Bachelor's thesis

Information Technology

Embedded Software

2012

Oliver Huuhtanen

MOBILE GRAPHICS TESTING

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Embedded Software

Completion of the thesis 5/2012 | Total number of pages 39

Instructor: Jari-Pekka Paalassalo, Lic.Tech., Principal Lecturer

Author: Oliver Huuhtanen

MOBILE GRAPHICS TESTING

Mobile devices require more graphical processing power every day, which means graphics processing units need to keep up with the evolution. As graphics processors evolve and new functionalities are implemented, testing becomes a bigger part of the development process. Automation has become more important for testing to make the testing process faster and thus less resource consuming.

The objective of this thesis was to add support for a new company wide vector format to a testbench. The support was added to two older Qualcomm Incorporated graphics core generation testbenches. The reason for adding this support was to help solve future support issues that might arise for these graphics cores.

Adding the support for the new vector format required the implementation of a new way of processing stimulus in the testbenches. The implementation was done iteratively, a small piece at a time. Thus, first a single iteration was implemented and tested and only if it passed was it possible to move on to the next iteration. After all iterations had been implemented, a test set was run through the testbench.

Support for the new vector format was successfully added to the testbenches, which now enables the use of both vector formats on these testbenches. This could be declared when a predefined test set had passed successfully.

KEYWORDS:

Testbench, test, vector, graphics core

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ
TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Sulautetut ohjelmistot

Opinnäytetyön valmistumisajankohta 5/2012 | Sivumäärä 39

Ohjaaja: TkL Jari-Pekka Paalassalo

Tekijä: Oliver Huuhtanen

MOBIILIGRAFIIKAN TESTAUS

Mobiililaitteet vaativat päivä päivältä enemmän laskentatehoa. Grafiikkasuorittimien on pysyttävä tämän kehityksen mukana, jotta kuluttajan vaatimukset täyttyvät. Grafiikkasuorittimien kehittyessä ja uusia toiminnallisuuksia toteutettaessa, testauksesta tulee suurempi osa kehitysprosessia. Automatisoinnista on tullut tärkeä osa testausta, jotta testausprosessi nopeutuisi ja täten voimavarojen käyttöä pystyttäisiin vähentämään.

Tämän opinnäytetyön tavoitteena oli lisätä uuden, koko yhtiön laajuisen vektoriformaatin tuki koestuspenkkeihin. Tuki lisättiin kahteen Qualcomm Incorporatedin vanhemman grafiikkaydinsukupolven koestuspenkkiin. Syynä lisäykselle oli näihin grafiikkaytimiin tulevaisuudessa mahdollisesti ilmentyvien tukitehtävien selvittämisen helpottaminen.

Uusi vektoriformaatti sisältää uuden heräteformaatin. Tämä heräte tuli käsitellä uudella tavalla koestuspenkissä, jotta se tukisi uutta vektoriformaattia. Toteutus tehtiin pieni osa kerrallaan iteratiivisesti. Jokainen iteraatio testattiin ja vasta kun se oli läpäissyt testin voitiin siirtyä seuraavaan iteraatioon. Kun kaikki iteraatiot oli toteutettu, koestuspenkin läpi voitiin ajaa testisarja.

Uuden vektoriformaatin tuen lisääminen koestuspenkkeihin onnistui, mikä mahdollistaa molempien vektoriformaattien käytön näissä koestuspenkeissä. Tähän tulokseen päästiin, kun ennalta määritelty testisarja oli ajettu ja ajo suoriutui onnistuneesti.

ASIASANAT:

Koestuspenkki, testi, vektori, grafiikkaydin

CONTENT

FIGURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

AHB	Advanced High-performance Bus
RTL	Register transfer level
API	Application Programming Interface
AXI	Advanced eXtensible Interface
HW	Hardware
SW	Software
ARM	Advanced RISC Machines
RISC	Reduced instruction-set computer
GPU	Graphics processing unit
SoC	System on a chip
DUT	Device under test

1 INTRODUCTION

Mobile phones are an ever-growing commodity, which have evolved enormously in the past two decades. From brick sized phones with a monochromatic 48 x 84 pixel display to palm sized phones with a 960 x 640 pixel touchscreen display. With these improvements the processing power of mobile phones has increased significantly and these days mobile phones have a separate Graphics Processing Unit (GPU) to enable flawless 3D graphics to the display. As smartphones get more and more popular the technology needs to advance at a steady pace to keep the consumers happy. This means that GPUs need to steadily get more and more powerful.

The increasing new requirements for the GPU require more features to be implemented and more importantly these features need to be tested. Testing is a very time consuming part of a development process and therefore usually very costly. For testers the objective is to increase the automation of testing to decrease the amount of manual labour used. There are many tools that help testing. This thesis concentrates on the testing of a digital system design with a testbench.

The objective of this thesis is to add support for a new company wide vector format to two older graphics core generation testbenches. To achieve this objective it is necessary to

- acknowledge the behavior of the graphics pipeline
- understand IP-core verification
- comprehend the functionality of the testbench
- grasp the new and the old vector format
- add new functionality to the testbenches
- convert old format vectors in to the new format
- test the new functionality of the testbenches with a sufficient test set

2 BASIC CONCEPTS OF 3D GRAPHICS

The basic idea behind 3D graphics is to simulate real world occurrences in 2D display devices. The main objective of the 3D graphics pipeline is to project three-dimensional objects on to a two dimensional screen. This is achieved by using the appropriate lighting effects, projection calculations, textures, geometry transformations and others. [1]

This chapter presents the basic concepts of 3D graphics and gives the reader an insight to the complexity of 3D graphics.

2.1 Fixed Function Pipeline

The 3D graphics pipeline is divided in to three main stages application, geometry and rendering (Figure 1.). The geometry and rendering stages have recently been introduced with programmability brought by APIs such as DirectX and OpenGL, which enable support for various graphics effects. [1]

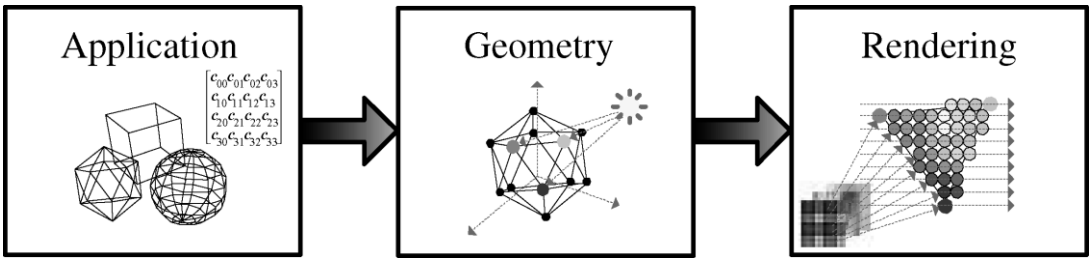

Figure 1. Fixed function pipeline. [1]

2.1.1 The Application Stage

The 3D graphics pipeline starts with the application stage, which feeds 3D models to be rendered in to the pipeline. The movement of the 3D objects is also generated in the application stage and is determined by environmental information. This environmental information includes user interaction and

internally generated information. To generate all this information the application process handles the artificial intelligence, collision detection and physics simulations, which are mainly used in gaming applications. Some other graphics applications include graphical user interfaces and virtual environments. All the 3D objects are built from sets of primitives and transformation matrices specify the movements of these objects. [1] Triangles, lines, points, polygons and quads are some of the used primitive types.

