Design under test interface implementation and stimulus in the verification of a RISC-V Vector Accelerator

January 26th, 2022

Author: Víctor Jiménez Arador

Supervisor: Nehir Sonmez — Barcelona Supercomputing Center
Co-supervisor: Óscar Palomar — Barcelona Supercomputing Center
Tutor: Miquel Moretó — Universitat Politècnica de Catalunya

Fall 2021
Master in Innovation and Research in Informatics
High Performance Computing Specialization
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Acknowledgments

First of all, I would like to thank my thesis supervisors: Miquel Moretó, Óscar Palomar and Nehir Sonmez from the Barcelona Supercomputing Center. I want to thank them for their help and supervision, both for this thesis and my work at BSC. Thanks to them, I could learn and start my current career.

Then, I would like to shout out to the whole BSC Verification team involved in verifying the Vector Accelerator, as their efforts are also reflected in this thesis. Extra mention to Francesco Minervini from the RTL team, who made our job much easier and was always really kind to us.

Special thanks to my closest partners and friends in the BSC Verification team, Josep Sans, Marc Domínguez and Mario Rodríguez, without whom a big part of my work would not have been possible.

Finally, I would like to give a huge thanks to my family for keeping me motivated and supporting me at every moment of my life.

Abstract

The production of a microprocessor is one of the most complex and expensive processes in the industry these days. These high costs are why big companies dedicate most of their efforts to design verification during the development of these projects. Design verification is vital to be able to deliver an error-free design. As the final manufacturing of these products is expensive, no company can afford to spend money on defective designs. Governments and associations are investing in research projects with the recent open-source trends. These allowed entities like the Barcelona Supercomputing Center (BSC) to start developing their designs. Considering how hard it is for these entities to receive inversions of this type, they have to work hard in design verification.

One of the critical aspects of design verification involves applying the correct stimulus to the IPs or modules to be verified. The verification engineers must generate a correct but diverse stimulus to drive the design under test. These stimuli are often achieved using Universal Verification Methodology (UVM) and directed testbenches. However, this task is sometimes not easy, where the design under verification might have a very complex interface or have a vast range of stimulation possibilities.

In this thesis, a UVM-based testbench is presented for the design verification of a RISC-V Vector Accelerator. From design specifications to the testbench implementation, this work explains its structure and the reasoning behind its specific characteristics. This testbench can provide random stimulus through the interface of the Accelerator and handle the execution of vector instructions from the RISC-V Vector specifications. Although it is full of features, we will be focusing on the module interface treatment part of the testbench in this work. Finally, we will review its strengths and weaknesses and how we could improve these.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Producing a microprocessor or a similar design is a costly and challenging process. Currently, almost all these designs are done by huge companies because those are the only entities with resources to afford these processes. These companies have big and diverse teams devoted to each step of the production process, one of which is the Design Verification team.

In the last few years, research entities have started developing their designs thanks to governments support. With as few chances as researchers have, the teams must assert that their design is as correct as possible before sending it to the factory. That is why verification has such enormous relevance in modern-day projects, as it is the primary tool that engineers have to ensure the correctness of their designs.

In this chapter, Section 1.1 introduces the circumstances and the need for verification that motivates this master thesis. In Section 1.2, the contributions provided in this master thesis are presented. Finally, in Section 1.3, the thesis structure is described.

1.1 Motivation

Even if RTL design and CPU microarchitecture are well and widely taught at many degrees, it is not so common to dedicate the time it deserves to design verification. Although it is mandatory to have a verification process in such a project as a core or an accelerator, the techniques involved in them are almost unknown for most students. As one could expect, this secrecy is because big companies do not often disclose their tools and workflows to the public. As these processes are only performed in massive projects like producing the newest generation core, they never reach the smallest companies or researchers. The whole environment produces a cycle where the most prominent companies must train all the professionals and cannot always hire the desired level of experience.

At the same time, this cycle implies that smaller companies or research entities do not get access to optimal tools and knowledge commonly used that is standard in the industry, which means that their projects are behind right from the start. It gets even worse if we focus on research entities, like the Barcelona Supercomputing Center (BSC), that rely on government funding to produce their designs and most of the time do not sell them.

For the newly created microarchitecture department at BSC, the chance of participating in the *European Processor Initiative* (*EPI*) was a huge opportunity. In this project, the BSC takes on the design of a Vector Accelerator integrated with a scalar core in a more extensive environment. This accelerator needs to be bug-free and capable of communicating with the rest of the project modules. Therefore, thorough verification of the IP is mandatory and necessary in developing the Vector Accelerator.

As soon as this project came across the BSC's horizon, the corresponding teams started to get prepared or formed, in the case of the verification team. The experienced verification engineer is not very affordable and the previous verification experience in the BSC was almost inexistent, so the newly created team was formed by graduate engineers with no knowledge in verification. This little experience meant, apart from the delay due to the team's formation, that the team had to go through a ramp-up process that gave them the pointers to where they should direct the verification process. This delay caused the Vector Accelerator to be almost implemented by the time the verification efforts started and the need for a fast and efficient initial implementation.

1.2 Contributions

In this document, we present two main contributions to the verification process of the Vector Accelerator:

- Design under test interface interaction: using Universal Verification Methodology (UVM) and other sources, stimuli must be generated to observe and test as many capabilities of the design as possible.

- Memory operations emulation: in this project, the interaction with memory is done through the scalar core, and in simulations without it, the verification team must handle both the interaction with the core and the emulation of its memory.

The verification of the Vector Accelerator was a team effort and we developed multiple tools and features for that purpose. However, some members of the team specialized in certain aspects of the verification infrastructure. Josep Sans and Iván Díaz were in charge of tweaking a RISC-V binary generator to fit our needs. Mario Rodríguez was mainly in charge of adapting an Instruction Set Simulator (ISS) to our testbench and creating a set of Continuous Integration pipelines together with Marc Domínguez. These will be explained in Subsections 3.4.1, 3.4.2 and 3.4.6, respectively. Together, we developed a testbench for the Accelerator full of features, among which we find the ones explained in this document.

Firstly, we will describe how we managed to feed the Vector Accelerator with instructions to execute and the responses in the interface necessary to complete them. All the techniques used and their goals and motivations will be explained. In addition, we will explain how we asserted that we were taking the right approach. This interaction aimed to stimulate the Vector Accelerator in the broadest way possible, showing and testing all the design features.

After that, the document will focus on the Vector Accelerator memory operations. These had to be implemented in a particular way in the verification environment as they involve much communication with the scalar core. These operations implied significant changes

to the verification environment and a clever and complex way to follow and check their results.

Memory operations are one of the critical aspects of the Vector Accelerator, but they are also one of the most delicate points. That is why they were essential to verify and why their dedicated part of the environment was fundamental in the verification process.

1.3 Thesis structure

The following chapters describe the contributions to the verification of the project mentioned above. The remainder of the document is structured as follows:

- In Chapter 2, there is an introduction to the project along with the necessary background of RTL design and verification.
- In Chapter 3, EPI Architecture and Design Verification Infrastructure, there are details on the Vector Accelerator, our design under test, and the whole verification environment the whole team created.
- Chapter 4 describes the first main contribution, the UVM environment and its stimulus generation, with a particular focus on its interaction with the Vector Accelerator interface.
- In Chapter 5, following with the interface interaction explanation, there is a description of the peculiarities and insights of the memory operations handling in the environment.
- Finally, in Chapter 6 and Chapter 7, there is a reflection on how the contributions helped in the verification process and how they could be improved along with possible future work and the conclusions.

Chapter 2

Background and Related Work

In this chapter, key microprocessor design and production concepts are introduced and explained in Section 2.1. A brief introduction to design verification and the EPI project are provided in Sections 2.2 and 2.3, respectively. In Section 2.4 the RISC-V ISA is presented, with particular focus on its vector specifications. Finally, pointers to related work and a description of the previous work in verification in the Barcelona Supercomputing Center are shown in Section 2.5.

2.1 Processor Production

Since the Intel 4004, the first commercial semiconductor processor, came out in 1971, many companies and entities have produced their microprocessor designs. Fifty years and many generations of processors later, making a chip is still a complex task, or even more, considering designs have grown bigger. Producing a design involves several steps, from the initial conception and design to its physical manufacturing.

Every stage of this process must be done carefully and accurately to achieve a successful design. Among these stages, we find the following:

- Specification: In this step, the features and characteristics of the processor are listed and described. Often underestimated, the specification stage is one of the most critical steps in producing a design. All details must be specified and everything must be correctly connected, with no space for ambiguities. Any problem could lead to issues in the implementation and verification stages, complicating the work across teams. For example, a poor interface protocol description could lead to a misunderstanding between the engineers that would cause a delay in the already tight production schedule.

- Implementation: Once the specification is complete comes the implementation phase. The details in the specifications are portrayed in the RTL code, usually written in the Verilog or VHDL languages. In the beginning, each engineer will work in a module or set of modules, writing the code and the needed testbenches to test them quickly. After that, all the modules must be integrated, and all the engineers will collaborate to connect everything. If many engineers can work

independently in their assigned modules and integrate them easily, the quality of the specification is proven. This step tends to be the longest, but the quality of the specifications highly influences this.

- Verification: In the meantime, and with specifications in hand, a verification plan will be implemented to ensure that the design works as expected. Together with the specification stage, this step is one of the most important in the process. In it, as in the implementation one, the quality of the specifications is asserted. The verification engineer needs these to be complete and exact so that he or she can test all the necessary features and develop an ad-hoc environment to verify the design. This environment and all the verification features included in the plan must follow the specifications and a misunderstanding could cause the whole verification effort to be useless.

- Fabrication: After a design is verified, it will be sent to the factory for its manufacturing. This step is the most expensive one, as producing a physical chip is costly. Therefore, the design must be as perfect as possible before this stage. Very few companies or entities in the world can afford to produce a processor, but almost none can afford to produce a deficient processor. Once the processors are received, the engineers will perform further physical tests to ensure that the design works as expected in the real world.

In the previous list there are many missing steps, which are shown in Figure 2.1.

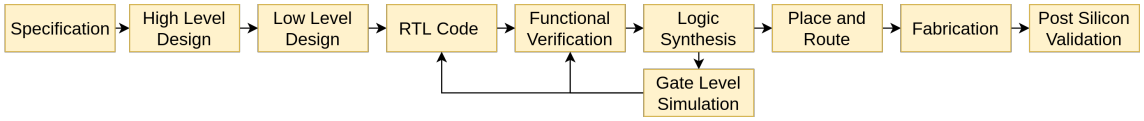


Figure 2.1: Design production flow [7]

Even if all engineers have improved these steps and have learned better and more efficient ways of producing their designs, the expensive nature of the fabrication of a processor still makes it a challenging task. Specialized teams for each step of the process collaborate with the rest to successfully carry out the production of the design.

The whole process is costly and almost no entities can afford to waste money on fabricating deficient processors. Hence, companies typically dedicate a big part of their project funds to verification. A good verification process ensures that the design sent to the factory is correct and follows the corresponding specifications.

2.2 Processor Verification

Design verification is the set of techniques used to assert that a design works as expected. These include tasks like creating a simulation environment, finding ways to check the behaviour of the design or setting up automatic testing tools. As stated, verification is vital to avoid producing a deficient design. However, it is also beneficial to help the design team ensure that everything specified has been implemented as expected.

Initially, the verification strategy was custom and made up for every project. For example, for the Intel 4004, Federico Faggin says that he “had to figure out and create a random-

logic design methodology for silicon gate technology that did not yet exist" [14]. He worked under a tight schedule and had to do the design and verification tasks simultaneously. However, after decades of refining the verification process, it is much more standard and straight than then. Even if each design requires verification planning and thinking before the process starts, we have tools and previous work that leads us to at least a reasonable verification process.

Strangely enough, the basic methodology has not changed that much since the first Intel 4004 was sold. Random testing, if well implemented, is still one of the best ways of verifying a design. Random stimulus, together with many other techniques, allows verification teams across the world to assert the correct behaviour of their designs. Even though each process is different, their verification processes resemble many aspects. For example, they all usually have a testbench or similar, where the design unit is instantiated and through which stimuli are provided. Additionally, extra features are added to this testbench to check the correctness of the behaviour (like *Scoreboards* or *Assertions*) and to measure the quality of the verification process (*Coverage*). More recently, automatic testing pipelines have been implemented to increase the number of situations to which the design under test (*DUT*) has been exposed (*Continuous Integration* (*CI*)). These techniques will be described in the following sections of this document.

2.2.1 UVM

Universal Verification Methodology (UVM) [2][17] is a standardized methodology for verifying RTL designs. Based on System Verilog [10] and standardized by Accelera, it inherits many ideas from Open Verification Methodology (OVM).

The structure of UVM testbenches makes it easy to support constrained random stimulus, which is one of their essential parts, as it enables the users to generate a wide range of cases to test a design. In addition, UVM is thought to be re-usable and extendable. For that, it takes advantage of Object-Oriented Programming (OOP).

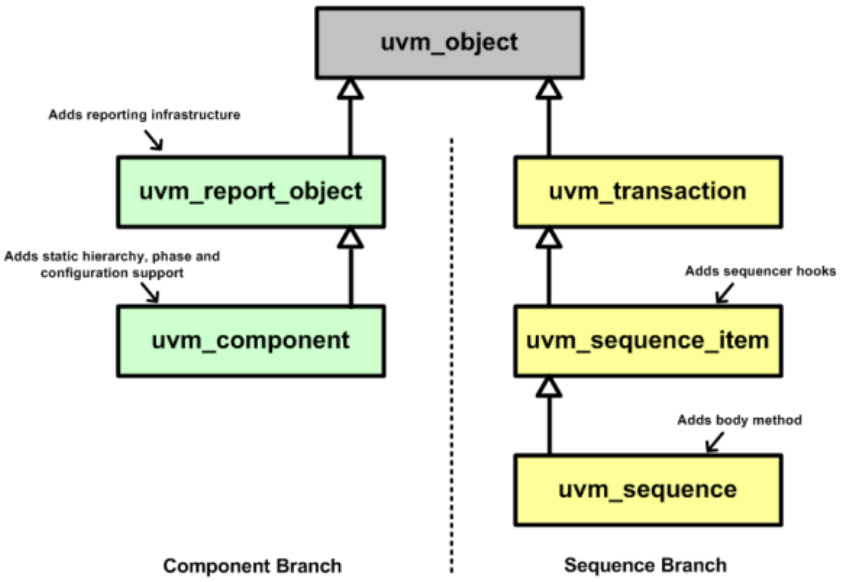


Figure 2.2: UVM Inheritance Diagram [17]

All of the types of classes in UVM inherit from a base class called UVM Object. In Figure 2.2, from the UVM Cookbook, all the basic children classes from UVM Object can be seen. This way, the main classes that *extend* from UVM Object are: UVM Component, UVM Sequence Item and UVM Sequence.

In summary, UVM Sequences produce Sequence Items following a determined or random pattern, which UVM Components use to communicate data between them. Generally, a UVM testbench strength in producing valid and complete stimulus comes with a proper Sequence implementation.

Class instances are created and destroyed as in many other OOP utilities or languages. In the case of UVM, Sequences are requested Sequence Items (also called *transactions*), so they create an instance of the class Sequence Item and *randomize* it with the corresponding function, particular of the class. By extending the base Sequence Item class, one can create a transaction class that fits the needs of its testbench. In addition, the base class methods can be overridden to customize the new one.

In this way, one can create a transaction class that contains different values and has a custom randomize method that sets these to random, using UVM standard methods or custom values. This transaction is created and randomized inside a Sequence that will send it to the component that requested it through the Sequence methods.

Furthermore, one could have more than one custom transaction type and Sequence, randomizing what type of transaction it creates at every request. This feature makes it possible to introduce random stimuli in a directed test.

This is possible thanks to polymorphism and OOP, which eases reusability and future environment extensions. For example, one could create a custom "base class" of many of the UVM Objects needed to verify a design. Later in that same project, the engineer could extend his class to create a custom dedicated case. Additionally, later in some other project, the engineer could use that same base class, if generic enough, to extend it and create a custom dedicated class for the other project.

Apart from ordinary objects, polymorphism is used in UVM Components, which typically compose a UVM testbench. These perform actions related to the testbench and have their functionalities. For example, some components create Sequences (and transactions) and others communicate with the DUT.

UVM Components make use of what is called UVM Phases. These are used to have a consistent testbench execution flow and in them, big steps are executed in order. There are three groups of phases:

- Build phases: where the testbench is configured and constructed.
- Run-time phases: where the testbench runs the test case.
- Clean-up phases: where the results of the run-time phases are collected and reported.

The complete set of phases and the group that they belong to can be seen in Figure 2.3.

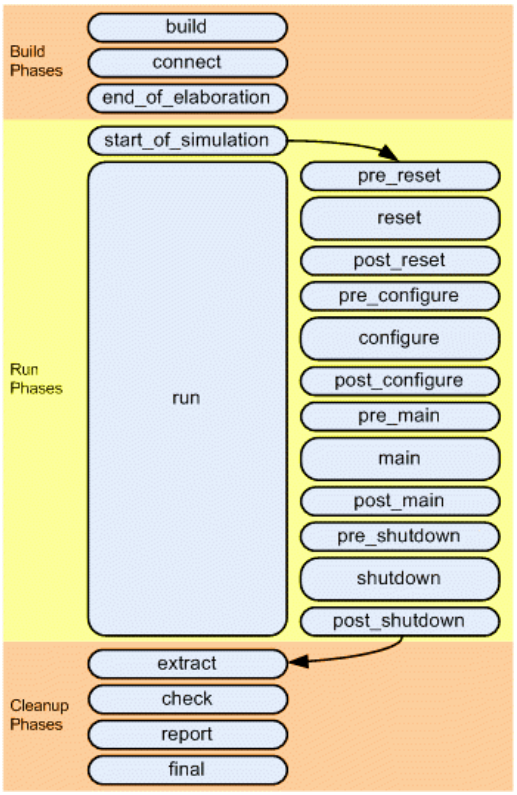


Figure 2.3: UVM Phases [17]

The UVM Component base class contains virtual methods that are called during each phase. When extending from this base class, the engineer can choose whether to implement these methods or not. If so, the component will take part in the corresponding phase.

For example, one component might need to initialize the instance of another one in the build phase. To do so, it must have its virtual method *build_phase* implemented. Otherwise, the component will do nothing in phase. It is worth noting that all components execute the specific phases in the same time window. That means that if two components implement the run phase, they will execute them simultaneously.

Using these phases allows UVM Components to be developed independently but still cooperate and execute their tasks with the certainty that things they depend on are ready as they were processed in previous phases.

In addition to the previous and to ease the use of polymorphism, UVM has the UVM Factory. Its purpose is to enable an object of one type to be substituted with an object of a derived type without changing the testbench structure or even the testbench code. This replacement is called “override” by either instance or type. This functionality is convenient for changing Sequence behaviour or replacing one component with another. Certain coding conventions must be followed to take advantage of the factory, where the components or objects must be registered in it, default constructors must be added to these and they have to be *created* in a particular way.

In Figure 2.4 we have an overview of a simple UVM environment. In it, we can see the

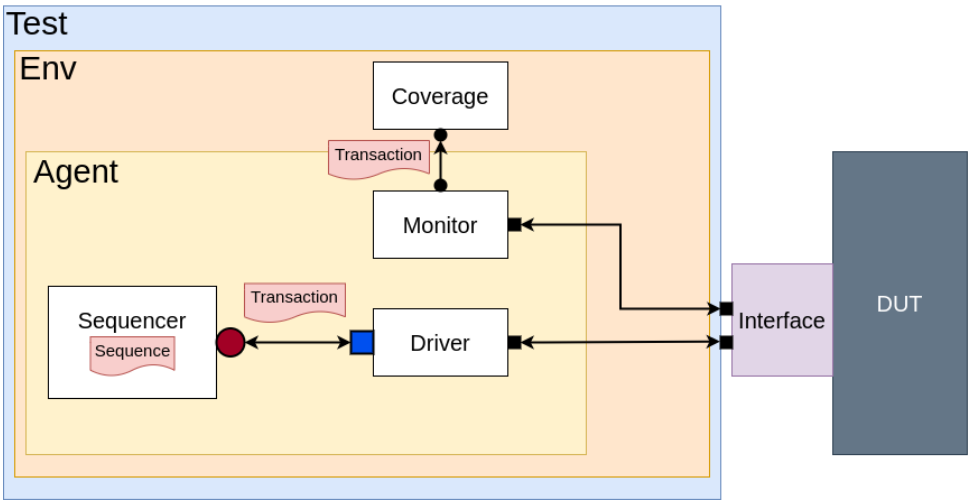


Figure 2.4: UVM Testbench Overview

main UVM Components that are almost essential to building a UVM testbench. These are:

- Test: It contains all the components of the UVM environment. In essence, the test is a wrapper for the environment. A UVM testbench could have more than one, being able to switch between very different environments by just instantiating one or another test.
- Environment: Instantiated inside the UVM Test, the environment is where all the interacting UVM Components are instantiated and connected. For example, the different communicating components are connected inside it using communication ports.
- Agent: The agent contains all the UVM Components that generate or receive values from the DUT. These are the following:
  - Sequencer: Is in charge of obtaining the transactions from the Sequences and sending them to the driver through the ports.
  - Driver: Requests for data transactions to the Sequencer and stimulates the DUT with the corresponding values after receiving them.
  - Monitor: Observes the DUT interface or values and creates transactions. These could be incoming or outgoing values from the DUT and will later be used to react or create different driver or sequence values.
- Coverage: Additionally, UVM can be used to record coverage numbers. Coverage, as will be explained in further sections, is used in design verification to track how many possible values have been observed in the environment, both as inputs and outputs of the DUT.

The UVM Test is run from the UVM Top module. This top module also instantiates the *test harness*, which is the entity containing the DUT connected to a SystemVerilog interface.

This interface is then declared in the UVM database, a utility of UVM in which one can declare objects to be used and communicate between different points of the testbench. Using this UVM database, this *virtual* interface can be accessed in the Driver and the Monitor, amongst others, as seen in Figure 2.4.

As mentioned before, one could create a generic UVM Driver class with the main methods and attributes and extend it to a custom class adapted to the DUT connected to and with the type of transactions required. This makes it faster to build whole environments for multiple modules or projects. There are tools such as Easier UVM [13] that provide empty testbenches adapted to the DUT that the user specifies.

A typical use case is creating a base test class and then extending it to other child classes that can be switched easily in the testbench, stimulating the DUT very differently by just instantiating one or another UVM test. These tests could be, for instance, a random one in which all components work randomly or a directed one, where specific sequences produce controlled stimulus. These different tests could also use the same UVM Components but change the generation of values.

2.3 European Processor Initiative

In recent years, governments and other entities have been investing in different research areas, one of them being microprocessor design and production. This is the case of the European Processor Initiative (EPI). In this project, funded by the European Commission, many research entities and companies collaborate to design and build a new family of European low-power and high-performance processors for various applications like supercomputers, Big Data, automotive or Machine Learning. These processors will use the open-source *Instruction Set Architecture* (*ISA*) RISC-V, explained in Section 2.4. This and more details will be further discussed in the following sections, but the fact that it is an open-source ISA favours the proliferation of these projects. Without setting these correctly, vector instructions may be executed differently depending on the architecture and the code executed. If used well, this is one of the most attractive features of the RISC-V vector extension [23]. Many entities cooperate to take this project to terms and each of them has its tasks and responsibilities. For example, the BSC is mainly in charge of producing a Vector Accelerator that implements the recent vector extension of the RISC-V ISA. Similarly, other companies or entities design scalar cores, floating-point units and other accelerators. In the end, all partners combine their efforts and collaborate to integrate the different units.

Considering the costs of producing a design and how hard it is for a research project to start and be funded, verification takes an even more significant part and relevance. Therefore, all partners must deliver their units thoroughly verified and a narrow margin can be given to them to fail post-integration. If some of the units failed or were deficient, it would be a catastrophe and it would probably be preferable to leave it out of the final tape out. An integration failure would be a considerable drawback for the project and the responsible entity.

The module itself and the interface with the rest of the design must be verified. The interconnections of the module are one of the crucial and most essential parts of the unit. They must be a collaborative effort with the corresponding colliding partners to ensure

the perfect functioning of the interfaces. In this document, we will focus on the interface verification of the Vector Accelerator, among others, which involves the connection to a scalar core.

2.4 RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) [22] based on reduced instruction set computer (RISC) principles. As said, the main difference between RISC-V and other ISA is its open-source license, which means that using it for producing a design does not require paying a fee.

Since its appearance in *University of California Berkeley* in the year 2010, it has been used in several projects to produce open-source designs. Being open-source is especially interesting for research and academic entities, which cannot sometimes afford to pay for other ISAs. With already ten years of history, the RISC-V ISA is at its best moment and has become a popular choice even for big companies when looking for an ISA to produce their designs. Early 2021, Huawei has announced the development of its first RISC-V design, the HiSilicon Hi3861 development board [19]. Giant e-commerce company Alibaba has also recently announced that its Xuantie 910 processor aimed at cloud servers would be using a RISC-V core [46]. Previous to that, storage drives companies Western Digital and Seagate had announced that they would be producing RISC-V processors to control storage and security of data [8].

RISC-V has established itself as a modular ISA, which means that it has a base specification to be implemented and many optional extensions. The base specifications are called *rv32* and *rv64* depending on the size of registers and buses in the design.

Among the ratified RISC-V extensions [21] that can be implemented we find the following:

- Multiplication (M): which contains the scalar multiplication and division instructions.
- Atomic (A): which contains the atomic memory operations, used to operate safely on values inside memory when communication between different cores is required.
- Floating Point (F): specifies the single-precision floating-point operations.
- Double Precision Floating Point (D): which specifies the double-precision floating-point operations
- Quad Precision Floating Point (Q): which specifies the quad-precision floating-point operations
- Compressed (C): contains the compressed set of instructions, encoded in 16 bits.
- Vector (V): contains the specifications for vector-processing operations and architecture-specific details.

Most of these will be implemented in the EPI project. However, we will focus on the Vector Extension, which is the part of the RISC-V ISA that the Vector Accelerator developed by

the Barcelona Supercomputing Center is implementing. More details on the RISC-V ISA can be found in their website and their specification documents [5] [6].

2.4.1 RISC-V Vector Extension

Vectors could be described as elements of the same type arranged in the same structure. Typically these are used to operate on big groups of data to which one has to perform the same operation. For example, a vector ADD immediate instruction can be used to add the same immediate value to a vector. This type of instruction is typically used while operating with media like video and audio. Nowadays, most CPU designs contain Vector Accelerators that are exclusively in charge of implementing vector instructions.

Vector processors allow operating on multiple data with one instruction because operations ensure that there are no dependencies within a vector [34]. Vector instructions reduce fetch bandwidth requirements as each of them generates multiple operations. Additionally, these operations follow a very regular execution pattern, which eliminates unnecessary delays in the execution and allows removing explicit code loops, which means fewer branches in the execution and potentially fewer delays.

This type of processor is typically divided into lanes. Each lane contains a set of elements of the vector and one or more functional units, which are used to operate on the elements. This way, multiple elements of the vector can be operated on simultaneously, one in each lane. The elements are stored in vector registers, often interleaved in the different lanes. When a vector operation starts, the lane retrieves elements successively from the vector register and performs the execution pipeline for the instruction for each of them until it potentially saves the resulting element in the vector register again.

Classic Vector architectures required the programmer or compiler to make data structures in the code fit the size of the structure in the hardware. This is one of the disadvantages of standard Vector processors, where they would only work efficiently if parallelism is regular.

In addition, in typical Vector or SIMD extensions, a change in the size of the elements in the code forces a change in the instruction set to expand the vector registers (in the case of x86, from 64-bit MMX registers to 128-bit Streaming SIMD Extensions (SSE), to 256-bit Advanced Vector Extensions (AVX), and AVX-512 [11]). The result is a growing instruction set and a need to port previously working code to the new instructions.

In the RISC-V Vector extension, rather than fixing the vector length in the architecture, the *vsetvl* instruction can be used to vary both the vector length and element width of the registers. This way, operations are agnostic to architecture and implementations. This approach makes it easier to be compatible with a broader range of vector processing units.

The vector extension introduces 32 new vector registers, which are used in the vector units to do their operations. In addition, it presents new *Control Status Registers* (CSR) that are used to configure the execution of vector instructions. The most important ones are:

- Vector length (VL): which indicates vector elements used in each vector instruction.
- Vector Type (vtype): which contains other configuration fields such as:

  - Standard Element Width (vsew): specifies the length of the elements in the vector. It can take values corresponding to 8, 16, 32, 64 and 128 bits.

  - Vector register group multiplier (vlmul): specifies the number of vectors to operate with for each instruction. It can take values corresponding to 1, 2, 4 and 8.

- Vector start position (vstart): indicates the element from which the operation must be performed on the vector register.

There exist more CSRs than the previous ones that control how particular instructions are executed and other relevant features of the vector extension. Without setting these correctly, vector instructions may be executed differently depending on the architecture and code. If used well, this is one of the most exciting features of the RISC-V vector extension.

At the start of the EPI project, the vector extension was in a pre-release state, with version 0.7.1. Therefore, the partners decided to stick to it until the end of the project and the Vector Accelerator implemented it [23]. In the later stages of the project, version 1.0 [24] of the vector extension was released, which the Vector Accelerator will use in the future.

2.5 Related Work

When looking at previous work in design verification, there is not much disclosure for the verification of decoupled accelerators. Big companies tend to be hermetic and do not detail their work. They use UVM, formal techniques and continuous integration pipelines, but they do not release any code or documentation. This secrecy makes it hard for newcomers to learn the "industry standard way to verify designs".

Therefore, we must rely on the few design verification efforts done in open source hardware projects, where there is not much consensus. Every team interprets the verification process differently, coming up with ideas and flows that put together the techniques that must be used. However, these teams often publish articles and write documentation on how they worked and what they used or not during the verification process. In this sense, we have examples like the Opentitan [32] project from lowRISC [30], the OpenHW Group projects [18], and the rocket-chip [9] and Syntacore SCR1 [44] projects. All these are open source projects that aim to produce a RISC-V core (among other designs).

Opentitan is a massive project with exhaustive documentation. They have an entire site with information related to the project, but they also explain how they generate this information and provide pointers to all the tools they used. Additionally, this site contains a dedicated verification section with clear and detailed explanations of their verification process features, from which we took inspiration.

In Opentitan they developed a whole integrated chip containing, among other IPs, the RISC-V Ibex [29] core. For the whole design, they developed multiple verification tools. They created testbenches, test plans for their IPs and multiple scripts that automatically create from UVM testbenches to register models to verify the RISC-V CSRs.

