Development of a new verication environment for a GPU hardware block using the Universal Verication Methodology (UVM)

NIKLAS KARLSSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

LUND UNIVERSITY

EITM01 Degree Project in Electrical and Information Technology

Development of a new verification environment for a GPU hardware block using the Universal Verification Methodology (UVM)

Master Thesis

Author
Niklas Karlsson (19930809-5117)
tfr13nk1@student.lu.se

Erik Larsson, Main Supervisor LTH
erik.larsson@eit.lth.se

Pedro Araújo, Supervisor ARM
pedro.araujo@arm.com

Pietro Andreani, Examiner LTH
pietro.andreani@eit.lth.se

Project Start Date
2020-01-20

Project Completion Date
2020-06-07

Department of Electrical and Information Technology, Faculty of Engineering, LTH,
Lund Univeristy, SE-221 00 Lund, Sweden

Printed in Sweden
Tryckeriet i E-huset, Lund

POPULÄRVETENSKAPLIG SAMMANFATTNING

Uppfinnandet av den integrerade kretsen banade väg för dagens datorer, mobiltelefoner och all annan elektronik som är en självklar del av vår vardag. En integrerad krets består idag av hundramiljontals halvledarkomponenter som sitter ihop med varandra på ett så kallat chip. Komplexiteten och antalet komponenter på chipen fortsätter att öka vilket har förändrat utvecklingsprocessen. Ett chip som tidigare sågs som ett system utvecklas idag som separata block och monteras sedan ihop till det slutliga systemet. För att företag ska kunna hålla produktionstiden kort och kostnaderna låga har verifikation med åren blivit en allt viktigare del i utvecklingen av integrerade kretsar. Verifikation innebär att en krets kontrolleras så att den uppfyller kraven i specifikationen innan den skickas för tillverkning genom att simulera designen. Därigenom kan eventuella fel upptäckas och åtgärdas tidigt i utvecklingsprocessen.

För att verifiera kretsen används en verifieringsmiljö, även kallat testbänk, vars uppgift är att generera insignaler till designen och sedan samla in utsignalerna. Genom att analysera utsignalerna går det att avgöra om kretsen har korrekt beteende eller om fel behöver åtgärdas. I takt med att antalet komponenter ökar och kretsarna blir mer avancerade ökar också svårigheten i att på ett effektivt sätt utförligt verifiera all funktionalitet i designen. Ofta verifieras block först enskilt och sedan tillsammans med de andra blocken i kretsen. Många olika simulatorer och språk har utvecklats och förfinats för att möta behoven hos ingenjörer som utvecklar verifikationsmiljöer. SystemVerilog har blivit det dominerande språket men kompletteras oftast med en verifikations metodologi.

Det här arbete kommer att utveckla en ny verifieringsmiljö enligt Universal Verification Methodology för att undersöka hur denna verifierings metodologi kan användas när olika separat utvecklade block ska verifieras tillsammans. Ett hårdvarublock i en grafisk processor kommer användas och UVM standarden kommer att analyseras utifrån hur den påverkar strukturen på verifieringsmiljön och dess prestanda. Det kommer resultera i ett antal riktlinjer för hur metodologin ska användas effektivt för block integrerande verifiering generellt och verifikation av grafiska processorer specifikt.

ABSTRACT

The invention of the integrated circuit is a key milestone in the history of electronic circuits. Since its introduction the number of components on a chip have increased rapidly, making them more powerful and able to perform complex operations, but it has also changed the design process. Today different parts of a chip can be developed separately as Intellectual Property (IP) and then put together to form the final system.

Over the last years making sure that the design follows the specification, also known as functional verification, have become a key part of the development life cycle. Finding bugs early is crucial for keeping cost down and achieving time-to-market requirements. This means that as the complexity of the design continues to increase, the time needed to thoroughly verify it cannot follow the same line of increment. This has pushed engineers to come up with new tools and methodologies to improve the verification process.

The Universal Verification Methodology (UVM) is created by Accellera together with experts from electronic design automation vendors Synopsys, Mentor and Cadence. Its emphasis is on improving the development of verification environments by increasing interoperability and making it easier to reuse verification components.

This project will develop a new verification environment verification environment according to the Universal Verification Methodology (UVM) to investigate how it can be implemented when separately developed blocks are verified together. A hardware block with sub-components from a Graphics Processing Unit (GPU) will be used and the methodology will be analysed based on how it affects the structure and performance of the verification environment. The dissertation will result in a new implemented verification environment along with guidelines on how to possibly improve block-integrating verification in general and verification of graphics processor specifically.

ACKNOWLEDGEMENTS

This project was suggested by arm and originated from the need of a new verification environment for a hardware block responsible for a big part in the graphics pipeline. I want to thank the GPU Hardware Verification team for their support and especially Pedro Araújo, his endless support and availability to answer all my questions at any time was essential for completion of this project. I also want to thank Erik Larsson and Pietro Andreani for their guidance in scoping the project and finalising the documents.

Finally I am grateful for all friends that have been apart of creating unforgettable memories throughout my academic journey and my family for their continuous support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DUT Device Under Test.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HVL Hardware Verification Language.

IC Intergrated Circuit.

IP Intellectual Property.

RTL Register Transfer Level.

TLM Transaction Level Modelling.

UVM Universal Verification Methodology.

CHAPTER 1

INTRODUCTION

Intergrated Circuit (IC) designs are growing in size and complexity at the same time as companies want to keep production costs low and time to market short. For example, a typical microprocessor life cycle from exploration to start of production that before ranged between three and four years can now last less than a year [1]. This amounts to challenges to the verification process that need to both properly and quickly functionally verify the design and find the bugs as early as possible in the project. New tools and languages that help verification engineers are constantly being developed and improved. SystemVerilog with its object-oriented and random-constraint stimulus features is a powerful and popular language used by many companies. The problem is that creating testbenches that take advantage of those features is time-consuming and does not scale up or enable reuse across environments and projects. To address this issue verification methodologies that sit on top of the language are being adapted in testbench development.

The Universal Verification Methodology (UVM) is a hybrid of technologies originating from Mentor's AVM [1], Mentor & Cadence eRM [2] and Synopsys's VMM [3]. It has a base class library that is supported by all simulators, making the methodology portable. Another key feature is the well-defined class hierarchy that give each component a specific purpose and a distinct interface to the rest of the testbench. This strongly facilitate the creation of components that allow re-usability. The lowest layer in the hierarchy, also known as the agent, communicates with the Device Under Test (DUT) by converting transactions to pin wiggles and vice versa. It can include an configuration object to provide possibilities to control aspects of the agent. This allows for reuse of the same agent for different scenarios and the test writer can think at an transaction level, focusing on the functionality to be verified [2].