2.1.2 The Geometry Stage

The second stage of the graphics pipeline is divided in to two major operations. One is the geometric transformation of the vectors according to the matrices determined in the application stage. The other is to determine the colour intensity of each vector, which depends on the relationship between the properties of the vertex and the light source. [1]

Local space

There are several coordinate transformations that the geometric transformation goes through (Figure 2.). The 3D objects are first developed in the local coordinate space from which they’re gathered in to the world coordinate space through modelling transformation. Modelling transformation involves shifting, rotating, scaling and shear mapping operations on the 3D object. [1]

World space

In the second step the models developed in the local space are gathered in the world coordinate space to construct a 3D world. In this space the light sources are defined and intensity calculations are performed for the objects. Whether the actual lighting operation takes place in this coordinate space depends on

the chosen shading strategy. To observe the 3D world from a certain location a camera is set up in the world space. [1]

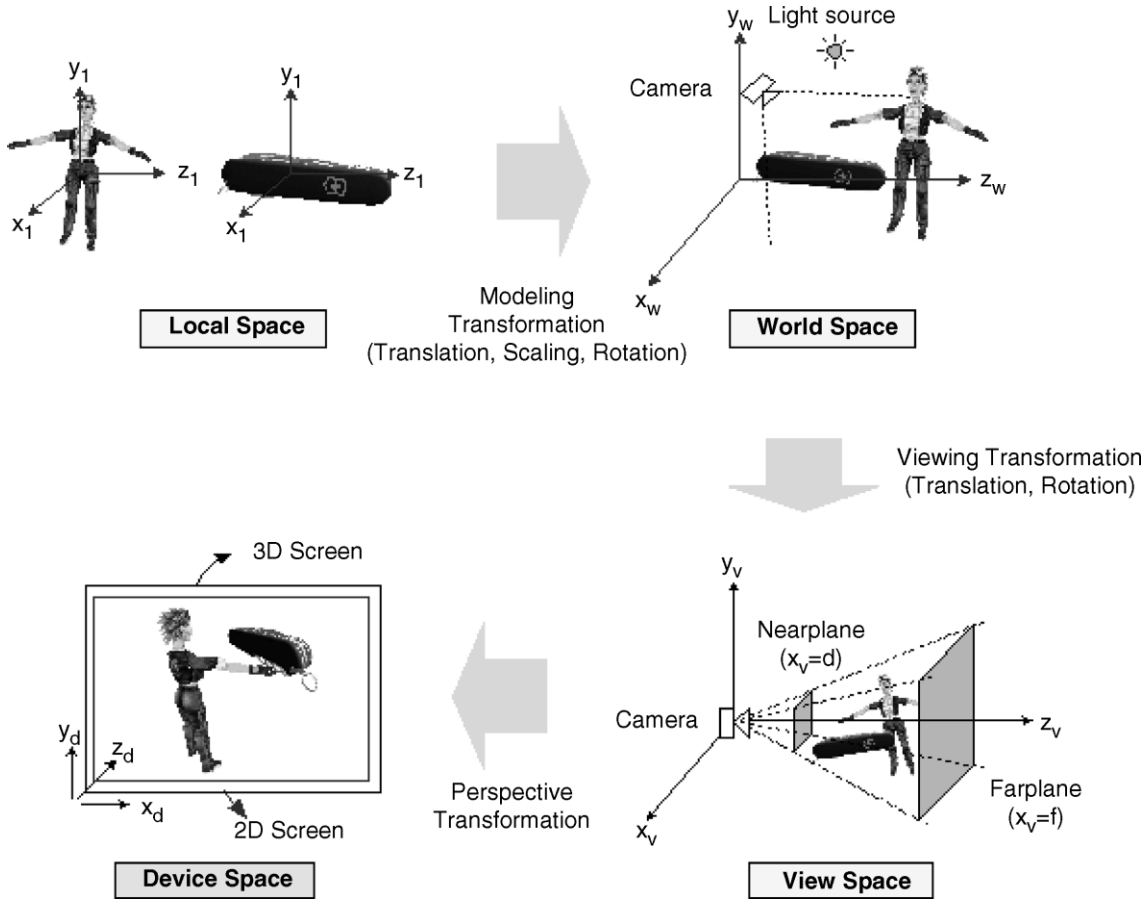

Figure 2. Spaces and coordinate systems in 3D graphics. [1]

Viewing space

A viewing transformation takes place when the world coordinate space is transformed in to a viewing coordinate space. In this transformation the camera is located at the origin. After the viewing transformation has completed the objects are aligned with respect to the camera and the origin of the viewing space. To prepare for the later rendering stages culling and clipping operations are carried out in the view space. The culling operation can remove polygons of a 3D object that aren't visible on the 2D screen, so that only front-facing polygons are processed. Culling is done by rejecting polygons that are back-facing when seen from the camera position. Culling saves a large amount of

processing in later stages. The view frustum (Figure 3.) is also defined in the view space. [1]

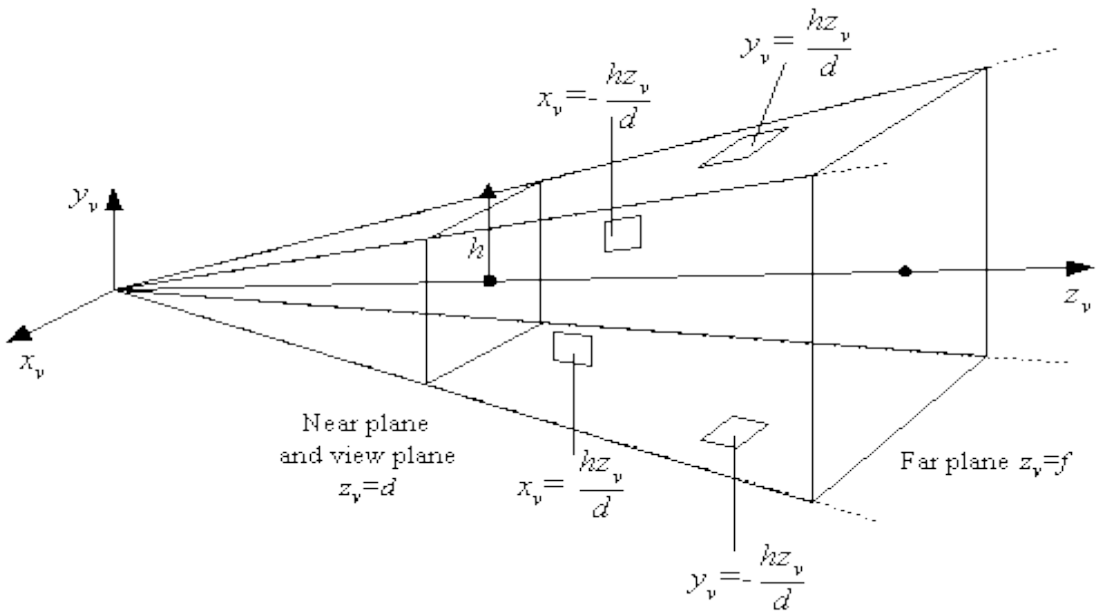

Figure 3. View frustum. [5]

Clipping space

The view frustum is defined by the camera position, which is the apex of the pyramid. The near plane or the view plane truncates the pyramid and the base of the pyramid is the far plane. [4] There are six clipping planes which include the far and near plane. These clipping planes are defined to determine which objects are to be considered for a scene. [1]

Perspective transformation, which is defined by the view frustum, transforms the objects from the viewing space to the clipping coordinate space. Clipping can also be done in the viewing space, but to avoid solving plane equations it is done in the clipping space. Clipping tests if polygons are completely outside, completely inside or straddling in reference to a square volume. When polygons are completely outside they are rejected, when they’re completely inside the polygons are processed normally. In the straddling case polygons are clipped

against six clipping planes and the ones that are inside are processed normally. [1]

Device space

The polygons in the clipping space are converted from a homogeneous coordinate system into a normalized device coordinate space. From here they are transformed in to device coordinate space by the viewport transformation. The device coordinate space is the final space in the Geometry Stage and is the space that will be seen on the display. The mapping of a scene on to the device screen is determined during the transformation. Also the size and the shape of the screen area are defined by the viewport. Depending on the aspect ratio of the viewport the polygons in the normalized device space are enlarged, shrunk or distorted. Finally in the device coordinate space all the pixel-level operations are performed. The pixel-level operations include Z –testing, texture mapping and blending. This stage is already a part of the rendering stage. [1]

2.1.3 The Rendering Stage

The last stage of the graphics pipeline is the rendering stage. During this stage various pixel-level operations take place. These operations include pixel rendering or shading, depth testing, texture mapping and other extra effects. [1]

Triangle setup

The triangle setup operation uses vertices from the geometry stage and generates a triangle before the rasterization can start. Attribute deltas between triangle vertices and edge slopes for the rasterization are calculated. The rasterizer then interpolates the triangle attributes for the pixels inside a triangle by incrementing the delta value while moving one pixel at a time. [1]

