



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ARTO OINONEN
IMPLEMENTATION OF SYSTEMVERILOG AND UVM TRAINING

Master of Science Thesis

Examiners:
Professor Timo D. Hämäläinen and
Doctor Teemu Laukkarinen
Examiner and topic approved by the
Council of the Faculty of Computing
and Electrical Engineering
on 9th December 2015

ABSTRACT

ARTO OINONEN: Implementation of SystemVerilog and UVM Training

Tampere University of Technology

Master of Science Thesis, 72 pages

January 2017

Master's Degree Programme in Electrical Engineering

Major: Embedded Systems

Examiner: Prof. Timo D. Hämäläinen, Dr. Teemu Laukkarinen

Keywords: SystemVerilog, UVM, verification, education

Integrated circuits have become more complex every year and their verification has become more time-consuming. Therefore, effective education of new verification engineers is important for industry. This thesis covers planning of an efficient exercise package for education of verification engineers. The exercises cover key principles of SystemVerilog language and Universal Verification Methodology (UVM). The object of the exercise package is that a person with programming experience but no previous experience of system design or verification should be able to digest the most important concepts in five training days and be able to perform verification tasks using UVM after the training.

The planned exercise package was divided into four exercises on SystemVerilog language and seven exercises on UVM, which cover the methods the designer can use to aid in verification process and the basic principles of UVM methodology. The exercises were implemented as independent work so that the assistant was present to help solving problems and to answer questions. The planning of the exercises adapted to the needs of the participants on different levels so that every student was able to learn the most important concepts and additional more advanced tasks were provided for faster students. The advanced tasks did not introduce new crucial concepts, but deepened the understanding of the concepts used in the mandatory exercises.

The exercises were used as a part of digital design and verification education, where the participants had a programming background. The completion of learning objectives was metered by a time usage survey and a feedback form. Based on the results the learning objectives were fulfilled and every student was able to comprehend the concepts. The students were contented with the content and the structure of the exercises.

TIIVISTELMÄ

ARTO OINONEN: SystemVerilog- ja UVM-harjoitusten toteutus

Tampereen teknillinen yliopisto

Diplomityö, 72 sivua

Tammikuu 2017

Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Sulautetut järjestelmät

Tarkastajat: Prof. Timo D. Hämäläinen, TkT Teemu Laukkarinen

Avainsanat: SystemVerilog, UVM, varmennus, verifiointi, koulutus

Integroitujen piirien koko on kasvanut jatkuvasti ja suunnittelun varmennukseen käytetty aika on entistä suurempi osa projektin kokonaiskestoista. Siksi on tärkeää kouluttaa uusia varmennuksen osaajia mahdollisimman tehokkaasti yritysten tarpeisiin. Tämä diplomityö kuvaa tehokkaan varmennusharjoituspaketin suunnitteluprosessia. Harjoituksissa käydään läpi SystemVerilog-kielen sekä UVM-varmennusmenetelmän keskeisimmät ominaisuudet. Harjoitusten tavoitteena on, että työntekijä, jolla on ohjelmointitaustaa mutta ei aiempaa suunnittelu- tai varmennuskokemusta, pystyy sisäistämään tärkeimmän sisällön viiden koulutuspäivän aikana ja työskentelemään koulutuksen jälkeen varmennustehtävissä käyttäen UVM-menetelmää.

Harjoituspaketti on jaettu neljään SystemVerilog- ja seitsemään UVM-harjoitukseen, joissa esitellään käytännöt joita suunnittelija voi tehdä varmennuksen kannalta sekä UVM-menetelmän perustoiminnot. Harjoitukset suunniteltiin itsenäisesti tehtäväksi niin että assistentti on paikalla avustamassa ongelmatilanteissa sekä vastaamassa kysymyksiin. Harjoitukset mukailevat opiskelijoiden erilaisia lähtötasoja niin että jokainen ehtii oppia perusasiat mutta nopeimmille opiskelijoille on syventäviä lisätehtäviä. Lisätehtävät eivät esittele uusia tärkeitä asioita vaan syventävät perusharjoituksissa hankittua tietoa.

Harjoituksia käytettiin osana Digitaalisen suunnittelun ja varmennuksen opintokokonaisuutta, minkä osallistujilla oli enimmäkseen ohjelmointitaustaa. Oppimistavoitteiden saavuttamista tarkkailtiin ajankäyttö- ja palautelomakkeiden avulla. Tuloksien perusteella oppimistavoitteet saavutettiin hyvin ja koulutukseen osallistujat olivat tyytyväisiä harjoitusten sisältöön sekä rakenteeseen.

PREFACE

This Master of Science Thesis was done for the Department of Pervasive Computing at Tampere University of Technology as a part of a digital design and verification education arranged by Edutech during fall 2015. The education team was full of great people and experts on their fields and I was privileged to be part of it.

The exercise package that was built in this thesis was designed as independent work, but the UVM exercises were partly based on OVM exercises designed by Ville Yli-Mäyry for a previous verification course arranged by the Department of Pervasive Computing at Tampere University of Technology.

I would like to thank professor Timo D. Hämäläinen, Dr. Teemu Laukkarinen and Taru Hakanen for all the support and for giving me this opportunity. I also want to thank Ville Yli-Mäyry for the basis of the UVM exercises.

The UVM video lectures on Verification Academy were of invaluable help when learning UVM. I extend special thanks to Tom Fitzpatrick for the video lectures and Dave Rich for active online help on multiple forums.

My deepest gratitude to my family and friends for all the support and for pushing me forward.

Tampere, 22.12.2016

Arto Oinonen

CONTENTS

1.	INTRODUCTION	1
1.1	Background of verification methodologies	2
1.2	Motivation, methods, and scope of thesis	4
1.3	Outline of thesis	6
2.	SYSTEMVERILOG	7
2.1	SystemVerilog for design	7
2.2	SystemVerilog for verification	12
3.	UNIVERSAL VERIFICATION METHODOLOGY (UVM)	16
3.1	UVM building blocks	17
3.1.1	Objects and components	18
3.1.2	Macros and methods	21
3.2	UVM architecture	24
3.2.1	UVM Environment	24
3.2.2	UVM Tests	26
4.	REQUIREMENTS AND METHODS FOR THE TRAINING	28
4.1	Requirements and student background	28
4.2	Tools used in the exercises	29
4.3	Structure of exercises	30
5.	SYSTEMVERILOG EXERCISES	33
5.1	Learning objectives	33
5.2	Structure	34
5.2.1	Exercise 1: Introduction to SystemVerilog	35
5.2.2	Exercise 2: Decrypter module	37
5.2.3	Exercise 3: Assertions and testbenches	39
5.2.4	Exercise 4: Bus connectivity	41
6.	UVM EXERCISES	43
6.1	Learning objectives	43
6.2	Procedures and UVM concepts used in the exercises	44
6.3	Structure	45
6.3.1	Exercise 1: Introduction to UVM	47
6.3.2	Exercise 2: Creating classes	49
6.3.3	Exercise 3: Subscribers and analysis ports	53
6.3.4	Exercise 4: Automatic checking and randomized input	57
6.3.5	Exercise 5: Verifying FIFO blocks	61
6.3.6	Exercise 6: Verifying the decrypter	61
6.3.7	Exercise 7: Integration level testbench	63
7.	TRAINING SESSIONS AND RESULTS	65
7.1	Learning outcomes	65
7.2	Student feedback	67
7.3	Problems faced	69

8. CONCLUSIONS.....72

LIST OF FIGURES

Figure 1.	<i>Design and verification gaps.</i>	1
Figure 2.	<i>ASIC Project time spent on verification, as found in the Wilson Research Group study in 2014. [12]</i>	2
Figure 3.	<i>The evolution of verification methodologies. [8]</i>	3
Figure 4.	<i>The ASIC design size trends in 2014. [12]</i>	4
Figure 5.	<i>ASIC/IC testbench methodology adoption trends. [11]</i>	17
Figure 6.	<i>UVM base class hierarchy. [3]</i>	19
Figure 7.	<i>UVM component class hierarchy. [3]</i>	19
Figure 8.	<i>Port – export – implementation connection. [3]</i>	20
Figure 9.	<i>UVM Phases. [5]</i>	22
Figure 10.	<i>A complete block level testbench. [5]</i>	24
Figure 11.	<i>The UVM exercise index page.</i>	30
Figure 12.	<i>An example of an exercise page. 1 describes a bullet with the concrete task and 2 is body text containing the motivation for the task. 3 is an example of a code block.</i>	31
Figure 13.	<i>An information block example.</i>	32
Figure 14.	<i>The complete encrypt-decrypt system. [20]</i>	37
Figure 15.	<i>A state diagram for the decrypter module.</i>	38
Figure 16.	<i>Block diagram of the testbench for the decrypter module.</i>	40
Figure 17.	<i>AXI wrapper.</i>	41
Figure 18.	<i>Testbench folder layout.</i>	45
Figure 19.	<i>The final testbench after finishing exercise 5.</i>	46
Figure 20.	<i>The base testbench delivered to the student.</i>	48
Figure 21.	<i>Testbench with the monitor attached.</i>	50
Figure 22.	<i>Delivering DUT interface to monitor.</i>	51
Figure 23.	<i>Coverage collector added to the testbench.</i>	54
Figure 24.	<i>Scoreboard added to the testbench.</i>	58
Figure 25.	<i>The final UVM testbench.</i>	60
Figure 26.	<i>The integration level testbench. [18]</i>	63
Figure 27.	<i>Completion of each exercise.</i>	66
Figure 28.	<i>Time usage in each UVM exercise.</i>	66

LIST OF SYMBOLS AND ABBREVIATIONS

AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced Extensible Interface
ASIC	Application Specific Integrated Circuit
DUT	Design under test
EDA	Electronic design automation
eRM	e Reference Manual
FIFO	First in, first out
IC	Integrated Circuit
IP	Intellectual property
OVM	Open Verification Methodology
PCI	Peripheral Component Interconnect
RAL	Register Abstraction Layer
RVM	Reference Verification Methodology
UVM	Universal Verification Methodology
USB	Universal Serial Bus
VHDL	Very High Speed IC Hardware Description Language
VMM	Verification Methodology Manual
XOR	Exclusive or operation

1. INTRODUCTION

The transistor revolution has brought complex digital systems as an integral part of everyday life where the applications of integrated circuits (ICs) vary from simple toys to mainframe datacenters. As the Moore's law has dictated, the number of transistors in integrated circuits has doubled approximately every two years, which has consequently increased the complexity of ICs. The increase in design complexity creates a burden on design and verification engineers so that the design effort needed for a project exceeds maximum productivity. A rendition of so-called design and verification gaps that portray the difference between effort and productivity is shown in Figure 1. The feature size that follows the Moore's law depicts the design effort needed for completing a complex project.

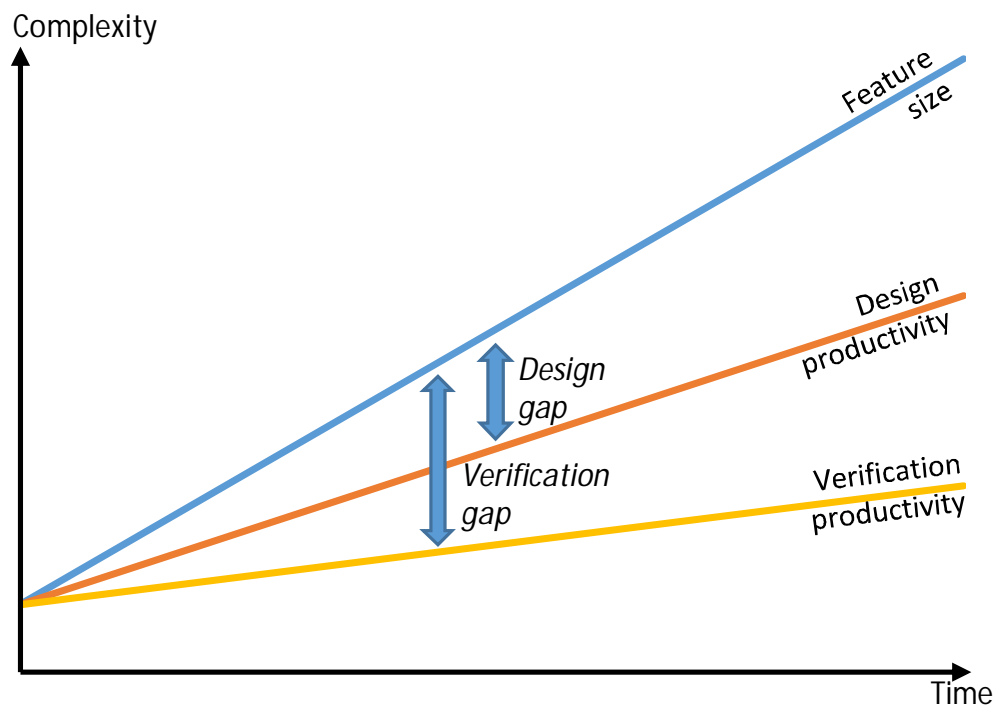


Figure 1. Design and verification gaps.

Increasing design reuse has decreased the design gap, but reuse in verification has not been as mature, so the trend is that the growth of the design gap will slow down but the verification gap will continue growing, as found in study done by Wilson Research Group in 2014 [12]. They found out that the percentage of application specific integrated circuit (ASIC) project time spent in verification has been growing rapidly as shown in Figure 2. Notably, the amount of projects where verification covers over 70% of the project time has been growing every time the study has been made.

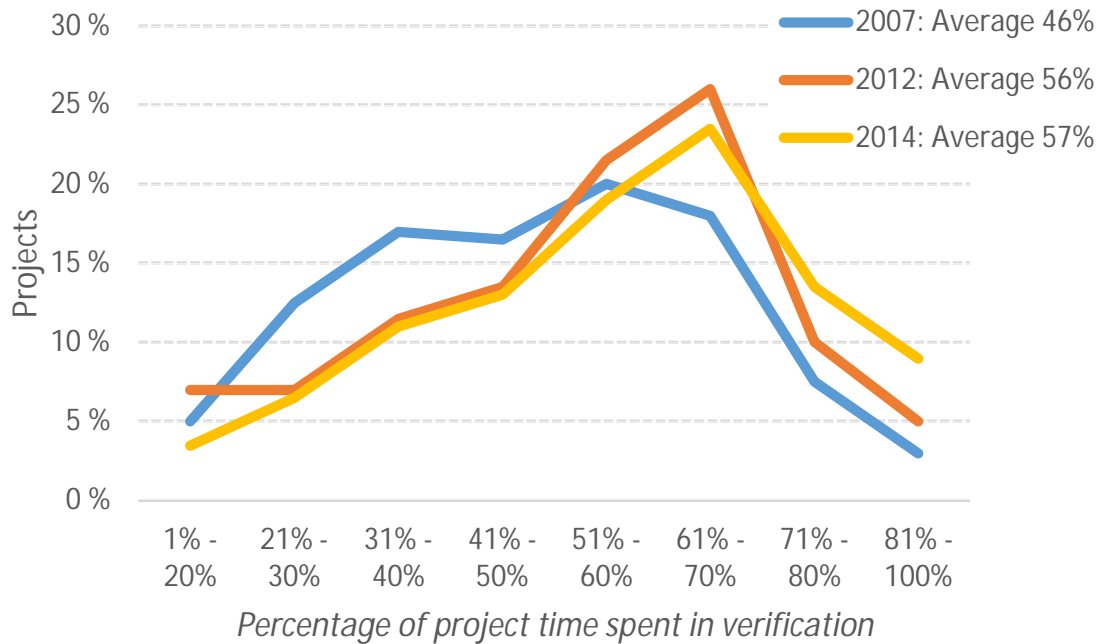


Figure 2. ASIC project time spent on verification, as found in the Wilson Research Group study in 2014. [12]

1.1 Background of verification methodologies

Verification is a process that examines if all the aspects of the design follow the specification correctly [9]. Verification can be *functional* or *formal*. The difference is that functional verification runs a simulation of the design under test (DUT) in a *testbench* that feeds the design test input and checks its output, when formal verification proves the correct functionality with a mathematical model. *Validation* is another procedure in design functionality checking. Validation is used to ensure that the design meets the functional requirements and the needs of the customer.

Traditionally systems were first designed using hardware description languages such as VHDL and Verilog and then functionally verified using testbenches written in the same languages [19]. However, the hardware description languages were not designed to be effective in verification purposes and the system complexity quickly outgrew the verification capabilities of the language. Writing complex tests for large systems would be more effective with languages that function on higher levels of abstraction.

Specific *verification languages*, such as e and Vera, were created as an answer to the problem [10]. They were closer to traditional high-level programming languages and introduced for example object-oriented properties, complex assertions, coverage metering and constrained randomization to aid verification engineers. The problems with these languages were that learning a completely new language for verification meant a lot of unproductive work and the large set of languages might have prevented design engineers

from doing also verification. In addition, some of the languages were tied to specific software tools produced by the developer of the language and were not supported in other tools. The worst-case situation was that a company had to use multiple verification languages with accompanying tools to verify different parts of the design.

Verification languages are powerful and versatile tools but writing verification environments from scratch every time for new tests is not effective. Some parts of the code in existing systems can be reused, but there are no standard procedures built into the languages and therefore components written by different verification engineers may not be compatible. Companies can have internal guidelines, but the procedures used in one company may not be compatible with the guidelines of the subcontractor or the verification IP vendor. To ease the reuse and to provide a common system for testbench design, *verification methodologies* have been introduced. Simply put the verification methodologies are guidelines of functional partitioning of the testbench so that most of the components can be reused in different test configurations. In addition, the methodologies could provide methods, macros and control of the simulation procedure.

All the three design tool market leaders - Cadence, Synopsys and Mentor Graphics - had multiple competing methodologies written in different languages [8]. A hierarchy of the evolution of some of the methodologies is shown in Figure 3. E Reuse Methodology was first introduced in 2002 and it was followed by competing OpenVera Reference Verification Methodology (RVM) developed by Synopsys and Advanced Verification Methodology (AVM) by Mentor Graphics. New methodologies were later introduced that were built on the previous methodologies.

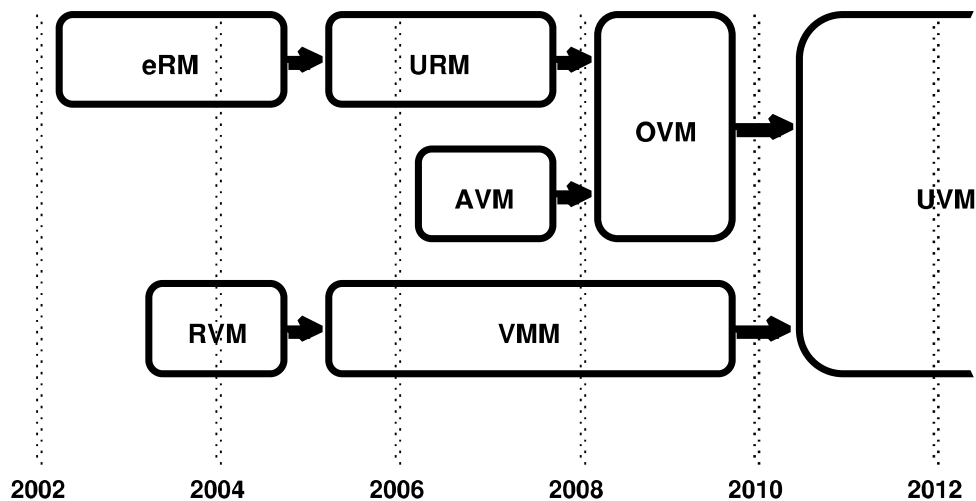


Figure 3. The evolution of verification methodologies. [8]

The diversity in languages, methodologies and tools created a need for universal industry standards. Accellera Systems Initiative was created as a consortium of EDA companies

for the task. The basis for a new standard language was a combination of Verilog hardware description language and OpenVera that was donated to Accellera. The result was *SystemVerilog* that was standardized in 2005 as an extension to Verilog [19]. A plan for a new standard methodology was introduced by Accellera in 2009 with a proposed name *Universal Verification Methodology (UVM)* [6]. It was built on the SystemVerilog language combining the previous methodologies OVM and VMM. SystemVerilog and UVM have since evolved and their usage in the industry has been constantly growing [11].

1.2 Motivation, methods, and scope of thesis

The trend in design sizes were observed in the Wilson Research Group study in 2014 to follow the Moore's law, as shown in the Figure 4 [12]. The majority of design sizes in 2014 were still on the same scales as in 2007, but the amount of designs on the highest border of the spectrum has been growing at the same time as the border has been moved to cover designs that are more complex.

