POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Tesi di Laurea Magistrale

A Vector Processing Unit implementation for RISC-V Vector Extension: Functional Verification and Assertions on submodules

Supervision:
Prof. Luciano Lavagno (Politecnico di Torino)
Prof. Francesc Moll Echeto (Universitat Politecnica de Catalunya)
Dr. Nehir Sonmez (Barcelona Supercomputing Center)

Student:
Luca Valente
matricola: s257373

Anno accademico 2019-2020

A mio nonno Costantino.

Acknowledgements

I would like to express my gratitude to Professors Luciano Lavagno and Francesc Echeto Moll, for giving me the opportunity to conduct a work on a topic I am very passionate about. Furthermore, I would like to thank my supervisors Nehir and Oscar for their continuous support and guidance during the last six months.

As I see this Master's thesis as the culmination of the last two years, I would like to thank all my friends, that made this period much more terrific than I would have ever imagined.

This last paragraph will be in Italian, to be understood by the people it is addressed to. Per ultimo, vorrei ringraziare la mia famiglia per il continuo sostegno, i consigli, e il supporto delle mie scelte. In particolare mia madre, mio padre, mia zia Leonarda e mio nonno Costantino.

Abstract

Power dissipation and Energy consumption of digital circuits have emerged as important design parameters in the evaluation of microelectronic circuits. This has led electronic architects to value Parallel Architectures that allow to perform many calculations, or the execution of several processes, simultaneously: exploiting data parallelism reduces instruction bandwidth, reduces required memory bandwidth and lowers the power consumption. A classical example of this trend is the 'Single Instruction, Multiple Data (SIMD)' Instruction Set extension which is nowadays implemented in the majority of Instruction Set Architectures.

Developing Parallel Architectures that can deliver good performance without the energy and design complexity of higly Out-of-Order superscalar processors is of great interest: Data-parallel applications have had a huge impact not only in high-performance scientific computing but in a diversity of domains such as financial analysis, data-processing, machine learning and many others.

An elegant implementation of a Parallel Architecture are Vector Processors: the two main features of Vector Architectures are the presence of a vector register file (VRF), where each vector register can hold a large number of elements, and the presence of many deeply pipelined Functional Units (FU) to directly operate on vectors instead of individual data item.

Within the European Processor Initiative (EPI) project, at the Barcelona Supercomputing Center the work is to design a Vector Processing Unit (VPU) to work as a co-processor of a RISC-V based CPU, to allow it to support the RISC-V base vector extension. So far the RISC-V base vector extension is a draft of a stable proposal. It is based on version 0.7.1 which is intended to be stable enough to begin developing toolchains, functional simulators, and initial implementations, though will continue to evolve with minor changes and updates.

This Master's Thesis work has been carried out during an Internship within the Verification Team for the VPU developing. Firstly, we explored the RTL design of the project to understand the behaviours of the whole architecture. As a second step, the goal was to develop solid specifications for some of the submodules of the VPU, observing their desired behaviour in response to the different Vector Instructions. In particular, the work focused on the communication system to move data within the VPU, called Ring. Once the specifications were fixed, it was possible to develop an Assertion-based Verification plan targeting the Ring modules: the whole set of assertions and the assumptions formalized during this process was binded to the Ring modules and included in the simulation tests of the whole Architecture, to ease the research of bugs for the Design Team.

In conclusion, it was a 6 months internship that led to several results:

- the generation of the specifications for different submodules of the Architecture,
- the development of different reusable UVM testbenches for these submodules,
- the implementation of Formal Verification on the submodules,
- the realization of directed binaries for the VPU to stress the Ring modules,
- the design and actualization of an Assertion-based Functional Verification Plan able to trace the 36% of bugs (4 out of 11) relative to the Ring in its functioning period.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The work presented here has been carried out in the Verification Team for the European Processor Initiative (EPI) Project which aims to develop a Vector Architecture to work as a co-processor of a RISC-V core, to support the RISC-V Vector Extension targeting High Perfomance Computing (HPC) applications ([9]). The focus of the work has been to develop a verify the Submodules of the VPU designed to move the data within the Architecture. In particular, we implemented an Assertion-based Verification plan that, as expected, brought great advantages in the Verification of the targeted Submodules.

