Final Master Thesis

Master in Innovation and Research in Informatics

# Verification of a microprocessor's memory pipeline with UVM

June 2022

**Author:** Josep Sans i Prats
**Director:** Pedro Marcuello
**Ponent:** Roger Espassa

Facultat d'Informàtica de Barcelona (FIB)

# Acknowledgments

I would like to thank my thesis supervisors Pedro Marcuello and Roger Espassa for their help and guidance during the development of the project, as well as in the redaction of this document.

To all the team at Semidynamics who helps me become a better engineer everyday, and special thanks to Sebastiano, the main developer of the memory pipeline, who has had the patience to explain in detail how the design works numerous of times. Also to Esther Valbuena and Miquel Ortega to help me debug the developed testbench.

To my family and Clàudia, without their support and love, this thesis would have not been possible.

# Abstract

The development of a microprocessor requires many efforts in order to be able to successfully tape out a bug free design. Due to the high costs and waiting times of fabrication, the option of doing multiple iterations of a design is not usually available. That is why companies have big verification teams responsible of the good functional behaviour of the design logic. They provide reliable verification environments to test all the different scenarios the design under test will be exposed, and help the design team debug the issues encountered. Techniques like *Universal Verification Methodology*, coverage, assertions, test generation are de facto standard in verification.

This thesis presents the contributions made in the environment developed for the verification of the memory pipeline of a RISC-V core. A UVM testbench, along with a golden model, has been developed which is able to functionally verify the behaviour of the memory pipeline. To generate tests to stress the different functionalities of the memory pipeline, a test generation flow based on a genetic algorithm has been set up. With it, several issues on the memory pipeline logic have been found, and helped improving the RTL logic of the design.

# Contents

# List of Figures

# Chapter 1

# Introduction

Developing hardware is a very complex process that involves numerous of highly trained teams with different specialties. On top of that, the fabrication process in order to get the physical chip is very expensive. That is why, once the design is sent to the fab, there has to be a highly confidence that it works accordingly to its specification. The team responsible of this verification is the Design Verification team.

In the last years, and the ones to come, Moore's law will help less and less in the improvements of microprocessors and every Application-Specific Integrated Circuit (ASIC). Since the demands of faster, smaller and efficient designs by the market won't stop, hardware architects need to add more complexity to their designs in order to make them better on each generation. This increment of complexity, translates into more risk of errors in the design, which makes the verification process even more important.

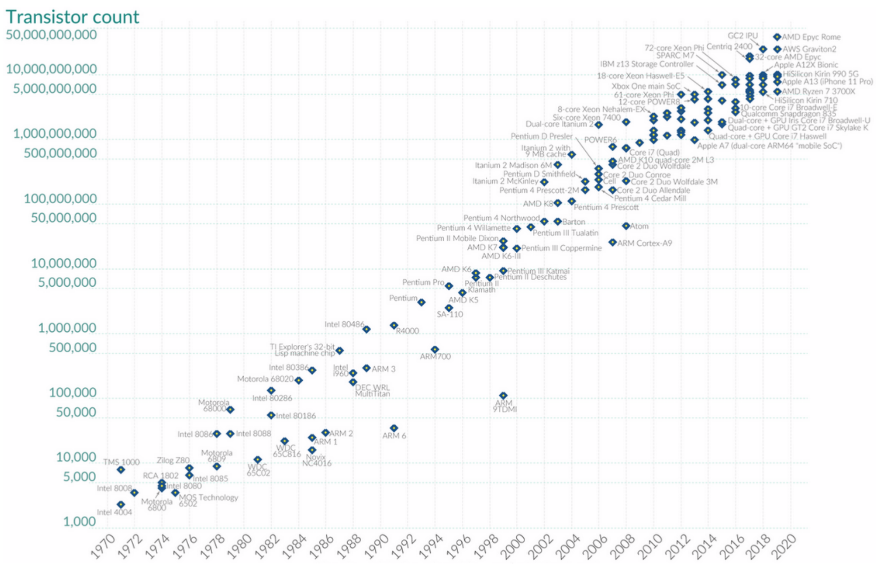

Figure 1.1: Number of transistor per millimetre square over the years.

## 1.1 Motivation

Although the verification process of any digital design is crucial for a successful tape-out, there are very few, to almost none university level subjects about the topic. Computer Science degrees and masters are usually focused on architecture design but they leave no place for one of the most important and extensive tasks in ASIC development, design verification.

Moreover, little is the number of companies that disclose their verification process and techniques. Comprehensive because the enormous effort and money they invest in the task, but makes it even harder for inexperienced teams to familiarize with all of the new concepts of the verification process. Thanks to the irruption of RISC-V to the microprocessor world, different open-source organizations[4][5] have started their own development of new RISC-V cores and documented some aspects of the verification process[6]. These resources are of great help for newcomer verification engineers as they serve as a non trivial verification project examples.

One of the main purposes of this thesis, is to document and explain all the verification process along with the important decisions that were taken in order to successfully verify the memory pipeline of a microprocessor. And add a little bit of public knowledge about design verification.

## 1.2 Contributions

This thesis was done in the context of the Verification Team at Semidynamics, and it presents some of the work performed in the verification process of the memory pipeline of Avispado[1], one of the in-order company cores.

The main objective of this project was to develop a testbench for the memory pipeline that enables a coverage directed test generation strategy in order to stress the design. The key contributions to achieve this objective have been:

- **Memory pipeline testbench**: A testbench based on the UVM [2] methodology has been developed. A custom language has also been developed in order to control the actions of the testbench. This has helped to speed up the creation of tests and to be able to generate test cases with external tools.
- **Coverage directed genetic test generator**: A genetic algorithm has been used to develop the test generation flow. It makes use of the test coverage to evolve the quality of the generated tests and eventually, discover bugs in the design.

Figure 1.2 summarizes the contributions presented in this thesis. It presents the final verification environment developed for the memory pipeline, where the generated tests from the genetic algorithm are run in the UVM testbench and give feedback to the test generator so it can improve the quality of the tests. Thanks to this environment, 8 functional bugs and 4 performance bugs have been discovered in the memory pipeline logic.

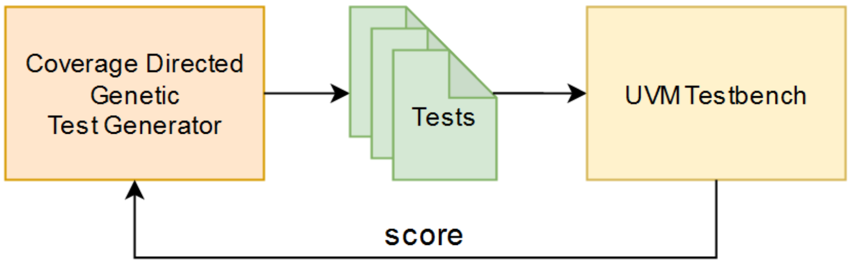

Figure 1.2: Schematic of the verification environment

## 1.3 Thesis structure

The thesis will follow the following structure:

- *Chapter 2* covers the essential aspects required to understand the project. It dives into the *Design Verification* world, describes all the key components of the UVM framework, and explain the coverage directed test generation methodology.
- *Chapter 3* provides an overview of the *Memory Pipeline*, the Design Under Test (DUT), and all of the modules that interacts with, as well as how they have been simulated in the verification environment. The implementation of the verification environment, will be also discussed in this chapter.
- *Chapter 4* does an in-depth explanation of the new language created with all the implemented instructions and features of it. It also describes how it has been integrated with the testbench.
- *Chapter 5* describes the coverage directed genetic test generation approach, the coverage results obtained from the tests generated by the algorithm and the bugs found in the design.
- *Chapter 6* summarizes the project by describing the conclusions of the thesis and the possible future work.

# Chapter 2

# Background and Related Work

This chapter introduces the reader into the necessary topics to fully understand the thesis.

Section 2.1 presents the *Design Verification* topic and its importance in the ASIC and *Digital Design* worlds. In Section 2.2 the UVM framework is presented and explained along with all of its components.

## 2.1 Design Verification

The verification of a digital design consist in the creation of different software elements which helps checking that the behavior of the Design Under Test (DUT) is the one stated on its documentation. The verification process, spans during almost all of the development steps. Since each of this steps produces some modification or transformation *e. g.* synthesizing the RTL to a netlist, it needs to be checked that the DUT keeps behaving as specified, and no error has been introduced.