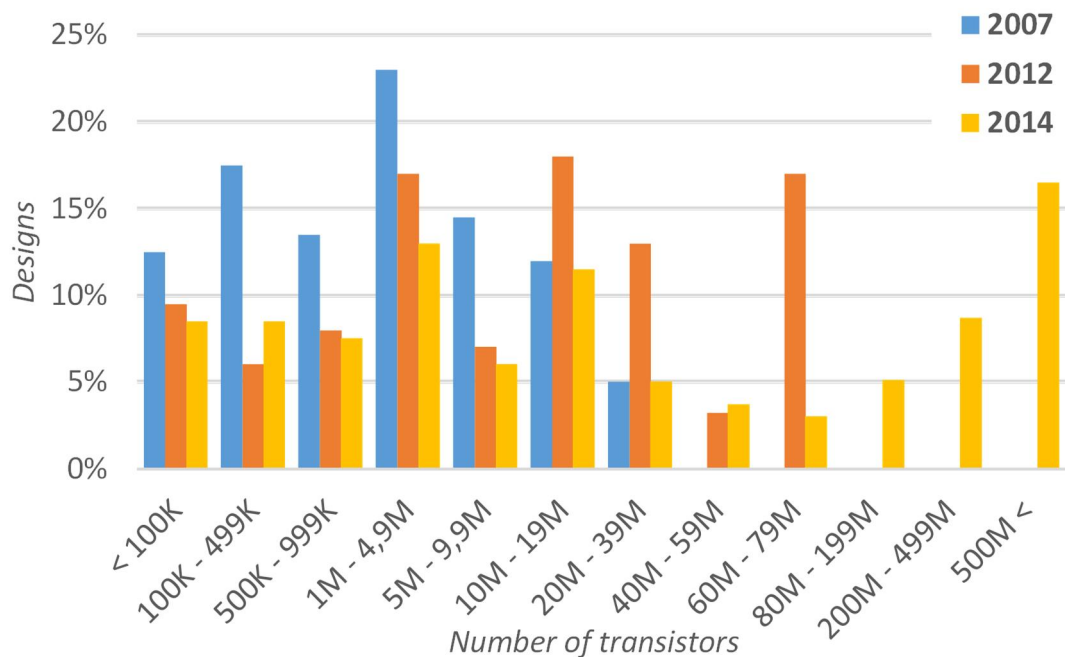


Figure 4. The ASIC design size trends in 2014. [12]

As the design sizes trend to increase, efficient usage of verification methodologies would have a strong impact on verification productivity and shorten the time to market. The integration of verification methodologies as a valuable part of design projects requires education of verification engineers who master the concepts, because the methodologies are complex collections of precisely defined rules and principles that would take a long time to adopt from large reference manuals. An education that follows the standardized guidelines would ensure that the work done by different verification engineers is compatible.

Especially the companies starting to adopt UVM, but also those that have already integrated UVM and orientate new verification engineers, would benefit from a short and effective training. A specifically tailored education would cover all the target areas the company finds the most important and produce quickly new verification engineers that are ready to start implementing verification environments. An efficient education should also include lots of hand-on training that simulates the real verification tasks so the students can apply their knowledge to real-world problems and learn new principles at the same time.

This thesis presents the planning and implementation of an exercise package that can be used in an efficient verification education. The objective is to create an efficient exercise package that introduces the key concepts of SystemVerilog language and Universal Verification Methodology on a compact schedule. After finishing the exercises, a student with no previous verification or design background should understand the reasons for the amount of work done on verification and validation and be able to perform verification work using the current standard language and methodology.

The exercises were made as a part of a specifically tailored digital design and verification education that was ordered by a company. The aim of the education was to introduce employees with a programming background into hardware design and verification concepts and reeducate them to perform verification work using UVM.

The requirements for the verification module made by the customer dictated that the participants should understand the basic principles of verification, master the key mechanisms and syntax of SystemVerilog in the verification perspective and be able to use the UVM class library so that he can produce a working test environment.

The underlying theory of SystemVerilog and UVM is introduced on a level that would be required for understanding the scope of the exercises. The planned contents of each exercise is portrayed precisely, so that this thesis would help new assistants responsible for the same exercises to understand the concepts and the reasons for the used methods. Solutions for the exercises are not provided, so this thesis alone would not be sufficient for orientation material.

The planned exercise projects are tested by implementing them in the classroom as part of the education module. The metered results are the time usage in each exercise and a feedback form that was returned by the students after the education. Another interesting meter would be to follow if the participants in the education were placed later to perform verification tasks in the company and their later opinions on how the education prepared them for the tasks. This could have been monitored with a survey one year later after the education, but such a survey has not been done.

Pedagogy would be an important factor in education design to ensure that the selected teaching methods would best serve the learning process. Pedagogy was left out of the

scope of this thesis, but the planning of teaching methods follows subjective experiences of well-designed exercises during previous study work.

1.3 Outline of thesis

The thesis is divided into seven parts. Chapter 2 introduces the SystemVerilog language considering both the design and verification perspectives of the language. The purpose of the chapter is to provide an overview of the basics of the language and to explain the details of the concepts that are used in the following chapters. The most important concepts that would be emphasized in the exercises are outlined. Chapter 3 outlines the basic principles of Universal Verification Methodology on a level that was needed in this thesis and drafts a plan, how the principles are introduced to the student when planning the exercises. Small examples of advanced concepts are also introduced to widen the perspective and to introduce the possibilities that integrating UVM in verification process offers.

Chapter 4 explains the preliminary requirements for the training and introduces the selected teaching methods. It also provides information of the practical arrangements, such as the development environment and the tools used in the training. Chapters 5 and 6 explain the contents of the exercise instructions. Chapter 5 describes the SystemVerilog exercises and chapter 6 concentrates on the UVM portion of the training. A large number of code examples in these chapters are provided to offer a deeper understanding of the theory in Chapters 2 and 3.

Chapter 7 reviews the results of the training. The completion of the learning objectives was monitored with a time usage survey and a feedback form that are analyzed in the chapter. The chapter also explains the problems that were faced during the exercise sessions and offers improvement suggestions if a similar education is arranged later. Finally, Chapter 8 concludes this thesis.

2. SYSTEMVERILOG

SystemVerilog is a unified hardware description, specification and verification language that was first described in the standard IEEE 1800-2005 and most recently updated in 2012. It started as a set of extensions for the Verilog design language described in standard IEEE 1364-2005, but the two standards were merged into a single language in 2009 [15, 16]. The purpose of the original SystemVerilog extensions was to provide verification engineers with tools on higher abstraction level to improve productivity, readability and reusability. They also provided design specification methods such as new data types, packages, extended port declarations and interfaces [13, 14].

Because the two standard have been merged, this thesis does not differentiate the Verilog language from the SystemVerilog extensions, but describes the design and verification properties of the unified language following the IEEE 1800-2009 standard. Therefore, not all the code examples are fully compatible with the older Verilog standard. There are a few common Verilog design principles that are still used but upon which SystemVerilog has made improvements, so they have been considered to be worth explaining.

Understanding the basic SystemVerilog principles is mandatory for completing the exercises in the education module. This chapter introduces extracts from the language standard that explain the basic concepts that are used in the SystemVerilog and UVM exercises. The most important parts of the basic concepts are selected to form the learning objectives for the education.

2.1 SystemVerilog for design

The focus of the training module is on verification, but before inspecting the high-level verification properties of the SystemVerilog language it is beneficial to know the design-oriented side of the language, because the high-level verification language was originally an extension to the Verilog hardware description language. A verification engineer will also encounter designs to be tested written in SystemVerilog and knowing the design principles of the language will enable him to find and correct the bugs in the design. This section introduces the most basic design concepts of SystemVerilog that would be used in the SystemVerilog exercises with code examples.

The SystemVerilog design hierarchy consists of *modules* that are the basic building blocks [15]. Modules represent design units that have input and output ports and logic combining them. Modules communicate with each other usually through the input and output ports. Program 1 describes the declaration of a simple module that performs a bitwise and operation for signals a and b.

```

module simple_and (input wire a, b, output y);
    assign y = a & b;
endmodule: simple_and

```

Program 1. *Module declaration.*

The module declaration in the Program 1 is started with the module header and ended with keyword `endmodule`. The module header describes the *ports* and gives the module a specific *name* to differentiate it from other modules. The input and output port declarations in the header include the port directions and data types. If the data type is omitted, as is the case for the output signal `y`, it will be implicitly declared by the compiler. An *assign* primitive is used to set the output value and to form the logic between the ports.

The module header in Program 1 where the port directions are listed follows the *ANSI* type declaration of SystemVerilog. The ANSI type declaration is not supported in the Verilog language and the traditional way is to declare only the data types in the header and the directions on the first lines after the declaration. The ANSI method is advised to be used, because it reduces unwanted repetition.

Modules can be instantiated inside each other to create design hierarchy, as shown in Program 2. In the example a top-level module, that has no input or output ports itself, creates an instance of the `simple_and` module that was described in Program 1. Internal variables `in1`, `in2` and `out1` are connected to the ports of the instantiated module.

```

module top; // module with no ports
    logic in1, in2; // variable declarations
    wire out1; // net declaration

    // module instance
    simple_and u1 (
        .a(in1),
        .b(in2),
        .y(out1));
endmodule: top

```

Program 2. *Module instantiation.*

Modules were chosen to be an integral part of the learning objectives, as they are the basis for every SystemVerilog design. The first thing the student should learn in the exercises should be to declare a module using the ANSI type declaration but he should know of the existence of non-ANSI type declaration as well. He should also be able to instantiate modules to create hierarchical designs.

Procedural statements describe behavioral code, where programming statements, for example if-else, case or for loop structures, can be used to describe the functionality. The statements usually contain a *sequential block* delimited by keywords `begin` and `end`, where the statements are executed sequentially in the given order so that all the statements

inside the procedure act syntactically like a single assignment. There are two basic types of procedures: *initial* and *always*. Initial procedures run only once and they are usually used for variable initialization. Always procedures describe combinational or sequential logic that is run triggered by events in the sensitivity list. The sensitivity list is declared using the character @.

SystemVerilog introduces separate *always_comb* and *always_ff* procedures that specifically state the designer intention. The benefit of using these new procedures is that the compiler should generate a warning, if the desired function is not achieved. An *always_comb* procedure has an inferred sensitivity list so that it is run every time any of the signals used in the procedure change. An *always_ff* procedure always generates sequential logic and the sensitivity list should contain clock and reset signals. Simple examples of *always_comb* and *always_ff* procedures are shown in Program 3. The *always_comb* procedure describes a 2-to-1 multiplexer. The output signal *y* is connected to the *a* or *b* input depending on the selection signal *sel*. The *always_ff* procedure is set to run on every rising edge of the clock signal when the asynchronous reset is set high and update the *d* input value to *q* output. On the falling edge of the reset signal, the output is set to 0, where it stays until the reset is set high.

```
// An always_comb procedure describing combinational logic
always_comb begin // procedural block
    if (sel) y = a; // procedural statement
    else y = b;
end

// An always_ff procedure describing a D flip-flop
always_ff @(posedge clock iff reset_n == 1 or negedge reset_n) begin
    q <= reset_n ? d : 0;
end
```

Program 3. Procedure example. [15]

The learning objectives should include procedures and sequential blocks near the beginning of the SystemVerilog exercises. Because most of design work uses synchronous logic, the exercises should concentrate on the *always_ff* procedure, but also introduce initial and combinational procedures.

The data types in SystemVerilog are divided to *nets* and *variables*. Nets represent physical connections and do not store data. Therefore, they cannot be used in procedural statements. The most common net type is *wire*, which is a 4-state type. 4-state types can have values '1', '0', 'X' or 'Z', where 'X' represents an unknown logic value and 'Z' is a high-impedance state. 2-state data types also exist. They can have only values 0 and 1, but they are not synthesizable and are meant to be used in simulation to reduce overhead.

Variables can store data and can be written in procedural statements unlike nets. The most common variable type is *logic*, which is a 4-state type. In Verilog, the basic variable type was *reg*, but it has been renamed to *logic* to avoid confusion, as the name *reg* can be

perceived to imply a hardware register. The reg data type still exists in SystemVerilog, but the use of logic is preferred.

Variables and nets can be declared as *arrays* in either *packed* or *unpacked* structure, as shown in Program 4. A packed array declaration has the dimensions declared before the name. Packed arrays always represent a contiguous set of bits. A one-dimensional packed array is often referred to as vector, in which every element can be conveniently accessed. Multidimensional arrays are also supported. There are data types, for example integer, that are already packed arrays of predefined widths and therefore they cannot be declared as packed arrays.

Unpacked arrays can be used to declare multiple similar signals that are not required to be contiguous, for example if the signals are connected to different components. The declaration has the array dimensions after the name. Any data type can be used as unpacked arrays. The statement on the second line of Program 4 declares an 8-element unpacked array vectors that contains 32-bit vector values.

```
logic [31:0] vector1; // a 32-bit wide unpacked array of type logic
logic [31:0] vectors [7:0]; // packed array of 8 32-bit unpacked arrays
```

Program 4. *Packed and unpacked array declarations.*

The most important data types are logic and wire and their usage should be part of the training. Packed arrays should also be emphasized and only mentioning the existence unpacked arrays would be sufficient. The exercises should concentrate on the current SystemVerilog syntax, so the data type logic is preferred over the older reg type. 2-state data types would require less attention in the SystemVerilog exercises as they are a more abstract concept, but their usage would be introduced in the UVM exercises.

To enhance reusability modules can be *parameterized*. The list of module parameters is declared in the module header between the module name and the port list using the character # as shown in Program 5. The example shows the header of an adder module, in which the widths of the input and output data signals are set using the data_width_g parameter.

```
module adder #(parameter data_width_g = 32)
(
  input          clk, rst_n,
  input          [data_width_g-1:0] a, b,
  output logic  [data_width_g-1:0] sum_out,
  output logic          carry_out
);
```

Program 5. *Parameterized module header.*

Defining array widths is an important design concept in terms of reusability, because then the same component can be used in different systems that incorporate different data

widths. Therefore, parameters should be introduced in the first SystemVerilog design exercises. More advanced parameters would not be needed in the SystemVerilog part of the training.

Interfaces is a concept introduced in SystemVerilog that enhances design reuse by encapsulating the communication between design blocks. An introduction to interfaces was requested by the customer, and their usage would be deeper explained in the UVM exercises.

For example, in a large design where multiple components connect to the same complex data bus, the bus signals can be declared only once in the interface construct and then the modules can be declared to use the interface. An example of a memory interface is shown in Program 6. The interface block describes all the signals and two *modports* for master and slave components. The modports are used to set the signal directions. Interfaces can be parameterized as well.

```
interface mem_bus(input logic clk); // Interface header

    // Signals in the interface
    logic read, write;
    logic [7:0] addr, wdata;
    logic [7:0] rdata;

    // Modports set the signal directions
    modport master (input rdata, output read, write, addr, wdata);
    modport slave (input read, write, addr, wdata, output rdata);

endinterface: simple_bus

module memory(mem_bus.slave bus0);
    ...
endmodule

// Connect the modules on top level
module top;
    mem_bus bus0();
    memory 12(.bus0(bus0));
endmodule
```

Program 6. *Interface example.*

Assertions were an important part of the learning expectations for the SystemVerilog exercises and they should require a separate exercise. They ensure the behavior of the system and are used to validate the design. SystemVerilog has an advanced assertion language built in to allow complex assertions that could monitor the execution during multiple clock cycles. Assertions can be declared using keywords *assert* and *assume*. Assert assertions specify obligations for the design that must always hold and assume assertions are used to specify assumptions for the environment, for example the format of the input data that formal verification tools could use to generate test input. Assertions can also be used to provide functional coverage data.

SystemVerilog assertions can be either *immediate* or *concurrent*. Immediate assertions are checked every time the statement line is run following the simulation event semantics. They are intended to be used in simulation. Concurrent assertions are always active based on clock semantics and use sampled variable values. They can be used to describe behavior that spans over time, for example to check signal values during multiple clock cycles. The concurrent assertions can also be used in formal verification tools in addition to event-based simulation.

Program 7 introduces an example of an immediate assertion that can be used in simulation. The assertion checks on every rising edge of the clock that logical or operation of signals req1 and req2 is always true. If the assertion fails, the simulator will print an error message including the current simulation time, as specified in the else branch. If the else condition is not specified, the default procedure is to use an error print.

```
time t;

always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2)
    else begin
      t = $time;
      #5 $error("assert failed at time %0t", t);
    end
```

Program 7. *An immediate assertion. [15]*

Program 8 shows an example of concurrent assertions. This assertion ensures that every time the enable_in signal has a rising edge, the signal valid has to be set to 1 after two clock cycles. The assertion is tied to the rising edge of the clock signal, but disabled if the system is in reset. Concurrent assertion can be very complex and used to validate correct signal behavior over long time.

```
assert property (@(posedge clk) disable iff(!rst_n)
  $rose(enable_in) |-> ##2 valid == 1);
```

Program 8. *A concurrent assertion.*

2.2 SystemVerilog for verification

A significant difference between Verilog and SystemVerilog is the support for object-oriented programming for verification purposes on a higher abstraction level. The vast language reference is not completely explained in this section, but only the basic object-oriented properties that are needed for understanding the structure of UVM. The learning objectives for the exercises required that the object-oriented properties should be covered in the exercises and this section provides an overview of the required concepts. The concepts would be introduced to the student in UVM exercises.

The object-oriented properties in SystemVerilog follow object-oriented programming guidelines very closely, so the structure is familiar for designers who have programming experience in C++, Java or other object-oriented languages [19]. The main benefit in object-oriented testbench design is that the designer can declare complex data types and combine them with routines that use the data. Instead of toggling bits in the DUT directly, those routines can be used to perform even complex transactions without considering the state of every bit on every clock cycle.

The basic building block for a high-level testbench in SystemVerilog is *class*. The class encapsulates *data* and *routines* together in a single block of code. A class declaration for a simple transaction packet is shown in Program 9. The class BusTran contains variables for an address vector and a data array and it has two routines: one that prints the address in the packet and one that calculates a cyclic redundancy check of the data to the variable *crc*.

```
class BusTran;
  bit [31:0] addr, crc, data[8];

  function void display;
    $display("BusTran: %h", addr);
  endfunction : display

  function void calc_crc;
    crc = addr ^ data.xor;
  endfunction : calc_crc

endclass : BusTran
```

Program 9. *A simple class declaration. [19]*

Routines can be either *tasks* or *functions* [15]. The example in Program 9 uses functions with the return type of *void*. The difference between tasks and functions is that the functions can have input and output values and are processed without blocking the simulation time – in the simulation perspective they return their value immediately. Tasks do not return a value, but can block the simulation time during execution, so tasks have a concept of time. Tasks can incorporate delays to bind the processing to a certain moment of time, a signal value or to another event.

Classes are instantiated as *objects*. An object has a type and a name and the instantiation is done by first creating a variable of the type of the class to hold an object *handle* and after that the object is created and assigned to the variable using the function *new* that is called a *constructor*. An example instantiation is shown in Program 10. The object creation in SystemVerilog reminds of C++ or Java, but the memory allocation and deallocation of C++ is not needed. Object construction is simple and garbage collection is performed automatically.

```

class Packet;
  integer command;
  function new();
    command = IDLE;
  endfunction
endclass

...
Packet p; // declare a variable of class Packet
p = new;  // initialize variable to a new allocated object
         // of the class Packet

```

Program 10. *Class instantiation.* [15]

New classes can be derived from base classes using the *extends* keyword [15]. The declared properties and methods in the base class can be accessed using the keyword *super*, or overridden by declaring new methods. Program 11 shows an example, where the class `LinkedPacket` is derived from the `Packet` class described in Program 10. The `LinkedPacket` class is a special form of the class `Packet`, which introduces a new method `get_next` that returns an object of the type `LinkedPacket`. As the `LinkedPacket` is extended from the `Packet` class, every `LinkedPacket` object is a legal `Packet` object. A `LinkedPacket` object handle can be assigned to a variable of type `Packet`.

```

class LinkedPacket extends Packet;

  LinkedPacket next;

  function LinkedPacket get_next();
    get_next = next;
  endfunction
endclass

```

Program 11. *Class inheritance.* [15]

Class declarations in SystemVerilog, including routines and inheritance, will be introduced to the student in the UVM exercises and therefore they will not require a separate exercise in the SystemVerilog part of the training. The syntax of SystemVerilog for verification is not emphasized as the motive for any UVM exercises, but it will still be listed as one of the important learning objectives that the student will learn on the side when he is getting familiar with UVM.

SystemVerilog has also introduced new block statements in addition to the sequential blocks that are delimited by the `begin` and `end` keywords [15]. There is also a *parallel block* that is delimited by keywords `fork` and `join`. All the statements in the block are processed concurrently, so the code in Program 12, that sets the value of the variable `r` after certain clock cycle delays, finishes 200 clock cycles after entering the block. A similar sequential block would process the assignments one at the time, so the processing time would be 500 clock cycles when all the delays are added together. If the execution does not have to wait for all the forked statements to finish, additional `join` keywords

`join_none` and `join_any` can be used to change the behavior. As functions do not have a concept of time, the usage of parallel blocks in the functions is limited.

```
fork
  #50 r = 'h35;
  #100 r = 'hE2;
  #150 r = 'h00;
  #200 r = 'hF7;
join
```

Program 12. *An example of a parallel block [15].*

Parallel blocks are not important for the education because they would not be required for the designs in the exercises. Therefore, they will be omitted from the exercise instructions. Using parallel blocks in a UVM testbench would allow more complex testbench designs, but finding them out is left for the student himself. If the student declares a need for parallel processing in his testbench, the assistant can hint the usage during exercise sessions.

3. UNIVERSAL VERIFICATION METHODOLOGY (UVM)

Unified Verification Methodology (UVM) is a class library written in SystemVerilog. It is developed and standardized by Accellera, which is a consortium of EDA tool vendors and users including Synopsys, Mentor Graphics, Cadence, AMD, Intel, ARM and other companies [2]. Accellera released a press release in July 2015 where it was announced that UVM will be submitted as IEEE standard IEEE 1800.2 and the work is currently in progress [1].

