can increase the chance of including bugs as described in Appendix C, however, and may also be more verbose. The effects are otherwise very similar; both approaches are used for creating *combinational logic*, which is characterized by the outputs depending only the current inputs.

Sequential Logic

Most digital hardware designs requires memory or data storage and are implemented with *sequential logic*, where (unlike combinational logic) outputs depend on the past inputs, also called the *state* . Such hardware is described with the help of *synchronous processes* in VHDL. A synchronous process contains only the clock (and often reset) signal in its sensitivity list. This means the assignments inside are only triggered when the clock value changes (usually from 0 to 1, on the "positive clock edge"). Listing 2.5 shows how a D-type flip-flop can be modeled with a VHDL synchronous process. Note that the value of `data_in` is copied to `data_out` only on the positive clock edge; this implies that the value of `data_out` is stored until the next clock edge.

Listing 2.5: A synchronous process describing a D flip-flop.

```
1  entity DFlipFlop is
2      port (
3          data_in : in std_logic;
4          data_out : out std_logic;
5          clk : in std_logic
6      );
7  end DFlipFlop;
8
9  architecture Behavioral of DFlipFlop is
10 begin
11     -- architecture body
12     DFFBehaviorProcess: process(clk)
13     begin
14         if rising_edge(clk) then
15             -- copy input to output on rising clock edge
16             data_out <= data_in;
17         end if;
18     end process;
19 end Behavioral;
```

If you were left wondering what the value of `data_out` is *before* the first rising clock edge, that is what we will be covering next. Sequential logic often contains a *reset signal*

that is used to bring the design into some initial state, and assigns a default value to all registers or flip-flops. Listing 2.6 enhances the previous D flip-flop example with a synchronous[10] reset signal which sets its default value to zero.

Listing 2.6: D flip-flop with reset.

```
1  entity DFlipFlop is
2      port (
3          data_in : in std_logic;
4          data_out : out std_logic;
5          clk, reset : in std_logic
6      );
7  end DFlipFlop;
8
9  architecture Behavioral of DFlipFlop is
10 begin
11     -- architecture body
12   DFFBehaviorProcess: process(clk)
13   begin
14     if rising_edge(clk) then
15       -- note that reset takes priority when active;
16       if reset = '1' then
17         -- value on reset
18         data_out <= '0';
19       else
20         -- copy input to output on rising clock edge
21         data_out <= data_in;
22       end if;
23     end if;
24   end process;
25 end Behavioral;
```

Number Crunching in VHDL

The final piece of the puzzle we will need to build `blinky` in VHDL is how to represent numbers and perform operations on them. The first thing you must ensure before you can work with numbers is importing `ieee.numeric_std.ALL` in the beginning of your code (see Section 2.2.2), this defines arbitrary-bit-wide integers and operators on them[11]. Digital circuits always represent and process all information as ones and zeroes, but the definitions in this library allow us to work at a higher abstraction level. Listing 2.7 shows an example of an adder for 32-bit unsigned numbers. Note that the input and output ports are still specified as `std_logic_vector` types in the code. Since all types boil down to wires and binary signals in hardware, this is a way of making different modules easier to connect to each other.

[10] A synchronous signal is aligned with the clock edges; it takes effect only at a rising or falling edge.

[11] VHDL itself actually defines integers, but you should use the IEEE standard types `unsigned` and `signed` while making hardware that operates on numbers.

Listing 2.7: Combinational adder for unsigned numbers.

```
1  entity CombinationalAdder is
2      port (
3          A, B : in std_logic_vector(31 downto 0);
4          C : out std_logic_vector(31 downto 0)
5      );
6  end DFlipFlop;
7
8  architecture Behavioral of CombinationalAdder is
9      signal addResult : unsigned(31 downto 0);
10 begin
11     -- architecture body
12     AdderProcess: process(A, B)
13     begin
14         -- we need a type conversion to get VHDL to treat the
15         -- inputs A and B as unsigned numbers
16         addResult <= unsigned(A) + unsigned(B);
17     end process;
18     -- convert result back to logic vector and write to C
19     C <= std_logic_vector(addResult);
20 end Behavioral;
```

2.3.3. Step 3: From RTL to VHDL

We will now describe the process of converting from the RTL sketch into VHDL by taking `blinky` as an example. Note that there is no single "correct solution" here; there are different ways of describing the same design in VHDL.

Since we will be designing `blinky` as a separate VHDL module, the first step is to create a new VHDL module with this name and three ports: two input ports called `clk` and `reset`, and an output port called `pulse`. You can use the "Add Sources" wizard as described Section 2.2.2. Also remember to add `use ieee.numeric_std.all;` in the beginning of the code.

Generics in VHDL

The concept of *generics* is useful when you need to parametrize some part of your modules at design-time. For example, the pulse duration for the LED depends on the system clock in our case. If the system clock was running at 10 MHz instead of 100 MHz, we would have to change the number of ticks the counter goes up to before resetting to zero. By using generics as part of the entity declaration, we can make the number of ticks a changeable part of the entity's interface, without having to change anything in the implementation itself. This is similar to generics in Java or templates in C++. Listing 2.8 shows how the entity declaration for `blinky` can be expanded to include a configurable number of clock ticks. Note that the generics have default values set; this

means that they if they are unassigned while instantiating this entity, they will take on these default values.

Listing 2.8: Entity declaration for `blinky` with parametrizable tick count.

```
1  entity blinky is
2      generic (
3          -- note that the generics are declared as integers
4          -- since they simply express the constant values,
5          -- no need to use signed/unsigned types here
6          ticksBeforeLevelChange : integer := 100000000;
7          ticksForPeriod : integer := 200000000
8      );
9      port (
10         clk, reset : in  STD_LOGIC;
11         pulse : out  STD_LOGIC
12     );
13 end blinky;
```

Building the counter

You may recall that the heart of `blinky` was a counter register that incremented by one for each clock tick, and reset to zero when it reached 200 million. Having covered the concept of sequential logic and synchronous processes, we now should be able to model the behavior of this component. Listing 2.9 shows how the counter register can be built with a single sequential process. Afterwards, a single `when` statement in the architecture body is enough to create the comparator and drive the `pulse` output high or low, depending on the tick counter value.

Listing 2.9: Implementation for `blinky`.

```
1  architecture Behavioral of blinky is
2      signal tickCount : unsigned(31 downto 0);
3  begin
4      -- clock tick counter implementation
5      CountPeriodTicks: process(clk)
6      begin
7          if rising_edge(clk) then
8              -- The reset button is active-low, so if the reset signal
9              -- is '0' the reset button is being pressed.
10             if reset = '0' then
11                 -- note the others <= '0' syntax; this is useful
12                 -- for generating a sequence of zeroes of the appropriate
13                 -- length.
14                 tickCount <= (others => '0');
15             elsif tickCount < ticksForPeriod then
16                 -- period not complete, increment ticks
17                 tickCount <= tickCount + 1;
```

```
18              else
19                  -- period reached, back to zero ticks
20                  tickCount <= (others => '0');
21              end if;
22          end if;
23      end process;
24
25      -- drive the output depending on the counter value
26      pulse <=    '0' when tickCount < ticksBeforeLevelChange
27    else '1';
28  end Behavioral;
```

Putting it all together

You should now have two VHDL modules, `tutorial` and `blinky`, in the project. These modules are completely separate for the moment. The final step for completing the design will be instantiating `blinky` inside `tutorial` and linking its output to the X output port. This is done with an instantiation statement in the architecture body that specifies values for generics and connects the ports of the instantiated component as desired. The updated version of the `tutorial` module is displayed in Listing 2.10. The changes from the Part 1 code are:

- the addition of `clk` and `reset` to the `tutorial` entity declaration
- the instantiation for `blinky`
- removing the `X <= A and B;` statement from the architecture body

Also note that the tick configuration is set to smaller values to see the results faster in simulation. Simply commenting out the `generic map` line will use the default values when testing on the real FPGA later on.

Listing 2.10: Instantiating `blinky` in `tutorial`.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity tutorial is
5    port (
6      X : out  STD_LOGIC;
7      Y : out  STD_LOGIC;
8      Z : out  STD_LOGIC;
9      A : in  STD_LOGIC;
10     B : in  STD_LOGIC;
11     C : in  STD_LOGIC;
12     clk, reset : in std_logic
13   );
14 end tutorial;
15
```

```
16 architecture Behavioral of tutorial is
17   signal tempSignal1, tempSignal2 : std_logic;
18 begin
19   -- architecture body
20   -- instantiate blinky
21   -- comment out the generic map line to restore default values
22   BlinkyInst: entity work.blinky
23   generic map (ticksBeforeLevelChange => 100, ticksForPeriod => 200)
24   port map (clk => clk, reset => reset, pulse => X);
25
26   -- drive the internal signals from inputs
27   DriveInternalSignals: process(B, C) is
28   begin
29     tempSignal1 <= B or C;
30     tempSignal2 <= B xor C;
31   end process DriveInternalSignals;
32
33   -- drive the outputs
34   Y <= tempSignal1;
35   Z <= tempSignal1 when A = '1' else tempSignal2;
36 end Behavioral;
```

2.3.4. Step 4: Testbenches in VHDL

In the previous part of the tutorial, we covered how to use ISim interface to generate stimuli and verify VHDL designs. As the designed hardware becomes more complex, manually setting signal values at the correct time can become complex and error-prone. The alternative to this method is to create a *testbench* in VHDL. A testbench is a VHDL program that instantiates a component and uses processes to drive and observe its signals. Since testbenches are not intended to be synthesized into hardware themselves, the full range of VHDL language constructs are available while writing them. To test our new `tutorial` module design, we will be using a testbench.

To add a test bench, click Add Source and select "Add or create simulation sources". Click "Create File", and name the test bench `tb_tutorial`. For test bench entities no inputs and outputs are required, so the port list can be empty. Having created the file, the next task is to fill it with content, so double-click the file to open it. When writing test benches one will often start by writing the same kind of "boiler plate" code, as all test benches in general must:

1. Instantiate the design being tested, aka Design Under Test (DUT).
2. Create a clock signal to drive sequential logic in the DUT.
3. Use a process to create test stimuli for the inputs to the DUT.

For our case, this results in the test bench skeleton code in Listing 2.11.

Listing 2.11: Test bench boilerplate.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  -- The test bench does not need inputs/outputs, so the entity is empty.
5  entity tutorial_tb is
6  end entity tutorial_tb;
7
8  architecture Behavioral of tutorial_tb is
9    -- We need signals to connect to the ports of the DUT.
10   signal X            : std_logic;
11   signal Y            : std_logic;
12   signal Z            : std_logic;
13   signal A            : std_logic;
14   signal B            : std_logic;
15   signal C            : std_logic;
16   signal reset        : std_logic;
17   signal clk          : std_logic := '0';
18   constant CLK_PERIOD : time      := 10 ns;
19 begin
20   -- clock generation: flip value after half a clock period.
21   clk <= not clk after CLK_PERIOD/2;
22
23   -- We instantiate the DUT, and connect it to the test bench signals.
24   DUT : entity work.tutorial
25     port map (
26       X     => X,
27       Y     => Y,
28       Z     => Z,
29       A     => A,
30       B     => B,
31       C     => C,
32       clk   => clk,
33       reset => reset);
34
35 -- Stimulus process: inside this process the inputs of the DUT are
36 -- set, and the outputs are checked.
37   stim_proc : process is
38   begin
39     wait;
40   end process;
41
42 end architecture Behavioral;
```

The next step is to add appropriate stimuli code to the `stim_proc` process. Many features are available while writing VHDL testbenches, including file and console I/O, random value generation, timed/conditional waits and assertions, which we will not cover here. By taking advantage of the fact that all VHDL processes run in parallel, a diverse range of input stimuli can be created to provoke different types of behavior in the tested design. Listing 2.12 shows an example implementation for the `stim_proc` process that prints the value of the X, Y, Z outputs every 100 cycles, and shifts and negates the values of the A,

B and C inputs. Once the stimuli generation code is added into `tb_tutorial.vhd`, you can use ISim to run the simulation in the same manner as described in Section 2.2.3.

