Lab Workbook

SS 2024

Industrial Hardware Verification

Jakob Lechner
Markus Ferringer

Part 1: VHDL & FLI & OSVVM

Contents

Lab Mode

- You have to elaborate these exercises on your own. This lab part is **not** a team effort.
- We provide stubs containing basic templates for you which are to be extended with your implementations
- Questions are to be ansered in the respective source-files of the corresponding exercise. Just add appropriate comments at the end
- Additional files, should there be any, must be located in the folder corresponding to the exercise
- I strongly recommend to read through an entire task before starting with the implementation (including the questions section). You will sometimes find hints later that might simplify things
- Once done with all exercises, please execute `./make.sh clean` in each folder, zip everything (`ex1 - ex7`) and submit it via TUWEL

Chapter 1

Installation

This chapter explains how to install the necessary software and get the lab exercises. You can do most of the work on your own personal computer. Only when special simulator features (such as Code Coverage or PSL) are needed, you need to work on the lab machines in order to use the respective licenses.

Both Windows and Linux should work, although Linux will be less effort to set up. While it is possible to use Microsoft Windows (eg., using CygWin or WSL), these instructions focus on Linux-based systems like Ubuntu.

You don't need to install any software on the Lab Workstations, everything you need is preinstalled.

1.1 ModelSim/QuestaSim

On the lab workstations, everything is set up. Should you want to work with your personal equipment, you need to download the simulator first:

1. Download ModelSim-Intel FPGA Starter Edition. Sounds easy? Well...
2. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html
3. You need to register. It's free, but it's a requirement.
4. Once registered, sign in; Unfortunately, the site is not well structured...
5. Currenlty the download link is on the start page. This can of course change at any time.
6. Maybe also try this link for download: https://www.intel.com/content/www/us/en/software-kit/819742/questa-intel-fpgas-pro-edition-software-version-24-1.html
7. Unfortunately, you also need a license. It's free, but it's a pain to get it. You have to follow the instructions on the webpage for "Request a License File from the Self Service Licensing Center"
8. Good luck!

The free version has a couple of restrictions, so for some exercises you have to use the lab workstations.

- Reduced performance (which will not be an issue, though)
- No support for code coverage → use lab workstations
- No support for PSL → use lab workstations

1.2 C Compiler

For the FLI examples you need a C compiler. Under Linux, `gcc` is just perfect. Under Windows, I used `mingw32`. Please notice that you need a 32 bit compiler (or one that can compile in 32 bit mode using a switch like `-m32`). It might also be necessary to install 32 bit support first, which can be done with

- (Ubuntu): `sudo apt-get install libc6-dev-i386`
- (CentOS): `sudo yum install glibc-devel.i686`

1.3 Source Code

We prepared a framework for you which contains

- OSVVM (in a separate folder)
- Exercises (each in a separate folder)
- A suitable `make.sh` (one for each exercise)
- Stubs for all the exercises (containing basic templates where you can fill in your solution)
- The entire source-code needed for all exercises

You can download the execises via TUWEL. It will be available after the respective lectures.

Remark for Windows Users: The `make.sh` files are symbolic links, which are not supported in Windows. If your make files don't work or are empty, try to copy the original `make.sh` from the root-folder into the respective exercise folder.

1.4 Compilation

Each exercise is located in a separate folder. There is a `make.sh` file which supports the following targets. Please notice that this is not a regular Makefile, but rather a shell script named `make.sh`. I prefer this over Makefiles.

- `./make.sh clean`: Cleans up temporary files and compiled libraries
- `./make.sh sim`: Compiles the exercise's code, then executes the simulation. When called the very first time (or after a `make.sh clean`), OSVVM is compiled.
- `./make.sh gui`: Opens the ModelSim/QuestaSim GUI and sources the OSVVM scripts, so you can use them right away

You need to provide the path to your / the lab's ModelSim/QuestaSim binary. In the root folder (the one that contains OSVVM and all exercises) there is a file called `settings.make`. Open it and edit the `MODELSIM_PATH` variable according to your setup. If ModelSim is available through your `PATH` variable, just keep the default assignment `MODELSIM_PATH=vsim`.

For the FLI example, you also need to provide the path to ModelSim's `include` folder. Just edit the `compile.sh` script accordingly. Depending on your setup, you might also need to change the C compiler used. There is a `compile_win.bat` for Windows, but you need to update the paths as well.

