Hardware-software model co-simulation for GPU IP development

Jaime Gancedo Rodrigo
jaimegancedo@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Asst.Supervisor: Reimar Döffinger (Arm)

Examiner: Erik Larsson

April 19, 2018

Printed in Sweden
Tryckeriet i E-huset, Lund

"Head first, hands afterwards."

From the Spanish:
"Primero la cabeza, después las manos."

Life lesson, Someone very wise

Abstract

This Master's thesis project aims to explore the possibility of a mixed simulation environment in which parts of a software model for emulating a hardware design may be swapped with their corresponding RTL description. More specifically, this work focuses on the sofware model for Arm's next-generation Mali GPU, which is used to understand system on chip properties, including functionality and performance. A component of this model (written in C++) is substituted with its hardware model (written in SystemVerilog) to be able to run simulations in a system context at a faster emulation speed, and with higher accuracy in the results compared to a pure-software model execution. For this, a "co-simulation" environment is developed, using SystemVerilog's DPI-C as the main communication interface between C++ and SystemVerilog. The proposed environment contains new software and hardware blocks to enable the desired objective without major modifications in neither the software Mali model nor the substituted component. Metrics and results for characterizing this co-simulation environment are also provided, namely timing accuracy, data correctness and simulation time with respect to other previously available simulation options. These results hope to show that the proposed environment may open new use-cases and improve development and verification time of hardware components in a system such as the Mali GPU.

Acknowledgments

I would like to begin by thanking my supervisor, Professor Liang Liu and my examiner, Professor Erik Larsson for their help during this project, specially on the initial arrangements. Likewise, I am grateful to my supervisor at Arm, Reimar Döffinger, for guiding me in such a challenging task and Noelia Rodríguez, for making me a part of the GPU modelling team during my time at Arm Lund.

To my family, specially my parents, who have supported me ever since I can remember, been by my side at the best and not so good times and helped me reach my goals and dreams. I can hardly express how grateful I am.

To my professors and friends of *Universidad Politécnica de Cartagena*, for helping me become an engineer, make me realize what I wanted my profession to be and grant me some of the best opportunities of my life. This Master's Thesis probably wouldn't have happened without you.

To my friends in Lund: Berta, Javier G., Javier M., Sergio, Manuel, Iratxe, Diego and everyone else for making me feel like at home, remember my Spanish, get to love vegetables, enjoy the small things, for all the technical discussions after office hours and, in general, for making these two years in Sweden a memorable experience.

To my life-long friends from Cartagena: Ignacio, Marina, Daniel, Helena, Ester, Carlos, David, Deborah and everyone else for having me forget about everything else and feel as if time had stopped when I am back home. I am also grateful to Peter Casanave, for being my English teacher for so many years and my friend. You would not believe it if I told you I have just looked at the dictionary two or three times to write this document!

To every person that is not mentioned here, but has played a smaller or bigger part in my life, and the aforementioned: Everything we live and experience, both good and bad, makes us who we are. Thank you for being a part of this journey.

Popular Science Summary

Hardware-software model co-simulation for GPU IP development: Software and hardware under the same simulation

The possibility of combining hardware designs and software in the same simulation environment opens new options and improves significantly the flexibility of verification processes as well as characterization time of electronic designs. A practical method to realize this is developed and presented in this work for the case of a real Graphics Processing Unit IP.

Nowadays electronics designers and manufacturers compete in an increasingly faster race to be able to provide the best and most efficient solutions to the market's expectations. The easiest example is the tendency of smartphone designers to provide a brand-new mobile phone model every year to meet consumers' demand. To meet these tighter and tighter deadlines, these companies need to find new ways of designing and verifying their products faster and more efficiently. In this context enters the work presented in this thesis: One of many possible solutions to improve the verification time of a hardware unit/block.

Digital electronic circuits are commonly designed and modelled using Hardware Design Languages (HDLs), which are similar to computer languages such as C or Java, but different in the sense that HDLs actually describe the physical layout and connections of a digital circuit. These HDL designs can be simulated to verify their correct performance and characteristics with very high detail but, at the same time, this type of simulations are costly in terms of computational time and resources, due to the nature of the magnitudes and mechanisms being replicated on the computer running the simulation.

On the other hand, software is written in computer languages directly, compiled to machine language and run sequentially by computers, in a much faster and efficient manner. Therefore, what if the best of the two could be combined to simulate a digital design in which only a specific internal block is described in a HDL while the rest of the design is a software program? This would allow to reduce the simulation time of that block greatly, while at the same type preserve the accuracy that a simulation of a HDL design can provide.

This thesis work is based on a specific part of Arm's next-generation Mali Graphics Processing Unit (GPU), for which a solution for mixing hardware and software in the same simulation is proposed. For this specific case, such mechanism will allow to improve the development and testing time of new features for a Mali hardware IP, while at the same time open new use-cases for future work in this direction.

Table of Contents

List of Figures

List of Tables

Abbreviations

ABI Application Binary Interface

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

CAD Computer-Aided Design

CPU Central Processing Unit

DPI Direct Programming Interface

DUT Device Under Test

EDA Electronic Design Automation

FIFO First In First Out

FMI Functional Mock-up Interface

GLES OpenGL for Embedded Systems

GPU Graphics Processing Unit

GUI Graphical User Interface

HDL Hardware Description Language

HLS High-Level Synthesis

IP Intellectual Property

LOC Lines Of Code

RTL Register-Transfer Level

Tcl Tool command language

TU Texture Unit

VHDL VHSIC Hardware Description Language

VLSI Very-Large-Scale Integration

Chapter 1

Background

In this introductory chapter several terms and concepts, which are the basic foundation from which this thesis work is built, are presented and explained. The first section will focus on the Graphics Processing Unit (GPU), discussing its differences with a Central Processing Unit (CPU) and presenting an example very related to this work: the Mali-G72 by Arm. Following this, the concept of co-simulation is introduced, for which the definition of a hardware model simulation and software models for hardware designs need to be provided and described first. Finally, a collection of previous works that are relevant for this thesis are shown and discussed, from which abundant information for both implementation options and metrics for the results can be extracted.

1.1 The Graphics Processing Unit

A GPU is a specific processor optimized for those calculations primarily needed in graphical operations. These calculations are usually parallelized and performed in floating-point representation, requisites important for rendering 3D graphics. This specific hardware architecture allows for a processing power several orders of magnitude higher than a conventional CPU in the previously mentioned tasks [1].

The GPU used to be a hardware component exclusive to static machines and computers, as its use usually implied an increased complexity in the system as well as a higher power consumption (in opposition with superior performance for graphical tasks). However, technological advancement has allowed the miniaturization and power optimization of the GPU, turning it into a suitable solution for superior graphical performance in mobile devices like smart-phones.

1.1.1 Functional differences between a GPU and a CPU

A CPU is a general-purpose circuit capable of executing common computer instructions, such as arithmetic and logic operations or writes/reads to the system's memory. CPUs exploit the concept of pipelining, as well as other mechanisms, to provide a very high instruction throughput with a reasonable power consumption. It is the basic building block of a modern computer system due to the fact that, in principle, it is optimized to perform any operation and has access to every subsystem of the main architecture.

The vast majority of programs are typically written in a manner such that its operations are executed sequentially [2]. Due to this, a single CPU is optimized to run this type of tasks as fast as possible. This is a very important obstacle for improving the computational efficiency of programs or algorithms that are inherently paralelizable, such as 3D graphics rendering, machine learning or cryptography. GPUs are a solution to this inconvenient, as their architecture is designed to handle efficiently parallel tasks.

A typical CPU architecture contains few cores, accompanied with cache memory to make the handling of several software threads simultaneously feasible. On the other hand, a GPU is designed with hundreds of cores capable of processing thousands of threads at the same time. This means an occasional advantage for the GPU with respect to CPU, which will be able to accelerate certain software over a factor of 100 compared to a CPU. Furthermore, a GPU will achieve this speedup being more power and cost efficient than a CPU [3].

1.1.2 An example of GPU architecture: The Mali-G72

With the aim of introducing the reader to a system similar to the one in which this work has been based on, as well as to show an example of a GPU architecture, the Mali-G72 is presented in this section. This Arm's mobile GPU is one of the most popular commercial solutions for a wide range of devices. Only in 2016, more than one billion Mali GPUs were shipped. Depending on the main requisite for each system, different Mali models are available, each with specific emphasis on high performance, high area efficiency or ultra-low power consumption. The Mali-G72 is the most powerful model in the "High performance" family [4].

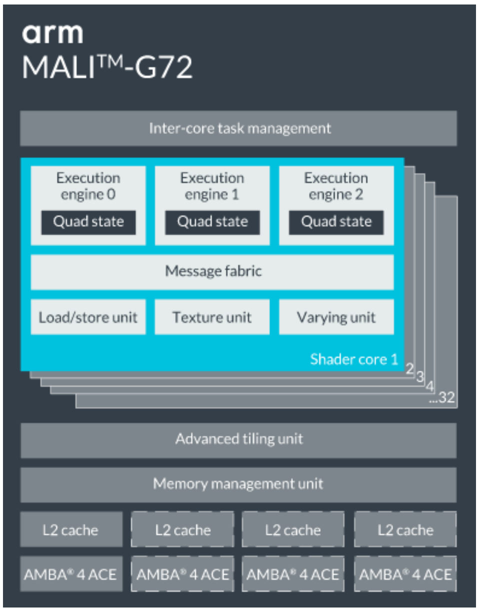


Figure 1.1: Mali-G72 high-level architecture - Arm [4]

In general terms, this GPU is composed of up to 32 cores, each of which contains several execution engine blocks, a message fabric and some specific units, like the Texture Unit (TU). Along this array of cores, other caches, interfaces and blocks complete the architecture of this system. As it has previously been presented, this system exploits the inherent parallelization of tasks such as graphics rendering and machine learning by employing several computing cores for running many threads simultaneously.

The work of this thesis is focused on the top-level interface of the TU for the next-generation Mali GPU.

1.2 Hardware-software model co-simulation

In this section, two of the most widely used principles for simulating hardware designs, hardware models simulation and software models, will be described in order to later introduce the concept of *co-simulation*, the general idea behind this thesis work.

1.2.1 Hardware simulation

Digital hardware designs are typically written in a Hardware Description Language (HDL), such as VHSIC Hardware Description Language (VHDL) or Verilog. These computer languages are created to allow a designer to describe the behavior of a digital electronic circuit from a higher-level perspective, while still being precise. Designs written in a HDL may undergo processes like synthesis (for netlist generation) or compilation for simulation in specific tools designed for these languages.

Looking more closely into the latter, behavioral simulation for HDL compiled code is typically realized using an event-driven approach. In an event-driven simulation, time is represented as an integral multiple of the resolution, which is the minimum time unit in the simulation. Any simulation contains two phases: *Statement execution* and *Event processing* which are executed in every time step. In the former phase, for each event (a statement containing a signal in a VHDL sensitivity list, for example), a new value and the time when a target signal will be changed are created and stored in a queue, forming a value/time pair called *transaction*. In the *Event processing* phase, transactions happening in the current time are removed from the queue. For each removed transaction, if it represents a value change of a signal, it will be treated as an event, triggering the signal update [5].

Event-driven simulation is the most popular mechanism for simulating hardware designs due to the accuracy it offers and moderate efficiency in computational expense. However, this approach is still costly, making the simulation of very big or complex systems slow.

1.2.2 Software models for faster hardware simulation

Software models are programs written in any computer language which mimic the intended behavior (and optionally other features) of a digital system with different degrees of accuracy. These are very valuable for exploring design options, simplify verification of certain parts of a system or be used as a golden reference for hardware verification. Software models of a hardware design can be classified according to the level of abstraction they provide [6]. From lowest to highest:

- **Continuous time**. Operations are described as continuous actions, p.e. the behavior of an analog electronic circuit using differential equations.
- **Discrete-event**. Activities are grouped into time points called *events*. Changes in the inputs of a digital circuit will trigger toggles in intermediate nets resulting in a change of the outputs in times that may be irregularly spaced. This level of abstraction provides relevant simulation information such as propagation delays in combinational logic, glitches and clock edges. This is a model-wise analogy to event-driven simulations for digital circuits, described in the previous section.
- **Cycle-accurate**. A cycle-accurate model uses clock edges as the basic timing unit. These level of abstraction can't provide propagation delays or glitches, as all activities between clock edges will be grouped at the positive or negative edge. Therefore, activities happen "immediately", or after an integer number of clock cycles. This is usually the abstraction level chosen for use as a golden reference in verification of hardware designs for its balance between accuracy and complexity.
- **Instruction-accurate**. In this level of abstraction, activities are expressed in steps of "one microprocessor instruction". Effectively, every step of such simulation contains several cycles of processing. This is used to simulate complex software, such as operating systems. Since the time count in these simulations is in terms of instructions, the only possible mapping to real time units would be to assign a processing time (p.e. in clock cycles) for each instruction.
- **Transaction-accurate**. If the system is too complex for an instruction-accurate approach, transaction-accurate models are used. In this type, the behavior of the system is expressed as interactions between components. These interactions are called *transactions*. The time for every transaction depends on what is actually being modeled, but may range from thousands to millions of clock cycles. These models are most common in early phases of a design, to explore different options and evaluate how they affect the system as a whole.