Listing 2.12: Stimulus process for the testbench.

```
-- Stimulus process
stim_proc: process
begin
  report "Hello world!";
  -- intial values for the inputs
  A <= '0'; B <= '1'; C <= '0';
  -- hold reset state for 100 ns.
  -- Note: reset is active low.
  reset <= '0';
  wait for 100 ns;
  reset <= '1';

  wait for 50*clk_period;
  -- display X values over time
  for i in 0 to 10 loop
    report  "X = " & std_logic'image(X)
            & " Y = " & std_logic'image(Y)
            & " Z = " & std_logic'image(Z);
    -- change the inputs
    A <= not B; B <= not C; C <= not A;
    -- wait for 100 clock cycles
    wait for 100*clk_period;
  end loop;
  wait;
end process;
```

You should be able to see the debug output on the ISim console, as well as observe the signal changes in the waveform display. Verify that the X value changes between 0 and 1 at every iteration of the printing (you can use `run 10 us` in the console), and that Y and Z are set to correct values depending on the input.

2.3.5. Step 5: Testing on the FPGA

Having verified that our design works as expected with the testbench, we can now deploy it on the FPGA and blink the LED. To do this, first switch back to the `tutorial.vhd` file (click "Project Manager" in Flow Navigator), and comment out the `generic map` line in the `blinky` instantiation, which will restore the tick counter limits to the correct values for a 100 MHz clock. You also have to add several lines to the user constraints file, as shown in Listing 2.13.

Listing 2.13: Final version of constraints file.

```
#### Constraints for the LEDs
set_property -dict {PACKAGE_PIN H5 IOSTANDARD LVCMOS33} [get_ports {X}];
set_property -dict {PACKAGE_PIN J5 IOSTANDARD LVCMOS33} [get_ports {Y}];
set_property -dict {PACKAGE_PIN T9 IOSTANDARD LVCMOS33} [get_ports {Z}];

#### Constraints for the push-buttons (input)
set_property -dict {PACKAGE_PIN D9 IOSTANDARD LVCMOS33} [get_ports {A}];
set_property -dict {PACKAGE_PIN C9 IOSTANDARD LVCMOS33} [get_ports {B}];
set_property -dict {PACKAGE_PIN B9 IOSTANDARD LVCMOS33} [get_ports {C}];

#### Configuration voltage settings specific to the Arty board.
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

#### Constraints for clock
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports {clk}];
create_clock -add -name clk -period 10.00 -waveform {0 5} [get_ports {clk}];

## Reset is connected to the upper-right-corner red RESET button.
set_property -dict {PACKAGE_PIN C2 IOSTANDARD LVCMOS33} [get_ports {reset}];
```

One all of this is done, you can click the Generate Bitstream to re-run the synthesis operations, and upload the resulting bitfile to the FPGA (as described in Section 2.2.4). You should now observe that the LED labeled LD4 is blinking with the expected pattern. The other two LEDs will be controlled by the buttons, as in Part 1.

You may also want to check the RTL diagram generated by the synthesis tool (Elaborated Design → Schematic) and compare it to our original sketch. If you zoom into the `blinky` instantiation, you will notice that the generated RTL (shown in Figure 2.8) is very similar to our original sketch.

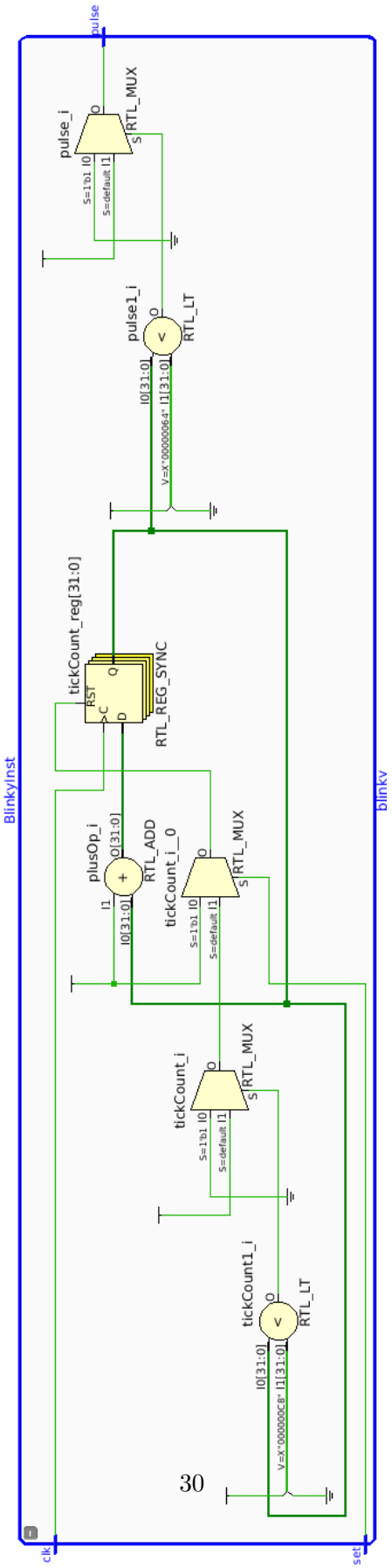


Figure 2.8.: RTL for `blinky` generated by the synthesis tool.

3. Exercise 0 - Implementing a Simple Stack Machine

In this chapter, you will use what you have learned from the previous two chapters to build a small stack machine, which will implement a reverse polish notation (RPN)[1] calculator. The exercise is a step up in complexity from the previous tutorials, but should provide a stepping stone to exercise 1 and 2.

The first section will present the specification for the machine. As opposed to exercise 1 and 2, the next section will then break up the problem into bite-sized sub-problems you will then be asked to solve. You are not required to write a report for this exercise, although you must still deliver the files specified in Section 3.3.3.

3.1. Problem Specification

The year is 2257 AD and a team of hardware archaeologists has just finished an excavation in the bustling metropolis of Trondheim. The dig has unearthed many technological artifacts from bygone eras, including a fascinating piece of early computer technology called a *stack machine*. Your company has won a contract to recreate a simple but functional stack machine for the Computer Museum, which will perform addition and subtraction on 8-bit signed numbers. The piece of circuitry you have been tasked to create, will provide the main computation engine in a stack machine. A separate team will create an instruction buffer, which will be filled with instructions the computation engine should execute. The complete architecture is illustrated in Figure 3.1.

The communication between the computation engine and the instruction buffer will be controlled using the two signals `read` and `empty`. If the `empty` signal is set high, the instruction buffer is empty and the `read` signal should not be set high. Otherwise, the `read` can be set high to request a new instruction. A new instruction is transferred using an `instruction` signal the cycle after `read` was set high. A timing diagram depicting this behaviour reading from a buffer with two elements can be seen in Figure 3.2.

The instructions will be 16-bit wide. The instruction format and the semantics of their fields are illustrated in Figure 3.3. The figure also demonstrates the operation of the instructions with a few examples. The *push* instruction should push the immediate operand value on the stack. The *add* instruction should pop the two top values off of

[1] `http://en.wikipedia.org/wiki/Reverse_polish_notation`

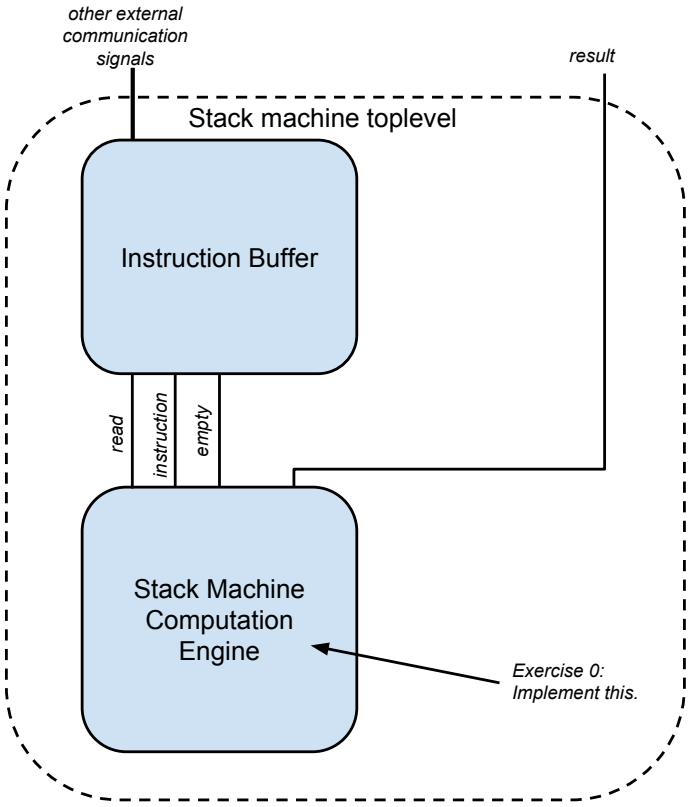


Figure 3.1.: The top level architecture of the stack machine.

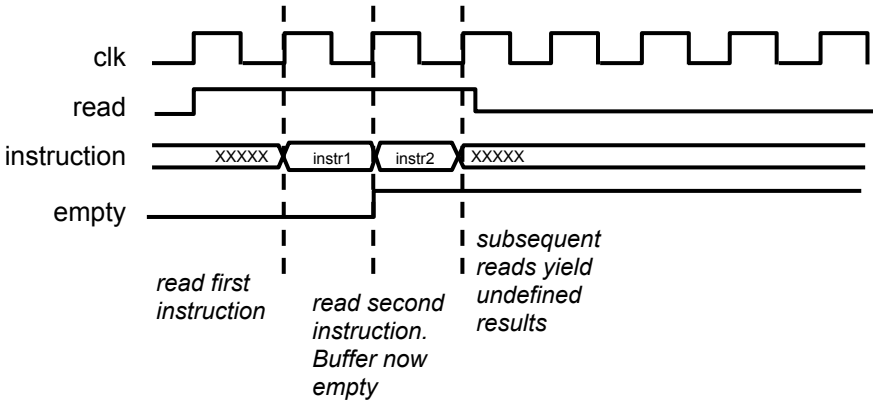


Figure 3.2.: Timing diagram depicting instruction read behaviour.

the stack, add them, and push the result back on the stack. The *sub* instruction should pop the two top values off the stack, subtract the top operand from the bottom one, and push the result back on stack.

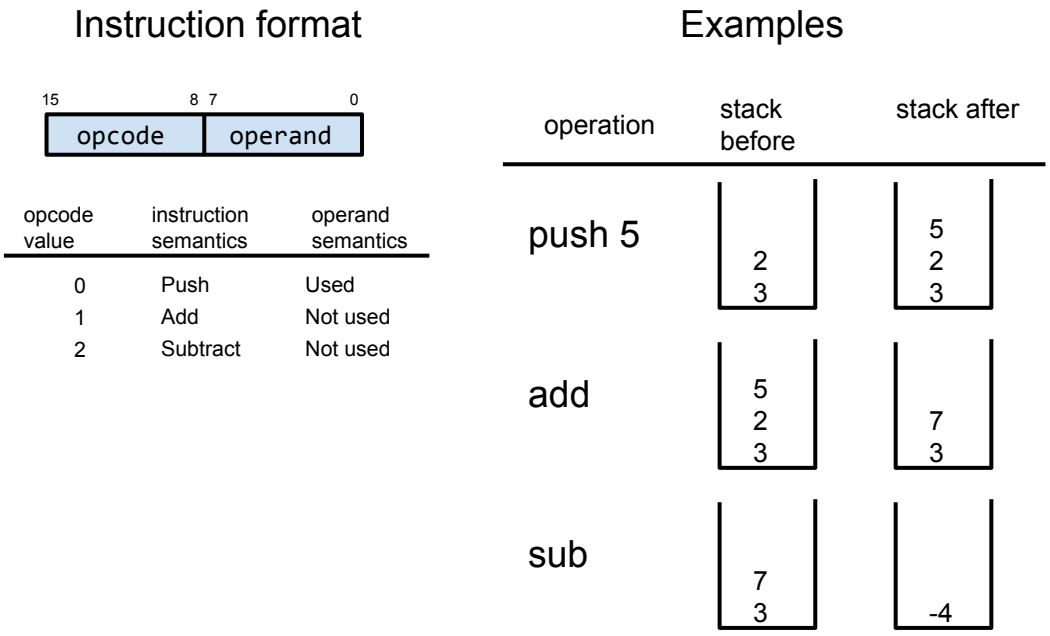


Figure 3.3.: Examples demonstrating the operation of the stack machine.