[1] Advanced Verification Methodology
[2] e Reuse Methodology
[3] Verification Methodology Manual

1.1 Problem Description

The rapid development in the electronic industry have changed the perception of a system and thereby the design process. With today's size and complexity different parts of the system are today developed separately and then put together to form the final circuit. As technology have continued to advance these sub-systems can today contain hundreds of millions of gates and be very complex by themselves. With this magnitude testbenches for each individual block are needed to thoroughly verify all features. However, the customer is not interested in an isolated module and all blocks must be verified together to make sure that the final system fulfil the functional specifications. Therefore multiple testbenches are needed and verification can take about 70% of the design effort and the number of verification engineers can be twice the number of RTL designers in the same project. The amount of testbench code can after project completion be up to 80% of the total volume [3]. As the testbench used for verifying a system with multiple blocks together will share functionality with those at unit unit level, reusing existing components has become one of the biggest opportunities to improve the verification process by reducing the development time and code size. It can potentially remove duplication of functionality and reduce testbench code maintenance. It can also provide improved internal visibility, making it easier to debug errors detected at system level [4].

Graphics Processing Units (GPUs) are no exception to the trends in the electronic industry and are becoming more and more complex with an increasing number of cores. Different parts of the GPU are today developed and verified separately before they are put together to form the final processor, making the verification challenges mentioned above a problem. This is exemplified by a testbench used for verifying a hardware block connecting separate sub-blocks in the arm Mali GPU. It covers a big part in the graphics pipeline and the testbench does not reuse components from module level, adding unnecessary overhead to the project when making updates or adding new features. Furthermore the current structure, that does not follow any standard verification methodology, is very demanding in memory consumption and require a lot of time and effort when it comes to code maintenance.

1.2 State of the art

The Universal Verification Methodology provide a set of standards of how to use the SystemVerilog language and build verification environments with reusability in consideration. There is a big emphasis on configurability, enabling projects to share components horizontally. Therefore UVM typically have been applied to create generic verification environments that include the core architecture and supports multiple configurations, for example for memory controllers and communication protocols [5][6]. When components are reused horizontally their role are unchanged by it having the same responsibilities, making the use-cases more predictable.

UVM also enable vertical reuse, but despite the potentially benefits consistent and efficient reuse is not achieved in all projects. Components may be re-used with a different role when blocks are verified together, creating unknown use-cases. There is a growing demand for better guidelines as the existing ones can be to theoretical and does not take testbench performance into consideration. Today simulations are often run on computing clusters with shared resources between verification teams. A slow simulation time and high memory consumption could not only affect the the process of verifying the own design, but also impact other parallel projects using the same resources. A verification environment at unit level will have more components than at block-integration level since different blocks then will stimulate each other. Therefore its important to be considerate when reusing components from lower level so that the structural benefit does not come at the price of a negative impact on the performance [7][8].

1.3 Project aims

This project aims to investigate how the Universal Verification Methodology (UVM) can be adequate used for block-integrating verification and for verifying GPU Intellectual Property (IP)s. A new verification environment that follows the UVM standard will be proposed and implemented for a hardware block responsible for a big part in the graphics pipeline. Potentially it could provide a lot of reusability from components used to verify sub-blocks, meaning decreased time and effort needed when making updates and adding new features. The project will also examine the effect the UVM standard has on the verification performance. The project will result in best practice guidelines for how to reuse unit level verification environments to improve the structure and performance of testbenches development for block-level verification.

1.4 Structure of the document

Chapter 2 gives a short background about the development of integrated circuit design and verification tools and methodologies. An overview of the SystemVerilog language is presented and the Universal Verification Methodology is discussed in more detail. Chapter 3 starts with an analysis of the hardware block and its current verification environment and discusses how the structure of the new testbench was derived. Then follows chapter 4 which presents the new verification environment and highlights the result of this project and finally chapter 5 discusses conclusions and further possibilities.

CHAPTER 2

BACKGROUND

The introduction of the integrated circuit started the rapid increase in number of components in a electronic system. Engineers have then continued to make transistors smaller and smaller to be able to fit more of them on a chip to make it perform more powerful and complex operations. To be able to take advantage of the possibility of more transistors on a chip its been essential to develop new ways of designing them.

2.1 Integrated circuit design

In the beginning designers would manually place components and connections when creating the circuits. This quickly became a bottleneck with the increasing number of components. The solution was placing and routing programs that allowed the user to specify the gate-level netlist and the tool would then decide on the location of the gates and the wires connecting them. It did not take long until this also proved to be too detailed work and synthesis tools were introduced to allow the designer to express the functionality in a Hardware Description Language (HDL). The tool then generates the netlist out of the Register Transfer Level (RTL) code written in the language. HDLs are now the preferred way to enter the design of an integrated circuit[9].

2.1.1 Hardware Description Languages

Compared to a standard programming language that is a way to code an algorithm, a Hardware Description Language is used to describe the layout of a circuit using words and symbols. Two well-known HDLs are VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Verilog.

When Verilog was created in the mid-1980s, the typical design size was of the order of five to ten thousand gates. Throughout the 1990s, the Verilog language continued to evolve with technology, and in 2001 new extensions was added to the IEEE standard. As design sizes continued to grow a need for new features and better collaboration opportunities between designers and verification engineers grew. The result was the SystemVerilog language that unifies several proven hardware design and verification languages into one *Hardware Design and Verification Language* HDVL. The design part of the language is discussed in the following section and verification features is discussed in section 2.2.2 [10].

2.1.2 SystemVerilog for Design

SystemVerilog can be used for RTL design and is an extension of Verilog, meaning that it has all the previous functionality together with new features. The first release in 2002 began with a version number 3.0 to show that it was the third generation of Verilog. Some examples of significant enhancements are interfaces, C-like datatypes, user-defined types and packages.

The addition of packages is leveraged from VHDL and allow global declarations. Verilog limited variables, functions and other design information to be declared and used within a module. A package can instead hold information common for multiple blocks and be accessed by importing or references using the scope resolution operator (::). This removes the need for duplicated declarations and increases reliability [10].

2.2 Integrated circuit verification

Avoiding bug escapes into silicon is today necessary for companies to meet budgets and keeping projects within the planned timeline. This generates a need to develop digital circuits that comply with the specification at the first try, finding all bugs before tape-out. Therefore verification, to model or simulate a portion or the whole system and analysing the result, is an important part of the development pipeline. Verification uses a testbench to generate input data, also known as stimuli, to the DUT and then captures the resulting output. An abstraction of this is shown in Figure 2.1.