The main reasons encouraging the use of software models over HDL descriptions for a given hardware design are:

- *Faster implementation time*, as software languages are, in general, less verbose and demanding for the designer.
- *Higher flexibility*, since options for configuring the behavior of the modeled system can be toggled very easily.
- *Improved computational cost of simulations*, as the resulting "simulation" is simply an object code that can be directly executed by a computer, in opposition to HDL compiled code for running simulations in a dedicated event-driven simulation software.

In opposition to these advantages, hardware models generally cannot replace HDL described systems when:

- A high accuracy for the behavior of the system is required.
- The design is to be translated into a netlist (except for some High-Level Synthesis (HLS) languages, such as Catapult-C). They may only be used for simulation/verification purposes and not for physical implementation.

1.2.3 Co-simulation

For several reasons, which will be introduced shortly, it is very convenient and interesting to simulate simultaneously software and hardware models under a common framework. Such simulation, in which one or several pieces of software and hardware are communicated in some manner to provide a single result output, is called "Hardware-software model co-simulation".

This mechanism allows mixing these different entities under a common testbench, while still moderately preserving the loose coupling between them. In other words, if the co-simulation framework is well designed, most blocks can, for example, be replaced for an updated simulation with a new version of a hardware component while maintaining the former software. Co-simulation can therefore be very interesting for design exploration, where it provides a fast solution for testing new features and evaluating how they affect the system as a whole.

Another scenario where the use of co-simulation can be advantageous is in full-system integration, since nowadays virtually every electronic system is a combination of software and hardware. A possible flow would be to verify software and hardware separately, using the corresponding methods and tools for each, and then use co-simulation to verify that the two entities can work together as expected.

Finally, co-simulation also allows a more efficient parallel development of the hardware and software components of a system, which is directly beneficial for shortening development times nowadays that time-to-market is becoming shorter and shorter [7].

Since hardware designs are typically verified in tools based off event-driven simulation, the most common approach to enable co-simulation is integrating (via different solutions) the capability of running external object code in some HDL simulation tool. Some examples of these solutions will be presented and discussed in the following section.

1.3 Previous works on co-simulation environments

Simulations combining hardware and software models can be implemented in different ways, of varying complexity and results, depending on the engineering needs and available flows. One example is the use of scripting languages in Electronic Design Automation (EDA) tools to interface with Register-Transfer Level (RTL)-level signals in the own tools' default simulation environment. One example could be using the language Tcl inside Xilninx's Vivado simulator to generate and drive stimuli to a hardware block for unit-level testing.

Another typical example in the industry is SystemVerilog DPI (Direct Programming Interface), an interface that allows to communicate SystemVerilog code with foreign languages (typically C, C++ or SystemC, among others).

Apart from these examples, several academic works further prove the value of co-simulation. For instance, Stefano Centomo et al. present the results of a tool-independent environment for co-simulation using SystemC to describe the hardware part of the design and Functional Mock-up Interface (FMI) as the bridge between the RTL description of the design and the rest of the system modeled in software using the tool Modelica. In this case, the main metric used to determine the usefulness of this approach is the average CPU time that the environment takes to simulate a second of simulation time which, in this case, yields very promising results. However, the authors point out some shortcomings in FMI, such as the fact that event-driven simulation is not possible [8].

On a similar note, but more closely related to this Master's thesis, the work of Dominik Widhalm et al. presents the use of Perl for creating test stimuli and environment templates for the simulated verification of SystemC described hardware designs [9]. Four different methods are compared: code generation, which becomes complicated because for some basic constructs a C code has to be developed; code conversion, that is, to convert Perl code directly into C code, which presents the problem of outputting unreadable C code; code transformation, similar to the previous but using the intermediate constructs of the Perl interpreter and the embedding approach, that takes on running the actual Perl script in a real Perl environment running inside the SystemC simulation domain. To compare these different methods, a single test case is used: a linear voltage ramp used as input to an ADC, letting the simulation build the respective histogram for every voltage step, and then merging all the histograms to calculate the static parameters of the ADC. The results are varied, but an important contribution from this work are the metrics used for comparison of the different methods:

- **Lines Of Code (LOC)** required to implement the test case.
- **Size** of the resulting executables.
- **Runtime**: Total simulation time of the test case.

This study is concluded stating that the preferred option should be the embedded approach, as it is re-usable in every possible test-case, only having to change the Perl script to be used in that case. The only disadvantage is the fact that runtimes are longer than in other methods due to including the whole Perl environment embedded on SystemC. However, this inconvenient might be overshadowed by the overall saved time thanks to re-usability.

An additional interesting work, from industry in this case, is that of Michael D. McKinney, Senior Member of the Technical Staff in Texas Instruments [10], who describes the integration of Perl, Tool command language (Tcl) and C++ into a simulation-based verification environment for Application-Specific Integrated Circuit (ASIC) designs. In such proposed environment, simulations that combine HDL designs with software models (one written in Perl, in this case) are executed in the following manner: The HDL simulator is invoked, which then loads a core Tcl subroutine, responsible for several front-end tasks related to the simulator, Graphical User Interface (GUI) and VHDL verification environment. Later in each test-specific file, another Tcl routine is invoked again to execute a Perl interpreter, which will run the Perl-written reference model to generate the stimuli and expected output of the RTL unit under test. Tcl is then responsible again for driving these stimuli to the RTL design being simulated, control the simulation time execution (interestingly for the sake of this thesis work) and read the output generated, in order to compare it with that of the golden reference.

This approach, which yields very interesting results, does however present several inconveniences that are very relevant to discuss. First, since Perl and Tcl are different languages, they cannot share the interpreter, or the internal variables that each use. This was solved in [10] using files to write and read data between processes. However, this creates a considerable overhead due to the creation, writing, reading and parsing of the files. In addition, both processes need to agree on the internal format of the files, which can be a feature difficult to implement. Secondly, this approach grants complete control of the simulator to Tcl, which is a good solution for automating the flow and tests. However, this also means that the HDL debugging features that are present in the simulator are not available, creating an additional layer of complexity for the debugging of the RTL design. Finally, other issues arise as a result of choosing Tcl as the controller of the flow, along with the way this integration is realized. One is the fact that syntax errors in the Tcl scripts cannot be detected until runtime, due to the interpreted nature of the language. Another is that Tcl will sometimes be busy many real-time seconds operating with files (which are sometimes very long and heavy). This causes the simulator to appear "hung" for some time. Finally, once Tcl has completed the needed operations for a specific phase, it runs time steps for the simulation with the appropriate command. Since the simulator updates its GUI and other elements every time a "run" command is executed, if the time-step for each call to this command is too small, the GUI will keep refreshing every time, causing again an additional overhead to execution time as well.

Chapter 2

Challenges and objectives

In this chapter the specifics on the challenges and objectives of this thesis work will be presented. For this matter, the main actors have to be described, namely the Mali GPU software model and Mali's TU block. Once the scenario is presented, different options to implement co-simulation will be discussed and compared. Finally, the chosen option that complies best with all constraints and was found the most suitable is shown, with a motivation for this decision.

2.1 Main actors in this work

Before presenting the objectives and challenges of this work, it is necessary to briefly describe the two main actors on which this study is built upon: The software model for Arm's Mali GPU and the RTL implementation of the TU, a hardware unit which is part of the hardware micro-architecture of this GPU.

2.1.1 The Mali model

The Mali model is a C++ software developed by Arm which mimics the behavior of the Mali's GPU hardware, taking advantage of the flexibility that software can provide in opposition to HDL. For future reference, it is relevant to highlight that its abstraction level can be considered as in between the *cycle-accurate* and *discrete-event* levels, following the classification presented in section 1.2.2. It is also commonly referred to as *cycle-approximate*.

The model exposes a top-level interface for other applications. This interface, in the form of a C library, provides functions for creating an instance of the model, configure it or perform tasks that the top-level of the GPU would receive in a real implementation, such as processing certain graphics jobs. Once the model completes a top-level task, it will output a file containing the memory addresses and data that the Mali GPU would have written for that task in a real system. This can also be referred to as *memory dump.* The objective of the proposed co-simulation solution is to be able to run these tasks on the model to simulate the processing of graphics jobs just in the same way as it is "traditionally" done in the model, with the particularity of using the TU's RTL in place of the model's counterpart.

The model is one of the main actors in this work, and its operation and special characteristics determine to some extent how the final solution was designed and the main principles behind its performance and behavior. For instance, the model will read values from the previous cycle at the rising edge of the next, will process that data, and finally provide an output in the falling edge of that same cycle.

2.1.2 The Texture Unit (TU)

The Texture unit is a hardware block contained in the Mali micro-architecture responsible for processing several graphics operations with, surprisingly, textures. Jobs received by this unit are processed in a pipelined manner with the order for each job not necessarily being preserved in the output. That is, a job #0 which arrives before another job #1 may be finished by the pipeline after job #1 is.

The RTL implementation of this unit is also one of the major actors in this work, as its inclusion in a co-simulation with the software model for the rest of the GPU has been the main motivation for this thesis work. More specifically, its interface is where a great part of the co-simulation work has been carried out, so it is relevant to mention in a summarized manner the different sub-interfaces it exposes:

- A **control** interface, with signals to monitor the state of the unit and receive basic configuration.
- A **message input** interface, that receives texturing messages from other parts of the GPU.
- A **message output** interface, that sends texturing result messages to other parts of the GPU.
- A **shared data** interface, that reads shared data from other entities in the GPU micro-architecture that may be required in texturing operations, such as state, current operation status, etc.
- Two **Advanced eXtensible Interface (AXI)** interfaces for requesting data from other memories in the GPU to internal caches.

Finally, it is important to note that this hardware Intellectual Property (IP) is described in HDL with SystemVerilog. This fact will condition the options available for co-simulation, since the execution of object code is also conditioned by the event-driven simulation tool which, at the same time, is constrained by the possibilities existent in the specification of the HDL in which the hardware design being simulated is described.

2.1.3 Objectives of the proposed co-simulation environment

As was mentioned before, software models of a hardware design are a key resource for tasks like design exploration, behavioral emulation of the real system or performance evaluation. For the case of Arm's Mali, the C++ model allows to realize these tasks, as well as testing early features and others advantages. A different approach to software models for realizing these tasks would be simply designing a hardware block and simulating its behavior and performance once implemented. However, the development time penalty for every iteration in this latter case is much higher, and so is the computational cost of the simulation, which can become a real problem for top-level tests.

The main motivation for this thesis work is to explore whether the best of the two worlds can be combined: the accuracy of a typical RTL simulation (event-driven) with the speed and flexibility of a software model run. To achieve this, it is necessary to create a co-simulation environment where, in this specific case, the TU in the Mali model has been effectively substituted by its corresponding RTL block. The expected outcome of such setup is a testbench where both inputs and outputs are those that the Mali model would require/produce standalone but with the added detail/accuracy in the TU part of the design that would be expected from a GPU's RTL top-level simulation, with a much lower computational cost in comparison to running the equivalent simulation for the RTL description of the whole system.

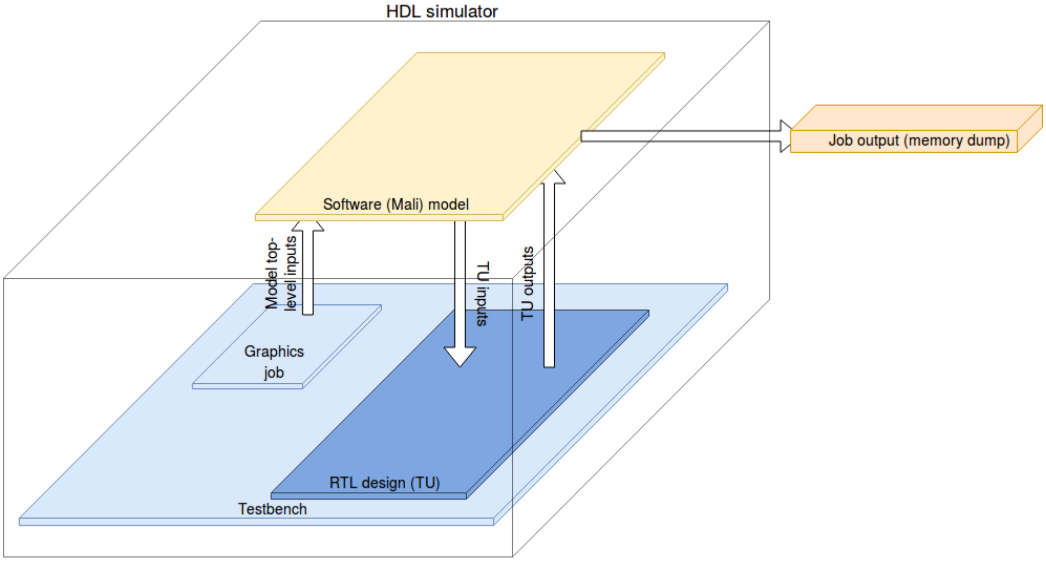


Figure 2.1: High-level diagram of the co-simulation environment

The co-simulation environment that this thesis aimed to create is specific to the case of executing a Mali model top-level run (such as a graphical job of some kind). The main particularity, and the main reason to speak of "co-simulation" is that the TU class in the model will effectively be replaced with its RTL block, in a way such that the model's execution hands over the stimuli to the TU's RTL (instead of simply letting the TU class process them), which will then, via the event-driven simulator (Cadence's Xcelium in this case) process the texturing messages and output texturing result signals. These output signals will be read by the model, which will process them and use them for the rest of the tasks necessary in the graphics job that was run. At the end of the job execution, the model will output the written memory by the result of the job in a file. In other words, the co-simulation will run a job in the Mali model, written in C++, whose TU is actually not software, but a RTL block, written in SystemVerilog. This idea is shown in the high-level representation of Figure 2.1, where synthesizable RTL blocks are depicted in dark blue, non-synthesizable RTL in light blue, software in yellow and output files of the simulation in orange.