They targeted the design in three ways; IP Level, Core level and Chip level, in which they verify all the IPs integrated with the core. For each of these levels, they developed dedi-

cated UVM testbenches. Their core testbench, for example, runs binaries in co-simulation with an Instruction Set Simulator (ISS), to which then they compare the execution trace logs to determine whether the DUT functionality was correct or not. The rest are very IP or module-specific testbenches that use features that range from UVM to memory models, depending on the DUT. The core level testbench resembles in many aspects, like ISS co-simulation, what we have done with the Vector Accelerator and it is much more relevant for this work. However, the IP or module level demands a large amount of resources as it involves creating a testbench for each submodule of the Accelerator, which we could not afford due to lack of time and people.

In addition to all the previous points, they implemented functional coverage and assertions for their modules. All this is explained in the design verification methodology page of the Opentitan site [28].

The other big source of data is the OpenHW Group. This is an organization composed of multiple contributors where hardware and software designers collaborate to develop open-source cores, related IP, tools and software. They are making considerable efforts to open-source as much information and code as possible to help create open-source IPs. The Barcelona Supercomputing Center is a member of the group. In their own words: "OpenHW provides an infrastructure for hosting high-quality open-source HW developments in line with industry best practices" [18].

They have multiple parallel projects, but it is of particular interest for this thesis the verification efforts performed on the CORE-V family of RISC-V cores. These are a collection of open-source RISC-V cores (such as RI5CY [36] or Ariane [27]) to which they execute industrial grade verification. All the code is in the same Github repository [18], which to begin with, makes it very handy to access their resources. Then, inside each core folder, the structure is clear; they have separate folders for the modules related to the DUT and the modules related to the testbenches.

For each core, they have a massive UVM testbench that contains assertions and functional coverage. In these testbenches, they generate the instructions that the core will be executing and check the results that it produced comparing to ovpsim [20], an Instruction Set Simulator developed by Imperas. All these are features that we have also adopted in our testbench, from co-simulation with an ISS, as explained, to checking specific properties with assertions and counting coverage, so these are valuable references for the work explained in this thesis.

Another interesting feature is the RISC-V Formal Interface [43]. This is an interface that can be potentially connected to any RISC-V scalar core, using certain signals that characterize the execution of an instruction. The model will then determine its result after detecting which instruction is being executed and compare it with the one in the core. This can speed up the initial stages of the verification process of many designs, as the engineers must only find the necessary signals and connect the formal interface to these. All this and more information is provided in the CV32E40X User Manual [12] and the CORE-V Verification Strategy [33] pages.

Finally, the rocket-chip and SCR1 projects, even though they have simulation environments, have more insufficient verification documentation. We may only find pointers to external repositories with tests used to stimulate the core and how to run them, but no

actual description of their environments is provided. In both cases, their repositories only contain testbenches on which they instantiate the core and provide instructions to it reading from a memory that has previously been loaded with the binary to execute. It is worth noting that these projects are usually made up of different partners and contributors so that this documentation may be distributed between the separate repositories.

As seen, what all these projects have in common is that they provide a complex simulation environment for their designs. These execute a test and stimulate the design to collect the results later and check their correctness.

The Barcelona Supercomputing Center had previously produced a few designs, but no similar environment was developed. The previous project was the preDRAC [4] tape out, a RISC-V 5-stage in-order core. This design was tested using random binaries generated with a RISC-V random binary generator, riscv-torture [37], loaded in a RAM. At the end of these tests, contents of the registers were dumped into this RAM using store instructions and then compared with a reference model. Additionally, in the later stages of the project, this tool was improved and extended to provide more information to the engineers, displaying the values resulting from each independent instruction and making it easy to debug possible errors in the design.

Even if this is a clever way of building an environment and was good enough to find many bugs in the design, it is not the standard way of implementing a verification environment. UVM is typically in the centre of these, and randomness is a critical factor for the proper verification of a design. It is worth noting that no dedicated verification team existed in the BSC at the time, which means that both the RTL design and the testing environment were developed simultaneously and by the same engineers for the preDRAC tapeout. Nevertheless, this vast effort resulted in a mostly successful tape out. Still, it showed the need for a dedicated verification team, which was formed shortly after the design was sent for production.

Chapter 3

EPAC Architecture and DV Infrastructure

This chapter will focus on the closest background, the design under test and a summary of the full verification environment. In Section 3.1 we will describe the EPAC Architecture. Then, the Vector Accelerator interface, OVI, and its architecture, will be explained in Sections 3.2 and 3.3, respectively. Finally, we will have a short description of the whole verification environment for the Vector Accelerator that we developed in Section 3.4.

3.1 EPAC Architecture

As said in Section 2.3, the EPI project comprises multiple partners, each with its module or purpose. The resulting chip of the project is called EPAC (European Processor Accelerators). This chip contains four vector-processing micro-tiles composed of an Avispado RISC-V core, designed by SemiDynamics, and a vector processing unit designed by the Barcelona Supercomputing Center and the University of Zagreb.

Each tile also contains a Home Node and L2 cache, designed by Chalmers and FORTH. In addition, the chip contains a Stencil and Tensor accelerator (STX), designed by Fraunhofer IIS, ITWM and ETH Zürich, and a variable precision processor (VRP) designed by CEA LIST. These tiles and accelerators are connected through a high-speed network on chip (NOC) and SERDES technology from EXTOLL.

The EPAC chip integration in GLOBALFOUNDRIES 22FDX low-power technology is led by Fraunhofer IIS. The architecture above can be seen in Figure 3.1, in which we can see all the accelerators and tiles along with the NOC. Additionally, the EPAC is integrated and evaluated in the FPGA-based board designed by FORTH, E4 and the University of Zagreb.

In around two years, the different partners have developed and verified this whole structure. In the second half of 2021, the chip has been taped out, marking the end of the project's first phase. This tapeout took the name of EPAC 1.0 [1] and in Figure 3.2 we can see the layout with 25 mm2 in GF 22FDX technology. The figure shows the area distribution between the tiles and the rest of the accelerators.

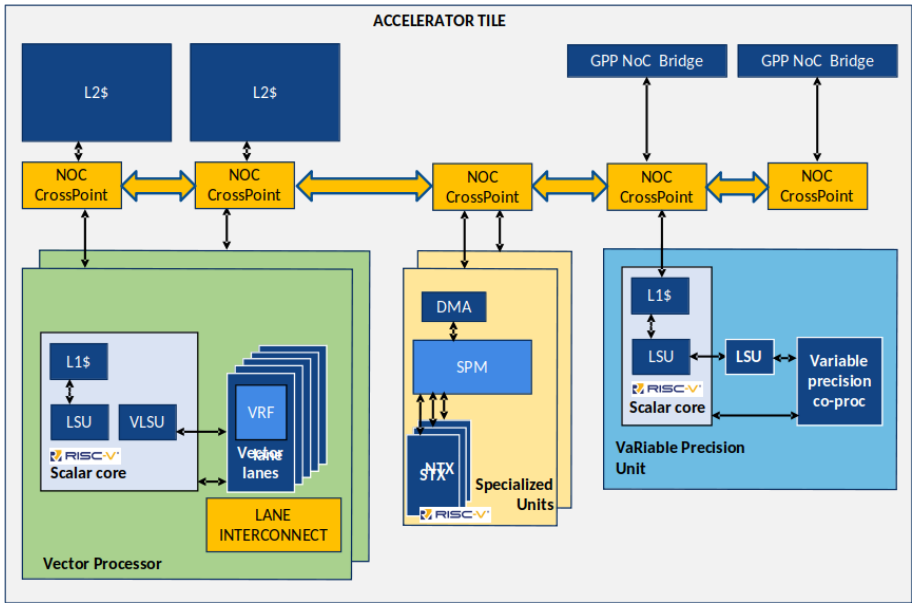


Figure 3.1: EPAC Architecture Diagram [35]

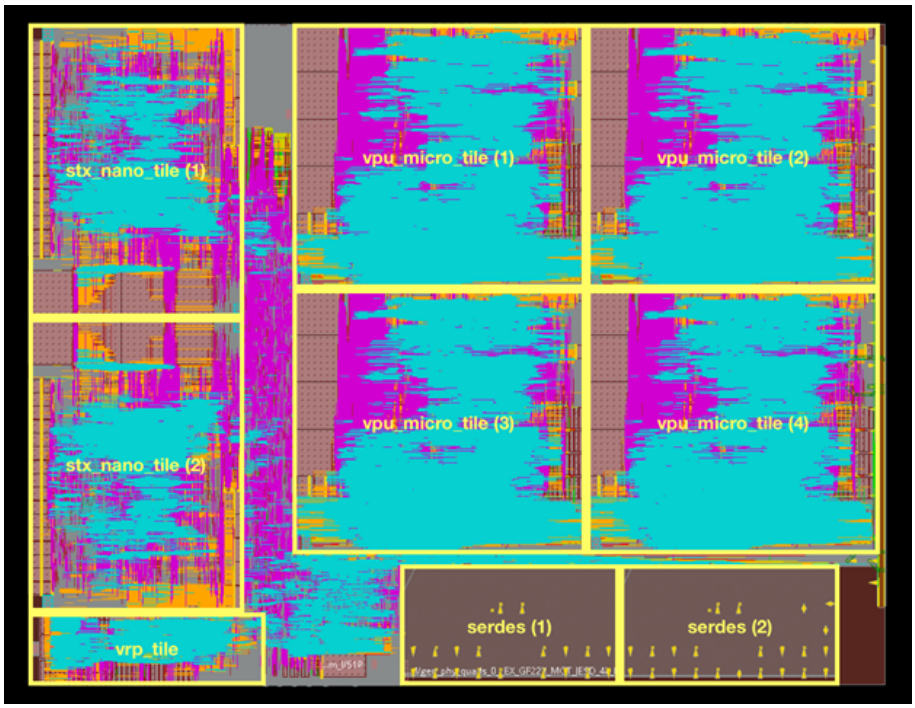


Figure 3.2: EPAC first tape out [1]

These tiles are the ones composed, as seen in previous images, of one Avispado core and one Vector Accelerator. We developed the accelerator in the Barcelona Supercomputing Center, while Semidynamics developed the Avispado RISC-V scalar core. As well as in the rest of the design, communication between two modules developed by different partners was vital for success and one of the critical parts of the chip. To communicate these two IPs (intellectual properties), the different partners decided to use Open Vector Interface (OVI).

3.2 Open Vector Interface

This interface was released and open-sourced by Semidynamics [41] to connect Vector Units to their cores. OVI eases the collaboration between the entities and helps focus only on the computation capabilities of the module one is developing. This is achieved by providing a specified interface to which the module must adapt instead of coming up with it. In its Github repository, one may find the specifications document for this interface [40].

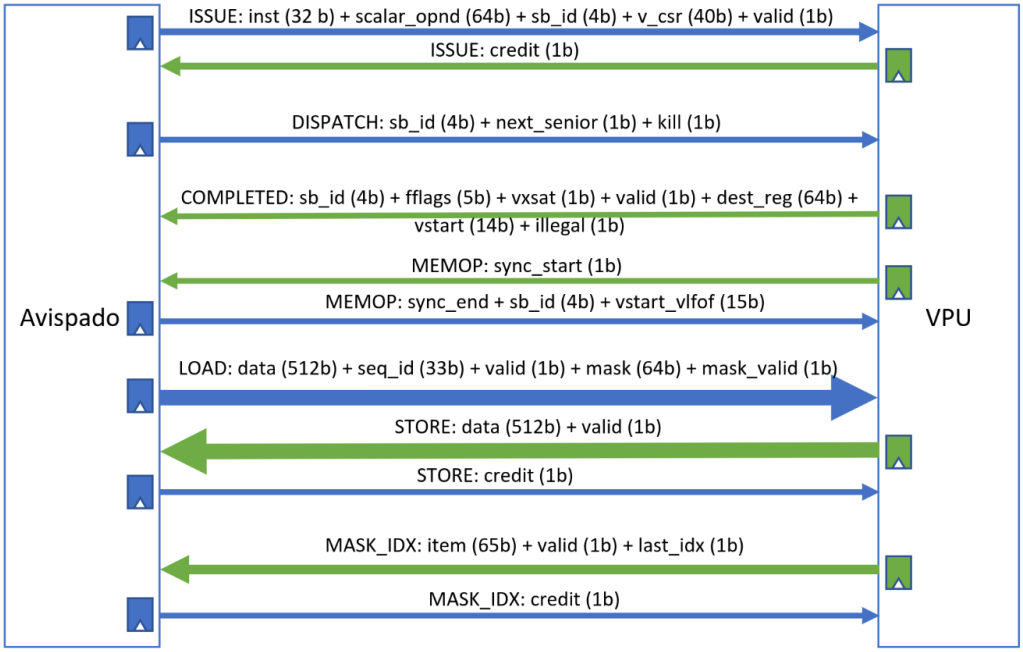


Figure 3.3: Open Vector Interface buses description

As seen in Figure 3.3, OVI is divided into multiple sub-interfaces. These are the Issue, Dispatch, Completed, Memop, Load, Store and Mask Idx sub-interfaces. Together, they send instructions and meta-data to execute them to the Vector Unit connected to the core. Each sub-interface is in charge of the following:

- Issue: Through this sub-interface, the core sends the Vector Accelerator the instructions and the necessary data for them. This sub-interface also has a credit system through which the Vector Unit controls how many instructions can be *issued*. Once the Vector Unit has treated (not necessarily executed) the incoming instructions, it will return a credit. If the core sends instructions non-stop without receiving a credit, it will have to stop sending them once it runs out of credits.

- Dispatch: All instructions must be *seniored* to finish execution or *killed* through this sub-interface. The Dispatch sub-interface is necessary due to the possibility of having an out-of-order core connected, which may discard already *issued* instructions. In addition, memory exceptions can cause instructions to fail execution and force the core to *kill* the following instructions.

- Completed: When the Vector Unit finishes the execution of an instruction, it will notify the core using the signals inside this bus. Along with this notification, the

buses may contain results and other meta-data.

- Memop: Previously *issued* memory instructions will start and finish their operation through the interface using this sub-interface.
- Load: If a load memory operation starts, the core will send the corresponding data and meta-data using this sub-interface. This meta-data includes a mask and a bus called *seq id* that specifies where the valid elements can be found inside the data.
- Store: If a store memory operation starts, the Vector Unit will use this sub-interface to send the corresponding data. This sub-interface uses a similar credit system to the one used by the Issue one, but it is the core now providing the credits instead of the Vector Unit.
- Mask Idx: If a masked memory operation starts, the Vector Unit will send the masks using this sub-interface. Indexed memory operations, usually called *scatter* and *gather*, also make use of the buses inside this group. The Mask Idx sub-interface also uses credits the same way as the Store one.

Not all cases were explained in the previous list but will be run down in depth in the following sections as these complicate the verification environment implementation.

As seen, many of these sub-interfaces may be used together to execute only one instruction. For example, if we wanted to execute a masked vector load, we would *issue* the load instruction through the Issue sub-interface and then confirm its execution using the Dispatch sub-interface. Afterwards, we would start and finish its memory operation through the Memop one, send the masks using the Mask Idx one, send the memory data through the Load one and communicate its ending through the Completed sub-interface.

Therefore, all these sub-interfaces are connected and must be used appropriately. Although the whole verification environment and process will be described, this work focuses mainly on the stimulus and verification of the OVI interface, because as can be seen, it carries many difficulties.

3.3 Vector Accelerator Architecture

In our case, the Vector Accelerator [15] is our design under test (DUT). This work and the whole verification process concerns this Vector Processing Unit, which we will call VPU from this point on. In the EPAC architecture, each tile connected to the NOC has one VPU exclusively dedicated to executing vector instructions.

Our VPU supports 64 and 32-bit floating-point vector operations and 64, 32, 16 and 8-bit integer vector operations. It has 32 logical and 40 physical vector registers. With these, the VPU can handle vectors up to a maximum vector length of 256 elements of 64 bits each (16Kb in total). The VPU is connected to the scalar core through the OVI, as explained in Sections 3.1 and 3.2. The module that first receives the instructions, connected to the issue sub-interface, is the Pre Issue Queue, which controls the credit system of the sub-interface.

Once the instructions can be executed and after being decoded, these will be sent to the Store or Load Management Unit or the Vector Instructions Queue. The first two control

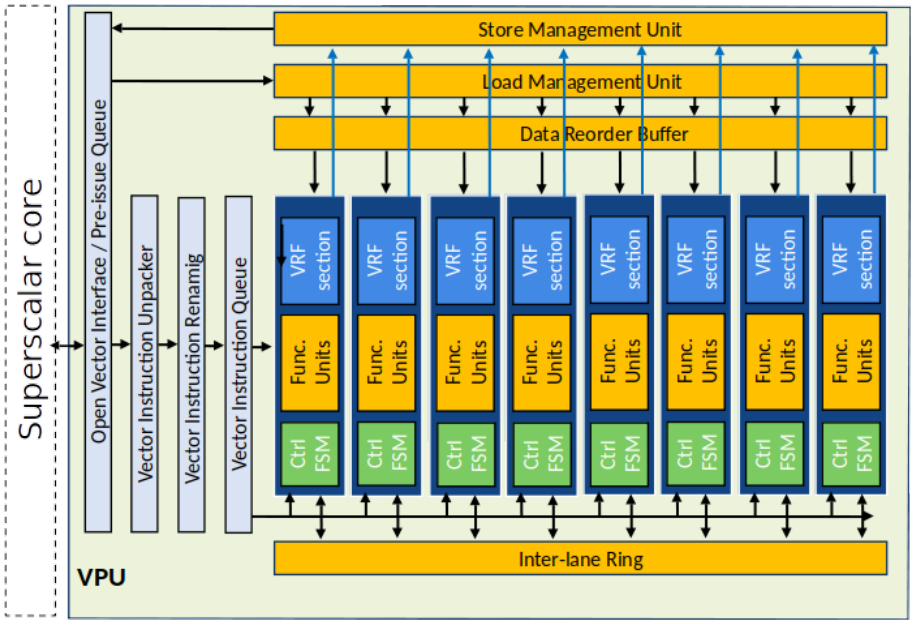


Figure 3.4: VPU Overview

the execution of memory instructions, while the Vector Instructions Queue provides the arithmetic ones to the Vector Lanes.

As seen in Figure 3.4, the VPU is composed of eight vector lanes. A vector lane is mainly in charge of processing vector arithmetic instructions. Each lane also has access and a part of the vector register file, used in almost every vector instruction. These lanes are interconnected thanks to the Inter-lane Ring.

Dividing the execution units into different vector lanes allows the operations to be performed concurrently. This parallelization is achieved by slicing the vector register file, which means that each lane contains only a portion of the vectors. Figure 3.5 shows the distribution of the elements of a vector in each slice of the vector register file.

In the figure, we can see the eight lanes. In every lane there are five banks with 256 registers of 64 bits each. The mapping of vector register elements is shown in the image. For the first vector (v0), element 0 is in lane 0; element 1 is in lane one and so on. This increases the throughput, as all lanes could be operating concurrently for the same vector to compute a vector instruction.

In addition, this parallel capability is also exploited in writing in the registers. It perfectly suits the data buses of the load and store sub-interfaces from OVI, which are 512 bits wide, meaning that from each transaction with the scalar core, potentially all the elements sent or received can be treated at once.

As seen in Figure 3.4, memory operations are treated in the corresponding Load or Store Management unit. Load memory operations have a limited out-of-order capability, as the scalar core could send some elements sooner than it should. This happens, for example, when it has a longer memory latency to access a previous element. This particularity of OVI is supported and handled in the VPU. Therefore, simulating all the possible OVI inter-

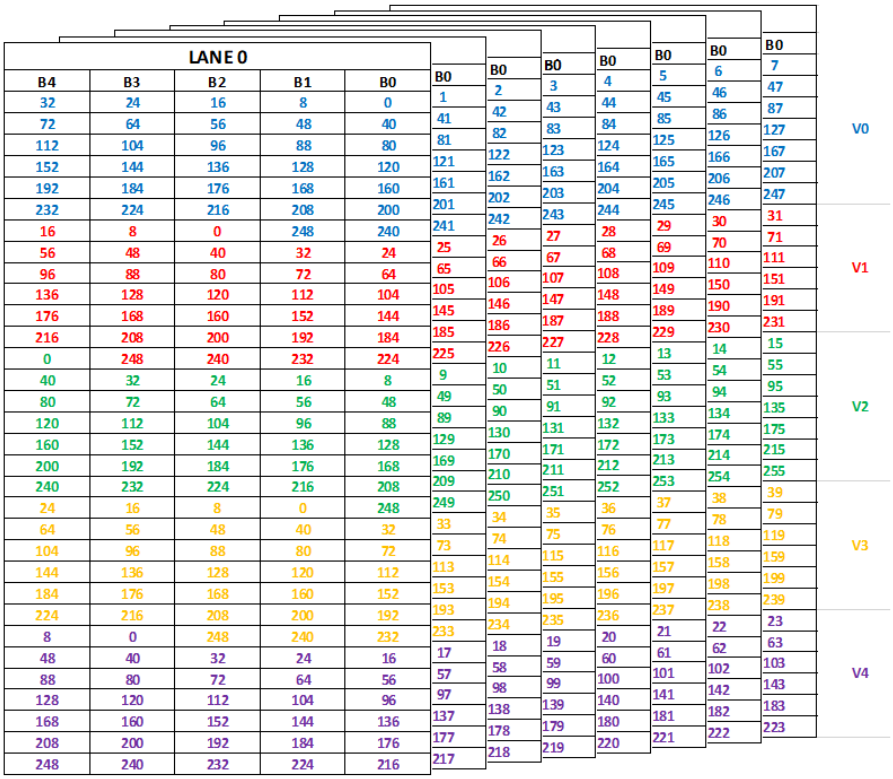


Figure 3.5: Vector Register File diagram [15]

face behaviours along with these semi-random events complicate the verification process of the VPU.

3.4 Verification environment

We first thought we could do block-level verification when planning the verification environment. That is, creating a verification environment for each module of the VPU. We came up with this idea from a mixture of the little information we had from the industry. However, big companies have big verification teams and the resources to take these projects to terms. In our case, we were four very junior verification engineers with no experience who had just learned what UVM was, so we soon saw that this strategy was unfeasible. In any case, developing the environments for some of these modules gave us the chance to make our first steps in UVM.

In the end, though, we decided to go directly for a full big UVM testbench for the whole design. Although it seemed pretty scary initially, this type of environment would help us verify the whole design at once, which would have taken months with the other strategy. We needed to emulate the OVI interface, such as the scalar core, to stimulate the whole design. For that purpose, as we had multiple "independent" sub-interfaces, we thought it would be interesting to have a sub-interface divided UVM testbench. These divided agents could all extend from a base agent class, making its development more manageable.

The UVM testbench and all of its features will be explained in detail in Chapters 4 and 5,

as this work mainly focuses on this part of the verification process.

3.4.1 Riscv-dv

Even if we planned on producing random stimulus, generating a valid instruction for the VPU would mean creating a complex set of constraints for the transaction. To solve this issue, we decided to go with an instruction generator.

After some search, we found a RISC-V instruction generator that supported the generation of vector instructions, Riscv-dv [16]. Riscv-dv is a System Verilog and UVM based open-source random instruction generator developed by Google.

However, we needed to perform several additions to the tool to use the generator. Among these, we find:

- Generation of *vsetvl* instructions through the code. The vsetvl instructions are used to change the element width and vector length of the registers during the execution of a binary. In addition to the prior, modifications in the generation of memory operations were done to allow the change of element width and vector length.
- An option was added to select the initialization pattern of the data pages of the generated binary.
- Constraining the memory addresses accessed by the binary to avoid memory exceptions, especially for vector memory indexed instructions, where some addresses could cause page or access fault exceptions.
- Adapting the generator to the 0.7.1 RISC-V Vector specifications.

Additionally, we had to blacklist different instructions in some stages of the project. This was either due to the VPU not supporting some instructions, because some modules were still under development, or because specific features needed to be tested and we needed a cleaner test.

The tool comes with a script to make it easier to execute. It offers capabilities like generating tests for different target architectures or generating multiple tests in only one command. When we run the generator, we obtain an assembly code, which becomes a binary after being compiled. This binary contains different random instructions, which we can feed to the VPU using the UVM and other environment features explained in the following sections.

3.4.2 Spike ISS

With the features explained so far, we have the stimulus of the DUT and its generation. However, we need to check that the responses of the VPU are correct. This is often done in design verification using a reference model, which can be developed specifically for the project or borrowed from previous ones.

In addition to this, while we were developing the environment, we thought it would be more interesting to generate scalar instructions with Riscv-dv to simulate better the actual case with the scalar core. The VPU would be connected to a core that would only send

the vector instructions while executing the scalar ones in the real environment.

Although this might seem inherent to the VPU, scalar instructions may modify values used in vector instructions. For instance, we have scalar to vector instructions, where a value in a scalar register is used for modifying vector registers, and memory instructions, where the memory space is shared between the scalar core and the VPU.

If we were not executing scalar instructions between the vector ones, the source values for the instructions mentioned above would always be the same, leading to possible missing cases or straight unreal situations.

For these two purposes, executing scalar instructions (acting as scalar core in general) and as a reference model for vector instructions, we decided that we needed an Instruction Set Simulator (ISS). These tools can execute binaries of the corresponding ISA and act as a design capable of interpreting their instructions. In the Barcelona Supercomputing Center, we have had previous experience with Spike [25], a RISC-V ISS, and we knew it satisfied our needs, so we decided to go with it.

However, we needed to perform some modifications and additions to use Spike as we wanted. These are the following:

- To call Spike from System Verilog, we defined functions that use Direct Programming Interface (DPI). This is necessary because Spike had to be compiled as a C++ library to be included. The main DPI functions are:
  - Method that resumes the Spike simulation until a vector instruction is found. The instruction is then executed and the reference results are returned to the UVM. The instruction is issued in the VPU and when it completes, the DUT results are compared against the Spike ones.
  - Function to set the result of specific instructions into Spike. This was necessary to avoid execution divergence in unordered floating-point reductions, as the VPU method for performing this operation was different from the Spike one, while both being correct. Even if we found a way to check that both results were correct, Spike would have a different value in its registers, which may later be used in another instruction, resulting in a mismatch. To avoid this, we inserted the result from the VPU into Spike.
  - Functions to read from Spike memory, used in vector load instructions to provide the same data as Spike.

All the methods above allow Spike to act as the scalar core, providing instructions and the tools to emulate a real environment. However, the version of Spike that we had at the time was not the exact ISA version that we needed, which was the 0.7.1 of the RISC-V Vector Specification.

To tackle this issue, we had to introduce some modifications in the actual Spike implementation, which are detailed below:

- Change in the implementation of the vector tail zeroing, replaced by a different policy after version 0.7.1.

- Change in the decoding of some instructions to follow the 0.7.1 specification.
- Change in the requirements of the Vector Context Status (VCS) fields in the *mstatus* Control Status Registers (CSRs).

Once Spike fitted our needs perfectly, it allowed the execution of all kinds of instructions between Spike and the VPU. In addition, we could obtain the results of the vector instructions from both sources and compare them to assert that the VPU worked as expected, as will be explained in Subsection 3.4.3.

3.4.3 UVM Scoreboard

The UVM Scoreboard is an optional UVM Component commonly used to ensure that the DUT works correctly. It is often composed of two main parts, the predictor and the comparator. The first obtains the expected outputs from the DUT resulting from the inputs it receives: a reference model. Usually, this reference model is written in a different, higher-level language than the DUT to make sure the implementation is different enough and correct to be used as a reference. On the other hand, the comparator takes the results from the predictor and the DUT and compares them to decide whether the DUT worked as expected or not.

In summary, the Scoreboard receives the same inputs as the DUT and compares the results it produces to those that the DUT produced. The UVM Scoreboard details can be seen in Figure 3.6.

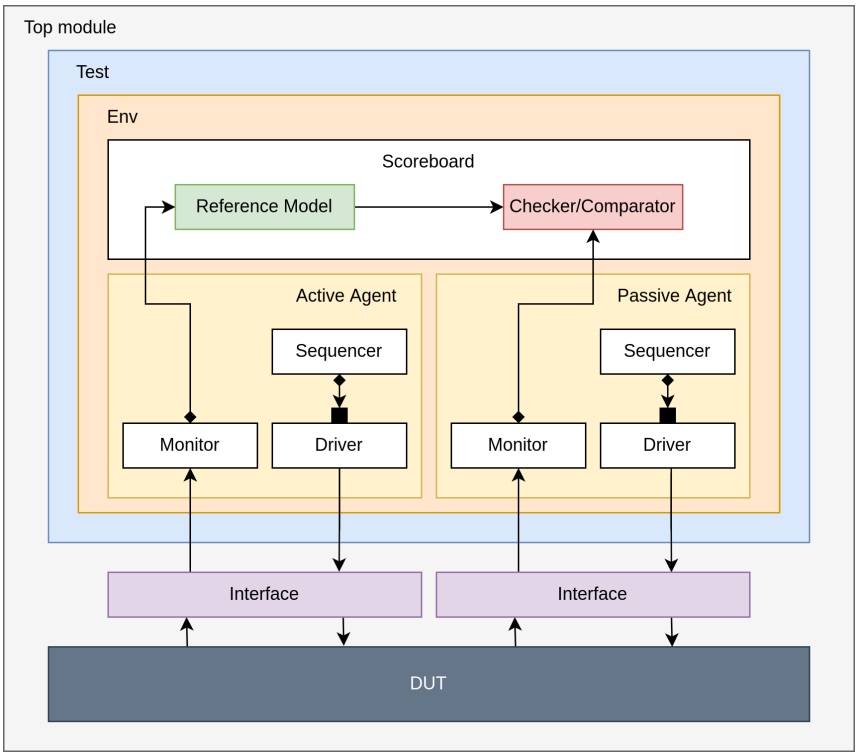


Figure 3.6: Typical UVM Scoreboard structure and connections, based on [47]

This figure shows that the UVM Scoreboard is typically instantiated in the UVM Environment component. We can see that the image has two agents, one active and one passive.

These stimulate and observe the DUT, respectively.

The active agent also contains a monitor in charge of recording the interface's state when new stimuli are being sent. This is how the Scoreboard's predictor gets the input values, in this case, the "Reference Model" box in the figure. The reference model will generate the corresponding expected outputs from the DUT and send them to the "Checker/Comparator" component.

On the other hand, the passive agent is in charge of recording the output values of the DUT through its monitor. For this, the whole environment must be aware of the delay of the interaction with the DUT because otherwise, it would capture incorrect values. Afterwards, the monitor sends the transaction to the "Checker/Comparator" component in the Scoreboard, where the predictor results are waiting.