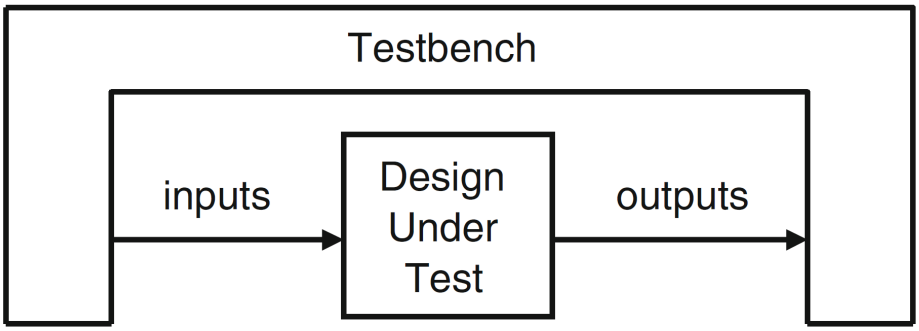

Figure 2.1: Generic Testbench [11]

Traditionally verification engineers have used direct testing. This means that after reading the hardware specification a verification plan with a list of tests is outlined. These test each concentrate on a set of features and have their respective stimulus vectors. The DUT is then simulated with these vectors and the result is determined by manually reviewing the log files and waveforms. This process can be improved by passing the stimulus vectors to a reference model that can compare its result with the output from the simulation to determine if the test was successful. Although this speeds up the process by removing the manual checking it still have limitations. Creating test vectors that cover all inputs in a big design is time-consuming, but the main issue with directed testing is that only the expected bugs are found. To properly twist and torture the design, random testing is needed.

Random stimuli can find scenarios that the verification engineers never anticipated and without spending time to create detailed vectors. However the stimuli should not be completely random, it should still be a meaningful stream of data. This is achieved by enforcing constraints over the stimuli and the inputs to the DUT is decided in a constraint solver. With constrained-random stimulus the output cannot be predicted and checked manually, a reference model is needed to be able to determine the success of the test. It is also important to keep track of what areas of the design that are visited, known as functional coverage. Setting up a testbench with constraint-random stimuli, reference model and coverage adds some up-front work, but is quicker than directed testing once up and running, which is illustrated in Figure 2.2 below. To limit this initial time needed its important to design testbenches with reusability and interoperability in mind.

In the last 1990s HDLs were used to simulate the design. They only had simple constructs for creating tests and the design size eventually outgrew verification capabilities. Therefore Hardware Verification Languages (HVLs) such as OpenVera and e were introduced, intended for verification only.

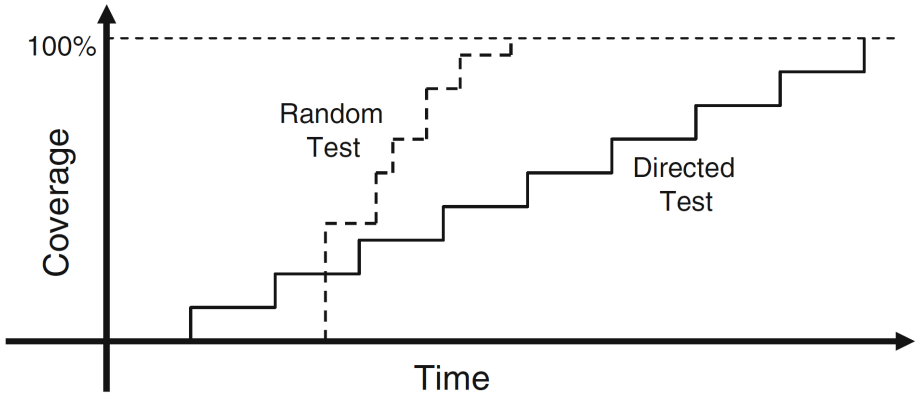

Figure 2.2: Directed vs Constrained-Random testing time progress [11]

2.2.1 Hardware Verification Languages

While a Hardware Description Language need to be synthesizeable into the actual layout of the circuit, a Hardware Verification Language (HVL) should provide stimulus to the design to make sure that is does what it is intended to do. This means that an HVL can be constructed differently to make it as easy as possible to build testbenches. Some typical features that are included in a HVL are constrained-random stimulus, functional coverage and Object-Oriented-Programming structures.

Verification is generally viewed as a fundamentally different activity from design and this split led to development of narrowly focused languages for verification. As mentioned previously in section 2.1.1 this started to become a bottleneck in terms of communication between the two groups and SystemVerilog was introduced with capabilities for both areas.

2.2.2 SystemVerilog for Verification

SystemVerilog introduces many datatypes that are useful for verification such as two-state variables, strings and classes with support for abstract data structures. Clocking blocks synchronise a group of signals on a particular clock and separates the time related details from other elements of the testbench. It helps in avoiding race condition by specifying a input and output skew a number of time units away from the clock edge the signals should be sampled or driven [11].

Although SystemVerilog have many good features and can be used to create powerful testbenches, a major challenge is providing scalable solutions. The size of the language can lead to huge diversity of testbench architectures, making them different across IP blocks and consequently hard to understand and to maintain. This also limits the possibilities of reuse since there is no shared understanding of best practice or conventions for verification components [12].

2.2.3 Verification Methodologies

The purpose of verification methodologies is to provide a manual with common set of standards to form a consensus of proper use of a verification language. It can be viewed as a blueprint for verification success and often include a base class library. The methodology should provide portable best practice skills that allow different domains to share knowledge and experience. An issue with the first methodologies was that they where specific to certain tool vendors [13].

2.2.4 Universal Verification Methodology

The Universal Verification Methodology (UVM) is a methodology for functional verification using SystemVerilog. Its provided as an open-source library directly from the Accellera website and it should be compatible with any HDL simulator that supports SystemVerilog, which means its highly portable. Another key feature of UVM is that it builds testbenches with reusability in mind. Changes to the DUT or wanting to apply different kind of stimuli should not engender big changes to the testbench [13].

A strength of the methodology is the strict class hierarchy, illustrated in Figure 2.3, leading to a separation between data (stimulus modulation) and components (structural parts of the testbench). Sequences and sequence_items that are used to represent stimuli are transient objects, meaning that they do not have a fixed simulation life-time and can be created and destroyed at any point. This brings a object oriented approach to stimuli generation that is more flexible. Traditional testbenches rely on being able to call sub-routines that exist as elaborated code at the beginning of the simulation.