To enable the effective substitution of this unit's RTL in the model execution, the data for each interface in the TU (see section 2.1.2) has to be handled in the final co-simulation solution in a way such that the model can process data generated from the event-driven simulation of RTL and, on the opposite, that the RTL can process data sent in from the C++ object code execution.

Moreover, since the model execution cannot handle discrete-events with as high precision as an event-driven simulation, timing mismatches and delays in data transitions should be taken into account if a seamless co-simulation is desired. These may affect performance metrics or the total cycle-count of a complete graphics job run, but shouldn't make notable differences in the functionality and data correctness.

An extra possibility is that the model may obviate hardware elements or subsystems that are necessary for the real GPU to work, but maybe not essential or trivial if the system is inspected in a pure functional level, as in the case of most software models. For example, a 256-bits wide bus in RTL may be represented as a class in C++ whose members are the different fields of the original message. Therefore some interface logic and/or data storage may be necessary in any of the two sides (software or RTL) to overcome this issue.

2.2 Co-simulation options and solution

The challenges described in this chapter for implementing a co-simulation for this specific case may be overcomed via different ways. Several practical possibilities exist in order to simulate a hardware design with a software component and, inversely, run a program which interacts with a hardware design simulation. The options are very diverse, each with its own advantages and disadvantages. Therefore, only those actually considered for this work will be briefly introduced in this section, with a motivation on why it might or not be the best solution for this concrete scenario. In the end, the chosen alternative will be presented, along with the reasons why it is considered the most suitable.

2.2.1 Description and comparison of possible options

Three candidate methods to implement the proposed co-simulation environment are presented now: Tcl, SystemVerilog's Direct Programming Interface (DPI) and DPI-C and SystemC. These are an interesting set of options, since each communicate with RTL designs and/or software models in a different way, and with different advantages and disadvantages.

Tcl

Tcl is a high-level, interpreted programming language which was developed by John Ousterhout as a scripting solution for his Very-Large-Scale Integration (VLSI) layout tool *Magic* [11]. Since this was one of the very first software of its category, and due to the open-source licensing that both Magic and Tcl were released with, Tcl has become a *de facto* standard for command scripting in most popular EDA and Computer-Aided Design (CAD) tools. Apart from being multi-platform, it can be used for many different tasks, becoming an interesting candidate for scripting an entire work flow, including verification under simulation [12].

A moderately successful example of using Tcl for co-simulation of a hardware and software model in the same environment was presented in Michael D. McKinney's work, in Section 1.3. This solution had some inconveniences, mainly because of the problem of sharing data between Perl and Tcl, but is very illustrative of how Tcl can be a powerful language for the purpose of co-simulation.

From the experience of the author of this thesis work, and the research conducted, Tcl presents two major problems:

- The **real-time overhead** it introduces if used as the driver/controller of the simulation. Applicable in general to any Tcl-driven simulation flow. Caused by the fact that Tcl interacts with the HDL, event-driven simulator every time it needs to change the value of a signal or control simulation time.
- The **data coherence with other computer languages**. This is not a problem if Tcl is used in a typical case (driving stimuli of a RTL Device Under Test (DUT) and collecting the output for processing) but is very relevant if software models become an actor in the simulation, as they are commonly written in some other language.

The first problem can be overcome if it is taken into account during the design phase of the Tcl scripts that will control the simulation. The main idea is basically to try to reduce as much as possible Tcl commands that interact directly with the HDL code, such as *force*, *run*, etc. To achieve this, some testbench-exclusive RTL units may be necessary. An example-comparison of this approach would be to improve the trivial algorithm:

1. Drive every input signal to the DUT using *force*.
2. Execute the *run* command for "clock period" seconds.
3. Repeat every clock cycle.

with the more efficient option:

1. Drive all stimuli in an ordered manner to a behavioral First In First Out (FIFO) memory, using *force*.
2. Execute the *run* command for "simulation-length" seconds. The FIFO will output stimuli signals in the desired manner every clock cycle.

As the reader will imagine, the faster approach requires the implementation/use of a behavioral FIFO, which in most HDLs is, however, not a complex task. With this improvement, calls to *run* are dramatically reduced and, therefore, the simulation time.

Regarding data coherency, the difficulties seem to be more complex. In [10], the solution of using files to share data variables between languages comes with a huge performance impact, as well as complications in the implementation. One could rapidly think that there might be a better way to communicate Tcl with some other language. In the specific case of this thesis work, the challenge would be to interface a C/C++ software model with/from Tcl scripts. According to Tcl's wiki [13], this can be somewhat easier than for other languages, since Tcl is a C library. It is then possible to create and call a Tcl interpreter from a C program, and also to wrap C/C++ code to make it callable from a Tcl script.

In summary, it is actually possible to use Tcl as a means to implement a co-simulation environment with a software and hardware model, but several non-idealities will come at a cost.

SystemVerilog's DPI and DPI-C

According to the IEEE's SystemVerilog language specification [14], DPI is an interface between SystemVerilog and some other foreign programming language in which two layers exist isolated: the SystemVerilog layer and the foreign language layer. Any programming language may be used as the foreign language, since the SystemVerilog side is absolutely indifferent in that sense. However, SystemVerilog currently has only one foreign language layer defined: for the C programming language.

DPI allows importing functions or similar constructs from the foreign language and, similarly, to export SystemVerilog subroutines to the foreign language: tasks and functions. For imported and exported subroutines, a rich subset of SystemVerilog data types is supported by the interface, being most of them C-compatible types, packed types and user-defined types built from types of the two previous categories.

For the case of a software model written in C++, it is possible to import and export subroutines, always assuring that in the language boundary only C types and conventions are used. If C++ features need to be supported by DPI for some reason, linkage semantics and the whole foreign language layer would have to be defined for it. This includes defining how actual arguments are passed to SystemVerilog, how they can be accessed from C++, how SystemVerilog specific data types are represented in C++ (by translating them to and from C-like types), etc.

Another important fact to take into account in this case is that some DPI imported subroutines (from the foreign language to SystemVerilog) will require that the context of their call is known. This is the case for subroutines that will call exported subroutines (from SystemVerilog to the foreign language) or access SystemVerilog objects as part of their execution. For this situation, the subroutines have to be specified as *context*, triggering a special instrumentation in their call instances that can provide the mentioned context. This however, presents the downside of decreasing simulation performance, and will affect directly the intended setup for this work, since the main idea is that calls to the top-level interface functions of the Mali model happen from a SystemVerilog testbench with those functions as DPI imports, which will then call functions imported in the model from SystemVerilog's subroutines to interact with the RTL testbench.

SystemC

SystemC is a set of C++ classes and macros that provide a transaction-level interface for modeling and verification purposes of hardware designs written in a HDL. This is a very interesting solution for models written in C++, as SystemC can be used directly as a means to communicate with a RTL hardware design.

The interoperability with SystemVerilog is, in principle, quite straightforward. SystemC modules can instantiate SystemVerilog modules and vice-versa, effectively creating a single module hierarchy with mixed languge components. Similarly, SystemC can call SystemVerilog tasks and functions, and SystemVerilog can call SystemC methods, which matches most design flow requirements and also provides a higher simulation efficiency. In fact, even SystemC signals can be bound to SystemVerilog ports and vice-versa [15]. Another advantage is that, since SystemC is part of C++, no extra interfaces or components are necessary between the model and HDL design.

However, despite all the previous positive facts about the use of SystemC for mixed simulation purposes, several complications exist:

- Because SystemC typically uses SystemVerilog's DPI as the communication bridge with SystemVerilog, and DPI defines a C interface, but not a C++ one, DPI cannot be used to traverse SystemC hierarchy, since handles to instances or objects are not easily possible.
- SystemC offers a transaction-level abstraction, which provides a lower detail when compared to the SystemVerilog scheduler.
- Because of the previous, an extra service layer is necessary on top of DPI to take care of the synchronization between the two "worlds".

Regarding the software side, yet more disadvantages exist:

- To use SystemC in a software model, the model has to be designed considering the support for these classes or, at least, providing some compatible interface.
- Currently every EDA vendor uses proprietary binaries for SystemC. This is an inconvenience if the simulation setup needs to be ported often, or if a more general work flow is in place.
- Following the stated above, each EDA tool will have compatibility for SystemC only for specific gcc compiler versions, which limits the options for the software model implementation.
- If a robust and long-lasting interface with a HDL simulation is desired, SystemC may again not be a good option, as it means using the C++ Application Binary Interface (ABI) directly. The problem is that C++'s ABI has changed several times historically, and this represents a risk in this case, as the compatibility of the SystemC interface can be lost when re-compiling for a new version [16].

Comparison of the three options

The three options previously presented to implement co-simulation in this work have different advantages and disadvantages, which the reader will probably have already detected in each one's description. To serve as a quick summary, pros and cons of each option are presented in Table 2.1, Table 2.2 and Table 2.3, for Tcl, SystemVerilog's DPI and SystemC respectively.

Tcl	
Advantages	**Disadvantages**
De-facto in EDA tools	Big overhead for simulations
Discrete-event level	Complex embedding in C++

Table 2.1: Advantages and disadvantages of Tcl as a co-simulation tool

SystemVerilog's DPI	
Advantages	**Disadvantages**
Widely supported by EDA tools	-
Discrete-event level	-
Computational overhead is low	-
Well-defined interface for C language	-

Table 2.2: Advantages and disadvantages of SystemVerilog's DPI as a co-simulation tool

SystemC	
Advantages	**Disadvantages**
Low computational overhead	Proprietary binaries for each tool
Easy integration with SV	Transaction level
-	C++ ABI changes
-	Limited compatibility with gcc versions

Table 2.3: Advantages and disadvantages of SystemC as a co-simulation tool

2.2.2 Choice of co-simulation tool: SystemVerilog's DPI-C

In the end, SystemVerilog's DPI, with its definition of the foreign language interface for the C language, DPI-C, was chosen as the tool with which the proposed co-simulation environment would be implemented. The main reasons for this decision are:

- The interface between C and SystemVerilog is well defined and consistent and since no extra elements (like the case of SystemC for C++) are introduced, the feasibility of this solution is assured so long as the C standard and SystemVerilog's DPI-C definition remain as they are at the moment of writing this document.
- SystemVerilog, with DPI-C, is supported by most, if not all, EDA vendors, and is fully-supported with Cadence's Xcelium simulator, which was used for this thesis work.
- It is the most logical option for the specific case of the Mali model, as it already exposes a C top-level interface with a set of functions to create an instance to the model, configure it and execute tasks with it, such as graphics jobs.
- In contrast with the previous, if another option was to be used for implementing a co-simulation environment between the Mali model and RTL designs, an important refactoring of the model's code would be necessary. This is completely out of the scope of this project.
- The computational overhead introduced by DPI-C, although existing due to effects like the *context* of imported subroutines, is much lower than in other solutions, permitting an efficient implementation of the co-simulation environment.

After presenting the reasons for selecting this tool, some important and more detailed background information about the data equivalency between SystemVerilog and C and SystemVerilog functions and tasks will be given hereunder, as they represent a central part of this work and the basic theory behind the co-simulation environment implementation.

Data equivalency between languages

The equivalency between data types of SystemVerilog and C is very well defined in DPI. Specifically, Annex H in SystemVerilog's language specification is dedicated to the foreign language specification of C for DPI, and all the details on data representation are clarified. However, to simplify as maximum as possible the implementation of the languages interface part of the co-simulation environment created in this thesis, only the basic types mapping was used. These one-to-one relations, extracted from the SystemVerilog's language specification, are shown in Table 2.4. It is worth noting that the encodings for the *bit* and *logic* types are usually given in a separate file, *svdpi.h*, which is implementation-dependent and typically provided by each EDA vendor with their HDL simulators.

Rules for functions and tasks

Functions and tasks of SystemVerilog, also collectively referred to as "subroutines", allow the division of large procedures into smaller ones. Both can contain *input*, *output* or *input/output* arguments, and will have visibility of variables and signals in their same scope or module. These constructs are the main enablers of the

SystemVerilog type	C type
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void *
string	const char *
bit	unsigned char
logic/reg	unsigned char

Table 2.4: Mapping of SystemVerilog and C data types [14]

communication between SystemVerilog and C via DPI, as they can be exported to C from SystemVerilog, and then imported in C as external functions. Likewise, C functions can be imported in SystemVerilog.

Following the SystemVerilog's language specification, several rules distinguishing tasks from functions exist, very relevant for this work:

- A task may contain time-controlling statements, such as a call to the simulator software to run the current simulation for a specific amount of time units.
- A function can enable other functions, but cannot enable a task, while a task may enable other tasks and functions.
- A nonvoid function shall return a single value, while a task or void function don't need to return a value.
- A nonvoid function can be used as an expression's operand, being the value of that operand the value that the function would return.