The purpose of UVM was to combine the principles of several verification methodologies, mainly OVM and VMM, into a single standard methodology to be used across the field. The Accellera verification IP technical subcommittee declared in a release on January 2010 that the methodology will “*enable users to deploy an efficient, reusable, and interoperable SystemVerilog verification environment.*” [6]

Mentor Graphics commissioned the Wilson Research Group Functional Verification Study in 2016 on the state and trends of verification on ASIC/IC market [11]. According to Harry Foster from Mentor Graphics, the studies performed every second year were “*world-wide, double-blind, functional verification studies, covering all electronic industry market segments. To our knowledge, the 2014 and 2016 studies are two of the largest functional verification study ever conducted.*”

The adoption trends found in the study for various ASIC/IC testbench methodologies built using class libraries are shown in Figure 5. Based on the study, UVM is already widely used in the industry and its popularity has been rising at the cost of older methodologies every time the study has been conducted. According to the results, over 70% of the participants already used UVM and the trend was that in 2017 the number would be higher. It should be noted that selection of multiple verification methodologies was possible in the study, so companies that have only started adopting UVM but mainly use other methodologies might be visible in the numbers. [11]

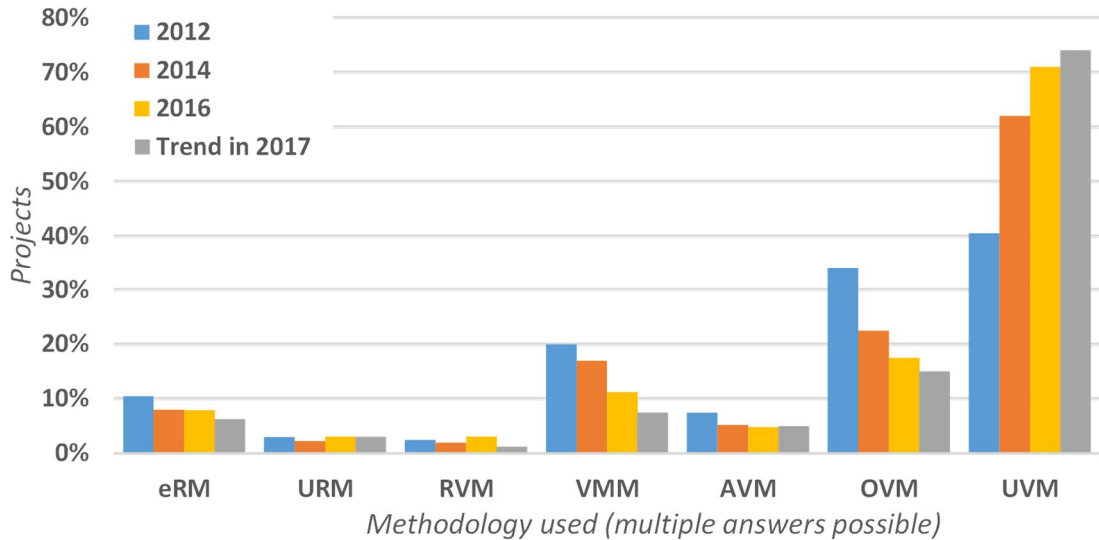


Figure 5. ASIC/IC testbench methodology adoption trends. [11]

This chapter provides an overview of the UVM concepts that have been described in the learning objectives for the exercises. The concepts are analyzed so that the most important content forms the basis for the training material and less relevant information can either be omitted, mentioned as a curiosity or used in more advanced exercises.

The code examples given are very abstract and the more precise usage of the UVM macros, methods and components is described in Chapter 6, which covers the contents of the UVM exercises. The first five exercises in Chapter 6 contain the steps needed for building a complete UVM testbench with code examples.

3.1 UVM building blocks

The structure of a UVM testbench consists of small and relatively simple components that are built hierarchically. The tests that contain information of how the DUT should be tested is separated from the environment that describes how the DUT should be connected to the testbench.

The benefit of dividing the testbench into small components is simplifying the testbench design and reuse. Reuse can be horizontal and vertical, so in addition to reusing components in new testbenches, the verification environments for single blocks can be integrated together to form subsystems that can be integrated further to implement system level testing.

Since the UVM testbench is built from dynamic objects, that do not exist in memory before they are created, a static component is needed for launching the simulation [5]. The static component in UVM is a top level SystemVerilog module that includes pin connections to the DUT and starts the test, which then configures the environment and runs a sequence of transactions to the DUT.

The student should be instructed to declare his own components in the UVM exercises and he should gain an understanding of how to reuse his own components. Both the horizontal and vertical reuse should be explained in advanced tasks. The existence and function of the top-level module should be explained, but writing such a module would not be the most important part in UVM to be mastered by the student.

The UVM testbench file hierarchy uses SystemVerilog *packages*. Packages are constructs that combine related declarations and definitions together in a common namespace that is a single compilation unit for the simulator. To access the namespace and the underlying definitions the package must be imported. The usage of packages allows the testbench developer to organize the code and ensure consistent references to types and classes.

A package file should contain all the related class declaration files [5]. For a simple UVM testbench a single package could contain all the definitions, but in a large system level testbench the declarations could be divided between multiple packages so that there is a separate package for every bus interface and a number of packages for different types of test sequences that contain all the declarations for running different tests. Instead of declaring all the classes directly in the package file, the coding guidelines by Mentor Graphics state that every class declaration should be in a separate file and all the declaration files are included in the package using a SystemVerilog include directive. The include directive instructs the compiler to insert the entire contents of a source file inside another file in place of the directive. The package should only contain the include directives for class declaration files.

The approved use of include macros and import directives should be explained to the student to ensure that the focus in reusability is emphasized from beginning. The testbenches in the exercises would be simple so that the whole hierarchy can be declared in one package, but in a more advanced additional exercise, multiple sublevel packages could be introduced.

3.1.1 Objects and components

Object is the basic building block in a UVM testbench and all the objects are extended from the `uvm_object` base class [3]. The primary role of the `uvm_object` base class is to define the common methods for basic operations, for example `create` and `print`, that are used for every object. It also defines instance identification interfaces, for example `name` and `unique id`. The most basic objects are data packages sent to the DUT that are instantiated as sequences of packages to generate test input.

A hierarchy tree of the most basic UVM classes is introduced in the Figure 6. The base class `uvm_object` is supplemented with a reporting interface to form a `uvm_report_object` class. The class `uvm_report_object` is extended further into `uvm_component` class that introduces a concept of hierarchy and properties needed for creation and connection of

components. The `uvm_component` class is further extended to the base classes of the components that are implementable by the user as shown in the figure. On the other branch `uvm_object` is extended into transactions and sequence items that form a sequence of items that are delivered to the DUT. The highlighted classes can be created by the user.

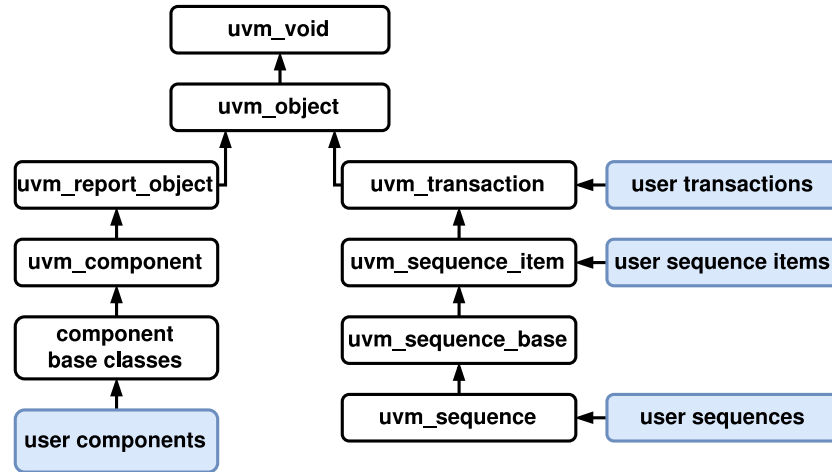


Figure 6. UVM base class hierarchy. [3]

All the component base classes are derived from the `uvm_component` that contains all the common properties making the base classes of the separate components relatively simple [3]. A hierarchy tree of the common UVM components is shown in Figure 7. Deriving every component from the matching base class allows the testbench designer to distinguish components from each other and ensures that the components will benefit from all the features built into the base class. Some component base classes, for example `uvm_monitor`, are just empty shells that do not add any additional features to those derived from the `uvm_component`, but functionality may be added in the future UVM versions.

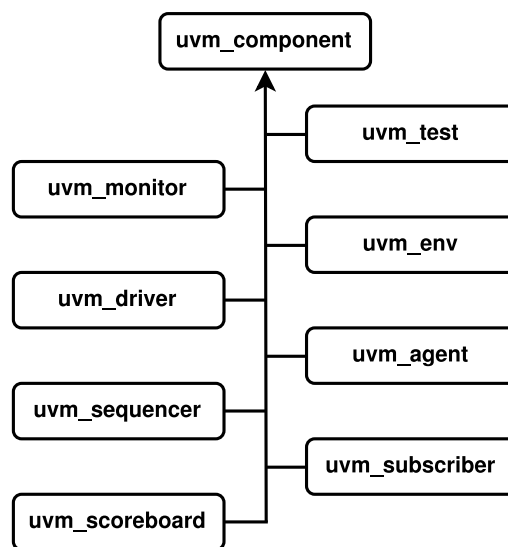


Figure 7. UVM component class hierarchy. [3]

The complicated class inheritance system allows the user to write simple implementations of components, because the core functionality is built into the base class. The user can extend his own class implementations as well enhancing reusability. When the user-defined components are simple, they require less rewriting when used in other testbenches. The selection of correct base classes for the student's own objects and components should be emphasized in the exercises. After the training, the student should be familiar with the class inheritance hierarchy of the most common classes in UVM so that he would know the difference between the object and component base classes. He would also see in practice in advanced tasks, what are the changes that are mandatory for the components when a UVM testbench is converted for testing a different design.

The components can communicate with each other by delivering transaction level modeling (TLM) transaction objects or by reading and writing the UVM configuration system. The TLM transactions are delivered via channels between ports and exports that are introduced in components and then connected to each other. A port is instantiated in components that initiate transaction requests. The ports are connected to implementations in components that implement the initiated methods. Exports are channel items that forward an implementation to be connected by the port.

Figure 8 describes a block diagram in which the consumer instantiated inside the component 1 wishes to get data from the producer that is instantiated in subcomponent inside component 2. To create the connection, the implementation of the producer is connected to the port of the consumer via ports and exports on higher levels. When the consumer calls the get procedure, the data flows in the direction of the arrows. The implementation is marked with a square (\square) in the picture. The diamonds (\diamond) are ports and the circles (\circ) are the exports.

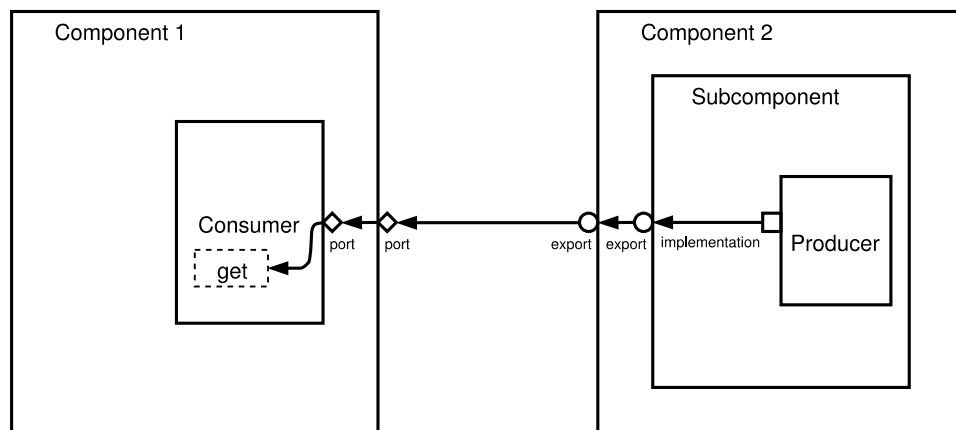


Figure 8. Port – export – implementation connection. [3]

The usage of ports and exports should be explained to the student. He would see in practice how the initiator calls the transfer functions of the implemented port and how the functions are declared in the components including the export. He would also declare ports and exports himself.

3.1.2 Macros and methods

UVM includes *macros* that provide users a shorthand notation for SystemVerilog constructs. The macros can define object behavior and interaction with the internal UVM mechanisms and assist in reporting. Macros are defined beginning with the grave accent character (`), which should not be confused with the apostrophe (').

All the reporting in UVM should be done using *reporting macros*, because they handle the filtering of unneeded messages to reduce processing overhead [3]. The reporting macros also automatically provide file names and line numbers in the prints done by the testbench and ensures that the user does not accidentally prevent printing of warning and error messages by setting a verbose level. Examples of reporting macros are shown in Program 13. The parameters for the macros are the message identification, INFO1 and WARN1 in the program, and the message to be printed [5]. The `uvm_info` macro can also include a level of verbosity for filtering of the messages. A `sformatf` method is used to format the info message using the syntax similar to `printf` function in the C language. There are also similar macros for errors and fatal errors.

```
`uvm_info("INFO1", $sformatf("data: %0d", data), UVM_LOW)
`uvm_warning("WARN1", "This is a warning")
```

Program 13. *Examples of UVM reporting macros.*

Other important macros are the *factory registration macros* [5]. The *factory* is a class internal to the UVM mechanisms, which takes care of creating UVM objects and components and maintains a list of every instantiation done in the testbench. All the objects and components should be registered to the factory by performing a factory registration macro:

```
`uvm_component_utils(my_class)
```

There are separate registration macros for objects and components. The purpose of the registration macro is to help the factory to keep a record of every object and component in the testbench. The classes can be later substituted with another compatible class by using the factory without changing the underlying component hierarchy code. In addition to the registration macro, the class instantiation should be done using a special factory method instead of calling the constructor function directly. The factory method will call the constructor function of the classes, but also performs additional procedures that are mandatory for the function of the UVM factory. The syntax for the factory method for instantiating an imaginary class `comp` is following:

```
comp_h = comp::type_id::create("comp_h", this);
```

The factory registration macro should be introduced to the student when he is instructed to declare his first class to ensure the correct structure of the testbench from the beginning.

Reporting macros should be introduced so that the `uvm_info` would be explained first and the knowledge would be deepened later by providing the more severe alternatives.

Another important internal mechanism of UVM is the *configuration database*. The configuration database stores variables to be read in the components to allow communication across the testbench during runtime. In addition to the variable name and value, a scope is set that dictates the hierarchical path to the component using the value. The configuration database can be written and read by every component by using functions `set` and `get`. Usage of the configuration database enhances efficient reuse by making the components in the testbench more configurable.

The configuration database would be used in the exercises to deliver a pointer to the DUT interface from the top-level module to the testbench components that communicate with the DUT. More advanced exercises could also mention other configuration for the testbench, but the delivering of the virtual interface would be enough to show the function of the database.

The simulation of UVM is divided into *phases*. The sequence of the UVM phases is shown in Figure 9. There are 21 simulation phases in total and they can be divided into three categories. In the beginning of the simulation the build time phases construct the test environment by building components using the factory, form the connections between the TLM channels and configure all the components using the configuration database. The build time phases do not consume simulation time.

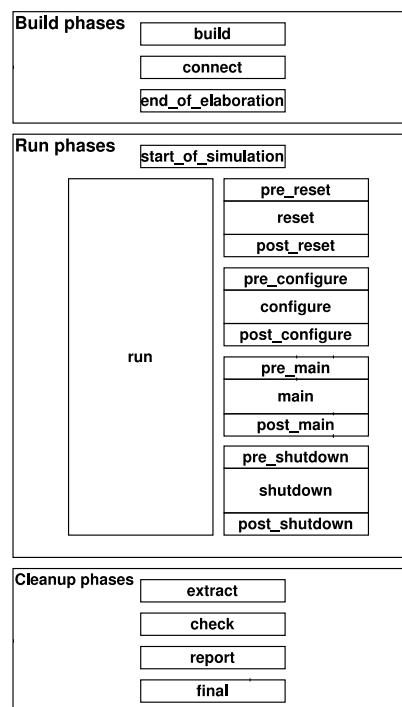


Figure 9. UVM Phases. [5]

After the test environment has been constructed, the run-time phases are started and the simulation time is consumed. The run-time phases carry out the actual simulation where the test case is run for the DUT. After the test has been stopped, simulation time is not consumed anymore and cleanup phases collect the results of the test case and report them.

The functionality of components in every simulation phase is configured by providing specific phase methods in the class declarations. Not all the 21 simulation phase methods have to be declared in every class declaration, but only the methods where the component should have user-specified activity. The high number of phases allows a common understanding on what should happen in each phase of the simulation when verifying complex designs, even when the components are developed by different engineers.

An example of the usual phase methods for an imaginary class is shown in Program 14. The build phase method creates components that are lower in the hierarchy by using the factory instantiation method. The connect phase function follows the build phase and performs the connections between the components created in the build phase. On run phase, an imaginary data object is created and commanded to start execution. The run phase consumes time during simulation, so the type of the phase method is task. All the phase tasks have to use specified method names and be parameterized by the UVM phase as in the example. [5]

```
function void build_phase(uvm_phase phase);
    // create two components of type comp
    comp1_h = comp::type_id::create("comp1_h", this);
    comp2_h = comp::type_id::create("comp2_h", this);
endfunction: build_phase

function void connect_phase(uvm_phase phase);
    // Call the connect method to connect the implementation to export
    Comp1_h.conn_imp.connect(comp2_h.conn_exp);
endfunction: build_phase

task run_phase(uvm_phase phase);
    ...
    // create a object of type obj and call its start method
    obj_h = obj_type::type_id::create("obj_h");
    obj_h.start( ... );
    ...
endtask: run_phase
```

Program 14. Examples of phase methods.

The exercises should focus on the most important build, connect and run phases, because these phases implement the basic methods of UVM. The instructions could mention that there are more phases as well, but the testbenches to be designed in the exercises would not require declaring methods for them.

3.2 UVM architecture

The architecture of a UVM testbench consists of user-defined components extending base classes in the UVM library [5]. Every component has a name and a parent in the testbench hierarchy and they are instantiated in the build phase by using the UVM factory in a top to bottom order. An example of a block level UVM testbench is introduced in the Figure 10. The example in the figure contains an UVM environment that is used in multiple tests. The example environment contains agents for interfacing the two bus interfaces and components for test coverage monitoring and the functional checking of the DUT.

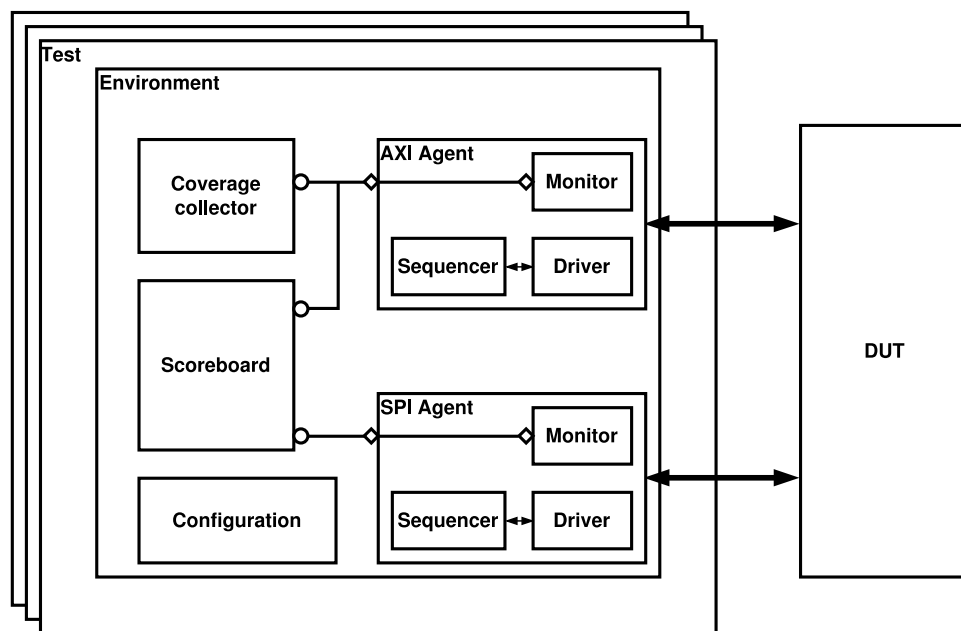


Figure 10. A complete block level testbench. [5]

The UVM architecture will be an important learning objective in the exercises. The student should be able to distinguish between the test and the environment. He should know the most common UVM components on a level that he should be able to declare a simple UVM testbench himself using a correct hierarchy.

3.2.1 UVM Environment

The *environment* is the component that describes the physical architecture of the testbench [4]. It instantiates all the components hierarchically and one environment can contain multiple sublevel environments. On system level testbenches there could be one top-level environment that instantiates multiple environments for each block of the design. The same block level environments could be used for block level testing before integration. The method of building a system level testbench from block level environments is often referred to as *vertical reuse*.

The environment contains one or more *agents*, which communicate with the DUT, and *subscribers*, which use the data provided by the agents. An agent is a component that instantiates the components that manage the stimulus flow, feed the input data to the DUT and monitor the signals that move between the testbench and the DUT. If the DUT communicates with multiple interfaces, there is usually one agent per interface. An agent could have an active or passive role set by the configuration database: an active agent provides stimulation data for the DUT and a passive one only monitors the transfers.