Shading

A shading algorithm needs to be determined at the start of the rendering stage. Gouraud, Phong and Blinn-Phong algorithms are some of the most commonly used shading algorithms. In Gouraud shading, which was invented by Henri Gouraud in 1971, the intensity of each vector is computed by a per-vertex algorithm. To determine the intensities of pixels inside a polygon, the rendering stage linearly interpolates the colour of each vertex determined in the geometry stage. In Phong shading, which was invented by Bui Tuong Phong in 1973, the intensity of each pixel in a polygon is computed; this is called a per-pixel algorithm. Phong shading requires high computational power, but is able to generate sharp specular lighting effects in cases in which the pixel colour changes rapidly. [1] In Blinn-Phong shading, which is a modification of the Phong model developed by Jim Blinn in 1977, instead of calculating the angle between the viewer and the reflected beam of the light source a halfway vector is calculated between the viewer and the light-source vectors. It can produce more accurate models in some cases. [6]

Texturing

The reality of 3D graphics scenes can be enhanced by texture mapping, invented by Edwin Catmull. 2D texture images can be applied to 3D objects by using a picture of the real surface of a 3D object. This reduces the amount of complex computations of lighting or geometric transformation, but instead requires a large memory bandwidth to fetch texels to be used for the mapping. Texture filtering is also needed, which reduces the aliasing artefacts of the textured image. Bilinear interpolation, mip-mapping and normal mapping are some of the various filtering algorithms. [1]

Z-testing

To remove hidden surface pixels, depth testing, or Z-testing, is used. Z-testing was also invented by Edwin Catmull. Applying depth testing a depth buffer is required. The size of the buffer is determined by the resolution of the 2D screen. Every drawn pixel is given a depth value. To determine if the pixel is visible or not the depth value of each pixel drawn is tested for whether it is in front or behind the pixel at the same position of the depth buffer. If the pixel passes the test it is drawn on the screen. In the other cases the pixels depth and colour values are discarded, because the pixel is further from the viewer than the pixel stored in the given position of the frame buffer. [1]

Extra effects

Finally several extra effects can be used to enhance the final scene. Some of these are alpha blending, fog effect and anti-aliasing. Alpha blending determines the opacity of a pixel, which is represented by the alpha value. The fog effect makes objects located further from the viewpoint fade away to make the scene more realistic. Anti-aliasing is used for the final 3D graphics scene to reduce its jagged look, which depends on the resolution of the scene. The higher the resolution, the less the jagged look. This is achieved by an anti-aliasing algorithm that calculates what fraction of the pixel covers the line. [1]

2.2 Programmable Graphics Pipeline

The fixed function pipeline only supports a set of predetermined graphics effects, but recently introduced programmable 3D graphics has enabled the support of various other effects. The programmable graphics pipeline (Figure 4.) has two major stages: vertex shading and pixel shading. [1]

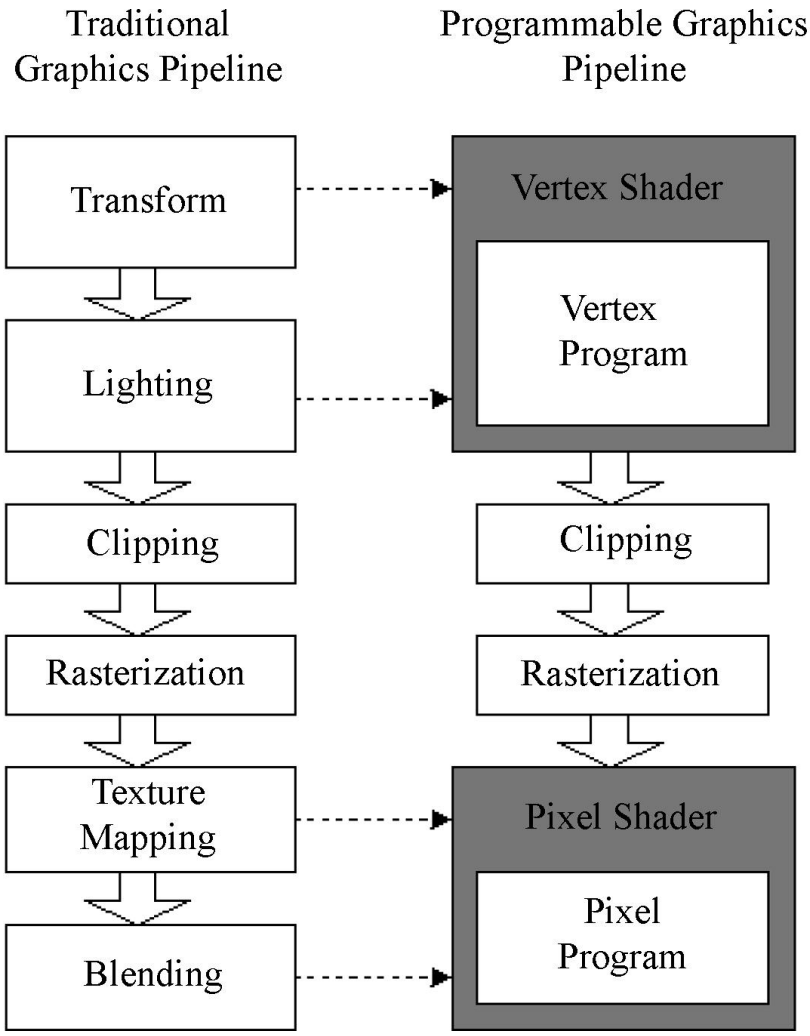

Figure 4. Programmable graphics pipeline. [1]

2.2.1 Vertex Shader

The vertex shader takes vertex streams as an input and as the output produces a new vertex stream. A vertex includes a number of attributes that the vertex shader updates to locate the new vertex position in the clipping coordinate space. The main functions of the vertex shader are to transform the vertex coordinates from the local coordinate space into the clipping coordinate space and to compute the intensity of the vertex colour. [1]

The vertex shader has several operand register files which include read-only input register files, constant memory, and read-write temporary register files.

The shader also has write-only output register files. The source registers contain the vertex attributes, which consist of changing per vertex or per frame input data. Changing per-vertex data includes data such as the vertex position, normal, texture coordinate and the colour; this data is located in the input registers. Changing per-frame data includes data such as the transformation matrix, light position and material, this data is located in the constant memory. [1]

When the vertex shading is complete the result is written to the write-only output register file. The transformed vertex position in the homogenous clipping coordinate space and lit vertex colours are located in the output registers. The output vertex then goes through the fixed-function stages to the pixel shader. [1]

2.2.2 Pixel Shader

The pixel shader takes in the interpolated attributes of each pixel output by the vertex shader [2]. The main operations of the pixel shader include texture mapping and colour blending. The pixel shader gives more flexibility in texture mapping with the texture sampler stage. The texture sampler enables more advanced rendering effects by being able to read a dependent texture. In a dependent texture read the texture coordinates of a later texture access are modified by an earlier texture read. [1]

The general model of the pixel shader is similar to the vertex shader, by having the same operand register files and the write-only output register file. Interpolated pixel attributes, such as colour and texture coordinates, are found in the input register file. The pixel shader determines the resulting colour that will be displayed, with the texture read values and the input register attributes, and stores it in the output register file. [1]

3 IP CORE VERIFICATION

Verification of a design ensures that it meets the functional requirements defined in the functional specification. Verification of a system on a chip (SoC) is a very time consuming process and can use up to 70 per cent of the total development effort of a design. Issues that challenge verification engineers are what strategies and technology options to use for verification, what is the sufficient amount of verification and how to plan for and minimize verification time. [10]

3.1 Technology options

Verification technology options can be categorized into three classifications: simulation technologies, formal technologies and static technologies. A combines of these methods ensures that the verification goals are achieved. [10] The following sections will briefly describe these technology options.