Because of these rules, and since in the suggested co-simulation environment the software model will be responsible for controlling the testbench clock and the simulator time steps, the following decisions were made regarding tasks and functions:

- Since only tasks can contain time-controlling statements, and for the sake of simplification, all exported subroutines given to the software model with the objective of **writing** values in SystemVerilog will be **tasks**.
- Because tasks can enable other tasks and also functions, but not the other way around, the top-level interface C functions provided by the software model will be imported in SystemVerilog as **tasks**.
- To improve readability and, again, simplify the setup, all exported subroutines given to the software model with the objective of **reading** values in SystemVerilog will be **functions**.

Chapter 3

Implementation and results

In this chapter, the details of the actual work developed in this thesis work are presented and explained. The solution created to satisfy the constraints given by the objectives shown in Chapter 2 is composed of several software and HDL blocks and constructs, which specifics will be explained, along with the reason behind such design. Following this, the results achieved with the proposed co-simulation environment for GPU IP development will be shown, with some metrics that are considered relevant for this case. Some of these metrics will be presented against other manners of simulating a top-level system test for a RTL block with the aim of highlighting the strenghts and weaknesses of the co-simulation solution.

3.1 The co-simulation environment

A general diagram of the final co-simulation environment that was created is shown in Figure 3.1 where, apart from the color correspondence explained for Figure 2.1 in Section 2.1.3, now RTL blocks in SystemVerilog with DPI-C functions and tasks are depicted in pink, a collection of functions in C that interact with SystemVerilog objects in red and C++ classes with imported SystemVerilog tasks and functions in green. In this proposed co-simulation environment, the testbench is a very simple block which acts only as the main simulator "container". It has some basic logic to instantiate the software model and call its top-level functions (via a model top-level wrapper written in C), as well as an instance to the RTL wrapper or, from now on in this text, *co-simulation wrapper*. The co-simulation wrapper is a structural RTL block which contains an instance to the RTL IP, in this case, Mali's TU, along with some interface logic necessary for co-simulation purposes. Finally, the Mali model has been modified to include a custom TU class, from now on, *co-simulation class*, which effectively replaces the C++ class that modeled the TU behavior and acts as an interface between the software model's TU interface and the co-simulation wrapper. It is important to note that, although testbench logic and the co-simulation wrapper both exist in the testbench, they are completely isolated from each other, being the custom Mali software model the only communication channel possible between them.

As the diagram hints, the co-simulation environment relies on DPI-C to:

1. Allow the testbench to create an instance of the software model, by using DPI-C imported functions from the model top-level wrapper.

2. Allow the testbench to call a top-level model's function to run a graphics job, using the same imported functions as before.
3. During model execution, let the co-simulation class of the custom software model call TU co-simulation wrapper's imported SystemVerilog tasks and functions to write and read signals to/from TU's RTL, respectively.

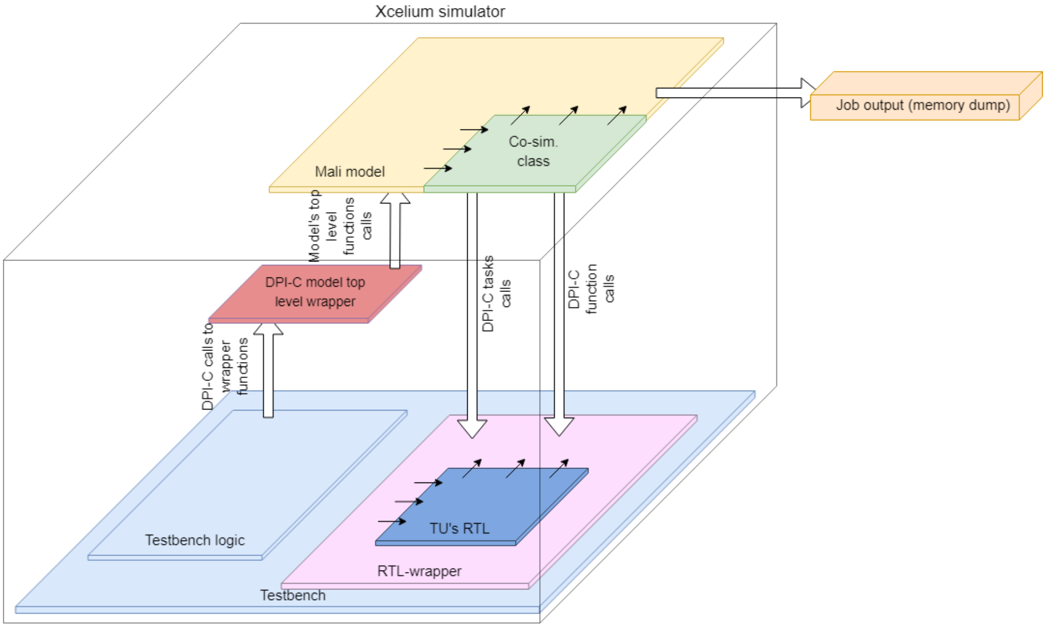


Figure 3.1: General diagram of the final co-simulation environment

The result of this co-simulation environment, in general terms, is a framework for executing top-level software model simulations in the same manner as a typical model run but with the added value of using TU's RTL model in place of the original C++ class responsible of modeling this GPU block. A more in-depth and detailed explanation of each block developed for implementing this co-simulation environment is provided hereunder.

3.1.1 SystemVerilog testbench and logic

As explained earlier, the SystemVerilog testbench is the starting piece of the co-simulation environment. This HDL block contains some basic logic to create a Mali software model instance and communicate with its top-level interface via the C-written model top-level wrapper (see Section 3.1.2), along with an instance of the co-simulation wrapper that contains the TU's RTL.

First of all, the testbench contains a set of declarations, among them, the instantiation of the co-simulation wrapper, import statements for the DPI-C functions used to interface the top-level of the custom Mali model and several variables that will later be used inside the *begin* block. The most relevant variables to mention are:

- Arguments with which the software model will be initialized. Among them, a flag specifically developed in this thesis work to state that it is the custom model for co-simulation that wants to be instantiated.
- Path to the graphics job that will be run in the co-simulation setup.
- A *print_stats* flag whose value tells the testbench whether or not model run statistics should be printed out.

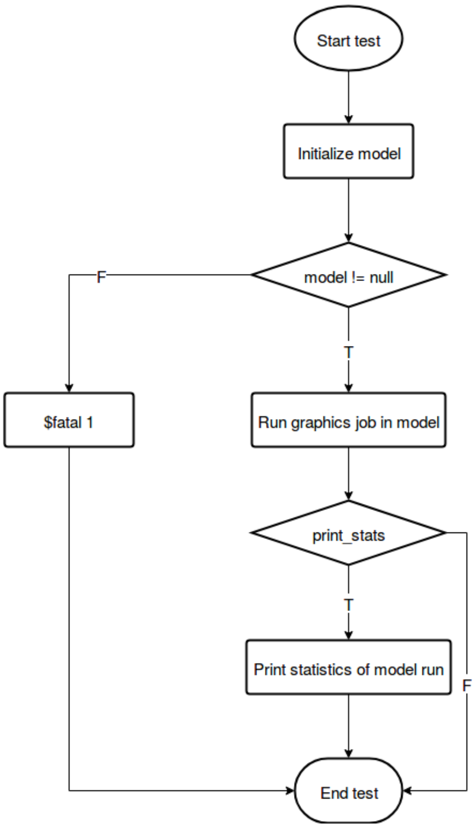


Figure 3.2: Flow chart of the testbench logic

Following the flow chart in Figure 3.2., the testbench logic works as follows. When the simulation starts, the testbench creates and initializes an instance of the software model using a C imported function from the model top-level wrapper, checking that this instance has been correctly created. If, for some reason, the instance does not exist at this point, the testbench returns a fatal error and terminates the simulation. Otherwise, the simulation will continue, and a graphics job will be run in the model using another C imported function from the top-level wrapper. At this point, the custom Mali model will run the graphics job, interacting with the co-simulation wrapper of the TU when necessary without the testbench intervention. If any execution error happens, the model is responsible for handling such situation. In practice, an error condition in the model will simply be automatically converted by the simulator in a fatal error. Having the model finished the graphics job, the testbench will then finalize by checking if the *print_stats* flag was set. If so, model statistics of the graphics job run will be required using another C imported function.

3.1.2 Model top-level wrapper

Written in C, between the simulation testbench and the custom Mali model, the model's top-level wrapper acts as a simplified DPI-C compatible interface between the model's top-level functions and the testbench logic. This block exposes several high-level C functions to the HDL testbench all of which internally call model's top-level functions. These wrapper's functions can be imported and called from SystemVerilog to create and control a model instance.

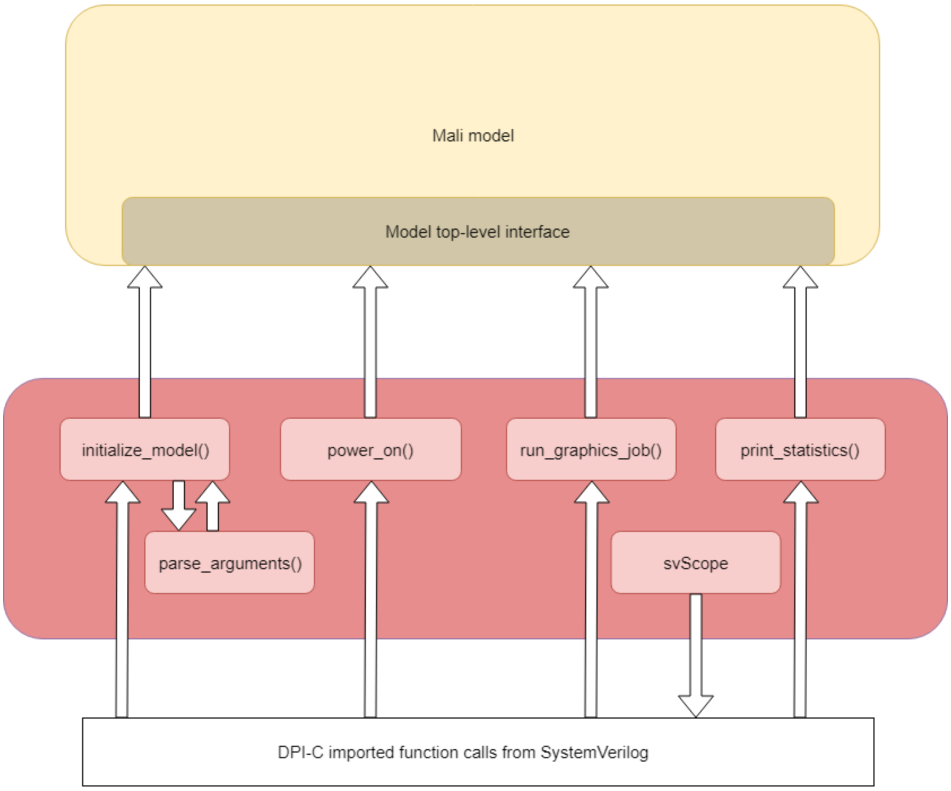


Figure 3.3: Block diagram of C-written model top-level wrapper

A block diagram representing this entity is given in Figure 3.3 and a summarized explanation of each function is given hereunder:

- **initialize_model()** is responsible for creating an instance of the custom Mali model with the specified arguments/modifiers. These arguments are parsed and converted to the format used by the model's top-level interface in the auxiliar function **parse_arguments()**.

- **power_on()** performs several operations in the model, such as register writes, mimicking the behavior of a power-on sequence of the GPU. This function was heavily used for debugging the communication between the testbench and the Mali model in the beginning of this thesis work, but has not been included in the final testbench logic, as it is not relevant for characterizing the behavior of co-simulation with the TU.

- **run_graphics_job()** encapsulates the necessary operations to request the model to compute a certain graphics job, given as a separate input file.

- **print_statistics()** will request the model to print the statistics obtained from running the graphics job with the previous function.

Finally, an important bit of this entity deserves a more detailed explanation, the DPI-C handler *svScope*. This variable exists in any SystemVerilog environment using DPI-C to communicate C object code with SystemVerilog and represents, as its name suggests, the current scope in the HDL hierarchy for which the imported C functions have visibility. Every time a C function needs to interact in some way with the HDL hierarchy (p.e. calling an imported SystemVerilog function/task or returning a result value), it will try to do so at the level specified by *svScope*, effectively being blind to any other levels of the hierarchy.

This fact brought many problems at the beginning of this thesis work, as the previously presented functions were called from the testbench and would return a value to the testbench, but also some would call SystemVerilog imported functions/tasks to interface with the co-simulation wrapper of the TU's RTL. More concretely, the *run_graphics_job()* execution would, at a given point in time, arrive to the co-simulation class, which would try to communicate with the co-simulation wrapper, using a wrong *svScope* (still pointing to the testbench). The solution to this problem was to let all of these functions modify the current *svScope* in the necessary way to ensure that the hierarchical reference pointed by *svScope* was always the intended.

3.1.3 SystemVerilog co-simulation wrapper

The SystemVerilog co-simulation wrapper was developed to act as a bridge interface between the RTL implementation of the TU and the custom Mali model or, more specifically, the co-simulation class of the custom model. Within it, a collection of logic, registers, tasks and functions are responsible for enabling communication via DPI-C to the co-simulation class, adapt the data received from the model for the TU's RTL to consume and adapt the data output from the TU for the model to consume. This entity works in such a way that for both the co-simulation wrapper and the TU's RTL the communication is transparent and direct, except for some effects introduced that will be explained later. A block diagram of the co-simulation wrapper is shown in Figure 3.4 and the details on each part of it are given hereunder.