1.5 Editor

Of course, you can use any editor you like. However, it will be more efficient (and way less frustrating) if you are using a good editor which is capable of understanding VHDL, and which offers at least some degree of support (like parameter help, tooltips, code completion, etc.). You have to deal with a lot of packages and libraries, and 100s of functions, procedures, and datatypes. You should use all the support you get from a good IDE.

As I have written my own VHDL editor (an extension for Visual Studio Code), I will of course recommend this IDE to you :-) VS Code is Open Source and thus for free, and I will provide you with a license for my extension (it will be included in the lab exercises, you don't have to worry about it).

- Download and install VS Code from `https://code.visualstudio.com/`
- Open VS Code (`code` in the terminal)
- On the left side, open the `Extensions` view
- In the search field, enter `VHDL for professionals` (V4P)
- Select `VHDL for Professionals` from the results, and click the `install` button
- If requested, restart VS Code
- Open file `workspace.code-workspace` which is located in the root folder of the provided zip file

All (VHDL-based) exercises come with a preconfigured VS Code workspace. Just open it in VS Code, and you are set to go. All the OSVVM libraries, functions, types, etc. are at your disposal. The license for V4P will also be part of the workspace.

Chapter 2

VHDL

2.1 Exercise 1: Attributes

2.1.1 Task 1 [1 Point]

Write a function `ColorToString(pos: integer) return string` with the following properties:

- The function shall take an integer parameter which specifies the "index" of the enum-value in type `ColorT`
- If the index is invalid, the function shall call `Alert()` with an appropriate error message and return `''OutOfRange''`
- If the index is valid, the string representation of the color at position `pos` shall be returned
- The function shall work without modification even if colors are added or removed to/from `ColorT`
- In the main process `stimuli_p`, add some calls to `ColorToString()` and see if it works
- Example:

```
architecture beh of ent is
  type ColorT is (Red, Green, Blue);
  ...
begin
  -- Expected output: ''green''
  report ColorToString(1) severity note;
end architecture;
```

2.1.2 Task 2 [1 Point]

Write a function `ColorsToList() return string`

- The function shall return a comma-separated list of all enumeration fields that are part of the `ColorT` type
- Example:

```
  architecture beh of ent is
    type ColorT is (Red, Green, Blue);
    ...
  begin
    -- Expected output: "AllEnums: red, green, blue"
    report "AllEnums: " & ColorsToList() severity note;
  end architecture;
```

- Hints
  - Use dynamic strings: Define a string access type (don't use `textio.line`)

  - Make sure the function handles one-element enums
  - The function shall work without modifications even if the `ColorT` type is changed

2.2 Exercise 2: Transactions

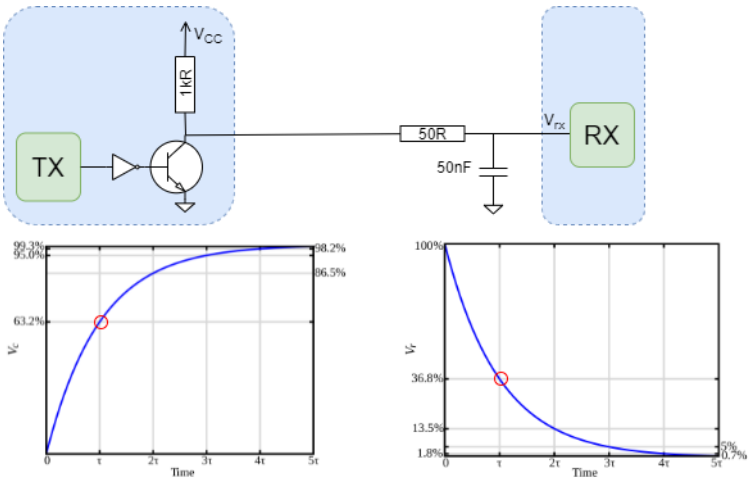
2.2.1 Task 1 [1 Point]

- We provide a very simple state machine that is controled by process `stimuli_p`. Every 20 clock cycles, the state is advanced.
- Take a look at process `state_p`: Depending on the state, signal `output` is modified in different ways
- Implement a process `event_count_p` that increments `eventCounter` every time `output` changes
- Implement a process `trans_count_p` that increments `transCounter` every time a transaction occurs on `output`
- Answer the following questions (directly in the source file as comment):
  - Explain the difference between the two processes and how they are triggered
  - Is there a difference between the assignment statements of the states `Idle, NotAffected, Keep`?