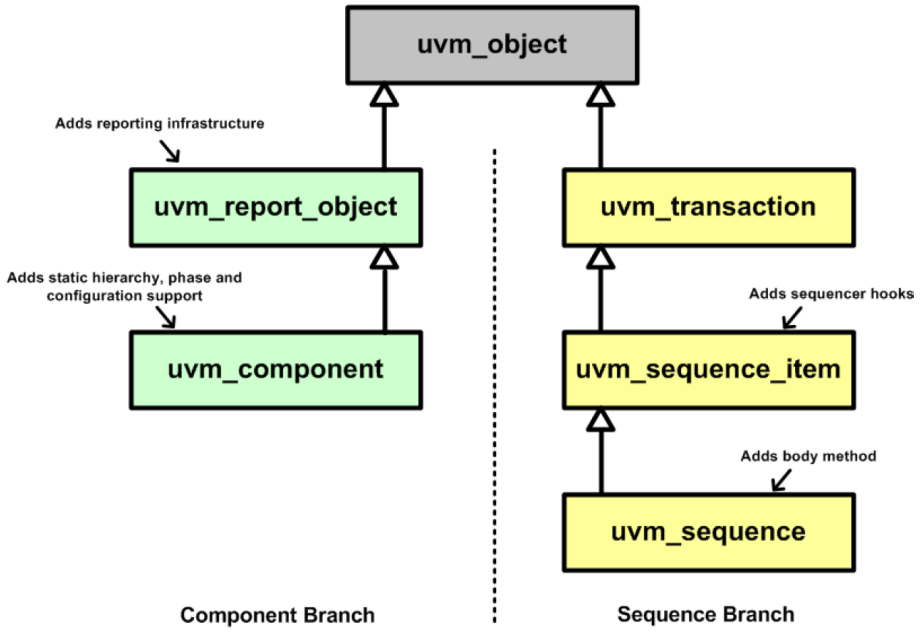

Figure 2.3: UVM hierarchy [2]

UVM also enforces a set of conventions concerning the life cycle of a testbench, know as phases. They order the major steps that take place during simulation and allow components to be developed in isolation since there is a common understanding of what should happen in each phase. The three main phases are

- Build phases - where testbench components are created, configured and connected
- Run-time phases - where time is consumed in running the testcase on the testbench
- Clean up phases - where the result of the testcase are collected and reported

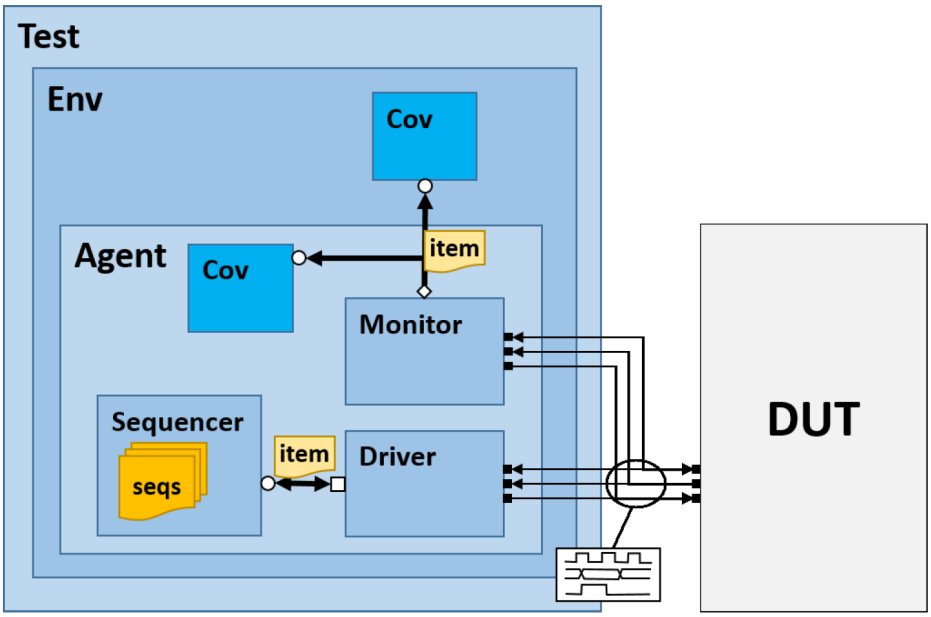

Figure 2.4: UVM Testbench [2]

Figure 2.4 shows an overview of an UVM testbench, highlighting the layers and typical components. At the lowest layer, closest to the RTL, the driver can in its *run_phase* requests sequence_items, also known as transactions, from the sequencer using the *get_next_item* method. The transactions contain the information that the driver need in order to interact with the DUT, mainly the values to drive on the connected virtual interface handle. The driver waits until the sequencer passes an item from its internal FIFO via the TLM ports and then converts it into pin-level signals. Once done, it signals back to the sequencer that the transaction object have been processed and can be destroyed.

UVM sequences are the containers that hold the stimuli. Their body method creates transactions and pushes them onto the sequencer FIFO. To execute the body method the sequence need to be started on a sequencer, either from a test case or from a virtual sequence. This allows full control of when and what type of transactions are created. The sequencer, also referred to as the stimulus generator, controls the flow of uvm_sequence_items generated by one or more sequences. When a request is received the sequencer selects an available sequence to produce the item and then forwards it to the driver.

The monitor listens to the communication on the interface and captures both the generated input stimuli and any output response coming from the RTL. It then converts the signal activity back to transaction objects and send them to compo-

nents higher up in the hierarchy through TLM analysis ports. The input data is needed by the reference model to predict the result and for coverage analysis.

The driver, sequencer and monitor are all static components extended from the root class uvm_component (see Figure 2.3). That means they inherit the defined phased test flow and are provided with interfaces to the factory and to the report handler. The three components are encapsulated in an agent along with optional configuration objects and functional coverage collectors. The configuration typically include an enumeration to set the agent in either active or passive mode. In active mode all sub-components are created and the driver and sequencer TLM ports are connected together. If configured passive, only the monitoring components are created since the agent does not drive any stimuli to the DUT in this mode.

The environment contains one or multiple agents, the scoreboard and a functional coverage collector. It should create all components in the build phase and define a default configuration. It can optionally retrieve configurations from a higher instance, for example the test, that override the default settings before it passes configuration objects to sub-components. The environment also need to connect different components, for example the TLM port of a agent monitor to the a port on the scoreboard.

The test is responsible for verifying specific features of the design and should create the environment in its build_phase. Different test scenarios can now be created with the same environment, meaning no changes to code in sub-components are needed, by overriding constraints, tweaking control knobs and enable or disable agents. The test also need to start the sequences, otherwise the drivers would never receive any transactions to drive. If multiple agents are used, a virtual sequence can be implemented to control the stimuli flow. It can be regarded as a container that in its body method can start multiple sequences on different sequencers. The test can start the virtual sequence on a virtual sequencer that contain handles to the agents sequencers. Finally there is a top module that instantiates the DUT, creates the interfaces and launches the global helper function *run_test*("*test_name*").