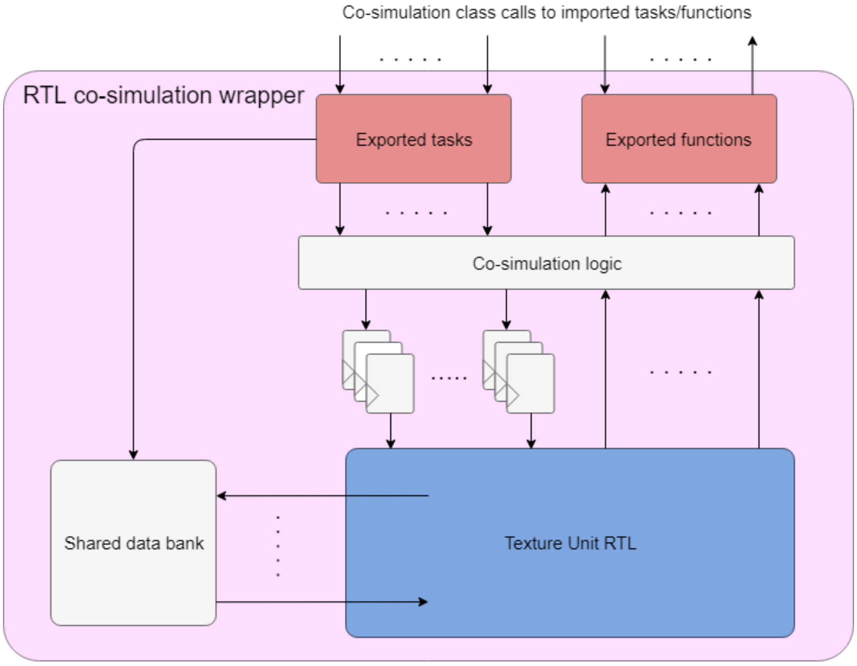


Figure 3.4: Block diagram of RTL co-simulation wrapper

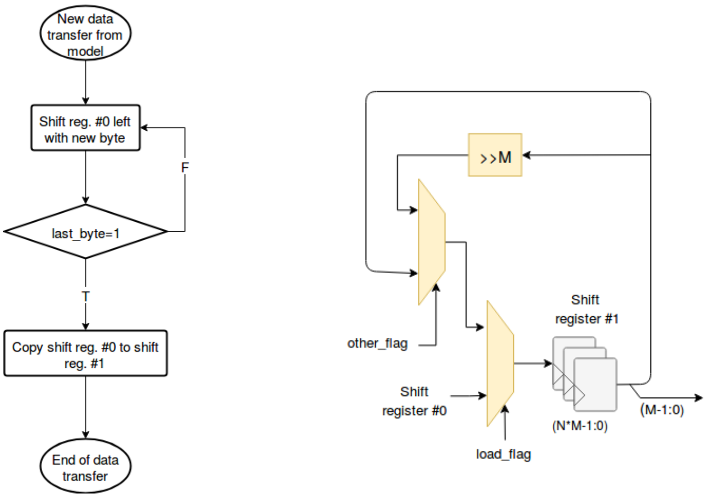
Exported tasks and functions

To implement a way for the co-simulation class to interface with RTL signals, a number of tasks and functions were implemented in SystemVerilog, one or more for each of the interfaces of the TU (see Section 2.1.2). Tasks were chosen as the mechanism for the model to write values into RTL signals, being one of the main reasons that tasks can consume simulation time, while functions do not grant this possibility. Similarly, functions were chosen as the manner for the model to read values of signals. This is essential for the case of driving the clock signal. Because the model controls the simulation, it needs to control the clock as well and, therefore, be able to advance the simulation time every time a clock edge is generated.

All of these tasks and functions are declared as DPI-C exports, so that the co-simulation class can import them as extern "C" functions and use them seamlessly from a software point-of-view. At this point, these subroutines simply allow the writing or reading of certain variables in the co-simulation wrapper.

Co-simulation logic

Data signals are treated differently in the software model and in the RTL implementation. This might yield some cases where, for example, some data may wait N cycles in the model to be available as a *struct* or *object*, while that same data in the RTL implementation would be received, in smaller words, during those N



(a) Flow chart of write mechanism of wide data signals to shift reg. #1

(b) Hardware logic for shift register #1

Figure 3.5: Co-simulation wrapper logic for wide-words data or data received in several cycles

cycles. Another case sharing the solution that is going to be presented is that of RTL input signals wider than 64 bits, which cannot directly cross the DPI-C interface with native C/SystemVerilog variable representations. The only possible approach to solve this is to divide the transmission of this type of data between model and RTL in several calls to the same task, once the data is available in the model, and use some extra logic in the co-simulation wrapper to manage that data in a way the RTL interfaces can accept. Such approach, used in every input signal affected with dedicated variables and logic in each case, is shown in two diagrams on Figure 3.5, where shift register #0 and #1 are two separate SystemVerilog variables. The decision of using two separate variables for this implementation is mainly to ensure that a wrapper-stored signal can be offloaded to some RTL unit's input port while new data for the same port is being written by the model at the same time.

When some signal affected by this situation is to be written, the process in Subfigure 3.5a will trigger. The model side of the setup will be responsible of writing only one byte in every task call, so that this new byte will be shifted left into a "shift register #0". When the last byte is written to this register, a flag will be set to indicate that the loaded value can be copied to the "shift register #1". The value stored every cycle in shift register #1 depends on some logic, depicted in Subfigure 3.5b. The output value (connected to the corresponding input signal of the RTL interface) will always be a M-1 to 0 slice of the register, being M the width of the RTL interface. Every clock cycle, the new value of shift register #1 can either be:

- That of shift register #0 (has the highest priority).
- The previous value shifted right M bits (the new data word is connected to the RTL input).
- The previous value, if for some reason the word offload has to be paused. This is the case for some of the input signals, and depends on the type of interface being used. For example, handhsake-based interfaces will require this if the slave marks "not ready" at some point of the transmission.

The last detail worth mentioning for input signals is that those whose data representation in the model are directly mappable to variables or signals in SystemVerilog are directly updated by the model but also others have to be registered before being connected to the RTL interface for timing alignment or other reasons, like the one just presented. As a result, these signals will suffer a one-cycle delay as a side-effect.

Regarding output signals from the TU's RTL, the intermediate logic of the co-simulation wrapper simply bypasses values in most cases. However, some signals which are wider than 64 bits have to be sliced in some way for the model to be able to read them in successive calls to SystemVerilog functions through the DPI-C interface. For these cases, a combinational copy of these signals is updated in the wrapper in the form of a byte vector of length $\frac{M}{8}$, where M is once again the total width of the signal in bits.

Shared data bank

While designing the part of the co-simulation environment corresponding to the shared data interface of the TU, a previously overlooked fact was found. As mentioned in Section 2.1.2, this interface reads data stored in other parts of the GPU micro-architecture required for texturing operations. In the software model implementation of this interface, the raw data structures that are read are given directly to the TU class while, in the RTL implementation, the shared data interface actually contains signals used to address a memory satellite inside the TU. The signals that would directly map to those the software model exposes to the former TU class are, in fact, one of the interfaces of this internal RTL memory satellite. Therefore, some mechanism was needed to overcome this interfaces' misalignment.

The implemented solution bypasses the RTL memory satellite and allows the custom Mali model to write the shared data directly to the internal consumer deeper into the TU's RTL implementation. A diagram of this structure can be observed in Figure 3.6. Additional hardware is introduced in the co-simulation wrapper in the form of a "shared data bank". This bank contains M lanes, one for each type of data, which at the same time contain I indexes each. This yields a total of $M \cdot I$ registers, of varying sizes depending on the data.

The access to this shared data bank can be done either by the custom software model via SystemVerilog exported tasks, to write into some specific lane and lane index, or either by the RTL data consumer, to address a specific piece of data and read it. However, the select signals of the data consumer, as well as the actual data signals are internal to the TU RTL implementation. To effectively bypass the

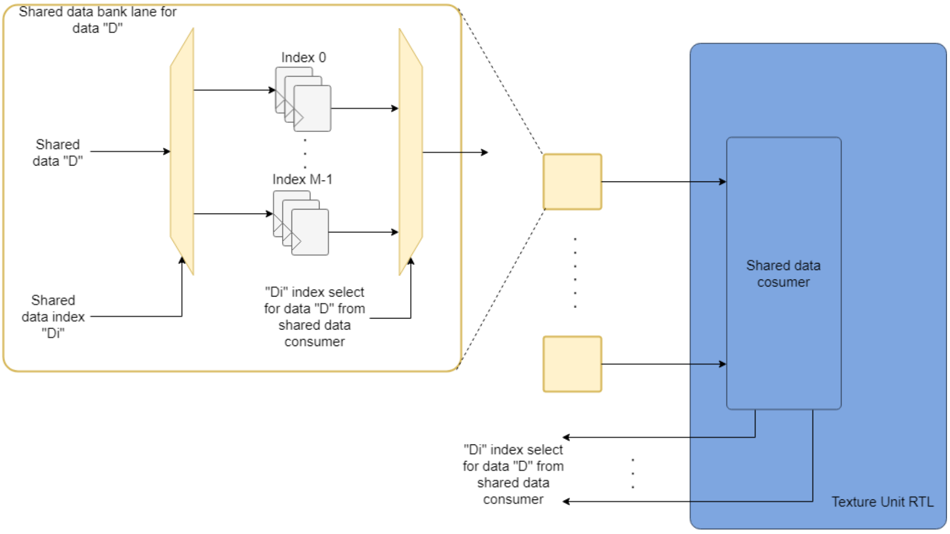

Shared data bank lane for data "D"
Index 0
Shared data "D"
Index M-1
Shared data index "Di"
"Di" index select for data "D" from shared data consumer
Shared data cosumer
"Di" index select for data "D" from shared data consumer
Texture Unit RTL

Figure 3.6: Block diagram of the co-simulation wrapper's shared data bank

memory satellite and give visibility to this signals from the co-simulation wrapper's perspective, two methods are put into place:

- For the input data signals to the consumer, their value is permanently altered during the simulation, using the SystemVerilog *force* statement. This allows to "connect" these internal signals directly to the output of the shared data bank's lanes, forcing the simulator to overlook the actual connection that the signals would normally have considering the RTL logic.

- For the select signals of the data consumer, which are outputs, their value is used in a combinational statement in the shared data bank to continuously address their corresponding bank lane using hierarchical expressions. An illustrative example of such expressions would be:

  $$u\_texture\_unit.u\_data\_consumer.select\_signal\_0$$

  Which, in this case, would return the current value of the data consumer's signal *select_signal_0*.

3.1.4 The model's co-simulation class

Most of the work put into the model during this thesis was carried out in a new class designed for the purpose of co-simulation with the TU, the *co-simulation class*. This class effectively replaces all the code related to the TU in the Mali model with a new implementation that communicates the rest of the model's units with the TU's RTL implementation. A flag in the software model was created specifically to indicate the TU interface class whether to spawn the normal TU class or the co-simulation class via a factory pattern, making the use of this setup very easy from the model side.

To interface with the co-simulation RTL wrapper, the co-simulation class imports the SystemVerilog, DPI-C exported tasks and functions provided by the wrapper as *extern C functions*, to write and read the value of RTL signals, respectively. This, along with several logic developed to solve the timing and behavioral differences between the model's and RTL implementation's TU interfaces allows for the seamless communication between software and RTL in this level of the GPU. A high-level block diagram of this co-simulation class is shown in Figure 3.7.

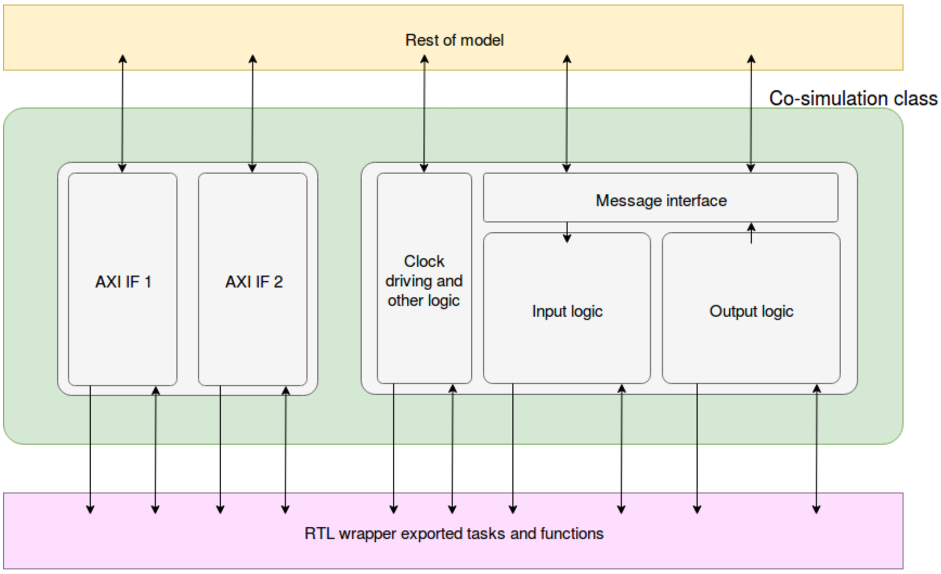
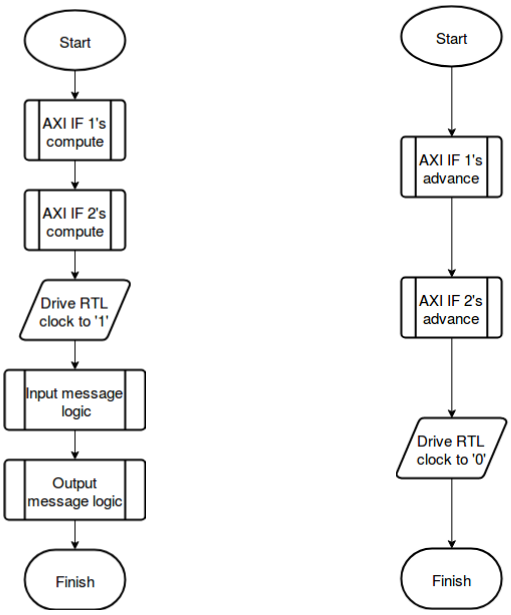


Figure 3.7: Block diagram of the co-simulation class

The co-simulation class is mainly structured in two parts. First, an entity containing the input and output logic for the message interface, the clock and other control signals' logic and the shared data interfaces (which is actually a part of the input logic for the message interface in the real implementation). The second entity would contain the instances to the two AXI interfaces (which share a common class).