Once the `empty` signal transitions from high to low, there are instructions available in the buffer. The execution engine is then required to start processing the available instructions. The `empty` signal can make several transitions between high and low during the execution of a program. Additionally, the top of the stack should be exported to be able to use the result of a computation. If there are no elements on the stack, zero should be output so that the output is well-defined at all times.

The computation engine is allowed to assume that invalid operations will not be attempted. Specifically, the implementation may assume that the stack never overflows or underflows. An additional permissible assumption is that the value of `instruction` will not change before a new read request has been issued. Clock and reset signals must be included. You may make further simplifying assumptions about the specification, should you find it necessary.

3.2. Initial Design Effort

To guide the solution, and more easily create bite-sized tasks, we will in this section guide you through an initial design effort, producing an RTL top-level description of the

stack machine computation engine. Our approach will be to follow the workflow outlined in section 1.2, demonstrating in the process how the workflow may be used.

3.2.1. Reading and understanding the specification

The first step is reading the specification, and getting a feeling for how the entire circuit is supposed to work. If you are not comfortable with RPN, you should try creating and calculating a few expressions. There are several web resources available which may help in understanding RPN[2].

Reading the specification, we can note the following points:

- The execution is pull-driven: the computation engine decides when to get the next instruction for the buffer, which means it can use as many cycles as it wants for each operation.
- The computation engine is required to pull a new instruction if it is not busy and there are instructions available.
- The `empty` signal can switch between high and low multiple times during execution. This means that the computation engine cannot assume that it can permanently stop doing work when the empty signal is set high. Rather, it must be prepared for the empty signal to go low again, and then resume computation.
- No external memory is made available to the computation engine. Therefore, it must contain its own stack.

3.2.2. Identify inputs and outputs

With a clear picture of how the stack machine is supposed to operate, and in what environment it exists, we can now try to classify what inputs and outputs it needs.

Outputs: The output from the computation engine is depicted in Figure 3.1. We require one single-bit `read` signal, and one eight-bit `result` signal that shows the top of the stack.

Inputs: Parts of the input is also shown in the figure; specifically, we require a 16-bit `instruction` signal, and a single-bit `empty` signal. Additionally, we require a `clk` clock signal and a `reset` signal.

[2] For instance `http://www.meta-calculator.com/learning-lab/reverse-polish-notation-calculator.php`.

3.2.3. Sketch RTL with basic blocks

Toplevel RTL Design Sketch

When designing a complex hardware unit, it is a good idea to decompose the design problem into smaller, more manageable parts. Which parts this should be is typically made clear by starting to reason about the requirements of the design at a high level, inserting black box abstractions in the design when necessary.

To begin with, we can make use of our work in step two and draw an RTL sketch containing only the inputs and outputs. With such a figure in place, the rest of the work is filling it with abstractions which implement the functionality required to transform the input into the correct output at the correct time. A sketch for the input and output of the stack computation engine, hereafter named SCE, is given in Figure 3.4.

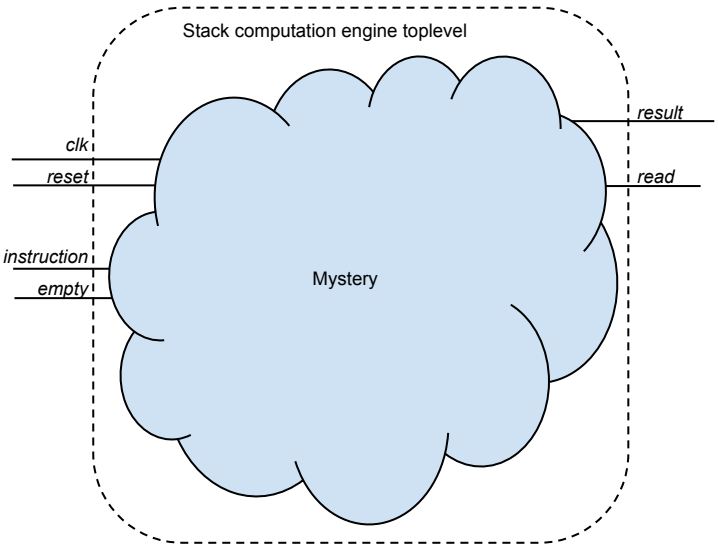


Figure 3.4.: An RTL sketch of the stack computation engine with only inputs and outputs specified.

Step 1: As we noted in subsection 3.2.1, we will be required to implement a stack. We can therefore start the design by including a stack black box abstraction. To do this, we must decide on what inputs and outputs our toplevel will require from the stack. Certainly, it must be possible to push and pop values to the stack. It must also be possible to neither push nor pop when the SCE is idle, so two separate signals are necessary. We also require a separate signal to transmit the value pushed into the stack. We also need a signal to transfer popped data out of the stack. To make matters simple, and remain compatible with the typical stack abstraction, we will only include one signal reporting the current top of the stack.

The resulting design is illustrated in Figure 3.5. How the `push`, `pop` and `stack_in` signals are controlled, remains unspecified. As the top of the stack contains the result of the last computation, `stack_top` can be directly routed to the `result` output line.

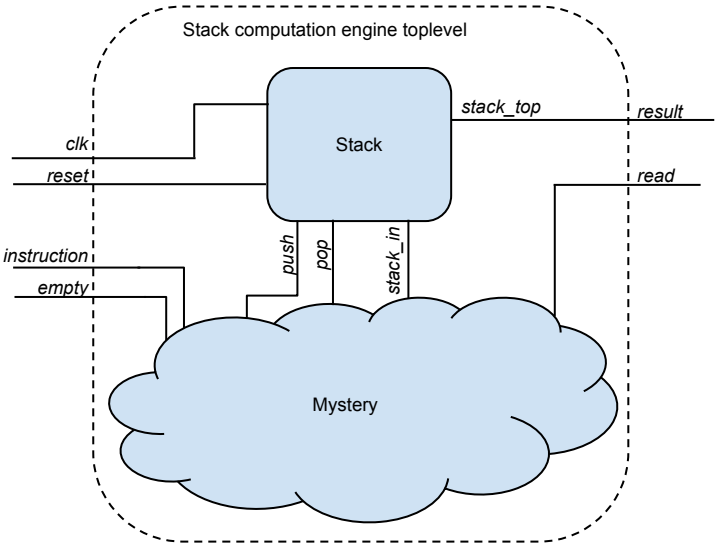


Figure 3.5.: Adding the first abstraction to the stack computation engine RTL sketch.

Step 2: So far, so good. What the mystery cloud still must provide us with is control logic which drives the stack and output control signals, and logic to compute results from `add` and `sub` instructions. Let us first consider what the control might look like. We know that the control of push and pop signals will differ depending on what instruction we are executing. For a push operation, we will only want to push a single operand; for an add, we will want to pop two operands, then push the result. The simplest way to model such series of control events with diverging paths is using a *Finite-State Machine (FSM)*. By designing an FSM which steps through an operation one state at a time, we will only require a single module which drives all control signals. Most hardware contains an FSM as part of the control logic.

The design with such a control module included is depicted in Figure 3.6. Its behaviour will be dependent on the `instruction` and `empty` inputs, as well as internal state. Its outputs will drive all computation control signals, as well as communication with the instruction buffer[3].

Step 3: Now, what is missing is the actual computation which supplies values to the stack. Let us consider what is required to implement this. First, we can note that the result from executing instructions can come from two sources. For push instructions, the result is the operand field in the instruction. Therefore, our computation must get the

[3]The clock and reset signal should also be input to the control module. However, to keep the figure cleaner these signals will not be explicitly propagated further.

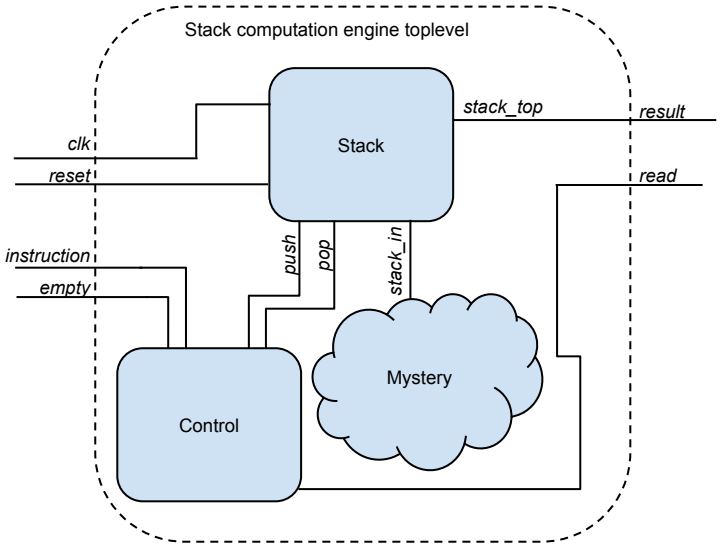


Figure 3.6.: Adding the control abstraction to the RTL sketch.

operand field as an input. For `add` and `sub` instructions, the result is the corresponding computation based on operands stored on the stack. To be able to perform these computations, we must add the `stack_top` as an input to the result computation. This reasoning leads us to the additions in Figure 3.7.

Since the input to the stack has two potential drivers, we must multiplex the `stack_in` signal between the operand signal and an `alu_result` signal. The means of controlling the multiplexer can be provided by the control module, which knows whether we are currently executing a `push` instruction or not. The implementation of this control logic leads us to the RTL sketch in Figure 3.8.

Finally, we must now drive the `alu_result` signal using the operands provided through `stack_top`. Since only one operand can be extracted at a time, we must at least provide a register for the first operand from the stack, since this operand will be gone from the stack when we read the second operand. For simplicity and regularity, we also include a register for the second operand. Finally, since the operands must be popped in different cycles, we need separate write-enable signals for the two registers.

With operand storage in place, all that is missing is an Arithmetic Logic Unit (ALU) for calculating the result, and an ALU operation selection signal. The final SCE toplevel design can be seen in Figure 3.9.

Subcomponent RTL Design Sketch

Having now sketched the RTL for the toplevel. what remains is designing the stack and control modules. This will be parts of the exercises you are asked to solve. However,

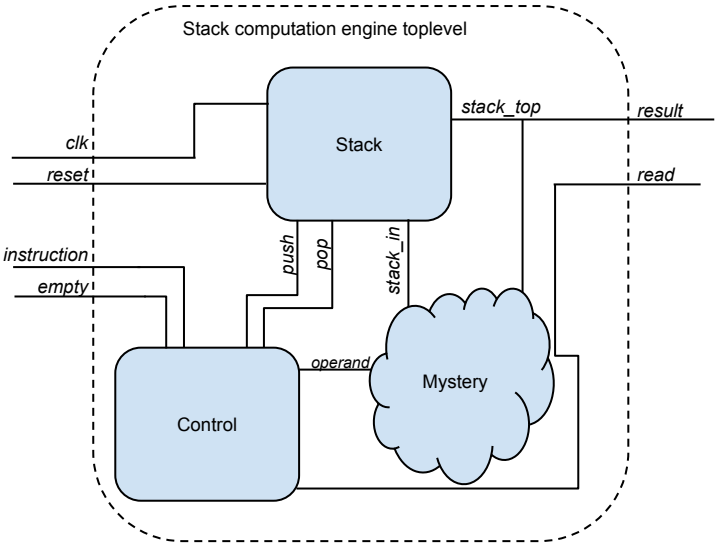


Figure 3.7.: Expanding the control abstraction in the RTL sketch.

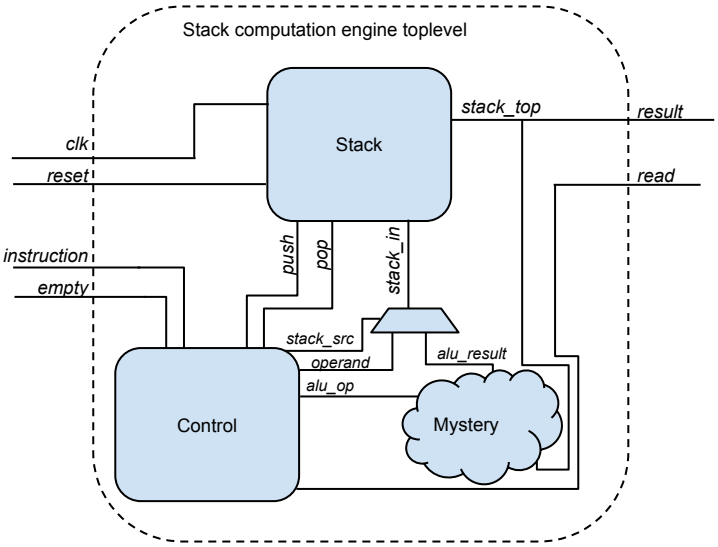


Figure 3.8.: The toplevel is expanded with stack input multiplexing.