The first testbench to be designed by the student would be simple and contain only one environment that has one agent. The passive and active roles of the agent could be introduced, but not emphasized in the beginning. More advanced tasks could introduce the usage of multiple agents. The student should know after the exercises what the role of the agent is and what components it includes.

The usual components instantiated within the agent are *sequencer*, *driver* and *monitor*. The *sequencer* is an arbiter that reads sequence objects from a list and controls the sequence flow. The sequences are delivered as TLM transactions. The *driver* receives the TLM transactions from the sequencer and drives them to the DUT. Thus, the driver transforms the abstract transaction level sequences into pin-level activity in the DUT.

Monitor is a component that follows the activity in the DUT interface and samples it. The pin-level activity in the DUT is converted into TLM transactions and sent out for analysis. The monitor includes an analysis port that delivers TLM data further into the testbench. In a passive agent, the sequencer and the driver are turned off and only the monitor is active.

The data sent out by the monitor is analyzed by the subscribers that implement analysis exports, which connect to the analysis port in the monitor. The subscribers usually reside outside the agent in the environment, but can be instantiated in the agent as well in more complex designs. The subscribers answer to the questions “How does the DUT perform?” and “How much have we tested so far?” [4]

Coverage collector is a subscriber that gathers functional coverage data [5]. It samples all the transactions sent by the monitor and uses the data to increment counters in *covergroups*. Covergroups specify the signals and conditions that are to be monitored as *coverpoints*. The counter values for each coverpoint represent real-time functional coverage data of situations that have and have not been tested.

Scoreboard is the UVM component that determines if the DUT functions properly. It specifies a reference model and compares the output produced by the DUT to the reference. In simple designs the reference the model and comparator can be declared in a single component, but it is also possible to use separate components or even use an outside model of the DUT as the reference. [5]

The exercises should include a coverage collector and a scoreboard, because they are components that use the data provided by the monitor and implement the testbench functionality that answers the questions “Have we tested enough?” and “Does the DUT perform correctly?” The student should examine test coverage data provided by the coverage collector he has declared and verify the functionality of the DUT by comparing its output to a functional model.

3.2.2 UVM Tests

Test is the top-level component of a UVM testbench [4]. The test controls the building and configuration of the test environment, selects the stimulus sequence to be used in the test and controls the simulation process. There can be multiple tests using the same environment but different sequences and configuration. It is common to declare a base test class that instantiates the environment and does the necessary configuration, and then extend tests from it to cover different test cases for the DUT.

Sequences are lists of objects that are delivered to the sequencer in the agent [5]. The sequencer processes the list item by item. The sequences can be layered on top of each other to provide means of describing the complex transactions that include multiple layers such as USB 3.0 or PCI express. A higher-level sequence can control the transactions on a higher abstraction level and command the lower level sequences that work closer to the hardware.

Variables in sequences can set to be *randomized* to allow randomized testing. *Constraints* can be set to limit the randomization to include a specific value range or to set distributions so that a signal can be for example set high 95% of the time, but low for the rest. The concept is used in constraint random testing that was part of the learning requirements.

The simulation run is controlled by an *objection* mechanism [4]. In the start of the run phase the test raises objection and the simulation runs until all the objections have been dropped. This way a component can inform the testbench that it is not ready yet for stopping the simulation by raising another objection.

If there are multiple tests, the top-level module of the testbench specifies the test to be run by declaring the test name as a parameter for the `run_test` method:

```
run_test ("test_base");
```

The test name can also be given as a parameter to the simulator, if the parameter is not provided for the `run_test` method. In terms of reusability, the better way to start the specific test would be to omit the test name from the top-level module and declare it when starting the simulation using the `UVM_TESTNAME` flag. That would allow the user the run different tests without modifying the code.

A base test and sequence should be provided ready for the student, so the first exercises could concentrate on the UVM environment. Later exercises, when the environment is ready, should include multiple tests that use a constraint-randomized sequence extended by the student from the provided classes. The student should also encounter the usage of objections. Layered sequences are more advanced UVM concepts and they will not be introduced in the exercises.

The original learning requirements made by the customer also dictated that the UVM *register abstraction layer* (RAL) should be introduced. The RAL provides a way of controlling the contents of the registers in the DUT and introduces a convenience layer to the register and memory locations. As it was later agreed in a meeting that the RAL should only be covered on a lecture basis, the deeper function of the RAL is not covered in this thesis.

4. REQUIREMENTS AND METHODS FOR THE TRAINING

The verification training module was split into ten days, of which five were lecture days and five were used in exercises. Two of the exercise days were for SystemVerilog and three for UVM. The schedule for the verification education module is shown in Table 1. The first four days of the module were lectures, after which there was a second part of six days with mostly exercises and one lecture day in between about UVM. All the lectures were held at the customer's premises and the exercise days were in the computer classroom TC221 at Tampere University of Technology.

Table 1. The schedule of the verification module.

Day	Theme
15.9.2015	Lecture: Verification - Principles & methodologies (1)
16.9.2015	Lecture: Verification - Principles & methodologies (2)
28.9.2015	Lecture: Verification – Systemverilog (1)
29.9.2015	Lecture: Verification – Systemverilog (2)
5.10.2015	Exercises: Verification – Systemverilog (1)
6.10.2015	Exercises: Verification – Systemverilog (2)
22.10.2015	Lecture: Verification - UVM
23.10.2015	Exercises: Verification – UVM (1)
29.10.2015	Exercises: Verification – UVM (2)
30.10.2015	Exercises: Verification – UVM (3)

4.1 Requirements and student background

According to the requirements for the module, the student should know the following concepts:

1. DUT
2. Testbench
3. Functional simulation
4. Coverage
5. Coverage driven verification
6. Directed test
7. Constraint random test
8. Assertions
9. Assertion based formal verification

He will also master the key mechanisms and syntax of the SystemVerilog language in the verification perspective and the UVM class library so that he can produce a working test

environment using UVM. The requirements covered both the lectures and exercises. The requirements for the content of the exercises were redefined in meetings with the customer and the final learning objectives are explained in the chapters 5 and 6 that cover the planning of the exercises.

The participants in the education were professionals in information technology and telecommunications with experience of programming and software development. Some had former hardware design experience as well and some participated in the digital design education module. It was dictated by the customer that because verification is programming by nature, the module should be introduced to the students so that they were essentially learning a new programming language.

4.2 Tools used in the exercises

The exercises were designed to be run using the Mentor Graphics Modelsim simulation tool. Modelsim was chosen, because it is created by a major vendor and therefore a common tool in the industry. Modelsim also has the UVM library included, so there was no need to compile it. Mentor Graphics has also a more advanced QuestaSim verification tool available, but it was decided that it would offer no real benefit in this case and the essential usage of Modelsim is completely similar.

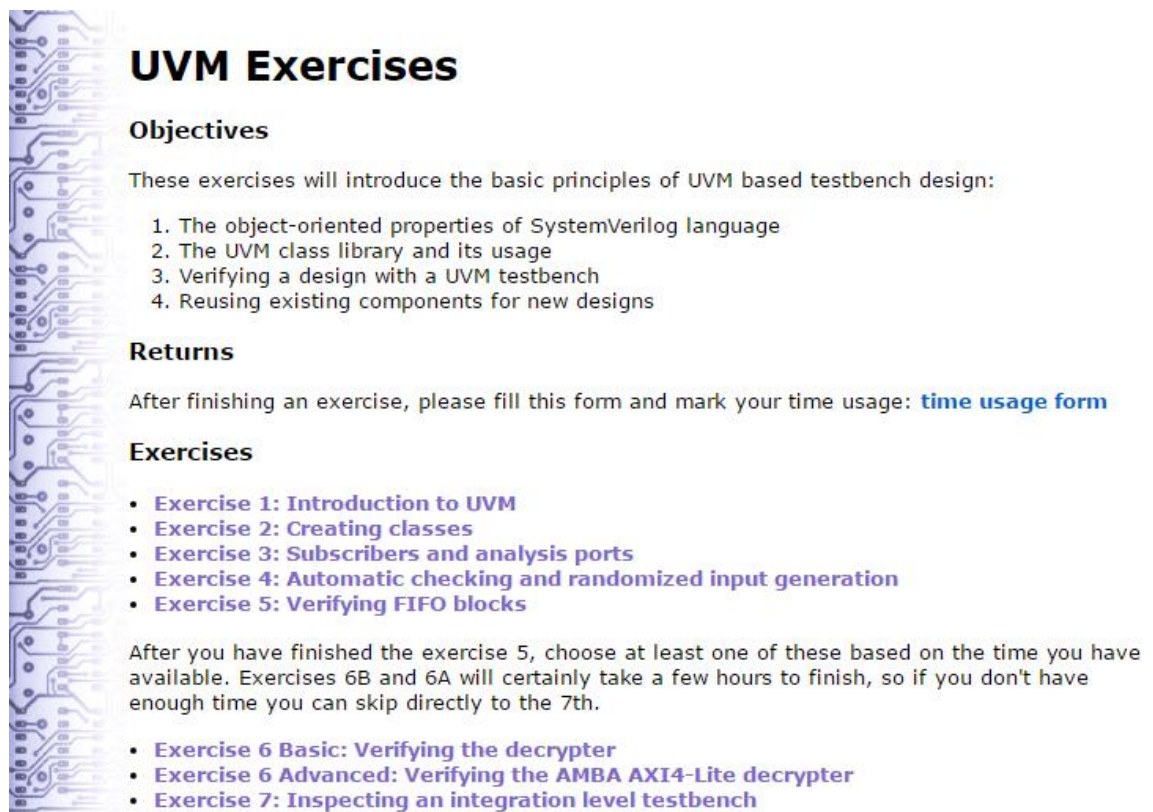
Instead of running the Modelsim directly on the class workstations running Windows 7, a Linux virtual machine was decided as a platform. The Linux environment allows the use of make automation tool that eases the compiling and running of the simulations because a multitude of parameters are needed for UVM simulations. Using a virtual machine also allows a diverse set of text editors – the students had a choice of common tools from both the Windows and Linux environments including Notepad++, Emacs, Gedit and vi.

The Linux Modelsim packages already available on the department network drive limited the selection of the operating system into Red Hat Enterprise Linux or its derivatives. Centos Linux 6 was selected, because it is a free community-driven alternative to Red Hat Enterprise Linux and the installation packages were compatible with it. There was also a newer Centos 7 version available, but the more traditional Gnome 2 user interface of Centos 6 was more appropriate for usage in class compared to the Gnome 3 in CentOS 7, and it was already proven to work on other courses on the department in which a virtual machine has been utilized.

The students had no access to any network drives because of their temporary guest accounts. Therefore, every student was given a USB memory for storing his or her data between exercise sessions. The virtual machine was reset every night to ensure similar experience between workstations and the students were advised to either use a temporary folder on the Windows host machine that would be shared for the virtual machine, or alternatively access the USB memory directly from the virtual machine for data storage.

4.3 Structure of exercises

The exercises were designed to be completed independently without a need for guided sessions. The instructions were planned to be delivered to students via the department website. The instructions for every exercise were written as a separate web page with links to them on the exercise index page. The index page would also offer an overview of the learning objectives and the UVM exercise index gave a small overview of the exercise project as a whole. An example of UVM index page is shown in Figure 11. The SystemVerilog and UVM exercises had both their own websites. Every exercise had more precise learning objectives listed in the same format as in the index page.



UVM Exercises

Objectives

These exercises will introduce the basic principles of UVM based testbench design:

1. The object-oriented properties of SystemVerilog language
2. The UVM class library and its usage
3. Verifying a design with a UVM testbench
4. Reusing existing components for new designs

Returns

After finishing an exercise, please fill this form and mark your time usage: [time usage form](#)

Exercises

- [Exercise 1: Introduction to UVM](#)
- [Exercise 2: Creating classes](#)
- [Exercise 3: Subscribers and analysis ports](#)
- [Exercise 4: Automatic checking and randomized input generation](#)
- [Exercise 5: Verifying FIFO blocks](#)

After you have finished the exercise 5, choose at least one of these based on the time you have available. Exercises 6B and 6A will certainly take a few hours to finish, so if you don't have enough time you can skip directly to the 7th.

- [Exercise 6 Basic: Verifying the decrypter](#)
- [Exercise 6 Advanced: Verifying the AMBA AXI4-Lite decrypter](#)
- [Exercise 7: Inspecting an integration level testbench](#)

Figure 11. The UVM exercise index page.

An example of an exercise page is shown in Figure 12. The exercise layout was designed so that all the real tasks to do would be marked with bullets (1. in the figure) to keep the instructions straightforward. Between the bulleted lists would be body text (2. in the figure) that explains what has been done and what will be done next for motivation. In the first exercises, the bullets were very thorough and the student was guided systematically, but the instructions loosened up along the exercises leaving more processing to the student.

Adding connectivity to the monitor

The monitor's job is to listen to the DUT interface and place the data in transaction objects for other components. To implement the functionality we have to add some connectivity first. **2.**

- Start by adding variable handles for DUT configuration and DUT interface **1.**

```
dut_config dut_cfg; 3.
```

```
virtual dut_if dut_vi;
```

- The conventional place to declare handles, local variables etc. is after the registration macro and before the constructor of the class
- A virtual interface (keyword **virtual** here) is a variable that contains a reference to an existing interface instance.


Next, we'll need some sort of way to know the DUT's interface. If you read the source code comments thoroughly in the last exercise, you know that the way the interface is given to objects is by using UVM's configuration database. **2.**

The interface (.if file) for the DUT is wrapped inside a configuration object (class dut_config) and given to the configuration database by the test.

Figure 12. An example of an exercise page. 1 describes a bullet with the concrete task and 2 is body text containing the motivation for the task. 3 is an example of a code block.

All the console commands and code blocks were highlighted with monospace font in a colored text box, as shown in Figure 12 (item 3.) A dollar sign (\$) was used as the first character to separate console commands from code examples. Some console commands were long and would not fit on one line, so a backslash (\) was used to escape the line break. Most of the code blocks given were ready to be copied and pasted to the text editor and they were commented when needed using the SystemVerilog syntax. Sometimes the code blocks gave only a syntax example that should be applied by the student to keep the students alert, but these cases should be obvious enough to not cause confusion.

The students were given additional information that would not be crucial for completing the exercises, but usually offered hints about the task, deeper explanation of methods used in tasks or some syntax help. The information could also be obtained elsewhere, for example from language reference manuals or design specifications. These information boxes were separated from the rest of the text with a colored background and rounded borders. An example of an information block about reporting macros is shown in Figure 13.



Error reporting

In addition to ``uvm_info` macro there are three more severe printing macros available:

- ``uvm_warning("Source", "Message")`
- ``uvm_error("Source", "Message")`
 - Displays an error, but doesn't end the simulation
- ``uvm_fatal("Source", "Message")`
 - Fatal error, end the simulation immediately

Note that these macros don't take a verbosity level toggle, but they are always printed by default (UVM_NONE).

Figure 13. *An information block example.*

5. SYSTEMVERILOG EXERCISES

The SystemVerilog exercises were an introduction to a new language before the student started to learn UVM. Because SystemVerilog is a vast collection of design and verification toolsets and most of the verification side of the language is utilized in UVM, it was decided that these exercises should concentrate on the design side of SystemVerilog. The student was encouraged to use validation and verification methods usable by hardware designers and was introduced to low level testbenches.

5.1 Learning objectives

The preliminary learning requirements made by the customer dictated that the student should know all the key mechanisms and syntax of SystemVerilog in the perspective of verification. This should have included syntax and basics, object-oriented programming with SystemVerilog, testbenches with SystemVerilog, constraints random stimulus and assertions. In a meeting with the customer it was agreed, that the object-oriented properties and constraints random stimulus will be introduced in the UVM exercises well enough and if the exercises would introduce an object-oriented testbench, it would be too similar to UVM already offering no learning benefits as such. It was decided that the SystemVerilog exercises could concentrate on the design side of the language. It was also hoped, that AMBA AXI bus would be used in the example designs.

The agreed learning objectives dictated that after the exercises the student should be able to create simple hardware designs with SystemVerilog knowing the basic syntax of the language. He should know how assertions are used in hardware design for validation and how testbenches can be used for simulation support and verification. The exercises should give an insight on how laborious it is to create a self-checking testbench on a low-level language as a motivation for the UVM training.

The final learning objectives are very design-oriented compared to the focus on verification in the education module. This was decided, because a verification engineer should know the basics of design as well. Not all the students attended the first education module that concentrated on design, and while the usage of VHDL language that was used there is similar, the student should know how the languages differ from each other. In addition, the focus was to introduce the student to a new language and this way he would get a more thorough overview of all the different properties of SystemVerilog. Assertions and testbenches were emphasized to keep up with the theme of the education module.

The exercises should have concentrated on the latest version of SystemVerilog for design. The language has improvements on older Verilog versions and exercises encouraged us-

ing the current syntax. However, some old Verilog structures are still a part of the language standard and it was predicted that the student would encounter them later. To avoid confusion later, the most common old syntaxes were introduced and then explained how SystemVerilog has improved them.

5.2 Structure

The exercises built on top of each other and the difficulty of the exercises quickly increased from introductory tasks to demanding design work. After the exercises, the student should have experienced a complete SystemVerilog design flow including a small module declaration, validation and verification of the design using assertions and testbenches and connection to a commonly used AMBA AXI communication bus using the SystemVerilog interfaces.

The exercises started with an introduction to the tools using a ready Hello World file written in SystemVerilog. Then, the students were given specifications of simple designs to get familiar with the syntax of the language. The later exercises build on top one of the designs made by the students. Along the exercises, they would see that the specification was not complete and the deficiencies may lead to errors. Assertions would be added to the design to catch the erroneous situations and a self-testing testbench would be used to catch bugs and to see if the assertions are triggered. Finally, a fixed and verified design would be connected to an AMBA AXI 4-Lite bus and the complete bus-connected module would again be verified with a self-checking testbench.

It was planned, that almost every student had enough time to start with the bus connectivity, but the testbench for the bus-connected module would be enough of additional work for the fastest performers. An estimation of the exercise schedule is shown in Table 2. The schedule estimation for each training day was shown to the students but it was emphasized, that they should not follow it strictly but do the exercises on their own pace.

Table 2. *Schedule estimation for SystemVerilog exercises*

Monday 5.10.	Headline
9:00 – 9:45	Setting up the simulation environment, Hello World exercise
9:45 – 10:30	First design with SystemVerilog: adders
10:45 – 11:30	Adders
12:30 – 14:00	Decrypter module
14:15 – 15:45	Validation with assertions
Tuesday 6.10.	
9:00 – 10:30	Verification with a testbench
10:45 – 11:30	Improving the design based on validation and verification
12:30 – 14:00	Bus wrapper for the module
13:15 – 14:00	Bus wrapper for the module
14:15 – 15:45	Bus wrapper for the module

5.2.1 Exercise 1: Introduction to SystemVerilog

The first exercise was an introduction to SystemVerilog. It introduced the usage of modules and the procedural statements, which were used as the basis in further exercises. In addition, the exercise served as an introduction to the simulation system used in the exercises so that the student could simulate his designs independently during the following exercises.

The first thing to do was to prepare the environment used in the exercises. This would be the first time the student would start the virtual machine, so it had to be set up. There was a sourcing script on the virtual machine that pointed the path variable of the operating system to the Modelsim installation folder and set up licensing information to allow running the simulation environment. The student was instructed in setting up the environment and then to test it by simulating a ready Hello World type design that is shown in Program 15.

```
module hello;

    initial begin
        $display ("Hello World");
        $info ("Edutech 2015");
        #2 $finish;
    end
endmodule: hello
```

Program 15. *Hello World code used as the first SystemVerilog example*

Running the example was instructed systematically. The student should first create his own design library and map it to a special library called Work that Modelsim uses for simulation files. Then the code can be compiled and simulated in console. The simulation environment should print “Hello world” and “Edutech 2015” and then finish.

After the student had tested the simulation environment, it was time to start the first design. An asynchronous adder would be used as an example, because it is simple enough and still a useful and quite a common design. The module would take in two 8-bit values from A and B operand inputs, add them together and output the result using 8-bit result and 1-bit carry signals. The student was instructed to create the structure for their first module systematically according to the Program 16, but had to combine some information to model the functionality of the module. Hints were given for continuous signal assignment and concatenation and for basic arithmetic operators.

```
// Start by declaring a module called adder:
module adder (operand_a_in, operand_b_in, carry_out, result_out);

// Declare ports as inputs or outputs:
// Inputs
input [7:0] operand_a_in;
input [7:0] operand_b_in;
```

```
// Outputs
output carry_out;
output [7:0] result_out;

// Add functionality and end the module

// When ready, end the module with
endmodule: adder
```

Program 16. *Structure of the module, as instructed in the exercise instructions*

The first example in Program 16 uses the non-ANSI syntax of Verilog language. All the input and output ports are first defined in the port list in module declaration and then later declared inputs or outputs, so there is some unneeded redundancy. This is still very common design practice and it was introduced in the beginning before a reason for the new, less verbose ANSI method was explained. An ANSI declaration for the signals in the module declared in Program 16 is as follows:

```
module adder (input [7:0] operand_a_in, input [7:0] operand_b_in,
             output carry_out, output [7:0] result_out);
```

After the adder was finished, it was simulated using a graphical waveform window of Modelsim. A ready adder.do file, that added the signals to the waveform window and generated some test input, was supplied to support the simulation.