All the operations performed in the co-simulation class are divided in two main methods following the model's general operation: the *compute* method, which contains all the work to be executed in a clock cycle, and the *advance* method, each corresponding roughly to the positive and negative semi-cycles of the clock, respectively. Flow charts for each of these methods are shown in Figure 3.8. An additional *reset* method is also present which, in this specific case, simply asserts the reset for a fixed time in the RTL wrapper.

The operation of the AXI interfaces will be presented at the end of this section, since their behavior is completely independent from the rest of the co-simulation class at runtime. What is most important at this point is the fact that the *compute*



(a) *compute* method **(b)** *advance* method

Figure 3.8: Flow charts of co-simulation class' main methods

and *advance* methods are responsible for driving the RTL wrapper's clock signal. *compute* will set the clock signal value to '1', and run the simulation for the time corresponding to half a clock cycle. Similarly, *advance* will set the clock signal value to '0' and run the simulation for the same amount of time. The input and output message logic are explained in detail hereunder.

Input and output message logic

The input and output message logics of the TU co-simulation class have probably been the central part of this work, as most of the timing and behavioral alignment necessary to couple the model's TU interface and the RTL wrapper was realized in this part of the design. These logics are quite complex, specially that of the input messages, as can be seen in Figure 3.9. Since the specific details of every part of the program are not really relevant for the purpose of this thesis work, some *states* are marked in the indicated flow chart, which can be used to explain in a summarized manner the operations that this part of the system performs.

However, before moving into the logic's internals, in order for the reader to understand the reason behind why this logic functions in the way it does, it is necessary to present some basic information about the input and output message interfaces. In general terms, both are typical *handshake* interfaces [17], in which:

- A master generates a *valid* signal to indicate when a new data is available.
- A slave generates a *ready* signal to indicate that it can accept the data.

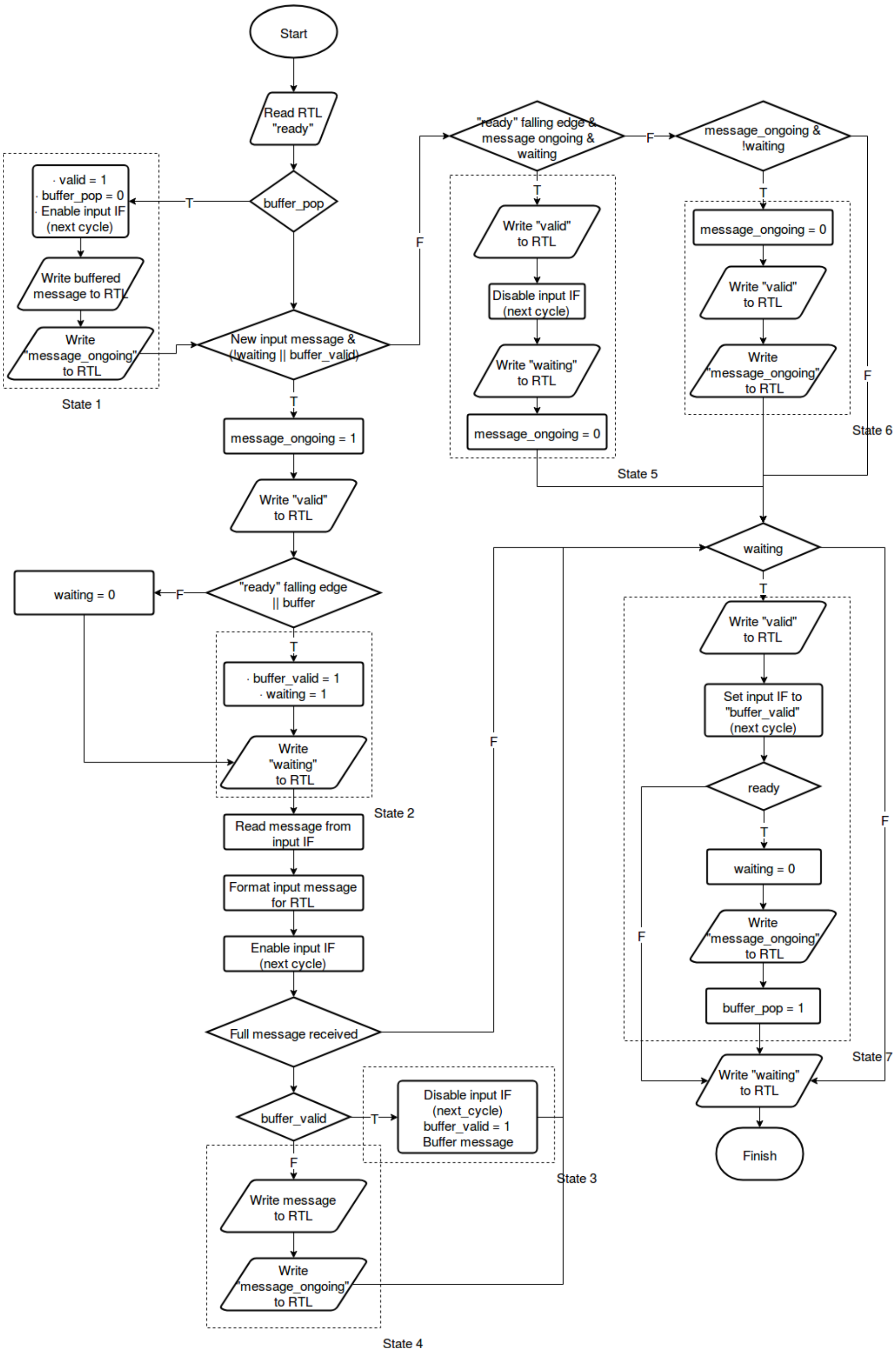

Start
Read RTL "ready"
buffer_pop
T
· valid = 1
· buffer_pop = 0
· Enable input IF (next cycle)
Write buffered message to RTL
Write "message_ongoing" to RTL
State 1
New input message & (!waiting || buffer_valid)
F
"ready" falling edge & message ongoing & waiting
F
message_ongoing & !waiting
T
Write "valid" to RTL
Disable input IF (next cycle)
Write "waiting" to RTL
message_ongoing = 0
State 5
T
message_ongoing = 0
Write "valid" to RTL
Write "message_ongoing" to RTL
F
State 6
T
message_ongoing = 1
Write "valid" to RTL
"ready" falling edge || buffer
F
waiting = 0
T
· buffer_valid = 1
· waiting = 1
Write "waiting" to RTL
State 2
Read message from input IF
Format input message for RTL
Enable input IF (next cycle)
Full message received
F
buffer_valid
T
Disable input IF (next_cycle) buffer_valid = 1 Buffer message
State 3
F
Write message to RTL
Write "message_ongoing" to RTL
State 4
waiting
T
Write "valid" to RTL
Set input IF to "buffer_valid" (next cycle)
ready
T
waiting = 0
Write "message_ongoing" to RTL
buffer_pop = 1
F
F
State 7
Write "waiting" to RTL
Finish

Figure 3.9: Flow chart of input message logic of co-simulation class

- A transfer occurs when both the *valid* and *ready* signal are set.
- If during a data transmission of several cycles (*valid* set), the slave unsets *ready*, the master will have to wait until the slave can process new data to continue.

With this information, it is now easier to explain the input logic operation. First, several flags are used for indicating or acknowledging states to the RTL wrapper's logic, some of which are actually signals in the input interface of the TU. These flags are:

- *ready*, read from the RTL wrapper, received from the TU input interface via an imported function.
- *valid*, sent to the RTL wrapper via an imported SystemVerilog task.
- *waiting*, which indicates that the TU's interface unset *ready* and the logic is on hold until it is set again.
- *message_ ongoing*, which indicates that a data transmission of several messages is taking place in the input interface.

Other flags are internal to the co-simulation class' input logic, like *buffer_pop* or *buffer_valid*.

The operation of the input message logic is now summarized by describing some of the relevant states in an order which hopes to be more comprehensible for the reader:

- On **State 4**, normal operation is carried out once a full message has been received from the model during a transmission to the TU's RTL input interface. The received message is written to the RTL wrapper.
- On **State 6**, no new messages were received from the model and the RTL is still available to process messages, so the end of a message stream has been reached, and the co-simulation class will set the *valid* signal to '1' only one more cycle (to account for timing differences).
- On **State 2**, the TU has indicated that it is not ready to receive further messages yet, so the co-simulation class will have to wait. Because of timing adaptation (a new message is received from the model, but *ready = '0'* in the same cycle), the received message has to be buffered, to be provided to the RTL interface once it is ready again.
- On **State 3**, the buffer enable is detected because of a "stop" in the RTL interface, so the interface connecting the input logic of the co-simulation class to the rest of the model communication network is disabled, indicating that no further messages can be received at the time. This change will become effective on the next clock cycle.
- On **State 7**, a *waiting* state is detected, and all the signals and flags are handled accordingly. If at this point a *ready = '1'* is detected from RTL, the *buffer_pop* flag will be set, triggering the buffer mechanism in the next clock cycle.

- On **State 1**, the mechanism triggered in State 7 during the previous cycle pops out the stored message from the buffer, writes it to the RTL wrapper and re-enables the interface connecting the input logic of the co-simulation class to the rest of the model communication network in the next clock cycle.
- On **State 5**, a special situation where the RTL input interface indicates "not *ready*" and also the last received message from the model at that same cycle is the last of a transmission, is processed.

Moving on to the output logic, which corresponding flow chart is depicted in Figure 3.10, the overall functioning is quite straightforward. For this interface, the model is always presented as "ready", so the corresponding *ready* signal is always written to '1' to the RTL wrapper. The RTL *valid* signal is then checked to detect new output messages. If no new message is available, this logic has finished. Otherwise, the output data is read from the RTL wrapper, parsed and converted into model objects and finally a GPU model's communication packet is created with those objects to be sent to the internal communication network as a TU output message.

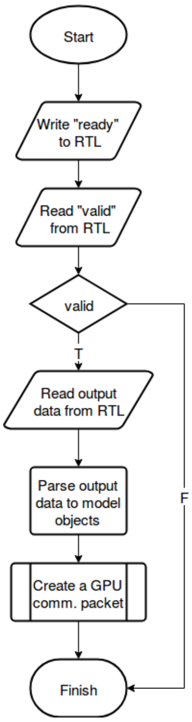

Start
Write "ready" to RTL
Read "valid" from RTL
valid
T
F
Read output data from RTL
Parse output data to model objects
Create a GPU comm. packet
Finish

Figure 3.10: Flow chart of output message logic of co-simulation class

Although the output logic may seem much simpler compared to the input one just by looking at its flow chart, the amount of effort put into its development and debugging was practically equal to its input counterpart. Most of the complexity is concentrated in the creation of the communication packet to be sent as a TU output message. A lot of work was put into this phase which is, probably, the most critical, since messages incorrectly constructed will trigger problems in other parts of the GPU model later during execution. The details on this implementation cannot be provided due to the proprietary nature of the information, but it might seem obvious that the task of implementing this process was far from trivial.

The AXI interface class

As explained earlier, the TU has two AXI interfaces to read data that might not be available in internal caches from other memories in the GPU. These interfaces were implemented in the co-simulation class by designing a single "AXI interface" class, exploiting the fact that the actual signals for both interfaces are exactly equal, being only the connection endpoint of each different (and the data they transport). This class, when instantiated, can be configured to function as the "interface 1" or "interface 2". Therefore, depending on which version of the interface is instantiated, the instance will use either the imported SystemVerilog tasks and functions corresponding to "interface 1" or "interface 2". The chosen version of the AXI interface instance will also determine to which model's internal AXI arbiter the interface is connected, and therefore other AXI parameters such as the ID range. To be able to communicate with the model's arbiters, the AXI interface class also contains an "AXI master", which simply implements a master of this protocol in the model. Finally, this class follows the previously mentioned model convention of having two main methods for the clock cycle operations and a *reset*.

Two different sets of operations are performed in this class: *read requests* and *read responses* inside the *compute* method. The former are always the first to happen, and will trigger at some point a read response as an answer. A flow chart of the *compute* method is presented in Figure 3.11. The very first thing to check by the AXI interface class is if, for the implemented interface, there is a read response available. If not, the execution jumps directly to the read request logic (which will be explained later on in this text). If a read response is found, it will be fetched from the corresponding AXI arbiter via the AXI master instance, and formatted from a model object to several values that can be written to RTL signals. Then, the memory data pointed by the response is read, and all the signals (response data and AXI control signals) are finally written to the RTL wrapper using the appropriate imported SystemVerilog tasks.