Once both values are in the comparator part of the Scoreboard, this will determine whether the DUT responded well to the stimulus or not. After this comparison is done, the Scoreboard might use the *uvm_info* to display some information about it and *uvm_warning* or *uvm_error* to warn that something went wrong. Additionally, *uvm_fatal* might be used to terminate the simulation instantly and show a message indicating what happened.

In the case of our verification environment, we decided to have a scoreboard for the results of the vector instructions. For this, we have a queue where issued instructions are stored as they come from Spike, with results and their sources. This works at the same time as the monitor before the predictor and the predictor itself from Figure 3.6 because it straight contains the results. Once one of these instructions completes, the monitor in the completed interface, which will be detailed in Chapter 4, will communicate to the Scoreboard that it can start its comparison.

For this, we could not use the typical scheme just described, as we did not have a monitor for the vector registers, only for the interface. We decided to create a different virtual interface to extract the values from the vector registers to face this issue. This is set up in the test harness and has two main issues:

- It has to keep track of the actual destination vector register. Due to the VPU having register renaming, the physical vector register where the actual values are may not match the one specified in the instruction. To solve this issue, we maintained discussions with the RTL team and found a signal that pointed to the corresponding physical register when the instruction was completed. We decided to send this signal with the contents of all the physical registers to the scoreboard to select the correct register.

- It has to solve the slicing of the vector registers. As explained in Section 3.3, the vector registers are sliced among the vector lanes of the VPU, making it harder to read the contents of a vector sequentially. For this, once we have the physical register to read and its contents, we call a function that "unslices" the vector from the registers. In the later stages of the project, the RTL team provided an already ordered structure that contained the vector registers, easing the implementation of the scoreboard.

After the scoreboard receives the instruction completion signal and the content of the

vector registers, it can compare them to the results from Spike and determine whether the VPU did right or wrong. If this last is the case, the Scoreboard calls the UVM particular method *uvm_fatal* described before and the simulation ends. If the results match, the test goes on and checking further instructions.

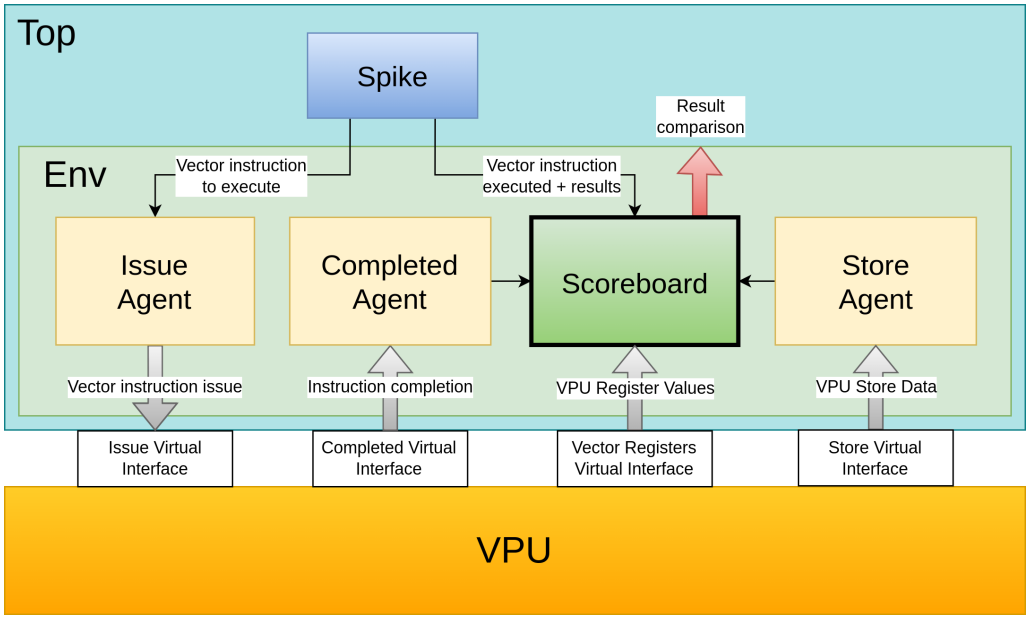


Figure 3.7: Scoreboard structure and connections for the VPU

Additionally, to check if the VPU correctly executed vector store instructions, we must compare what Spike stored in its memory to what the VPU has sent through its Store sub-interface. For this, when the instruction is retrieved from Spike, we go through all the memory addresses that the instruction changes and collect their values. This data gets to the Scoreboard together with the rest of the instruction information and is compared to the VPU data. This process can be observed in Figure 3.7, along with the rest of the components and connections mentioned before.

3.4.4 Assertions

Another design verification tool that is commonly used is Assertions. These are used to check that some behaviours and situations happen as they should. In the case of our project, we decided to use System Verilog Assertions [3].

Assertions are typically done by the RTL team and written along with the RTL code. These are inserted in each developed module and control small features of it. However, due to the lack of time and resources, the RTL team dedicated most of their spare time writing the design specifications.

Because of this, we created a list of assertions to check the VPU and our environment's behaviour at OVI. We decided to go this way because we had a very detailed specification document for the OVI by the time we started. Thus, we could simultaneously develop our implementation of OVI and these assertions to check that it is working as expected.

All our assertions are included inside a checker module connected to the DUT interface and instantiated in the test harness. In Chapters 4 and 5, some example assertions imple-

mented in the project will be detailed.

3.4.5 Coverage

Coverage is one of the most important verification features. It measures how well or how much of the design has been tested. How many of the possible cases that the DUT can handle have been tried out. There are two types of coverage:

- Functional coverage: Must be implemented by the verification team and has to follow the RTL design specifications. In System Verilog Coverage [45]; *covergroups*, *coverpoints* and bins have to be written to record whether certain values have been explored for the DUT signals.

- Code coverage: It is recorded by the simulation tool. The tool records information like if a function has been called or not, whether source code statements have been executed, and how condition statements come out. This is useful for detecting dead parts of the code and identifying a wrong condition in a conditional statement, apart from detecting non-tested parts of the design.

Through coverage, we know if our environment is stimulating properly the DUT or not. At the same time, the RTL team can see if certain conditions are not being passed or if some parts of the code are never being executed. Thus, coverage is a powerful tool that helps the RTL and design verification teams do their job. For this reason, we decided to record both numbers for our project, functional and code coverage.

In our case, we recorded the code coverage numbers using the corresponding feature of our simulation tool, Questasim [38]. In the case of the functional coverage, we have a dedicated module with plenty of covergroups with their corresponding bins. These count values are observed in the interface signals of the VPU.

For example, we extract numbers of the possible element count for load transactions in OVI. We also check what indexes have been seen at the mask interface for vector indexed load instructions. These numbers help us identify if we are testing enough possible cases for vector load instructions. In further chapters, the coverage numbers we have extracted will be analyzed.

3.4.6 Continuous integration

In addition to the previous techniques, we included our environment and tools in Continuous Integration (CI) pipelines. These allow us to create even more possible cases, potentially finding more bugs in the design. In addition, continuous integration is used in our repository to check that every commit has not broken previously working features.

In Figure 3.8 there is a simplified diagram of the whole environment and flow of the simulation of a random test. In it, we can see all the big main modules mentioned before, along with a representation of how a random Riscv-dv binary is executed. This is included in some of the continuous integration pipelines that we developed and gave us the chance to generate and execute an arbitrary number of random tests automatically. Our Continuous Integration infrastructure is built in the open-source CI server Jenkins. We created a set of pipelines that interact to have the most error-free design possible.

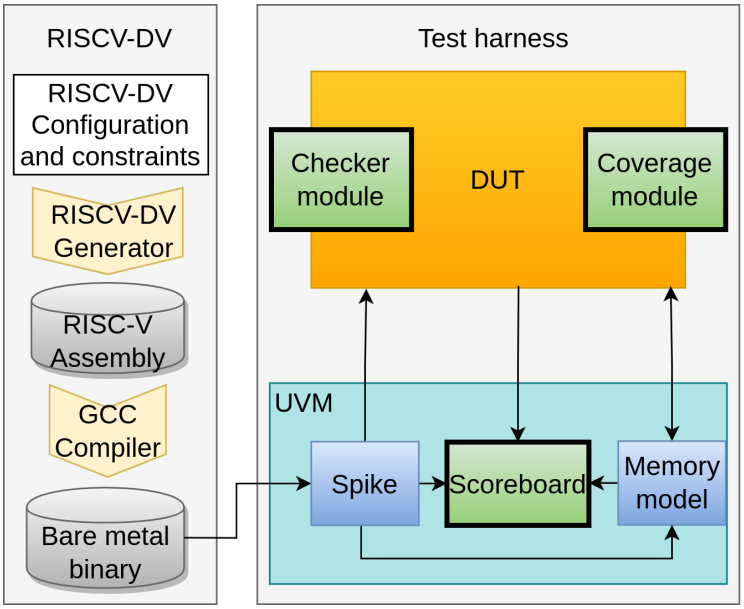


Figure 3.8: Simplified full environment diagram

At the beginning of the verification process, we had what we called "Night Runs". This was a CI pipeline that would execute 20 random tests every night. Later that day, another pipeline would collect the results of these tests and post them in a Gitlab issue. This way, the RTL team would have a table with the failing tests and interesting related information like the last executed instruction, the number of instructions or whether the test failed because of a timeout or a result mismatch.

This information helped the RTL team find the cause of the bugs and fix them faster. After the fix, most failed tests were saved as regression and executed in future commits to check that the RTL is still doing fine. However, we found the Night Runs approach too manual and at some point in the process, it even got outdated, as it was not finding as many errors as at the beginning. We decided to increase the number of tests generated and executed, but the design was getting healthier and we had to change the approach, as it was not enough.

Therefore, we introduced a new set of pipelines to find more bugs. We have the following pipelines:

- New tests: Generates random tests with Riscv-dv, compiles the DUT, executes the binaries and does a classification between passed and failed tests. Passed and failed tests are separated into two directories. The first ones are used to create a regression set and the last are kept for debugging and checking until the RTL team fixes the error. In addition to all this, coverage is collected and saved for these tests.

- Retry: For each change in the main branch of the DUT repository, the set of previously failed tests is re-executed and classified again in passed and failed. The assumption is that changes in that branch often aim to resolve known bugs, making them disappear from the failed folder.

- Selection: Every day at midnight, if the number of tests classified as passed is above

a certain threshold, tests are ranked by the collected coverage and we create two sets of regressions, a complete and a small one. The only difference between the complete and small sets is the number of tests in each one.

- Regressions: When there is a change in the DUT, which is a candidate to be merged, we execute the small set of tests called regressions to check the correctness of these changes. If these fail, the merge cannot be performed. Additionally, a more extensive and complete set is executed once per week to ensure recent changes do not break known-good tests.

All these have created a suitable testing environment that is automatically run and has been able to find many bugs independently, even much later than we started the verification process of another project. The splitting between passing and failing tests has created an extensive and reliable set of regression tests that help us detect whenever the RTL introduces a bug. In addition, it helps us to test the environment when we add new features to it, keeping it healthy.

In Chapter 6 the results of these CI pipelines and the collected coverage will be discussed and analyzed.

Chapter 4

UVM and interface interaction

In this chapter, we will go over the basic functioning of the environment together with the DUT. In section 4.1 we will describe the idea behind the UVM testbench structure and its implementation. From then, details will be given about every particular OVI sub-interface in sections 4.2, 4.3 and 4.4. This way, the basic execution flow of instructions in the VPU and the testbench will be explained.

4.1 UVM Environment overview

As explained in Section 2.2.1, a typical UVM would have one environment, agent, driver and monitor, for example. This is not always true, but how a UVM environment is presented in documentation. In our case, we were guided by this documentation when we were planning our testbench. We observed that only simple or small modules were verified in the examples.

These are often modules that receive some inputs and send some outputs back in an ordered fashion. This suits UVM perfectly, as while we keep the inputs constrained and legal, we will obtain the expected outputs. An engineer can get 100% coverage with constrained random stimulus by just running simulations.

However, this scheme does not seem to support modules with a complex interface, where the UVM testbench should be reacting to possible outputs or implementing one side of a protocol. Our DUT is not a simple module, so we had to implement a UVM testbench that could handle the VPU.

For this, we looked for ways of doing it in the documentation we had, but we found no good examples. We have found that in industry, each engineer takes on a module or two for big designs. This eases the development of the testbenches while at the same time providing a much more in-depth verification process.

In the previous chapter, we commented that we tried this approach at first, but after developing the testbenches for some of the most important modules, we could not keep up with that rate. In addition, most of the modules did not have specifications and the design was still in development, so many of the features taken into account when creating the testbench could change afterwards, making it even useless.

Moreover, we knew that after potentially creating a UVM testbench for every module in the project, we would have to test the whole integrated DUT, maybe in a regular RTL testbench. Otherwise, we would not know if it as a whole worked properly or not. All these reasons made us replan the strategy for the testbench and directly target the whole DUT.

As said, though, we did not have any example of how to do this. We were looking at how to create a testbench that implemented the whole scalar core part of the OVI to connect to the VPU and follow a normal execution flow.

Since we already had a very detailed OVI specifications document, we knew that it was divided into sub-interfaces. These can work independently at any point in time, although most of them collaborate in many functionalities of the VPU. We came up with the idea of creating a big UVM testbench that contained a UVM agent for each sub-interface to simplify the implementation of each of these.

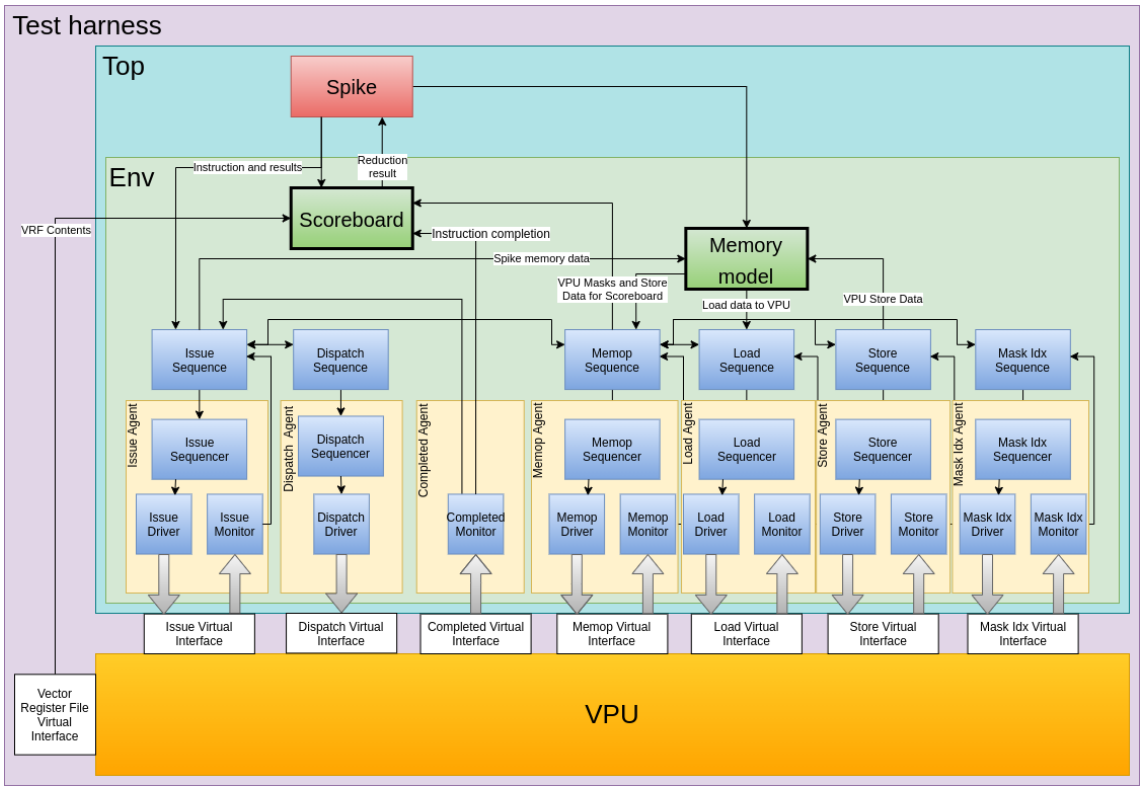


Figure 4.1: Full UVM Testbench

This can be seen in Figure 4.1. In it, we can observe each of the independent agents, all instantiated in the UVM environment, where we also have a memory model. In this diagram, we can also observe the whole structure of the UVM testbench, where the DUT and the virtual interfaces are instantiated inside the test harness. The UVM top module has access to this test harness and connects its virtual interfaces through the UVM config database with the rest of the environment. The different UVM agents instantiated inside the UVM environment are connected through these virtual interfaces to their corresponding sub-interfaces.

However, what outstands the most in this diagram is the number of interconnections be-

tween the agents. This is due to the different sub-interfaces being in almost constant collaboration for the VPU to execute an instruction.

In regular UVM testbenches, sequences are in charge of generating transactions that contain information to stimulate the interface of the DUT, most of the time by taking advantage of *randomize* calls. However, we needed the sequences to react to the DUT outputs and send precise inputs to respond to them in our environment. Also, as we decided to use different agents in our testbench, we needed a different sequence type for each. These could not be the same as they produced completely different data for their agents. Additionally, in many cases, these sequences would need information from others to produce their values, which we found particularly difficult to handle.

For tackling this need for different sequences and communication between them, we decided to use a system based on a UVM virtual sequence.

These are sequences that control stimulus generation using several sequencers and they are often used to coordinate the stimulus across different interfaces and the interactions between them. A virtual sequence is often the top level of the sequence hierarchy and differs from a normal sequence in that its primary purpose is not to send sequence items. Instead, it generates and executes sequences on different target agents. For this, it contains handles of the target sequencers and these are used when the sequences are started. This structure can be seen in Figure 4.2.

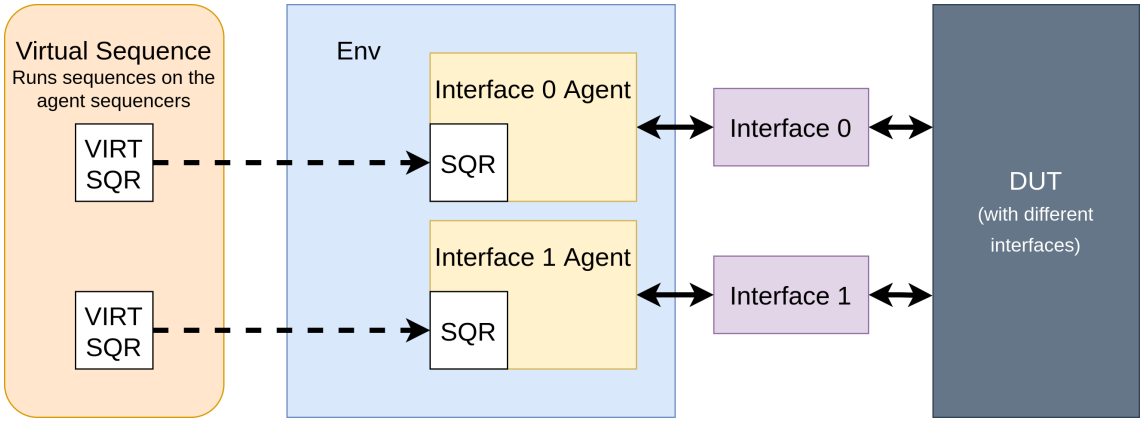


Figure 4.2: Virtual Sequences in UVM Testbenches, based on [17]

In the diagram, we can see that this UVM feature perfectly fits our needs. In the example, the DUT has two different sub-interfaces, for which the testbench presents two agents, one for each. These agents contain a sequencer, most likely connected to a driver interacting with the DUT. Each of these sequencers needs a different sequence type, as the sub-interfaces are different. The testbench can provide two different sequences that generate different transactions and assign them to their corresponding sequencer through the virtual sequence.

What is different in that system is that sequences are entirely independent. In our DUT, transactions could be generated independently in terms of time, but their values depend on ones sent to other agents. Therefore, we need to connect the different sequences somehow to communicate between them.

To tackle this, considering that the sequences are instances of the corresponding class, we can create them inside other sequences and directly access their values. These are only created once, so we took advantage of what is called a *singleton*. This means that the initially created instance of the corresponding sequence is unique and every following instantiation of the sequence will refer to it. Therefore, we instantiate sequences once in the environment but reference them inside the others when necessary, providing them access to their variables and generated transactions.

All this creates a complex class hierarchy. In a typical UVM testbench, we find the UVM top module, which instantiates the test harness. Inside this, the DUT is connected to the virtual interface to which the agent will have access. In addition, the test specified in the simulation command will be run in the top module, obtaining transactions from the corresponding sequence.

In our environment, we have all this but with additional particularities. The fact that we need different sequences for every agent creates new levels inside the class hierarchy. The virtual sequencer and sequence appear, instantiating the different sequencers and sequences, respectively.

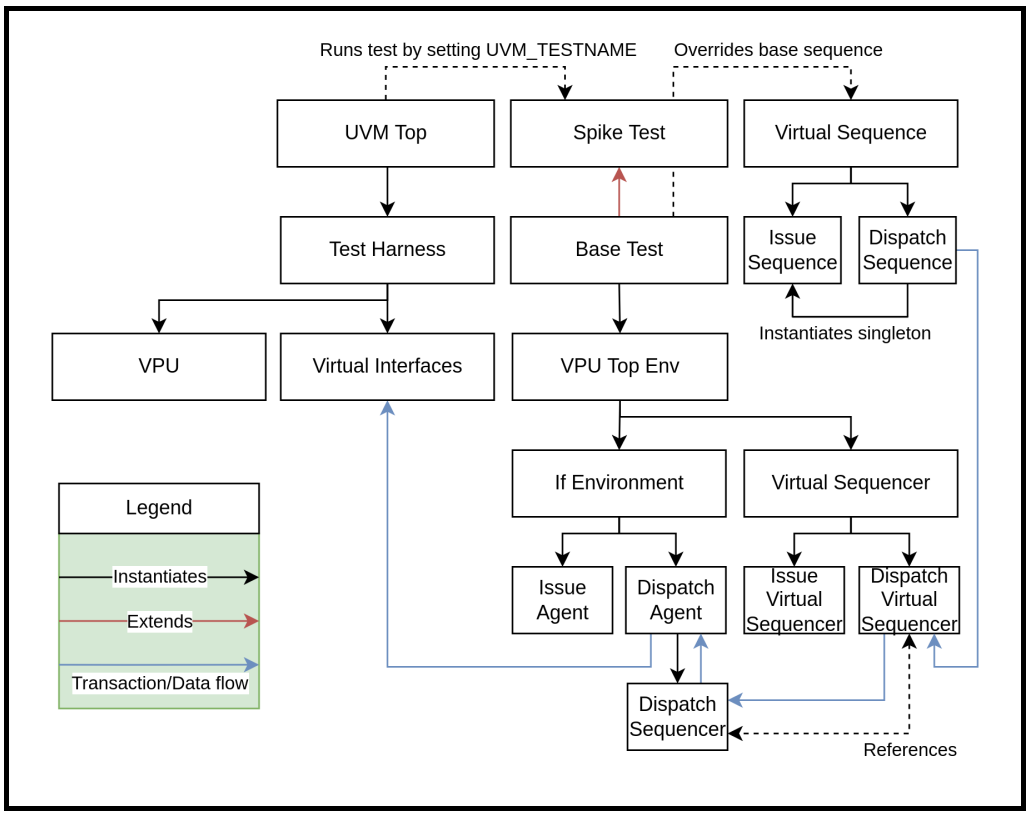


Figure 4.3: Class diagram of our UVM Testbench

All the previous can be seen in Figure 4.3. In it, we can see the class hierarchy of our UVM testbench and what class instantiates the other, the connections and other relevant information. It is worth noting that, to make it easier to read, only full Dispatch related classes are visible. The basic UVM structure can be seen following the base test, which instantiates the Top Environment, which contains the actual UVM environment that instantiates the UVM agents. Each of these contains a sequencer, a driver and a monitor.

These last two are connected to the virtual interfaces in the test harness using the UVM config database and have access to the values in the DUT interface.

The diagram shows a transaction going from the sequence to the virtual sequencer, specifically its corresponding sequencer, which refers to the sequencer inside the agent. There, the sequencer will send the transaction to the driver, which will stimulate the interface of the VPU.

Furthermore, in the figure we can see that the base UVM test instantiates the environment and how the Spike test, which extends from it and therefore contains its features, is run using the simulation command. Inside the base test, the virtual sequence overrides the base sequence, setting the UVM test to produce the corresponding values for the agents. Finally, we can see that the Dispatch sequence instantiates the Issue sequence singleton. As will be explained later in this chapter, this is necessary because it needs the Issue values to produce its ones.

To summarize all the previous points, we use the sequences inside the virtual sequence as the generators of transactions for the corresponding agents. For this, we connect them to the ports of the necessary sub-interfaces sequencer and react to the sent transactions, starting the necessary processes or responding to specific data or messages. This way, the virtual sequences create the transactions and send them to the driver through the sequencer, which via the virtual interface stimulates the VPU.

The following sections will go through the sub-interfaces needed to execute basic vector arithmetic-logic instructions; the Issue, Dispatch and Completed sub-interfaces. In them, we will analyze their particularities and how their specific sequences generate the stimulus to perform their specific purpose in the basic functioning of the VPU.

4.2 Issue sub-interface

As mentioned in Chapter 3, the Issue sub-interface is in charge of providing the vector instructions to the VPU. As it is the starting point for any instruction in the accelerator, the Issue sub-interface is almost independent of the rest of the OVI buses. However, as we will see in this section, it is not entirely independent.

Along with the instructions, additional data to execute the instructions must be provided through the Issue sub-interface. In Table 4.1 we have all the widths and directions (input or output) for the signals in the Issue sub-interface. When issuing a vector instruction through the OVI, the scalar core will need to provide an identifier, so both sides of the interface can collaborate to execute it together, called *sb_id*. Sometimes it will also provide the possibly needed scalar operand, used in instructions such as *vadd.vx* or *vsub.vx*, where the value of a scalar register is operated for the whole vector. This value will only be considered when one of these instructions is issued.

In addition, the *v_csr* signal will always be needed. This signal contains the value of the vector-related CSRs needed to proceed with and configure the execution of the instruction. It contains several fields, which can be seen in Table 4.2.

None of the above is taken into account if the *valid* signal from the interface is not set to '1', indicating that a new instruction is being issued. Then, all the signals in the inter-

Signal	Width	Direction	Description
valid	1	input	Indicates to the VPU that a valid instruction is being sent.
inst	32	input	Contains the instruction to be issued to the VPU.
scalar_opnd	64	input	Contains a value from the scalar core registers to be used in a vector instruction.
sb_id	4	input	Indicates the identifier that the scalar core assigned to this instruction. This will be used to track it during its execution and after its completion.
v_csr	40	input	Contains the value of certain Control Status Registers (CSR) from the scalar core related to vector instructions.
credit	1	output	Indicates that the VPU can process a new vector instruction.

Table 4.1: Issue sub-interface signals

Field	Width	Description
vstart	14	Indicates the first element of the vector that must be operated on for the instruction.
vl	15	Indicates the number of elements that the instruction must be executed with.
vxrm	2	Indicates the fixed point rounding mode in the core CSRs to be used for the instruction, if necessary.
frm	3	Indicates the floating point rounding mode in the core CSRs to be used for the instruction, if necessary.
vlmul	2	Indicates the number of vectors to be operated with for the same instruction. Using vlmul different from zero, by executing a vadd instruction one may write two vector registers. In the case of the VPU, only VLMUL == 0 is supported.
vsew	3	Indicates the vector element width. In the case of the VPU, element widths of 8, 16, 32 and 64 bits are supported.
vill	1	Indicates whether the issuing instruction is illegal. In this case, the VPU must mark it as illegal at completion.

Table 4.2: CSR signal fields

face must take a valid value for the instruction to be correctly executed. According to the Vector Specifications, that is a legal instruction and a valid configuration specified in the CSRs. Along with them, we provide a *sb_id* to identify the instruction during its execution, emulating what the scalar core would do. If we wanted to simulate the VPU more realistically, these could be random, but we decided to make them sequential as it was easier to implement and debug later.

In Figure 4.4 there is a diagram of the Issue UVM agent. In it, we have a simplified view of the previously explained structure, where we only show the sequence and interface-related UVM components. In addition, we can see the specific interconnections of the sequence with the rest of OVI. Spike has been coloured red in the figure to indicate that it is a library. This library is used in the Spike sequence, directly extending from the Issue sequence. This Spike sequence is the one used for the usual testbench and is in charge of providing instructions and other issue-related data to the driver of the Issue agent. The Spike library has a set of DPI functions to obtain specific data from the ISS.

For instance, the body of the Spike sequence calls the *run_until_vector_ins* DPI function from the Spike library to obtain a vector instruction until the binary has been entirely executed in Spike.

When Spike finds a vector instruction, it returns the encoded instruction, results and other relevant data for its execution in the VPU. This instruction must be sent to stimulate the VPU. For that, a set of functions save relevant information about the instruction for its execution, such as operands, memory data or results, among others. In addition, all this

information is sent to the scoreboard as explained in Chapter 3 to prove the correctness of the results of the DUT and provide additional debug information.

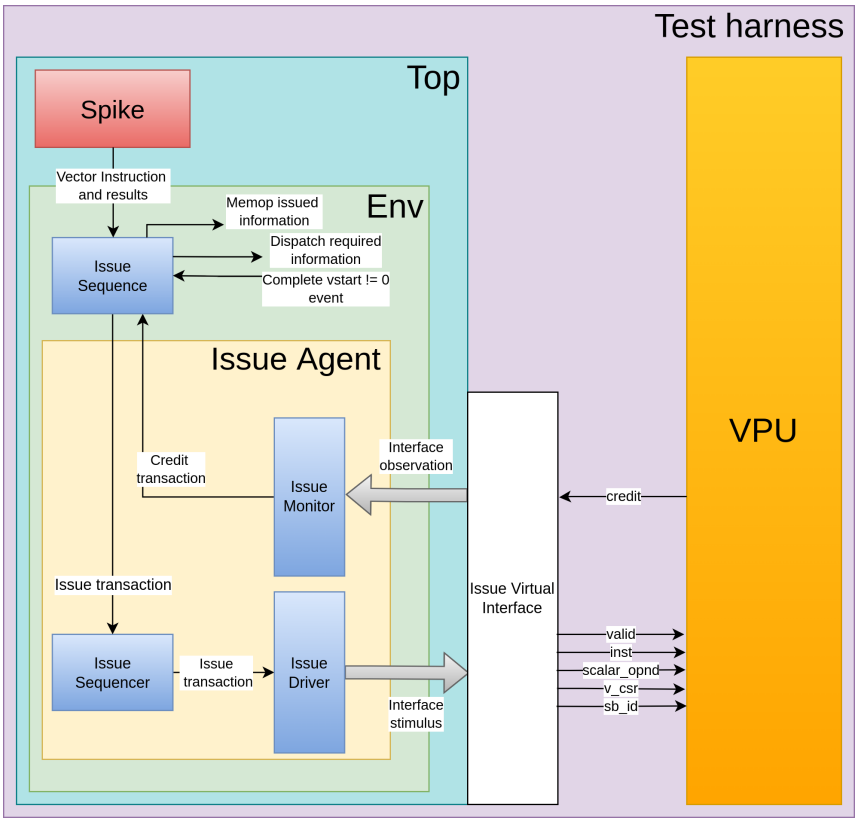


Figure 4.4: Issue UVM agent and connections in detail

Once the scalar core issues a valid instruction, the VPU will allocate it in its Pre Issue Queue module. This is the first stage of instructions inside the VPU. It is classified among arithmetic instructions or memory instructions and sent to the corresponding queue. The Pre Issue Queue is also in charge of the credit system of the VPU side of the Open Vector Interface.