At the beginning of a project, once the design architecture documentation has been written, the verification process starts. The verification team, will elaborate a document called Verification Plan, where all the different aspects of the verification will be discussed, annotated and distributed to the different verification teams. The plan will include:

- **Assertions**: Properties that are checked during simulation and are responsible of ensuring that the axioms of the design hold, and so the design works by its specification. One example of an assertion would be to check that in a queue design, data is not pushed when the queue is full. The SystemVerilog code would look like this:

  ```
  write_on_full : assert property (
    (posedge clk_i) disable iff (!rst_ni) full |-> !push_i)
    else $error("ERROR: Trying to push new data although the FIFO is full"
  );
  ```

  *The $|->$ is the implication operator in SystemVerilog.*

- **Assumptions**: Properties that are used to specify which are the valid inputs of the DUT. They are in special use by Formal Verification[1] tools. An assumption example would be to tell the tool that and input id is not greater that a certain value.

```
id_constraint : assume property ((posedge clk_i or negedge clk_i)
  sb_id_i < STORE_BUFFER_DEPTH_TOTAL);
);
```

- **Coverage**: Metrics that are collected by the simulator and are of use of the Verification Team, in order to know which areas of the design have been stimulated and which ones have not yet been reached. Toggle, Branch, Condition, Expression and FSM coverage are the types of coverage collected automatically by the simulation tools.
- **Functional Coverage**: This coverage is manually defined in the Verification Plan, and covers all the scenarios, input sequences, corner cases, etc. That are crucial for the design and need to be tested.
- **Testbench**: The different testbenchs that will be build to verify the DUT. Depending on the resources, the Verification Team will adopt a *Block Level* verification strategy, where each one of the modules that conforms the DUT will be tested on its own. Or a *Top Level* verification strategy where the verification will use the highest possible level to verify the DUT.

The Verification Plan might also include the kinds of tests to be perform post synthesis, using the netlist of the design in a *Gate Level Simulation* GLS fashion. GLS will check that the synthesis has been performed correctly, the timing requirements of the cells are met, there are no unconnected paths, X propagation, etc. Or the tests to be done once the chip is back from the fab, in order to know that all connections of the chip are alive and there have not been errors during its manufacturing process.

## 2.2 Universal Verification Methodology (UVM)

Universal Verification Methodology or UVM, is the standard verification framework and methodology in the industry, as it enables faster development and reusability of the code. UVM was created in 2009 by Accellera, a standards organization in the Electronic Design Automation (EDA) industry, and standardized in 2011 in the IEEE 1800.2. Before UVM there were different verification libraries: Open Verification Methodology (OVM), eReuse Methodology (eRM), etc. Which were developed by EDA vendors, but only supported by the company which had developed the library.

UVM has been developed using the SystemVerilog language and it makes use of its characteristics to build its software stack. Recently there has been a port of UVM to SystemC/C++, but it lacks some of its features.

[1]Verification technique that makes use of mathematical methods in order to proof the correctness of a design. It makes use of the assertions and assumptions declared in order to know the valid inputs and outputs of the DUT

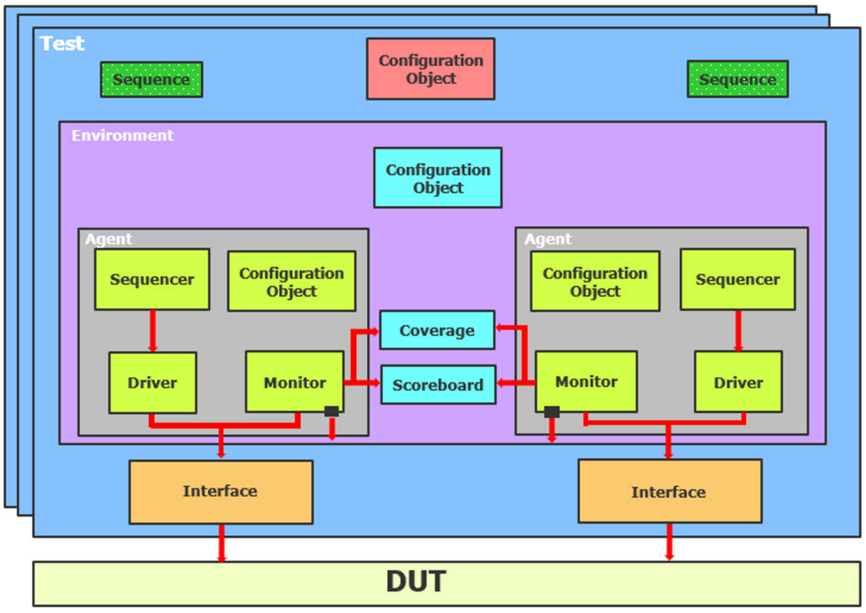

Figure 2.1: UVM testbench block diagram

The idea behind UVM is to provide a groups of classes and methods that serve as a structure for the testbench. Later the verification engineer will extend this classes to fit them into the needs of the verification project.

## 2.2.1 UVM structure

A typical UVM testbench starts with a module, usually called *tb* or *tb_top* where the DUT is instantiated, and it will serve as the top hierarchy file of the testbench. In this file, there will also be the connections between the DUT interface and the possible multiple virtual interfaces of the testbench. Once everything has been setup, a call to the method *run_test* will start the UVM test.

## 2.2.2 uvm_test

The *uvm_test* will be the first UVM class created in the testbench. It is responsible to instantiate the environment, and set it up according to the needs of the specific test via the configurations object. Once this configuration has been done and the environment is ready, it selects the specific sequences to be run for that test and starts them.

## 2.2.3 uvm_environment

The UVM environment, extended from the *uvm_env* class, acts as a harness of the different agents of the testbench and its shared components like the scoreboard, functional coverage collector, specific checkers, etc.
It will also be responsible to set up the default configurations or pass through the test configuration to the different agents and do the proper connections between those and the

different components of the environment. One typical connection would be from the agent's monitors to the scoreboard, so it can receive the observations in the interfaces and check that they are correct.

### 2.2.4 uvm_agent

It is the last wrapper class down the hierarchy. The *uvm_agent* contains all the functionality, and configuration, to drive and monitor an specific interface of the DUT. An agent can be configured as *active*, where all elements of it are enabled. Or *passive* where only the monitor components are activated.
An UVM agent is composed by a *driver*, a *monitor* and a *sequencer*.

### 2.2.5 uvm_driver

The driver is an active entity that has the knowledge on how to drive the signals of a particular interface of the design, it implements the interface protocol. For example it could be an AXI, APB, CHI, etc. or a custom protocol.
The driver will obtain the data to drive the interface, as transactional items which comes from the sequencer it is connected to. If it is in the needs of the testbench, the driver is also able to send the response, of the driven item, to the sequence using the response port of the sequencer.

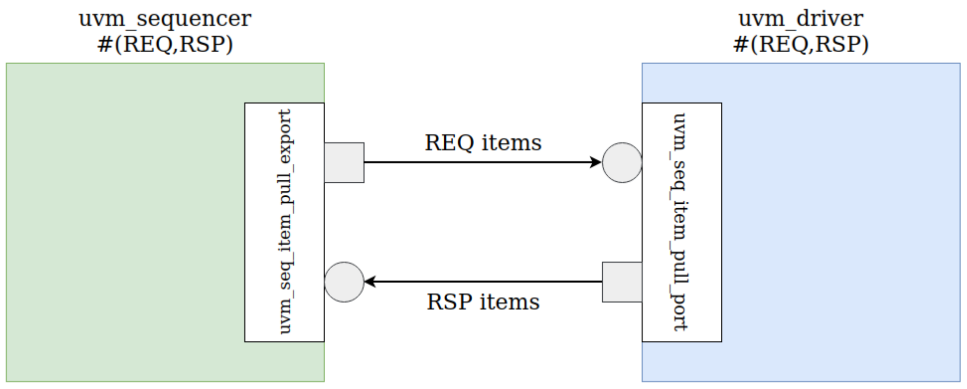

Figure 2.2: Connections between uvm_driver and uvm_sequencer

### 2.2.6 uvm_sequencer

The sequencer is a UVM component that serves the transactional items from the sequences to the driver. From one side it is connected to the sequence object, and in to other side it has the connection to the agent driver. In the case where there are more than one sequence, the sequencer is used as an arbiter between those sequences. It can be configured so each sequence has its own level of priority.