The logic for read requests is more complex, as depicted in Figure 3.12. Again, some groups of actions in the flow chart named *states* have been made to explain the operation in an easier manner. A nominal execution (without co-simulation induced conditions) of this logic would :

1. Read AXI read request signals of the implemented AXI interface from the RTL wrapper.

2. If the AXI's request *valid* is unset, no request is currently happening, and therefore no more operations are performed. Otherwise, the request data that was read is encoded into a model's object.

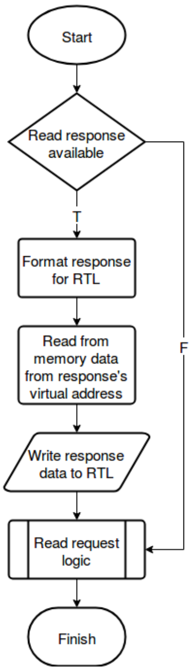

Start
Read response available
T
F
Format response for RTL
Read from memory data from response's virtual address
Write response data to RTL
Read request logic
Finish

Figure 3.11: Flow chart of *compute* method of AXI interface class

3. The master would accept data "always", so the *fifo_ en* flag would be unset as well.

4. The request will then be sent to the corresponding arbiter using the internal AXI master instance (**State 4**).

The previous procedure would occur only in an ideal execution. However, to implement co-simulation for these interfaces a problem arised. Sometimes the instance to the AXI master would flag that it cannot accept new read requests (probably by some side-effect dependent on the model's implementation). This resolves into a timing problem, as the RTL expects to have sent a new requests in the very same cycle as the master indicates it can't accept that new request. To overcome this inconvenience, another type of buffer-mechanism was implemented, in this case represented as a FIFO queue. How this mechanism works can be explained using the flow chart marked states:

- **State 2** marks the beginning of the described issue: A new read request is generated by RTL but the master cannot accept new requests at the moment. The FIFO mechanism is then enabled, and the current new read request is pushed to the queue, finishing the operations of the read request logic in this cycle.

- In the next cycle the most common scenario is that the master can accept new data again, which will take the execution to **State 3**. At this point, the FIFO contains the previous read requests that could not be send, and there is a new request received from RTL. The old request will be fetched from the queue, sent via the AXI master to the corresponding arbiter and the new request will be pushed to the FIFO. As the reader may expect, this mechanism will therefore result in additional delays every time this problem manifests.

- At a given point in time after the previous states have happened, no more new read requests will be received from the RTL wrapper, but an old request will still be outstanding, stored in the FIFO queue. This corresponds to **State 1**, where this last request will be popped out from the FIFO and sent to the AXI arbiter via the AXI master. Finally, the FIFO will normally be disabled, since the implementation assumes that no more than one outstanding request will ever exist in the queue (if a second one ever happens to be stored, an error condition was implemented).

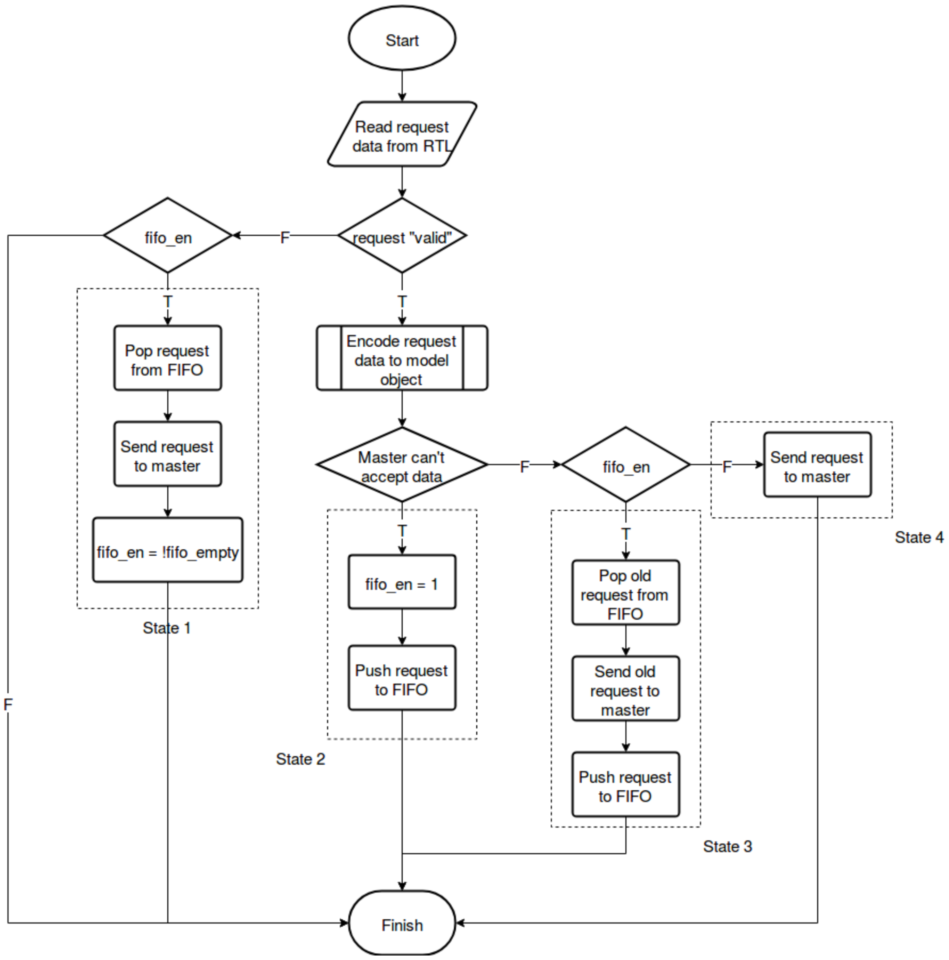


Figure 3.12: Flow chart of read request logic of AXI interface class

Finally, it is only worth to mention that the *advance* method for this class is trivial, and simply calls the *advance* method of the AXI master instance.

3.2 Results and analysis of the co-simulation environment

In this section different metrics will be presented with numbers to characterize the proposed co-simulation environment and to compare it with other previously available simulation options. First, the graphics job used in simulation for obtaining these metrics will be briefly discussed to better understand the results. Then the timing accuracy of the co-simulation environment will be compared to that of the model, with the full-system RTL's as a reference. After that, the data correctness of this setup is discussed. Finally, the total simulation time of the proposed environment is compared with two different RTL setups: the *texture replay* testbench and the full-system, being the model runtime the absolute reference.

3.2.1 The reference graphics job

In order to better understand the results that will be presented later in this chapter, it is convenient to show and discuss briefly the input stimuli that were used as a reference for the testing of this co-simulation environment and which operations these stimuli trigger in the different actors present in the simulation. However, to be able to explain this easily, it is also necessary to very briefly introduce OpenGL ES, which will help the reader understand what a GPU needs to do to process a graphical job and, therefore, give sense to the explanation about the reference benchmark used in this work.

OpenGL for Embedded Systems (GLES) is a subset of the OpenGL Application Programming Interface (API) for computer graphics rendering whose aim is to render 2D and 3D computer graphics typically used in applications such as video games, for example, and commonly accelerated using GPUs. This API is standardized and multi-platform, being one of the most commonly used. Avoiding most details, Figure 3.13 shows a block diagram of the pipeline of GLES. What is most relevant to understand is that, in general terms, the GPU needs to realize the following operations to implement graphics rendering using GLES [18]:

1. Obtain input triangles' vertices (vertexes) from application.
2. Execute the *vertex shader*, which is the program responsible for performing calculations on vertexes.
3. Execute *rasterization*, which is the process by which every primitive with 3D coordinates in the scene to render (polygons, lines, points, etc.) is converted to a two-dimensional image.
4. Execute the *fragment shader* which is the program responsible for applying operations to fragments (results of rasterizing primitives). Texturing operations are performed during this phase.
5. Perform other operations per-fragment and build the framebuffer (pixels arranged as a two-dimensional array).

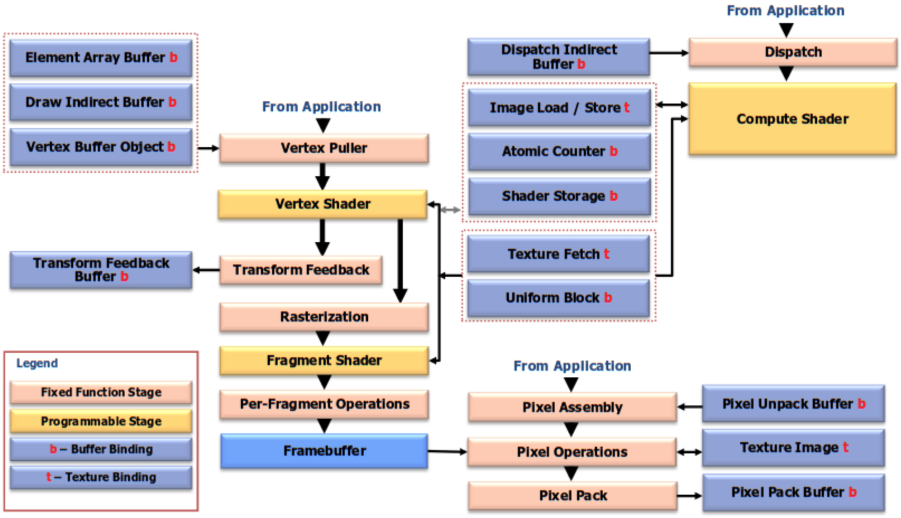


Figure 3.13: Block diagram of GLES 3.1 pipeline [19]

The graphics job used for obtaining the metrics presented in this chapter is a very simple set of operations intended to be a basic test of the TU from a GPU top-level perspective. These operations are summarized as follows:

1. Initialize the GPU and load system's memory with the input data.

2. Execute a very basic vertex shader, that performs calculations for a rectangle.

3. Execute a simple fragment shader that will simply apply a texture to the previous rectangle.

4. Finish execution by composing the image resulting of the previous operations.

Because of the nature of this test, the computational time invested in the texturing operations is much higher with respect to the other phases, making this test very suitable for characterizing the co-simulation environment proposed in this work. However, one additional fact must be known. Due to the time constraints, and the implementation complexity and debugging required for this simulation environment to work (specially on the message interface part of the co-simulation class in the software-model side), in the end the co-simulation environment could only run the presented test for one *tile*, that is, for a final image built out of 16x16 pixels, out of the bigger image that the full test would build. This is not ideal, as the "setup" time that the simulation takes (all the workload until the vertex shader, included) is then comparable or even longer to the time spent on texturing operations while, in the full test, the length of the fragment shader is enough to mask out the time overhead of previous phases, and provide more meaningful metrics. This effect and its consequences in the different metrics will be further discussed in the following sections.

3.2.2 Timing accuracy

If the co-simulation setup is to be used for performance or computational time estimations, an interesting metric to characterize this environment is its effect in the total cycle count of a simulation. In this case, the absolute reference would be the number of clock cycles spent in a RTL simulation, since it corresponds to the exact time that the chosen graphics job will take to be executed in real hardware. Before presenting the results, the reader must note that the total cycles difference in the co-simulation environment is caused by several factors, all related to how the environment was implemented, being some of which:

- Several input data to the RTL implementation of the TU is registered in the co-simulation wrapper before it is fed to the TU, with the mechanism explained in Figure 3.5. An example of this is the input messages received from other parts of the GPU. This results in a one-clock cycle delay for those signals every time their value is changed by the co-simulation class.
- In a similar fashion, the data of read responses in AXI transactions will also suffer additional delays, when compared to the nominal execution of the software model. In the model, every transaction happens instantly, bypassing the fact that the *count* configuration of a specific AXI interface might result in several data transmissions for the same AXI ID. An example of this is a situation in which *count=4* and the data width for that AXI interface is 64 bits. In a RTL implementation (like that of the TU in the co-simulation environment) this will result in 4 data transmissions of 64-bits width, each taking one clock cycle, for the AXI ID corresponding to the current response. In the model, however, such response, with its data, is received in one single cycle. Therefore a *count-1* cycles delay will be introduced in the AXI interfaces in the co-simulation environment with respect to the software model. This effect, however, is beneficial for the co-simulation environment with respect to a top-level simulation in the RTL description of the GPU, as the behavior is the same for this case.

Time metric	Model	Co-simulation	RTL
Active cycles	1733	1981	1918
Relative difference	-10.68%	3.18%	-

Table 3.1: Cycle count relative difference of software model and co-simulation with RTL-only simulation

Bearing the previous consideration, the relative difference in cycle count for this specific job compared side-by-side to the pure software model is shown in Table 3.1, where the *active cycles* roughly correspond to the clock cycles that the simulation took processing actual workload, which is a portion of the total simulation cycle count. The *relative difference* is a comparison between the model/co-simulation active cycles *versus* RTL's, following Expression 3.1.

$$Diff(\%) = (\frac{CC_{RTL}}{CC} - 1) \cdot 100 \tag{3.1}$$

Although the model seems to exhibit a further deviation when compared to the co-simulation environment, it is also true that the total cycle count of the simulation is not very high, and the software model is designed to provide more accurate timing metrics with longer simulations. All in all, the co-simulation result seems promising, and quite close to that of the RTL design simulation, suggesting that this environment would be a good candidate for computational time estimation and performance correlation solutions in a top-level context.