UVM provides an internal database where items can be stored under a given name and retrieved by some other testbench component. It can contain configuration settings to easily control testbench components without modifying the actual code. For example an agent can be changed between active and passive or its coverage can be toggled on or off. The uvm_config_db class is a convenient way to interact with the database. Its *set()* method will create a new or update an existing entry. The function parameters specify the scope where the data can be accessed and the name associated with the stored item. Items can be retrieved with the *get()* method with scope, name and variable to store the data in as parameters. The method returns false if no match is found which can be used to trigger faults or warnings [2].

2.3 Graphics Processing Unit

A Graphics Processing Unit (GPU) is an electronic circuit specialised to accelerate the process of image creation, especially in 3D graphics. Its job is to compute pixel colour and write the result into a buffer. This can to a high extent be done in parallel, meaning its beneficial to have a specialised unit for these operation. The process of turning a 3D model into what the screen later will display, also known as the graphics pipeline, consist of four steps.

- Application - the software loads primitives into the hardware
- Vertex Processing - vertices transformation
- Rasterization - transformation to 2D screen space
- Fragment Processing - compute the final pixel colour

In the application step changes are made to the scene. These changes can be triggered by the user moving the direction of view or interacting with objects. Every object in the scene are made out of primitives, usually triangles, combined to form complex structures. These primitives are loaded into the vertex processing pipeline step.

A vertex contains the spatial position of a triangle corner together with attributes such as colour and texture coordinates. Object are usually defined in a local coordinate system and during the vertex processing step vertices are processed and transformed from object space to world space to place multiple objects together. Rasterization then takes the image described by the different objects and coverts the three dimensional primitives into a two dimensional screen position called fragment.

Finally the fragments are processed with a number of operations to compute the final pixel colour on the screen. Depth calculations are performed to check if an object is behind or in front of something. If an object is obscured and will not be visible in the final image, the processor can potentially optimise the rest of the processing. Shaders are used for the colour calculations, using the fragment attributes together with lighting and reflections from surroundings. Since there are a huge number of objects in a screen a blender can be used to mix colours [14].

Traditionally desktop computers and consoles uses immediate mode rendering, meaning that the GPU draws the entire frame at once. However, there is an alternative where the image is divided into a grid and each section of the grid, also knows as tile, are rendered separately. This is used by many mobile devices to reduce the amount of memory and bandwidth needed. An example is the arm Mali series where the tile size is 16x16 pixels. A benefit with this approach is that the portion of the framebuffer for the tile currently being processed can be stored on chip. This provides quick access when performing blend or depth operations in contrast to fetching values from external RAM [15].

CHAPTER 3

APPROACH AND METHODOLOGY

The project starts with a study of the hardware block to understand its functionality. From there the current verification environment will be investigated to see how connecting block are currently being simulated and stimuli provided to the DUT. Parts that are hard to maintain and are heavy on computational resource will be identified as important to replace. Then sub-block testbenches will be examined for verification components that potentially can be reused in the new structure.

3.1 Analysis of the hardware block

The hardware block handles the fragment processing part of the graphics pipeline, meaning its job is to calculate the final pixel colour. It can be considered as a system with three sub-blocks that each have their own task. The responsibility of the system is to handle the connections and control logic with the rest of the GPU and between the different tasks. Figure 3.1 shows the fragment processing block and highlights the three main blocks connecting to it.

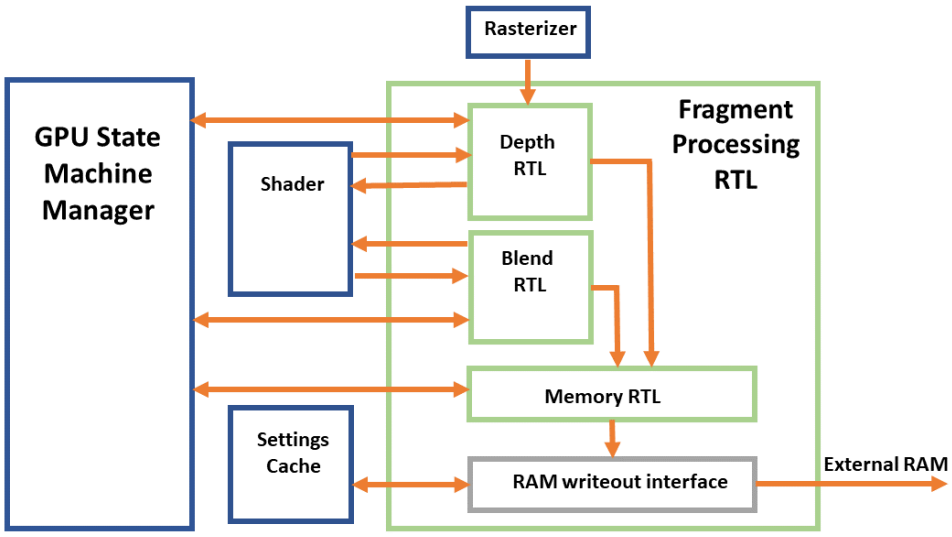

Figure 3.1: Hardware block diagram

The GPU State Machine Manager is responsible for controlling the flow of tiles and to keep track of when they are done processed. The rasterizer performs transformation on the fragments in the tile before they are grouped and sent to the depth block. There depth calculations, to determine the how far away from the camera position a fragment is, are performed either before or after passing them to the shader. The result of the calculations is then passed to the depth buffer. The blender receives colours from the shader and can potentially blend them with colours in the buffer and then the buffer is updated with the result. The memory block holds copies of the buffers for the portions of the screen that is currently being processed and writes them back to main memory once all fragments have been processed.

All blocks involved in the fragment processing pipeline step need settings to know when and how to perform its calculations. It could be to see how depth testing should be enabled or how to perform the blend operation. To provide quick access to the settings they are loaded in to a cache when requested.

3.2 Analysis of the current Verification Environment

The testbench should generate tiles and pass it both to the fragment processing block and a reference model. It also needs to capture the RTL output when the buffers are written to external RAM and compare it to what the model predicted. To help with the control flow the verification environment includes the GPU State Machine Manager as an RTL block. It needs to be stimulated with the settings and the generated tiles to make it output signals to trigger different tasks in the other blocks. In the full GPU the fragment processing block receives its inputs as fragments from the rasterizer. Therefore the tiles need to be transformed which is done in the rasterizer driver. The driver needs the tile positions as inputs and then outputs groups of fragments to the depth block. Both the depth and blend block expects communication with the shader that have its logic replicated in a driver. The settings cache is also simulated with a driver that need to be able to receive requests for a specific index and return the setting, or return a index along with the setting if the item was not already in the cache.

The testbench drivers are big monolithic components that spawns multiple processes to drive the data trough the different sub-blocks. They do not have sequences and receive data through big transaction objects. The stimuli is then pass to the DUT through non standard UVM interfaces. An overview of the active part of the testbench is shown in Figure 3.2.