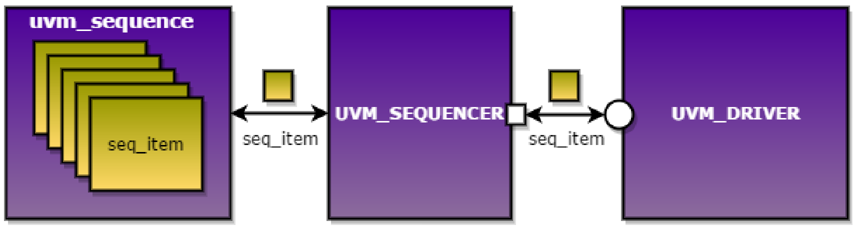

Figure 2.3: UVM sequencer connections [11]

### 2.2.7 uvm_sequence

A UVM sequence contains the information, usually in form of constraints, about the kind of data item it needs to be send to the driver in order to provoke a concrete stimulus in the DUT. In a UVM test we usually combine more than one sequence, in order to create interesting scenarios for the DUT.

### 2.2.8 uvm_monitor

The job of a UVM monitor is to observe the signal activity from the design interface. Capture it into transactional items which later can be send to the other UVM components. Usually the scoreboard or reactive sequence,are the most common cases.

### 2.2.9 uvm_scoreboard

The scoreboard is a verification component which contains checkers or a golden model instance of the DUT. It receives transactional items from the different monitors and drivers of the testbench and verifies that the design is working as expected, by comparing the results obtained from the reference models and the items received from the different UVM components.

The golden model could either be a pure functional model, which only produces the expected result of an operation, or a cycle accurate model of the DUT, where each cycle of the operation can be checked and not just the output result. In the scope of this project, a functional model of the DUT was created following the specifications of the design.

## 2.3 Coverage Directed Test Generation

Test generation is a very important subject in the verification process, pre-silicon and post-silicon, of an ASIC design. Usually a set of constraints is used to guide the test generation towards certain verification scenarios. For each of the test created, coverage is collected to measure the effects of the different tests. Then the results are evaluated and the constraints are modified in the hope that the coverage is increased in the next simulation round. This approach is known as Coverage Directed test Generation (CDG). In order to be able to keep improving the quality of the tests, the process of analyzing the coverage results and set up the new constraints needs to be automated. For simple designs, where the coverage space is not too large, a random algorithm could be enough. But for the DUT presented in this theses, and for designs with a little bit of complexity, an algorithm, usually from the area of Machine Learning (ML), is needed in order to keep evolving the quality of the tests, and finally close coverage. Examples of how these algorithms have been used are:

- **Bayesian Networks**[9] where it tries to find a relationship between the generation directives and the coverage archived by training a pre-constructed Direct Acyclic Graph (DAG). And once trained, apply inference in order to hit the remaining coverage tasks. It requires an initial engineering effort and expertise in order to construct the graph but once constructed they succeed in closing the coverage effectively and efficiently.
- **Genetic Programming**[3] where the tests are generated in accordance to an instruction library and are evolved using an evolutionary algorithm. This technique was the one decided to integrate for the test generator of this project. Chapter 5 gives an insight of the genetic algorithm used and which have been the results obtained.
- **Markov Model**[12], this approach makes use of a random test generation which is directed by an adaptive Markov Model (MM) created by the user. The goal is to adjust the weights of the links between the nodes of the MM graph accordingly to the coverage results of the DUT.

## 2.4 Related work

Regarding design verification, with the rise of RISC-V and its open-source nature, different organizations have started publishing their efforts on design verification of their projects. This documents are very valuable since they are the few available documents explaining the insights of a verification process in a project of industrial grade. The institutions that have the best documentation from a verification point of view are OpenHardware[4] and lowRISC[5].

Both organizations have developed different open-source hardware projects, RISC-V cores like Ibex[7] by lowRISC or Ariane[8] by OpenHardware. Or a root-of-trust project like the OpenTitan[10] where a massive effort in terms of documentation have been done. In all of this projects there are examples on how to structure a test and coverage plan, different approaches on how to verify the design, different simulation environments, etc. From this projects it is clear that there is not an unique approach to verification, but there are common techniques like UVM, assertions, coverage or a Continuous Integration (CI) system. All of these methodologies and concepts have been taken in mind when designing and building the testbench discussed in this thesis.

# Chapter 3

# Memory Pipeline Architecture and Verification Infrastructure

This chapter will take a closer look to the background needed to understand the verification environment that has been built. Starting by doing an overview of the design under test, the memory pipeline, in Section 3.1. Section 3.2 will describe the RTL modules that are connected to the memory pipeline and interact with it. Finally Section 3.3 will have an in-depth description of the UVM testbench.

## 3.1 Memory Pipeline

The memory pipeline that has been verified in this project is the one integrated in the *Semidynamics* company cores *Avispado* and *Atrevido*. It responsible of executing the scalar and vector loads and stores as well as Atomic Memory Operations (AMOs), while maintaining the coherence. Inside the core hierarchy, the memory pipeline receives the name of Load Store Unit (LSU).

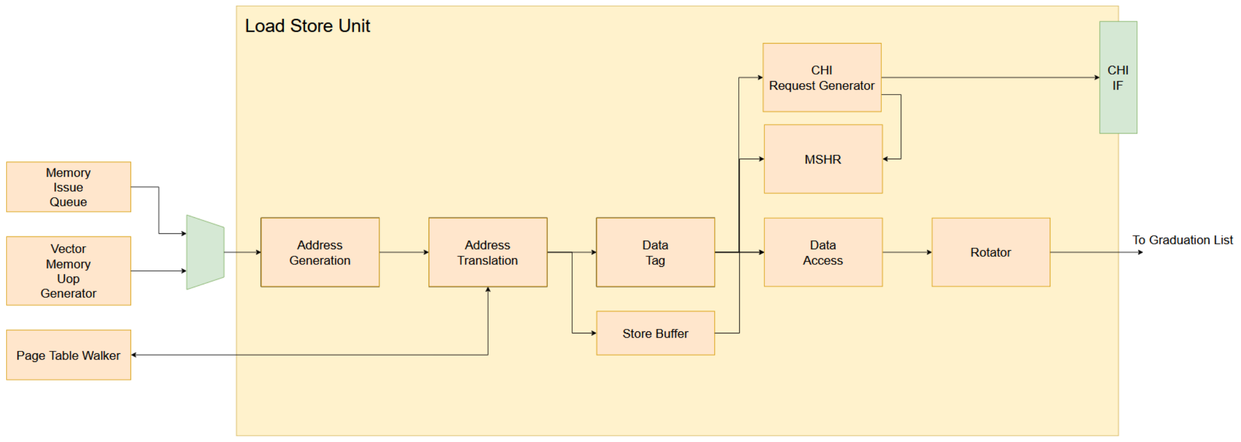

Figure 3.1: Block diagram of the memory pipeline

The load store unit is composed by a pipelined design of multiple stages. It is able to work with both virtual and physical addresses, supports unaligned memory accesses and different memory regions, with their own characteristics, for example, whether the region is cacheable or which kind of interface protocol, AXI or CHI, is needed to reach it. As mentioned above, the LSU supports coherent memory accesses over the CHI interface protocol[13]. The MESI protocol is the coherence mechanism implemented in the design, that is, a line can be in the following states:

- **Invalid**: The line is not valid, the data it contains can not be used.
- **Shared**: The line is valid and more than one core has it on the same state. The core can read from it, but if it needs to write into the line, it needs to notify the other cores of the write operation so they invalidate its line.
- **Exclusive**: The line is valid and it is the only core that has it. If it needs to do a write it doesn't need to notify the other processors.
- **Modified**: The line is valid, and has been modified by a write. In case other processors request the line or it needs to be evicted, the core will be the one responsible of serving the line and downgrading its state to either *Shared* or *Invalid*.

In order to interface with cachable memory regions, the LSU uses a coherence bus which implements the CHI specification. This interface is used to send and receive new cache lines, evict modified data and receive snoops in order to maintain data coherence. The CHI interface is divided into the different channels:

- **Request (REQ)**: This channel is used to request new lines from the L2 and coherence state updates for an already present line in the L1.
- **Fill (FILL)**: This channel is used to receive the requested data from the L2.
- **Evict (EVC)**: This channel is used to send the evicted data from the L1 to the L2.
- **Snoop request (SNPREQ)**: The snoops sent by the L2 in order to maintain memory coherence between the cores of the SoC, are received through this channel.
- **Snoop response (SNPRSP)**: This channel is used to send back the response of the received snoops. If it is required by the type of the snoop received, it will contain as well the corresponding data.