3.2.3 Data correctness

The most important metric to evaluate whether the achieved solution is valuable or not is the data correctness, or whether the output data generated in simulation by the co-simulation setup is correct. This is essential for the validity of the developed environment, as no functional verification would be possible if this requisite is not fulfilled. To evaluate data correctness, the output *memory dump* generated by the nominal software model (see Section 2.1.1 and Figure 3.1) is compared to the *memory dump* generated by the modified model at the end of the co-simulation, as the Mali model is, of course, functionally correct.

The *memory dump* generated during the job run in the co-simulation environment was equal to that generated by the model, therefore proving the data correctness of this solution.

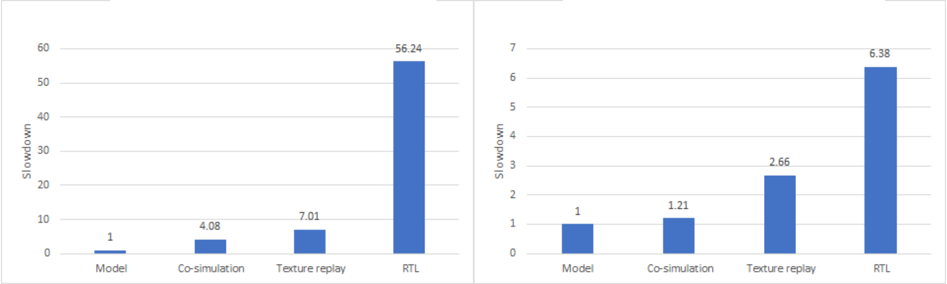
3.2.4 Simulation time

An additional metric to take into consideration to characterize the co-simulation environment is simulation time, or the real time in seconds that this setup takes to complete a job run. The idea behind evaluating this metric is to assess whether the proposed co-simulation environment gives any advantage in simulation speed over, mainly, the same simulation run with Mali's full RTL representation. If this is proved, then the co-simulation would prove to be an interesting approach to, for example, iteratively check the effect of unit-level design changes in the performance of the whole GPU system in an agiler manner.

Obtaining this metric wasn't straightforward. The main approach would simply be to measure the real-time execution of each simulation setup independently and compare them. However, Arm's infrastructure forces the use of the Cadence toolchain (and for this case, the Xcelium simulator) in a server cluster shared with many other users. The effect of this is that the simulation times vary in every execution, depending on the number of active users and the current load that the system is handling at that moment. To overcome this inconvenience in the best possible way, the simulation for each setup (model, co-simulation, *texture replay* and RTL) was run 30 times each. In this way, statistics like the maximum, minimum, and average simulation time for each setup could be obtained, giving a more faithful image of the result. However, the chosen statistic to make the actual comparison was the minimum simulation time of each case, as it provides the information with the least influence from the variability of the underlying infrastructure.

Additionally, the fact that the graphics job that could finally be run in the co-simulation environment is short in terms of texturing operations (see Section 3.2.1) means that the simulation time figure as is may not be as relevant as could be to characterize the proposed environment. Furthermore, for all the simulation runs, except for the software model, the Xcelium simulator is used, and this program introduces an extra time overhead since the time it is invoked until the time when the actual simulation starts. This overhead is not significant in longer simulations, but it is in this particular case. For all this reasons, and because the most meaningful result would be obtained from a texture-heavy job run (or, in other words, a job in which all phases are very short with respect to the fragment shader), the simulation time results will be presented for the 4 different setups for two cases: the full reference graphics job and for only the fragment-part of it (which would be the difference between the time taken to execute the full job and the time taken to execute all the phases up until before the fragment shader).

Finally, before discussing the results, and as it has been advanced before in the text, the simulation time of an additional setup was also measured: a *texture replay* simulation. This would correspond to a different test for the TU which had been used in the past for the same purposes as the proposed co-simulation environment. In this test, a RTL testbench in which the TU is the DUT drives TU's inputs with stimuli read from a file, which are previously created by running the nominal model and dumping (and partially parsing) all the transactions in the input interfaces of the TU interface class, effectively emulating, at least at a functional level, the system behavior and interaction with the TU.



(a) Relative slowdown in the full graphics job

(b) Relative slowdown in the fragment part of the job

Figure 3.14: Relative slowdown of the different simulation options with respect to the software model

The simulation time comparison for the two presented cases is shown in Figure 3.14. The number used as a relative comparison of the co-simulation, texture replay and RTL simulations *versus* a model run time is the *slowdown* (S_{down}) in simulation time, as expressed in Expression 3.2, which is simply the inverse of the *speedup* (S).

$$S_{down} = \frac{1}{S} = \frac{t_{sim}}{t_{sim_{model}}} \tag{3.2}$$

As can be observed, in both cases the co-simulation environment is the closest simulation option to the model in terms of simulation speed. Very interesting to highlight is the fact that the co-simulation environment is also much faster than the common simulation of the full-system RTL, confirming the initial expectation that the the proposed environment would be a faster but still accurate solution for evaluating in a top-level context changes in a hardware unit. The co-simulation also seems to be faster than the *texture replay* test. This seems reasonable if one thinks how the stimuli to the TU's RTL representation are generated in each case. In co-simulation, it is the model that interactively creates and sends stimuli to the TU's RTL, taking advantage of its much faster execution as object code. On the other hand, the *textre replay* simulation will use some testbench logic to open, read and parse the stimuli data from the replay files, spending a lot of computational time in these tasks before the stimuli for the TU are ready.

Finally, regarding the absolute *slowdown* numbers, discussions with experienced engineers in this GPU system yield that, if a longer simulation in terms of texture operations could have been run in the co-simulation environment, the *slowdown* for all cases will probably be in a middle point between the numbers in Subfigure 3.14a and Subfigure 3.14b. Therefore, an approximate *slowdown* of around 2.5 could be expected for the co-simulation environment which, compared to an approximate *slowdown* of around 20-30 for the RTL simulation would still give a significant advantage to the co-simulation environment in terms of simulation time.

Chapter 4

Conclusions and future work

In this chapter a summarized overview of the thesis project developed will be provided, recalling the objectives that this thesis aimed for and the decisions taken for implementing the proposed co-simulation setup. Some comments on the results obtained will also be included, along with a final conclusion. Finally a brief section on future work that can be derived from the work in this thesis is included.

4.1 Conclusions

This Master's thesis project, titled *Hardware-software model co-simulation for GPU IP development* had one primary objective: To investigate the feasibility of a mixed simulation environment where some parts of a software model (of a hardware system) can be swapped with their corresponding RTL description, and which advantages may this possibility offer. Another expected outcome of this work was that the resulting environment would open new use-cases, or improve tasks like design exploration or performance correlation in a system's top-level context. All of these objectives were pursued for a specific case: Arm's next-generation Mali GPU, of which its C++ software model, along with the RTL description in SystemVerilog of one of its internal blocks, the texture unit, were the main actors.

The implementation carried out to realize this environment makes use of SystemVerilog's DPI-C to enable the communication between the object code of the software model of the GPU with the RTL description of the texture unit. Several new sub-blocks had to be developed to adapt the interfaces of both the texture unit class in the model and the RTL representation of the texture unit in a way in which none of this two elements had to be modified. The resulting co-simulation environment complies with the expected objective of effectively replacing a component in the software model with its RTL representation, permitting the simulation of a subset of a graphics job that had been previously utilized for both model and full-system RTL simulations.

In general, interesting results were obtained with the proposed co-simulation environment, bearing in mind the additional conditions that the length of the graphics job used for characterizing the environment introduces. The timing accuracy, in terms of total clock cycles, is quite close to that of the full-system RTL, with a difference of around 3%. This might put this proposed environment as an interesting candidate for tasks like performance correlation or computational time estimation. Regarding data correctness, or whether the output data generated by the co-simulation environment corresponded to the expected, the outcome was positive, and this environment can therefore be used for functional verification in this case. Finally, the simulation time of this setup was compared to that of two RTL testbenches: the *texture replay* and the full-system RTL, being the model runtime the absolute reference for all. The general outcome of this metric evaluation is that the co-simulation environment was the fastest solution after the model, being able at the same time to provide a higher level of detail with respect to the software model in the TU part of the design thanks to the use of its RTL representation.

For the author of this work, this Master's thesis has been extremely valuable in terms of knowledge and experience, as the nature of the work demanded a certain level of understanding of both software and hardware languages. This work also demanded a rather deep understanding of the tools utilized to realize the proposed implementation. More specifically, a lot of time was spent studying the SystemVerilog language specification (specially for details on Subroutines and the Direct Programming Interface), and Cadence's Xcelium documentation, as the computer environment for running the co-simulation setup also had to be developed prior to actually implementing the mixed simulation. Finally, the opportunity of developing this thesis work based on a real, complex design in a company like Arm was also an invaluable experience, from which the author has learn a lot, both personally and professionally.

4.2 Future work

The amount of future work that this project opens up is considerable. Therefore, a selected group of suggestions are presented hereunder:

- The most immediate upgrade for the proposed co-simulation environment would be to continue the implementation work on the message interface part of the co-simulation class, to **enable longer and more complex simulations** to be run in mixed simulation.
- An interesting study that time did not permit in this work is evaluating the **feasibility** of using the proposed co-simulation environment for **performance correlation** simulations, since it could maybe be used to substitute the software model in certain situations when a higher level of accuracy is necessary in the TU part of the system.
- According to the results of this thesis work, it may be interesting to **extend the idea of co-simulation to other units** in the GPU system. However, this would actually require a combined effort of many engineers, because the software implementation of these units would need a minimum level of alignment with the RTL implementation to allow co-simulation in a fashion such as the one suggested in this work. This might be something worth keeping in mind if one of these units' software model implementation is ever developed from scratch at some point in the future.

- Starting again from the premise that this work might be interesting to continue developing, an **automated environment** could be created in which, if a specific unit in the software model complies with some defined design rules, a co-simulation setup can be automatically generated in SystemVerilog. To make this feasible, a set of rules or guidelines on how to design a software model unit to comply with the requirements would have to be developed, and put into practice when a new GPU unit is written in the Mali model.
- In a related, opposite approach, another co-simulation case could also be studied: a **full RTL setup in which only one or several units are replaced with their software model C++ implementation**. This option would also open new use-cases, such as design or features exploration of the system when a new block is developed in the software model before its RTL implementation is ready.

References

[1] Chris McClanahan, *History and Evolution of GPU Architecture.* Georgia Tech College of Computing, 2010.

[2] Winnie Thomas et al., *Performance Comparison of CPU and GPU on a discrete heterogeneous architecture.* Veermata Jijabai Technology Institute Mumbai, India, 2014.

[3] Nvidia corporation, *What's the Difference Between a CPU and a GPU?.* Santa Clara, California, United States, 2018.

[4] Arm, *Mali Graphics Processing.* 2018. `http://www.arm.com/products/graphics-and-multimedia/mali-gpu`.

[5] Mark Zwolinski, *Event-driven simulation.* University of Southampton, UK. `http://users.ecs.soton.ac.uk/mz/elec3017/vhdlsim.pdf`.

[6] Patrick R. Schaumont, *A Practical Introduction to Hardware/Software Codesign.* Boston, MA, USA. 2010.

[7] Chun-Jen Tsai, *HW/SW Co-Simulation.* National Chiao Tung University. Taiwan, 2011.

[8] Stefano Centomo et al., *Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements.* Euromicro Confierence on Digital System Design, 2016.

[9] Dominik Widhalm et al., *Augmenting Pre-Silicon Simulation by embedding a Scripting Language in a SystemC Environment.* University of Applied Sciences Technikum Wien, October 2016.

[10] Michael D. McKinney, *Integrating Perl, Tcl and C++ into Simulation-based ASIC Verification Environments.* Texas Instruments Inc., Dallas, TX, USA. 2001.

[11] R. Timothy Edwards, *Magic VLSI layout tool.* April 2017. `http://opencircuitdesign.com/magic/`

[12] Cadence Design Systems, *Tcl scripting for EDA.* 2018. `https://www.cadence.com/content/cadence-www/global/en_US/home/training/all-courses/82158.html`

[13] Steve Huntley, *Adding Tcl/Tk to a C application.* wiki.tcl.tk. 2014. http://wiki.tcl.tk/3474

[14] IEEE Standards Association, *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification and Verification Language.* New York, USA, 2017.

[15] Holger Keding, *SystemC / SystemVerilog Interaction.* Universitat Tubingen. http://www.ti.uni-tuebingen.de/uploads/media/Presentation-14-UP_1_keding.pdf

[16] Stephen Clamage, *Stability of the C++ ABI: Evolution of a Programming Lnaguage.* Oracle Technology Network. June 2016. http://www.oracle.com/technetwork/articles/servers-storage-dev/stablecplusplusabi-333927.html#here_there

[17] Junyoung Park, *Hanshake protocol.* University of Texas. Fall 2011. http://www.cerc.utexas.edu/~deronliu/vlsi1/lab3/2014_fall_VLSI_I/LAB3_Website/handshake/handshake.pdf

[18] Khronos Group, *OpenGL ES 2.0.25 specification.* November 2, 2010. https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf

[19] Khronos Group, *OpenGL ES 3.1 specification.* November 3, 2016. https://www.khronos.org/registry/OpenGL/specs/es/3.1/es_spec_3.1.pdf