3.1.1 Simulation technologies

There are various different simulation technologies. This section will explain briefly a few of these technologies: RTL simulation, which includes event- and cycle-based simulators and transaction based verification, code coverage, emulation systems and hardware accelerators. [10]

RTL simulation

Event-based simulators propagate events through a design one at a time until a steady state is achieved. A change in input stimulus is considered as an event. Event-based simulation offers high simulation accuracy but depending on the size of the design the execution might be slow. [10]

Cycle-based simulators evaluate logic between state elements and/or ports in the single shot because they do not have any notion of time within a clock cycle. They are very prone to errors, but can be significantly faster than event-based simulators. [10]

Transaction-based verification allows the design to be debugged and simulated at a transaction level and in which all possible transactions between blocks in a system are systematically tested. This improves the verification productivity by raising the level of abstraction. [10]

Code coverage

A specific test suite is determined and applied to a design. Code coverage analyses the amount of coverage the specified test suit achieves. It can be used either for block level or full-chip level analysis. The analysis produces a percentage of the amount of coverage achieved by the test suite. [10]

Emulation systems

An emulation system is typically a configurable and programmable environment. The system can take on the behaviour of the target design and emulate its functionality to the degree that it can be directly connected to the system environment in which the final design is intended to operate. These systems can in some instances approach the target design speeds. They can also perform at much higher speeds than software simulators as they are realized in hardware. [10]

Hardware accelerators

Hardware acceleration speeds up certain simulation operations by mapping some or all of the components in a software simulation into a hardware

simulation platform. Usually the verified design runs in the hardware accelerator while the testbench remains running in software. [10]

3.1.2 Static technologies

Lint checking checks the design code for syntactical correctness by a static check. It uncovers errors such as port mismatches, uninitialized variables, typecast and unsupported constructs. It also checks for the quality of the code. It identifies simple errors and can be performed early on in the design cycle. [10]

Timing verification makes sure that the timing requirements are met. These timing requirements are found from each storage element and latch in a design. Some of these timing requirements are setup, hold and various delay timings. In a complex design each input can have multiple sources and timing can vary, which makes timing verification challenging for these systems. [10]

3.1.3 Formal technologies

Detecting bugs that depend on specific sequences of events is very important. These bugs need to be detected early on in the verification. Formal verification has an exhaustive nature and enables early bug detection. They do not require testbenches or vectors and they promise very fast verification time and a 100 per cent coverage. There are three formal verification methods: formal equivalency checking, formal model checking and theorem proving technique, which is still under academic research. [10]

Formal equivalency checking tries to prove the equivalency of two different views of the same logic design by using mathematical techniques. It verifies the equivalence of a reference design and a modified design and can be used to verify RTL-RTL, RTL-Gate and Gate-Gate implementations. For formal equivalence checking it is critical that the reference design is functionally

correct. It has one issue, which is that it does not verify the timing of the design. [10]

Formal model checking verifies behavioural properties of a design, which is done by using formal mathematical techniques. The logical properties specified in the design specifications are compared with the design behaviour by a model-checking tool. [10]

3.1.4 Verification option comparison

It is necessary to choose the correct verification option for certain steps of the design cycle. Depending on the stage of the cycle one verification technology can suit the purpose better than another. In Table 1. different features of some of the verification technologies are shown.

Table 1. Comparing Verification Options. [10]

	Event-based simulation	**Cycle-based simulation**	**Hardware Accelerators**	**Emulation**	**Formal Verification**	**Static timing Verification**
Function	Yes	Yes	Yes	Yes	No	No
Abstraction Level	Behavioral, RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	Gate
Functional Equivalence	Yes	Yes	Yes	Yes	Yes	No
Timing	Yes	No	Yes/No	No	No	Yes
Gate Capacity	Low	Medium	High	Very High	High	Medium
Run Time	<10 Cycles	1K Cycles	1K Cycles	1M Cycles	Medium	High
Cost	Low	Medium	Medium	High	Medium	Low

For example when verifying a small design and timing and function verification is needed the best option would be event-based simulation. If a slightly bigger design is verified and no timing is needed, then cycle-based simulation is the best choice. Lastly if a larger design needs to be verified with high simulation speed, emulation would be the best verification option.

3.2 Verification methodology

Figure 5. describes the flow of a verification methodology. The flow starts with the designing of hardware RTL according to a pre-defined specification. A testbench is then created for the RTL. The testbench and the RTL are compiled by a simulator, which then takes a test or a set of tests as an input. The simulator outputs a result.

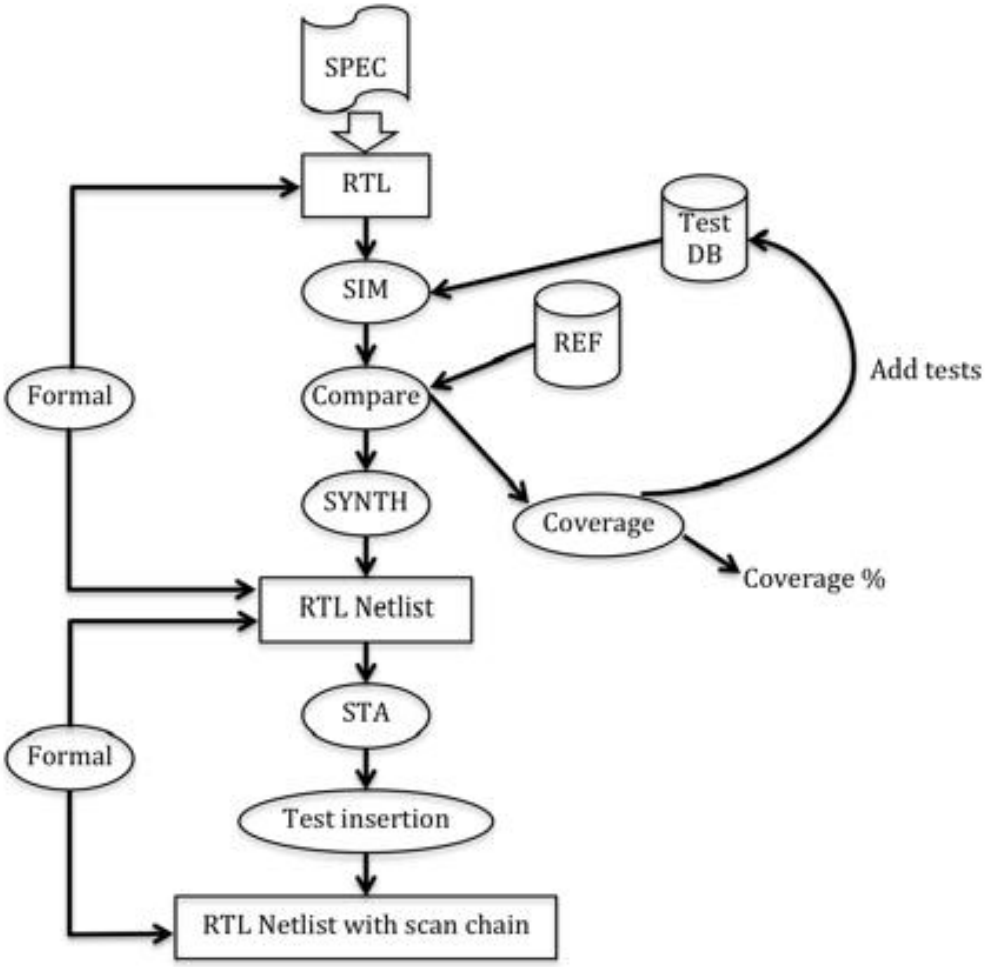

Figure 5. Verification methodology.

The result from the simulator can either be an image or a wave. The result is compared with a reference and checked if the test ran correctly, this comparison outputs a passing percentage. The result is also checked with the simulation and the specification for code coverage, which is also measured as a

percentage. When the determined pass and coverage percentages have been achieved the first phase of the flow can be determined complete. The work done for this thesis concentrates on this phase of the verification methodology.

The second phase starts with the synthesis of the hardware RTL, which generates a gate-level netlist. The generated RTL netlist can then be verified by formal equivalence checking against the RTL code, where the RTL netlist is the implementation and the RTL code is the reference design. This checking makes sure that RTL netlist and the RTL code are logically equivalent. When equivalence is achieved the second phase is complete. [10]

The last phase starts with a static timing analysis or static timing verification, which checks that the timing requirements are met. RTL netlist is then given test insertions that create a RTL netlist with scan chains. The RTL netlist with scan chains is then verified against the RTL netlist with formal equivalency checking to make sure that they are logically equivalent.