In order to increase the stores performance, the memory pipeline has a Store Buffer with it the memory pipeline is able to complete one store per cycle without affecting the design frequency.

Thanks to the MSHR structure, the memory pipeline is able to support a large number of outstanding misses. This feature is very important for the vector memory operations, since a single vector ISA instruction can produce multiple line misses.

## 3.2 Modules in the Memory pipeline border

The load store unit is a crucial part of most microprocessors since it has most of the logic to communicate with the components outside the core. For this reason, there are multiple modules that are connected to the memory pipeline. In case of *Avispado* core, the modules that are connected to the LSU are:

- **Memory Issue Queue (MIQ)**: It contains the instructions that will be executed next in the LSU, and the logic to properly serve instructions which the LSU has rearmed, and need to be issued again. It is also in charge of serving the store data to the SVB when a store has been issued and the data is ready.
- **Vector Memory Micro Op Generator (VMUG)**: For the case of vector loads and stores, the single RISC-V instruction, is to complex for the LSU to execute. Keep in mind that RISC-V defines three major types of vector memory operations:
  - *Unit stride*: where data is fetched contiguously from an initial address.
  - *Stride*: where data is fetched from an initial address and a fixed stride is added to fetch the following elements.
  - *Indexed*: where addresses are formed from an initial address plus an index that comes from the elements of the vector register.

  In order to reduce all the logic of address creation, and to be able to efficiently support cache misses, the VMUG module will split the vector memory operation and create, for each element it has to fetch, a memory operation similar to a scalar load or store. This new operation will be inserted into the MIQ to be issued to the LSU.
- **Data Page Table Walker (DPTW)**: When a memory instruction provokes a miss on the TLB, the DPTW will be in charge of correctly fetching the address translation needed from the page table by issuing loads to the LSU. Getting the cache lines that has the translation information, and inserting it to the data TLB.
- **Instruction Page Table Walker (IPTW)**: It does the same job as the DPTW but for the instructions TLB.
- **Graduation List (GL)**: Once the LSU has finished the execution of a memory operation, it will communicate it to the Graduation List in order to mark that instruction as completed.

When the LSU needs to access the next level of memory, in this case the L2 Cache, it does via one of the two supported protocols, CHI or AXI, depending the memory region requested.

## 3.3 UVM Testbench

The testbench that has been build in order to be able to verify the memory pipeline presented in the previous section, has been developed using the UVM methodology. Specifically it has used the UVM-SystemC port in order to be able to simulate the whole project using Verilator. Verilator is an opensource SystemVerilog to C++/SystemC translator, but it does not support all the SystemVerilog language features. That is why it is needed the use of the UVM-SystemC port which allows to build the UVM testbench in C++/SystemC. Verilator is a perfect tool for running a high number of simulations in parallel since it doesn't require of any license, making the limitation on the number of cores available and not the number of licenses the company can buy for a proprietary simulation tool.

To the final architecture of the DUT, has also been included the DPTW and VMUG modules. This has helped to reduce a great number of lines of the testbench code by not replicating a working and verified logic again, without compromising any testbench feature, nor the verification of the memory pipeline.

The LSU interface is very wide due to the number of different modules it interacts with. In order to organize the logic of the different subinterfaces, a UVM agent, per interface, has been created which is in charge of correctly driving and monitoring its corresponding subinterface. In Figure 3.2 it can be observed the different identified sub-interfaces of the LSU, and how they have been grouped and connected into the different agents of the testbench.

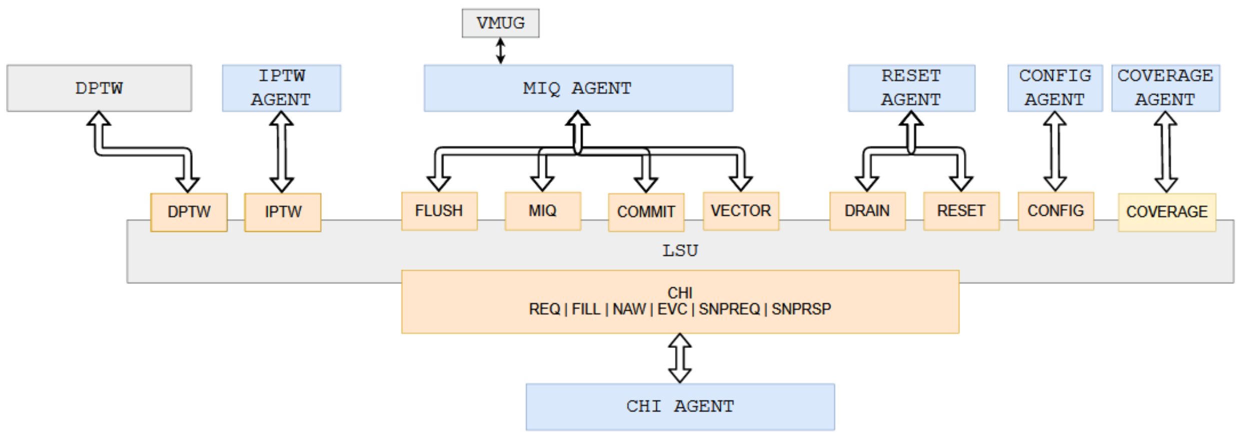

Figure 3.2: Testbench agents and the subinterfaces they are connected to

The MIQ agent is in charge of the correct issuing and completion of the memory operations issued to the LSU. All transactions related to memory operations, which have been produced by the test sequences, are collected by the agent driver. This component, will drive the corresponding LSU ports in order to correctly insert the instruction to the memory pipeline. On top of issuing instructions the *miq driver* needs to correctly control its internal structures to be able to re-issue instructions that were already issued, in the same order. This re-issue could happen either because the LSU could not process the instruction at the time it was issued, or because a previous instruction provoked an exception and the pipeline was flushed. The *miq driver* is also in control of the store commit signal and the SVB interface, it keeps track of the register dependencies and send the store data when it is available. As mentioned above, in order to reduce the logic in the *miq driver*, the VMUG module has been placed into the DUT. This way, instead of generating the micro operations in the driver, it interacts with the VMUG in order to be able to insert vector memory operations in the LSU.

Once an operation completes, the monitor of the MIQ agent, which is observing the completion interface, will capture the values of such interface and create a transaction packet which will be send to the scoreboard in order to verify that the instruction has been executed correctly. One important check that has the *miq monitor* is to detect livelock situations, where the same instruction has been issued and re-issued a significant number of times but has not progressed in the LSU pipeline.

The other important agent in the testbench is the CHI agent, which is in charge of responding back to the different CHI requests from the LSU. To do so, the *chi monitor* continuously observes the output ports of the CHI interface. When a petition from the LSU is captured, it is send to the chi sequence. This sequence, analyzes the received transaction and generates a response sequence that is then sent to the *chi driver*. The driver will send the response back to the LSU using the corresponding CHI subinterface. In the Figure 3.3 it can be observed the different parts of this process.