2.2.2 Task 2 [1 Point]

- Enhance process `stimuli_p` such that the testbench self-checks the counter values
- After each call to `WaitForClock()` in `stimuli_p`, add `AffirmIfEqual(actual, expected, message)` statements. Just hardcode the expected values.
- Open up the GUI (`./make.sh gui`), and execute `build project.pro`
- Now add all the signals to the waveform viewer. Save the waveform (filename `wave.do`)
- Restart the simulation with `build project.pro`
- Inspect the results in the waveform viewer. If there is a `wave.do` it is automatically opened.
- Answer the following questions (directly in the source file as comment):
  - How can you make `output'transaction` visible in the waveform viewer?
  - Why is `stimuli_p` reacting on the falling edge, while `state_p` triggers on the rising edge? Why don't both processes just wait for the rising edge?

2.3 Exercise 3: Delay Modes

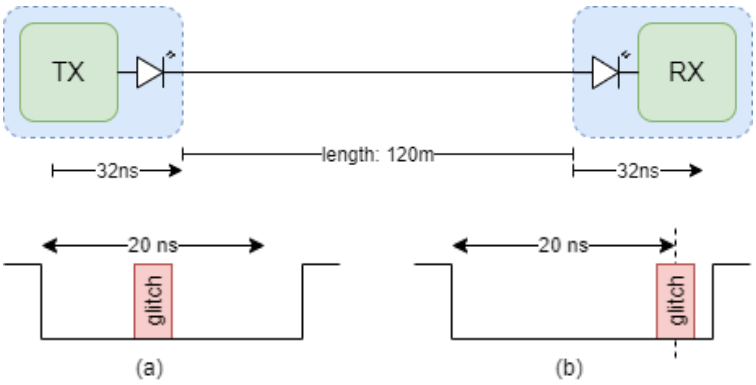


2.3.1 Task 1 [2 Points]

Consider the electrical transmission line in the figure above.

- Model the physical transmission line in VHDL
  - Output is open collector with a $1k\Omega$ pull-up resistor
  - The transmission line itself has an impedance of 50Ω and a parasitic capacitance of $50nF$ at the receiver
  - Assume that the delay from TX to RX is $23ns$
  - Assume that RX recognizes $V_{rx} \geq 0.632V_{cc}$ as logical 1 (1τ as indicated)
  - Assume that RX recognizes $V_{rx} \leq 0.368V_{cc}$ as logical 0 (1τ as indicated)
  - If V_{rx} is between these two thresholds, the internal logic value does not change
  - The transistor can be modeled as ideal switch
  - You can assume that signal transitions are far enough separated so that they don't interfere with each other

2.3.2 Task 2 [1 Point]



Now model an optical transmission line with the following properties

- Fiber length is 120m (speed of light is around 200000km/s in the optical medium)
- The laser's propagation time from electrical input to optical output is 32ns for both rising and falling edges

- After a transition on the input, the laser ignores all glitches on the input for 20 ns (see waveform above)
  - You can assume that "regular" input transitions are separated by at least 20 ns
  - You can assume that glitches fully occur (and disappear again) within those 20 ns (subimage (a) of the waveform)
  - You **don't** need to handle glitches that are exactly on the border (subimage (b) of the waveform)
- On the receive side, the optical detector has the same characteristics as the transmitter
- As the transmission line is quite long, you cannot longer assume that only one signal transition is transmitted at a time (i.e., a new transition can be sent before the last one has arrived)
- Hint: You will need an intermediate signal to combine `inertial` and `transport` delays