The next step was to make a new synchronous adder. The SystemVerilog ANSI method of declaring a module, in which the directions of the signals are declared directly in the module port list, was introduced and explained. This task introduced procedures, which is a central SystemVerilog concept. A synchronous `always_ff` procedure was instructed, that would trigger its execution every time the clock signal has a rising edge or reset signal goes low. The module should also be able to do subtraction, controlled by an `add_sub` signal, in comparison to the previous task. When `add_sub` is high, an addition is performed and otherwise the B input is subtracted from A. Controlling the functionality by the `add_sub` signal would require an if clause that was introduced as a skeleton syntax for reset handling already in the task.

When simulating the design, the student is advised to create a simple simulation testbench instead of a do file. The simple testbench can be declared in the same file as the design as a separate module that instantiates the design, and its purpose is only to initialize the simulation and generate clock, reset and input values. The basics of testbench design were given as an info block that included instructions for usage of initial procedures for clock and reset generation, delays and component instantiation.

After the exercise, the student should already have used most of the key SystemVerilog concepts. He has declared his own modules with some logic connecting the input and output signals and instantiated a module inside a testbench. He has used a procedure in a

synchronous module that follows clock and reset signals that are generated for simulation by the testbench using delays and initial procedures.

5.2.2 Exercise 2: Decrypter module

The second exercise introduced more design concepts of SystemVerilog, most importantly data types, parameters and state machines. The design in this exercise served as a base for next exercises in which the design is validated and verified using assertions and self-checking testbenches, and finally connected to the AMBA AXI bus.

The students were given a specification of a hardware module that is depicted in the Figure 14 as process P4. The module is a part of a system, in which data is first encrypted (process 1) and written to a FIFO, then written to the shared memory (process 2), read from the memory and written to another FIFO (process 3) and finally decrypted and output (process 4) [20]. The encryption is done with permutation of the data by cutting it in half and rotating the halves after which the data is encrypted with a bitwise exclusive or (XOR) operation with a key value. The students are instructed to create a decrypter module that reverses the encryption operation that is the P4 process in the Figure 14.

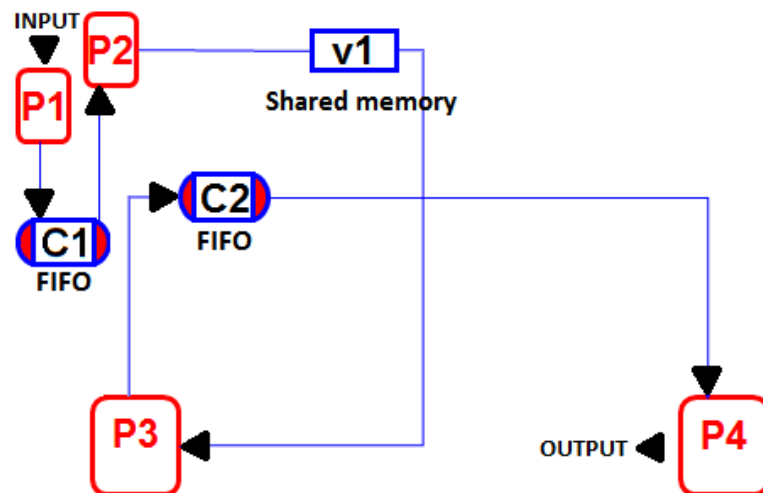


Figure 14. The complete encrypt-decrypt system. P4 is the process that the student has to implement in the exercise. [20]

The decrypter module was chosen as an example design for the exercises, because the system was already used in the SystemC exercises of the hardware design module of the education. This gives continuity to the exercise project, because a part of the students attended both the modules. The operation of the module is simple enough, so that most of the time would be used in learning the language and not polishing a complex algorithm. The module also requires handshaking signals and is a candidate to be connected via AMBA AXI bus in later exercises, because it is already a part of a complete system in which small components communicate with each other.

The specification for the module stated, that it should have a default data width of 32 bits, but should be able to be synthesized with all data widths that are multiples of 2. This would require usage of a parameter in the module declaration. The encrypted signal and encryption key would be input via separate ports and the decrypted signal has its own output port. The operation would be controlled by a master component with enable and valid signals so that the encoding starts with a rising edge of enable and when ready, the valid signal is set to indicate that the output port has a valid decrypted value. When the valid signal is high, the master confirms that the data is read by lowering the enable signal.

The student was encouraged to use a state machine with enumerated states for operation. The states would be for example idle, processing and ready, and a state graph matching the specification is depicted in Figure 15. It can be seen from the graph that the function of the system is not specified when the enable signal falls before the valid is set. The student is advised to draw a state diagram and find a fix for the errors in the specification. One example of an additional state transition that would fix the problem but lose the latest input data is marked in the Figure 15 with red.

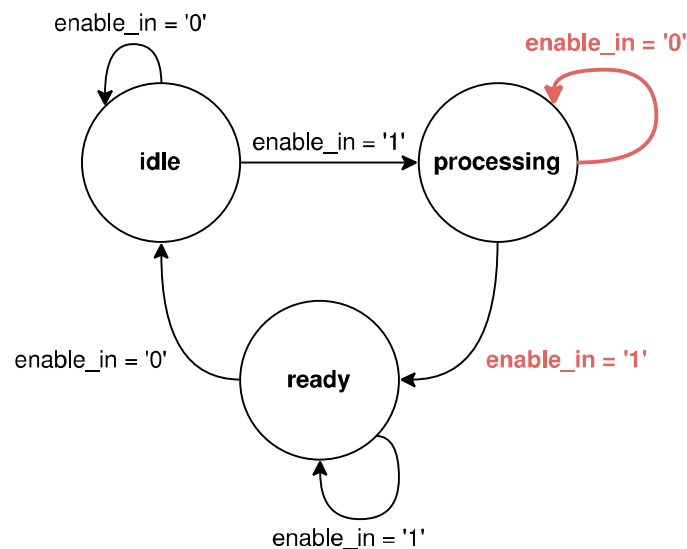


Figure 15. A state diagram for the decrypter module.

After the component was ready, it had to be simulated with a simple testbench. There were no additional instructions for creating the testbench, but the student should be able to apply experience from the first exercise to generate the clock, reset and input signals and run the simulation.

After finishing the exercise, the student should have more design experience with SystemVerilog. He should have a component that matches the specification. The component would be used as an example design in the rest of the exercises. Because the following exercises build on top of the design in this exercise, extreme care should be taken that every student is able to finish this exercise and all the problems should be answered without long delays.

5.2.3 Exercise 3: Assertions and testbenches

The third exercise introduced the student to validation and verification with SystemVerilog. The student was advised to use assertions to validate his own design and to verify the design using a self-checking low-level testbench that he has designed himself. According to the learning objectives in the module, the exercise 3 is the most important part of the SystemVerilog exercises.

The first task was to add assertions to the decrypter design. The student had to find the most critical points in the design and create assertions that would trigger if there is a change of losing data or corrupting it. The student was instructed to add at least three assertions. One valid assertion would be to check that the data width parameter is a multiple of two, because it is stated in the specification of the component.

Basic usage of assertions was instructed in an information block. The block contained examples for concurrent and immediate assertions with implication operators and system functions \$rose, \$fell, \$stable and \$past. One of the more complex example assertions is shown in Program 17. The assertion states that valid signal must rise after two cycles since enable has risen, but only if the component is not in reset.

```
assert property (@(posedge(clk)) disable iff(!rst_n)
    $rose(enable_in) |-> ##2 valid == 1);
```

Program 17. *example assertion*

The student was reminded that the reason for assertions is to catch unwanted performance and not to duplicate the hardware model [17]. For the SystemVerilog code in Program 18, the first assertion in Program 19 checks that when data is 0x05, the signal a is set on the next clock cycle. This assertion would be a poor one because it is bound to always be true according to the model. The second assertion in Program 20 assures, that the signal a is never set in any other case. This is not as certain, because some erroneous code elsewhere could set the signal a.

```
if (data == 0'h05)
    signal_a <= 1;
else
    signal_a <= 0;
```

Program 18. *Code sample for assertion examples.*

```
assert property (data == 0'h05 |=> signal_a == 1);
```

Program 19. *A poor assertion duplicates the hardware model.*

```
assert property (signal_a == 1 |-> $past(data) == 0'h05);
```

Program 20. *A better assertion checks that the signal_a is never set in any other case.*

The student was advised to simulate the design with various input signals when the assertions have been added to see if they are triggered correctly. The simple testbench in exercise 2 could be used in this task. It depends on the assertions and the testbench made by the student, if the assertions trigger. If the assertions do not trigger, the student should be advised by the assistant to add more input patterns to the simple testbench.

The second task in the exercise was to generate a self-checking testbench to verify the design. The testbench should be divided to multiple initial and always procedures that generate the clock and reset signal, give randomized encrypted input and key values to the DUT and compare the DUT output to the reference values. An example block diagram of the testbench connected to the DUT is introduced in the Figure 16. All the procedures in the testbench are shown as blocks in the picture. The print result procedure is used for final checking after the test is finished. The procedure checks that there are no more unprocessed input values in the buffer and if result counters were implemented in the monitor procedure it can show a final report indicating if the test was successful or not.

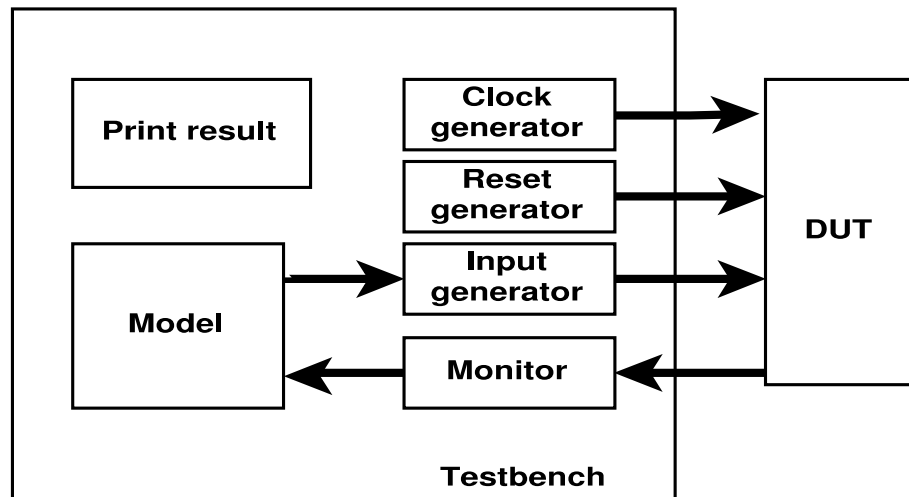


Figure 16. Block diagram of the testbench for the decrypter module.

The student was encouraged to implement some dirty testing as well using all kinds of transactions that do not follow the specification. Some examples for dirty test inputs were given, in which the testbench would lower the enable signal when it should not be done and change the key when the DUT is still processing the data.

After validation and verification, the student should have found out that the design is not completely stable for production and deficiencies in the specification would lead to data loss or corruption in some cases. The student was advised to improve the design to prevent these situations. One fix would be to improve the handshaking by adding a new signal for the master to confirm that the output has been read, so glitches in enable signal would not affect the functionality.

After the exercise, the student should have experience of SystemVerilog assertions and self-checking testbench design. He would have seen the results of verification and corrected behavior that does not match the specification. Furthermore, he would have noticed the problems caused by an incomplete specification for a simple design and taken measures to improve the specification. The changing of specification would not always be possible in real design work, but for education purposes, it served as an example.

5.2.4 Exercise 4: Bus connectivity

The exercise 4 introduced the student to SystemVerilog interfaces, which were part of the learning objectives of the education. The student had also more practice on design work that follows a strict specification.

When starting the exercise 4, the student should have a working and verified design for the decrypter module. The final task was to connect the module to an AMBA AXI bus. For simplicity, the bus standard used was the lightweight AMBA AXI4-Lite and some further simplifications were done to abstract away the unnecessary control signals. The bus connectivity would be implemented using a wrapper module, as shown in Figure 17.

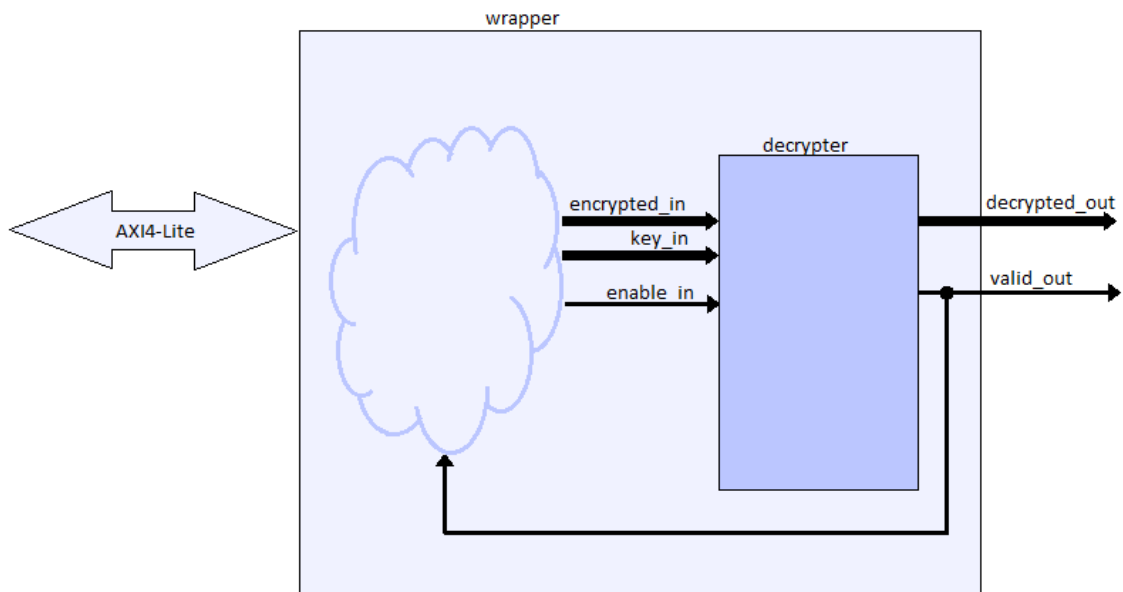


Figure 17. AXI wrapper.

The simplified AMBA AXI4-Lite bus in the exercise can be separated into five channels with similar signals but different directions and data widths [7]. Every channel has an information signal carrying data, address value or response, and valid and ready channels for bus handshaking. The implementation of the channels would be an introduction to SystemVerilog interfaces. A channel with a variable width information signal and handshaking wires would be declared as an interface and then instantiated in the design to form all the channels. Because the design in the exercise uses the bus only in one direction

in which the master writes to the slave and never reads, the read address and data channels will be omitted. The directions of the signals would be declared using modports so that the write address and data channels are directed from master to the slave and the response channel is reversed.

The student was given a specification for a wrapper module, which instantiates the decrypter module and connects its input to the AMBA AXI bus. The output of the decrypter is connected directly to output signals. It should be noted that the specification stated that the data transfers are written to the latest address sent, so multiple consecutive data transfers can be performed. This is not compliant with the AMBA AXI4-Lite specification, where every transaction should be started with an address, and was a simplification for education purposes [7]. The fact was emphasized in the exercise instructions.

A simplified AMBA AXI4-Lite bus specification was also included in the instructions [7]. The specification declared all the channels and their signals, the transaction schedule with timing diagrams of legal transactions and slave response information.

The exercise was very demanding compared to the previous ones. It was predicted that only the fastest students would be able to finish the exercise. If the student still had time after finishing the wrapper module, an additional task was to build self-checking a testbench for it. The most important content in the exercise was to get familiar with SystemVerilog interfaces and the exercises were scheduled so that everyone should be able to start defining the interface.

After the fourth and final SystemVerilog exercise, the student has experienced a complete design and verification flow of a simple module that can be connected via a bus interface to a larger system. He has used the key design features of SystemVerilog including declaring parameterized modules with synchronous procedures, state machines with enumerated state variables and interfaces that have been instantiated to form multichannel buses. Assertions have been used to ensure valid functionality.

6. UVM EXERCISES

The UVM exercises were designed based on old verification exercises that were used on the course TKT-1410 Suunnittelun varmennus (System Verification) during spring 2013. The exercises were built with OVM and because most of UVM is based on OVM, a large part of the exercises resembled an UVM testbench already. The exercise planning was started by completing the OVM exercises and finding out all the changes brought by UVM. The testbench in the OVM instructions was then rebuilt from scratch following UVM guidelines and the exercise instructions were rewritten.

It was predicted that the fastest students would complete the exercises in two days. Therefore, additional exercises were designed, where emphasis was not on new crucial concepts but on deeper understanding and reuse. This way the fastest performers would have enough tasks and can gain more experience with UVM, but still everyone would achieve the learning objectives.

6.1 Learning objectives

The original learning objectives made by the customer stated, that after the education module the student masters the UVM class library and methodology so that he can produce a working test environment using the following UVM mechanisms:

1. UVM Agent
2. UVM Scoreboard
3. UVM Environment
4. UVM Test
5. UVM Register Abstraction Layer (RAL)
6. UVM Configuration Database
7. UVM Factory
8. Constraint random with UVM
9. UVM Phasing
10. Reuse by extending UVM classes

In a meeting with the customer it was agreed, that the objectives stated would otherwise be followed in the planning of the exercises, but the UVM register abstraction layer would be omitted and it is enough if the register abstraction layer was mentioned on a lecture level. The reason for the omitting of RAL was that it was not easily implementable to the system and it would require a lot of time to master the concept, so the limited training time should be focused on more important basic principles.

The summary of the learning objectives on the exercise instructions state, that the exercises will introduce the following principles:

1. The object-oriented properties of SystemVerilog language
2. The UVM class library and its usage
3. Verifying a design with UVM testbench
4. Reusing existing components for new designs

6.2 Procedures and UVM concepts used in the exercises

In a meeting with the customer it was agreed, that UVM experience is more important than the internal testbench design guidelines of the company. Therefore, the exercises were built to follow the approved guidelines provided by Verification Academy, which is a training website for verification engineers maintained by Mentor Graphics. [5]

The testbenches were designed so that every stand-alone compilation unit files that are input to the compiler – the design under test, top-level module of the testbench and the package file - had an extension *sv* and all the class declarations were in separate files with an extension *svh*, SystemVerilog header. All the class declaration files would have names that match the class name and each file would contain only one class declaration. All these files would be included in the package file that is imported in the top level, and includes are not performed anywhere else. Notable exceptions are the *uvm_macros.svh* file that is part of the UVM source code and required to be imported in the top level and the DUT interface description, which also has to be included on top level. This way the whole testbench would be included in one package and connected to the dut in the top-level module. If the components were used later with another DUT with similar signals and function, the package could be imported in another top-level design. The package and top-level file had post-fixes *_pkg* and *_top* in the file and design unit name.

The layout of the testbench folder that would be delivered to students at the start of the exercises is shown in Figure 18. All the files marked with green are class description files that are included in the package file, which is imported on top level. The red files are given to the compiler and the blue *dut_if.if* file is included in the top-level module file. The other files are to aid in the start of the simulation and not a part of the testbench. The students will be instructed to follow the same layout.

Name	Type	Size
agent.svh	SVH File	2 KB
driver.svh	SVH File	3 KB
dut_config.svh	SVH File	1 KB
dut_if.if	IF File	1 KB
env.svh	SVH File	1 KB
fifo.vhd	VHD File	9 KB
fifo_tb_pkg.sv	SV File	2 KB
fifo_tb_top.sv	SV File	2 KB
Makefile	File	1 KB
sequence.svh	SVH File	2 KB
sequencer.svh	SVH File	1 KB
test_base.svh	SVH File	2 KB
transaction.svh	SVH File	2 KB
vsim.do	DO File	1 KB

Figure 18. Testbench folder layout.

6.3 Structure

The UVM exercises were divided into three parts. The first part was divided into 5 exercises with the goal of getting familiar with basic usage of UVM. In the second part, the student should be able to create a new UVM testbench himself for a different, more complex design with the experience gained in the first part. Finally, a complete integration level testbench, that tied together several block level environments via vertical reuse, was provided in the third part for examination. The second part contained two voluntary exercises and the third part contained only a single exercise, so the total number of exercises was eight.

The first part of the exercises was the most important content and the exercises were planned so that every student would be able to finish the part without schedule problems. The aim of the first part was to design a full UVM test environment for a simple design using all the most basic UVM concepts. A block diagram of the finished testbench is shown in the Figure 19. An error free FIFO block modeled in VHDL was provided to be used as a design under test and when the test bench was proven to be working correctly, it was used to find bugs in other FIFO designs.