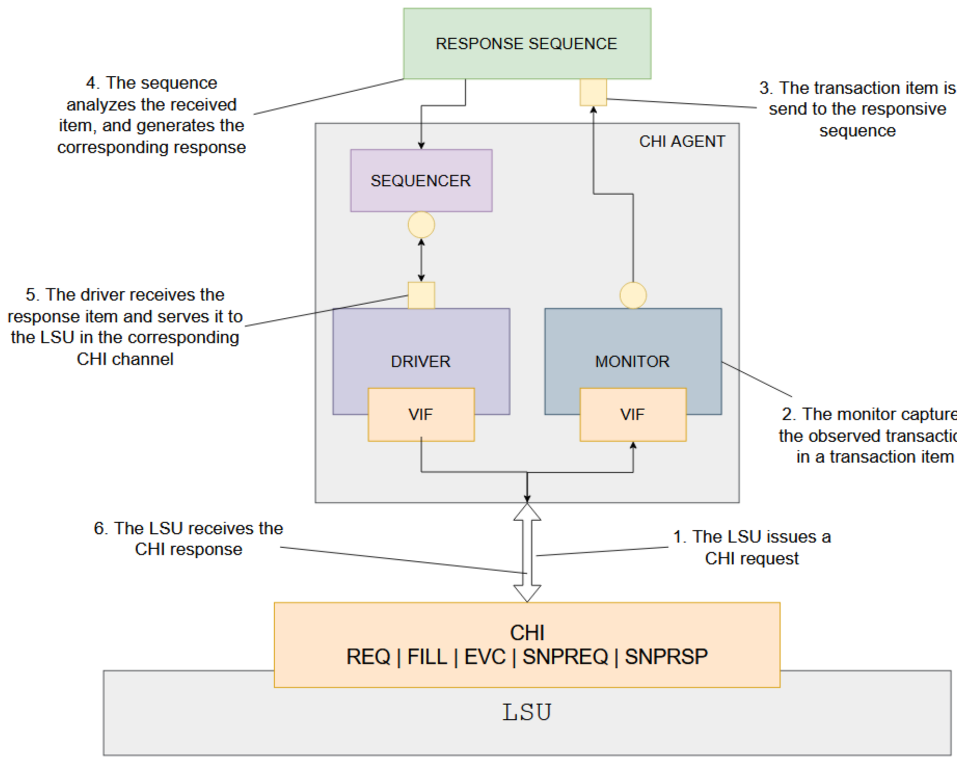
RESPONSE SEQUENCE
4. The sequence analyzes the received item, and generates the corresponding response
3. The transaction item is send to the responsive sequence
CHI AGENT
SEQUENCER
5. The driver receives the response item and serves it to the LSU in the corresponding CHI channel
DRIVER
MONITOR
VIF
VIF
2. The monitor captures the observed transaction in a transaction item
6. The LSU receives the CHI response
1. The LSU issues a CHI request
CHI
REQ | FILL | EVC | SNPREQ | SNPRSP
LSU

Figure 3.3: Interactions of a LSU CHI request with testbench agent

The CHI agent is also responsible to generate the snoop transactions the test has requested. This helps to simulate a multicore environment where other cores are also interacting with the shared L2 cache and request or send data to it. For example when requesting a line in order to read from it, a snoop of type *SnpShared* will be issued to the core that has the line either in exclusive or modified state. To invalidate a line, the snoop types that can be used are *SnpCleanInvalid* to obtain any dirty copy of the line, or *SnpMakeInvalied* so any dirty copy of the line is dropped.

The IPTW agent is in charge of the LSU interface that is connected to the core's frontend, and mimics the PTW requests it issues. Its driver controls the interface, sending the requests provided by the sequence and the monitor notifies the scoreboard when a new request has been issued and completed.

In order to test a reset sequence, the reset agent monitors the pipeline of the DUT. When it detects a window to insert a reset, blocks the other active agents so they do not send any requests, drains the L1, so modified lines are not erased, and resets the state of the LSU by asserting the reset signal of the module.

The last active agent in the testbench is the config agent. This agent controls the configuration ports of the LSU, whether virtual memory is activated, which level of permissions is active, etc. The driver will receive the configuration changes from the sequences, and will hold the same status until a new update is generated.

Finally the coverage agent is an only passive agent. Its job is to monitor different signals of the LSU interfaces as well as inner signals of the design, and collect coverage metrics from the values observed. This information is of great value, since it indicates which parts of the LSU is the test stressing and more importantly, which signals of the design the test is not being able to stimulate.

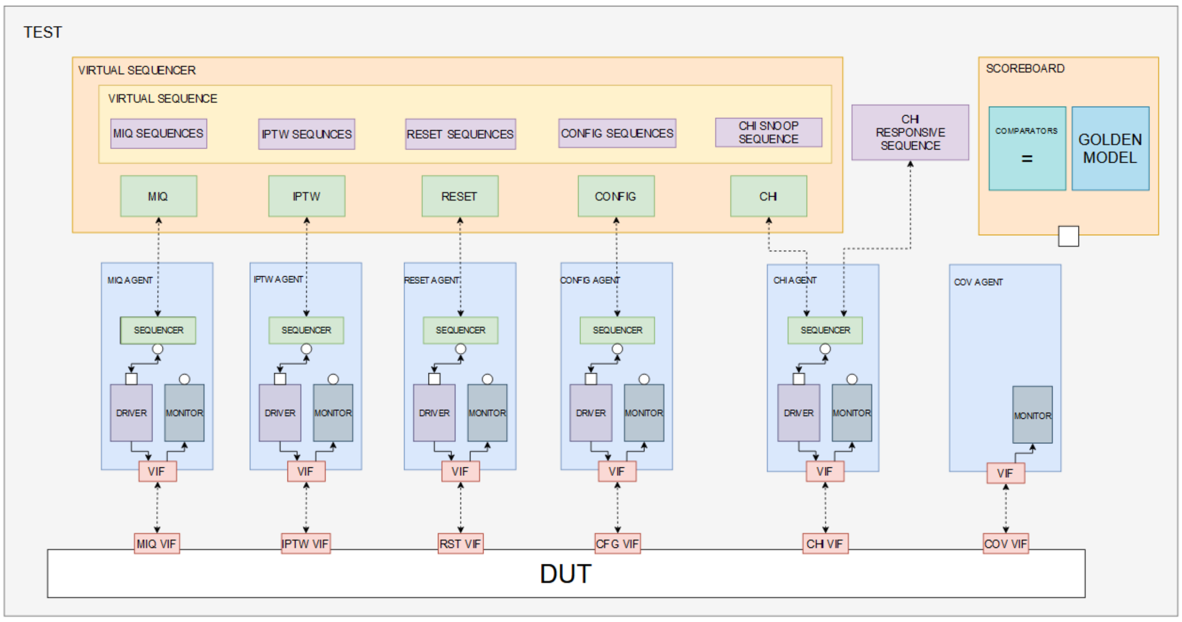

Figure 3.4: Hierchy overview of the different UVM components of the testbench

As in any other UVM testbench, all of this agents have been instantiated in an environment where all the connections between these different agents and the scoreboard have been done. The testbench scoreboard is in charge of checking that the memory pipeline is executing correctly the different instructions issued. In order to do so, it collects all the information captured by the different monitors and compares the results obtained with a golden model, which mimics the functional behaviour of the LSU and it has been written in C++. When it detects a mismatch between the values observed and the values generated by the golden model, it raises an error which stops the simulation and reports the information about it.

To organize and generate sequences a virtual sequencer and sequence has been used. Both are in charge of controlling the different sequences of the test. The following Chapter 4 gives a detail view of the virtual sequence.

# Chapter 4

# LSU language and virtual sequence

To generate the stimuli for the DUT in a UVM testbench, sequences that create random transactional items for the drivers are used. In the memory pipeline, since it can execute different kinds of operations issued in different subinterfaces, another strategy has been adopted. In order to facilitate the creation of new sequences, the strategy that was decided to be adopted was to create a language which the testbench would interpret, translate it to sequences and execute it via its agents. This way, the creation of new tests has been simplified without loosing any feature, a developer do not need to understand how the UVM sequences work in order to create an specific test. Also the automation of creating new tests can be handled to another programs, since they only have to generate an ASCII file compliant with the rules of this new language. Section 5.1 gives more details on how this test generation automation has been performed.

This chapter has been organized in the following way, Section 4.1 will explain the characteristics of the language used to create the test sequences and the interpreter to interpret them. Section 4.2 will describe how sequences are created from the interpreted instructions.

## 4.1 LSU language

The language created for the testbench receives the name of the DUT, *lsu*, and all instruction files will have the *.lsu* extension. The main purpose of the language is to be able to easily create different kinds of sequences to stimulate the DUT, without having to configure many parts of the testbench. The language has the following types of instructions:

- **Load**: When translated to a sequence it generates a load or vector load, with the specified size to the selected address, that is issued via the MIQ agent.

$$\begin{gathered} \textit{load.}\langle \textit{size} \rangle \;\; \textit{dest\_reg},\; \langle \textit{address} \rangle \\ \textit{load\_u.}\langle \textit{size} \rangle \;\; \textit{dest\_reg},\; \langle \textit{address} \rangle \\ \textit{v\_load.}\langle \textit{size} \rangle \;\; \textit{dest\_reg},\; \langle \textit{address} \rangle,\;\; \textit{vl} \end{gathered}$$

When the *load_u* instruction is used, the loaded value is treated as an unsigned value. For the vector loads (*v_load*), the last parameter indicates the number of elements that are going to be fetched.