3.3 Testbench creation

A testbench is designed to apply stimulus to the device under test (DUT) and to check if the DUT responds correctly to the stimulus. A testbench can be self-checking, which means that it applies input and then samples the output from the DUT with the expected result. It also provides error information. Self-checking testbenches are recommended for all designs as they make detecting, understanding and fixing of errors easy, they also decrease the amount of manual work. When creating a testbench it is necessary to have thorough understanding of the functional specifications. The following sections will describe three different functional verification approaches that use testbenches. [10]

3.3.1 Black-box verification

In black-box verification the DUT is thought of as a black box (Figure 6.). This means that the internal design details are unknown for verification and that errors can only be detected at the output. Exhaustive test vectors are used to simulate errors. Black-box verification attempts to find initialisation and termination errors, interface errors, performance errors and incorrect or missing functions. [10]

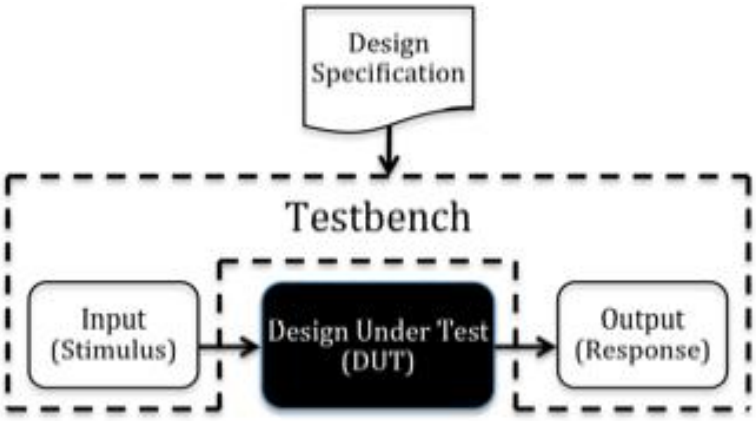

Figure 6. Black-Box Verification Approach. [10]

The work done for this thesis used the black-box verification approach.

3.3.2 White-box verification

In white-box verification, unlike black-box verification, the design data and structure are visible enabling good observability and controllability for verification. This widely used verification approach enables the source of errors to be detected and identified as corner cases can be easily generated. [10]

3.3.3 Grey-box verification

Grey-box verification is a mix of the previous two verification approaches. This means that some details of the DUT are known. Reasons for using the grey-box verification approach are contract restrictions or verifying at a greater level isn't necessary. [10]

3.4 Verification approaches

There are various verification approaches. This section will briefly cover the top-down design and verification, bottom-up verification, platform-based verification and system interface-driven verification, which was used for this thesis. [10]

3.4.1 Top-down design and verification

Top-down design starts off with a functional specification and from this a verification plan is developed. A system design is developed from the functional specification. The system design can then be functionally verified with a system-level testbench. After the system design has been verified sufficiently a hardware description language (HDL) design of the system design can be implemented. The HDL design can be verified with lint checking or formal model checking. After the HDL design is verified the implementation views can be verified with either equivalency checking tools or simulation. Finally to ensure correct chip implementation timing verification, physical verification and device tests are performed. For larger designs it might be sensible to use emulation, hardware accelerators or partitioning the design in to functional blocks. [10]

3.4.2 Bottom-up verification

The widely used bottom-up verification starts by validating incoming design data. This is done by passing the design files through a parser to ensure that they are compatible with the target tools. When the validation is complete the design files are passed through a lint checker. If the design is at a higher level it will only proceed through the system-level testing, but if the design is at a detailed level it will pass through an additional three levels of testing. Where the first level of testing verifies the individual components, blocks or units in isolation. The second level verifies the internal interconnect and the system memory map of the design. And the third level of testing verifies the basic functionality of the design and the external interconnect. After these tests the following verifications are performed netlist verification, timing verification, physical verification and device testing. [10]

3.4.3 Platform-based verification

Platform-based verification is used for designs that are based on a pre-existing platform that is already verified. Additional IPs are added and verified separately. This involves interconnect verification between the basic platform and the additional IP blocks. The whole platform can be verified by using the top-down or the bottom-up verification approach. [10]

3.4.4 System interface-driven verification

System interface-driven verification, which was used for this thesis, is shown in Figure 7.

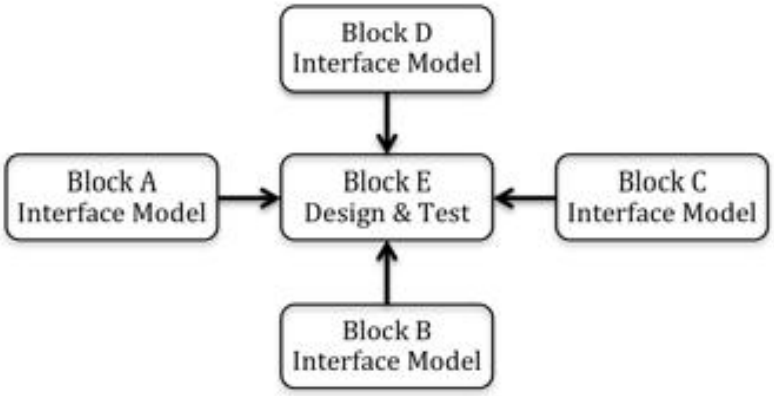

Figure 7. System interface-driven verification. [10]

In this approach during the system design the blocks to be used in the design are modelled at their interface level. The interface models can be used to verify the interface between the designed block and the system, which enables early error detection and eases the final integration. In Figure 7. block E is being designed and verified using the interface models of A, B, C and D blocks. If another block is to be verified block E is replaced with its interface model. [10]

4 DEVELOPMENT ENVIRONMENT

4.1 Testbench functionality

A testbench is used to verify the functionality of a digital system design. [3] The testbench has connections to different blocks of the DUT. These wires are used to give stimulus and check the response of the design. [3]

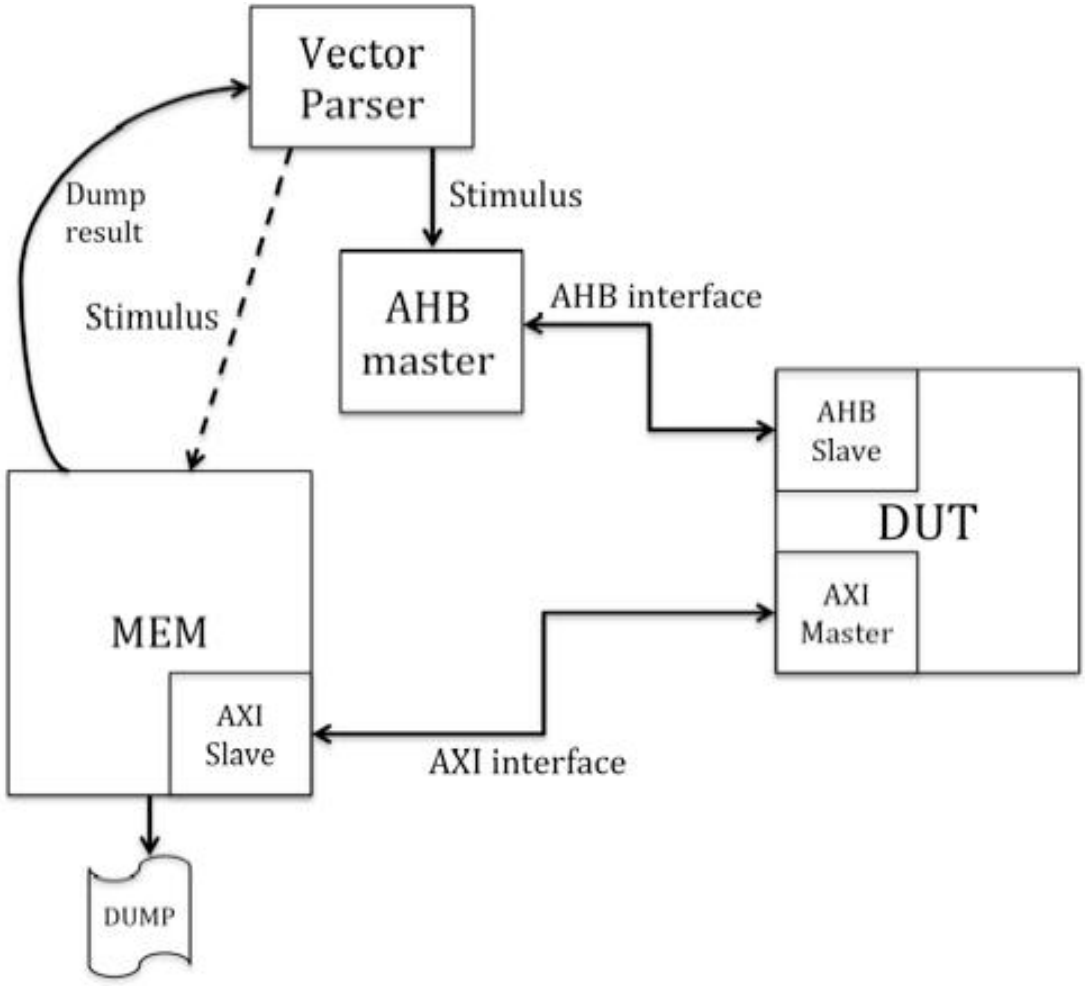

Figure 8. Testbench functionality

Several testcases are generated to test a digital system design, one testcase for each required feature. A testcase has a vector file, which consists of multiple command lines. When a testcase is driven to the test the RTL design, a block of

the testbench reads the vector file line by line and parses the command lines in to stimulus. Depending on the command data this stimulus either sent to the AHB master or the memory.