As seen in the interface and the OVI explanations, the Issue sub-interface uses a credit system. This is how the VPU can notify the core whether it can execute a new vector instruction. In the case of our VPU, it has up to four credits, but it can be configurable as it depends on the size of the Pre Issue Queue module. Therefore, the scalar core consumes a credit each time it issues an instruction through the sub-interface. In our case, if it issued four straight instructions without receiving a credit back, it would not be able to send any more instructions. As said, the Pre Issue Queue is in charge of returning the issue credits. It does this whenever an instruction leaves the queue, meaning that it can allocate a new one.

As we are performing the role of the scalar core, we have to issue the instructions, always taking into account the credit system. For this, we used our UVM agent and integrated all of its capabilities. After Spike returns an encoded instruction and its related data, it is then saved in the Spike/Issue sequence, which is now ready to create a transaction for the DUT.

At the beginning of the simulation, the scalar core has all the credits ready, which means we can always issue an instruction. Therefore, a transaction is created using the *uvm_do_on_with* UVM function (Code Listing in 4.1) that contains the instruction and the aforementioned necessary data from Spike. This transaction is then sent to the driver via the sequencer, which stimulates the DUT. This can be seen in Code Listing 4.2. Since we initially have four credits, this can be done these times in a row, each consuming one. If that happens, we will not be able to use a credit again until the VPU returns it.

```
`uvm_do_on_with(m_req, p_sequencer,
{
    valid == 1;
    instr == instruction.ins;
    data  == tmp_data;
    sb_id == instruction.sb_id;
    csr   == csr_bits;
})
```

Code Listing 4.1: Transaction generation in the Issue Sequence

```
task drive(issue_trans trans);
    vif.cb.valid <= trans.valid;
    vif.cb.instr <= trans.instr;
    vif.cb.data  <= trans.data;
    vif.cb.sb_id <= trans.sb_id;
    vif.cb.csr   <= trans.csr;
endtask : drive
```

Code Listing 4.2: Issue signals stimulus at the Issue Driver

Hence, if we did not have credits in a later simulation stage, the sequence would not produce a transaction. To observe the credit return from the VPU, we use the UVM monitor and watch the Issue sub-interface credit signal. Once the VPU returns a credit by setting this signal to '1', the monitor will encapsulate it in a transaction and send it back to the sequence, which will increase its credit counter. This will allow it to send new transactions to the driver, following the binary execution. At the same time, the issued instructions are sent to the Dispatch sequence to be treated, which will be explained in Section 4.3.

In addition to this, we needed to ensure that the VPU always received valid values while issuing an instruction. For that, apart from the UVM agent, we included the *a_no_issue_unk* assertion, which can be seen in Code Listing 4.3. This assertion essentially checks that whenever the Issue *valid* signal is '1', none of the rest of the Issue sub-interface signals can take unknown values.

```
a_no_issue_unk : assert property(disable iff(!rsn_i) @(posedge clk_i) issue_if.valid |->
    !$isunknown(issue_if.instr) and !$isunknown(issue_if.data) and
    !$isunknown(issue_if.sb_id) and !$isunknown(issue_if.csr))
    else begin `assertion_level_report($sformatf("error.VU.%m")); end
```

Code Listing 4.3: Unknown Issue signals assertion

This way, we complete the normal Issue flow as seen in Figure 4.4. However, in that diagram there are two extra inter-sub-interface connections.

The first one is related to vector memory instructions. Although it will be explained in Chapter 5, it is worth noting that memory operations (which will be called memops from now on) have a special place in OVI, as they require a higher amount of interaction between the scalar core and the VPU. Therefore, due to special needs for their execution explained in the dedicated chapter, we provide necessary data for its execution when issuing a memory instruction.

In essence, we provide all the Spike instruction-related information to the Memop sequence along with the *sb_id* with which we issued the instruction to track it down while in other agents. To do this, we created an instance of the Issue singleton in the Memop sequence, which may now access all the memory instructions being issued.

The second link is with the Completed agent. This connection is needed because some memory instructions may finish their execution earlier than expected due to various reasons, to be discussed in Chapter 5. As we will see in Section 4.4, this will be identified by the Completed monitor, which will capture these and trigger a UVM event. We connected the Issue Sequence to this event because if one of these happens, we will need to issue the instruction back and the rest of the issued instructions pending completion. Therefore the sequence will react to this event and re-build its instruction queue to insert the already issued instructions.

4.3 Dispatch sub-interface

The dispatch sub-interface is used to confirm or discard the execution of the instructions. As stated in the OVI specifications, "for each issued instruction, there must be a dispatch". Even though it assumes such important responsibilities, the dispatch sub-interface is rather simple. Its signals can be seen in Table 4.3.

Signal	Width	Direction	Description
next_senior	1	Input	Indicates that the instruction identified by sb_id is confirmed to complete execution.
kill	1	Input	Indicates that the execution of the instruction identified by sb_id must be discarded.
sb_id	4	Input	Contains the identifier that the scalar core assigned to an issued instruction.

Table 4.3: Dispatch sub-interface signals

In Figure 4.5 can be seen all details of the Dispatch agent and how it is connected to the corresponding sub-interface.

The figure shows that Dispatch only has a driver, as this sub-interface is unidirectional. This is from the scalar core to the VPU. Therefore, we do not need a monitor to watch possible outgoing transactions. In addition, the figure shows the only connection between agents. In the case of the Dispatch agent, it is only connected to the Issue Agent, which provides it with information about the issued instructions and what to send through Dispatch for them. Once again, we did this by instantiating the Issue singleton in the Dispatch sequence to have access to issuing instructions.

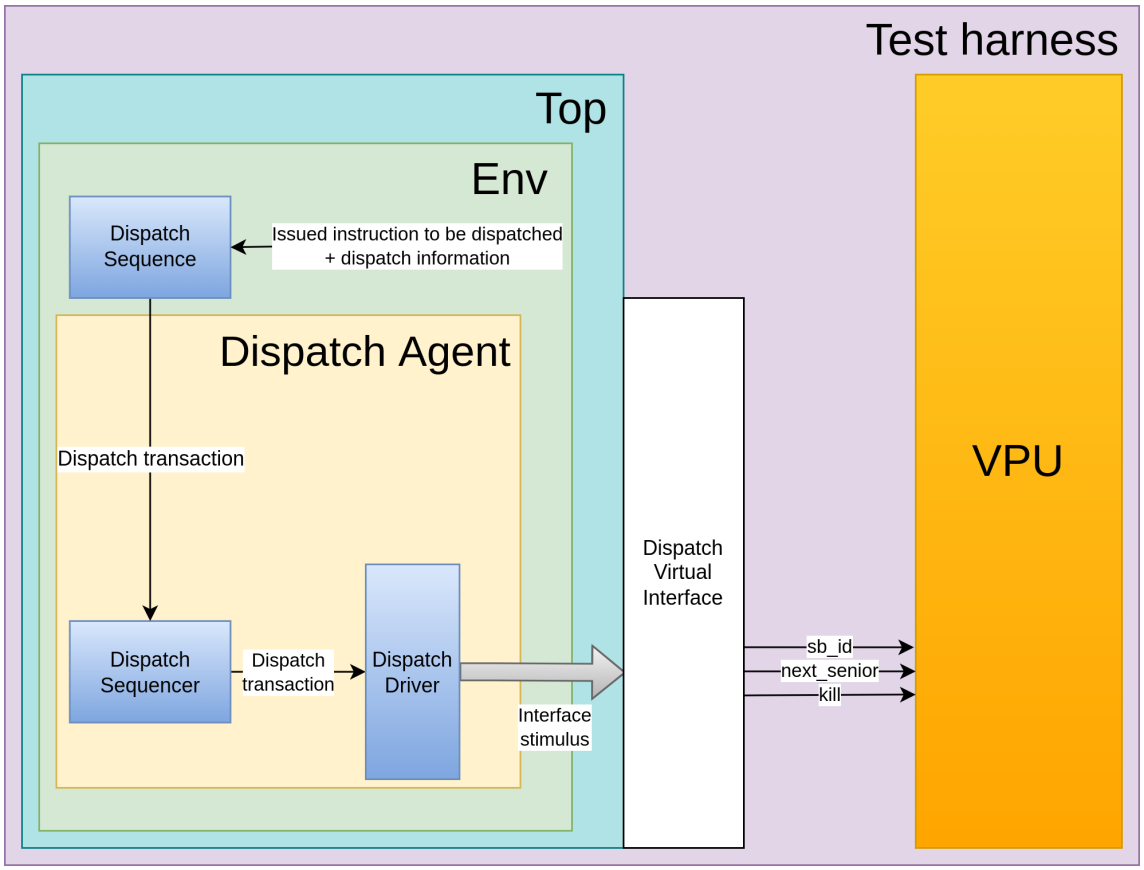


Figure 4.5: Dispatch UVM Agent and connections in detail

The need for the Dispatch sub-interface comes from the possibility that the scalar core issues instructions that may not be needed to complete in the end. This happens, for example, if the scalar core is executing speculative instructions. If the prediction results are wrong, the scalar core will discard all the executing instructions, including the vector ones, through the Dispatch sub-interface. Another option is if a vector memory instruction causes a recoverable exception, such as a page fault. In that case, the core will actively give up the instruction to solve the exception and issue it back afterwards. Something similar happens if a regular exception occurs, causing the scalar core to jump to an exception handling routine and discard the potentially issued instructions.

The last case where there might be discarded instructions is whenever an instruction completes with a *vstart* value different from zero. To be further explained in Chapter 5, vector

load instructions can be terminated earlier by the VPU in some cases, which it may indicate by setting a *vstart* value. If that is the case, the load instruction and the remaining pending ones will be re-issued. For that, all the previous instances of the instructions inside the VPU must be discarded using the Dispatch sub-interface.

As seen in Table 4.3, the *next_senior* signal ensures that a specific instruction will finish while the *kill* signal is used otherwise. Considering that for every issued instruction, there must be only one dispatch sent, if *next_senior* is sent for an instruction, it is confirmed that the corresponding one will complete. Likewise, if a *kill* is sent for an instruction, it is sure that it will not be completed. This, however, can change if the same instruction is re-issued, as happens in some of the previous examples.

The dispatch information must be sent only once for instruction and in issue order, but it may be sent in different time windows. For example, if the scalar core is sure that an instruction will be completed, it may issue the instruction and send the next senior for it at the same time. If that is yet to be determined, the scalar core will send the next senior once that happens.

Another particular case comes when there are instruction issues during the execution of a memory instruction. If that is the case, as said, the memory instruction may have to be re-issued again in the future, needing to kill the rest of the issued instructions. For this reason, instructions after a memory operation are not sent their dispatch information until this last is completed. As said, if this one completes but must be re-issued, a sequence of kills will arrive for the executing instructions. On the other hand, if the memory instruction completes successfully, many of the following instructions may get their dispatch information sent in the following cycles.

The Dispatch sequence will directly access the Issue sequence instance and check for what instructions it must send the dispatch information. It will do so by using the aforementioned *uvm_do_on_with* function and sending the next senior or kill depending on what the Issue sequence set for the instruction. This depends on various factors, such as memory instructions not being completed or completing with the need to be re-issued. The Completed agent tracks whether these must be re-issued or not and communicates it to the Issue sequence.

To ensure that the VPU receives Dispatch information for valid instructions, that is, that have previously been issued, we added another assertion. This can be seen in Code Listing 4.4. In it, we can see that the right side of the implication is triggered whenever a Dispatch bit is set, either *next_senior* or *kill*. Then, the assertion checks the *_valid_id* structure, which among others, contains information about what *sb_id* have been recently issued. This structure has as many positions as different *sb_ids* can be in the environment, in this case, 16. It is in the *ISSUED* field that it indicates whether the corresponding *sb_id* is inflight or not. While it is true that this assertion covers mainly the work of the testbench, which is necessary to ensure its correct behaviour, it was also helpful when we needed to integrate the VPU with the rest of the EPAC modules.

```
property p_dispatch_if_existing_id;
    @(posedge clk_i) (dispatch_if.kill || dispatch_if.next_sen) |->
    _valid_id[dispatch_if.sb_id][ISSUED];
endproperty

a_dispatch_existing_id : assert property(disable iff(!rsn_i) p_dispatch_if_existing_id) else
    `assertion_level_report($sformatf("error.VU.%m"));
```

Code Listing 4.4: Existing sb_id assertion for dispatch transactions

4.4 Completed sub-interface

The VPU uses the Completed sub-interface to communicate to the scalar core the instructions finishing, their "completion". The rule for this sub-interface would be that every *next_seniored* instruction by the Dispatch sub-interface will make its completion. As foreshadowed by previous explanations, there might be different completion conditions, detailed in the following sections. These conditions will be notified using the signals from the sub-interface, listed in Table 4.4.

Signal	Width	Direction	Description
sb_id	4	Output	Indicates the identifier of the instruction that is completing if valid is '1'.
fflags	5	Output	Contains the Floating-point accrued exception flags for the completing instruction, if applicable.
vxsat	1	Output	Contains the Fixed-point accrued saturation flag.
valid	1	Output	Indicates that an instruction is completing.
dest_reg	64	Output	Contains an scalar result for the completing instruction, if applicable.
vstart	14	Output	In the case that the instruction could not be executed for all elements, it indicates the next element that should be executed for the instruction. If the instruction was completed, vstart should contain a zero.
illegal	1	Output	If the instruction being completed was illegal, this bit should be set.

Table 4.4: Completed sub-interface signals

For many of the instructions, only a few signals will be used. If the instruction is not a floating-point one, the *fflags* signal must contain a zero. The same happens for instructions that are not fixed-point ones, for which the VPU must set a zero in the *vxsat* signal. Otherwise, these signals may contain the corresponding resulting values. In addition, *dest_reg* must be set to zero if the instruction is not a vector reduction operation with scalar result such as the *vredsum.vs* or *vredmax.vs* instructions. If that was the case, this signal should contain the result of the reduction. We added assertions in the environment for these cases. In Code Listing 4.5, the assertion *a_floatp_flag* checks that for every instruction issued that is not a floating point related one, either a kill is sent or it completes with the *fflag* signal equal to zero. A very similar assertion exists for *vxsat* and *dest_reg* signals.

```
sequence s_not_floatp_inst;
    !(issue_if.instr[INSTR_OPCODE_END:INSTR_OPCODE_START] == 7'b1010111 &&
    ↪  (issue_if.instr[INSTR_FUNCT3_END:INSTR_FUNCT3_START] == 3'b001 ||
    ↪  issue_if.instr[INSTR_FUNCT3_END:INSTR_FUNCT3_START] == 3'b101));
endsequence

property p_floatp_flag (clk);
    logic [SB_WIDTH-1:0] _sb_id;
    @(posedge clk) (s_not_floatp_inst ##0 issue_if.valid, _sb_id = issue_if.sb_id) |->
    ((dispatch_if.sb_id == _sb_id && dispatch_if.nxt_sen) [->1]  ##0
    ((completed_if.sb_id == _sb_id && completed_if.valid) [->1] ##0 !completed_if.fflags)) or
    (dispatch_if.sb_id == _sb_id && dispatch_if.kill) [->1];
endproperty

a_floatp_flag : assert property(disable iff(!rsn_i) p_floatp_flag(clk_i)) else begin
↪  `assertion_level_report($sformatf("error.VU.%m")); end
```

Code Listing 4.5: Fflags signal assertion

The *illegal* bit will only be set by the VPU if the instruction results to be illegal. This could be because the scalar core marked it as illegal in the CSRs sent through the Issue sub-interface. Another possible reason may be that the VPU found an illegal encoding or CSR configuration during the decoding of the instruction. Otherwise, the VPU would output a zero through this signal.

Finally, the *vstart* signal is only set for the memory instructions, which are the only ones that are allowed to complete before operating on all the elements of the vector. Although this will be explained more in detail in the following sections, this value is needed for re-issuing the instruction. If the instruction was executed entirely for all vector elements, this signal must contain a zero.

However, what the VPU will use for all completed instructions will be the valid signal set to '1', along with the corresponding *sb_id*.

As seen, this sub-interface is unidirectional and the scalar core does not have to provide any value through its interface. Therefore, we decided to go without a driver for the Completed agent, shown in Figure 4.6.

In addition to this, the figure shows the Completed monitor, which is directly connected to the Issue Sequence. This connection is through a UVM Event, which indicates to the sequence that an instruction completed with a *vstart* value different from zero, as seen in Section 4.2. This will unleash the re-issue of the instruction with the same *vstart* output by the VPU, which will be sent using the trigger data slot of the event.

Furthermore, the Completed monitor is connected to the scoreboard to trigger the comparison of the instruction results. This is a regular monitor functionality, where it observes the Complete sub-interface. After detecting a '1' in the valid signal captures it and the rest

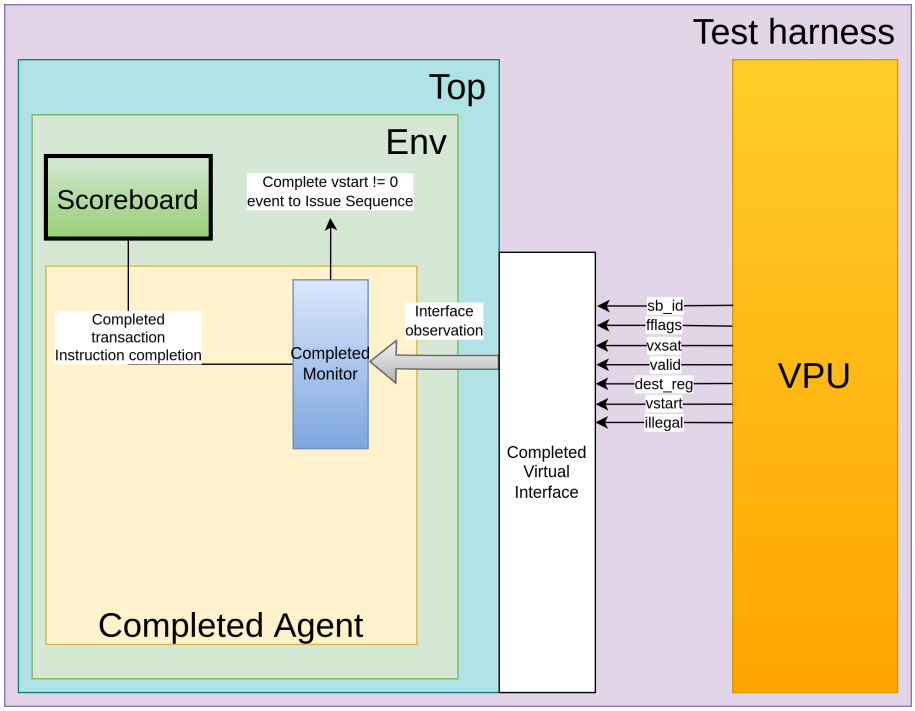


Figure 4.6: Completed UVM Agent and connections in detail

of the interface signals in a transaction. This transaction is then sent through an export port to the analysis port in the scoreboard.

```
1 // Function: write_completed
2 // Function that gets the values from the vpu_completed analysis port
3 function void write_completed (avispado_completed_trans completed_trans);
4     if (completed_trans.valid && m_cfg.enabled) begin
5         comparator_queue.push_back(completed_trans);
6         vdest_queue.push_back(vreg_if.rename_vdest);
7     end
8 endfunction : write_completed
```

Code Listing 4.6: Scoreboard write function of the port connected to the completed monitor

In Code Listing 4.6, we have the write function for the analysis port in the scoreboard. This function is triggered and run whenever a new transaction is written into the export port of the Completed monitor. Therefore, once the monitor observes a completed valid at the interface, it will send the corresponding transaction directly to the scoreboard.

In there, it is enqueued in a structure called *comparator_queue*, that contains all the completed instructions in order. Later on, a running task will see that the transaction is there and compare it to the results we obtained in Spike. In the code section, we can also see that there is another queue called *vdest_queue*. In this, we store the current value in the vector registers, where the actual results of the vector instructions are. We obtain them thanks to

a virtual interface hierarchically connected in the test harness module and passed through the UVM database to the scoreboard.

We need this structure because due to the register renaming feature of the VPU, we can only be sure of where the corresponding resulting values are at completion time. The physical register is indicated through a signal called *rename_vdest* in the Reorder Buffer of the VPU. If we compared the vector register contents some cycles after, this signal or even the values inside the vector might have changed. For this reason, we save the necessary vector register in the queue at completion time for the comparator task to compare them right with the correct Spike ones.

Apart from the regular vector register results, all the outgoing values from the Completed sub-interface arrive through the monitor to the scoreboard. This way, we can also compare different behaviours like scalar results (using the *dest_reg* signal), illegal instructions and *fflags* or *vxsat* bits.

We can handle and check the execution of basic arithmetic and logic vector instructions with all the previous mechanisms, which are shown in Figure 4.7. We can start their execution in the VPU through the Issue interface (step 1 in the figure), along with configurations and other possibly necessary data. We must use the same Issue *sb_id* to send the Dispatch information. In the most common case, the Dispatch Sequence will send a *next_senior* for the instruction, confirming that it will complete its execution (step 2 of the figure). Finally, after the VPU executed the instruction, we would notice that it finished thanks to the Completed Monitor (step 3 in Figure 4.7).

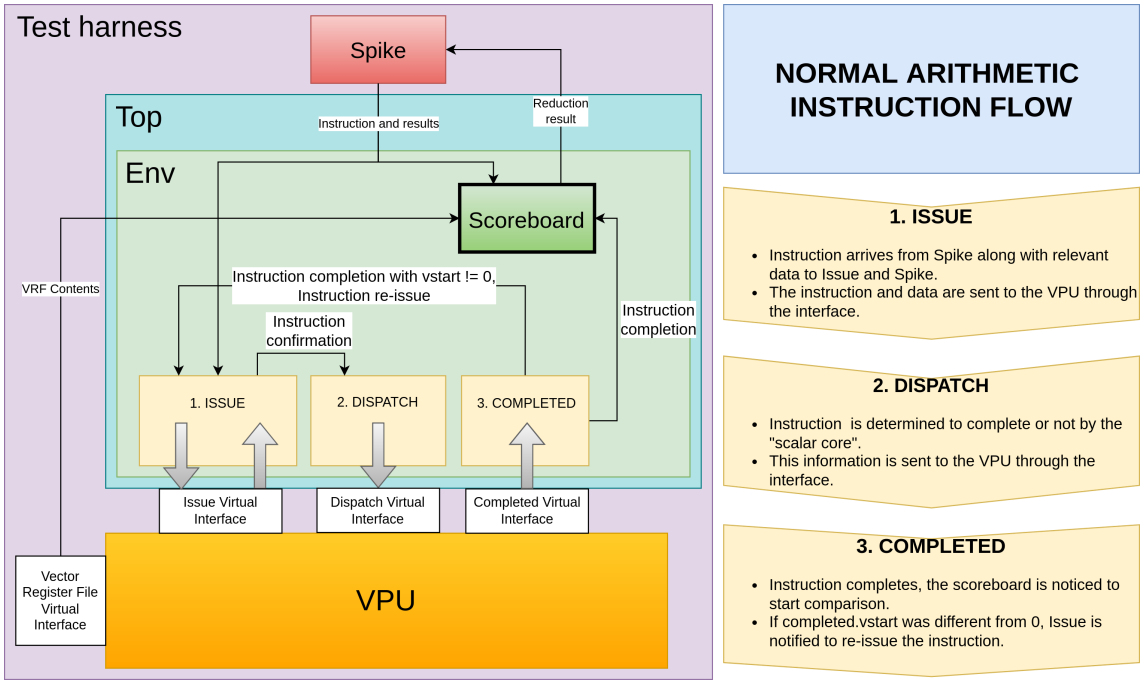


Figure 4.7: Simplified diagram of the arithmetic instructions flow

The monitor will then send a transaction to warn the scoreboard that the pending-to-compare instruction in the queue must be checked. The scoreboard will, at that point, obtain the results from the corresponding physical vector register and determine whether

the instruction was correctly executed or not after comparing them to the Spike ones.

All these connections can also be seen in a simplified way in Figure 4.7. The features mentioned above only support the execution of arithmetic/logic and other types of non-memory vector instructions. However, this was our first step and allowed us to test the environment in the first stages of the process while at the same time managing to find the first bugs. This was possible given that the environment could supply instructions and support their execution.

If, in addition to this, we consider that thanks to Spike, we can execute entire binaries, we can potentially provide any instruction to the VPU. This eases the generation of valuable stimulus for the design, as we can write our directed tests straight in RISC-V assembly. Furthermore, if we add Riscv-dv and its random binary generation into the mixture, we can supply random instructions continuously through Spike to the environment. Finally, we can check whether the VPU executes them correctly thanks to the scoreboard, which means we can perform the main basic verification tasks for arithmetic instructions.

In essence, although it might seem complex, it is just three interactions between the scalar core and the VPU. As we will see in the following sections, memory operations can use up to six sub-interfaces during one instruction. Therefore, the previously described arithmetic instructions environment is just the testbench's starting point. Nevertheless, it provides the most important or commonly used features of the whole environment: instruction Issue, Dispatch, and result comparison. These will be used for almost any instruction that reaches the VPU, whether it is an arithmetic or memory operation.

In the sections in Chapter 5, we will describe how vector memory instructions are handled inside our environment and how we interact with the VPU to be able to execute them. This includes from Issuing and Dispatching the instruction, as previously detailed, to providing memory data to execute load instructions.

Chapter 5

Memory operations emulation in the UVM testbench

This chapter will describe the execution flow of memory operations through OVI and how we provided sustenance for them in the UVM. In section 5.1 there is an introduction to memory operations in OVI and the main issues that they arise. In Sections 5.2, 5.3, 5.4 and 5.5 specific details of each memory sub-interface will be provided, in order to obtain a better understanding of the whole testbench.

5.1 Memory operations (*memops*) in the Open Vector Interface

Memory operations are the main ways of sending data to and from the Vector Accelerator. These include the load and store instructions. Specifically, in the EPI VPU implementation only the "element" version of these instructions are implemented, *vle* and *vse*, which operate with elements of the width specified by SEW. In the context of vector or SIMD instructions, these are often called *scatter* and *gather* operations. In addition to the distinction between load and stores, we have different types for both memory operations:

- Unit-strided: Memory elements are treated sequentially starting from a base memory address.
- Strided: Memory elements are treated sequentially with a gap between them, called *stride*, starting from a base memory address. If stride is '1', the instruction will essentially be a unit stride memory operation, whereas if the stride is '2', an element width gap will be left in memory between valid data.
- Indexed: Vector elements are assigned a memory address according to an index provided by the values in elements of another Vector register.

Whether a *vle* or *vse* is Unit-strided, Strided or Indexed is determined by the bits 26 and 27 of the instruction encoding. If this field of the instruction, called "mop", is equal to "00", the instruction is a Unit-strided one and if it contains "01" is a Strided memop. Finally, if the mop field is equal to "11", it means that the instruction is an indexed memory operation.

In a strided memory operation, the stride amount in bytes is determined at Issue time,

when it must be sent through the *scalar_opnd* signal of the sub-interface. Furthermore, all these operations can be masked or not, depending on the 25th bit of the instruction encoding.

Memory operations are critical delay points in vector implementations, as they are for other accelerators that use DRAM and processor cores. Memory delays depend mainly on latency, cache misses, and other memory-hierarchy-related reasons for scalar cores. For vector processing units, however, they depend on additional factors like the amount of data they operate with.

In many Vector Processing Unit projects, the IP is directly connected to a cache memory or to a module connected to it, ordering the data sent/received. This allows the Accelerator to have a big cache bus and obtain many elements in one single transaction. In the case of OVI, the scalar core is connected to the cache and is responsible for handling cache lines and communicating with the VPU. This causes the memory-related sub-interfaces of OVI to be very active and busy during the execution of these instructions, as the VPU and the scalar core have to exchange many transactions and data during them. In the waveform in Figure 5.1 we can see an example of this, where three memory operations are being executed. In the figure, all the memory-related sub-interfaces of OVI have much activity. These sub-interfaces are the Memop, Load, Store and Mask_idx ones, detailed later in this chapter.

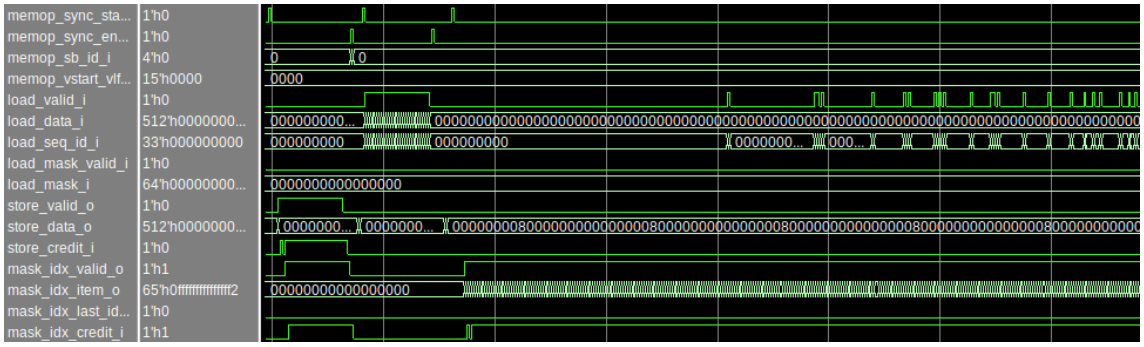


Figure 5.1: Memory operation related sub-interfaces during three memops

This also means that any memory-operation-related issue that may appear during the execution of the instruction will not directly be seen by the VPU, making it agnostic to how the cache reads/writes are being done.