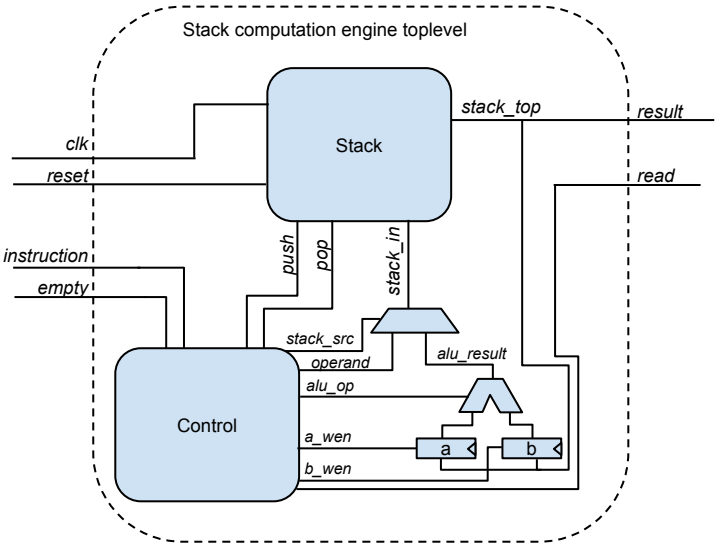


Figure 3.9.: The completed RTL sketch of the stack computation engine.

to make it possible to divide the exercise into smaller pieces we will here provide you with the state machine diagram for the control module. The state machine is a Moore machine, meaning that the outputs are determined by the current state and not the transitions between states. The diagram is presented in Figure 3.10.

3.2.4. VHDL Implementation

The VHDL which implements the RTL sketch of the toplevel is provided in the file `stack_machine.vhd`.

To make the design more flexible, type aliases are used for certain signals such as operands. A package containing these type aliases is provided in `defs.vhd`.

You will be asked to write the VHDL for the stack and control modules in the exercises. As the entity declarations are given by the expectations of the toplevel, the skeleton files `stack.vhd` and `control.vhd` are provided.

3.2.5. Testbench-Driven Simulation

To lessen the amount of work required in the first exercise, test benches are provided in the `tests/` folder in the handouts. The test bench for the stack module is in `stack_tb.vhd`, for the control modules in `control_tb.vhd`, and for the toplevel design in `stack_machine_tb.vhd`. To run a specific test, make sure these files have been added to your Vivado project. Switch to the Project Navigator view, right-click the desired

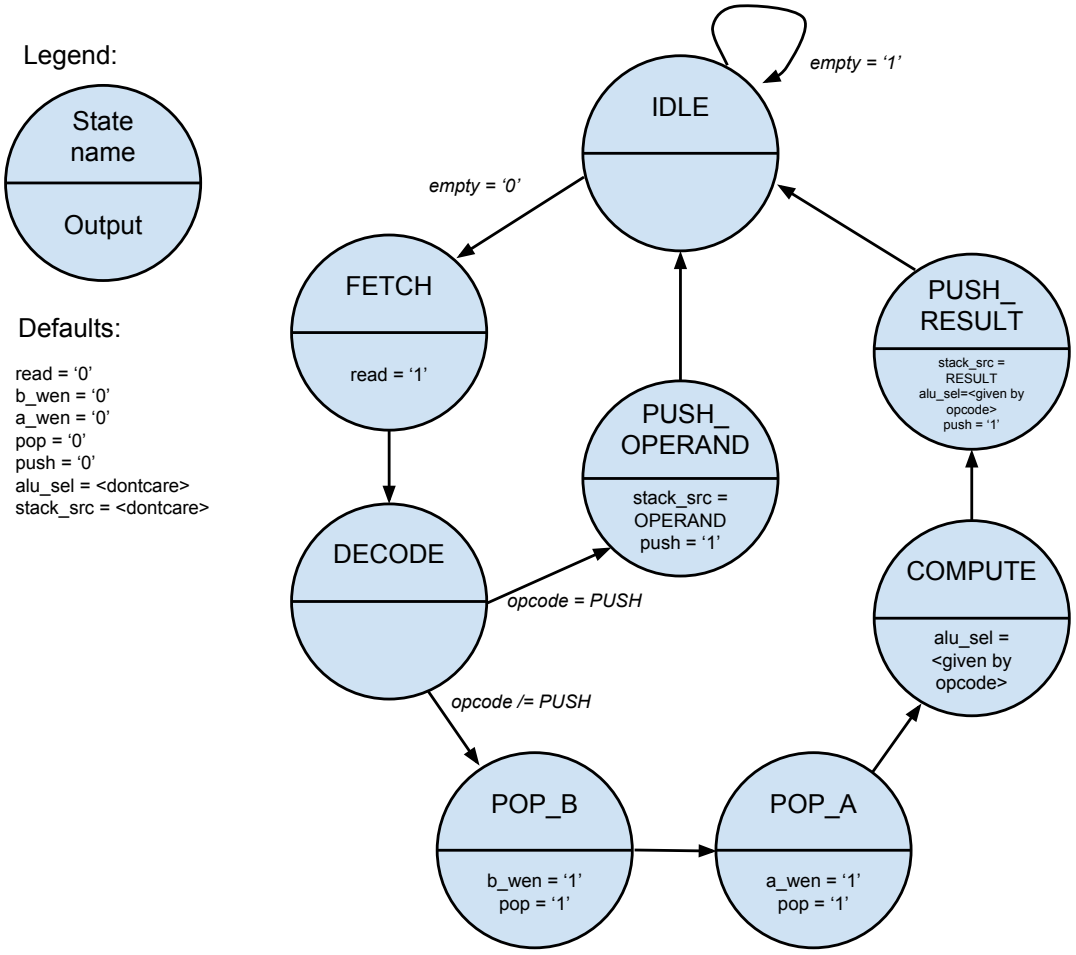


Figure 3.10.: The FSM design for the control module in exercise zero.

test bench in the Sources pane and click "Set as Top", and then run the simulation as described in Section 2.2.3. Note that you may need to also type "run all" in the console window run all the tests in the testbench. If all tests pass, the simulation ends with the message *"Failure: TEST SUCCESS"*.

The test benches contain assertion statements, which means that the simulation is halted when a check fails, and an error message will be printed. This resembles unit tests which you may be familiar with from software development. With the help of the error message, you may be able to realize the cause of the error from a quick source file inspection. Otherwise, you can debug the root cause of the problem by backtracking the signal transitions leading to the error in the waveform window.

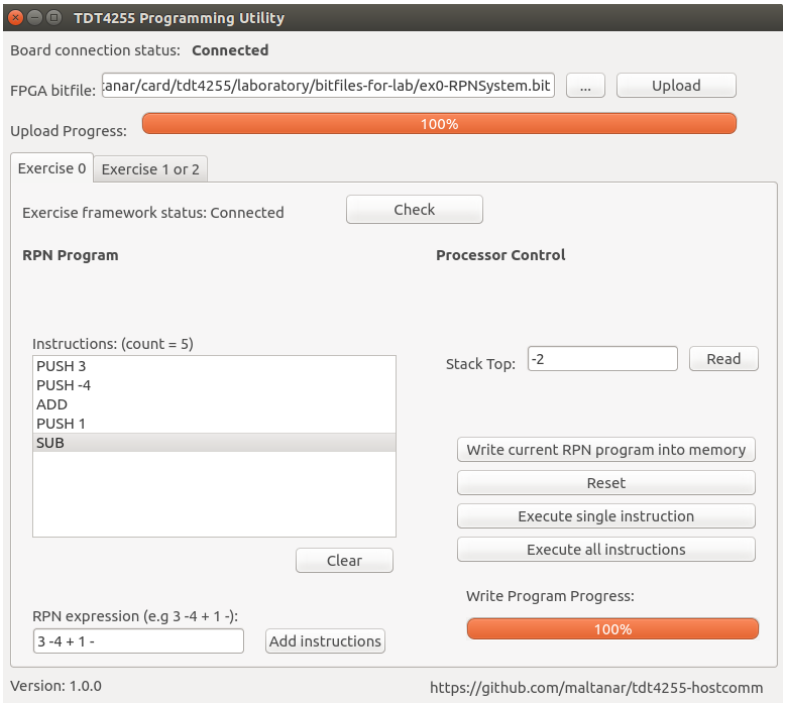


Figure 3.11.: The `hostcomm` utility showing the Exercise 0 control interface.

3.2.6. FPGA Synthesis and Configuration

To run our processor on the FPGA, we need communication with the external world. A framework which handles this is provided in the `framework/` folder. These files must be added to your Vivado project, and the top-level entity set to `Top_cfg` before you attempt to generate the FPGA bitfile. With the framework files properly added, bitfile generation and configuration can be done as was covered in chapter 2. Make sure that you do not ignore warnings from the synthesis tool, as explained in Appendix C.

3.2.7. FPGA Testing

To check that your design works as expected on the real FPGA, you can use the `hostcomm` utility program provided, shown in Figure 3.11. This program accepts RPN expressions containing numbers between -128 and 127, additions using '+', and subtractions using '-'. The expression is transformed into a series of instructions when the "Add instructions" button is pressed. You can transfer these instructions to the instruction buffer with "Write current RPN program into memory" and execute instructions one-by-one or all at once with the appropriate buttons. The "Stack Top" field will display the current value read from the `stack_top`. If your design works as expected, then the expressions

should be properly calculated. Table 3.1 contain some example expressions you may wish to test.

Expression	Expected Result
`1 2 +`	3
`1 2 -`	-1
`127 1 +`	-128
`-128 1 -`	127
`-128 127 + 100 -128 113 30 + - + -`	-86

Table 3.1.: Example RPN expressions for testing the design on the FPGA

3.3. Tasks

This section splits the remaining design and implementation effort into smaller tasks. By solving the tasks one at a time, in the order that they are presented, you will end up with a completed stack machine computation engine implementation.

3.3.1. RTL Design

The tasks in this section are drawing tasks: grab your favourite RTL sketch utility[4], and try to figure out which RTL building blocks you will need (see Appendix B), how they should be connected with each other as well the inputs and the outputs of the module.

Stack Module

When designing the stack, remember that you can expect that it will never overflow nor underflow, and that only a single operation is permitted at a time.

1. Select an RTL basic block which can provide the memory for the stack.
2. Sketch the RTL which selects the top-of-stack element stored in this memory.
3. Extend the sketch with RTL blocks which ensure that zero is output when there are no elements on the stack.
4. Extend the sketch with RTL blocks which enable pushing an element to the stack.
5. Extend the sketch with RTL blocks which enable popping an element from the stack.
6. Consider where the reset signal and clock must be connected. You do not have to include the wiring in your sketch.

[4]Our personal favourite is pen and paper

Control Logic Module

The control logic module should be structured as an FSM. Before solving these exercises, you should read about the FSM basic block in B.

For task 2 and 3, you may sketch a subset of the output and control sufficiently large to make you feel comfortable that you have an idea what the design you will be implementing might look like.

1. What state is required in this FSM?
2. Sketch the RTL required to implement state transitions.
   a) Generate the next state when in the IDLE state.
   b) Generate the next state when in the DECODE state.
   c) Generate the next state based on the current state.
   d) Update the state to the next state.
3. Sketch the RTL required to control output signals
   b) Control read.
   c) Control push.
   d) Control pop.
   e) Control stack source selection.
   f) Control operand write enable signals.
   g) Control ALU function selection.

3.3.2. VHDL Implementation

Before starting the exercises, you should familiarize yourself with the provided VHDL source files. Look through the `stack_machine.vhd` toplevel implementation, compare it to the RTL sketch in Figure 3.9, and try to understand which parts of the VHDL implements which parts of the RTL design. Then, look briefly over the contents of `defs.vhd` to see how the application-specific types are defined. You can find this file in Vivado by clicking on the "Libraries" tab in the "Sources" pane, and expanding the `Design Sources/VHDL/xil_defaultlib` directory. Finally, browse through the skeleton files `stack.vhd` and `control.vhd`.