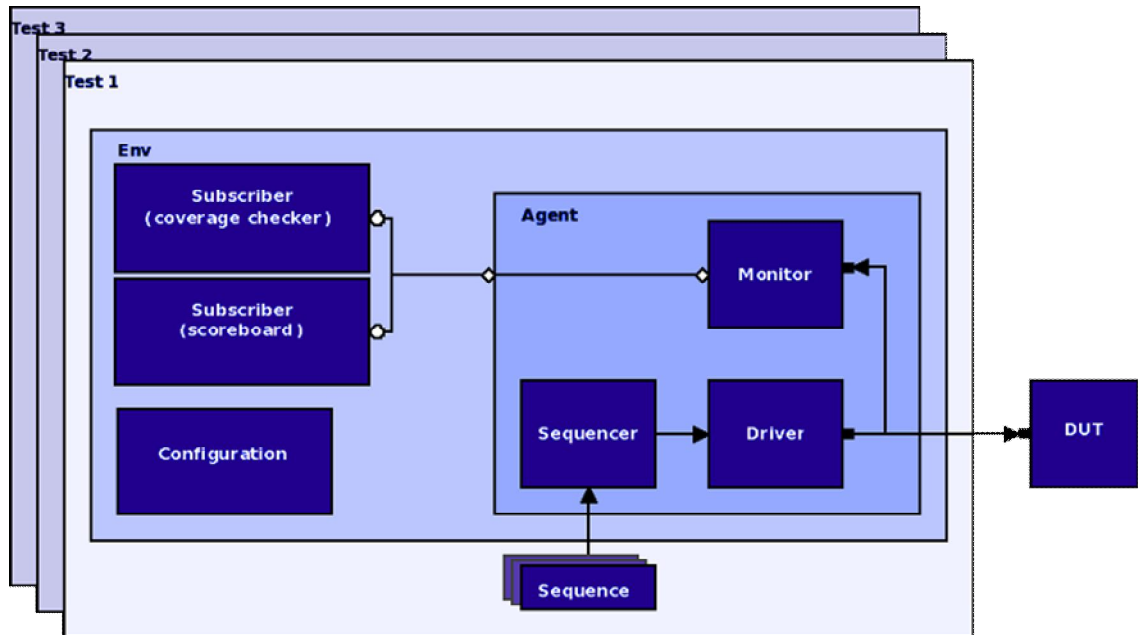


Figure 19. The final testbench after finishing exercise 5.

As in SystemVerilog exercises, an estimation of a schedule was given to the students but it was emphasized, that everyone does the exercises on their own pace. The estimation is shown in Table 3.

Table 3. Schedule estimation for UVM exercises.

Friday 23.10.	Headline
9:00 – 9:45	Introduction to UVM
9:45 – 10:30	Creating classes: UVM monitor
10:45 – 11:30	Creating classes: UVM monitor
12:30 – 14:00	Subscribers and analysis ports
14:15 – 15:00	Subscribers and analysis ports
15:00 – 15:45	Automatic checking: UVM Scoreboard
Thursday 29.10.	
9:00 – 10:30	Automatic checking: UVM Scoreboard
10:45 – 11:30	Automatic checking: UVM Scoreboard
12:30 – 14:00	Randomized tests
14:15 – 15:45	Randomized tests
Friday 30.10.	
9:00 – 10:30	Verifying a design with UVM testbench
10:45 – 11:30	Verifying the decrypter
12:30 – 14:00	Verifying the decrypter
14:15 – 15:45	Verifying the decrypter

Table 4 shows a summary of the exercises and how they connect to the original learning objectives made by the customer. *I* in the table means that the content has been introduced but not mastered after the task. *X* marks that the objective is an integral part of the exercise. As can be seen from the table, the first exercise introduces only a few concepts but

as the concepts are used in later exercises as well, the most basic principles get a lot of repetition and deeper explanation. Coverage, that was the theme of the third exercise, was not mentioned in the original learning objectives for the UVM part of the education but coverage-driven verification was listed in the main topics for the verification module, which is why the table does not show improvements in learning between exercises 2 and 3. The exercise 5 only includes running the final testbench to find bugs in designs. Therefore, the structure of UVM is not focused on in this exercise.

Table 4. *Learning objectives fulfilled in each exercise.*

Exercise	1	2	3	4	5	6	7
UVM Agent	I	X	X	X		X	X
UVM Scoreboard				X		X	X
UVM Environment	I	X	X	X		X	X
UVM Test	I			X	X	X	X
UVM Register Abstraction Layer (RAL)							I
UVM Factory		I	X	X		X	X
Constraint random with UVM				X	X	X	X
UVM Phasing	I	X	X	X		X	X
Reuse by extending UVM classes				X		X	X

This chapter explains the contents of all the exercises precisely. The precise explanation describes the amount of work the student has to do in each task, but on the other hand, it provides an insight of the UVM principles that would be required for creating a complete testbench. The code examples in this chapter support the theory of UVM architecture in Section 3.2.

6.3.1 Exercise 1: Introduction to UVM

The purpose of the first UVM exercise was to introduce the student to UVM. The exercise was planned to be more about exploration of UVM concepts and less about producing new code. The decision was made, because producing the whole testbench would take a lot of time and most of the work would be unnecessary repetition. In addition, the ready-made code would function as a syntax example that would help the student get familiar with the object-oriented side of SystemVerilog language and concepts in UVM. Inspecting a ready-made code that is ready for simulation helps to understand the procedures needed for producing new code.

The student was provided with a basic UVM testbench that was used to verify that the simulation system runs correctly. After running the simulation for the first time, the main task in the exercise was to find out what was the purpose and function of all the ready-made components. Figure 20 shows a block diagram of the state of the testbench that is delivered to the student.

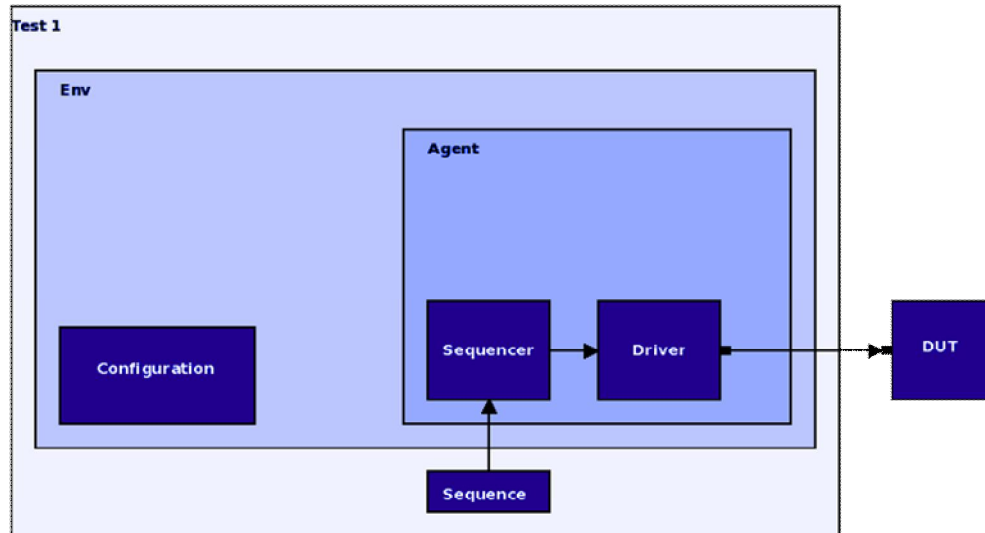


Figure 20. *The base testbench delivered to the student.*

As can be seen from the Figure 20, the testbench contained only bare necessities to run a simulation. The only component in the environment was the agent, which was missing the monitor. A sequence with only one sample transaction was delivered by the sequencer to the driver and driven to the DUT. The configuration database was used to deliver the virtual DUT interface to the driver. The missing components would be added by the student in the later exercises.

First, the student had to download the compressed exercise package and extract it with console commands given. There were reminders for configuration of the virtual machine environment that was already familiar from the SystemVerilog exercises. The exercise package contained a Makefile for the make automation system, so a console command “make” was sufficient for compiling the testbench with the DUT and running the simulation in console. An example simulation output was given for reference.

The final task was to add more transactions to the sequence so that at least one read and one write operation is done. A sufficient sequence file is shown as an example in Program 21. The student was instructed to use specific `create`, `start_item` and `finish_item` methods for starting and finishing the transaction and the contents of the transaction could be copied from a transaction that was already in the sequence.

```
class basic_sequence extends uvm_sequence #(transaction);
  `uvm_object_utils(basic_sequence)

  function new(string name = "");
    super.new(name);
  endfunction: new

  task body;
    // handle for a transaction object
    transaction tx;
```



```

// Make a write to the DUT
tx = transaction::type_id::create("tx");
start_item(tx);
tx.data_to_DUT = 3;
tx.write_enable = 1;
tx.read_enable = 0;
tx.rst_n = 1;
finish_item(tx);

// Make a read from the DUT
tx = transaction::type_id::create("tx");
start_item(tx);
tx.data_to_DUT = 0;
tx.write_enable = 0;
tx.read_enable = 1;
tx.rst_n = 1;
finish_item(tx);

// Make a clearing transaction in the end
tx = transaction::type_id::create("tx");
start_item(tx);
tx.data_to_DUT = 0;
tx.write_enable = 0;
tx.read_enable = 0;
tx.rst_n = 1;
finish_item(tx);

#10;

endtask: body
endclass: basic_sequence

```

Program 21. *sequence.svh file contents after completing exercise 1.*

The first exercise was a very straightforward introduction to UVM. After the exercise, the student should be familiar with the UVM class library and its usage. He should have examined the most basic UVM classes and found out their purposes. He has also experience of running simulations in the Modelsim tool using the make automation system.

6.3.2 Exercise 2: Creating classes

The objective in the second exercise was to introduce the student to class declarations in UVM focusing on the monitor component. The monitor was the only component that was missing from the agent, which is a key component in UVM architecture. The exercise also explained the architecture of UVM environment by teaching the student to instantiate his first own component in the hierarchy.

A block diagram for the system the student would have after the exercise is shown in the Figure 21, where the new monitor component is highlighted in yellow. As the exercise was the first introduction to UVM class creation, the instructions were very straightforward and the exercise could be completed by following the instructions systematically copying and pasting all the code lines provided.

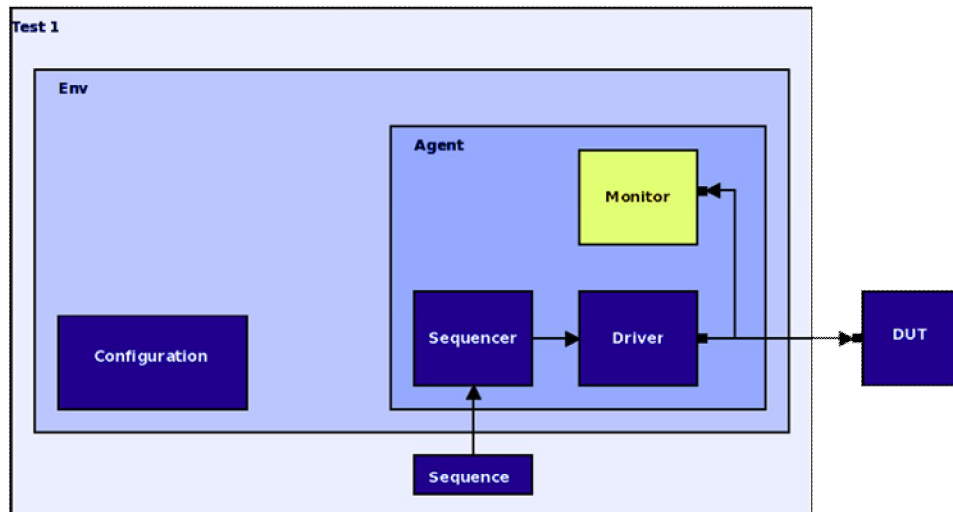


Figure 21. Testbench with the monitor attached.

The student was first instructed to fill out a skeleton for the monitor using a basic structure. The structure would contain an introduction of a new class extended from an appropriate base class including a registration macro, a constructor and almost empty phase method declarations. Only a debug print was advised to be added to the build phase method. An example skeleton created by following the instructions is shown in Program 22 and it contains all the common code lines that are used every time when creating a component in UVM – only the required phase methods and their contents change between components.

```

// Class declaration
class monitor extends uvm_monitor;

    // Registration macro
    `uvm_component_utils(monitor)

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new

    function void build_phase(uvm_phase phase);

        // Debug prints
        `uvm_info("monitor", "Created monitor", UVM_HIGH)

    endfunction: build_phase

    task run_phase (uvm_phase phase);
    endtask: run_phase

endclass: monitor

```

Program 22. Monitor skeleton.

The still empty phase methods were next filled out with explanations for every code line. Creating the component systematically starting from the highest possible level and then

fine-tuning the contents of the methods helps the student to keep focus on the task. This order also prevents blind copying from the exercise instructions, because the student has to be more alert when the file is not constructed line by line in a linear order. Between the instructions, the student was provided with additional information and reminders about UVM phases, the configuration database and the info reporting macro with verbosity levels.

The first thing that the monitor should be able to do is connection to the DUT interface as shown in Figure 22. The connection is done by getting the configuration object containing a virtual interface handle from the configuration database. The configuration object was already created in the test and used by the driver. To read the configuration database the monitor needs variable handles for the configuration object and the virtual interface. Creation of the handles was instructed to the student using the following lines:

```
dut_config dut_cfg;
virtual dut_if dut_vi;
```

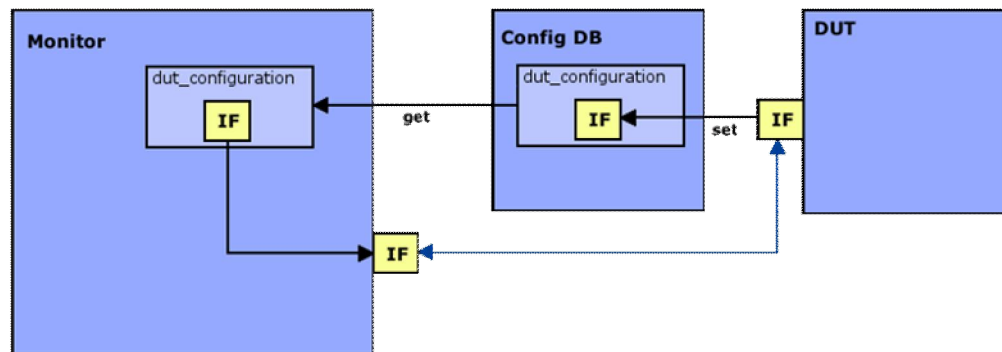


Figure 22. Delivering DUT interface to monitor

The configuration object is read and connected to its handle by using the get method of the configuration database and the virtual interface is then connected to the interface handle as shown in Program 23. If for some reason the configuration object cannot be fetched, the testbench would not be able to run and therefore the fetching is enclosed in a conditional clause that ends the simulation using the `uvm_fatal` macro in case of a failure.

```
// Get the configuration object from config DB
if(!uvm_config_db#(dut_config)::get(this, "",
    "dut_configuration", dut_cfg) )
    `uvm_fatal("NOVIF", "No virtual interface set");

// Connect the handle to the virtual interface in the object
dut_vi = dut_cfg.dut_vi;
```

Program 23. Getting the object from configuration database in the build phase method

The final thing for the student to fill out in the monitor in this exercise was the run phase method that contains all the functionality of the component. The run phase contains a loop that runs infinitely until the simulation is forced to stop. On every iteration of the loop,

the monitor reads the status from the DUT interface into a transaction object. The run phase method should be completed by copying the code lines in the instructions is shown in Program 24. The student was shown how to copy the input data of the DUT to the transaction object, but he had to apply the information to copy all the remaining signals. As there were no components reading the transaction object yet, the student was instructed to add a debug print in the run phase loop. A more advanced usage of the reporting macro, which prints the contents of the data, is used in the example in Program 24, but printing for example “Got data” would be sufficient in this exercise. In the example, the contents of the data are printed using a `psprintf` method and its usage was instructed in the information block about the info reporting macro.

```
task run_phase (uvm_phase phase);
  forever begin: mon_loop
    // Transaction handle
    transaction tx;

    // Wait for clock tick and copy DUT state to transaction
    @(posedge dut_vi.clk);
    tx = transaction::type_id::create("tx");
    tx.rst_n      = dut_vi.rst_n;
    tx.data_to_DUT = dut_vi.data_to_DUT;
    tx.write_enable = dut_vi.we_out;
    tx.read_enable  = dut_vi.re_out;
    tx.data_from_DUT = dut_vi.data_from_DUT;
    tx.full        = dut_vi.full_in;
    tx.one_p       = dut_vi.one_p_in;
    tx.empty       = dut_vi.empty_in;
    tx.one_d       = dut_vi.one_d_in;

    // Write the transaction object to analysis port
    ap.write(tx);

    // Debug prints
    `uvm_info("monitor",
              $psprintf("Got data: to: %0h,
                        write: %b, read: %b,
                        from: %0h",
                        tx.data_to_DUT,
                        tx.write_enable,
                        tx.read_enable,
                        tx.data_from_DUT),
              UVM_HIGH)
  end: mon_loop
endtask: run_phase
```

Program 24. *Run phase task in the monitor*

The final task in the second exercise was to connect the component to the testbench. The student was reminded that the correct place to instantiate the monitor would be inside the agent. The instructions advised to include the code file in the testbench package and create a handle that the monitor is instantiated to in the agent using the UVM factory call. The student was bound to encounter an example of component instantiation in the agent, where the driver and the sequencer were already in place, so there were no step-by-step

instructions. After the monitor had been connected, its function was tested with simulation.

After the second exercise the student should have experience of declaring his own UVM classes and instantiating them to the testbench. He would know which methods are needed at the minimum for a component declaration and when the methods are run in the scope of simulation. He has also read a configuration object from the configuration database using a ready code snippet and connected the contained virtual interface to a handle, but at this point it cannot be required that he understands the usage of configuration database or virtual interfaces and could produce similar code himself. It is only mandatory to know the reason, why the process has been done.

The exercise is straightforward, but there are two possible problem points. The student has to examine the DUT interface declaration file to find out names of the signals in the virtual interface. They have to be copied in the run phase task to the transaction object that has some differences in the signal names. Another danger point is that the order of the include macros in the package file is crucial for the compiler. For example, an agent that includes the monitor cannot be compiled, if the monitor has not yet been declared. These are not instructed in the exercise and the student has to figure them out himself, but the assistant should be ready to help if students are stuck. Wrong order of compilation and faults in signal names are usual sources of errors, so first-hand experience would help to keep them in memory.

6.3.3 Exercise 3: Subscribers and analysis ports

The third exercise introduced a coverage collector to the testbench as shown in Figure 23. The coverage collector is the component that is responsible for coverage-driven verification in UVM, which was considered an important learning objective. When implementing the coverage collector, the student was also introduced to the analysis ports that are an important concept in UVM.

The coverage collector reads the transaction objects sent by the monitor and saves signal state history to a covergroup. The contents of the covergroup can be examined to get numerical data of input combinations that have been written to the DUT and the information can be used to check if the functional coverage of the testing has been sufficient.

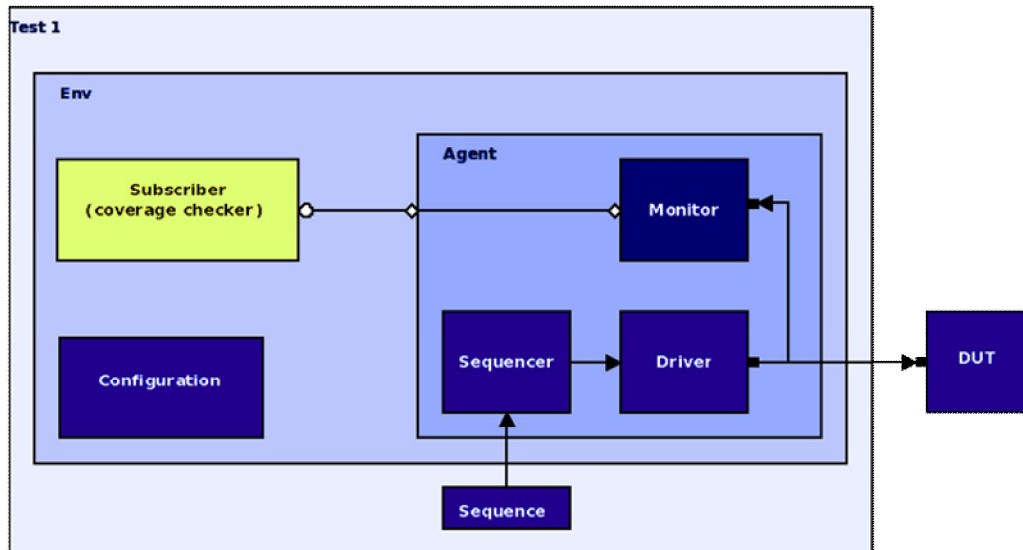


Figure 23. Coverage collector added to the testbench.

The student was instructed to create a skeleton for a coverage_collector class that is extended from uvm_subscriber like in the previous exercise. The only difference to the skeleton of the monitor component is that uvm_subscriber is a parameterized base class. The parameter for the class would be the name of the transaction object class that is received from the monitor. The student was given an example of parameterized class declaration as shown in Program 25 and told to parameterize the class with the transaction class. The instructions reminded that the skeleton should have a registration macro, a constructor and a build phase method.

```
class my_class extends base_class #(parameter);
```

Program 25. declaration of a parameterized class

After the student had a skeleton for the coverage collector, the next step was to start implementing the coverage collector functionality. A covergroup was declared with the example code in Program 26 that the student was told to copy right after the registration macro. The covergroup should also be created in the component's constructor using the new function. An information block explained that the covergroup consists of coverpoints, which are broken into bins. The example code in Program 26 declares one coverpoint for the data that is input for the DUT. The student's task was again to fill out rest of the signals that in his opinion should be monitored for coverage. The bins were not declared, so for the 32-bit data value the coverpoint was divided evenly by the numerical value to 64 bins by default.

```
covergroup cg;
  c_data_to_DUT : coverpoint sample_tx.data_to_DUT;
  ...
endgroup: cg
```

Program 26. Declaration of a covergroup.