4.1.1 AHB interface

The Advanced High-performance Bus (AHB) is a bus register interface that provides high-bandwidth operations and supports a single bus master. It is an industry standard interface from ARM. It implements features including burst transfer and single-clock edge operation required by a high-performance, high clock frequency system. [7]

An AHB-Lite system consists of these main components: Master, Slave, Decoder and Multiplexor. The master initiates the read and write –operations by providing the address and control information to the system. The slave's response to a transfers initiated by a master is controlled by the HSELx signal from the decoder. The HSELx, where the x is a unique identifier for each slave, indicates which slave should be used. A signal, which implicates the status of the data transfer, is sent back from the slave to the master. The decoder takes in the address of each transfer and decodes it. After which it provides a select signal for the slave that will be used in the transfer. The multiplexor is controlled by a control signal from the decoder. The multiplexer takes in the read data bus and response signals from the slaves and sends them to the master. [7]

AHB write burst example

A simple transfer (Figure 9.) does not have any wait states, it only consists of one address cycle and one data cycle. The HWRITE signal from the master indicates if the transaction is a read or a write, when high it is a write transfer and when low it is a read transfer. HADDR signal from the master contains the address and the HWDATA the write data. The HREADY signal from the

multiplexor indicates to the master and the slaves when the previous transfer is complete. [7]

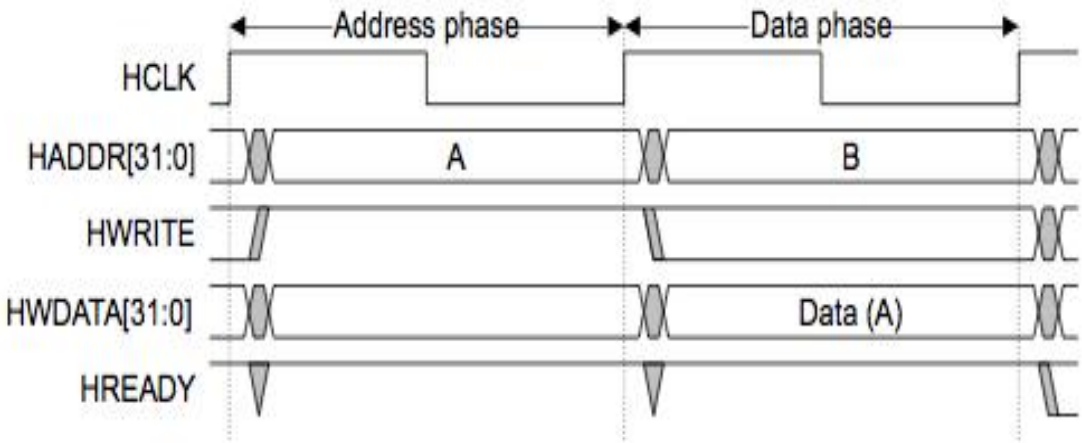

Figure 9. AHB write transfer. [7]

The AHB controls the register reads and writes to and from the DUT. The DUT has a block that acts as the AHB –slave. The AHB –master is a block between the vector parser and the DUT. The stimuli from the vector parser include data that defines whether the transaction is a read or a write, the register address and the read or write data.

4.1.2 Advanced eXtensible Interface

The Advanced eXtensible Interface (AXI) is an interface that is designed for high-performance and high frequency system designs. Some features that enable this are

- the address/control and data phases are separated
- a burst of data can be accessed by only issuing the start address
- a low-cost *Direct Memory Access* (DMA) is enabled by having separate read and write channels. [8]

The address and control information for each transaction are sent through the address channel. The control information describes the transaction type, read or

write. The master has a read data channel from the slave and a write data channel to the slave. During a write transaction there is an additional write response channel that enables the slave to signal to the master that the transaction is complete. All of these channels are independent. [8]

AXI read burst example

A simple example of an AXI read burst is show in Figure 10. All the channels have a set of information signals. ARVALID, which is a sent from the master, indicates when the address and control information is valid; it will stay high until the address acknowledge signal ARREADY from the slave goes high. The RVALID signal from the slave indicates that the required read data is available and the transfer can complete. RREADY signal from the slave lets the master know when it can accept read data. ARADDR consists of the address and control information and the RDATA consists of the read data that was requested. A write burst is very similar to a read burst. [8]

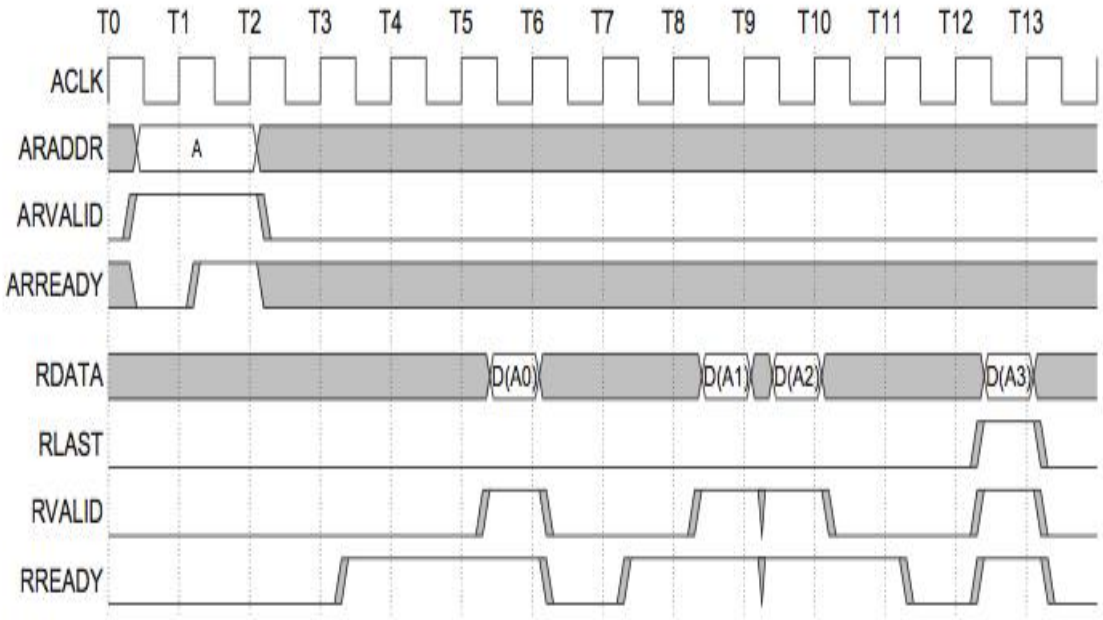

Figure 10. AXI read burst. [8]

The AXI controls the memory interactions between the memory and the DUT. The AXI –master is a block in the DUT and the AXI –slave is a block in the memory. The memory model is filled by the stimuli gotten from the vector parser. The master receives memory read requests from different blocks of the DUT and sends the request to the slave. These requests include the address of the data that is located in the memory. The slave receives the data and fetches the data from the specified address and sends it back to the master. Finally the master sends the data to the block that originally requested it. Most of the transactions processed by the AXI during a simulation are reads. At the end of the simulation there is a memory write –operation that writes the final image to the memory.

4.2 Vector interface

A testcase has a test vector file that determines what happens while running the test. The file consist of multiple command types, that all have a specific function. The functionality of each command type is defined in a control block, which is connected to the top level of the testbench.