- **Store**: When translated to a sequence it generates a store or vector store, with the specified size to the selected address, that is issued via the MIQ agent.

$$\begin{gathered} store.\langle size\rangle \;\; src\_reg,\; \langle address\rangle \\ v\_store.\langle size\rangle \;\; src\_reg,\; \langle address\rangle,\;\; vl \end{gathered}$$

- **Snoop**: When translated to a sequence it generates a snoop request that will reach the LSU by the SNPREQ CHI channel.

$$\begin{gathered} snoop.w \;\; \langle snp\_type\rangle,\; \langle address\rangle \\ snoop.nw \;\; \langle snp\_type\rangle,\; \langle address\rangle \end{gathered}$$

  The *snoop.w* instruction will wait until all older instruction of the test to be completed before being issued by the CHI agent. The *snoop.nw* will not wait for any instruction to be completed, and if the channel is ready it will be issued as soon at it arrives to the CHI driver.

- **Control**: In order to control the testbench and the execution of the test, the language supports the following instructions:
  - *block*: It will block the issue of new instruction until all older instructions have completed.
  - *enable_vm*: It will enable the virtual memory in the memory pipeline.
  - *disable_vm*: It will disable the virtual memory in the memory pipeline.
  - *set_chi_wait* It will change the random distribution of the number of cycles a CHI transaction has to wait to be issued to the LSU. The testbench supports four distribution types: *always_zero*, *small_uniform*, *large_uniform* and *binomial.*
  - *reset*: It will set up a reset instruction in the reset agent.
  - *flush*: It will create a flush sequence that will flush the memory pipeline as well as its internal structures.

- **Atomic**: When translated to a sequence, it generates an atomic instruction with the specified size to the selected address, that is issued via the MIQ agent.

$$\begin{gathered} lr.\langle size\rangle \;\; dest\_reg,\; \langle address\rangle \\ sc.\langle size\rangle \;\; dest\_reg,\;\; src\_reg,\; \langle address\rangle \\ amo.\langle size\rangle \;\; \langle amo\_type\rangle,\;\; dest\_reg,\;\; src\_reg,\; \langle address\rangle \end{gathered}$$

  The *lr* instruction is the same as the one in the RISC-V ISA[14]. It will load a value to the specified register and set to valid the reservation monitor. If any other memory instruction or snoop to the same line of the *lr* instruction is issued, the reservation monitor will be disabled.

The *sc* instruction will store the value of the source register to the specified address if the reservation monitor is valid, else the store won't success. The result of the outcome of wether the store has been able to succeed will be indicated in the destination register. The *amo* instruction performs an atomic memory operation of the indicated type to the selected address. The supported operations are: *swap*, *add*, *and*, *or*, *xor*, *max*, *maxu*, *min* and *minu*.

The supported values for the size parameter are: *8*, *16*, *32* and *64*. It indicates the size in bits of the values that are going to be loaded or stored. To specify an address it can be use an hexadecimal value or a reference to another instruction, for example:

```
lr.32 r1, 0x800000000000
block
snoop.w unique @1
sc.32 r2, r3, @1
```

In the above code, the address of the two last instruction will be the same as the instruction they are referencing, the first one.

The *lsu* files will be lexed and parsed by the testbench interpreted who will generate an array of instructions that will be consumed by the virtual sequence.

## 4.2 Virtual sequence

As it has been discussed along the thesis, the DUT has multiple interfaces and so multiple agents to control each one of them. That also requires different kinds of sequences to create the events for each one of the agents drivers. In order to organize all the test sequences the strategy of a virtual sequence is the most practical one. In a generic UVM view, a virtual sequence is a container to start multiple sequences on different sequencers in the environment.

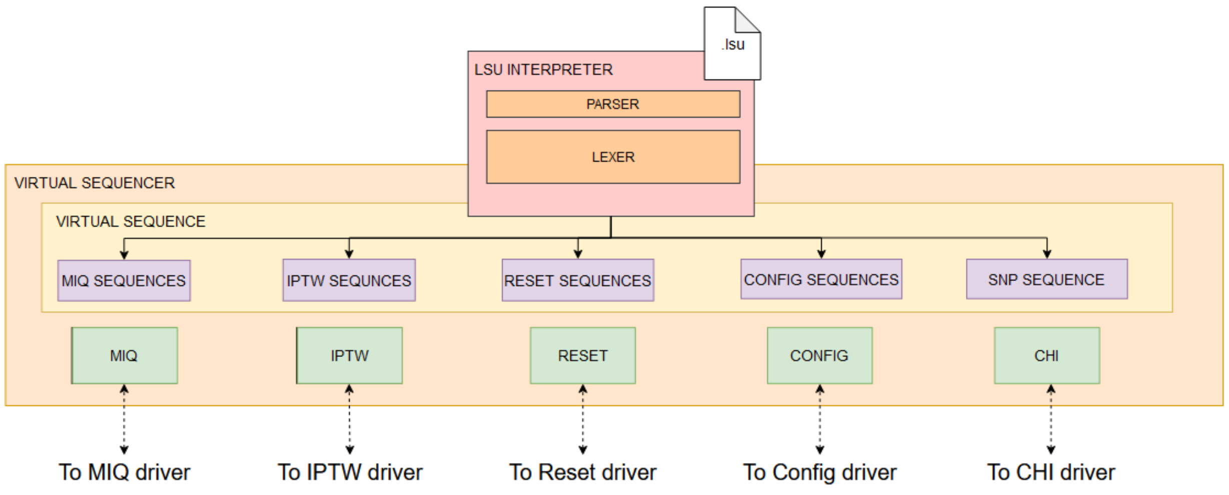

Figure 4.1: Diagram of the virtual sequence and lsu interpreter

In the testbench developed in this thesis, the virtual sequence object has been used to receive all the instructions parsed from the *lsu* file. Then after decoding which kind of instruction is, it creates and configures the specialized sequence that sends the transactional item containing the action described by the parsed instruction. A virtual sequencer is used to hold the different references of the agent sequencers. When starting the new configured sequence, the virtual sequence makes use of these references to pass them to the created sequence.

The instruction is also sent to the golden model in order to be executed there as well to produce the expected result, and compare it later on the scoreboard, once the LSU has finished the execution of the instruction.

# Chapter 5

# Coverage directed genetic test generation

The traditional way to generate sequences is by randomizing, usually with some constraints, the different possible values of the transactional item. With this method, a coverage around 70-80% can be easily archived, and in order to cover the rest, directed sequences that stimulate specific parts of the design are created. When the complexity of the DUT is not high, or the number of different input scenarios is relatively low, the constrain-random technique, to generate input sequences works fairly well. The memory pipeline that has been verified in this project, has a large number of different interfaces that interact with each other. On top of that the DUT is able to execute different kinds of operations. Also the execution state in a given moment is very complex, since there are multiple stages in the pipeline, different kinds of arrays, a coherent state in the L1, etc. For all of this reasons, when discussing the testbench architecture, it was very clear that a random approach would not yield great coverage numbers, and a lot of effort would be needed to create directed sequences to close coverage.

The strategy decided to implement in order to generate tests for the DUT, has been a coverage directed test generation based on a genetic algorithm. As explained in Section 2.3, this strategy makes use of the coverage results of the tests generated in order to improve the next generations of tests. The genetic algorithm implemented in this solutions is explained in Section 5.1. The coverage results are presented in Section 5.2. Thanks to this test generator 12 issues have been found in the memory pipeline RTL. Some of these bugs are analyzed in section 5.3.

## 5.1 Genetic algorithm

This approach makes use of a software called *Micro GP3* [3] developed by a research team at University of Torino, for the purpose of automating test generation specifically for microprocessor design verification.

The *Micro GP3*, makes use of a genetic algorithm, which is a search heuristic that reflects the process of natural selection, where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. With this approach, the program tries to find, in several iterations, the characteristics needed so the generated tests yields the maximum coverage. *Micro GP3* is configured with an instruction library, that is, the different kinds of operations, and its configurations, it can use to create the tests. After that, the program starts with an initial population. Once the tests have been run, and produced a score, two types of genetic operations are performed on the tests in order to produce the new generation of tests:

- **Crossovers**: Where two parts of two different programs are selected and interchanged between them.
- **Mutations**: Where parts of the program are added, removed or altered from the original one.

These new generated tests, are run on the DUT and a score is given to each one of them. Based on this score, usually extracted from the coverage, the tests that performed better are selected to be the parents of the new population.