Another difference between the monitor and subscriber components is that a subscriber reacts on transaction made by other components and not every clock cycle. Therefore, it does not require a run phase method but the functionality is implemented in a write function instead. The write function is implemented in the subscriber and then can be called by the component that is sending data, which in this case would be the monitor.

The student was instructed to declare a write function with code lines that have been combined together in Program 27. In the example code, the transaction is assigned to a variable and the covergroup cg is updated based on the transaction data using its sample method. This would be all that is required for a write function in a coverage collector.

```
function void write(transaction t);
    sample_tx = t;
    cg.sample();
endfunction: write
```

Program 27. Write function of a coverage collector.

The structure of a coverage collector component is simple and the code lines in Program 26 and Program 27, when inserted into a component class skeleton, would be enough to declare a working coverage collector into a UVM testbench. The coverage data is not printed in the console, but it can be examined by the vcover tool included in the ModelSim. In addition, an info block hinted that the covergroup has a get_coverage function that returns an integer value of the coverage percentage at the moment it is called. The value can be for example used for printing or finishing the simulation when a desired coverage percentage has been reached. The function is internal to the covergroup and cannot be called by the other components in the testbench, but a function declaration was introduced that can be used if the student wanted for example to read the coverage value in the test component. The hint block explained that it is not necessary to implement this functionality now but it could be useful in later exercises.

The student was told to instantiate the coverage collector in the testbench environment. Instantiation was not instructed specifically, but the student was able to refer to the exercise 2 instructions if the process was not yet familiar enough. When the component had been instantiated, it was still not connected to any component and the next step was to add an analysis port connection to the monitor. The connection is highlighted in yellow in Figure 23, where a diamond connection indicates an analysis port and the round connection in the subscriber is an analysis export.

In the analysis port system, when a component writes to its analysis port, the write function call is delivered as a broadcast to all the analysis exports that are connected to it. Therefore, a component that implements the transaction and has a write function declared has an analysis export and the component that initiates the transfer by calling the write function includes an analysis port. Because the monitor is inside the agent and cannot be reached directly by the subscribers outside the agent, the agent should have an analysis

port as well, which is connected to the monitor and delivers the transaction through. The creation of analysis ports for the monitor and the agent was instructed with code lines in Program 28. A parameterized class, which has been declared in the UVM library, was used as it was.

```
// Declaration of an analysis port
uvm_analysis_port #(transaction) ap;

// Create the analysis port in the build phase method
ap = new("ap", this);
```

Program 28. *Creation of an analysis port.*

After the analysis ports had been created, they had to be connected to the export. A new UVM phase, connect phase, was introduced to the student. The connect phase is executed after the build phase and its purpose is to make all the connections between components, so it would be a valid place for connecting the analysis ports. The student was advised to create connect phase methods to the environment and agent classes, in which a connect method of the analysis port class is called to make the connection. The agent had a connect phase method already declared, because the connection between the driver and the sequencer had already been done. Code lines for the contents of the environment and agent connect phase methods were given to the student as shown in Program 29. The handles for both the analysis ports are called ap in the Program 29 and it should be noted that the codes expect for the coverage collector's handle to be cov_h. This might vary in the student's own implementation and has to be changed according to the real handle name.

```
// Connect the agent's analysis port in the environment's connect phase
agent_h.ap.connect(cov_h.analysis_export);

// Connect monitor's analysis port in the agent's connect phase
mon_h.ap.connect(ap);
```

Program 29. *Connect phase method contents for connecting the analysis ports*

The analysis port in the monitor had now been connected to the export and in the second exercise, the DUT pin state was copied into a transaction object. The final thing to do was to add a line of code in the monitor, in which the write function of the analysis port is called using the transaction object containing the current state of the DUT inputs and outputs.

The simulation tool can be set to collect code coverage data during the simulations in addition to the functional coverage data collected by the coverage collector. To save all the coverage information to a file to be examined later, the Makefile had a ready command that had been commented out:

```
#coverage save -onexit vsim.ucdb
```


The command saves the coverage data to a file `vsim.ucdb`. There were ready targets in the Makefile for examining the data either in console or by generating an html report out of it. The student was advised to examine the coverage data with simulating the testbench again.

The instructions for the exercise were less verbose than in previous exercises. The student should have had already some experience of creating a component skeleton and instantiating it in the testbench, so repeating the instructions was unnecessary. The new component declaration was simple, but the student had been provided with enough information for declaring even complex coverpoints with specific bins if he wished to. The hardest part to comprehend in the exercise would be the concept of analysis ports.

The exercise introduced new and important UVM concepts to the student, notably parameterized classes, analysis ports and covergroups, and after completing the exercise, he should understand the function and purpose of them. After the exercise, the student should be able to declare his own component classes without any help and instantiate components in the testbench.

6.3.4 Exercise 4: Automatic checking and randomized input

The fourth UVM exercise introduced the student to the automatic checking capabilities of UVM. The exercise introduced the student to UVM scoreboard class, randomization of transactions and multiple tests that are extended from the previous base test class. Randomization enables constraint random testing, which was declared as one of the most important objectives in the verification module. Deriving new tests from a previous class declaration introduced the student to the hierarchical reuse of UVM components.

The task was to declare the final basic component, a scoreboard that is highlighted in yellow in Figure 24, in the testbench. The testbench would be capable of automated testing and ready for running simple directed tests using the ready `test_base` component. As a second task, the testbench was improved by adding a new test class that performs constrained randomized testing with a different sequence. The exercise was hardest of the first part in the UVM exercises and most of the implementation was left for the student. Only the necessary guidelines for completing the exercise were given.

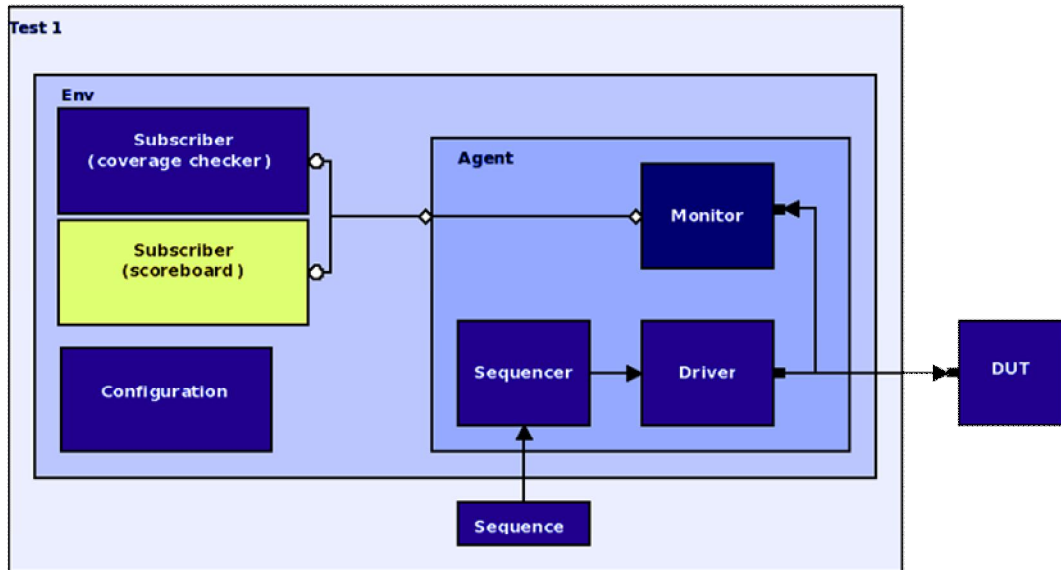


Figure 24. Scoreboard added to the testbench.

The scoreboard is a component that contains a model of the DUT against which the behavior of the DUT can be compared. It would be connected to the same analysis port as the coverage collector and the checking would be performed in the write function. When writing and reading from the DUT model at the same time as the actual DUT performs the operations, the outputs can easily be compared.

The student's task was to create the DUT model himself using a data structure for storing the data. A SystemVerilog queue was given as a suggestion, but the student was free to use whatever structure he found suitable. In addition to the data storage, the model should raise status flag signals that indicate the amount of data stored according to the specification of the DUT. The student was advised to use either assertions or conditional statements with the `uvm_error` reporting macro when the DUT behavior differs from the model.

A specific difference to the third exercise is that the base class used for the scoreboard is `uvm_scoreboard` that is not a parameterized class and does not include an analysis export. The student had to declare the analysis export himself using ready code lines shown in Program 30. The analysis export is an instantiation of the analysis implementation class provided by the UVM library and it is parameterized by the type of the transaction object and the name of the component in which the analysis export is instantiated.

```

// Declare an analysis export:
uvm_analysis_imp #(transaction, scoreboard) analysis_export;

// Create the analysis export in the constructor (function new):
analysis_export = new ("analysis_export", this);

```

Program 30. Declaring an analysis export.

After the student had completed the scoreboard and connected it to the environment, the testbench was ready for directed testing. The sequence could be improved by adding more transactions and if enough transactions were added, the DUT behavior could be checked in all cases. However, writing directed tests that check all the corner cases is very laborious and not effective even when the DUT is as simple as a FIFO. The amount of transactions needed for checking all the usage cases would grow high and it would be easy to miss some features. Therefore, the next step in the exercise was to extend the testbench with randomized sequences.

The student was instructed to create a new sequence that is extended from the basic sequence used in the exercises. The new sequence should use the `randomize` method built into the transaction object. The declaration of the transaction can specify the signals that should be randomized using the `rand` keyword. When the `randomize` method is called in the sequence, the simulator generates random values for those signals. When the sequence contains a loop that generates new randomized input on every iteration, more corner cases will eventually be revealed as the amount of iterations is increased. Using a loop in the sequence was not instructed and it had been left for the student to find out, but the assistant could hint the student to that direction. In addition, the transaction object declaration specified only some signals to be randomized, and this should be hinted to the student as well.

Constraints are an additional UVM concept that would aid in randomized testing. Constraints can be used for example to limit the randomized values as in the first example in Program 31, or to specify distributions as the second example shows. The distribution in the second example specifies that the `write_enable` signal should be high every 10 cycles on average. An information block hinted the purpose and basic syntax for adding constraints in either the transaction object or in-line in the sequence.

```
// Base constraints set the values the signal can have:
constraint c_data { data >= 0; data < 1024; }

// Constraints can also be used to set distributions. For example:
constraint c_we_dist { write_enable dist { 0 := 9 , 1 := 1 }; }
```

Program 31. Constraint examples.

The student was advised that rather than using the new random sequence in the existing test, it would be better to declare a new test for the new sequence. This way the directed test already specified could be still be used without modifying the testbench every time the user wishes to change the method of testing. A new test, which is extended from the ready `test_base` class, was instructed to be created. The random test inherits the `build` phase method, which creates the environment, from the `test_base` class, and only declares a different run phase task that uses a different sequence. The build phase task is inherited by calling `super.build_phase` method in the build phase. The student should be able to declare his own test using the `test_base` class declaration for reference. The final UVM

testbench would appear as is shown in the Figure 25. The testbench in the figure has three different tests, each using different sequences.

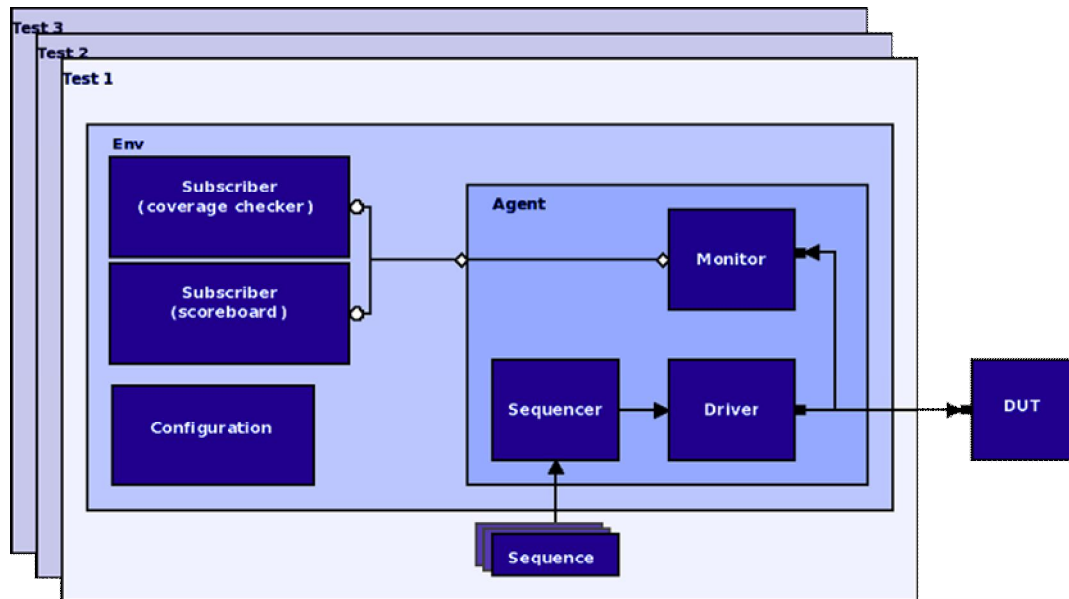


Figure 25. The final UVM testbench.

Furthermore, an alternative that reuses also the run phase task but replaces the sequence object using the UVM factory override system, was provided in an information box. The test could specify an override, as shown in Program 32, that changes the type of the sequence with another one extended from it in the end of elaboration phase that is run after the build phase but before the run phase in which the sequence is actually created. This is a more advanced concept of UVM and was not mandatory in the exercises, but the basic usage of the factory overrides was provided as a curiosity.

```
basic_sequence::type_id::set_type_override(random_sequence::get_type());
```

Program 32. *Override example*

As the student had already seen in the first exercise, the name of the test to be run was specified in the top-level component of the UVM testbench as a parameter for the `run_test` method. The instructions told that the name of the test can be changed there, but a better approach would be to leave the parameter of the `run_test` method empty. This way the name of the test to be run can be specified on the command line when starting the simulation. A variable for the test name has been already included in the make file.

After the exercise, the student should be familiar with all the key concepts of UVM. He has created a complete UVM environment using all the basic components and declared a constrained randomized sequence that is run in a randomized test. He would have experience of creating a high-level model of the DUT in the scoreboard and used the data provided by the monitor for automated testing. He would also understand the concept of tests and how they differ from the UVM environment.

6.3.5 Exercise 5: Verifying FIFO blocks

The fifth exercise was designed to show the student the UVM testbenches in action. He would see himself how the testbench is able to find bugs in incorrect designs.

When starting the fifth exercise, the student should have a complete UVM testbench that is able to find bugs in misbehaving designs using coverage driven constrained randomized testing. So far, the DUT in the exercises had been an error-free FIFO, but in this exercise, the student's testbench design was used for checking other FIFO designs that introduce small bugs.

The other FIFO designs had been modified from the error-free FIFO by changing one code line to behave incorrectly and then the VHDL code was obfuscated. The obfuscation had been done so that the bugs could not have been easily seen by examining the DUT and the student had to use the UVM testbench for finding them. The code was obfuscated by replacing all the internal signal and variable names with a single letter and then removing all the line breaks.

The student was instructed to find out a single bug in two different FIFO designs. The errors might be for example data corruption, incorrect behavior of the status signals, incorrect reset handling or something else. There could be a possibility, that the student's testbench is not able to find the bugs. In this case, the assistant should point directions for improving the testbench.

After the exercise, the student should have experience of verification using a UVM testbench. He would have seen in practice, how the bugs in designs show up in the output of the testbench and analyzed the output data to discover the source of the problem.

6.3.6 Exercise 6: Verifying the decrypter

Exercise 6 was an introduction to horizontal reuse in UVM. In this exercise, the task was to create a completely new testbench for the decrypter module designed in the SystemVerilog exercises. The student was encouraged to reuse components from the previous exercises. The purpose of the exercise 6 was to give students more experience of UVM testbench design and finishing the exercise would not be crucial for completing the learning objectives of the education. All the learning objectives should have been met in the first five exercises. The student had a choice of using his own implementation from the SystemVerilog exercises as the DUT or downloading a working example module.

Because of the diversity in the students' background it was predicted, that some students would have more than enough time to finish this exercise and some would use the three training days for the first five exercises. To serve the purpose of giving everyone a situa-

ble amount of work the exercise was split in two options. There was a basic and an advanced version of this exercise, and the student was allowed to choose freely between them or if there was enough time, he could finish both of them starting from the basic one.

The aim in exercise 6 basic was to verify the decrypter module with a UVM testbench. The concepts introduced in the first five exercises should be otherwise sufficient for finishing the task, but because the DUT behavior utilizes handshaking signals, the student was instructed how the communication could be implemented in UVM. The driver can deliver the DUT's response back to the sequence for determining the next sequence item, or the sequence can be more abstract and the driver would implement the handshaking by reading the response of the DUT. Both the principles are valid, but the latter would be a better method because of its higher level of abstraction. Both options were explained to the student.

The advanced version of the exercise 6 was to verify the decrypter module wrapped inside the AMBA AXI4-Lite interface wrapper. The UVM environment would be specified with two agents. An active agent would drive the AMBA AXI4-Lite interface and a separate passive one would read the output signals of the DUT. For simplification, the student was advised to declare the passive agent so that it contains only a monitor instead of declaring a complete agent with the sequencer and driver switched off, because an active version of such agent that only interfaces the output signal would never be needed.

In addition, the advanced version of the exercise 6 contained an information block about response handling. Another hints given for the student contained a suggestion for a design order in which the base functionality should be done first one agent at the time and then the rest of the components are added one by one. The student was also advised to think if he needs multiple types of transactions with different levels of abstraction. Another question was if explicit bins could be practical in the coverage collector for example in the coverpoint for address signal, because the address in the design can either be valid, invalid but in the address range of the component or out of the address range and therefore not interesting. Therefore, covering all the values would not be important.

The horizontal reuse in the exercise was done by modifying the testbench used in the first five exercises to be used to verify a completely different design. Not all the components could be used in the new testbench directly because of the differences in the DUT signals and functionality, but the student gained an understanding of how much of the code had to be rewritten.

After finishing the sixth exercise, either on the basic or the advanced level, the student should be able to create his own UVM testbench design without any outside support and to verify his own design using UVM. He knows how the response of the DUT can affect the signals generated by the driver and what the possibilities are for handshaking signal

generation in UVM. Furthermore, the advanced exercise 6 introduced the student to the different roles for multiple agents in a UVM environment.

6.3.7 Exercise 7: Integration level testbench

A final additional work for the fastest performers was an introduction to vertical reuse with an integration level testbench in exercise 7. The student was instructed to examine and run a more complex testbench design including a hierarchy of multiple environments. The purpose of the task was to provide the student an overview of a real UVM system.

The complete testbench was acquired from an example in Verification Academy [18]. A block diagram of the complete system is shown in Figure 26. The environment integrates separate GPIO and SPI block level environments together to form a higher-level peripheral subsystem environment and adds a passive AHB bus agent. The example testbench also utilizes the register abstraction layer of UVM. [18]

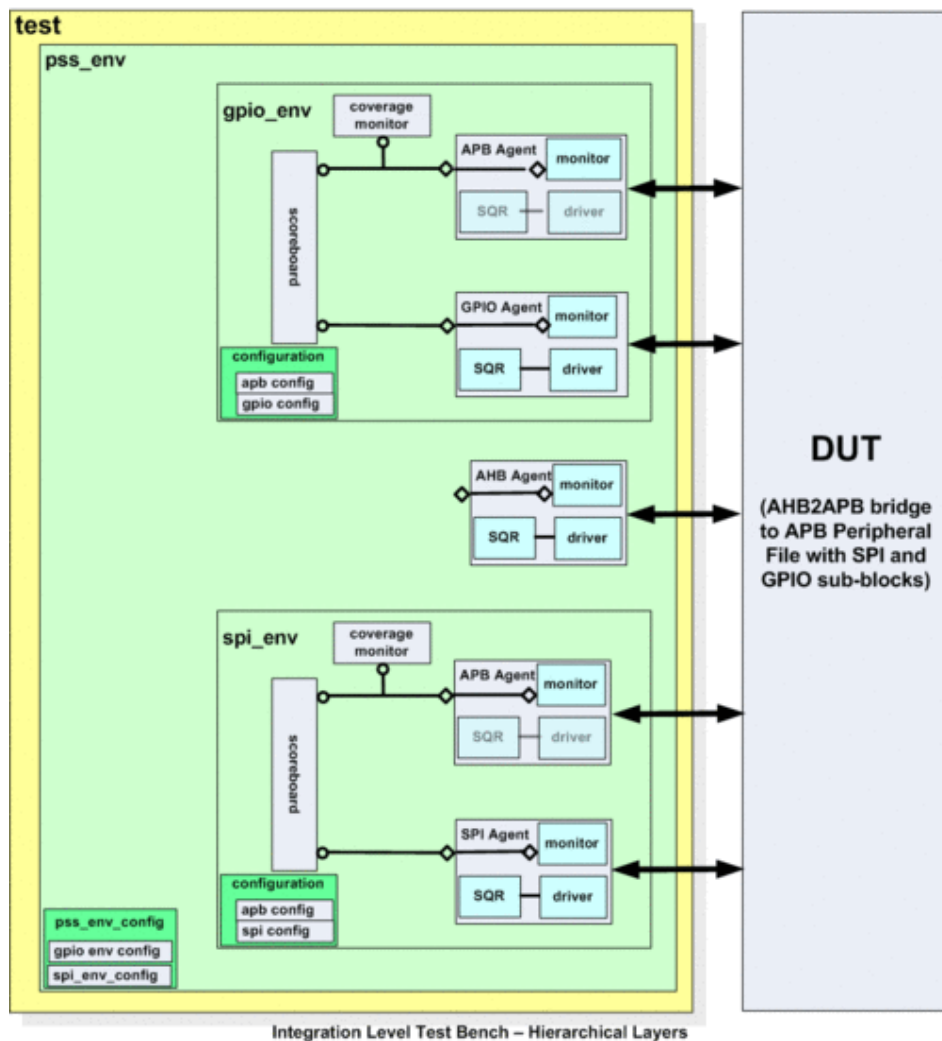


Figure 26. The integration level testbench. [18]

Both the block level environments could be simulated using block level tests and there was an additional integration level test for running the complete system. The student's task was to examine the testbench structure and see how the simulation output appears on different levels. The DUT had bugs, but the task description did not instruct to find corrections for them. A student already experienced in digital design could be able to debug the design using the test output data, but that was not required in the exercise.