Considering that we, as the testbench designers, are in charge of reproducing the same situation as the scalar core would face in actual execution, we have to consider all these and more aspects. For example, the memory would have values stored by the core in a real execution. In our case, we need to provide these to the VPU for load instructions. As we will see in the following sections, we do these using Spike's memory, as it will act as the scalar core itself. As explained later in this chapter, we will use this reference model to extract the data and save it in our memory model. An additional use of the Spike ISS will be to check the correctness of the values and execution of the Store operations performed by the VPU.

As one may expect, providing actual data through Spike and sustaining the execution of the memory instruction already implies facing many challenges. However, these get even

worse if we consider that we need to reproduce all possible cases that can occur during their execution. The following sections will explain these cases and how we tackled them.

5.2 Memop sub-interface

The Memop sub-interface is used to coordinate the start and finish of a memory operation. This is necessary because even though the scalar core issues the instructions, these may get enqueued inside the VPU. If this was the case and the scalar core started sending memory-related transactions through the interfaces, these could get lost. For example, this can happen if multiple memory operations are issued in a row into the VPU. As only up to two loads and one store can be executed simultaneously, the rest are put inside a queue. In the case of our VPU, when instructions are issued, they are distributed among an arithmetic instruction queue and the load-store queue, where memops are put. Afterwards, once the memory instruction is at the head of the load-store queue and it is ready to accept or send transactions for it, the VPU will send a signal to start the operation.

Signal	Width	Direction	Description
sync_start	1	Output	Indicates that the execution of a memory operation has started.
sync_end	1	Input	Indicates that the execution of the memory operation identified with sb_id is finishined.
sb_id	4	Input	Contains the identifier of an in-flight memory operation that is being finished.
vstart_vlfof	15	Input	Contains the next element to operate for the instruction in case the scalar core requires to restart the instruction. If the instruction was completely finished, it will contain a '0'.

Table 5.1: Memop sub-interface signals

In Table 5.1 there are the descriptions of the Memop sub-interface signals. As seen, the *sync_start* signal will be employed by the VPU to indicate that a memory operation must be started through the OVI. On the other hand, the scalar core will set all the other signals together to indicate it once the operation has finished.

At that moment, the scalar core will set *sync_end* to '1' along with the *sb_id* to indicate that the corresponding memory instruction has finished the operation. In the standard case, the *vstart_vlfof* signal will be set to zero, indicating that the memory operation was finished without problems in the scalar core and having operated on all the elements of the vector (as set in the vector length at issue time). However, the scalar core may find some issue while executing the memory operation that temporarily forces it to stop its execution, like a page fault exception. If that is the case, the scalar core will notify the VPU by setting the *vstart_vlfof* signal to the first element that could not be treated. Later on, the scalar core would resume the execution of the instruction by re-issuing it with a *vstart* value in the corresponding field of the CSR Issue signal equal to the previous *vstart_vlfof*.

In addition, there is a different case in the signal for fault-only-first loads (*vlfof*). These are related to a specific type of load that can only trap in element 0. If an element greater than 0 causes an exception, the vector length is updated through *vstart_vlfof* and no trap is taken. Nevertheless, the initial implementations of the VPU did not support these types of loads, so we did not take them into account for designing the environment. For the rest of the cases, as the testbench does not suffer from page faults or other possible problems,

we have to reproduce them through the corresponding agent.

In Figure 5.2 we have a detailed diagram of the UVM agent for the Memop sub-interface. As it can be seen, it is much more complex than the ones that we have seen before because it needs many connections to other agents in the environment.

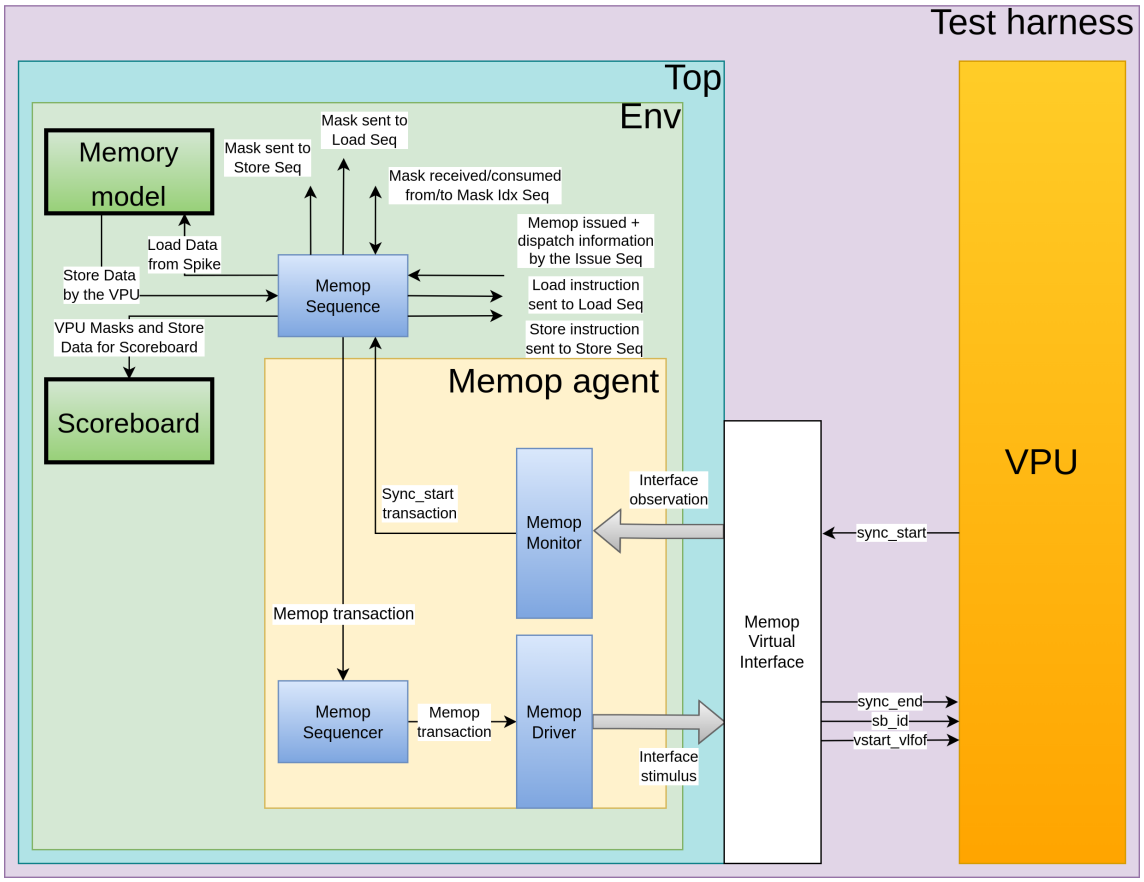


Figure 5.2: Memop UVM Agent and connections in detail

The first doubt that may arise from considering all the previous information is why does the *sync_end* signal have a *sb_id* associated and there is not one for the *sync_start* signal. The answer to this is because, as explained, the memory instructions are enqueued inside the VPU and it sends the *sync_start* for them in issue order. While this is perfectly fine for the scalar core, as it is the same entity that issues the instructions and sends the dispatch information, it presents an issue if we want to have a fully distributed and independent set of agents for the environment.

For this, we also instantiated the singleton corresponding to the Issue sequence inside the Memop sequence. From it, every time we observe a *sync_start* through the interface thanks to the monitor, we extract the first pending-to-start memory instruction. These are detected and saved in the Issue sequence when retrieved from Spike. In addition to this, we included an assertion that checks whether the VPU sends the correct amount of *sync_starts*. This assertion can be seen in Code Listing 5.1. This portion of the code belongs to a *checker* [45] we created for the VPU. A System Verilog *checker* is often used in these projects to add verification code such as assertions to the modules.

```
always_ff @(posedge clk_i) begin

    int tmp_memops_to_sync_start;
    if (!rsn_i) tmp_memops_to_sync_start = 0;
    else begin
        tmp_memops_to_sync_start = memops_to_sync_start;
        if (dispatch_if.kill && issued_memop(dispatch_if.sb_id)) tmp_memops_to_sync_start--;
        if (memop_if.sync_start) tmp_memops_to_sync_start--;
        if (issue_if.instr[INSTR_OPCODE_END:INSTR_OPCODE_START] == LOAD_INST ||
            issue_if.instr[INSTR_OPCODE_END:INSTR_OPCODE_START] == STORE_INST &&
            issue_if.valid) tmp_memops_to_sync_start++;
    end
    memops_to_sync_start = tmp_memops_to_sync_start;
    a_correct_sync_start : assert (memops_to_sync_start >= 0) else
        `assertion_level_report($sformatf("error.VU.%m"));

end
```

Code Listing 5.1: System Verilog assertion to check correct Memop *sync_start*

As shown in the code section, at Issue time, we detect in an *always_ff* block whether the issued instruction is a memop (load or store). If so, we increase a counter (*memops_to_sync_start*). Afterwards, when a *sync_start* is received, the counter is decremented, meaning that there is one less memop to be started. The block also covers the case where a reset or a *kill* for the instruction happens, meaning that it will not be *sync_started*. Then, we have the *a_correct_sync_start* assertion checking at every point in time that the counter did not fall to negative numbers. That would mean that the VPU *sync_stated* more instructions than it had been issued.

One problem with this is that the sequences do not have direct access to the interface, causing timing issues. These may happen after the Dispatch sequence sent a *kill*, known by the sequence, but not to the VPU yet. It then may set *sync_start* for this instruction, which has not been killed yet in its queues but is no longer in the Issue sequence.

All these cases must be considered and were difficult to track down. To tackle them, we included a UVM event in the Dispatch driver that got triggered at the same time that the VPU would receive it, so we could have a better idea of for what instruction the *sync_start* was. If the start signal was sent for an already killed instruction, it would get ignored. However, if it was sent for a ready instruction, the operation would be started.

The first point is determining whether the *sync_started* operation was a load or a store. This is needed because they are executed differently and through different sub-interfaces of OVI. For this, we retrieved all the Issue information, encoded instruction, the results, the vector length and SEW, and so on. It is through the OPCODE field of the instruction [23] that the type of operation is determined.

Then, it will be pushed into an "in-flight memop" queue or another depending on the type.

These are inside the corresponding Load/Store sequence, and further specific details will be given in the corresponding 5.3 and 5.4 sections. However, it is worth noting that these structures hold structs with load or store related fields. These include fields specific from each type of operation and some common ones extracted from the instruction arriving from the Issue sequence, which are copied into the struct before pushing it into the corresponding queue. Furthermore, additional instruction-specific information is generated using the corresponding sequences and added to the structs.

Before executing the recently pushed instruction, in the case of a load instruction, we must consider possible data dependencies. Even if these instructions take many cycles to be executed, they usually take an ample address space, making it probable that consecutive instructions have colliding addresses.

The typical case is a real store-load data dependency, in which the load must have data that the VPU has previously stored. In fact, in OVI, we can have loads and a store concurrently in-flight, potentially having matching addresses, generating these issues. To solve data dependencies, we decided to save started loads in an additional queue until all the colliding addresses have been treated for the previous instructions. Once that happens, we move these loads to the actual in-flight loads queue in the Load sequence. Although it is a very coarse grain solution, it was much easier to implement and was more similar to the real case, as the scalar core would rearrange the data appropriately after treating it.

The next step for load instructions is to obtain the data to be loaded into the VPU. This is needed because if we were going to compare against Spike results, we would need its data in memory before executing the instruction. Therefore, this is one of the things that we retrieve from Spike through the Issue sequence. In it, we have a set of functions that calculate the addresses to be accessed during the execution of the instruction and retrieve that data, either for feeding the VPU or comparing its results if it is a store operation.

At the moment of the conception of the environment, we thought that what would fit better our testbench would be simulating a real case. That is, the sequence being the scalar core and having an entity that acted as the cache or memory system. For that, we decided to use a memory model.

An entity model, like a memory or register model, is often used in design verification to simulate the behaviour of the corresponding module and usually to check its correct functioning. In our case, we only needed to simulate the behaviour of a DRAM-like cache. This is due to the scalar core and the VPU exchanging complete 512-bit cache lines through OVI.

The memory model we took as a base is the one from the Opentitan project [31]. It is entirely configurable and we tweaked it to access it with any granularity, so it fits all our purposes. The memory model class definition and its methods can be seen in Code Listing 5.2.

```
class mod_mem_model extends uvm_object;

  typedef bit [63:0] mem_addr_t;
  typedef bit [511:0] mem_data_t;

  // Memory model data
  bit [7:0] system_memory[mem_addr_t];

  `uvm_object_utils(mod_mem_model)

  mod_mem_model_cfg m_cfg;

  // Singleton definition
  static mod_mem_model mem_model_single;

  // Creates the singleton instance
  static function mod_mem_model create_instance(string name = "mod_mem_model");
      if (mem_model_single == null)
      begin
          mem_model_single = mod_mem_model::type_id::create(name);
      end

      return mem_model_single;
  endfunction : create_instance

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

  // Returns the byte contained in address addr in the memory model
  function bit [7:0] read_byte(mem_addr_t addr);

  // Writes a byte in address addr in the memory model
  function void write_byte(mem_addr_t addr, bit [7:0] data);

  // Writes the 512 bits inside data into the memory model in address addr
  function void write(input mem_addr_t addr, mem_data_t data);

  // Writes the number of bits of data specified by sew into the memory model in address addr
  function void write_el(input mem_addr_t addr, logic [63:0] data, int sew);

  // Returns the 512 bits in the memory model starting at address addr
  function mem_data_t read(mem_addr_t addr);

  // Returns the 64 bits in the memory model starting at address addr
  function logic [63:0] read64(mem_addr_t addr);

endclass
```

Code Listing 5.2: Memory model class derived from Opentitan project

In our environment, the memory model is a singleton class, instantiated in the UVM environment and has its instance created whenever used. In Figure 5.2 we have the uses that the Memop sequence gives it in our testbench. The first one is to save the Load data from Spike, so when the Load sequence accesses the memory model for retrieving it, the latter will have the same as Spike used. For this, we store the data in the same addresses of the memory model as the ones we calculated in the Issue sequence.

The second one is to retrieve the data saved during store operations by the VPU. We use a similar method as for the load initializations, where we go over all the previously calculated addresses and read the data inside the memory model. We need this data to compare with the one retrieved from Spike at the scoreboard.

This comparison will be made once the memory operation finishes, once the instruction meets some criteria, which depends on whether it is a load or a store. In OVI it is the scalar core that notifies the end of a memory operation, through the signals seen in Table 5.1; *sync_end*, *sb_id* and *vstart_vlfof*. In a normal case, once the operation with issue *sb_id X* is finished, the scalar core will set *sync_end* to '1', *sb_id* to *X* and *vstart_vlfof* to zero. As said, the criteria above vary depending on whether the memop was a load or a store. These are calculated or determined at *sync_start* time and will be detailed in the corresponding load and store sections.

The Memop sequence will be analyzing whether the criteria are met or not at any cycle during which the memory operation is in-flight. Once they are met, it will create a transaction containing a *sync_end* to '1' and proceed to collect and tidy up all the "belongings" of the finishing instruction. Among these, we find store results, which are first saved in the memory model by the Store sequence. Later, they are collected from the memory model, packed into a transaction and sent through an analysis port to the scoreboard. If everything went as expected, the data inside the memory model will be the same as the one that Spike has for the same addresses.

Moreover, if it was a masked memory operation, at *sync_end* time the Memop sequence retrieves the potential masks for the instruction and packs them into a transaction which is sent similarly as Store results to the scoreboard. There, they are compared to the Spike vector register that contained the mask used for the instruction. This way, we can assert that the VPU, responsible for sending the masks, sent the correct values.

The Memop sequence is also in charge of distributing the incoming mask transactions. This is needed because, as will be explained in Section 5.5, this sub-interface does not have a *sb_id* signal in it. If the memory instruction is masked, at *sync_start* time it is placed into an extra queue for this type of instructions, which will later be used for distributing them. Additional details will be provided about the mask distribution in its dedicated section.

In essence, all the previous are the essential functions of the Memop sequence within our testbench. However, we must consider that many specific cases only arise when executing memory instructions in OVI. These are often related to the Dispatch sub-interface and confirming the completion of the execution of the instructions.

In addition to the previously explained problems that Dispatch timings may cause in the environment, the fact that this information may be sent at any point of the instruction creates a broad set of possibilities for memop execution and completion. It is vital to track

down the possible killing of instructions and obtain the confirmation that the instruction will complete at any point of the execution timeline. None of this is trivial, as we do not have a centralized place where this is considered. The Issue/*sync_start* timing case above mentioned is an example of the problem that this decentralization causes, increasing the environment complexity considerably.

Waiting for kills is mandatory, as these may invalidate already sent Memop related data. However, to ensure the correct behaviour and program order inside the testbench, we must wait for completion confirmation to commit the results of memory instructions. This may not be very relevant for load instructions, as even if the scalar core sent more load or memop data after a kill happened, the VPU would ignore it. However, suppose we want to emulate the actual behaviour with the scalar core. In that case, we must overcome the intra-environment delays between Dispatch and memory operations. On the other hand, Dispatch is a crucial thing to consider for the store instructions. It is similar to the load instructions, as the VPU might be sending additional store-related information. Additionally, we will only commit the stored values inside the memory model once we confirm that the store instruction will complete.

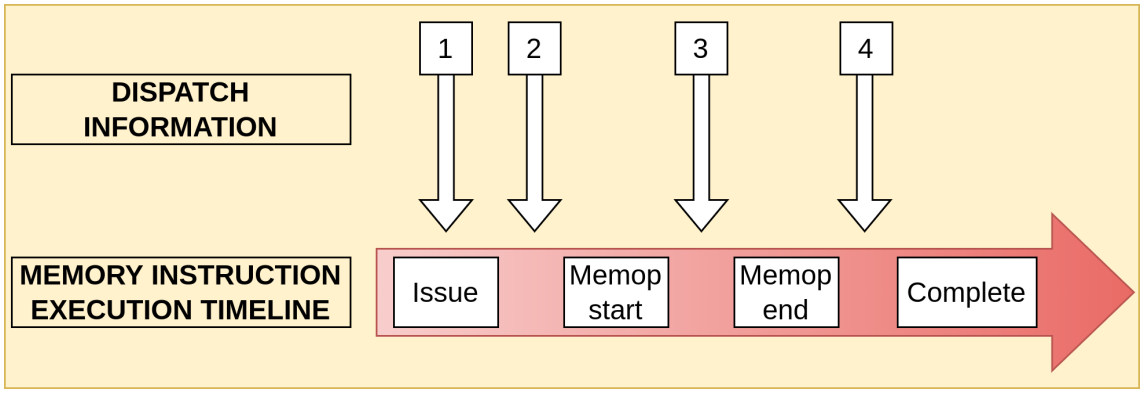


Figure 5.3: Dispatch timing in the memory operation execution timeline

In Figure 5.3 there is a representation of the memory operation execution timeline. This is basically split in three stages; issued, *sync_started* and *sync_ended*. The first stage is delimited by the instruction issue and the *sync_start* of the memop and the second one between the start and the *sync_end* (the memory operation execution itself). The last stage starts after the memory operation is finished through OVI and ends when the instruction completes.

There are four possible time windows to receive the dispatch information for a memory instruction, each having different repercussions in the way these are executed:

1. Issue time: Only confirmed to complete instructions can be sent the dispatch information at the same time as they are issued. Therefore, we only need to pass the *next_senior* information along with the instruction all over the environment. This way, once the memory operation execution finishes, its results may be directly committed.

2. Before *sync_start*: As explained, the memory instructions are stored in the Issue sequence since they are issued and until they are *sync_started*. Therefore, similarly to the previous point, the Dispatch information must be added to the instruction once

the Issue sequence observes it from the Dispatch agent. However, if that happened at *sync_start* time, which is something that the UVM does not have control of, we have the timing problem mentioned above. This forced us to notify the Dispatch information both at Issue and Memop sequences to catch these.

3. Inflight memop: During all the time that the memop is in-flight, the Memop sequence is in charge of detecting Dispatch transactions and marking to complete or discarding as fast as possible outgoing memop related data from the UVM.

4. After *sync_end*: There are some instances in which the VPU might obtain its Dispatch information after the memop has been ended by the scalar core. In our case, this only happens if there is a previous memory instruction to be completed. If this load completes with a *vstart* value different from zero, all the following instructions, including possibly *sync_ended* memops, must be killed, invalidating all the results that they may have provided. This does not affect load operations, but for stores, on the other hand, it is mandatory to detect the Dispatch information and the Memop sequence is also in charge of that.

As previously explained, at the Dispatch driver we introduced a set of UVM events that communicate it with the necessary sequences in the environment. For example, we have a UVM event instantiated in the Issue sequence to track down the two first possible cases. Once the Dispatch driver receives a transaction to stimulate the VPU, it creates another and sends it via triggering the UVM event. When the other side of the event, the Issue sequence, observes the trigger, it can read the *sb_id* inside the transaction and whether it was a *next_senior* or a *kill*. This way, the sequence may mark the instruction structure as next senior or delete it from the queue of memory instructions pending to start.

In addition, we have one event connected to the Memop sequence, which completes the treatment for the second case. We need to have two of these because otherwise, there could be a race condition between the two, leading to possible timeouts or problems by not detecting the Dispatch information. This one is also used for the last two cases, in which the memory operation has safely arrived or even finished the in-flight stage. It is triggered in the same way as the Issue one and the Memop sequence has a similar response. It searches for the *sb_id* in all the possible in-flight memory operation queues, that is, load, store and masked operation queues, and marks them as next senior or deletes them from the corresponding structure. Additional cleaning may be necessary when stopping the execution of a memop, which is specific to the instruction type.

When the criteria are met and a memop is sent a *sync_end*, the Memop sequence will access its next senior bit. It will remove the instruction from the sequence if it is a load regardless. However, if it is a store, this cannot happen until the instruction has been *next_seniored*. As explained, this is necessary as we keep the data to be stored in structures for these operations until we are sure that we are going to commit. Otherwise, if we got a *kill*, we should do a rollback of some data inside the memory model, which would undoubtedly be a big issue for multiple stores in the UVM.

Therefore, when a store instruction finishes, we decided to remove the store struct from the in-flight one and keep it in another "commit" structure. After every clock cycle, this structure is searched to check if any of its instructions has had its dispatch information sent. This is possible because we have the UVM event mentioned above connected to the

Memop sequence, which also looks for the instruction in the new structure.

As said, if the commit structure is observed and has a "dispatched" instruction, it is processed. If it received a *kill*, the instruction is easily removed, as we did not save anything in the environment or commit the data into the memory model. In the case that it was sent a *next_senior*, the store gets its data saved in the memory model and its results and masks sent to the scoreboard. After that, the instruction is finished and can be removed from the structure.

Once we have explored the main capabilities of the Memop sequence, it is time to see how the other three memory-related OVI sub-interfaces work and interact with it within our environment.

5.3 Load sub-interface

The Load sub-interface of OVI is used for the instructions with the same name. Through it, the VPU can obtain data from memory and set values inside the vector registers. The scalar core is in charge of requesting the necessary memory data and sending it through the sub-interface signals along with some metadata and masks.

The VPU does not have direct access to the cache providing the data in the OVI scheme, so the scalar core must retrieve it and forward it to the VPU. In essence, it requests a full 512-bit long cache line, creates a signal that contains the format of the data being sent and sends it together with a previously received mask. As the VPU does not communicate with the cache, the scalar core does not provide the real addresses to be accessed at the memory. Therefore, when forwarding the cache line, the core or the UVM environment must send the exact location of the valid elements inside the 512-bit signal.

In Table 5.2 there are the signals of the load sub-interface.

Signal	Width	Direction	Description
valid	1	Input	Indicates that valid load data is being sent.
seq_id	33	Input	Contains different configurations to order the data being sent.
data	512	Input	Contains memory data to be stored in vector registers.
mask_valid	1	Input	Indicates, for a masked load operation, that the mask being sent with the data is valid.
mask	64	Input	Contains the mask for the current data being sent.

Table 5.2: Load sub-interface signals

For a usual load instruction, many load transactions will be sent through the interface. For each of these, at least *valid*, *data* and *seq_id* signals must take valid values. In addition, in case it is a masked load instruction, each cache line will be sent along with a *mask_valid* set to '1' and accompanied with the corresponding *mask*. For example, for a load instruction with vector length 256 and standard element width of 64, the VPU will receive a minimum of 32 transactions. That is, a maximum of 8 elements in each 512-bit cache line and up to 256 elements, which means that we need 32 cache lines. This is for the case of unit-strided load operations and if the base address of the vector load instruction aligns to 512 bits. Otherwise, we would need an additional line. In some cases, which will be explained later, 256 elements long vector loads could get up to 256 transactions.

The *seq_id* signal contains different fields that together locate and format the valid data inside the cache line being sent. These fields are shown in Table 5.3.

Field	Width	Description
sb_id	4	Contains the identifier it of the load instruction for the current data being sent.
el_count	7	Indicates the number of valid elements inside the cache line being sent.
el_off	6	Indicates the offset in elements of the first valid element of the cache line.
el_id	11	Contains the identifier of the first valid element of the cache line.
vreg	5	Indicates the vector register in which the load data must be stored.

Table 5.3: Seq_id signal fields

With the *seq_id* signal, one can find what the first valid element is in the cache line and where to find it. Afterwards, it is up to the VPU to read the following elements. For example, for the previous case with a VL of 256 and a SEW of 64, the first load transaction would arrive with a *seq_id* of *sb_id*, 8, 0, 0 and *vreg*, respectively. If the base address was not aligned to 512 bits, the *el_off* field may take other values and also will force other fields to change. For instance, if it was aligned at the third possible element in the line, the *seq_id* would be *sb_id*, 6, 2, 0, *vreg*. For the Vector RISC-V ISA, no misaligned accesses can happen. Therefore, the *el_off* field will only take values inside the number of elements that fit in a cache line. In this case, it can take values from 0 to 7.

In the following valid cache lines being sent, the *el_id* field would contain 8 and 6, respectively. In consecutive valid cache lines for the same load instruction, the *el_id* will be equal to the previous *el_id* plus the *el_count*. The previous examples can be seen in Figure 5.4. In it, we can see a representation of the elements inside a cache line and coloured in light blue we have those that contain valid elements for the line currently being sent. On the right, we have the values that the *seq_id* would contain for each of the lines. Example 1 shows the 512-bit aligned load, while example 2 shows the case where the base address is aligned to the third possible element.

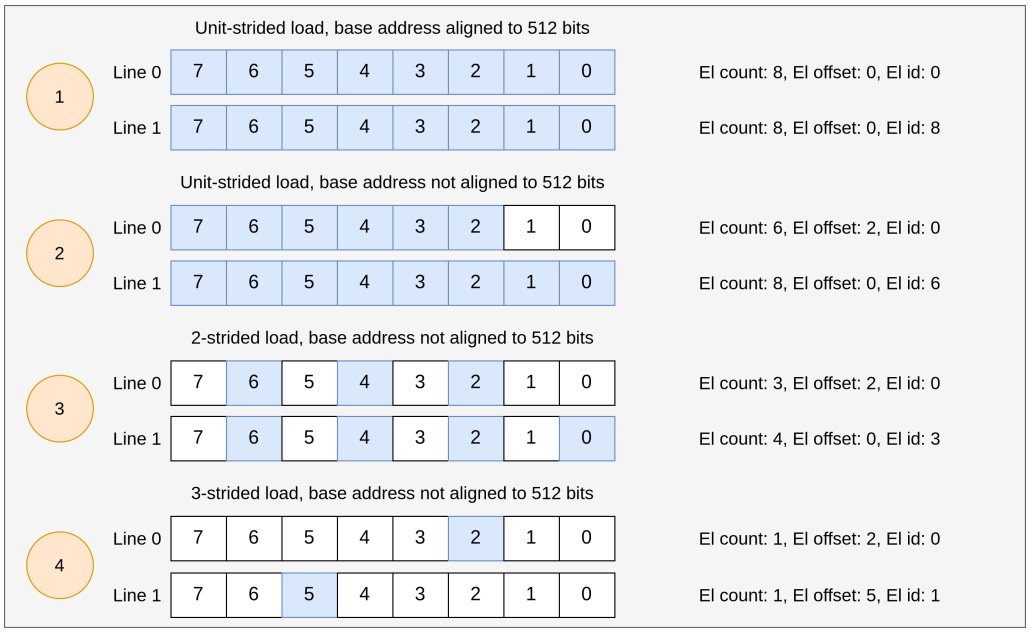


Figure 5.4: Cache line examples for load operations

However, *el_count* taking the maximum value possible every time only happens in unit-strided loads. As explained, this access all the memory positions after a base address. For strided loads, however, the *seq_id* must take different values. For example, if for the case above we used a stride of 2, the *el_count* field would take values equal to 3 and 4, respectively (example 3 in the figure). This would also affect the *el_off* field, which may vary from line to line, depending on the stride.

In OVI, there are two types of strided load operations, optimized and non-optimized ones. This depends on whether the stride used is -4, -2, -1, 0, 1, 2 and 4 or not. If the stride is in the specified group, the instruction can be executed as expected through the interface. However, only one valid element may be sent in each cache line if it is a different stride due to this limitation. Even if there are multiple valid elements inside the cache line, only one will be taken into account in the *seq_id* and marked as valid. This way, one cache line may be sent multiple times to provide different elements to the VPU. If that is the case, each time the *el_off* field will point to a different position of the line, where the current element being sent is. This can be seen in example 4 of Figure 5.4. In it, we can see a 3-strided load aligned to the third possible element. In both lines, the core is sending the same data, but with different *seq_ids*, always keeping *el_count* to one. It is worth noting that in both lines, element 5 contains the same valid data, but it will only be processed by the VPU when the *seq_id* specifies that it is a valid element.

Similarly, for indexed vector load instructions, only one valid element will be sent in each cache line, and the *el_off* field will depend on the previously sent index. This is why it is essential to have memory addresses aligned to SEW in OVI, as offsets must be specified in elements. If the VPU sent an index that set the address to be accessed aligned at two elements inside the cache line, the corresponding *seq_id* would be *sb_id*, 1, 2, 0, *vreg*.

The last aspect of the load interface, as seen in Table 5.2, are masks. For these, two signals are used, *mask_valid* and *mask*. If a load instruction is masked, for every cache line sent, it will set the *mask_valid* signal to one. If this is set, the VPU will take into consideration the value inside the *mask* signal.

Only as many as *el_count* inside the *seq_id* will be taken into account, so the scalar core should be partitioning the masks that the VPU sent it before sending them. For example, suppose the cache line being sent contains three valid elements and starts with the *el_id* 2. In that case, the *mask* signal should only contain the mask related to the following three elements. For instance, it may contain the value "101", meaning elements 2 and 4 of the vector must be written while 3 must not be modified. The same happens with indexed or non-optimized load instructions, where only one bit of the mask may be sent with each cache line, specifying whether the only element being sent in it must be written in the vector registers or not.