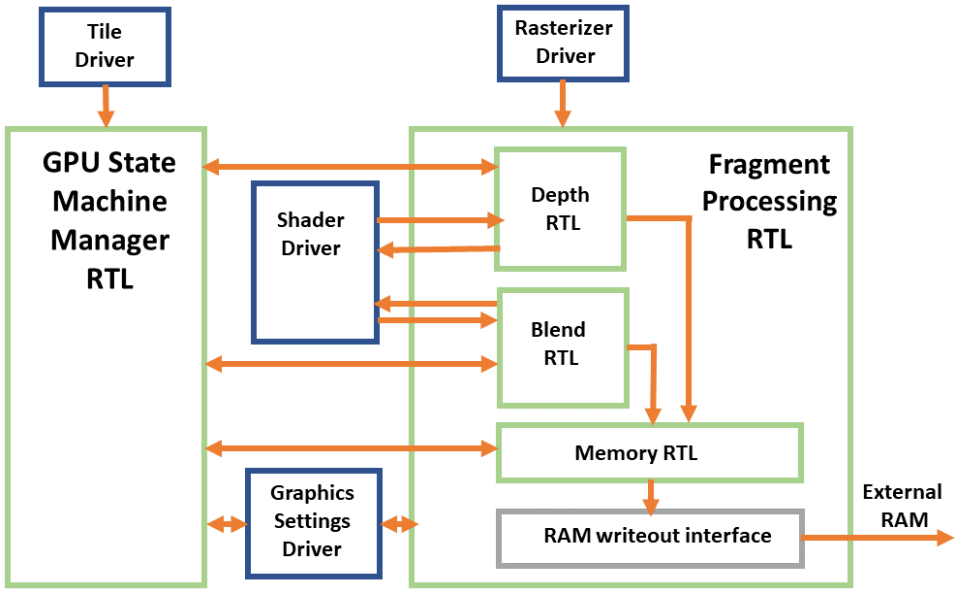

Figure 3.2: Active part of old testbench

The lack of sequences means that in order to generate stimuli to the drivers each test need to extend all transaction classes. A memory analysis was performed to investigate the memory consumption. The testbench was compiled with two different testlists, one containing multiple tests and the other only a single. A measurement was made at simulation time 0, before stimuli objects are generated and put on the heap, to capture the minimum amount memory needed by the testbench. The same analysis was made on a testbench that follows a UVM approach to compare the difference. Another drawback with the current test structure having verification components directly inside the test class is that any changes to the DUT will require updating all test files. This can be time consuming and does not enable reuse or configurability of the testbench since the test rely on a specific structure.

The depth RTL is a complex block and its difficult to exercise all its functionality from a system point of view. Therefore it has its own verification environment to properly verify the depth calculations. The interaction between the fragment processing block and the tiles is complicated, even with the extra included RTL, but the GPU State Machine Manager also have a unit level verification environment that potentially could facilitate the interactions. These two environments were analysed to investigate if they have components that could be re-used in the new fragment processing verification environment.

3.3 Analysis of sub-block testbenches

The Depth testbench should generate fragments as inputs and capture what the block writes to the memory buffers. An overview of the active components can be seen in Figure 3.3 below.

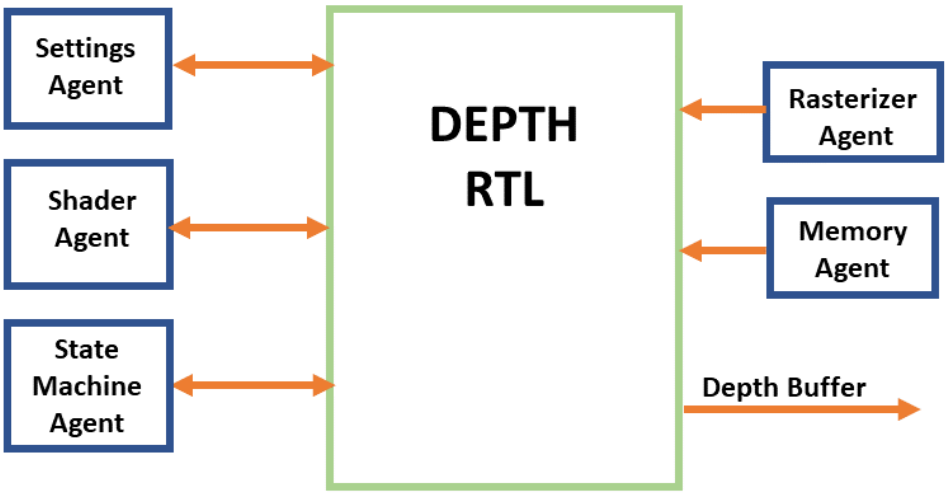

Figure 3.3: Active part of depth testbench

In the graphics pipeline the depth block receives its inputs from the rasterizer. The testbench therefore includes an agent to control the flow of fragments into the DUT. The agent does not create any transactions, instead the data is generated in testcases and passed to the driver via a transaction level modelling fifo. All settings needed for the depth block is modelled in an verification component and likewise the shader behaviour. It also communicates with the state manager and memory buffers that have their respective agents to stimulate the RTL.

The analysis of the Depth verification environment show that it contains a lot of the functionality needed for verifying the whole fragment processing block. Having two distinguished testbenches result in duplication of work when adding new features or making changes in for example shader calculations or depth settings. Reusing these components in the new verification environment could reduce the project overhead and improve the development process.

The GPU State Machine Manager have a unit level testbench to test state transactions. It should verify that the RTL block can receive tiles, output control signals for other blocks to start processing and monitor their responses. An illustration of the active components is shown in Figure 3.4