When solving these exercises, you should keep running the provided testbenches. For each subtask you complete, if completed in the order they are given, one more test should pass in the relevant testbench.

When the testbenches for both the stack and the control module pass, run the provided testbench for the stack machine toplevel. If this test does not pass, it is likely that some expectations have been violated. If the test does pass, then the design, if synthesizable, should also work when configured on the FPGA.

Stack Module

1. Write the VHDL code to drive `stack_top` to zero after reset. Test 1 and 2 should now pass.
2. Write the VHDL code which implements your storage basic block.
3. Extend your code to support pushing a single value. Test 3 and 4 should now pass.
4. Extend your code to support popping the single value, leaving the stack empty. Test 5 should now pass.
5. Extend your code to support pushing one value each successive cycle. Tests up to 9 should now pass.
6. Extend your code to support popping one value each successive cycle. All the tests should now pass.

Control Module

In the following tasks, proper behaviour means behaviour in compliance with the state machine diagram in Figure 3.10.

1. Implement proper behaviour for the state machine after reset, with the empty signal set high, by setting state to IDLE and driving output accordingly. Test 1 should now pass.
2. Implement the transition to FETCH state from idle when empty goes low.
3. Drive output properly from the FETCH state. Test 2 should now pass.
4. Implement the transition to DECODE from FETCH state. Test 3 should now pass.
5. Implement the transition from DECODE to PUSH_OPERAND. Drive output accordingly.
6. Drive the `operand` output using the `instruction` input. Test 4 should now pass.
7. Implement the transition to IDLE from PUSH_OPERAND. Tests up to 11 should now pass.
8. Implement the transition to POP_B from DECODE, as well as POP_B output. Tests up to 12 should now pass.

9. Implement the transition to POP_A from POP_B, as well as POP_A output. Test 13 should now pass.
10. Implement the transition to COMPUTE from POP_A. Make the ALU perform an ADD. Test 14 should now pass.
11. Implement the transition to PUSH_RESULT from COMPUTE, with appropriate output. Test 15 should now pass.
12. Implement the transition to IDLE from PUSH_RESULT. Tests up to 20 should now pass.
13. Implement proper ALU functionality selection. All the tests should now pass.

3.3.3. Deliverables for Exercise 0

For this exercise you do not have to write a report, but you should still deliver the following:

- Your RTL sketch of the stack and control modules.
- Your stack and control module VHDL files.
- Text files with the output you get when running the test benches.
- Feedback on the tutorials and exercise 0.

4. Exercise 1 – A Pipelined Processor

TBA.

4.1. Introduction

4.2. Suggested Architecture

4.3. Requirements

5. Exercise 2 – A Dual-Issue In-Order Pipelined Processor

TBA.

5.1. Introduction

5.2. Suggested Architecture

5.3. Requirements

A. VHDL Cheatsheet

VHDL is an extensive language, and lots of documentation and tutorials on it can be found online. However, the more advanced language constructs are usually not necessary to build the hardware you have in mind. Thus, we provide a "cheatsheet" on the next page to help you get up to speed with writing VHDL code for the course.

If you need more comprehensive information on VHDL, we recommend the "VHDL Cookbook"[1].

[1] http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

VHDL Cheat-Sheet

Concurrent Statements

Concurrent Signal Assignment
(dataflow model)

```
target <= expression;
```

```
A <= B AND C;
DAT <= (D AND E) OR (F AND G);
```

Conditional Signal Assignment
(dataflow model)

```
target <= expressn when condition else
          expressn when condition else
          expressn;
```

```
F3 <= '1' when (L='0' AND M='0')  else
      '1' when (L='1' AND M='1')  else
      '0';
```

Selective Signal Assignment
(dataflow model)

```
with chooser_expression select
   target <= expression when choices,
             expression when choices;
```

```
with SEL select
MX_OUT <= D3  when "11",
          D2  when "10",
          D1  when "01",
          D0  when "00",
          '0' when others;
```

Process
(behavioral model)

```
opt_label: process(sensitivity_list)
begin
   {sequential_statements}
end process opt_label;
```

```
proc1: process(A,B,C)
begin
   if (A = '1' and B = '0') then
      F_OUT <= '1';
   elsif (B = '1' and C = '1') then
      F_OUT <= '1';
   else
      F_OUT <= '0';
   end if;
end process proc1;
```

⇔

Sequential Statements

Signal Assignment

```
target <= expression;
```

```
A <= B AND C;
DAT <= (D AND E) OR (F AND G);
```

if statements

```
if (condition) then
   { sequence of statements }
elsif (condition) then
   { sequence of statements }
else --(the else is optional)
   { sequence of statements }
end if;
```

```
if    (SEL = "111") then F_CTRL <= D(7);
elsif (SEL = "110") then F_CTRL <= D(6);
elsif (SEL = "101") then F_CTRL <= D(1);
elsif (SEL = "000") then F_CTRL <= D(0);
else  F_CTRL <= '0';
end if;
```

case statements

```
case (expression) is
   when choices =>
       {sequential statements}
   when choices =>
       {sequential statements}
   when others => -- (optional)
       {sequential statements}
end case;
```

```
case ABC is
    when "100" =>  F_OUT <= '1';
    when "011" =>  F_OUT <= '1';
    when "111" =>  F_OUT <= '1';
    when others => F_OUT <= '0';
end case;
```

B. RTL Building Blocks

We remind you that logic design and HDLs are large and complicated topics, and this compendium barely scratches the surface. There are many excellent resources on the web or in the library, if you don't find the answers to your questions in the compendium or if you would like to learn more.

In this section, we will describe a number of typical building blocks that you can use to sketch RTL designs. We also provide code examples (where appropriate) to demonstrate how that particular building block could be instantiated in VHDL[1]. Most of these blocks are rather simple and one would not dedicate a separate VHDL file for such small blocks; you can just add the corresponding code into a bigger module. Keep in mind that the blocks listed here are not set in stone; indeed, the blocks discussed towards the end are actually composed of the smaller blocks first discussed. You can add more functionality into a block or adjust it to fit your design as needed.

B.1. Logic Data Types and Operators

The most basic data type in VHDL is the so-called standard logic type, `std_logic`. A signal of this type can be thought to contain a single binary or logic value. For convenience, multiple-bit data is assembled into logic vectors, `std_logic_vector`. Single logic values are written inside single quotes, like `'1'` and `'0'`. Multi-bit logic vectors are written inside double quotes, like `"11011110101011011011111011101111"` for a 32-bit vector. Constants may also be expressed in hexadecimal (`x"DEADBEEF"`) or octal (`O"33653337357"`).

VHDL contains a variety of logic operators for operating on these types, which are summarized in Table B.1. The `std_logic` types and operators form the heart of combinational logic.

B.2. Arithmetic Data Types Operators

Although everything is treated as logic ones and zeroes in hardware, HDLs offer us a higher level of abstraction. As was covered in Section 2.3.2 of the tutorial, the

[1]Note that there are many different ways to instantiate components in VHDL, the way shown here is not the only way.

Table B.1.: Logical operators in VHDL.

Operator	Operation
not	Logical inverse
and	Logical AND
or	Logical OR
xor	Logical XOR
nor	Logical NOR
nand	Logical NAND
=	Equality test
/=	Inequality test
&	Vector concatenation
()	Vector element
(x downto y)	Vector element range

Table B.2.: Arithmetic operators in VHDL.

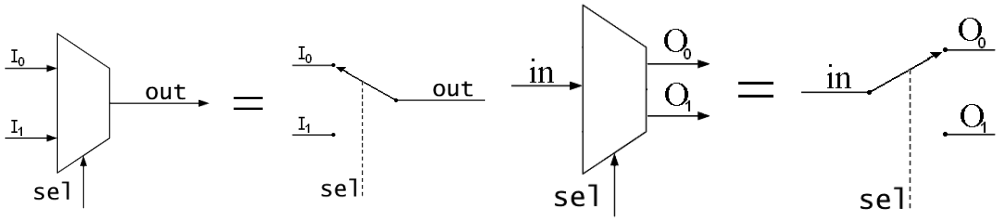
Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
=	Equality test
/=	Inequality test
<, >	Less than, greater than
<=, >=	Less than or equal, greater than or equal
unsigned(*std_logic_vector*)	convert std_logic_vector to unsigned
std_logic_vector(*unsigned*)	convert unsigned to std_logic_vector

`ieee.numeric_std` package contains the definitions for the data types `signed` and `unsigned`, as well as the operators that operate on them. Remember that the bit widths of these types are customizable; you do not have to use 32-bit numbers if you only need 24 bits.

Table B.2 summarizes some of the VHDL arithmetic operators. The synthesis toolchain will normally recognize these and generate appropriate hardware for performing the operations in a fast and resource-efficient way. Thus, you do not have to construct e.g a hardware adder in pure logic from scratch. If you need more operations, you may want to look at the contents `numeric_std.vhd`.

B.3. Multiplexers and Demultiplexers

A multiplexer (or mux) uses a *select signal* to choose one signal from a group of input signals and forward it to its single output. Conversely, a demultiplexer (or demux) takes a single input and uses its select signal to forward it to one of its multiple outputs. Both components are illustrated in Figure B.1, and are very common in combinatorial circuits.



(a) 2-to-1 multiplexer (b) 1-to-2 demultiplexer

Figure B.1.: Multiplexer and demultiplexer.

The easiest way if describing a multiplexer in VHDL is by using a concurrent assignment statement directly inside the architecture body, in the following manner:

```
MuxOutput <= MuxInput0 when MuxSel='0' else MuxInput1;
```

Constructing demultiplexers is slightly trickier, as the output signals which are not selected have to be assigned some default value. This could be done by n concurrent assignment statements for n signals, but an easier way to do this is to use sequential statements[2] inside a VHDL process:

Listing B.1: Demultiplexer example.

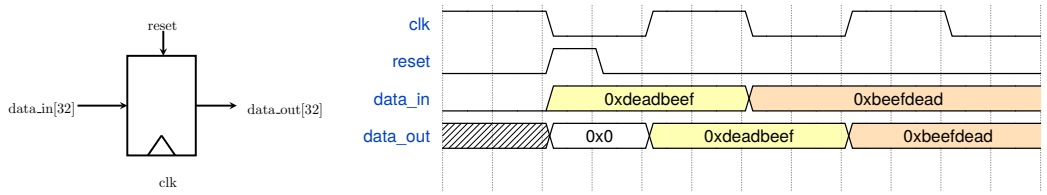
```
1  -- both input and sel are 'read' inside process,
2  -- remember to include them in sensitivity list
3  Demux: process(DemuxInput, DemuxSel)
4  begin
5      -- assign default values to demux outputs
6      DemuxOutput0 <= (others => '0');
7      DemuxOutput1 <= (others => '0');
8      -- now describe the output selection behavior
9      -- sequential statements inside process:
10     -- later statements override earlier ones
11     if DemuxSel = '0' then
12         DemuxOutput0 <= DemuxInput;
13     elsif DemuxSel = '1' then
14         DemixOutput1 <= DemuxInput;
15     end if;
```

[2]Note that sequential statements are unrelated to sequential logic. The first is a way of writing code in VHDL, whereas the second is a type of circuit with memory.

```
16 end process;
```

B.4. Registers and Counters

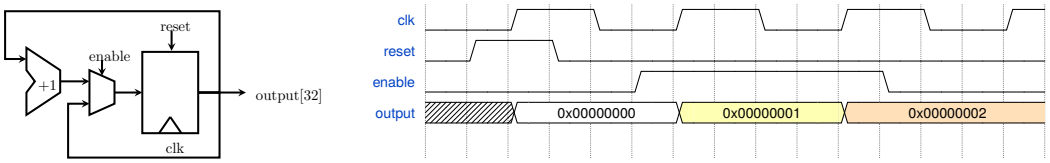
Registers store data and are the key to making designs that have the concept of *state*. Thus, they form the backbone of sequential circuits. The very basic building block for data storage is a *flip-flop*, which can store either a 1 or 0, but in this course we can work with a higher level of abstraction. A group of flip-flops storing related data is termed a register. For our level of abstraction, registers typically are defined by their width (how many bits of information they can store) and reset behavior (whether reset is synchronized to clock edges and which value is stored upon reset). An example of a 32-bit register is displayed in Figure B.2. Various implementations of registers exist, which may include inputs like enable/set all to 1/set all to 0 for extra functionality. You do not have to manually implement this while writing your HDL code, as the synthesis tool will map your design to the most appropriate register types.