Additional details about masks and how they are used in memory instructions will be provided in Section 5.5.

One extra feature that the Load sub-interface in OVI has is that it supports minimal out-of-order capabilities. Given that the scalar core requests complete cache lines, it may happen that its requests hit or miss at the cache. If the cache line containing elements from 0 to 7 was a miss but the one containing the following ones was a hit, the core may send first this last through the sub-interface, with the corresponding *seq_id* signal. This hides possible

miss penalty delay from the vector memory instructions.

In addition, as said, OVI supports up to one in-flight store and two in-flight load memory operations. Due to this, cache lines of different instructions could arrive very close in time or even out of order to the VPU, which is why the *seq_id* signal contains a *sb_id* field.

Additionally, load instructions in OVI have what is called retries. Since the VPU does not have direct access to memory, it will not issue the requests to the cache. The scalar core will do this at a specific rate, which may sometimes be faster than what the VPU can handle, creating some problems. Specifically, the VPU has a structure called Load Buffer, shown in Figure 5.5, which might get full if the cache lines are sent too often or in an order that causes these to get full.

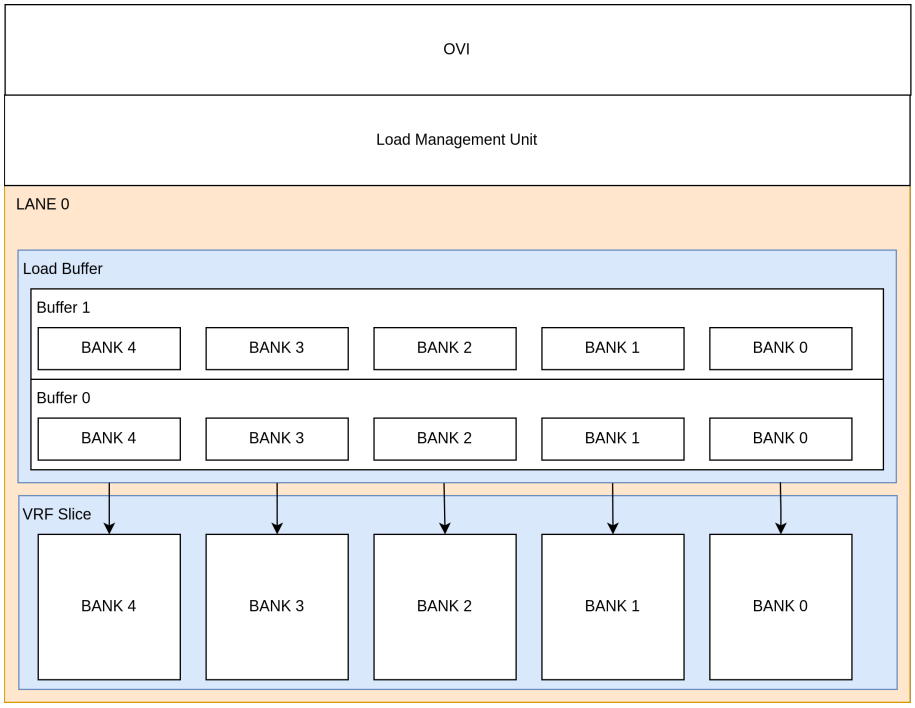


Figure 5.5: VPU Load Buffers structure

As seen in the figure, these structures have the particularity that they receive 64 bits and output 64 bits per bank, that is 320 bits. This happens because there is a Load Buffer per lane and the VPU has eight lanes. Therefore the data coming through the OVI load *data* signal, which is 512-bit wide, is split into eight 64-bit wide signals. To increase the throughput of the module and to exploit at maximum every time the buffer gets to write in the vector registers, it can store up to five elements, one per bank in the vector registers (as seen in Figure 3.5). Only elements that will go to the determined bank can be stored in each of these.

For example, for lane 0, the physical vector register 0 has elements 0, 8, 16, 24 and 32 in banks 0, 1, 2, 3 and 4, respectively. In the optimal case, these elements would come in order and be stored in the buffer banks, to later be stored in the vector registers concurrently, each in its bank. Before that, it could happen that the Load Buffer received element 40. As this element should go to bank 0 in the buffer but is occupied, it cannot be stored and one of them should be discarded, either element 0 or 40. For this space of time, the VPU

has two different buffers to simultaneously prepare two "vector register writes".

However, and considering that OVI supports limited out-of-order capabilities, it could happen that before writing the first full buffer and with element 40 already in the second one, the Load Buffer received element 64. A delay in the VPU could not cause this, as it is a controlled latency. On the other hand, it could happen that the scalar core sent the corresponding cache line before its time, as explained previously. In this case, the Load Buffer will have to discard one of the elements.

When the buffer cannot allocate one of the incoming elements, the VPU will let the scalar core *sync_end* the in-flight load operation. After that, it will notify through the *vstart* signal of the Completed sub_interface the element following the last one that it could save in the vector registers. As previously explained, the scalar core would observe this through the interface and then kill all the following instructions to re-issue the retrying load, this time with a CSR *vstart* equal to the one outputted by the VPU.

The fact that the memory transactions are not sent in a request-response manner creates retries. Even if having to re-issue and kill the following instructions might seem very problematic, it indeed makes the VPU memory-related modules much more straightforward, as they do not have to communicate with the cache. Retries are necessary to recover lost elements. However, they are something that we want to avoid when executing a benchmark or actual application because they introduce a considerable delay in the execution. In our case, as we want to stimulate the maximum possible cases for the VPU, we want to generate retries.

With all this explained, we can see that this sub-interface is among the most complex ones, so its agent and connections are crucial to its correct behaviour.

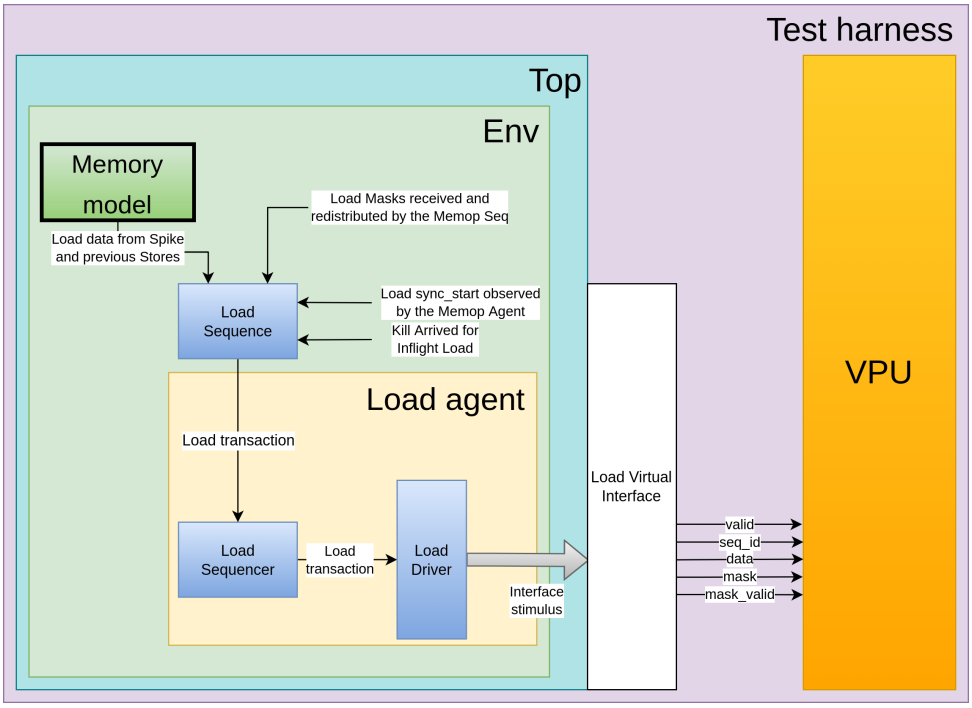


Figure 5.6: Load UVM agent and connections in detail

In Figure 5.6, there is the diagram of the Load sub-interface UVM agent and its connections.

The Load sub-interface is a one-way one. Only the scalar core sends information through it towards the VPU. As explained, each time the valid signal is set to '1', a cache line will be sent along with relevant load metadata. We decided to only include a driver in the agent for these reasons.

As in most sub-interfaces, the sequence generates transactions and feeds them to this driver. We pick up here the thread we were following on the Memop sequence. In it, we explained that after the VPU *sync_starts* a memory operation, the instruction details are packed into a struct and sent to the corresponding specific memop sequence. Thus, a so-called load struct is created and inserted into the in-flight loads structure for the load operations.

The Memop section also mentioned that each memory instruction has its specific necessary data created and inserted into the corresponding Load or Store struct. For the case of the load memory operations, we need different data before sending transactions to the VPU:

- Memory data to provide to the VPU.
- *Seq_ids* that configure the cache lines.
- Masks from the memory operation.

To obtain the first two, we need the data coming from Spike. For the data, we use a particular function in the Spike sequence. In it, we use the type of instruction and the base address to access all the memory positions to be read in the operation to obtain the data inside Spike. In OVI, there are no addresses and the VPU is utterly agnostic to them, but we, as the scalar core side of the interface, need to provide the actual data if we want to sustain the simulation properly. However, as we are using Spike, we can easily access the base address of the instruction, which is stored inside one of the scalar registers and specified in the encoded instruction. With it, we can read any memory position of the instruction.

If it is any strided operation, the positions are accessed starting from the base address and read sequentially after applying the corresponding stride to the address. However, if it is an indexed operation, the addresses are calculated by adding the indexes to the base address. These indexes are located inside the corresponding vector register used as an index vector for the instruction. We obtain them by directly reading the values from the Spike registers. This way, we avoid renaming the physical vector register of the VPU and isolate possible previous failures of instructions that are pending to write in them.

When all the addresses are accessed and this data is ready, it is put into the instruction struct to be loaded into the memory model at *sync_start* time. This way, by the time the load memory operation starts to its execution, the data is present in the memory model and the Load sequence only needs to access it to provide the actual data. The memory model uses the same addresses as Spike. Therefore, when accessing it to send the cache lines, we can use the same as the scalar core. We decided to do it this way because it felt

the most realistic way possible.

The addresses to be accessed are computed in a function called at *sync_start*. In that same function, the *seq_ids* are generated. For this, the Spike sequence follows a similar process. The addresses are walked sequentially, starting from the base address. Indexed loads do not execute this function; instead, they wait for the index to arrive to generate the *seq_id* and the address to access. The scalar core provides 512-bit wide cache lines through OVI, so these addresses will be aligned to 64 bytes.

The first address is obtained in this function by performing a bit-wise AND operation to a mask that aligns it to 512 bits. From there, the memory addresses are walked sequentially. For a unit-strided load, the following address used in the loop will be the first aligned one plus the SEW in bytes. This is necessary because we need to create the *seq_id* and give values to their fields. For that, we must traverse all the valid elements inside each of the cache lines.

For example, let us set take 0x8000404c as the base address for our 64-bit SEW unit-strided load instruction. The first valid element will be the one located at the address 0x8000404c and the second one will be located at 0x80004050. As the first valid element of the line is not aligned to 512 bits, the *el_off* field of the *seq_id* will contain a two. That is, the address of the first valid element minus the 512-bit aligned address divided by SEW. The *el_count* will also depend on the element offset, lowering in this case from a maximum of eight elements to six.

If we had a load instruction with a stride equal to two or four, traversing the whole cache line avoids writing every independent case. If the base address is aligned to 512 bits, every offset and element count in the *seq_ids* will be the same. However, if it is not aligned, each cache line could start in a different element and have a different amount of valid elements. Therefore, if we move from element to element and check whether each address should be accessed or not, the process is easier.

Each time the current address is aligned to 512 bits, a *seq_id* is generated and the previous 512-bit aligned address is saved, meaning that a cache line must be sent with those parameters. This information is stored inside the same queue position in the load struct. This way, whenever a load transaction is to be sent, whatever *seq_id* it is, the first pending one or another one, the *seq_id* and address to access will match.

In both previous cases, as we send complete cache lines, the first address to be enqueued for the instruction will be 0x80004040. The following cache line will be that one plus 64 bytes, 0x80004080. However, if we had a non-optimized strided load operation and more than one valid element in each cache line, these must be sent in different load transactions. This OVI limitation is also covered in the function that implements this algorithm, which creates the corresponding *seq_ids* and enqueues the same address as many times as needed inside the load structure.

Two additional cases to be considered are retrying loads and zero-strided load instructions. For zero-strided loads, the *seq_ids* and addresses must be the same for all cache lines sent, and therefore we provide the same values in each iteration. For retrying loads, the scalar core must provide a *vstart* and then start the load operation from it. For this, we propagate this value from the Issue sequence through the Memop one and use it to gen-

erate *seq_ids* and addresses starting at that point. Instead of starting the algorithm from element zero, we start it from the *vstart* element.

Once all of the *seq_ids* and addresses are inside the load structure and the memory model is filled with the necessary values, the UVM may start sending cache lines to the VPU. It is worth noting that we could do this cache line by cache line, but then we would lose the randomness of choosing a different pending one. If we implemented that in such a manner, the already complex sequence would turn even more challenging.

At this point, every clock cycle a cache line will be selected and sent as a transaction by the Load sequence. This will be done using the *uvm_do_on_with* function, which sends the transaction through the sequencer to the driver. Typically, the *seq_id* and address used are in the first position of the corresponding queues, which are stored in expected operation order. These are popped from the queues inside the load structure and used in the current transaction, meaning that they may not be used again in further transactions. This way, we avoid sending the same cache line twice.

Nevertheless, not always the first pending cache line will be sent. As explained, in OVI a certain out-of-order functionality is supported, which allows *seq_ids* to be sent out of order, intra- and inter-operation wise. The scalar core may send cache lines out of order to hide cache misses in the real case. As we do not have cache misses in our environment, we have to simulate these situations. For this, we use UVM Configurations.

These are UVM that allow increasing the customizability the engineers can get from an environment. By setting a configuration object associated with a class, one can direct the way a test is going or stress a particular design region. In addition, these configurations may be randomized at the start of the test, achieving a broader range of possibilities when simulating with the environment.

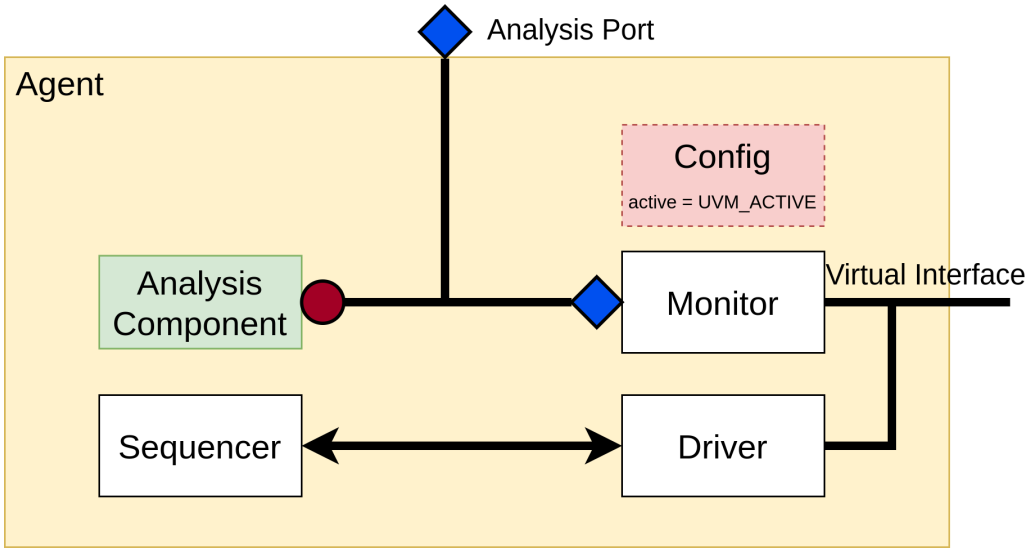


Figure 5.7: Configuration objects in UVM, based on [39]

The configuration object is instantiated inside a UVM component or object. This instance is associated with a configuration object at the build phase or during the environment's set-up. This way, during the test, the component or object may access specific attributes

of the configuration object. These may determine certain properties of the behaviour of the entity.

An example of this can be seen in Figure 5.7. In it, we see an example of a UVM agent with a configuration object "config" associated with it. In this case, the configuration object has an attributed named *active*. This attribute determines whether the UVM agent is active or passive. That means whether it sends and receives transactions or exclusively receives them. This can be interesting for integrating the DUT with other modules feeding the input values. If that is the case, the *active* attribute will be set up as "UVM_PASSIVE", for example, meaning that the driver will not stimulate the interface or that it will not be instantiated. These attributes are often specified in a System Verilog enum construct to be changed more quickly and it is clearer to read the configuration set up.

Another example could be that the configuration object had an attribute called *delay* which determines the delay in clock cycles that it applies between receiving a transaction and stimulating the DUT. Whenever the driver receives a transaction, it can access this attribute through the configuration object and use its content to wait a certain amount of cycles before driving its values to the signals. When setting up the test, this configuration object can be modified or randomized to change the values inside its attributes. This allows the driver to apply a different delay at the interface for each simulation. This would help create scenarios where some buffers in the DUT got full in a random stimulus environment.

We use this UVM feature to create out-of-order Load transactions in our environment. We implemented the environment so that every component and almost every object has its configuration object associated. These are set with the desired values when starting the test and will be further discussed in the following sections. To send cache lines out of order, we have the following configurations.

- Instruction mode: Through this configuration, we enable the chance that a cache line for the second in-flight load might be sent. There are three main modes:
  - Sequential: In this mode, all the first in-flight load cache lines will be sent before starting with the second one.
  - Random: In this mode, every time the load sequence is about to pick one load to send a cache line for, it will randomize the index to use in the in-flight loads queue. This is not a very realistic case but helps in creating very extreme cases which may still be correct and relevant for the VPU.
  - Realistic: In this mode, every time the load sequence is about to pick one load to send a cache line for, it will randomize whether to use the first in the in-flight loads queue or another. The chance of obtaining a different load will be customizable through the instruction change chance configuration.
- Cache line mode: Through this configuration, we enable the chance that a cache line different than the first pending one is sent. For this configuration, we have the same main modes as for the instruction mode, but the indexes obtained will be used in the *seq_ids* queue of the selected load.

- Instruction change probability: When using the realistic instruction mode, this configuration sets the probability of sending a cache line for the second in-flight load.
- Cache line change probability: When using the realistic cache line mode, this configuration sets the probability of sending a cache line different than the first pending one.

Typically, we would set realistic modes for both configurations for most simulations and a low chance of sending an out-of-order cache line, both inter and intra-instruction. We set these configurations to simulate the real case where the scalar core obtained a miss in cache. It will always be approximate and never follow the same behaviour as the core, but by being random, we ensure that all the possible cases happen at some point.

Whenever the Load sequence sends a cache line, it will first select a load for which to send it. The sequence will do this by accessing the corresponding attribute in its configuration object, the instruction mode configuration. Depending on the mode, the sequence will either select the first in-flight load or randomize the load or the fact that a different load might be used. The latter is randomized using the instruction change chance attribute, which may be set in the configuration object. This object is accessed from the sequence and used as a constraint as part of the *randomize* call.

Once an in-flight load has been chosen, the same process is followed to select a cache line of the corresponding instruction. It is worth noting that as these are all independent configurations, they can be used combined and changed; however, we want to create new cases. In addition, we can set them to be randomized at the start of every test, allowing us to test more cases automatically.

In the end, these processes provide us with an index to access first the in-flight load queue and then an index to select the *seq_id* inside the corresponding queue from the selected load. These are what characterize a cache line inside our environment.

After a cache line has been selected to be sent, the sequence is ready to retrieve the data. For this, it will simply do a read to the memory model with the address popped from the queue. A full 512-bit wide cache line will be returned, which is already set to be sent to the VPU. With this, the sequence has the main necessary data to send through the Load sub-interface of OVI. That is, the *seq_id* and the data. Therefore, it creates a transaction using *uvm_do_on_with* and sends it to the driver.

When these transactions arrive at the driver, it will set the values inside into the corresponding signals of the interface. An additional feature that we added to the Load driver involved configurations. If we just left the driver to stimulate the interface as soon as it received the transactions, it would do that every cycle, which is at the rate we generate them. Considering how the Load Buffers of the VPU work, we may stress the DUT too much using this approach. To address this, we decided to use a set of configurations.

We have some attributes of the configuration object of the driver that concern the delay applied to stimulating the VPU. These include modes, such as a random delay mode, a burst mode and a configurable fixed delay mode. The burst mode sends the stimulus in the standard way, one cache line per cycle, which is not realistic but may help cause retries. The second mode randomly selects a delay inside a determined range to stimulate

the VPU with the incoming transaction. Finally, we have a configuration attribute that, together with the fixed delay mode, sets the same delay for every transaction.

With these, we can control the rate at which the environment sends the cache lines to the VPU. Together with the cache line configurations mentioned above, these create new cases when randomized and help create custom and specific cases in particular simulations. For example, an RTL engineer can create a case where the cache lines are sent to fill the Load Buffer modules inside the VPU. This can be done by using a specific combination of *seq_ids* at a very high rate, meaning that the VPU cannot process the operation quickly enough and that it may need to retry the load.

The previous scheme works for unmasked strided loads. However, OVI supports masks and indexed memory operations through combining sub-interfaces. The functioning for both types of operations is similar. The VPU will send indexes or masks through the Mask_idx sub-interface and the scalar core must use them properly.

For the masked strided loads, the environment will receive 64 bits of masks through the interface. Then, the load sequence will partition these to fit the cache line to be sent, considering the element count and the elements already masked. When the necessary mask bits for the line have been selected, they are set inside the *mask* variable of the transaction. In it, the *mask_valid* bit will also be set to one to indicate that a mask is being sent. Then, when the transaction arrives at the driver, the corresponding values in the interface will be set.

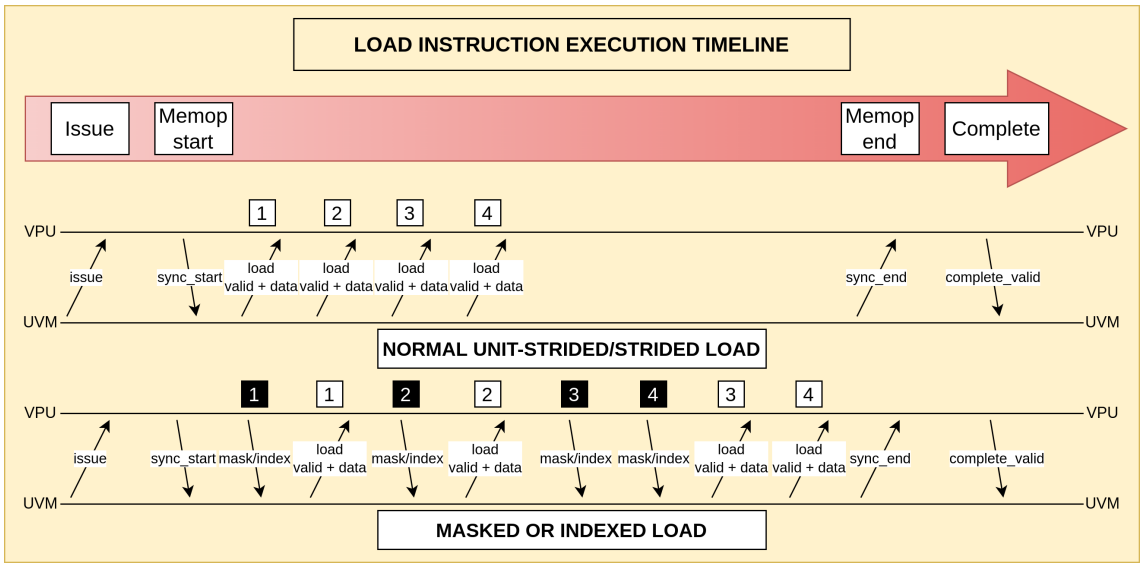


Figure 5.8: Load memory operations flow between the UVM and the VPU

The index will be sent through the same signal of the Mask_Idx sub-interface for indexed loads. These will be sent in the standard order and will determine the offset to the address of one element of the operation. This index will arrive at the sequence, which will apply the offset to the base address when ready to prepare a new transaction. With the resulting address aligned to 512 bits, it will perform a read to the memory model and construct a *seq_id*, which will determine where inside the line the valid element can be found. The data and *seq_id* will then be packed into a transaction and sent to the driver.

Additionally, when an indexed load operation is masked, the most significant bit of the *mask* signal will contain the mask for the element being sent. In this case, the environment will place it inside the least significant bit of the *load_mask* variable of the transaction.

Therefore, the masked and indexed operations force the cache lines to be sent in order. In addition, this means that the cache line must not be sent until it receives the corresponding mask bits or index. Figure 5.8 shows this, where we can see the entire load operations execution flow in OVI. While in a regular load operation all cache lines may be sent directly and in a possibly short amount of time, in the masked and indexed operations, the UVM must wait for the necessary masks to arrive before sending the corresponding cache line. Therefore, the VPU may send the masks optimally or delay the load operation for some cycles, where the core is waiting for the masks to send the cache line. The diagram shows how cache lines 1, 2, 3 and 4 are sent one after another in a regular operation. However, for the masked/indexed load operations, we can see that lines 3 and 4 cannot be sent until both get their mask bits and the UVM sends them through the interface.

In the Memop section 5.2, it was mentioned that for both loads and stores, the Memop sequence checks at every cycle whether the memory operation is finished or not. This is determined by different factors depending on the specific memory operation. For both unit-strided and strided loads, we know that the instruction ended its activity through the interface when all cache lines had been sent. That is, when the *seq_id* queue inside the load struct is empty.

For the case of indexed loads, this does not work, as *seq_ids* are created every time an index is received. To solve this, we use a counter incremented each time a mask or index is received from the Mask_Idx sub-interface. Every time the Memop sequence checks whether the operation has finished or not, it will additionally compare if the mask counter is equal to the vector length of the instruction. If this happens, it means that every element of the instruction got its index and every cache line was sent to the VPU, meaning that the memory operation was finished.

With all the explained, we support all kinds of load operations. Through Spike and the Memop sequence, we obtain the data to send to the VPU and to create the *seq_ids*. We provide different configurations to ease the generation of new situations, like causing retries. These are one of the trickiest parts of memory operations, but the Load sequence simplifies its treatment. First, it executes the first time for the full operation just as the scalar core would. Then, the second time the operation is *sync_started*, it just generates the *seq_ids* and cache lines from the *vstart* position onwards.

Hence, all the load features are supported through the Load sequence in the environment. Let us go over the store memory operations in the following section.

5.4 Store interface

The Store sub-interface of OVI is used for the vector store memory operations. The VPU sends all the data to store in the cache through this sub-interface. As the VPU does not have direct access to the cache, it will send the data to the scalar core, which is connected to the cache.

In Table 5.4 there are the signals of the store sub-interface.

Signal	Width	Direction	Description
valid	1	Output	Indicates that valid store data is being sent.
data	512	Output	Contains register data to be stored in memory.
credit	1	Input	Indicates that a credit is returned, allowing the VPU to send more store data.

Table 5.4: Store sub-interface signals

In this case, the 512 bits of the interface are used exclusively for valuable data. That is, the VPU does not format the cache line in any way. It sends all the elements of the vector that it can fit in the width of the bus every time it sets valid to one. This simplifies much this sub-interface, as no format signals must be sent together with the data. Masks disappear from the signal table as they are directly sent through the Mask_Idx interface.

The signals table shows that the Store sub-interface has a credit system, such as the Issue one. In this case, the scalar core side gives credits and the VPU side consumes them. A credit is consumed whenever the VPU sets the valid signal to one and sends a data burst. Typically, if the vector is long enough, the VPU will send many bursts in a row, eventually consuming all the credits and not sending more for some cycles. Then, after the scalar core has treated the data sent by the VPU, it will set the *credit* signal to one, meaning that it may send another burst of data.

This sub-interface is much simpler than the Load one, but this means that the hard work must be done on the scalar core side. In our case, in the environment and the Store sequence.

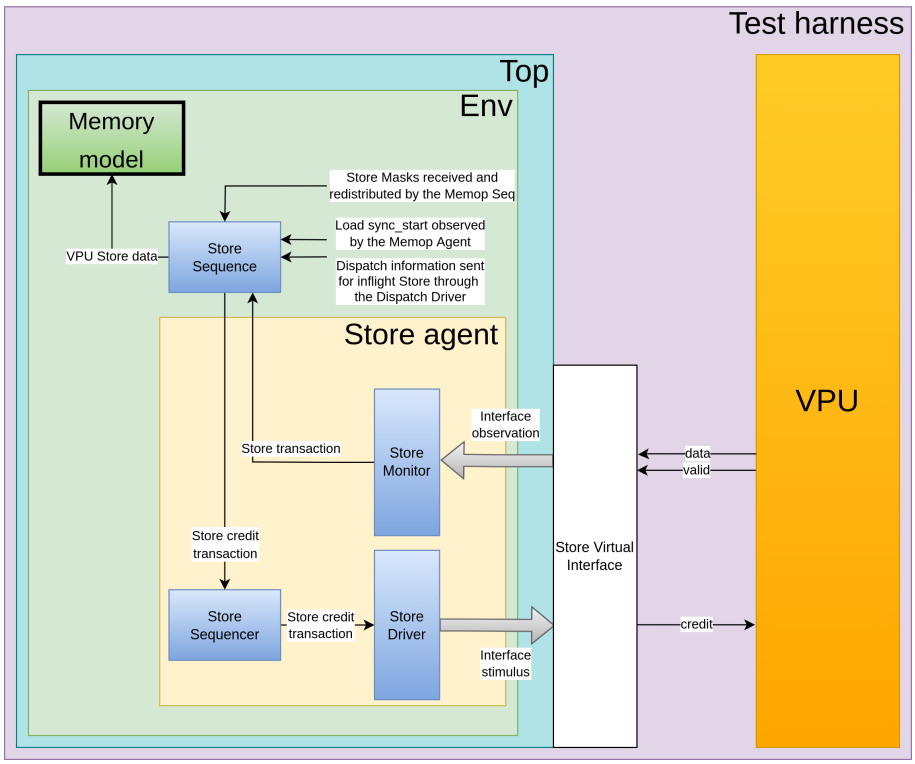


Figure 5.9: Store UVM Agent and connections in detail

In Figure 5.9 there is the diagram of the Store agent of our UVM environment. In it, we can see the monitor in charge of receiving incoming data bursts from the VPU and creating transactions. These are analyzed and treated at the sequence, which produces its transactions to return the credits whenever it is ready. The credit transactions are then sent to the Store driver, which stimulates the interface of the VPU through the *credit* signal.