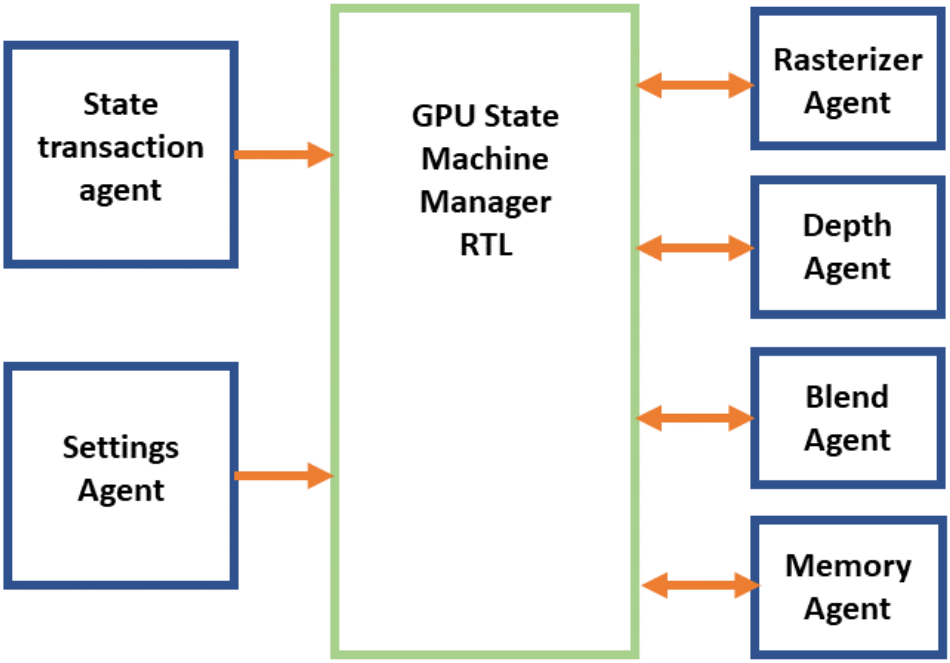

Figure 3.4: Active part of the GPU State Machine Manager testbench

The flow of tiles are driven by the state transaction agent. The testbench implements a virtual sequence to drive multiple different sequences on the agents sequencer. This quite complex stimuli generation is also needed in the fragment processing testbench and could be reused to avoid duplicated functionality along with the rasterizer agent to capture the tile positions and the settings agent to simulate the cache.

3.4 Development of the new Verification Environment

After analysing the current testbench and the sub-block environments the following areas were identified as the ones in most need of improvements.

- Smaller testbench components with a clear interface to the DUT
- A new test structure to reduce memory consumption and facilitate updates
- Reusage of verification components to decrease project overhead when adding new features

The development of the new testbench started with disabling the old drivers and creating a new structure that follows the UVM. The old top module that defines testbench signals and instantiates the RTL blocks can be reused, with the addition of the agent specific interfaces. A test base class is created that holds the common standard setup. Test cases for specific scenarios can then extend this base and override configurations to create different testbenches. A virtual environment is created to instantiate the Fragment Processing, GPU State Machine and Depth environment. The motivation for having a virtual environment and not creating the sub-environments inside the Fragment Processing environment is discussed in chapter 5.

A virtual sequence and virtual sequencer is implemented to start the sequences in the correct order to match the graphics pipeline flow. Ideally the virtual sequence would extend one from the GPU State Machine Manager testbench to reuse the logic for generating and driving tiles. However, that virtual sequence test a high number of state transactions to thoroughly verify the RTL. This would risk wasting computational resources on things not needed for verifying the fragment processing block. Therefore a new virtual sequence is created with simpler state transactions. On the other hand, the state transaction, settings and rasterizer agent is reused from the GPU State Machine environment. Other verification components, such as the blend and depth agent, that are now stimulated by the design are configured to passive or disabled completely. Likewise the Depth environment is configured with the settings, shader and rasterizer agent active and the rest set to passive.

Since the Depth environment only need fragments to verify its calculation, the new Fragment Processing testbench need an agent to convert the generate tiles to fragment and load into them into the rasterizer agents fifo. This new agent, called *Tile to Frag agent*, can take advantage of that the GPU State Machine Manager testbench captures the generated tile positions for the next tile to process in its rasterizer agent. The new environment also need settings for the blender and memory since they are not generated by the sub-environments. This agent is smaller and simpler than the old big driver, but imposes a challenging aspect of UVM when it comes to creating transactions on top of transactions for reuse across multiple layers. Different settings are modelled as individual transactions and some are used by multiple components, for example settings that act on the entire frame. This is resolved by creating a main setting transaction inside the virtual sequence every time a frame is started. This transaction is written into the resource database for other components to gain access to it. To resolve constraints between different transactions not being available upon creation SystemVerilogs *post_randomize*() is used. An illustration that highlights the reused and newly implemented active components in the new verification environment is shown in figure 4.1.

3.5 Analysis of the new Verification Environment

Although the sub-environments allow for reuse of verification components there are things in the testbenches that are not used at integration level since RTL blocks now provide the stimuli. The impact on the testbench size when having non-used components was investigated by analysing the number of classes and objects in different configurations at the start of simulation.

CHAPTER 4

RESULT

4.1 Test structure analysis

The result from the test structure analysis discussed in section 3.2 are showed in Table 4.1 below. To configure the analysis a fatal message was added at the start of a test in the old fragment processing testbench to abort the test at simulation time zero. The simulator captures the memory needed by the structural parts of the testbench, refereed to as the minimum amount of memory needed, and outputs it into a log file. The test was then compiled and executed, once with a full testlist and once with only the basic test. The same seed was used to ensure the same setup in both cases. The UVM testbench used for comparison followed an identical setup, using a basic test with an abort trigger at simulation time zero. The test was compiled and executed with both a full testlist and only containing the single test, using the same seed in both cases.

Table 4.1: Memory for different test structures

Test structure	Full test list	Single test
Old Fragment Processing testbench	699 MB	605.4 MB
UVM structured testbench	211 MB	211 MB

The result show a big difference in structural memory needed depending on the testlist size for the fragment processing testbench. Having components and transactions directly in each test consumes memory for every test case, which explains the reduced memory for a shorter testlist. In contrast the UVM testbench is unaffected by the testlist size. Note that this is not a comparison between the old and new verification environment, instead the results highlight the need of a test structure extended from a test base class and transactions created inside sequences which was implemented in the new testbench.

4.2 New Testbench Structure

The resulting new verification environment for the fragment processing block follows the Universal Verification Methodology and the active parts are showed in Figure 4.1 below. The new structure is expected to improve maintenance and reduce development time since the identified problem areas with the old testbench have been resolved. The big drivers that spawned multiple processes that were tangled together have been replaced with smaller verification components with a specific purpose, making them easier to understand. The reuse of agents from sub-environments removes the duplication of already existing features and changes are no longer needed in multiple places when making updates. Also, the introduction of phases makes the testbench flow easier to understand.