The seventh exercise introduced the student to more advanced concepts in UVM and gave an example of real-world use of UVM. The student experienced an integration level testbench in action and encountered the use of register abstraction layer. He saw how the environments of different levels were encapsulated together and used to run in a higher-level test.

7. TRAINING SESSIONS AND RESULTS

The education was divided into two lecture and two training days for SystemVerilog and one lecture and three training days for UVM. Because of the structure of the exercises and the diversity in student backgrounds, guided exercise sessions were not an option. All the training was independent work, where the students were encouraged to discuss with each other and the assistant was present to answer questions and to help to solve the problems faced.

In the beginning of the first exercise days of SystemVerilog and UVM, the assistant showed the usage of tools so that everyone could focus more on the exercise tasks and less on the technical problems with the tool usage. If some unusually common problems were faced, they were discussed together.

7.1 Learning outcomes

Based on visual monitoring, all the students managed to at least start the exercise 4 in the SystemVerilog exercises and at least almost finish the exercise 4 in the UVM exercises. The progress was as was expected and everyone had enough exercises for the whole time while no one was in the situation that there was not enough time to learn the most important content.

There was an anonymous survey form for monitoring time usage in each exercise. The survey was not added until the second UVM exercise day, so there is no reliable monitoring data for SystemVerilog exercises. The students were asked to fill out an estimation later, but only two of them answered so it was considered that the results would not represent an estimation for the whole population. Therefore, only the data from UVM exercises is inspected in this section.

The students were informed to fill out the form after each exercise or in the end of the training even though they had not finished the task. The form asked for the exercise number and the time usage rounded to half hours. There was also a checkbox for marking that the exercise had been started but not completed and a question if the student worked alone or in a pair. The Figure 27 shows the completion of each exercise based on the survey.

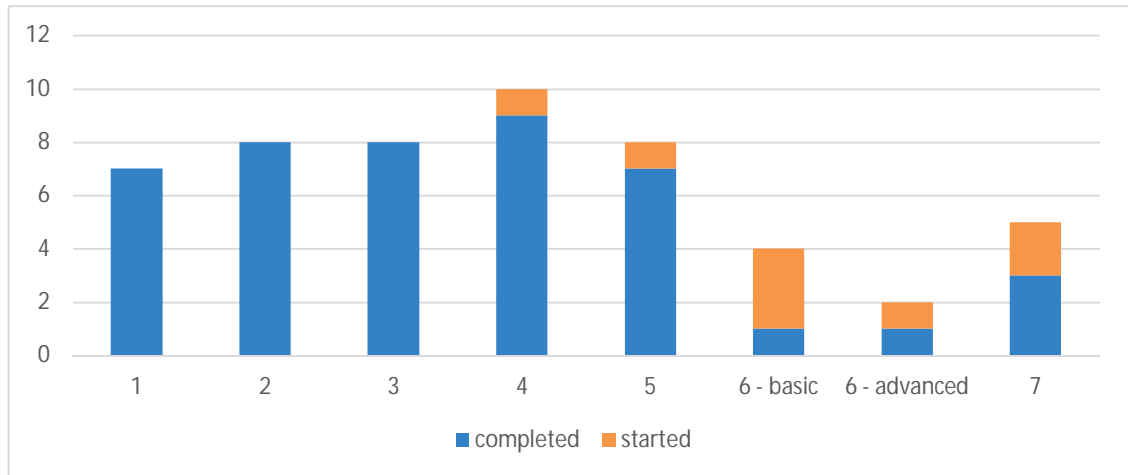


Figure 27. Completion of each exercise.

Based on the answers, it can be seen that not everyone participated in the survey after each exercise but those who did, managed at least to start the UVM exercise 4. The one answer, where the exercise 4 was not finished, reported that 8 hours were spent on the task. Based on visual monitoring, more than the 6 people reported were able to start the exercise 6 or 7. The exercise to be started after exercise 5 was the student's own choice.

The time usage for each exercise is shown in Figure 28. The figure shows the average time and the deviation. More data would be required for accurate assumptions, but the chart shows that the difficulty of the exercises increased from exercise 1 to 4. As the difficulty increased, more time was used on the task and the deviation of the time usage increased. The more experienced students had fewer difficulties in solving the task.

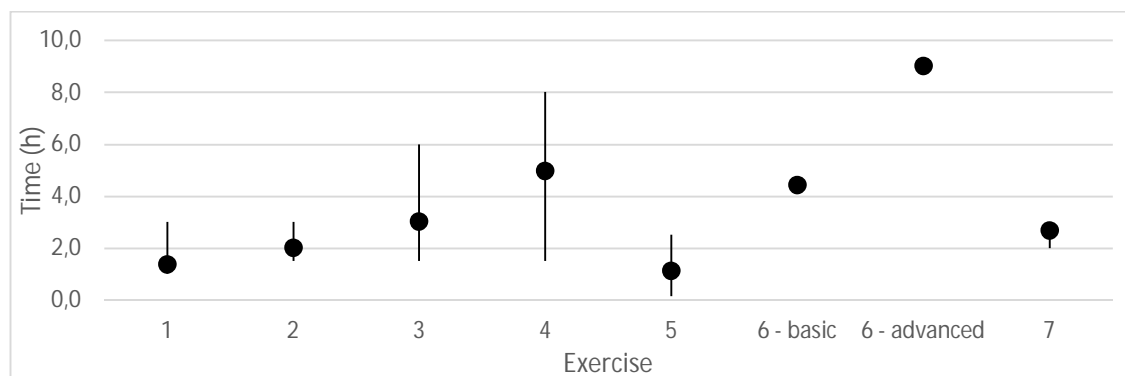


Figure 28. Time usage in each UVM exercise.

The chart only notices the students who were able to finish the task. As the choice between the exercise 6 and 7 was left for the student, the time usage reporting is divided between these tasks and a small population does not present an actual average. Still it can be seen that the advanced version of the exercise 6 required significantly more time. The exercise 7 did not require specific work to be done, so the students used the remaining time to inspect the integration-level testbench.

Another observation from the chart is that no task was too overwhelming, although the reports of exercise 4 taking 8 hours, of which one was not able to fully complete the task, should be taken into consideration. The design of the scoreboard was the most time-consuming part of exercise 4 and the randomization and new test declaration could be separated to for a new exercise to make the exercises more balanced.

7.2 Student feedback

The training office had feedback forms for the students to fill out twice per module. The first of the forms covered only the first four lecture days of the verification module and second one covered all the SystemVerilog and UVM training days and the one UVM lecture day. The students were asked to rate their overall satisfaction and more specific concerns on a scale of 1 to 5. In addition, there was a change for verbal feedback. A summary of the numerical feedback to both the feedback forms in the verification module is in the Table 5. The students had also an option to answer 6, meaning they were not present, to some questions, but that was omitted in the table because it was not available in all the questions and there were no such answers.

Table 5. Feedback summary.

	1	2	3	4	5	Total	Mean	σ
Please rate your overall satisfaction with the training on a scale of 1-5 (1 = very dissatisfied, 5 = very satisfied)								
Overall satisfaction (1/2)	0	2	5	1	1	9	3,11	0,87
Overall satisfaction (2/2)	0	0	1	8	0	9	3,89	0,33
On a scale of 1-5, please rate how well the intended learning outcomes were addressed during the training and how well the instructional strategies supported your learning (1 = very poor, 5 = excellent, 6 = I was not present)								
Learning outcomes (1/2)	0	1	1	5	2	9	3,89	0,97
Instructional strategies (1/2)	0	2	3	3	1	9	3,33	1,00
In total (1/2)	0	3	4	8	3	18	3,61	
Learning outcomes (2/2)	0	0	0	7	2	9	4,22	0,44
Instructional strategies (2/2)	0	0	6	3	0	9	3,33	0,50
In total (2/2)	0	0	6	10	2	18	3,78	
Please rate the instructors ability to provide you with a deeper understanding of the subject matter (1 = very poor, 5 = excellent)								
Trainer A (1/2)	0	2	3	3	1	9	3,33	1,00
Trainer B (2/2)	0	0	3	6	0	9	3,67	0,50
Trainer A (2/2)	0	0	3	6	0	9	3,67	0,50

The numbers 1/2 and 2/2 in parentheses indicate the feedback form. Trainer A was responsible for the lectures and trainer B for the exercises. Some of the students informed in the comments for the second feedback form that their grade is for the education as a whole and not only the second part. Out of 22 students, 9 returned the feedback form.

The results show that the students were more satisfied with the second part of the training, which covered all the exercises. The mixed overall satisfaction during the first lecture part of the module changed to a solid answer of 4 from almost every participant. In addition, the mean value of learning outcomes improved to a satisfactory level, which means that the exercises supported the training module well. The mean opinion on instructional strategies remained unchanged between the first and the second feedback form, but the deviation in the answers decreased.

As it has been pointed out in the verbal comments, the disappointment in instructional strategies was only a scheduling issue. Full lecture days without any hands-on experience between lectures can be very overwhelming whatever the subject. Many of the students pointed out in the verbal feedback, that theory and practice should be mixed more to make the education easier to follow. These kinds of education modules would be better arranged and easier to follow if the hands-on training was in shorter sessions between lectures, but this time it was not an option. The fact was acknowledged before the module, but there were practical problems with the training room reservations that forced the schedule. On the other hand, one student commented that the UVM education schedule with one full lecture day before practical training was a good choice, because it was easier to start the practical exercises after all the lectures.

The contents of the training received better feedback, with comments for example *“Target was good and contents was well absorbed”* and *“Good overview on design verification”*. The number of replies to the feedback form was only under a half of the number of the students and only a half of those who replied provided also verbal feedback.

One of the verbal comments that concerned the exercises and demands the most attention was *“UVM exercises were better than the SystemVerilog exercises. In SystemVerilog exercises we were required to create design(s) which took me a long time to do.”* It is true that the theme of the education was verification and not design. Still, SystemVerilog is a new language for most of the participants and the design portion was chosen because that way the students would get a better overview of the whole language, not only the verification aspects – a verification engineer would have to understand the design portion of the language as well. The assistant should have been more active so that everyone would have spent most of the time learning the language, not struggling with design issues. This is supported by another verbal comment *“I would have preferred more guided sessions, like showing what is a correct way to do the exercises after a while.”* The situation was helped by providing the students with example solutions via e-mail after the training sessions, but discussing good testbench design principles in the classroom would have been a good addition.

The only verbal feedback about the performance of the assistant responsible for the training sessions was *“The assistant could've been more resourceful, and practical experience*

on larger test environments was missing. Not able to answer all questions.” More practical experience on UVM would have helped a lot to provide the students with additional knowledge outside the scope of the exercises.

The structure of the exercises, especially the UVM part, received acclaim. To the questions “*What did you find most valuable and helpful during the training in terms of your professional growth?*” and “*Do you have any additional feedback or comments that you would like to share?*” one student commented that building the UVM testbench in steps helped to understand the general picture and two of them pointed out how the tasks progressed well from aided copying to more realistic ones. The amount of information provided seems to have been correct in all the exercises.

Overall, the practical training seems to have been the most educating part of the education module, as it was planned. The exercises were challenging enough but still not too hard to follow. Out of nine responses to the feedback form, eight answered that they would recommend this training to others. Still, because no one was completely satisfied there is a room for improvement if the same training was arranged for a second time. The most critical improvement would be to mix the lectures and exercises more.

7.3 Problems faced

Along the exercises, it became clear that some tasks were not declared clear enough in the exercise instructions. The biggest problems the students faced were mostly simple things in the first SystemVerilog and UVM exercises when the amount of new information was most overwhelming. Making an exercise instruction is always balancing between systematic instructions and encouraging student’s own thought process, and the final balance can best be achieved with experience. In the beginning when everything is new information even simple things can seem very complex, and when the student has some experience already, too thorough instructions can lead to following and copying them blindly without any learning.

In SystemVerilog exercises, when the task was to model the functionality of asynchronous adder, many had problems understanding the concatenation hints and did an unnecessarily complex solution, where they for example generated the carry signal with conditional clauses. However, the hints provided were only a suggestion of the simplest approach and because the main purpose of the exercise was to get familiar with the syntax, the exercise was more rewarding than it was designed to be.

A bigger fault in the SystemVerilog exercises was the AMBA AXI 4 Lite specification in the exercise 4. A picture depicting the timing of transactions would have helped to understand the specification much better and the incomplete description led to diversity in solutions. The picture was added later, but many of the students had done the exercise already by then so they were allowed to use whatever solution they had found. After all,

the main purpose of the exercise was not to understand clearly the whole AMBA AXI specification but train to make their own design following outside guidelines and a misunderstanding caused by an incomplete specification still requires the same thought process in design.

When learning a new programming language, step-by-step guides with single code lines to copy one by one give the correct syntax but complete example codes would aid in understanding the structure. Even a relatively simple complete code file that performs some completely different operation but shows the correct structure would help to clarify which code block goes where, when the language is unfamiliar. This was realized after a suggestion from a student during SystemVerilog exercises and answered by creating example codes with different designs during exercise session breaks. Links to the example code files were added to information boxes in exercise instructions. The added example codes were for simple simulation testbenches in exercise 1 and state machines in exercise 2. The problem only existed in SystemVerilog exercises, because the UVM exercises already had complete code files for reference.

In UVM exercises surprisingly many used a completely new handle for their first own transaction in exercise 1. Using a new handle for every transaction is not feasible in real-world solutions because the amount of handles will grow massive, and this was explained to the students. This was a good example of simple errors caused by an overwhelming amount of new information.

There was a small error in the preconfigured driver component of the UVM testbench that caused additional thinking work. The retrieval of a sequence item from the sequencer was done after waiting for a rising clock edge on which the sequence item should be transformed into pin activity in the DUT. Because the communication between the driver and the sequencer introduces a small delay and the monitor component done by the students in exercise 2 had no such delays, the monitor managed to read the state of the DUT's signals before the driver wrote new values to the input pins. Although both the components were working on the same clock cycle, the monitor output seemed to arrive one clock cycle later. The proper order of activity in the driver would be to first get the sequence item from the sequencer and then write the information to the DUT directly after the clock edge.

This problem was particularly exciting and the reason and fix for it was explained to the students during the training sessions. It was agreed with the lecturer, that if the exercises were reused later on another course, the problem should be left unfixed but documented for the course staff and then introduced later in the exercises. This way the students will see concretely how crucial the timing of the transactions really is.

The colors used in the exercise instructions were a surprising source of puzzlement. There was a large diversity in the brightness and gamma curves of the monitors in the class and

this lead to the situation where some students had difficulties to distinguish the colored info and code blocks from body text, because on some computers all the text had white background. The class monitors should have been calibrated and tested one by one or the color scheme should have been chosen more carefully to prevent this.

If these exercises were used later, a big improvement would be to tie the UVM exercises to a verification plan done by the students themselves before the exercises. This time the schedule did not allow concentrating on the verification plan otherwise than on lecture level, but planning which DUT features have to be tested and where to focus most would be educating. In addition, the improved specification for the design in SystemVerilog exercise 3, that the student has made according to the results of validation and verification, should be returned for examination.

8. CONCLUSIONS

The objective in this thesis was to design an effective exercise package that could be used in training of new verification engineers. The exercises would introduce the student to the SystemVerilog language and UVM methodology. The target was to cover key concepts of SystemVerilog both in design and verification perspective and to provide the student with understanding of UVM so that he would be able to design a complete UVM testbench using the common UVM components and methods by himself after the training. The amount of exercises was decided so that every student should have time to learn the most important content but a sufficient amount of additional exercises was provided for the fastest performers.

The planned exercise package was tested in a tailored education module arranged for a company. The exercises were implemented as independent work during five training days so that the assistant was present to answer questions and to help in problem solving. The results were monitored with a time usage survey and a feedback form that had an option of verbal feedback.

The time usage survey and visual monitoring during the exercise sessions showed that the learning objectives were fulfilled during the training and the difficulty level of the exercises was as planned. Every student managed to finish the most important content but many had time to proceed to the advanced tasks.

The students were also quite satisfied with the content and structure of the exercises according to the feedback. The verbal feedback praised the aspects that were most focused on when planning the exercises, especially progression from aided copying to tasks that demanded more thought process while still providing all the syntax help. On the other hand, they criticized all the problems that were known beforehand, mainly the division of lecture and exercise days during the training module so that the hands-on training occurred after multiple consecutive lecture days.

Overall, the exercise packet is considered ready for further use based on the experiences gained during the implementation of the training module, when little corrections have been done. The noted corrections were minor clarity issues in the exercise instructions. One of the further uses for the exercises could be a new verification course at Tampere University of Technology, as the demand for verification engineers will continue to increase and the previous verification course has not been arranged after 2013.

REFERENCES

- [1] Accellera Systems Initiative Delivers UVM 1.2 to IEEE for Standardization, Accellera Systems Initiative, 2015. Available (accessed on 23.11.2016): <http://www.accellera.org/news/press-releases/201-accellera-systems-initiative-delivers-uvm-1-2-to-ieee-for-standardization>
- [2] Members, Accellera Systems Initiative, 2016. Available (accessed on 23.11.2016): <http://accellera.org/about/members>
- [3] Universal Verification Methodology (UVM) 1.2 Class Reference, Accellera Systems Initiative, 2014, 929 p. Available (accessed on 23.11.2016): <https://verificationacademy.com/verification-methodology-reference/uvm>
- [4] Universal Verification Methodology (UVM) 1.2 User's Guide, Accellera Systems Initiative, 2015, 184 p. Available (accessed on 23.11.2016): http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf
- [5] G. Allan, M. Baird, R. Edelman, A. Erickson, M. Horn, M. Peryer, A. Rose, K. Schwartz, UVM Cookbook, Mentor Graphics, 2013, 591 p. Available (accessed on 23.11.2016): <https://verificationacademy.com/cookbook/uvm>
- [6] T. Alsop, VIP-TSC Standardization Update, Accellera Systems Initiative, 2010. Available (accessed on 23.11.2016): http://www.accellera.org/images/activities/committees/uvm/VIP-TC_standard_effort_update_Jan_2010.pdf
- [7] AMBA® AXI™ and ACE™ Protocol Specification, ARM, 2011.
- [8] M. Bartley, Migrating to UVM: how and why! 2010. Available (accessed on 23.11.2016): https://www.testandverification.com/wp-content/uploads/mike_bartley_snug_reading_2010.pdf
- [9] J. Bergeron, E. Cerny, A. Hunter, A. Nightingale, Verification Methodology Manual for SystemVerilog, Springer, 2006, 503 p.
- [10] J. Bergeron, Writing Testbenches: Functional Verification of HDL Models, Second Edition, Springer, 2003, 182 p.
- [11] H. Foster, Prologue: The 2016 Wilson Research Group Functional Verification Study, Mentor Graphics, 2016. Available (accessed on 23.11.2016): <https://blogs.mentor.com/verificationhorizons/blog/2016/08/08/prologue-the-2016-wilson-research-group-functional-verification-study/>
- [12] H. Foster, Trends in Functional Verification: A 2014 Industry Study, Mentor Graphics, 2014.

- [13] IEEE 1364-2005 Standard for Verilog® Hardware Description Language, Institute of Electrical and Electronics Engineers (IEEE), 2005.
- [14] IEEE 1800-2005 Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, Institute of Electrical and Electronics Engineers (IEEE), 2005.
- [15] IEEE 1800-2009 Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, Institute of Electrical and Electronics Engineers (IEEE), 2009.
- [16] IEEE 1800-2012 Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, Institute of Electrical and Electronics Engineers (IEEE), 2012.
- [17] A. B. Mehta, SystemVerilog Assertions and Functional Coverage, Springer, 2014, 356 p.
- [18] Verification Academy, Testbench/IntegrationLevel, Mentor Graphics, 2013. Available (accessed on 23.11.2016): <https://verificationacademy.com/cook-book/testbench/integrationlevel>
- [19] C. Spear, SystemVerilog for Verification, Springer, 2006, 301 p.
- [20] J. Virtanen, SystemC Exercise, Tampere University of Technology, 2015. Available (accessed on 23.11.2016): <http://www.tkt.cs.tut.fi/training/edutech2015/SystemC-koulutus/ex.html>