The main difference between the Load interface and the Store one is that the data comes packed up, taking the total profit of the 512-bit wide bus. This makes it easier to locate where the data is. For example, for a 64-bit SEW, up to eight elements might be sent through the bus each time the valid signal is set. Using the whole width depends on how long the vector is or whether it is the last burst corresponding to that vector. For instance, a vector of 64-bit SEW with a vector length of three elements will only have a burst through the sub-interface, containing 192 bits of valid data inside the data signal. A vector of 64-bit SEW with a vector length of ten elements will have two bursts through the sub-interface; one containing the full bus as valid data and a second containing only 128 bits of valid data.

This fact includes all types of store operations; unit-strided, strided and even indexed stores use the total size of the bus each time if needed. This means, for example, that no blank spaces will be left between valid elements in strided store transactions.

On the other hand, the sub-interface loses other interesting metadata, like *sb_id*. Without it, we cannot know for which instruction the data is sent. The OVI solves this issue by only supporting one store memory operation in-flight at a time. That is, between Memop *sync_start* and *sync_end* and in issue order. In addition, as we have seen *sync_start* does not have a *sb_id* associated. Fortunately, we solved this problem in the Memop agent, which has access to the memory instructions in issue order and ensures that the store data is associated with the proper store instruction.

As there may only be one store in-flight at a time and the UVM as the scalar core is in charge of finishing them, every time it receives a Store transaction through the sub-interface, it knows for what instruction it is. These transactions are saved inside the store struct of the sequence until the end of the memory operation.

Before that happens, if the instruction was masked or indexed, transactions must be received through the corresponding interface. For store memops, masks and indexes can be sent through the corresponding interface at any time. The VPU can send them in different time windows, unlike with load operations, in which they must be sent before the actual data.

These masks or indexes can arrive before, at the same time or after its corresponding element of the vector. This makes the operation possible, as with one store valid many elements may be sent, but in an indexed store, only one index may be sent at a time through the Mask_Idx sub-interface. The scalar core would use these indexes immediately to perform stores inside the cache. However, we need first, as seen in the Memop sequence, to determine whether the instruction is going to be killed or not. For this reason, these masks and indexes are saved together with the data in the in-flight store struct.

All these cases can be seen in Figure 5.10. In it, there is the difference between the execution flow of a regular store and a masked/indexed one. First, we can see the base case

where the mask or index is sent before the data, in which the UVM testbench will automatically consume the store credit. After that, the VPU sends an additional burst of data, for which the mask or index is not ready yet. Once it is received and simplifying the diagram as there should be many other transactions in the middle, the VPU runs out of store credits. This can happen because the VPU did not send the necessary masks for the elements, so the UVM is waiting for these to return the store credits. After the VPU sends the last masks needed, the UVM consumes them and can free a store credit now, which sends using the *credit* signal. At this point, the VPU can send the last chunk of pending store data.

As said in the Memop section, these operations have different criteria that determine that the interaction through the interface has finished. In the case of the store operations, at *sync_start* time and with the vector length and SEW, it is calculated how many store valids must arrive for the operation. Every time data is sent through the interface, a counter is incremented. That counter is checked at every cycle by the Memop sequence to determine whether or not it has reached the previously calculated number. If so, the operation is finished and it is ready to be *sync_ended* by the Memop sequence. In the case of masked or indexed memory operations, all necessary Store transactions must have been received and all the mask transactions. These are also tracked through a counter and compared with the vector length.

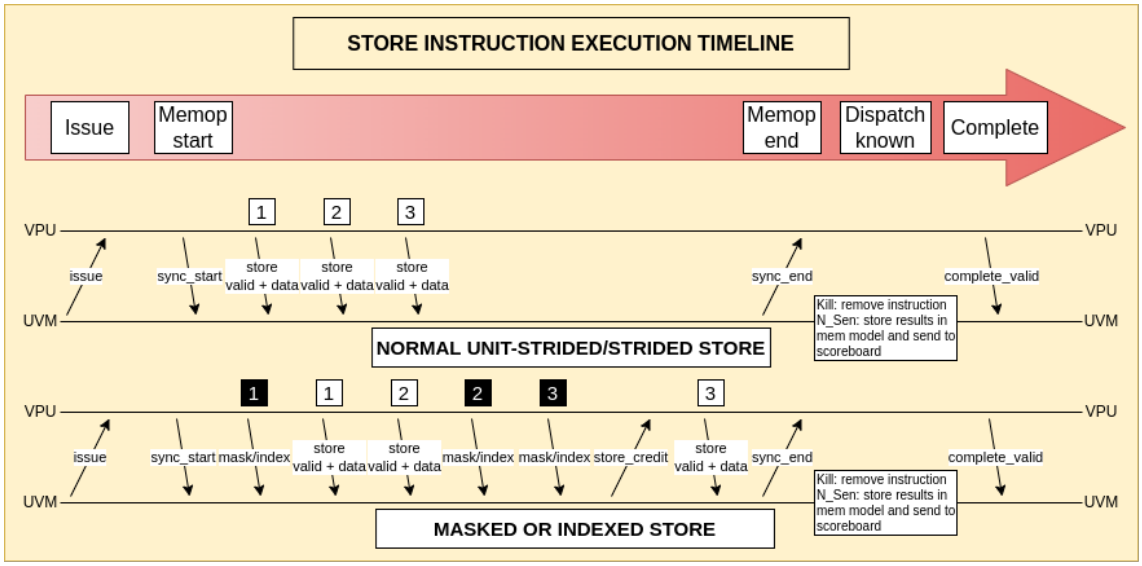


Figure 5.10: Store memory operations flow between the UVM and the VPU

After the operation has been decided to be finished, as explained, the *sync_end* will be sent. However, the treatment of the instruction by the Memop sequence has not finished. First, store data must be written into the memory model. Then, this data must be sent to the scoreboard to compare the instruction results with the Spike ones. As explained in the Memop section, due to a possible pending kill to the instruction, the Memop sequence must have the Dispatch information before committing the store data. Therefore, it saves the finished store struct into another queue until it receives a *next_senior* or *kill*.

When the Dispatch information is determined, the Memop sequence will perform or not the processes mentioned above. This is also shown in Figure 5.10, where "Dispatch known" marks the moment the Dispatch information for the instruction is acknowledged.

That is, because it arrived previous to the *sync_end* or after that moment. As explained, if a *kill* arrives, all data from the store operation will be deleted. On the other hand, if a *next_senior* is sent for the instruction, the store struct will be popped from the commit queue. Then, the addresses for the store operation will be used, either obtained through the indexes or calculated using the instruction type, to store the data inside the memory model.

After that, the Memop sequence will use the addresses used to read the memory in Spike to access the memory model. It will read all the addresses that the operation should have written and create a transaction with the values inside the model. This transaction will then be sent to the scoreboard, which will compare element by element and determine whether the instruction was well executed or not, both by the VPU and the environment. Once the stored data has been sent out of the Memop sequence, all the store operation information can be deleted from the commit structure, finishing all the treatment of the instruction until the Completed monitor observes its completion.

5.5 Mask sub-interface

The VPU uses the Mask_Idx sub-interface of OVI to send masks and indexes of the memory operations to the scalar core. The core will later use these to treat the data from the instructions. In Table 5.5 all the signals of the sub-interface are shown.

Signal	Width	Direction	Description
valid	1	Output	Indicates that valid mask or index is being sent.
item	65	Output	Contains the mask and/or index to be used for an in-flight memory operation.
last_idx	1	Output	Indicates that the mask/index being sent is the last for an in-flight memory operation.
credit	1	Input	Indicates that a credit is returned, allowing the VPU to send more masks.

Table 5.5: Mask Idx sub-interface signals

Similar to the Store sub-interface, this is a tiny one. The VPU will set the valid signal to indicate that a mask or index is being sent through the *item* signal. This is 65-bits wide and is composed of two fields, distributed as follow:

- The **64 lowest bits** contain up to 64 mask bits in strided masked operations. In indexed memops, the index will be located in these bits.
- In masked indexed memory operations, the mask corresponding to each element will be located in the **65th bit** of the signal. In the rest of the operations, this bit will not be considered.

For indexed memory operations, when the VPU is sending the last index for the instruction, it will set the *last_idx* signal to one to indicate that the following *items* sent are for the next masked instruction. Finally, the *credit* signal is used by the scalar core to notify that it is ready to accept new masks or indexes through the interface.

As seen, this sub-interface is not a complex one, as the VPU sends the content of vectors through it and the scalar core only responds with credits once it has used the mask or indexes to treat the memory operation data. It is simple but necessary since the masks and indexes are stored in vector registers. Since the VPU does not have access to the cache and the scalar core is the one in charge of interacting with it, the core must obtain the masks

through the interface. However, this causes a significant slowdown for operations. From the VPU side, store instructions are not always delayed. On the other hand, Loads must have their masks or indexes sent before the scalar core can respond with the cache lines.

Another big problem that this sub-interface presents is that the mask/index items are sent by the VPU without a *sb_id* associated. Once again, they are sent for memops in issue and *sync_start* order. Therefore yes, the Mask_Idx sub-interface is one of the most simple interfaces but at the same time creates issues and complications when trying to emulate the scalar core.

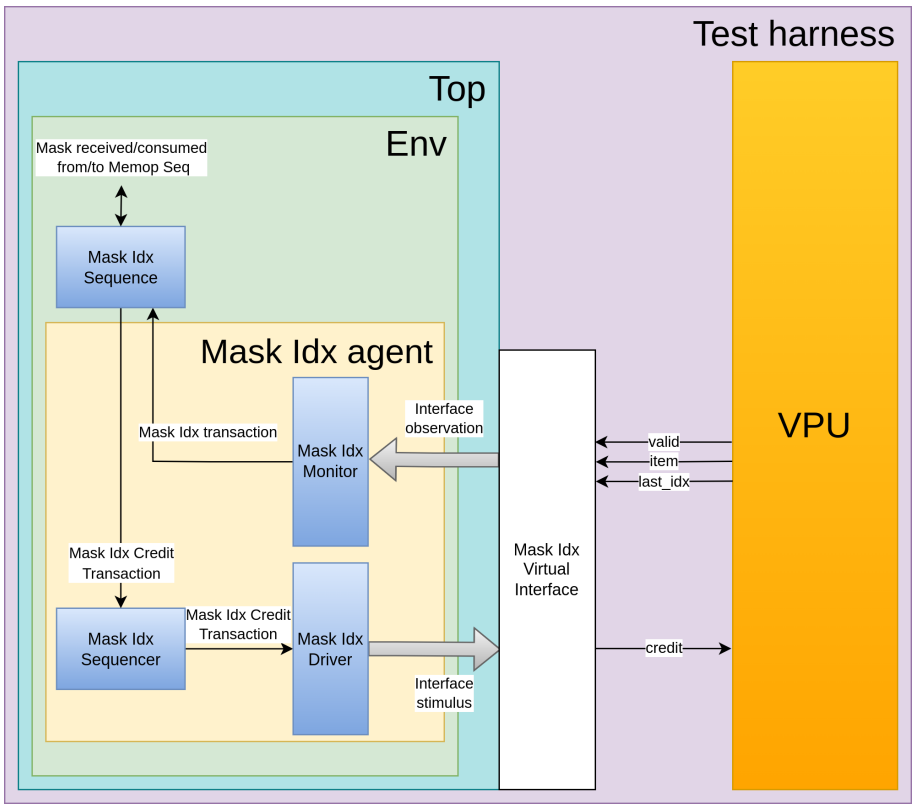


Figure 5.11: Mask Idx UVM Agent and connections in detail

In Figure 5.11 there is the diagram for the Mask_Idx UVM agent. This is similar to the Store sub-interface, but the code related to its sequence is much smaller. The reason for this is that we decided to focus this agent on only distributing incoming *items* and returning *credits*. In addition, this part of the environment is only used when masked memory operations are in-flight, so its behaviour is straightforward and dedicated.

The sequence essentially observes the interface and checks whenever an item is sent through it to store it into a structure. Additionally, every clock cycle checks the content of a counter, which if it is not zero means that a *credit* must be sent through the interface. This last is done by using the *uvm_do_on_with* function.

The main reasons for the Mask_Idx sequence being this simple are two. The first one is that items come through the interface without a *sb_id*, which means that from the sequence, we cannot tell to which memop structure the item will go. The second one is that due to the structure of the environment and the singleton system we use to have access

to other sequences, the Mask_Idx one cannot instantiate the Memop one and at the same time instantiate the Load and Store ones. This is because the Memop sequence already contains an instance of these and there would be more than one instance accessible from the Mask_Idx one. This limitation represents an issue because, from the Mask_Idx sequence, we would want to send the masks directly to their corresponding instructions and see when they have consumed these.

For these reasons, we decided to use the Memop sequence as the central point for the mask and index treatment, connecting it to the rest of the sequences. This duty of the Memop sequence starts whenever a masked or indexed is *sync_started*. As said, the *sb_id* of the memory operation is pushed into a queue. This way, the incoming items will be easily associated with the first element.

After that, we included a task in the sequence that checks every clock cycle whether the structure in the Mask_Idx sequence contains any *item*. It does this by accessing it directly through the Mask_Idx singleton instance. These are steps 1 and 2 of Figure 5.12, in which we can see how the distribution of masks across sequences is done by the Memop one.

If there exists any *item* inside the structure, the Memop sequence will look for the *sb_id* of the first masked memop in the queue (step 3 of the figure). For this, it will access the Store and Load singletons and the corresponding in-flight instructions structures. When found, the sequence will push the *item* inside a specific queue for "pending to treat" items, which finishes the distribution part of the mask duties.

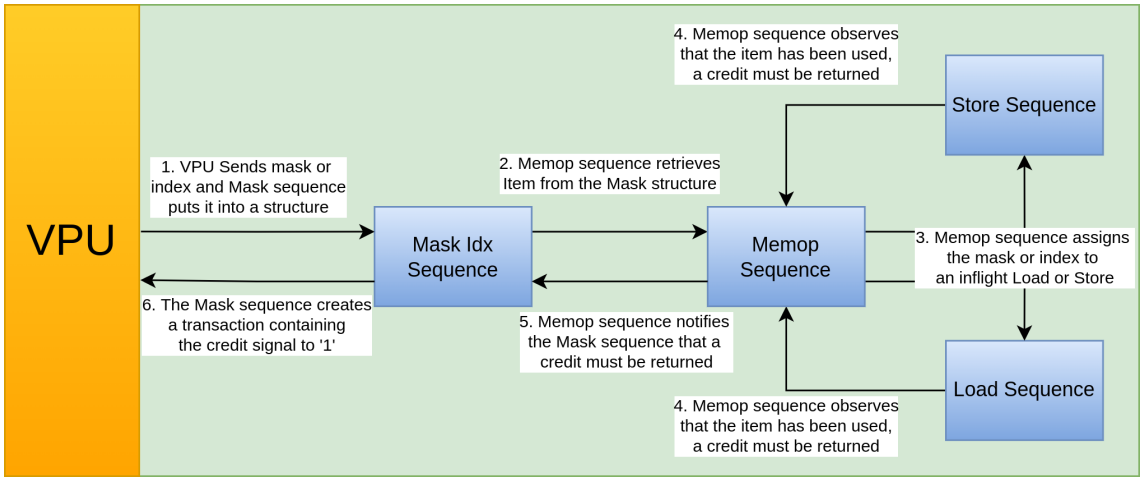


Figure 5.12: Mask distribution and usage in the UVM testbench

From that point on, the Memop sequence will observe these queues. The corresponding instructions will continue their execution, applying these masks and indexes and deleting them from the queues. Once the sequence observes that the items disappear from the queues, it will increase the counter in the Mask_Idx sequence, indicating that a credit may be returned (steps 4 and 5 of Figure 5.12. We do this in this specific way to try and emulate how the scalar core would return the credits in the closest way possible. That is, once these have been used.

```
always_ff @(posedge clk_i) begin
    automatic credits_count_t _credits_aux = _credits;
    if (!rsn_i) begin
        _credits_aux.issue = INIT_CREDITS;
        _credits_aux.store = STORE_CREDITS;
        _credits_aux.mask = MASK_CREDITS;
    end
    else begin
        if (!$rose(rsn_i)) begin
            a_issue_excess_credits : assert(_credits_aux.issue <= INIT_CREDITS);
            a_store_excess_credits : assert(_credits_aux.store <= STORE_CREDITS);
            a_mask_excess_credits  : assert(_credits_aux.mask <= MASK_CREDITS);
        end
        if (issue_if.credit) _credits_aux.issue++;
        if (issue_if.valid) _credits_aux.issue--;
        if (store_if.credit) _credits_aux.store++;
        if (store_if.valid) _credits_aux.store--;
        if (mask_idx_if.valid) _credits_aux.mask--;
        if (mask_idx_if.credit) _credits_aux.mask++;
    end
    _credits = _credits_aux;
end

final begin
    a_issue_credit_return : assert (_credits.issue == INIT_CREDITS);
    a_store_credit_return : assert (_credits.store == STORE_CREDITS);
    a_mask_credit_return : assert (_credits.mask == MASK_CREDITS);
end
```

Code Listing 5.3: Mask and Store credits assertions

To control that neither the testbench nor the VPU did not return or use too many credits, we added a set of immediate assertions that check this for the interface. These can be seen in Code Listing 5.3. These assertions are also inside the *checker* explained in Section 5.2. In this case, we used an *always_ff* System Verilog construct to update a set of counters at each clock cycle. These counters control the number of in-flight credits and are increased whenever a credit is returned and decreased when consumed. Then, they are checked in different assertions, as seen in the code section. In it, there are two groups of assertions; *excess* and *return*.

- *excess*: This set of assertions checked that nor the VPU nor the testbench returned too many credits at any point in time.
- *return*: The *final* construct is run at the end of the simulation time. We control that the VPU and the testbench have returned all credits when the simulation ends with these assertions.

We mostly caught errors in the testbench development through these assertions, where too many credits were returned for store and mask sub-interfaces. Moreover, we could observe cases where the VPU ignored the number of credits it had for both interfaces and sent too many transactions, meaning that we found a bug in the DUT. As seen in the code section, we also had assertions for Issue credits, but these have worked fine since the beginning of the process.

In addition to the previous, the Memop sequence increments a counter for the instruction when an item is received. Once this counter reaches the vector length, meaning that all the necessary items have been sent, the Memop sequence will delete the *sb_id* corresponding to the instruction from the first position of the queue. Therefore, when the next *item* arrives, it will directly find the next memop in the first position of the queue.

Furthermore, every time an *item* is received and distributed, it is pushed into an additional structure inside the instruction. This is done to check whether these have been sent correctly or not. We do this by creating a transaction whenever the instruction is deleted from the memop structures and sending it to the scoreboard via an analysis port. There, all items will be compared to the values that Spike contains inside its vector registers to determine whether the VPU sent the correct values or not.

These mechanisms make it possible to execute masked memory operations while keeping the environment structure as it was first thought. The Mask_Idx sub-interface, together with all the previous ones, allows the execution of all types of instructions through the environment.

Chapter 6

Evaluation of Contributions

In this chapter, an analysis will be done of the previously mentioned contributions and the verification process results. The latter will be detailed in Section 6.1, while the evaluation of the two main contributions itself will be done in Section 6.2.

6.1 Results

The contributions to the VPU verification process have been presented during the last two chapters. The testbench was a team effort, and all members were somehow involved in all its aspects. The team managed to put together a set of features and techniques that exceeded expectations, providing a verification environment for the project.

The two contributions mainly discussed in this document, the UVM testbench and its Memory operation part in detail, were necessary and a central point of the verification process. The primary way the verification of the VPU is performed, though, is by the instructions sent by the Issue sub-interface. These were obtained through Spike, which had a binary loaded randomly generated using Riscv-dv. Therefore, to both send the instructions and support their execution through OVI, especially for memory operations, the UVM testbench played its role perfectly.

6.1.1 Test results

As explained, during months, diverse continuous integration pipelines were set up to run tests on the VPU. Several random tests were generated with Riscv-dv and simulated using the UVM testbench in most of these.

During weeks, we managed to find many bugs in the RTL with this environment. Among these, we include design issues and instruction mismatches. That is, either "timeouts", the VPU not being able to complete the instruction, or a mismatch in the vector registers after instruction completion compared to the expected values from Spike. We also found specification errors in the VPU and the OVI interface document. When an error is found, the necessary information to reproduce it (e.g. binary file or the assembly of the faulty instruction) is provided in an accessible report along with the results with the rest of the set tests.

In Figure 6.1 there is the distribution between faulting instructions in these set of random tests run every night. In the plot, we focus on memops, which are the most interface dependent instructions, but "Others" contains all the rest of the arithmetic/logic vector instructions.

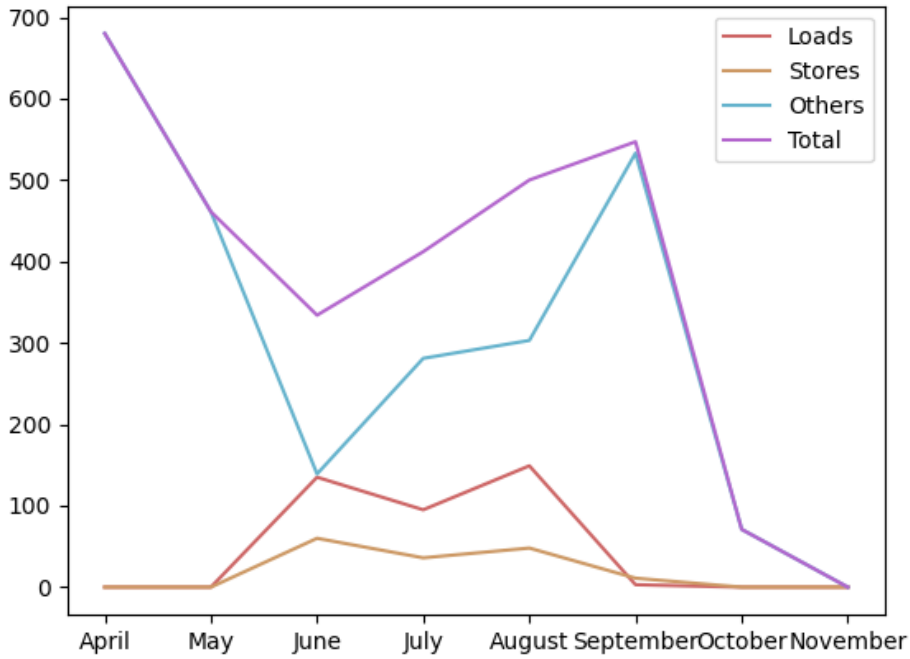


Figure 6.1: Errors encountered per month

These tests include the ones run from April to November of 2020. Twenty-four tests were run every night between April and July, while fifty tests were run nightly between August and November. Afterwards, as seen in the chart, almost no bugs were found, and the CI approach had to be changed. More tests were run from then on, but bugs appeared occasionally and were often solved the day after.

When we analyse the chart in-depth, we can see that almost every test failed at first. This means either that a failing feature was used in every test or that there are many bugs. In our case, it was due to timeouts and the VPU not completing some of the instructions. Once this was solved, much fewer bugs started appearing.

During the first month of CI, we did not have the memop parts of the UVM environment ready yet, so these instructions were blacklisted in the test generator. After May, memory operations were ready and whitelisted in Riscv-dv. These took the top spots of the most failing instructions list for weeks. This was mainly due to timeouts and problems with the OVI protocol implementation in the testbench.

For example, many of the issues in the memory operations involved the VPU not sending the *sync_start* signal for some instructions. In these specific cases, we know that the problem resided in the memory instruction queue inside the VPU and the sending of dispatch information for the operations inside it. Certain combinations of instructions and its issue order caused the VPU not to be able to send the *sync_start* for a pending memop. This caused timeouts in the environment, as it was waiting for that signal and blocking following memops. This also blocks sending dispatch information, as the environment

is waiting for its completion to treat a possible retry. Additionally, no more instruction issues could be done because no credits were returned, which means that other memops could not be *sync_started* and so forth. We detected these timeouts by counting cycles without instruction completion in OVI.

Another bug that we found was concerning the OVI specifications. We found it hard to create the same behaviour as the scalar core for retries. As explained, instructions after loads are not sent their dispatch information to cover the case where these have to be retried. This creates the case where a possible following memop is in the queue waiting to be dispatched but does not necessarily have to wait to be *sync_started*. Therefore, a memory operation might be inflight and sent a kill.

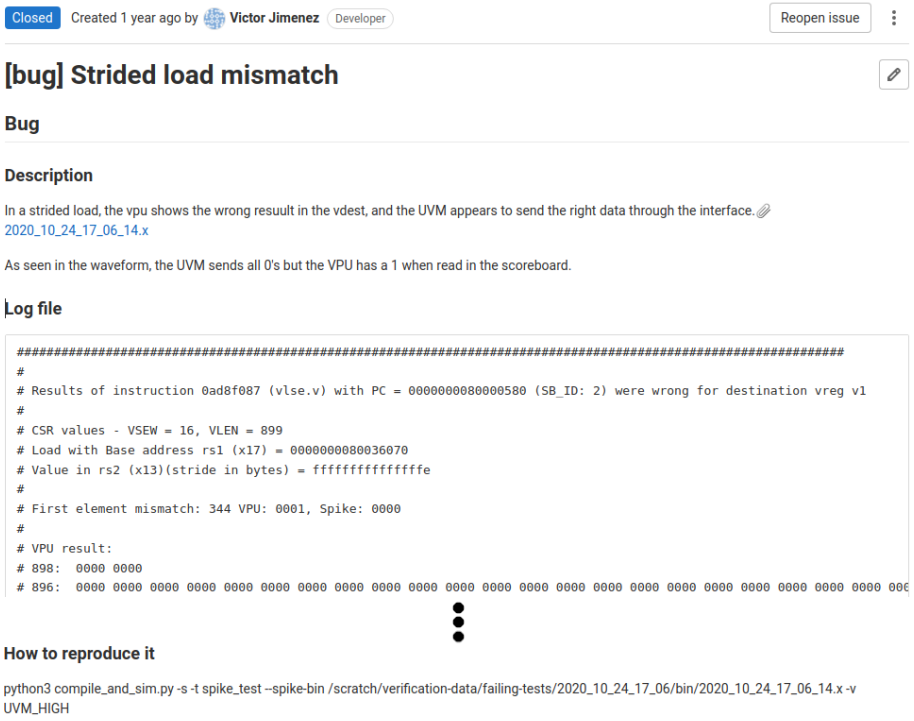
The OVI specifications stated that "a *sync_end* must be sent for all memops that have been *sync_started*", which includes killed instructions. When we were developing the UVM environment, we followed this statement and developed it so it could send the corresponding *sync_end*. However, in the EPAC integrated environment, we could observe that the scalar core was not sending a *sync_end* for these operations. Additionally, the VPU was not waiting for them as all the killed instructions had already been removed from its structures. We asked the team in charge of OVI for this detail, so we could all sync and align to have the same behaviour and they agreed that this aspect of the interface was not clear. The specifications were changed to be more explicit about this, and since version 1.03 of the document, it is stated that it is a *sync_end* must not be sent for killed memops [42].

A different and prolific source of bugs that we found were strided load operations. As said, when issuing a strided memory operation, we needed to send the stride, which is determined by the value inside a scalar register, through the *scalar_opnd* signal of the Issue sub-interface. The Load Management Unit (LMU) from the VPU (seen in Figure 3.4) uses the stride value directly from the interface to determine the alignment of the elements inside the cache lines and feed them directly into the banks.

Three different types of bugs appeared related to this. At first, the stride was not correctly applied by the LMU, causing the vector registers to mismatch with the Spike ones. Later on, we detected a bug where the VPU was writing values inside the registers where it should not for a unit-strided load. The LMU was considering the stride being sent through the interface for this instruction when it should detect unit-stride from the instruction encoding instead, applying the one on the interface to the incoming elements and causing a mismatch. Finally, we also found a bug where while executing two strided loads with different strides, the VPU would not correctly apply the corresponding one to its instruction. The LMU used a direct signal instead of keeping a table with the different inflight strides, which was added after finding the bug. In general, the RTL team quickly detected and fixed all these errors.

As the verification level we decided was for the whole Accelerator, we did not have to dig deeper to find the bugs. In these cases where the error could not be easily seen in the interface, we filled up a Gitlab issue with related information about the bug and notified the RTL team.

As seen in Figure 6.2, we added a short description of the bug, the transcript of the error and the waveform. In the example of the figure, we can see that there was a mismatch in the vector registers after a strided load was executed, where the VPU sets an element to '1'



Closed Created 1 year ago by Victor Jimenez Developer Reopen issue

[bug] Strided load mismatch

Bug

Description

In a strided load, the vpu shows the wrong resuult in the vdest, and the UVM appears to send the right data through the interface.
2020_10_24_17_06_14.x

As seen in the waveform, the UVM sends all 0's but the VPU has a 1 when read in the scoreboard.

Log file

```
##################################################################################################################
#
# Results of instruction 0ad8f087 (vlse.v) with PC = 0000000080000580 (SB_ID: 2) were wrong for destination vreg v1
#
# CSR values - VSEW = 16, VLEN = 899
# Load with Base address rs1 (x17) = 0000000080036070
# Value in rs2 (x13)(stride in bytes) = fffffffffffffffe
#
# First element mismatch: 344 VPU: 0001, Spike: 0000
#
# VPU result:
# 898:  0000 0000
# 896:  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 000
```

How to reproduce it

python3 compile_and_sim.py -s -t spike_test --spike-bin /scratch/verification-data/failing-tests/2020_10_24_17_06/bin/2020_10_24_17_06_14.x -v UVM_HIGH

Figure 6.2: VPU Bug Gitlab Issue example

when all elements sent were zeroes. In these cases in particular, if the UVM is sending the correct data through the interface, which can be seen in the attached waveform in Figure 6.3, it is always an error in the VPU. In addition, we attached the failing binary and the steps to reproduce the bug in the environment.