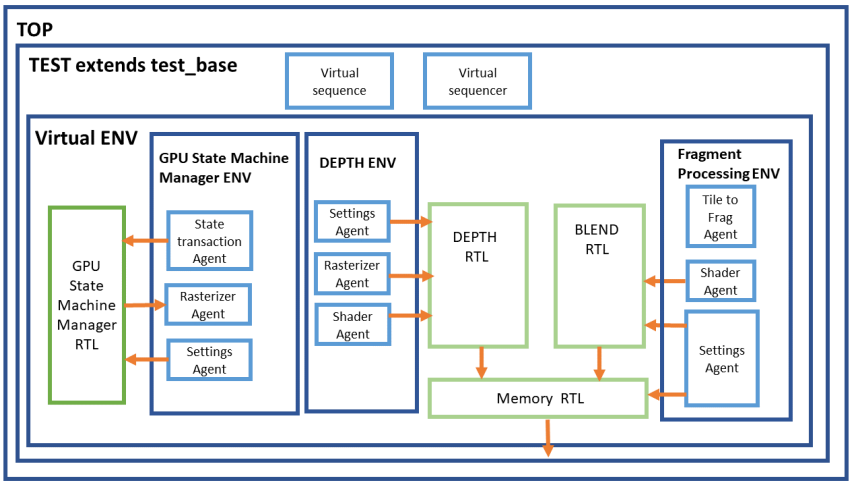

Figure 4.1: Active part of new testbench

A basic test with the default configurations and constraints is used to exercise the testbench. To inherit the standard setup the test extends the test base and calls the super method in its build and connect phase. Thereby the virtual environment is created and the agents showed in Figure 4.1 are configured to active while the rest of the components in the sub-environments are passive. To start generating stimuli the test starts the virtual sequence on the virtual sequencer in the *run_phase*. The virtual sequence is essential for simulating the pipeline flow of the graphics processor. It creates the tiles and settings needed for multiple components and the starts the sequences in the correct order.

4.3 Testbench size analysis

As discussed in section 3.5 reusing the sub-environments will include many components not needed to actively generate stimuli to the DUT. The analysis of unutilized agents affect on testbench size was performed by adding a *uvm_fatal* report macro to the start of the *run_phase* of the basic test. Thereby the test will abort after creating and connecting all structural components and the the result is not dependent on any stimuli randomisation or number of transactions created. Three different configurations was then analysed. In the first configuration only the driving agents that can be seen in Figure 4.1 are constructed. This is compared with having all components from the GPU State Machine environment created, with the non-driving agents in both active and passive mode. Table 4.2 shows the number of objects and classes that were created in the different configurations.

Table 4.2: Testbench size for different configurations

Configuration	Total Objects	Classes
Only driving agents from GPU State Machine ENV enabled	112 111	30 172
All GPU State Machine ENV agents enabled	114 377	30 794
All GPU State Machine ENV agents enabled and active	117 955	31 776

The analysis shows that unutilized components from sub-environments affect the testbench size, adding up to 5% extra objects. This difference is expected to grow as more environments are included, for example if the depth environment also was configured in the different settings. Moreover, having many monitoring components risk affecting the number of run-time objects and memory consumption further. Methods to keep the size to a minimum is discussed in chapter 5.

CHAPTER 5

CONCLUSION

This project have showed that following the Universal Verification Methodology when building testbenches for block-level integration and graphics processing verification have significant structural benefits. Using smaller components with a clear purpose makes the testbench easier to understand and re-usage from sub-block environments facilitate maintenance and reduce project overhead.

The test structure that was identified as a key area for improvement is resolved by the test_base class having the common build and configurations that test cases can inherit from. A test writer can build different testbenches without rewriting the whole test by changing configurations or overriding constraints. The importance of the test_base grows when reusing sub-environments for integration level verification since the number of components is increased. The observed high memory consumption caused by the test structure is also resolved by the test base and having transient stimuli that is generated inside the agents sequences, instead of statically elaborated code from big transactions classes. The run time memory is also expected to improve since transactions are created only when requested and destroyed when processed instead of being extended by all tests at the start of simulation.

The usage of a virtual environment is motivated with further levels of integration in consideration. The fragment processing testbench can be reused at a higher level without having to configure the depth and state machine environments, which have had been the case if the new environment created the sub-environments directly. Not being able to reuse the virtual sequence from the GPU State Machine environment is a weakness in the new testbench. This could have been resolved by having a base virtual sequence that both could be inherited by the fragment processing testbench and extended at block-level to add more specific functionality. However, at integration-level verification its important to have in mind that SystemVerilog support single inheritance, meaning that the new virtual sequence only can inherit from one sub-block.

However, its important to have a couple of things in mind when building integration testbenches according to the UVM. Replacing a few big drivers with multiple smaller components and including complete sub-environments can have a negative impact on performance if not implemented carefully. As Table 4.2 shows its important that the verification engineers designing unit-level testbenches provide the option to not only configure an agent passive, but to disable it completely. This may seem as unnecessary work at the time since that testbench will use all components, but is important for keeping the number of objects to a minimum at a higher levels of verification.

Another drawback with an increasing number of components is that the resource database can easily get populated with a high number of entries, making the processes of finding an item slow. This could be optimised by writing a wrapper interface to the database and then writing the specific interfaces to the agents in the environment only if they are enabled. The same can be done for the configuration objects. The testbase can create one configuration object containing the component configurations for the entire environment and only writing the a agent entry in the database its going to be used.

Contemplating these conclusions the Universal Verification Methodology can be used to build testbenches that reduce project overhead without affecting the verification performance.

REFERENCES

[1] W. Chen et al. "Challenges and Trends in Modern SoC Design Verification". In: *IEEE Design Test* 34.5 (2017), pp. 7–22.

[2] Mentor Graphics. *Universal Verification Methodology UVM Cookbook*. Verification Academy, 2016.

[3] Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.

[4] Brian Bailey and Kathy Werner. *Intellectual Property for Electronic Systems: An Essential Introduction*. Intl. Engineering Consortiu, 2007.

[5] A. El-Yamany, S. El-Ashry, and K. Salah. "Coverage Closure Efficient UVM Based Generic Verification Architecture for Flash Memory Controllers". In: *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2016, pp. 30–34.

[6] Pedro Araújo. "Development of a reconfigurable multi-protocol verification environment using UVM methodology". MA thesis. Faculdade de Engenharia da Universidade do Porto, 2014.

[7] Mark Litteric. *Vertical & Horizontal Reuse Of UVM Environments*. DVCON Conference. 2015.

[8] Janick Bergeron et al. *Verification methodology manual for SystemVerilog*. Springer Science & Business Media, 2006.

[9] Louis Scheffer, Luciano Lavagno, and Grant Martin. *Electronic Design Automation For Integrated Circuits Handbook*. Boca Raton, FL: CRC Press Taylor, Francis Group, 2006.

[10] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[11] Chris Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.

[12] Nisvet Jusic and Jan Nilsson. *Design and verification languages*. Dec. 2007.

[13] J. Bromley. "If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language". In: *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. 2013, pp. 1–7.

[14] Edward Angel, Dave Shreiner, et al. *Interactive computer graphics: a top-down approach with shader-based OpenGL*. Boston: Addison-Wesley, 2012.

[15] *Graphics And Gaming Development — Tile-Based Rendering – Arm Developer*. <https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/learn-the-basics/tile-based-rendering/single-page>. Accessed: 2020-04-08.

Printed by Tryckeriet i E-huset, Lund 2020

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-766
http://www.eit.lth.se