2.4 Exercise 4: Advanced Data Types, Generics

2.4.1 Task 1 [4 Points]

Your task is to implement a (singly) linked list in VHDL

- The list shall be implemented as protected type inside package `linked_list`.
- The linked list shall be located in a generic package, which takes two generic parameters
  - `type ElementT`: The type of the items that are stored in the list
  - `function ToStringF(item: ElementT) return string`: A function that converts an item to a string representation
- This generic data type denotes the type of the elements that can be stored inside the list
- Implement the following functions/procedures:
  - `function Count return integer`: Returns the number of elements in the linked list
  - `procedure AddFirst(item: ElementT)`: Adds an element to the front of the list
  - `function GetAt(index: integer) return ElementT`: Return the i-th item of the list. The first item (head of list) has index 0.
  - `procedure RemoveAt(index: integer)`: Remove the i-th item from the list. The first item (head of list) has index 0.
  - `function Dump return string`: Returns a comma separated list of string representations of all items in the list (in the stored order, of course)
- In case of invalid operations (eg, get a value from an empty list), create a log entry by calling `Alert(...)`.
- In case of invalid operations, if there is a mandatory return value, just return VHDL's default for the respective data type
- Avoid obvious memory leaks. (When returing a dereferenced pointer-to-string from a function, there is no (easy?) way to avoid leaks)

2.4.2 Task 2 [3 Points]

Your task is to implement a testbench which tests the linked list (process `stimuli_p`)

- Instantiate the generic list, use `PrimeRecT` as generic type and implement the `to_string()` function for this data type
- Test all public subroutines that are part of the public interface
- Make sure that Statement Coverage is at 100% (for the linked-list protected type)
- You can use the GUI (Tools - Coverage Report...) to generate annotated source files
- Attach the annotated source code (coverage report) to your submission

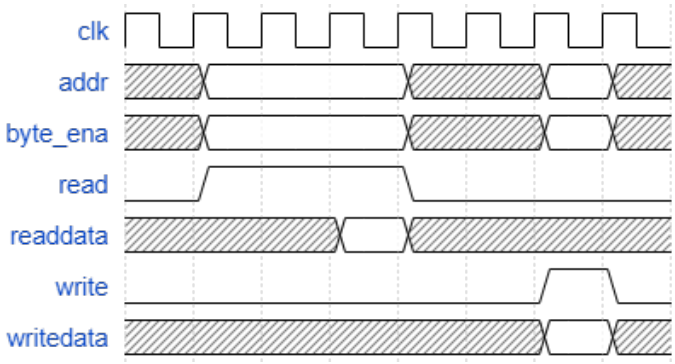
Chapter 3

OSVVM

3.1 Exercise 5: OSVVM Verification Unit

3.1.1 Task 1 [4 Points]

Your task is to implement a Verification Unit for the Avalon Memory Mapped (AVMM) Bus. Your are implementing a master device, which can in turn control DUTs with memory mapped slave interfaces.



- `addr, byte_ena, read, write, writedata` are **outputs** of the master, while `readdata` is an **input** to the master
- The read and write timing is shown in the image. This timing is fixed, and need not be configurable. The shown waveform corresponds to `writeWaitTime=0` and `readWaitTime=2` according to the Avalom Memory Mapped Interface Specification. It means that writes take place immediately, and reads have a delay of two cycles until data is available.
- It is not necessary to support simultaneous read and write access.
- The interface uses byte enables. For an ordinary write operation, the byte_ena is supplied with the address and data. For read-modify-write you can calculate the byte_ena from the provided bitmask (see below).
- The interface does not use `WaitRequest`, and `ReadDataValid` (the latter is implicit due to timing). You don't have to support bursts. You don't have to support pipelined reads / writes (i.e., just one read/write at a time, until the entire sequence is finished)
- In other words, you only have to support the simple cases shown in the figure.
- There is a template package file which contains definitions of the public interface of the VU
  - Implement the subroutine bodies of the respective header definitions

  - Use the `AddressBusTransactionRecT` data structure, it is well suited for this task
  - There are a couple of predefined OSVVM subroutines (eg, `send, read, write`) which are already performing lots of the necessary steps. Use them, or review them to adapt you own implementation
  - You might want to take a look at OSVVM's UART and AXI implementation for reference
  - The command `AvmmReadModifyWrite` has an argument `write_mask`. It is of the same length as the data vector, and a '1' on a specific location means that this bit is written to the register accoring to the supplied data. A '0' means the bit's old value is preserved. Out of the bitmask, the byte-enables can easily be derived.
- There is the main VU's source file which already contains a basic skelton of the VU
  - For simplicity, both reading and writing shall be blocking - so there is no need to implement FIFOs to buffer incoming or outgoing data transactions
  - Implement the VU. As there are only blocking operations, it is simplest to just use one process which does all the work (read data, write data)