(a) RTL Sketch. (b) Timing diagram.

Figure B.2.: A positive edge triggered 32-bit register with asynchronous reset.

Counters can be built from registers and some combinational logic. The output of the register is passed through an arithmetic operator (typically an adder), which is then fed back into the register input if the counter enable signal is high (or implemented with a register with an enable input). An example of such a counter is illustrated in Figure B.3. Overflow output from the counter register is also common, which is useful for creating infrequent events from frequent events.



(a) RTL Sketch. (b) Timing diagram.

Figure B.3.: A 32-bit up-counter with synchronous reset.

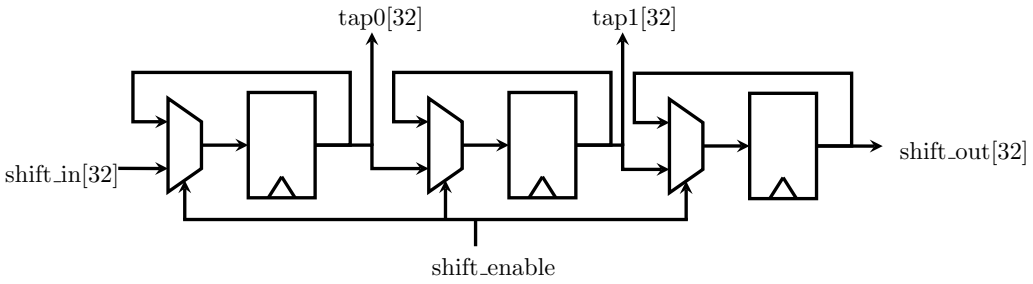


Figure B.4.: A 32-bit wide 3-stage shift register with two taps.

VHDL will infer registers for every signal which is assigned a value only on a clock edge. Basic register and counter synthesis in VHDL is covered in Sections 2.3.2 and 2.3.3 and will not be repeated here.

B.5. Shift Registers

A *shift register* is a cascade of same-width registers whose inputs and outputs are connected together. This structure performs a data shift operation, and can be thought of as the hardware equivalent of a queue. Each stage may contain an output (or "tap") which can be used to retrieve its contents. Although most frequently used to implement serial-to-parallel data stream conversion, they can be used to implement FIFO queues or stacks. Figure B.4 illustrates a shift register, and the corresponding VHDL implementation is described in Listing B.2. Note that several variants of shift registers can be constructed using different stage widths. A common variety is a shift register with stage width 1, which shifts single bits.

Listing B.2: Shift register example.

```
1  architecture Behavioral of ShiftReg is
2      signal stage0, stage1, stage2 : std_logic_vector(31 downto 0);
3  begin
4      -- implement the shifting logic between stages
5      ShiftProcess: process(clk)
6      begin
7          if rising_edge(clk) then
8              if shift_enable='1' then
9                  stage0 <= shift_in;
10                 stage1 <= stage0;
11                 stage2 <= stage1;
12             end if;
13         end if;
14     end process;
15     -- expose the output of the last stage
16     shift_out <= stage2;
```

```
17      -- expose intermediate stage values as taps
18      tap0 <= stage0;
19      tap1 <= stage1;
20  end Behavioral;
```

B.6. Random Access Memories and Register Files

In programmable processors, the *addressable memory* abstraction is frequently used. In hardware, this is provided by a form of random access memory (RAM). There are different types of hardware RAM implementations, although the interface used to access the memory remains the same. Typically, there is an address input to select a particular location in the memory, a data output to retrieve data from the selected address, a write enable signal to distinguish between read and write operations, and a data input for supplying data to be written. These signals are typically grouped into a *port*. A RAM has one or more ports that can read/write data in parallel[3] RAM is also characterized by width (the number of bits read/written in each operation) and depth (the number of addresses available). The RTL sketch for a register file is shown in Figure B.5 and the VHDL implementation in Listing B.3.

The implementation differences give rise to important distinctions in terms of use cases for different memory types. For instance, in FPGAs, we distinguish *register files* and *block RAM* by availability of reset value: each location in a register file is initialized to a known value upon reset in a register file, whereas block RAM may not have individual value initializations or even a concept of reset. Therefore, by removing the reset logic in Listing B.3 one would get the VHDL code template for creating a block RAM. Block RAM can work faster with larger sizes and consume less resources, since it is mapped to special hardware elements inside the FPGA (see Section B.8).

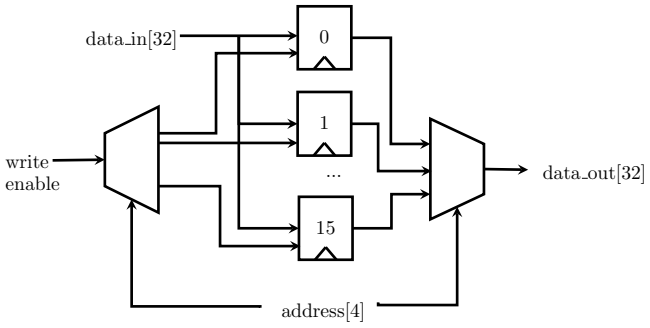


Figure B.5.: A 32-bit wide 16-bit deep single-ported register file.

[3]When using two ports, avoid doing read+write or write+write to the same address, this may result in undefined behavior.

Listing B.3: Register file example.

```
architecture Behavioral of RegFile is
    -- define a type for the register file:
    -- array of std_logic_vectors (=2D array of bits)
    type RegisterFileType is array(0 to 15) of std_logic_vector(31 downto 0);
    signal regFile : RegisterFileType;
begin
    -- register file behavior
    process (clk, reset) is
        begin
            if reset = '1' then
                -- use nested 'others' clause to specify reset value
                -- (zeroes) for regfile
                regFile <= (others => (others => '0'));
            elsif rising_edge(clk) then
                -- data write
                if write_enable = '1' then
                    regFile(to_integer(unsigned(address))) <= data_in;
                end if;
            end if;
    end process;

    -- data read. Not dependent on clock and reset, which makes
    -- the read combinational. If this statement is placed alongside
    -- the data write, the effect is an additional register after
    -- data_out in the figure.
    data_out <= regFile(to_integer(unsigned(address)));

end Behavioral;
```

B.7. Finite State Machines

Finite State Machines (FSMs) describe machines which can be in one of a predefined number of states. The transition between states depend on input and the state itself. Outputs can either be dependent only on the current state, in which case the machine is called a Moore machine, or it can be dependent on both the current state and the input, in which case the machine is called a Mealy machine. Mealy machines may implement the same functionality using a smaller number of states than a Moore machine, but are usually more complex.

Modelling a problem as an FSM decomposes the problem into several distinct stages or phases, one for each state introduced, which may be solved independently. Having designed an FSM solution, it is then possible to employ a generic RTL template to implement the machine. This greatly simplifies the implementation effort, as it turns into textually describing the FSM design using the appropriate HDL syntax.

The generic FSM template is illustrated in Figure B.6. The FSM is separated into a sequential part, which contains all the state elements, and a combinational part, which computes the output and the next state based on the input and the current state.

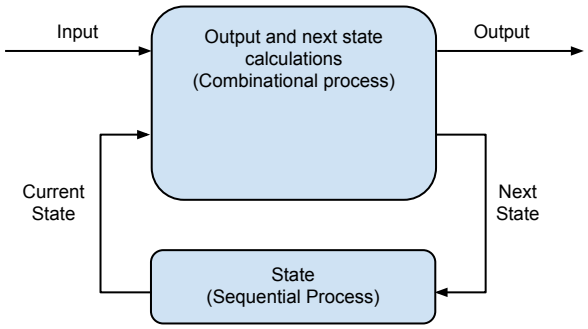


Figure B.6.: The generic structure of the FSM template.

The VHDL template for the same structure is given in Listing B.4. Similarly to Figure B.6, the VHDL code is split in a sequential part and a combinational part. The sequential part handles writes to the state registers. The bulk of the logic is handled in the combinational part, dealing with the computation of the next state and output. Although this split might seem too simple to be useful, it provides a clear separation of concerns. Following this pattern also makes it easier for other developers to recognize what your code is doing. The synthesis tool is also more likely to recognize the code as an FSM, enabling it to run its FSM optimization algorithms.

Listing B.4: Finite state machine template

```
architecture Behavioral of FSM is
  type (STATE_1, STATE_2, ..., STATE_N) is state_t;
  signal state, next_state: state_t;
begin
  -- Combinational part, computing the next state and outputs.
  with state select
  next_state <=
    -- assume STATE_1 always leads to STATE_2
    STATE_2 when STATE_1,
    -- The function invocations represent the other state transitions
    state_2_transition(input) when STATE_2,
    ...
    state_n_transition(input) when STATE_N;

  -- If the FSM is a Moore machine, the output is determined solely by
  -- the current state. This is simpler to reason about, and should be
  -- the design to start with, but it may introduce more delays in the
  -- system since the FSM reacts to inputs the cycle after they first
  -- appear.
  process (state) is
  begin
```

```
22      -- Always remember to set default values for all signals to
23      -- avoid undesired register/latch creation or undriven signals.
24      output_1 <= default_output_1_value;
25      ...
26      output_n <= default_output_n_value;
27
28      -- Drive non-default output as necessary
29      case state is
30        when STATE_1 =>
31          output_1 <= state_1_output_1_value;
32          ...;
33        ...
34        when STATE_N =>
35          ...;
36      end case;
37    end process;
38
39    -- If the FSM is a Mealy machine, the output is driven based on
40    -- transitions. next_state computations and output computations are
41    -- therefore likely to overlap, so using next_state to select output
42    -- may be more efficient.
43
44    -- process (next_state, input) is
45    -- begin
46    --   output_1 <= default_output_1_value;
47    --   ...
48    --   output_n <= default_output_n_value;
49    --
50    --  -- Drive non-default output as necessary
51    --   case next_state is
52    --     when STATE_1 =>
53    --       output_1 <= transition_to_state1_output1_value;
54    --     ...
55    --     when STATE_N =>
56    --       ...;
57    --   end case;
58    -- end process;
59
60
61    -- Sequential state updates all occur in this process
62    process (clk, reset) is
63    begin
64        if reset = '1' then
65          state <= STATE_1;
66        elsif rising_edge(clk) then
67          state <= next_state;
68        end if;
69    end process;
70  end Behavioral;
```

Note that there are two ways to drive the output in the code template: one for Mealy machines, which is commented out, and one for Moore machines. Note that the next state transition is written using combinational VHDL, and the output is written using

sequential VHDL. Using combinational VHDL, it is arguably clearer exactly how each signal is computed. It is also simpler to ensure that all signals are driven, and that no latches are accidentally inferred. However, sequential VHDL may be more concise, and how the values of signals relate to each other may be clearer since they can be driven at the same location in the code. Which syntax is the best to employ is dependent on the FSM in question, and it is advisable to be familiar with the different styles.

B.8. FPGA-specific IP Blocks

We advise against using FPGA-specific IP block components as part of your processor design for TDT4255 since the development of your own HDL skills is a learning goal for the course. However, IP blocks are an important part of many FPGA designs today (including the exercise infrastructure for TDT4255), hence we discuss them briefly here.

Similar to how pre-compiled libraries are used for software development, the concept of *Intellectual Property (IP) Blocks* is used in hardware development for using pre-made components. Modern FPGA chips contain so-called *hardened IP blocks* (or *hard macros*) in addition to the regular configurable logic. These blocks are bits of hardware that are not reconfigurable, but fulfill predetermined functions with great efficiency. The particular types of hard macros available depend on the particular FPGA, but usually include DSP blocks for performing multiply-add operations, block RAM (BRAM) for instantiating random-access memories and various other blocks that communicate with external high-performance interfaces (like PCI Express or DRAM).