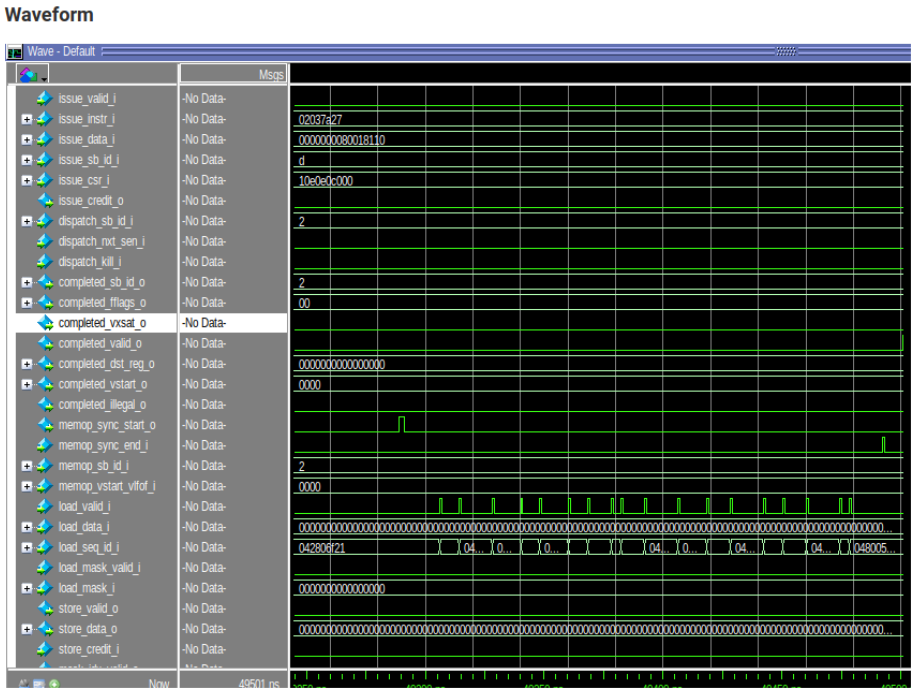


Figure 6.3: Issue waveform example

After we published the issues in the RTL repository, it would be assigned to a person of the RTL team in charge of fixing the bug. In the comments section of the issue, there would be a discussion if something was not clear and between the different members of the RTL team working on the problem. After fixing the bug, they would close the issue and re-run some of the previous tests to ensure that everything was working as expected before merging the changes to the repository's main branch.

As explained and it can be seen in Figure 6.1, we saw a dramatic decrease in the number of bugs found per tests set by the end of June, thanks to finding and fixing the bugs above along with other arithmetic ones. This is why we decided to increase the number of tests generated and run per night. Then, some other features were enabled too, that together with the test increase, mark the growth in the number of errors that we find between July and August.

In addition, masked and indexed memops were enabled during this period, which were blacklisted until this point. These introduced new errors in the UVM implementation of OVI, mainly in terms of protocol. They were causing timeouts and other errors that affected the normal execution of the instructions through the interface. Furthermore, the VPU has a limitation when using masks. All these can only be used by accessing the first vector register *v0* in instructions. However, they are stored in a separate register inside the VPU. Many errors appeared in masked memory operations when we enabled them due to a bug when writing to this mask register in the VPU. After these errors were fixed, almost no errors could be seen for memops in the random simulations.

The last high slope seen in the later stages of the chart in Figure 6.1 was caused by two specific types of arithmetic instructions, widening and narrowing ones. Despite being generated and executed since the beginning of the simulations, it was not until some specific cases were allowed in the generator that they started causing many tests to fail. At that point, very few memops were causing bugs so after the RTL team fixed these errors, we can see how almost no tests failed during October and November.

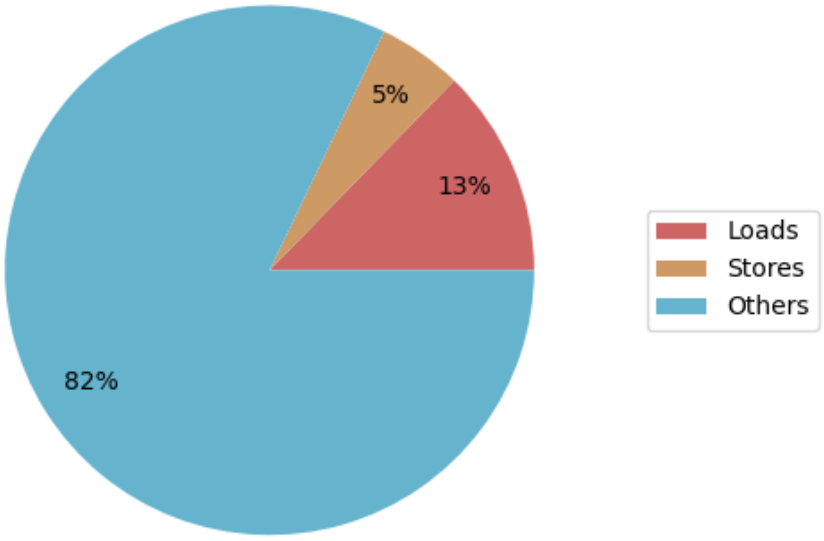


Figure 6.4: Errors distribution among instruction types

In Figure 6.4 we have the distribution of all the failing tests based on the instruction they failed on. Once again, we focus on memops, which take around 18% of all the failing tests. This may not be very impressive at first sight, but this view changes when we assert that there are only six types of memory instructions among the 400 vector instructions the ISA has. This means that only 1.5% of instructions caused more than 18% of the failures in the tests we ran. This demonstrates how important it was to have a good verification process for OVI and that we could catch many bugs in memory operations with our testbench.

It is worth noting also that some of the errors that came out during these simulations were caused by the verification environment, especially at the beginning. In the later stages of the verification process, when we found bugs in the RTL, these were due to new features, as the design was stable and the environment had all the necessary features.

After that point, we could barely see a test failing every one or two weeks, so we decided to change the CI approach and run more tests using the pipelines explained in Subsection 3.4.6. These took much more advantage of randomisation, as we introduced the configurations mentioned in the Memops section, like the load ones. Retries were generated more often, delays were different for every load valid, and so on. This created new cases that could not have appeared previously, causing previously hidden bugs to be detected. Thanks to the simulation tool, these configurations were now randomised along with the test. After the test was executed, the random seed was kept to re-execute the test in the same circumstances.

Unfortunately, we do not have all the tests results of that phase as there were too many to record. However, the design was stable and almost no tests were failing in any case. We can extract the coverage numbers from the current CI infrastructure, as they are still being collected today.

6.1.2 Coverage results

In design verification, we use coverage to determine the success of a verification process, which is how we will determine ours. Even if we managed to find many bugs which contributed to delivering a correct design, if we do not analyse what parts of it we are stimulating or observing, we do not know if we found them all. Many errors could be hidden behind untested cases or unseen conditions.

There are two main types of coverage in design verification; functional coverage and code coverage.

For functional coverage, we implemented covergroups and coverpoints that tracked down this metic in our environment, most of them targeting instruction coverage. As Riscv-dv generates the binaries, we analyse what cases are not generated. We look for generating all types of instructions and all possible combinations of SEW and vector length for these instructions. These may generate new cases each time, especially in memory operations with high vector length and low SEW, which take much longer to execute.

In Table 6.1 we can see the functional coverage achieved for the interface-related modules. Indeed, we did not manage to explore all possibilities of the RTL. However, if we include the rest of the bins outside the scope of this work, we obtained an average functional coverage of **95.79%**. We know that we are not driving the design appropriately in

Design Unit	Coverage
OVI/Pre-issue Queue	**91.95%**
Store Management	**87.50%**
Load Management	**100.00%**
Item Management	**100.00%**

Table 6.1: Functional coverage per design unit, interface-related modules

Type	Bins	Hits	Misses	Coverage
Branches	135279	106251	29028	**78.54%**
Conditions	17908	10353	7555	**57.81%**
Expressions	95290	52349	42941	**54.93%**
FSM States	186	170	16	**91.39%**
FSM Transitions	335	285	50	**85.07%**
Statements	227695	206982	20713	**90.90%**
Toggles	2526053	1258760	1267293	**49.83%**
Total				**72.64%**

Table 6.2: Code coverage for the whole VPU

some cases, which were difficult to implement in the testbench, which is reflected in the coverage numbers. These are memory exceptions and other OVI specific cases that we did not implement. As mentioned, memory exceptions would cause a retry, and although we support load retries through the testbench, we did not get to support store retries. That is why we are getting slightly lower numbers in the coverage of the Store Management Unit and the Pre-issue Queue.

In any case, these are outstanding numbers if we consider it was our first design verification project. Furthermore, we are still running tests up to today, so these keep improving day after day and there are members of the team in charge of analysing the unseen bins.

As we use Riscv-dv for stimulating the VPU, it is our primary source of covering the bins. However, this only includes the instructions issued through the sub-interface, but we have many other sub-interfaces. We have several bins related to them, including *seq_id* combinations in load instructions. These are especially interesting as different alignments in the cache lines are treated differently inside the VPU. Many cases are randomly generated through Riscv-dv but do not appear in the Issue sub-interface.

In addition, we can create more cases using randomisation in the sequences and the configurations. These may be caught in functional coverage using the cover property feature of System Verilog, which we did not use, but they are for sure handy for code coverage. We can create new unseen cases in the interface by randomising delays or other OVI protocol features. With these, we might execute a statement that had not been previously passed or a condition can appear for the first time.

Overall, we have acceptable code coverage numbers, even when looking at the memory-related modules, which may be some of the hardest ones to stimulate. Additionally, we have found that often, when a statement or condition is never reached, it is related to a

problem in the design. Hence, code coverage in simulations helps the RTL team to find impossible-to-reach sections of the code. This is something that the design team does not want to find in their modules, so apart from determining how well we stimulate the design, it may find issues in it. In Table 6.2 there is the detail of the code coverage results for the whole Vector Accelerator. In it, we can see that they are lower than the functional coverage ones, with an average of **72.64%**. This means that either we are not reaching all the possible cases or it could indicate some unused data structure or condition in the RTL. These results are still being worked on, as we need to improve the numbers and test all possible cases, but we have found both of the previous cases.

For both types of coverage, the optimal goal is always 100%. However, that is not possible or achieved in every project. Furthermore, functional coverage is a feature to be specified and implemented, which means that a 100% is not always a guarantee that the design is fully verified. Many companies have different teams working at different verification levels and in different stages of the design production, so they can afford to get a slightly lower coverage number for a particular stage. Moreover, these teams are huge and it is way more difficult to miss a test case.

In our case, we never had that requirement when building the environment. However, we still wanted to provide the most complete set of cases possible, and that is why we added the random features in the testbench on top of the random Riscv-dv binaries. Therefore, considering the available resources and if we state that at least we wrote all the functional coverage that we initially expected, achieving the coverage numbers mentioned above has been one of our greatest successes during the verification process of the VPU.

6.2 Environment evaluation

After evaluating the numbers, it is time to discuss and evaluate the contributions presented in this work. Not only we provided a large environment full of capabilities, but we also learned during the process. We tried to use every possible verification technique to get a taste of all of them and see which fit our needs most. Therefore, we learned many verification techniques in just one project. That said, we created the needed environment for the VPU, which was our fundamental goal. Furthermore, this environment has been beneficial, finding many issues in the design and stimulating it for months.

The continuous integration pipelines still find issues today, although they often are extraordinary and unusual cases. Often, they are related to recent changes in the VPU, out of the scope of the verification process that we performed for months. However, these show that the environment can still be used today and that the pipelines are generating interesting cases and executing useful tests every day. In addition, we are up to today providing support to the environment and implementing features to handle new versions of the design and to address the missing cases detected with coverage.

However, not everything is as good as it seems. Yes, the environment and tools work and provide features to verify the design. It is also true that it is not the most trivial of environments. The initial idea of subdividing the interface turned out to be a challenging way to implement the environment.

When we initially thought of the environment, we did not have the knowledge we have

now about OVI and the VPU. Our idea about OVI was that all sub-interfaces could work independently. We had the full specifications for the interface, but we made the mistake of not fully understanding it. Either that or we did not evaluate well what it meant for the testbench structure that we planned.

For example, one of the main problems we had with the OVI emulation was the *sync_start* of the memop interface. Not only it did not carry a *sb_id* associated, but also, if we wanted to respond to that signal with the Memop sequence standalone, we had no information at all. Additionally, it brings the Dispatch timing problems we discussed in Section 5.2.

The workaround for this has already been explained. We used a set of UVM events and the singleton system described in previous sections to tackle the abovementioned issues. However, this singleton system caused circular dependencies that meant a big challenge sometimes, where we had to think at the best way to give sequences access to the others. The UVM Events set entailed adding communication between agents, which is not the most optimal way to implement the environment.

Since we completed the initially planned verification process, we have reviewed the testbench and compared it to ones directed to similar DUTs. We have seen that all these issues would not appear if the Issue and Dispatch information were present in structures accessible by the Memop sequence. Actually, the fact that many other sequences depend on one, like the Memop one, means that currently, the environment is like a spider net. It does not by any means happen for every feature, but changing any sequence can suppose that the tests do not finish or significant malfunctions in the testbench. In Figure 6.5 we can see all the connections between sequences in the testbench explained in this thesis.

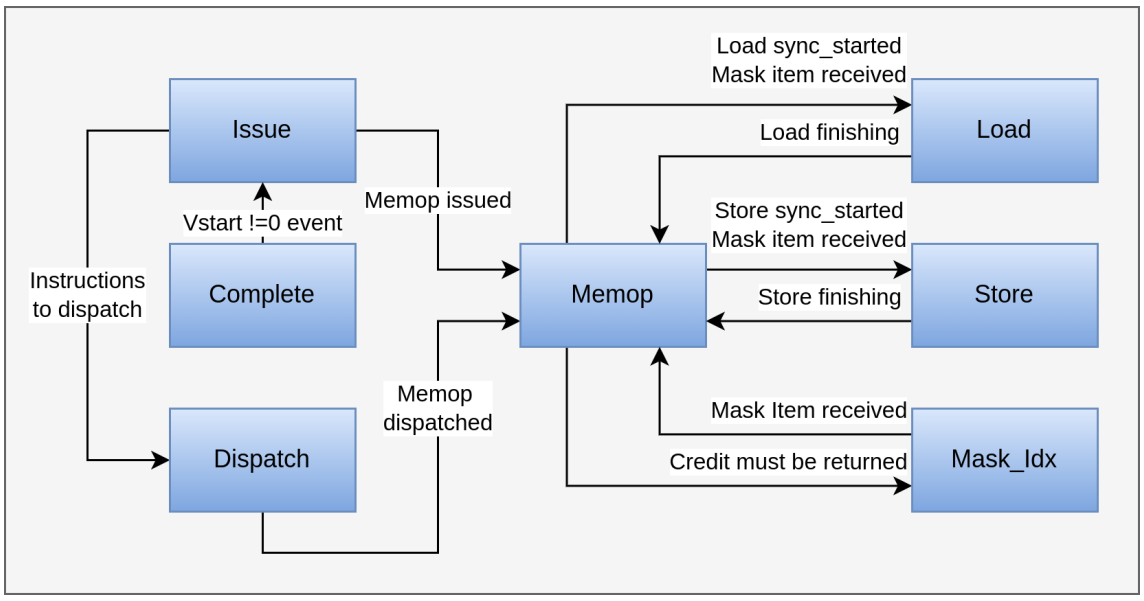


Figure 6.5: Inter-sequence communication in the testbench

A clear example of this issue is mask treatment in the environment. In Figure 5.12, explained in Section 5.5, we can see that the Memop sequence is the center of all the mask treatment. Not only the Mask_Idx interface is not independent of the rest, but the Memop one basically drives it. If this last failed for this feature, no masked memory operations could be executed in the environment. The same happens for almost the entirety of the

load and store instructions execution through OVI.

Therefore, by trying to create a sub-interface divided environment, we ended up reaching completely the opposite, where communicating the sub-interfaces is one of the main problems of the environment. This not only makes it difficult to understand and extend but also to maintain.

The conclusions are similar when we focus on the contributions, the UVM and interface treatment, and the memory operations emulation.

For the UVM and interface treatment, it is tied to what was just described. Sure we can be proud of what we achieved, but it also has several issues and possible improvements. The initial goal of providing random instructions and random independent stimulus only through UVM was not achieved. Nevertheless, we gave it a turn and provided a more "realistic" approach by using Spike as the scalar core. From that point, the sub-interfaces agents do a perfect job of driving the correct values to the DUT and following the execution of the instruction, which is the exact environment we needed.

Certain corner cases cannot be reached by only stimulating the interface through signals and random instructions, but that is why we have coverage. With it, we can identify them and try to generate these cases with configurations and direct tests. Overall, we can be happy with what we managed to do for the interface, as we did a good job considering our previous experience and how ad hoc these environments tend to be.

There are more things to be said when focusing on memory operations emulation. First, we can evaluate it in the same way as the previous contribution. We wanted to make an entirely random environment, but that was not possible in the end. For us, it did not make sense to run a random binary using Spike and then use random values generated by the UVM for the memory operations. Also, most of the problems derived from the structure of the environment come with memory operations. The singleton and delay issues are mostly appearing in the memop related sub-interfaces.

However, what we achieved with memory operations is quite realistic in all terms. We could provide an environment that executed all kinds of instructions, but the OVI protocol is critical for memory-specific ones. We support all memory operation modes and, in addition, provide configuration modes for many of the features to be random and different in each simulation. This is especially interesting for the project because we can execute all instructions and generate a broader range of possible cases, which is really what design verification is about.

Additionally, a particular mention of the whole memory operation flow of the environment has to be made, which was brilliant. The fact that we used a memory model to make cache reads and simulate delays enables many possibilities. Apart from that, we get the data from Spike to correctly execute the instruction, provide the correct data to the VPU and later compare results with such a reference model. Therefore, we integrated Spike in every stage of the execution flow to simulate in the best way possible what would happen if the VPU was in the real environment. By doing that and taking into account how new the team was, we think that we provided the best environment possible in terms of the actual stimulus.

Still, the whole environment has the structure issues mentioned above, which sometimes overcome the benefits. For example, the environment is really slow when using all the features in a simulation and certain combinations of instructions/configurations. We are confident that these performance issues are caused by the massive amount of communication in the testbench. This issue should be solved in further projects or iterations of the VPU, which we are already working on.

When we look at the big picture, though, we are proud of what we accomplished. We managed to provide a very practical and complete environment that, together with other tools, is key in the verification process of the design and helped find many bugs. Furthermore, we have the verification metrics previously discussed, which show through the coverage numbers and the number of bugs found that our environment succeeded in finding a wide range of test cases. In that context, the two contributions described and discussed in this work were crucial, as stimulating the DUT was one of the biggest challenges in the project.

Chapter 7

Conclusions and Future Work

Verification is one of the critical phases in the life cycle of a design. Without a proper verification process, an RTL design will likely go to production with many errors, which will be a massive waste of time and money. Therefore, most companies dedicate a big part of their resources and efforts to verifying their designs. In this work, we have presented a different scenario, where the objective was achieving the most complete verification environment possible with a minimal amount of resources.

In this thesis, we have described the implementation of a verification environment for a RISC-V Vector Accelerator. It can provide instructions to the accelerator and handle their execution through the OVI as if it was the scalar core connected on the other side of the interface. The UVM environment that does this has a very complex structure that allows the treatment of each OVI sub-interface in independent UVM Components. This structure is achieved by using different UVM agents and virtual sequences. The testbench is also capable of comparing the results of the vector instructions and other events of the VPU thanks to a UVM Scoreboard. Moreover, this environment is embedded within a set of tools that together complete the verification process of the accelerator.

We have gone into detail about how the interface is handled in the environment in this work. To begin with, we have described how the general OVI sub-interfaces are and what challenges they suppose to the verification engineers in charge of stimulating such design. After that, we have shown the dedicated UVM agent built for each of them and how it fulfils each sub-interface special needs. Finally, and to complete the execution flow of all kinds of instructions, we have given details of how the memory operations are handled in the environment. From Issue to Complete, our environment provides full support for the VPU to execute the instruction, always ensuring that the final results are the same as those from our reference model.

This testbench was done by inexperienced verification engineers who delivered an environment up to expectations. Although we had no previous experience in verification, we had the correct lead and references to provide a thriving environment. We took the challenge with willingness and knowledge of what was necessary to verify the design, so we designed an environment and a set of tools that allowed us to make it come to terms. Furthermore, we made what we thought was the best option to manage the complex interface. We went straight for a testbench for the whole design, which we considered the

best way to take advantage of all our resources.

However, some criticism can be made as it is true that the given environment could have been better. As explained in previous chapters, the environment is complex, making it hard to expand or sustain whenever a feature is added to the VPU. This complex structure is also causing performance issues, as UVM is not supposed to have so many components waiting for each other and there is no trivial way of solving these issues without changing the whole environment. We want the RTL team or anyone using the testbench to feel comfortable while simulating the design. These are problems that need to be solved, as we would like to explain how it works quickly and make it run much faster.

In any case, we can be overall happy with our work. We emulated the scalar core side of OVI most realistically and broadly we could using UVM and provided checks to verify the design. Through those, we managed to find plenty of errors in the design. This project also allowed us as juniors to read about and use almost all verification techniques big companies use. As the team was new, we could build ourselves a way of working. Therefore, we managed to deliver a successful verification environment and had the chance to learn and become real verification engineers during the project. To conclude, we can say that we are very proud of ourselves and how we managed to live up to the expectations to create a complete verification environment.

7.1 Future Work

As we mentioned in Section 6.1, there are certain missing features in the testbench, like Store exceptions. These must be implemented in the future to support these kinds of events, as the VPU might find them when connected to the Avispado core. Another case we did not explore in the interface is context switches on the core side. These would provoke the saving and posterior restoring of vector registers in memory. As Spike solves this kind of events on its own, we would need to generate and issue the necessary instructions to see this case in the interface.

For future versions of the VPU and OVI, we must also adapt our testbench to handle more than two in-flight loads. This is a current limitation of the VPU and the testbench that might be an issue in the future.

Another feature being considered is adding a cache model to emulate the real case of load instructions. This way, we could model real cache misses and the scalar core side sending a different cache line than the expected one.

Additionally, during the verification process of the VPU and after reviewing our work, we have learned that we could do it better and will do so in future iterations. The BSC plans on using the VPU for many different projects in the future, so taking this into account, we will need to improve the environment as soon as possible. As explained, many of the problems that we have in the environment are due to its sub-divided nature. If we managed to centralize all OVI treatment into one entity, we would avoid most communication in the environment.

For this, we are working on a revision of the testbench. Our idea is to use what is described in section "Bidirectional Protocols" of the "UVM Cookbook" [17]. In it, the virtual interface is replaced with one in charge of responding to the interface and executing its

protocol. Suppose we use only one UVM agent to send vector instructions and observe its completion. In that case, we get rid of all the previous inter-agent communication and the issues it was causing. Additionally, using this approach, we may end up in a more extendable and reusable environment, as the whole testbench may be independent of the actual interface of the DUT.

Finally, a kind of tool we did not consider that is widely used in industry is formal verification [26]. This uses vendor tools to disprove the correctness of assertions more mathematically. Instead of simulating hours and hours of different tests, the formal tools apply every possible value inside a previously defined range to the property and check whether or not it is broken for any of them. This is done using a mathematical algorithm that tries to disprove the correctness of the assertion, which is much faster than generating stimulus at the module's interface, as we did. At the time of the development of this project, we did not have many licenses of formal tools and we were not used to working with them, so we decided to leave them out of the verification process. Nevertheless, it is a really interesting methodology and we would like to integrate it more in future projects.

Bibliography

[1] European Processor Initiative (EPI). *EPI EPAC1.0 RISC-V Test Chip Taped-out*. 2021. URL: https://www.european-processor-initiative.eu/epi-epac1-0-risc-v-test-chip-taped-out/ (visited on 09/20/2021).

[2] Accelera. *UVM (Standard Universal Verification Methodology)*. 2012. URL: https://www.accellera.org/downloads/standards/uvm (visited on 10/02/2021).

[3] Eduard Cerny et al. *SVA: The Power of Assertions in SystemVerilog*. Springer, 2014.

[4] J. Abella et al. "An Academic RISC-V Silicon Implementation Based on Open-Source Components". In: *Conference on Design of Circuits and Integrated Systems (DCIS)* (2020).

[5] John Hauser Andrew Waterman Krste Asanovic. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (visited on 12/02/2021).

[6] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf (visited on 12/02/2021).

[7] asic-world. *Processor production flow*. 2014. URL: http://www.asic-world.com/verilog/design_flow1.html (visited on 08/30/2021).

[8] Search Storage Carol Sliwa. *Seagate, Western Digital outline progress on RISC-V designs*. 2020. URL: https://searchstorage.techtarget.com/news/252493477/Seagate-Western-Digital-outline-progress-on-RISC-V-designs (visited on 01/05/2022).

[9] chipsalliance. *rocket-chip*. 2020. URL: https://github.com/chipsalliance/rocket-chip (visited on 09/19/2021).

[10] Greg Tumbush Chris Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2012.

[11] Intel Corporation. *Intel Intrinsics Guide*. 2021. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (visited on 12/21/2021).

[12] OpenHW Group Davide Schiavone. *OpenHW Group CV32E40X User Manual*. 2019. URL: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/index.html (visited on 01/06/2022).

[13] Doulos. *Easier UVM*. 2015. URL: https://www.doulos.com/knowhow/systemverilog/uvm/easier-uvm/ (visited on 10/02/2021).

[14] Federico Faggin. "The Making of the First Microprocessor". In: *IEEE SOLID-STATE CIRCUITS MAGAZINE* (2009).

[15] Oscar Palomar Perez Francesco Minervini. "Vitruvius: An Area-Efficient Decoupled Vector Accelerator for High Performance Computing". In: *RISC-V Summit 2021, San Francisco* (2021).
[16] Google. *Riscv-dv*. 2020. URL: https://github.com/google/riscv-dv (visited on 10/02/2021).
[17] Mentor Graphics. *Universal Verification Methodology Cookbook*. 2016.
[18] OpenHW Group. *core-v-verif: Functional verification project for the CORE-V family of RISC-V cores*. 2020. URL: https://github.com/openhwgroup/core-v-verif (visited on 01/06/2022).
[19] Tom's Hardware. *Huawei's HiSilicon Develops First RISC-V Design to Overcome Arm Restrictions*. 2021. URL: https://www.tomshardware.com/news/huaweis-hisilicon-develops-first-risc-v-design-to-overcome-arm-restrictions (visited on 12/06/2021).
[20] Imperas. *riscvOVPsim - Free Imperas RISC-V Instruction Set Simulator*. 2021. URL: https://www.imperas.com/riscvovpsim-free-imperas-risc-v-instruction-set-simulator (visited on 01/10/2022).
[21] RISC-V International. *Recently Ratified Extensions*. 2020. URL: https://wiki.riscv.org/display/TECH/Recently+Ratified+Extensions (visited on 12/07/2021).
[22] RISC-V International. *RISC-V*. 2021. URL: https://riscv.org/ (visited on 12/02/2021).
[23] RISC-V International. *RISC-V "V" Vector Extension, Version 0.7*. 2019. URL: https://github.com/riscv/riscv-v-spec/releases/download/0.7.1/riscv-v-spec-0.7.1.pdf (visited on 12/02/2021).
[24] RISC-V International. *RISC-V "V" Vector Extension, Version 1.0*. 2020. URL: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf (visited on 12/02/2021).
[25] RISC-V International. *Spike RISC-V ISA Simulator*. 2020. URL: https://github.com/riscv-software-src/riscv-isa-sim (visited on 10/02/2021).
[26] Yuji Kukimoto. *Introduction to Formal Verification*. 1996. URL: https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html (visited on 12/05/2021).
[27] lowRISC. *CVA6 RISC-V CPU (Ariane)*. 2021. URL: https://github.com/lowRISC/ariane#ariane-risc-v-cpu (visited on 01/07/2022).
[28] lowRISC. *Design Verification Methodology within OpenTitan*. 2019. URL: https://docs.opentitan.org/doc/ug/dv_methodology/ (visited on 01/06/2022).
[29] lowRISC. *Ibex RISC-V Core*. 2019. URL: https://github.com/lowRISC/ibex (visited on 01/06/2022).
[30] lowRISC. *lowRISC Site*. 2021. URL: https://lowrisc.org/ (visited on 01/07/2021).
[31] lowRISC. *Opentitan Memory model*. 2019. URL: https://docs.opentitan.org/hw/dv/sv/mem_model/README/ (visited on 11/20/2021).
[32] lowRISC. *Opentitan project*. 2019. URL: https://opentitan.org/ (visited on 09/19/2021).
[33] OpenHW Group Michael Thompson. *OpenHW Group CORE-V Verification Strategy*. 2019. URL: https://core-v-docs-verif-strat.readthedocs.io/en/latest/ (visited on 01/06/2022).
[34] Onur Mutlu. *Computer Architecture Lecture 8: SIMD Processors and GPUs*. 2017. URL: https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture8-afterlecture.pdf (visited on 12/26/2021).
[35] Mauro Olivieri. "EPI: Accelerator Tile". In: *HiPEAC 2020, Bologna* (2020).

[36] PULP Platform. *RI5CY*. 2020. URL: https://github.com/openhwgroup/cv32e40p (visited on 01/07/2022).
[37] UC Berkeley Architecture Research. *RISC-V Torture Test*. 2012. URL: https://github.com/ucb-bar/riscv-torture (visited on 01/17/2022).
[38] Siemens. *Questa Advanced Simulator*. 2021. URL: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/ (visited on 10/02/2021).
[39] Manish Singhal. *UVM Configuration Object Concept, Figure 1*. 2015. URL: https://learnuvmverification.com/index.php/2015/07/22/uvm-configuration-object-concept/ (visited on 11/26/2021).
[40] SemiDynamics Technology Services SL. *AVISPADO - VPU Interface (OVI Specifications)*. 2019. URL: https://github.com/semidynamics/OpenVectorInterface/blob/master/open_vector_interface_spec.pdf (visited on 09/20/2021).
[41] SemiDynamics Technology Services SL. *Semidynamics: silicon design and verification services*. 2021. URL: https://semidynamics.com/technology (visited on 09/20/2021).
[42] SemiDynamics Technology Services SL. *Updated to v1.03: a kill signal precludes a sync_end*. 2021. URL: https://github.com/semidynamics/OpenVectorInterface/commit/b91a72daa4dd8a295ea1f016038a3bf6adc77f93 (visited on 12/05/2021).
[43] SymbioticEDA. *RISC-V Formal Interface (RVFI)*. 2021. URL: https://github.com/SymbioticEDA/riscv-formal/ (visited on 01/10/2022).
[44] Syntacore. *SCR1*. 2020. URL: https://github.com/syntacore/scr1 (visited on 09/19/2021).
[45] Mentor Graphics Verification Methodology Team. *Coverage Cookbook*. 2013.
[46] Techradar.pro. *Alibaba's new 16-core CPU will challenge Intel Xeon in datacenters*. 2021. URL: https://www.techradar.com/news/alibabas-new-16-core-cpu-will-challenge-intel-xeon-in-datacenters (visited on 12/06/2021).
[47] Chip Verify. *UVM Scoreboard*. URL: https://www.chipverify.com/uvm/uvm-scoreboard (visited on 10/02/2021).