3.1.2 Task 2 [3 Points]

Your task is to implement a basic (directed) testbench for your VU

- Test all the functions of the public interface
  - Use an address-width of 5 bits, and a data-width of 16 bits
  - Use a scoreboard. For each call to one of the public functions, create the respective expected waveforms on the avalon bus and store them in the scoreboard.
  - Write a process that monitors the bus; put the observed waveforms in the scoreboard for checking.
  - For this exercise, it is good enough to hardcode the waveforms. Usually, you should of course implement a reference model.
- Reach 100% statement coverage for the main VU design, and the public package

3.2 Exercise 6: Constrained Random Testing

3.2.1 Task 1 [4 Points]

Addr	7	6	5	4	3	2	1	0
0	IE	IF	-					RST
1	NAME				CHK			
2	-				ID			
3	CNT							

Name	Access	Default	Remarks
RST	R/W	0	Write 1 to reset all registers to their default. The field auto-clears to zero.
IF	R	1	Initialized with 1, auto-clears when read. Then remains 0 until RST
IE	R/W	0	
CHK	R	1010	NAME xor ID
NAME	R/W	0000	
ID	R	1010	
CNT	R	00000000	Increments by one with each clock cycle. Overflows to 0 when maximum is reached

Your task is to use constrained random testing in combination with functional coverage to test a given DUT.

- The DUT is a simple register interface

- The register description is shown in the table
- Use the procedures `Read` and `Write` to handle Avalon access to the register interface
- use the procedure `ReferenceModel` as a reference for checking the DUT against a (hopefully bug-free) reference implementation
- It is not necessary to test the bus protocol itself. Limit your testing to sending valid read/write requests.
- Use OSVVM's functional coverage capabilities to specify cover points for the defined register fields
  - Eg, all addresses have been read from
  - Eg, all addresses have been written to
  - Eg, all addresses have been read directly after they have been written
  - ...
- The DUT has some undocumented "features/bugs", which should not be detectable with simple directed tests.
- Can you find such "features/bugs" using constrained random testing? Which ones?
- Please notice that you don't have to modify the DUT. You can just ignore the *insert your code here* comments!

Chapter 4

FLI

4.1 Exercise 7: FLI

4.1.1 Task 1 [3 Points]

Your task is to implement a Mandelbrot Set renderer in VHDL and C, and compare the performance of both variants

- We prepared a basic framework for you to easily compile C code and use it from within vhdl
  - Use `compile.sh` to compile for linux. It should work right away
  - Use `compile_win.bat` to compile for windows. You need to update the path to your compiler. Might make problems...
- Calculcating the Mandelbrot set is quite easy. Google is your friend.
- In short, given a complex number c, calculcate $z_{n+1} = z_n^2 + c$ with $z_0 = 0$. If z_n evaluates to ($|z_n| > 2.0$), the point is not part of the set.
- We are counting and "colorizing" n, i.e. the number of iterations we need until the point escapes. We also need to define an upper limit for n for which we consider the point to belong to the set. We define it to be 200.
- Implement the following functions / procedures in C
  - `procedure GetTimeC(hour, minute, second: out integer)`: Write a C function to get the current time (including seconds). We need this for performance measurement.
  - `procedure OneStepC(zr, zi, cr, ci: real; or, oi: out real)`: Write a function that performs one step of the mandelbrot calculation $o = z^2 + c$. The parameters are the real/imaginary parts of z, c, and the output o, resp.
  - `function IterateC(x, y: real) return integer`: Calculate the Mandelbrot escape value for the given (complex) point (x, y). Don't do more than 200 iterations. Stop if the magnitude of the complex vector exceeds 2.0
  - Take a look at the data type mapping table in the slides to learn how to pass certain data types back and forth via FLI
- Define the subroutine defintions in VHDL and mark them as foreign.
- Implement the same functions and procedures in VHDL. You need to give them distinct names because we want to use both VHDL and C functions at the same time. Use the `ieee.math_complex.Complex` data type.
- Run the following simulations and record the execution times:
  - Everything VHDL: `Image` calls `Iterate` calls `OneStep`

- `Image` calls `Iterate` calls `OneStepC`
- `Image` calls `IterateC` (calls `OneStepC` directly in C)