There are two ways of using these components. The first (and preferred) method is simply modeling the desired component in VHDL; FPGA synthesis tools can often recognize HDL that can be mapped to hard macros. The other method is to use vendor-specific tools (such as Xilinx Core Generator), which is then instantiated in HDL as a black box. This black box will be mapped to the appropriate hard macros by the synthesis tool.

C. Antipatterns in VHDL

This section will describe some common antipatterns of VHDL code patterns which might seem reasonable during development, which the synthesis tool reasonably transforms into compliant hardware, which ends up having unreasonable behaviour in the final implementation. As a general precaution to avoid such issues, make sure that you consider all warnings issued by Vivado carefully. Whenever you see a warning message from Vivado, especially after Synthesis as in Figure C.1, do your best to fix it. Although the message might look benign, ignoring warnings can lead to wasted hours of excruciating debugging!

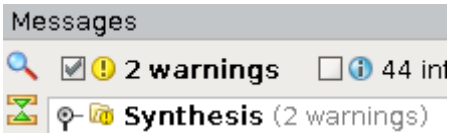


Figure C.1.: Warnings during synthesis is an indication that something may be wrong with your design. Warnings during "Implement Design"-stages are often not as dangerous, but should also be considered.

C.1. Latches

Of all the many different warnings Vivado can issue, the warning message about latches exemplified in figure C.2 is perhaps the most infamous. Latches are level-triggered storage elements, which stores the input value when a control signal is logically enabled. In contrast, the standard flip-flop storage element described in Section B.4 is edge-triggered: only when the control signal (the clock) makes a transition from 0 to 1 is the input value stored. Latches can introduce timing problems which lead to designs which work in simulation, but fail inexplicably when uploaded to an FPGA. As this is obviously not appealing, it is a good idea to avoid using latches whenever you can.

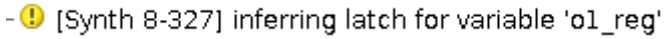


Figure C.2.: The message Vivado issues when inferring a latch. Ignore this warning at your own peril!

The next section will explain in what situations latches will be inferred, and how to avoid these situations. For those interested, Section C.1.2 will proceed to explain *how* the timing issues the warning message describes can arise.

C.1.1. Latch-Inferring Code Patterns

Latches are storage elements. They are accidentally included whenever VHDL code is written which requires the circuit to remember a previous value for a signal outside of an `if rising_edge(clk)`-region.

Listing C.1 shows a VHDL module containing several example snippets which lead to inferred latches. In order of presentation, the problems are the following:

1. An incomplete if-statement. Signal `o1` must retain its old value when `i1 = '0'`, which necessitates a latch.
2. An incomplete sensitivity list. Since the process is only supposed to run when a signal on the sensitivity list changes, a memory element is required so that changes to `i2` do not otherwise propagate through. Vivado interestingly ignores the sensitivity list for synthesis, but other synthesis tools may infer a latch.
3. An incomplete conditional signal assignment. As with the incomplete if-statement, signal `o3` needs to remember its value when `i3` is neither `"01"` nor `"10"`.
4. An incomplete case statement. Note that the incompleteness arises from a lack of output driver specifications, not a lack of `when others`.

Listing C.1: VHDL snippets which may infer latches.

```
1  entity latch_examples is
2    port (
3       signal i1, i2 : in std_logic
4     ; signal i3 : in std_logic_vector(1 downto 0)
5      ; signal o1, o2, o3, o4, o5 : out std_logic
6    );
7
8  end latch_examples;
9
10 architecture Behavioral of latch_examples is
11 begin
12
13 -- Example 1): incomplete if-statement
14 process (i1, i2) is
15 begin
16     if i1 = '1' then
17         o1 <= i2;
18     end if; -- what if i1 = '0'? Must keep old value of o1
19 end process;
20
```

```
21 -- Example 2): incomplete sensitivity list
22 process (i1) is -- no change in o2 unless i1 changes!
23 begin           -- must store o2 until i1 is changed.
24     o2 <= i1 and i2;
25 end process;
26
27 -- Example 3): incomplete conditional signal assignment
28 o3 <= i1 when i3 = "01" else
29       i2 when i3 = "10";
30
31 -- Example 4): incomplete case-statement
32 process (i3) is
33 begin
34   case i3 is
35     when "00" =>
36        o4 <= '0';
37      when "11" =>
38        o5 <= '1';
39     when others =>  -- latch even with default case,
40        o4 <= '1';   -- since o4 is not set for case "11"
41        o5 <= '0';   -- and o5 is not set for case "00"
42   end case;
43 end process;
44
45 end Behavioral;
```

How to Avoid Latches The solution to the problem is generally simple. If you actually want to create a memory element, which changes value only on certain conditions and otherwise keeps its previous value, use a flip-flop instead of a latch by assign the state within an `if rising_edge(clk)`-region. Otherwise, make sure that you specify a driver for a signal for all possible inputs. For combinational VHDL (outside of processes, example 3 above), make sure you always include a final `else` or `when others`. For sequential VHDL (inside processes, example 1 and 4 above), assign default values to the relevant output signals in the beginning of the process. See the FSM-example in Section B.7 for an example of how to do this.

C.1.2. Why Latches Are Bad

Let us first start by understanding why behaviour dependent on timing can lead to seemingly non-deterministic behaviour. Timing in this context refers to the propagation times of signals, and at what time exactly value updates reach different points in the circuit. When behaviour is dependent on timing, the system will only have deterministic behaviour for a fixed timing characteristic. Timing characteristics, however, are volatile beings. If you expand or shrink your logic, your circuit may be routed differently on the FPGA. This can alter timing behaviour in parts of the design logically unrelated to the change. Timing characteristics can also change dynamically, since electrical properties

such as resistance is dependent on temperature. As a chip heats up, its behaviour can therefore change in a timing-dependent design.

Having understood why timing-dependent behaviour is scary, let us examine how an unintended latch can lead to timing dependence. The beauty of synchronous designs, where state updates are guarded by a global clock signal, is that the only aspect of timing which matters is for signals to reach a steady state in time for the final signal values to reach their destination registers. What values the signals assume within a clock cycle is largely irrelevant. A latch, on the other hand, is a state element which is active whenever the control signal is logically high. This means that if the control signal is set high, even in the middle of a clock cycle while signals are stabilizing, the state will be updated. Since the activity of signals in-between clock edges is dependent on timing, latches may also end up being dependent on timing.

As an example, consider the design in Listing C.2. The module propagates its `c` input to its `x` output if both its `a` and `b` inputs are true. Since the VHDL code uses an incomplete if-statement, the `x` output will be created using a latch.

Listing C.2: A module with latched output.

```
1
2  entity latch_module is
3      Port ( a : in  STD_LOGIC;
4             b : in  STD_LOGIC;
5             c : in  STD_LOGIC;
6             x : out  STD_LOGIC);
7  end latch_module;
8
9  architecture Behavioral of latch_module is
10   signal e : std_logic;
11 begin
12
13 e <= (a and b);
14 process (e, c) is
15 begin
16     if e = '1' then
17         x <= c;
18     end if;
19 end process;
20
21 end Behavioral;
```

When testing the module in simulation, things may appear to be working fine as figure C.3 attempts to demonstrate. Changing `a` to 0 and `b` to 1 simultaneously will not lead to a situation where both are true simultaneously, and so `x` is not updated with the value of `c`.

However, as the module output is latched its behaviour is susceptible to timing variations in the arrival of input signals `a`, `b`, and `c`. As an example, assume that input `a` is the

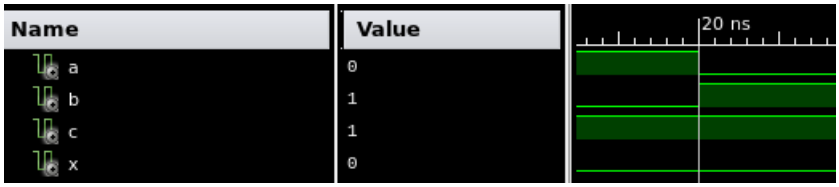


Figure C.3.: When simulating the latching module, everything appears to work. The changes of `b` and `a` apparently happen simultaneously, so that they are never both true. Accordingly, `x` does not take the value of `c`.

result of a combinational expression, whereas input `b` comes directly from a register. The effect would be that input `b` arrives slightly ahead of input a, which we can model using the top-level module in Listing C.3.

Listing C.3: An instantiation of the module with a latch, where one of the input signal arrives slightly later than the others.

```
1
2  entity latches is
3      port (
4          a : in std_logic
5        ; b : in std_logic
6        ; c : in std_logic
7        ; x : out std_logic
8      );
9
10 end latches;
11
12 architecture Behavioral of latches is
13     signal a_prop : std_logic;
14 begin
15
16 a_prop <= a after 1 ns;
17
18 lm: entity work.latch_module
19   port map (
20       a => a_prop
21     , b => b
22     , c => c
23     , x => x
24   );
25
26 end Behavioral;
```

This timing artefact could lead to the situation depicted in Figure C.4. The inputs which modify signal `a` and `b` change simultaneously: `b` goes to 1, and `a` goes to 0. However, since the `a` signal is delayed, there is a short interval in which both `a` and `b` are both 1. This glitch in the latch enable signal, represented by signal `e <= (a and b)` in the

figure, causes a capture of the `c` value only because of the timing characteristics of the input signals. Note that if `b` was slightly delayed relative to `a`, the output `x` would not be altered.

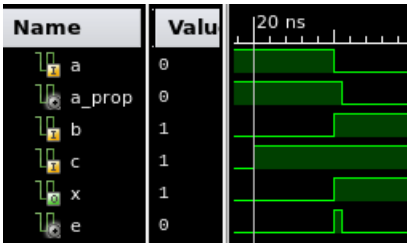


Figure C.4.: Glitching behaviour causes unintended capture of the `c` signal: note how `x` changes from 0 to 1.

Although reacting to glitches in signals is not obviously erroneous in this example, in most synchronous designs it is desirable to let the hardware state only reflect the steady-state behaviour of the circuit. You can imagine the horror of an FSM where latched control outputs suddenly change because inputs had unstable intermediate values. Oops!

In concluding this section, it is worth mentioning that legitimate use cases for latches exist when making high-speed ASICs since latches are faster than flip-flops. However, for an FPGA implementation these benefits do not exist. As a case in point, in its chapter on latches the Vivado Synthesis user guide begins by stating that "Inferred Latches are often the result of HDL coding mistakes", a testament to the low expected utility value of latches in FPGA implementations. Consequently, steering clear whenever possible is the way to go.

C.2. Combinational Loops

A combinational loop is a piece of combinational circuitry where one of the outputs is also used as an input. Put differently, it is any loop in an RTL sketch which does not include a memory element like a register. The problem with combinational loops is that outputs influence the inputs, which of course influence the outputs. For certain inputs, it is therefore possible to get an unstable system where changes to the input changes the output, which changes the input, thus changing the output, and so on indefinitely. Therefore, Vivado warns you whenever it detects a combinational loop with a warning message like the one in Figure C.5. The reason such a construct is not an outright error is that oscillating behaviour is not guaranteed. Depending on the toggling possibilities of inputs and outputs, it may be impossible for oscillations to arise. Even if this is the case, combinational loops limit the timing analysis capabilities of the synthesis tool, which has to make a conservative estimate. This can lead to slower circuits than necessary.

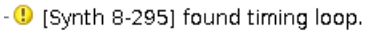


Figure C.5.: An example of the warning message stating that your design has a combinatorial loop.

A trivial example of a combinational loop is given in Listing C.4. The intermediate signals `a` and `b` depend on each others' values, leading to a cycle in the RTL diagram. Additionally, there are no memory elements separating this dependency, so changes to one of the signals will immediately affect the other.

This example also includes a situation which leads to infinite oscillations, namely when the input `i` transitions from 1 to 0. When the signal `i` is 1, `a` will also be 1 regardless of the value of `b`. Since `b` is 1 if `a` and `c` are equivalent, `b` is also set to 1. Now, when `c` changes to 0 `a` remains 1 since `b` is 1. However, `a` and `c` now have different values, so `b` is updated to 0. This causes `a` to be set to 0, since neither `b` nor `i` is 1. This leads to `a` and `i` being equivalent again, as both are now 0, so `b` is updated to 1. This change again leads to an update of `a`, and an endless update cycle ensues. Indeed, if this test input is run in the simulator one gets the error message *"FATAL ERROR: Iteration limit 10000 reached. Possible zero delay oscillation where simulation time can not advance."*.