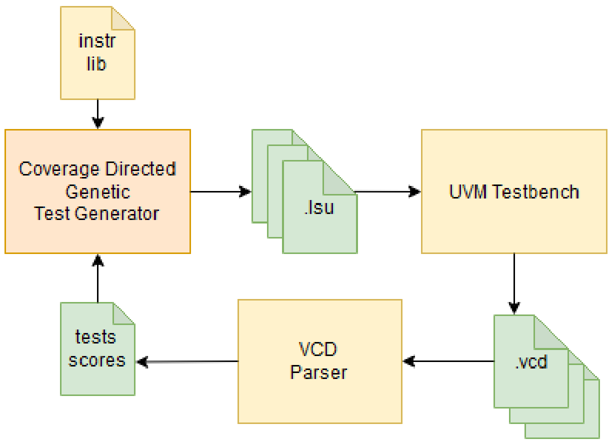

Figure 5.1: Genetic test generation flow

The integration of the *Micro GP3* program, to the testbench can be observed in Figure 5.1. The instruction library has been populated with all the memory operations supported by the testbench and the LSU, then all the tests produced by the genetic algorithm are run in parallel on the testbench. Of each run, a file in *vcd* format is generated containing the waveforms of all of the signals of the DUT. Parsing this file, it is obtained the toggle coverage of each signal of the design and base on that, a score is computed for each test.

These scores are then introduced to the genetic algorithm, in order to select the best tests of this generation as the parent candidates for the next population of tests.

## 5.2 Coverage results

Automatizing this flow has allowed to stress test the DUT, while upgrading the quality of the tests. This improvement can be observed in Figure 5.2 where it is plotted the LSU coverage of the tests generated by the genetic algorithm. In this plot it can be observed as well the genetic variations the algorithm performs on the tests, when sometimes a test performs worse than one from the previous generation. When the observation is done with the accumulated coverage, it can be seen how on each generation there are contributions to the coverage. This is shown in Figure 5.3 where the coverage of the LSU and different submodules of it has been plot.

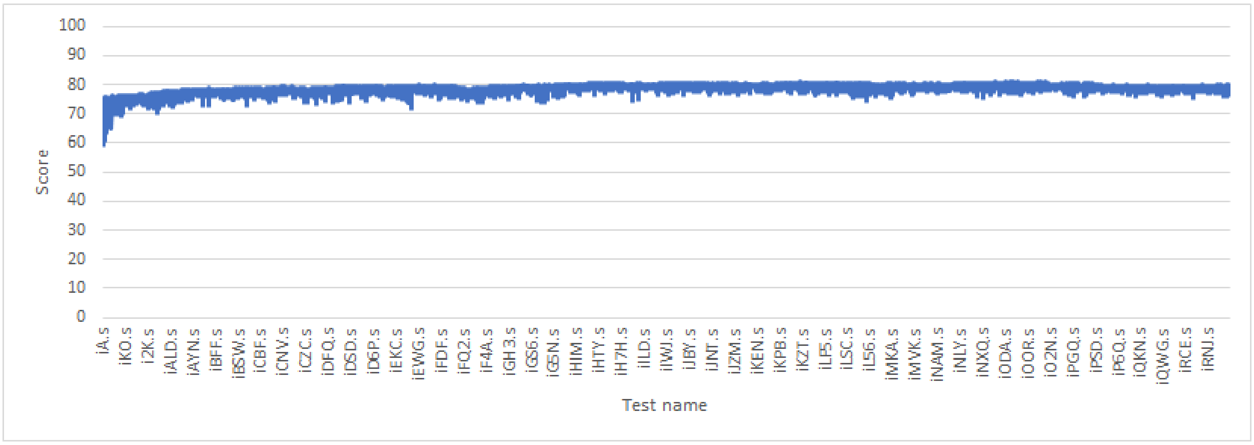

Figure 5.2: Score of the tests generated by the genetic algorithm.

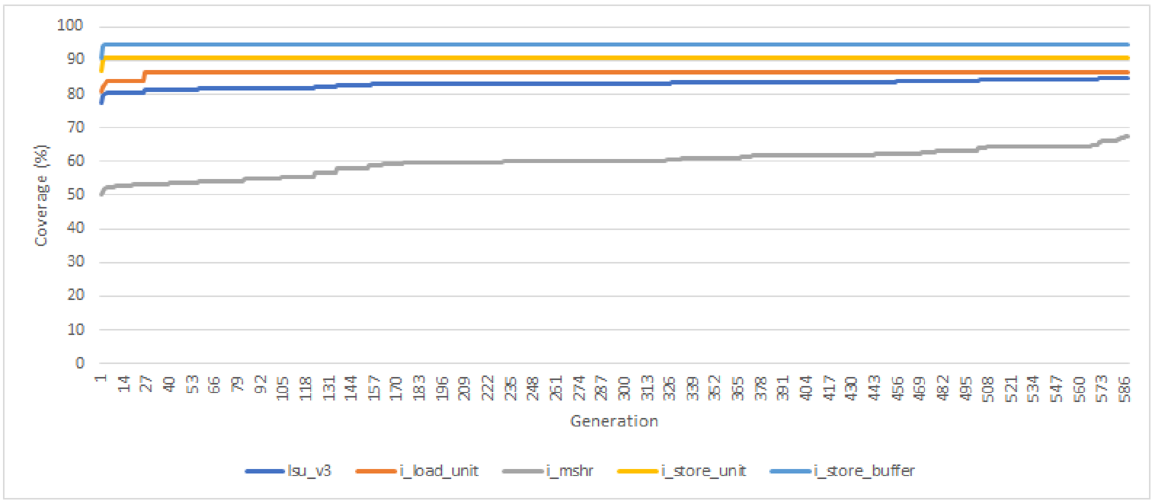

Figure 5.3: Accumulated coverage of each genetic generation

Some experiments with specific data structures have also been performed. For example Figure 5.4 shows the occupancy of the store buffer structure, that is, how many store operations are in-flight in the SB. As it can be observed in the graphic, the occupancy mean across the different generations improves for the first six generations. By the nature of the genetic algorithm, which relays on doing random changes and swapping parts of the tests, accomplish a test that is able to full the store buffer, which would be to issue a lot of stores with some load, is really hard.

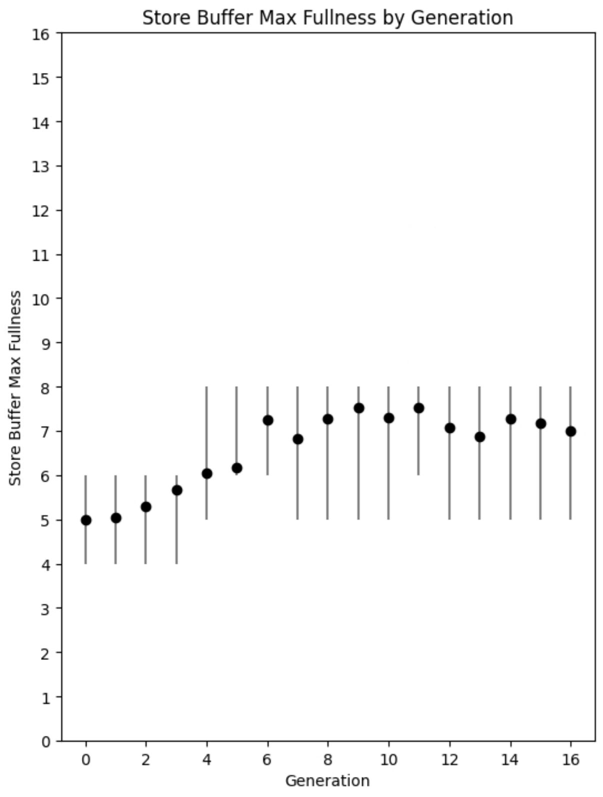

Figure 5.4: Store Buffer occupancy along different genetic generations

Thanks to this approach, although it did not excel in some coverage tasks, it did manage to find several bugs in the design. The following section will discuss the nature of this bugs and how they have been fixed in the RTL code.

## 5.3 Bugs found

By the time this project started the core that integrated the memory pipeline, was already booting Linux and executing several scalar and vector benchmarks without any errors. Although most of the bugs were already found, some of them were still to be discovered.