The testbenches for the older GPU's had a test vector format that is no longer used in the newer GPU's testbenches. However the newer testbenches support the old vector file format as well as the new company wide format. The old and the new vector formats have the same functionality, but the syntax differs. The old testbenches needed to be updated so that they support both vector formats. This meant that the testbench needed to be able to check which format was in use. It also meant checking the test folder for which test format is found, as the different test vector files are never to be in the same folder.

4.2.1 Vector generation

Test vector generation has three main layers, the user, the software and the hardware layer. These layers are depicted in Figure 11. The user layer consists

of an application. The software layer consists of the software driver and the hardware layer consists of the reference model and the RTL model. The vector generation flow starts off with an application. Depending on what is being tested and on the size of the DUT, the application can either be a block level test or a top level test. A block level test tests a specific block in the RTL model and a top level test tests a larger DUT for example a set of blocks.

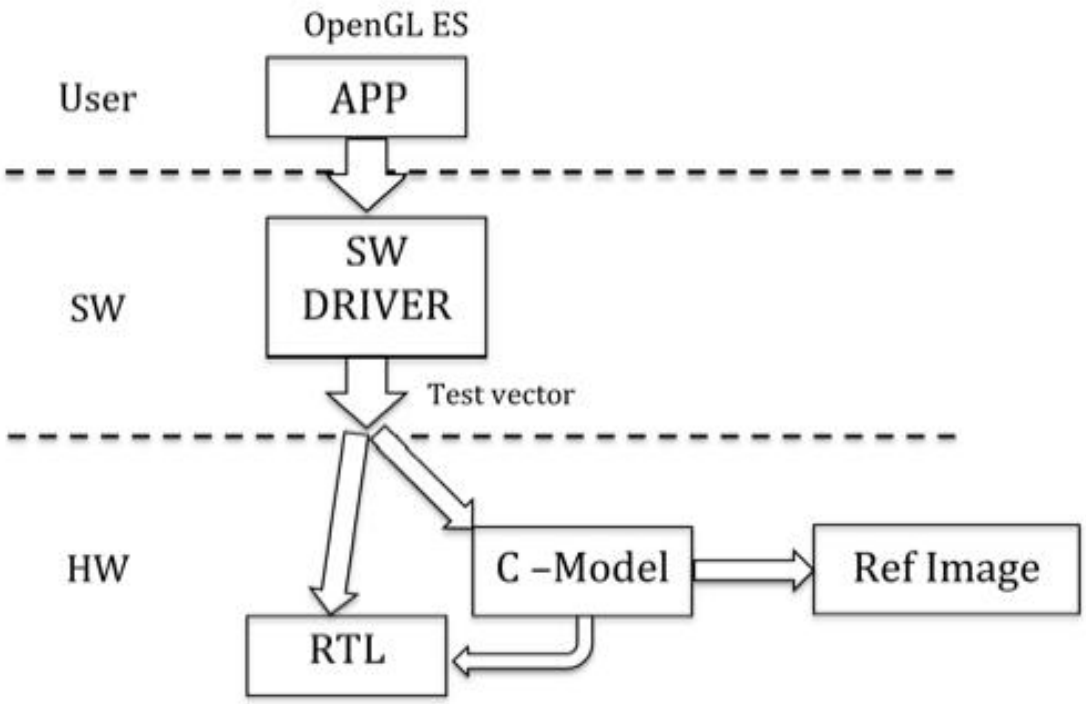

Figure 11. Test vector generation.

The application, which is written with a graphics application programming interface (API) e.g. OpenGL-ES, is driven to the software driver. The software driver processes the application and dumps out a command stream, which is called a test vector. The test vector can then be driven straight to the RTL or if a reference is needed it will be driven first through the reference model. When a test vector is driven through the reference model it dumps out a picture. This picture can then be used as a reference to the picture that is dumped out when the vector is driven through the RTL.

4.3 Waveform viewer

The waveform viewer is a software tool used for verification of a digital design in union with a simulator. It enables the visualisation of multiple signals over time and their relationship with other signals. Figure 12. shows an example of a waveform viewer. [11]

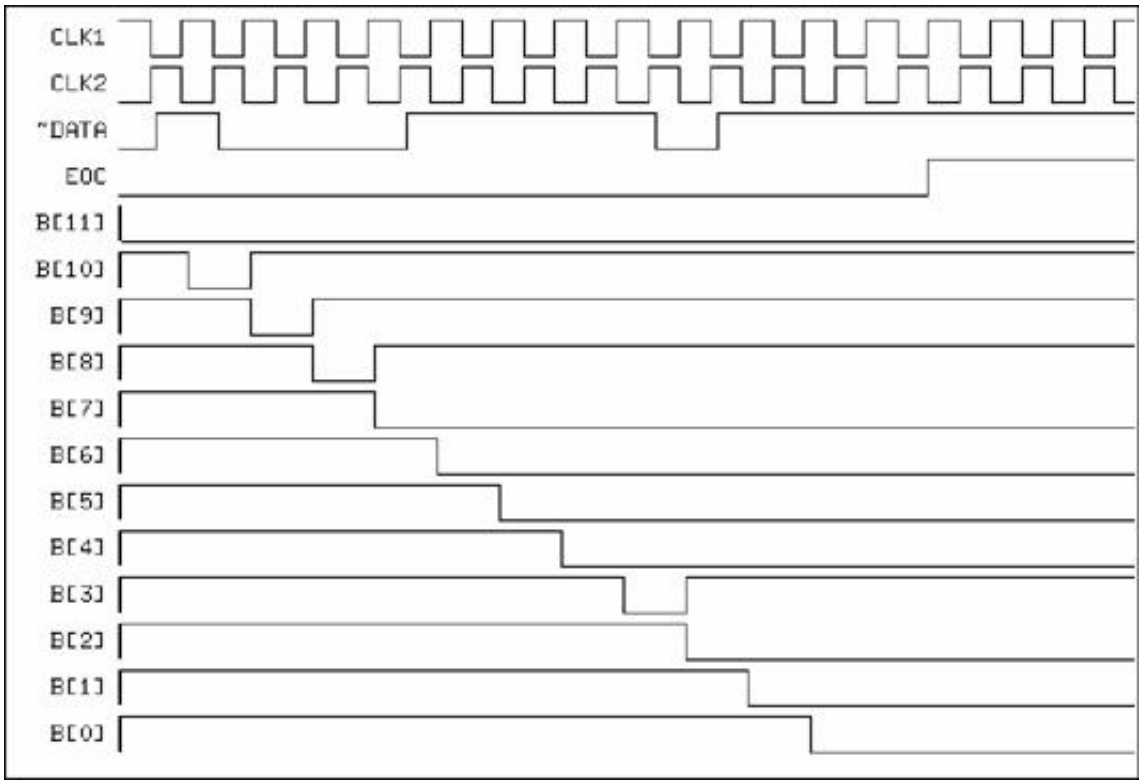

Figure 12. Example waveform. [12]

Waveform viewers are an integral part of checking if the design is functioning correctly. They can be used interactively or after the simulation has completed to check why and when problems occur in a design or testbench. [11] In this thesis a waveform viewer was used as a post-processing tool after the simulation had completed.

4.4 Regression environment

A regression environment is an environment that enables a large set of tests to be run simultaneously. It is an integral part of mobile graphics testing, as usually more than a hundred tests are wanted to simulate concurrently to save time and resources.

To be able to simulate multiple tests concurrently, a server farm was used. The farm had multiple machines that could run multiple jobs simultaneously. The regression environment that was used for this thesis had multiple job queues that all had a set priority level. The higher the priority of the queue the fewer jobs could be run simultaneously. Each queue had a job limit and a per user job limit, which enabled multiple users to use the same regression queue. When a job was sent to the server farm a run log could either be written to a file or received by e-mail. When a job was done the farm sent a response back to the user.

4.5 Version control

Version control is an integral part of any development process. It enables users to do concurrent, parallel work on files and it keeps track of changes done to these files [13]. Version control keeps track of who, when, why and where a file was modified. This helps to find when a bug might have been introduced and when the bug was fixed and who to ask for further questions. Merging a document is a very important part of version control as it enables the concurrent work on a document by many people. [14] Version control also helps with running a project by providing a central coordinating force among all the different areas involved in the project.