Listing C.4: A trivial example containing a combinational loop.

```
1  entity combinational_loop is
2    port (
3        signal i : in std_logic
4      ; signal o : out std_logic
5    );
6  end combinational_loop;
7
8  architecture Behavioral of combinational_loop is
9      signal a, b : std_logic;
10 begin
11
12 process (a, b, i) is
13 begin
14
15 a <= b or i;
16 b <= not (i xor a); -- Calculates the equivalence of i and a
17
18 end process;
19
20 o <= a xor b;
21
22 end Behavioral;
```

The previous example may seem silly, as the resulting circuit is obviously unstable. Matters quickly get more convoluted once a design grows, however, with multiple modules connected together. When making a five-stage pipelined processor, for instance, outputs

from one stage is typically connected to input of the next stage. However, inclusion of stall and forwarding logic may necessitate a value propagation in the opposite direction. If one is not sufficiently pedantic when considering which pipeline registers to include, one may end up with combinational paths across pipeline stages in less-than-obvious ways.

In summary, when Vivado warns you that there is a combinational path, it is best to consider this an indication that the design is either flawed or the implementation bugged. Trying to manually verify that instability cannot occur is painful, error-prone, and leads to low-performing designs.

C.3. Variables

This section concerns the VHDL-notion of variables. Admittedly, it is debatable to call the use of variables in VHDL an antipattern, since their employment does not necessarily cause any direct harm. However, it is never required to use variables when writing synthesizable VHDL (defined in Section 2.2). This section will try to convince you that the alternative leads to clearer hardware descriptions.

The main argument again variables is that their sequential-update semantics are at odds with the semantics of hardware, where updates happen continuously in parallel. Therefore, an urge to use variables in VHDL is often a sign that one is thinking in terms of behaviour (like a software programmer) and not in terms of the synthesized hardware (like a digital designer). Resisting the temptation, instead trying to create a working design through structural composition without relying on sequential update semantics, is therefore typically sound advice.

A second argument is that another digital designer trying to understand your work will have to translate the sequential update semantics into suitable hardware structure to understand how the hardware will be implemented. As an example, consider the clocked process in Listing C.5. It is easy to see how the input control signals impact the register `x`, and that the `o1` output is set based on the new value of `x`. However, it is not easy to see how the synthesis tool will manage to translate the behaviour into hardware. For instance, the register required to store `x` is apparently updated before `o1`. However, since register updates happen simultaneously in hardware this cannot be the case. Instead, what happens is that `o1` is calculated based on the new value waiting to be stored in the register required for `x`, i.e. the calculation of `o1` is based on the input to the register `x`. In the semantically equivalent code based on signals, presented in Listing C.6, this fact is arguably clearer. The next value of `x_reg` is computed and represented by `x`, which is used to generate next values for both `x_reg` and `o1`.

Listing C.5: A process using variables to store a counter value.

```
1 process (clk, rst) is
```

```
2       variable x : unsigned(3 downto 0);
3   begin
4       if rst = '1' then
5           x := (others => '0');
6           o1 <= '0';
7       elsif rising_edge(clk) then
8           if i1 = '1' then
9               x := x + 1;
10          end if;
11          if i2 = '1' then
12              x := x - 1;
13          end if;
14          if i3 = '1' then
15              o1 <= x(3) xor x(2) xor x(1) xor x(0);
16          end if;
17      end if;
18  end process;
```

Listing C.6: An architecture using signals to store a counter value and represent its next value.

```
1   architecture signals of variable_tests is
2       signal x, x_reg : unsigned(3 downto 0);
3   begin
4
5   x <= x_reg + 1 when i1 = '1' and i2 = '0' else
6        x_reg - 1 when i2 = '1' and i1 = '0' else
7        x_reg;
8
9   process (clk, rst) is
10  begin
11      if rst = '1' then
12          x_reg <= (others => '0');
13          o1 <= '0';
14      elsif rising_edge(clk) then
15          x_reg <= x;
16          if i3 = '1' then
17              o1 <= x(3) xor x(2) xor x(1) xor x(0);
18          end if;
19      end if;
20  end process;
```

To conclude, although using variables may seem enticing for a developer, the fact that variable semantics do not match hardware semantics can make the resulting hardware description more opaque to other readers of the code as well as the author him/herself. Since it is possible to write proper, synthesizable VHDL without using variables, it is recommended to leave this VHDL construct for writing testbenches[1].

[1] When writing testbenches, however, variables can be very useful - a testament to the origins of VHDL as a verification tool.

D. The MIPS Instruction Set Architecture

In the TDT4255 lab exercises, you will be responsible for implementing a MIPS like Instruction Set Architecture (ISA). MIPS is an acronym for Microprocessor without Interlock Pipeline Stages. MIPS is very popular microprocessor in embedded devices. Instruction words could be set up as follows:

Table D.1.: R-Type instruction format

name	opcode	rs	rt	rd	shamt	funct
bits	31–26	25–21	20–16	15–11	10–6	5–0

Table D.2.: I-Type instruction format

name	opcode	rs	rt	immediate
bits	31–26	25–21	20–16	15–0

Table D.3.: J-Type instruction format

name	opcode	target
bits	31–26	25–0

R-Type: This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions (jalr and jr). All R-type instructions use a 000000 opcode. The operation is specified by the function field.

- `opcode`: is the instruction opcode, and function specifies a particular arithmetic operation.
- `rs, rt and rd` : are source and destination registers
- `funct` field used for choosing the instruction's behaviour (ADD, SUB, AND etc.)
- `shamt`: no of bits to be shifted

I-Type : This group includes instructions with an immediate operand, branch instructions, and load and store instructions. In the MIPS architecture, all memory accesses

are handled by the main processor, so coprocessor load and store instructions are included in this group. All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions

- `rt` is the destination for lw, but a source for beq and sw.
- `imm` is a 16-bit signed constant.

J-Type: This group consists of the two direct jump instructions (j and jal). These instructions require a memory address to specify their operand. J-type instructions use opcodes 00001x.

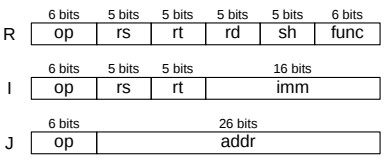
More detailed information about the MIPS architecture is given in Fig.D.1:

MIPS reference card

add	rd, rs, rt	Add	rd = rs + rt	R 0 / 20
sub	rd, rs, rt	Subtract	rd = rs − rt	R 0 / 22
addi	rt, rs, imm	Add Imm.	rt = rs + imm±	I 8
addu	rd, rs, rt	Add Unsigned	rd = rs + rt	R 0 / 21
subu	rd, rs, rt	Subtract Unsigned	rd = rs − rt	R 0 / 23
addiu	rt, rs, imm	Add Imm. Unsigned	rt = rs + imm±	I 9
mult	rs, rt	Multiply	{hi, lo} = rs * rt	R 0 / 18
div	rs, rt	Divide	lo = rs / rt; hi = rs % rt	R 0 / 1a
multu	rs, rt	Multiply Unsigned	{hi, lo} = rs * rt	R 0 / 19
divu	rs, rt	Divide Unsigned	lo = rs / rt; hi = rs % rt	R 0 / 1b
mfhi	rd	Move From Hi	rd = hi	R 0 / 10
mflo	rd	Move From Lo	rd = lo	R 0 / 12
and	rd, rs, rt	And	rd = rs & rt	R 0 / 24
or	rd, rs, rt	Or	rd = rs | rt	R 0 / 25
nor	rd, rs, rt	Nor	rd = ~(rs | rt)	R 0 / 27
xor	rd, rs, rt	eXclusive Or	rd = rs ^ rt	R 0 / 26
andi	rt, rs, imm	And Imm.	rt = rs & imm0	I c
ori	rt, rs, imm	Or Imm.	rt = rs | imm0	I d
xori	rt, rs, imm	eXclusive Or Imm.	rt = rs ^ imm0	I e
sll	rd, rt, sh	Shift Left Logical	rd = rt << sh	R 0 / 0
srl	rd, rt, sh	Shift Right Logical	rd = rt >>> sh	R 0 / 2
sra	rd, rt, sh	Shift Right Arithmetic	rd = rt >> sh	R 0 / 3
sllv	rd, rt, rs	Shift Left Logical Variable	rd = rt << rs	R 0 / 4
srlv	rd, rt, rs	Shift Right Logical Variable	rd = rt >>> rs	R 0 / 6
srav	rd, rt, rs	Shift Right Arithmetic Variable	rd = rt >> rs	R 0 / 7
slt	rd, rs, rt	Set if Less Than	rd = rs < rt ? 1 : 0	R 0 / 2a
sltu	rd, rs, rt	Set if Less Than Unsigned	rd = rs < rt ? 1 : 0	R 0 / 2b
slti	rt, rs, imm	Set if Less Than Imm.	rt = rs < imm±? 1 : 0	I a
sltiu	rt, rs, imm	Set if Less Than Imm. Unsigned	rt = rs < imm±? 1 : 0	I b
j	addr	Jump	PC = PC&0xF0000000 | (addr0<< 2)	J 2
jal	addr	Jump And Link	$ra = PC + 8; PC = PC&0xF0000000 | (addr0<< 2)	J 3
jr	rs	Jump Register	PC = rs	R 0 / 8
jalr	rs	Jump And Link Register	$ra = PC + 8; PC = rs	R 0 / 9
beq	rt, rs, imm	Branch if Equal	if (rs == rt) PC += 4 + (imm±<< 2)	I 4
bne	rt, rs, imm	Branch if Not Equal	if (rs != rt) PC += 4 + (imm±<< 2)	I 5
syscall		System Call	c0_cause = 8 << 2; c0_epc = PC; PC = 0x80000080	R 0 / c
lui	rt, imm	Load Upper Imm.	rt = imm << 16	I f
lb	rt, imm(rs)	Load Byte	rt = SignExt(M1[rs + imm±])	I 20
lbu	rt, imm(rs)	Load Byte Unsigned	rt = M1[rs + imm±] & 0xFF	I 24
lh	rt, imm(rs)	Load Half	rt = SignExt(M2[rs + imm±])	I 21
lhu	rt, imm(rs)	Load Half Unsigned	rt = M2[rs + imm±] & 0xFFFF	I 25
lw	rt, imm(rs)	Load Word	rt = M4[rs + imm±]	I 23
sb	rt, imm(rs)	Store Byte	M1[rs + imm±] = rt	I 28
sh	rt, imm(rs)	Store Half	M2[rs + imm±] = rt	I 29
sw	rt, imm(rs)	Store Word	M4[rs + imm±] = rt	I 2b
ll	rt, imm(rs)	Load Linked	rt = M4[rs + imm±]	I 30
sc	rt, imm(rs)	Store Conditional	M4[rs + imm±] = rt; rt = atomic ? 1 : 0	I 38

pseudo-instructions

bge	rx, ry, imm	Branch if Greater or Equal
bgt	rx, ry, imm	Branch if Greater Than
ble	rx, ry, imm	Branch if Less or Equal
blt	rx, ry, imm	Branch if Less Than
la	rx, label	Load Address
li	rx, imm	Load Immediate
move	rx, ry	Move register
nop		No Operation



registers

$0	$zero
$1	$at
$2–$3	$v0–$v1
$4–$7	$a0–$a3
$8–$15	$t0–$t7
$16–$23	$s0–$s7
$24–$25	$t8–$t9
$26–$27	$k0–$k1
$28	$gp
$29	$sp
$30	$fp
$31	$ra
hi	—
lo	—
PC	—
co $13	c0_cause
co $14	c0_epc

syscall codes for MARS/SPIM

1 print integer
2 print float
3 print double
4 print string
5 read integer
6 read float
7 read double
8 read string
9 sbrk/alloc. mem.
10 exit
11 print character
12 read character
13 open file
14 read file
15 write to file
16 close file

exception causes

0 interrupt
1 TLB protection
2 TLB miss L/F
3 TLB miss S
4 bad address L/F
5 bad address S
6 bus error F
7 bus error L/S
8 syscall
9 break
a reserved instr.
b coproc. unusable
c arith. overflow

F: fetch instr.
L: load data
S: store data

Figure D.1.: MIPS Quick Reference

Bibliography