To the date this thesis has been written, thanks to the testbench and the genetic algorithm approach, 12 functional and performance bugs have been found. Out of this 12 issues found, 3 of them have been related with IPTW petitions, 3 bugs about Load Reserve and Store Conditional interactions, 2 bugs about snoops and store interactions, 1 bug related with unaligned memory accesses in AXI regions, 1 bug in the MSHR structure, 2 bug related with vector memory operations and 1 performance issue regarding stores and rearms. The following sections explain more in depth some of this bugs found.

| | Functional bug | Performance bug |
|---|---|---|
| IPTW | 2 | 1 |
| LR/SC | 1 | 2 |
| SNOOPS | 2 | 0 |
| AXI | 1 | 0 |
| Store Buffer | 0 | 1 |
| Vector operations | 2 | 0 |
| MSHR | 1 | 0 |
| **Total** | **8** | **4** |

### 5.3.1 Performance bug: LR monitor is cleared when a read is performed in the same line

As it has been explained in section 4.1 when a Load Reserve (LR) instruction is executed, it does two things:

1. It loads the requested value to the destination register.
2. It sets the reservation monitor to valid.

The reservation monitor it stores as well the address of the line in which the LR was obtaining the data from.

Before fixing this issue, the logic of the LSU was being extra conservative, and was disabling the LR monitor if a memory operation was begin executed on the same line the last LR got its data. This behaviour is necessary when executing a store instruction since the line can potentially change its state. In the case of a load, since the line already has the necessary read permissions from the previously executed LR instruction, and no state update is needed, the monitor doesn't need to be disabled.

In order to solve this performance problem, when executing a load operation whether the line address matches or not, the LR monitor is not disabled.

An example code that was provoking this performance issue was:

```
lr.64 r28, 0x800000000040
load.32 r4, 0x800000000048
sc.32 r4, r10, 0x800000000040
```

### 5.3.2 Functional bug: Snoop wrongly invalidates a store and re-request line

This bug is caused by a store that has missed and is waiting on the store buffer for the requested line to come from the L2 via the CHI fill channel. During this wait, a snoop request of type SnpOnce arrives making the LSU logic to wrongly re-request the line.

The SnpOnce snoop type, will request to obtain the latest copy of the cache line, preferably without changing the state of the cache line at the Snoopee. That means the snoop is a request for the LSU to return the current value of that line, without changing its state.

In this case, the logic of the LSU was only checking that a snoop entered the pipeline, in order to invalidate the store and request again the line. For the other supported types of snoop, this behaviour was okay since they either invalidate or downgrade the state of the line, but in the case of the SnpOnce, the state of the line is not modified. In order to fix this an extra comparison with the snoop type was added to the store invalidate logic.

The *lsu* code that provoked this issue was:

```
snoop.nw shared 0x8000000569
snoop.nw unique 0x80000000005A9
store.32 r23, 0x80000000005A8
snoop.w once 0x80000000005A8
store.32 f8, 0x8000000569
```

### 5.3.3 Functional bug: Store conditional incorrectly executed as successful

In this case the bug was related with the instruction pair LR and SC and cache line eviction. The *lsu* code that provoked this issue was:

```
lr.32 r1,      0x800000004000
load.32 r1,    0x800000001000
v_store.32 v1, 0x800000002000, 1
sc.32 r1, r0,  0x800000003000
```

This code sets the reservation monitor to valid in the first instruction, then the two instructions before the SC will evict the cache line of the LR and replace it with another different one which will be allocated in the same way. When the SC arrives it will incorrectly succeed since it will be a hit in the cache, and the cache set and way will be the same ones as the LR instruction.

To be able to detect this scenario, the reservation monitor will store as well the tag of the line of the LR instruction that has set it to valid. When the SC is executed, in order to check if the reservation is valid, it will compare the tags of the reservation monitor and the address of the SC. Another fix to this issue would had been to monitor the cache evictions for the reservation monitor, and invalidate it when the line that is evicted is the same as reservation is being hold.

# Chapter 6

# Conclusions and future work

## 6.1 Conclusions

This thesis presents the efforts done in order to be able to verify the memory pipeline of a RISC-V coherent core as a standalone module. In the first chapters, the capabilities of the design to be verified and the requirements of the testbench are explained. Then a detailed view of the verification environment and how the modules that interact with the memory pipeline have been emulated is given. Finally, it is disclosed the test generation methodology adopted for this project and how it has been integrated with the verification flow.

The project contains most of the important parts of a verification project for RTL. From defining the necessities of the design under test, building a testbench that serves as envelop for the RTL logic and allows its simulation as it was integrated in the whole chip design, providing a reference model of the module in order to continuously check its correct behavior, and creating a test generation flow that is aware of the quality of the generated tests, and tries to improve it on each generation.

One of the main goals of the project, was to improve the quality of the RLT logic of the memory pipeline has been accomplished, since a total of 12 functional and performance bugs have been found and corrected. Also the coverage numbers obtained indicate that the logic is in very good shape.

Finally, the contributions of this project will enable the design team to stress and verify future improvements and modifications of the memory pipeline design much faster, with the ability to create different scenarios and configurations for the test to execute.

## 6.2 Future work

When verifying digital design, there is always space for improvement and developing a more complete verification infrastructure. Better coverage, new test sequences, defining new functional coverage points, etc. Moreover, the DUT will be upgrading until the freeze date, and for futures releases either by adding new functionalities, reducing timing paths or improving the area. The verification environment must be evolving with those changes as well in order to keep up with the verification.

On top of the list of future work for this project is the creation of a coverage plan. There it will be listed all the cases that must need to be tested in the design in order to consider it in good shape. This will require the creation of more functional coverage points that will give new objectives for the test generation algorithm.

One verification aspect that would complement this project is the creation of assertions. In verification, assertions work as in any other programming language. In execution time, they check that a condition remains true, otherwise they fail. Assertions are a key aspect in any verification environment since they help to detect bugs, and more importantly indicate where the bug is located, and which signals are responsible of it. This save a lot of time that it would have been dedicated to review waves, in order to locate where the bug has been produced.

This project has focused on the verification of the whole memory pipeline, which is composed by different modules. Sometimes, a certain logic of the design, needs a specific state, or a consecutive specific states in order to be reached. When verifying from a top module it is difficult to archive the necessary conditions, that is why a standalone verification of that specific module needs to be done. From here there are different verification approaches that can be taken, either a UVM testbench is build around this specific module or a formal verification approach is taken. Using formal, there is no need to build a complete testbench for the module, and only instrumenting the code with assumptions and assertions is required. With them the formal tool is able to verify that the logic of the design, behaves as the assertions state.

Finally having another type of algorithm for the coverage directed test generator would be an important contribution for the project, firstly because generated tests would not be biased by the genetic algorithm, and other kinds of tests would be created. Secondly an algorithm that is able to work with specific tasks would help to complete the coverage gaps, the genetic algorithm have. A good candidate for it would be an algorithm based on Reinforcement Learning (RL). RL is a machine learning method based on rewarding desired behaviors and punishing undesired ones. With this approach, an RL agent is able to interpret its environment, the DUT, and take actions by learning through trial and error.

# Bibliography

[1] Avispado. *Avispado web page.* URL: https://semidynamics.com/products/avispado

[2] Accellera. *Universal Verification Methodology (UVM).* URL: https://www.accellera.org/downloads/standards/uvm

[3] Corno, F. and Cumani, F. and Squillero, G. *Exploiting Auto-Adaptive µGP for Highly Effective Test Programs Generation.* In: ICES'03 *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware. 262–273*

[4] OpenHardware. *OpenHardware web page.* URL: https://www.openhwgroup.org/

[5] lowRisc. *lowRISC web page.* URL: https://lowrisc.org/

[6] Ibex RISC-V core. *Verification page of Ibex.* URL: https://ibex-core.readthedocs.io/en/latest/03_reference/verification.html

[7] Ibex RISC-V core. *Ibex repository.* URL: https://github.com/lowrisc/ibex

[8] Ariane RISC-V core. *Ariane repository* URL: https://github.com/openhwgroup/cva6

[9] Fine, S. & Ziv, A. *Coverage directed test generation for functional verification using bayesian networks.* In: '03 *Proceedings of the 40th annual Design Automation Conference. 286-291*

[10] OpenTitan. *OpenTitan web page.* URL: https://opentitan.org/

[11] Verification Guide. *UVM sequence.* URL: https://verificationguide.com/uvm/uvm-sequence/

[12] W AGNER , I., B ERTACCO , V. *Microprocessor verification via feedback-adjusted Markov models.* In: '07 *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits Syst. 26, 6, 1126–1138.*

[13] AMBA 5 CHI Architecture Specification. URL: https://developer.arm.com/documentation/ihi0050/c

[14] RISC-V ISA. *RISC-V Unprivileged Instruction Set Architecture.* URL: https://riscv.org/technical/specifications/