5 RTL TESTBENCH INTERFACE UNIFICATION BETWEEN DIFFERENT GRAPHICS CORE GENERATIONS

5.1 Testbench modifications

With the new vector format came the new syntax. The way that this new syntax was processed could be found from the newer testbenches, which was then integrated to the old testbenches. Some command types had been introduced with additional functionality and some all-new command types had been added to the new testbenches. The additional functionality and the new command types were integrated to the old testbenches with a step-by-step approach. After the command type functionality had been integrated, a testcase was run. The output of the run was written to a log file and checked for errors, if any errors were found a wave dump was needed. The testcase was then run again with a dumping parameter. The wave dump was then opened with a waveform viewer, in which the signals that were used by the functionality could be viewed. By looking at the signals' waveforms it could be determined whether they were working correctly. When the issue was found the functionality could be fixed and then tested again.

5.2 Testing

To test the new functionality, new vectors were needed. For getting the new format vectors, old format vectors had to be converted. Register differences restricted the use of vectors generated for newer RTL designs. To create a new vector from an old vector a script was run. This script read through the old vector file command line by command line and by using the data, wrote the corresponding command in the new format in to a new format vector file. The

same script also generated the reference files needed for the test to run correctly.

For initial functionality testing only one testcase, that was sure to test the newly implemented functionality, was run. If that test passed the next step was to run a set of testcases. A script had been written to speed up the process of running multiple testcases. This script had a list of testcases, which it processed one by one, and ran them in the regression environment. When all the jobs were done in the environment a response was sent back. The script waited for the response and as soon as it got it, ran a script to check for errors in the simulation log files. If an error was found, the error message was written to an error log file. This enabled an easy way to check if a test had not run successfully to the end. If an error occurred or if the reference image created by the C –model didn't match with the image dumped out from the simulation, debugging was necessary.

5.3 Debugging

Debugging is a very important step in any development cycle. It is also a very time consuming step as the problems that need to be debugged are seldom clear and straightforward. In Figure 13. a debugging flow used for this thesis is depicted.

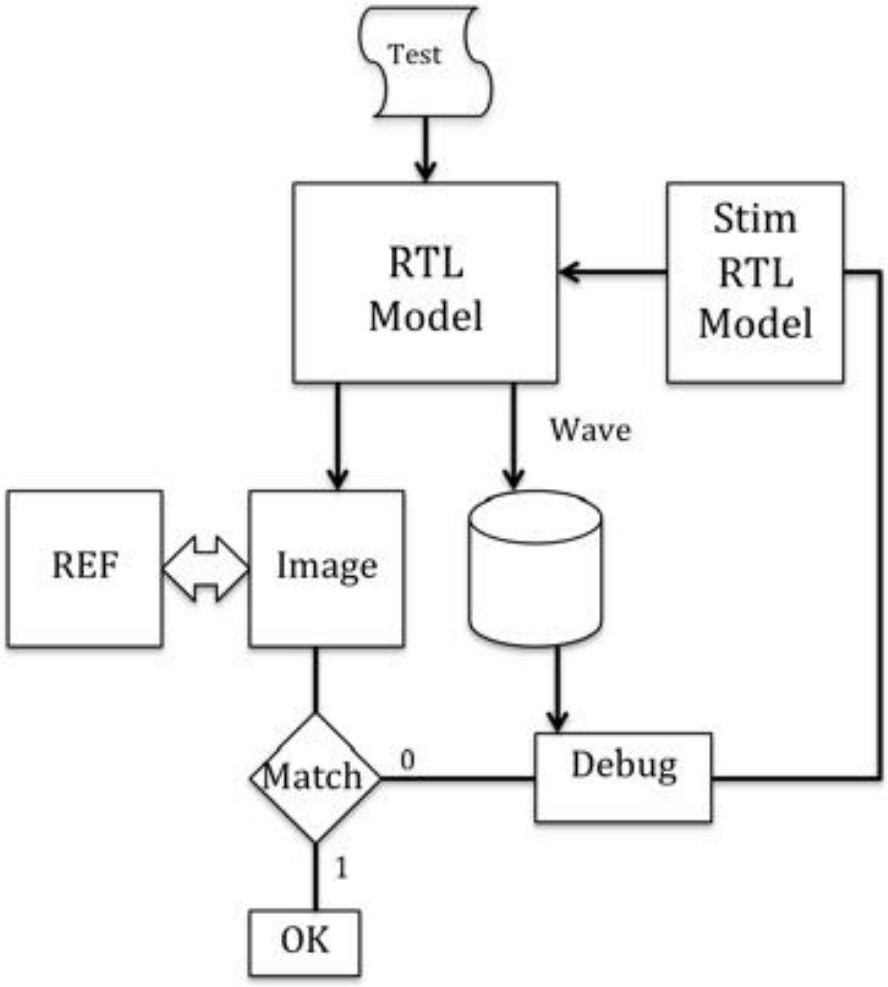

Figure 13. Debugging flow

When a test simulation returns with an error or the images do not match, it is simulated again. The second simulation is run with a parameter that enables a wave file to be dumbed out. The reason why simulations are not run with the wave dumping parameter for the first simulation is because the simulation time increases significantly when a wave is also being dumped. After the second simulation the wave dump is loaded in to a waveform viewer. In the viewer a set of signals that are of interest can be picked and examined, which means checking if the signals are functioning as they should when they should. When the issue is found, the testbench is fixed so that it correctly passes on the stimulus to the RTL model. Finally the testcase is run again and the images are compared and the log files checked for errors. If the test still isn't working correctly the debugging process is done again.

6 CONCLUSION

The objective of this thesis was to add support for a new company wide vector format to two older graphics core generation testbenches. To achieve this objective it was necessary to test the new functionality of the testbenches with a sufficient test set. To be able to test and debug the changes made to the testbench, it was essential to understand the behaviour of the graphics pipeline.

The new testbench functionality was tested with a specific test set that was known to cover GPU to system interactions. It consisted of 127 tests of which all passed successfully. During the process of the thesis the understanding of the behavior of the graphics pipeline increased significantly.

REFERENCES

[1] Woo, Sohn, Nam, Yoo 2010. Mobile 3D Graphics SoC: From Algorithm to Chip. Hoboken: Wiley.

[2] Scabia, M. 2011. Vertex and Fragment Shaders. Adobe. WWW document. Available from: http://www.adobe.com/devnet/flashplayer/articles/vertex-fragment-shaders.html [cited 1.2012]

[3] Bergeron, Cerny, Hunter, Nightingale 2006. Verification Methodology Manual for System Verilog. New York: Springer.

[4] Lightouse3d.com. 2011. View Frustrum Culling. WWW Document. Available from: http://www.lighthouse3d.com/tutorials/view-frustum-culling/ [cited 1.2012]

[5] Rheingans, P. 1999. Viewing. WWW Document. Available from: http://www.csee.umbc.edu/~rheingan/435/pages/res/gen-8.Viewing-single-page-0.html [cited 1.2012]

[6] Xiaorui. 2009. Blinn-Phong shading model. WWW Document. Available from: https://sites.google.com/site/sxralanwebsite/theme/job2-blinn-phong-shading-model [cited 1.2012]

[7] ARM. 2008. AMBA 3 AHB-Lite Protocol Specification. PDF document. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html [cited 4.2012]

[8] ARM. 2008. AMBA AXI Protocol Specification. PDF document. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html [cited 4.2012]

[9] Larson, Dickol, O'Brien. Performance verification of a complex bus arbiter using VMM Performance Analyzer. WWW Document. Available from: http://www.design-reuse.com/articles/24477/bus-arbiter-verification-vmm-performance-analyzer.html [cited 4.2012]

[10] Rashinkar, Paterson, Singh 2001. System-on-a-chip Verification. Norwell: Kluwer Academic Publishers.

[11] Bergeron, 2000. Writing Testbenches: Functional Verification of HDL Models. Dordrecht: Kluwer Academic Publishers.

[12] SILVACO. 1994. Mixed-Mode Simulation Using SmartSpice. WWW Document. Available from: http://www.silvaco.com/tech_lib_TCAD/simulationstandard/1994/oct/a4/a4.html [cited 4.2012]

[13] Wingerd, 2006. Practical Perforce. California: O'Reilly.

[14] OSS Watch. 2005. What is version control? Why is it important for due diligence? WWW Document. Available from: http://www.oss-watch.ac.uk/resources/versioncontrol.xml [cited 5.2012]

[15] Fogel, K. 2005. Producing Open Source Software. WWW Document. Available from: http://producingoss.com/en/index.html [cited 5.2012]