In this first Chapter, the two main concepts that have shaped the work of the last six months will be presented: Vector Processing and Functional Verification. Then, it will be possible to progress further and extensively present the details of the tools used and the work that has been carried out. The rest of the manuscript is organized as follows: in Chapter 2 we are going to describe the framework where this investigation has been carried out, with a focus on the VPU, the Submodules and on the Verification tools. In Chapter 3 we are going to illustrate the work done, while in Chapter 4 the obtained results and discuss them.

1.1 Vectors

Vector Processing is a particular kind of processing that performs operations on arrays and vectors, compared to the classical General Purpose processing that operates only on scalars. Instruction sets with these characteristics are called SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata). During the 70s and the 80s, Vector Machines coincided with Supercomputers. Indeed, implementing an instruction set containing operations on vectors greatly improved the perfomance on the typical workloads of Supercomputer, such as numerical simulation and similar tasks involving huge computations on large data set.

The Cray-1, designed in the 1975 by Seymour Cray, was the first supercomputer to successfully implement the vector processor design and SIMD instructions. For this reason, Vector Architectures have been closely identified with the Cray-1. The two main features that characterized the Cray-1, were the presence of a Vector Register File (VRF) with 64 Element Vector Registers and many deeply Pipelined Functional Units to simultaneoulsy operate on several elements ([14]).

Throughout the 90s, given also the scaling of price and area of microprocessors, the SIMD Instruction Sets became more and more common and are currently implemented in the broad majority of Instruction Set Architectures. Nowadays, SIMD extensions exist of many different types and of different sizes, according to needs of the targeted application: we can change the number and length of Vector Register, depending if the SIMD will be for embedded market or HPC.

As we will see in the next Chapter, the VPU under development is optimized for HPC applications and is inspired by the the traditional Vector Processing architecture. Indeed, it features 32 Vector Registers of 16kbits each, compared to embedded SIMD ISA which may support just 8 Vector Register of 128 bits each.

Generally speaking, the conventional Vector Programming Model presents two main *advantages* with respect to the classical Scalar Processing:

- utilizes deeply pipelind ALUs to operate on multiple elements at the same time, without data hazards.
- when it is possible, is able to exploit the data paralellism of the on-going task, eliminating all the standard branch-control that would be massively present otherwise.

The second advantage means that when we have code that can be executed on a Vector Machine, that execution will be drastically more efficient with respect to the same execution on a scalar core. Also, this implies a reduction of the number of assembly instructions needed to perform a simple computation. For example, if we execute the following C-code:

```
for (i=0;i<64;i++)
        C[i]=A[i]+B[i];
```

we would see a huge difference between the scalar and vector code.

Scalar Code:

```
li x4, 64
loop:
fld f1, 0(x1)
fld f2, 0(x2)
fadd.d f3,f1,f2
fsd f3, 0(x3)
addi x1, 8
addi x2, 8
addi x3, 8
subi x4, 1
bnez x4, loop
```

Vector Code:

```
li x4, 64
setvl x4
vld v1, x1
vld v2, x2
vadd v3,v1,v2
vst v3, x3
```

It is possible to notice immediately that the Vector Code needs just 6 instructions, while the Scalar Code needs 9 instructions, repeated 64 times. Naturaly, this is a great advantage when the code that it has to be executed can actually take advantage of the Vector Architecture. However, it is important to recall that every machine that supports a Vector Extension, needs also to support a Scalar ISA. This is necessary since scalar computation would be extrememly inefficient in an Architecture that supports only Vector Operations.

1.1.1 RISC - V

The RISC-V [3] is an open-source hardware born in 2010 at the University of California, Berkeley as an academic research project. Since that moment the contribution to the project has been growing and growing, setting the RISC-V as a common free and open standard to which software can be ported, and which allows everyone to freely develop their own hardware to run the software. Anyway, the RISC-V International does not make available any open-source RISC-V Implementation, only the standard specifications. This void has been filled with many open-source hardware implementation [2], and many others are coming. RISC-V and these open-source hardware implementation allow the possibility to design RISC-V based accelerators, expanding the Instruction Set with specific Instructions, suggested directly by the algorithm that needs to be performed.

RISC-V is a modular instruction set. The base RISC-V instruction set contains the RV32I that contains 45 instructions or RV64I, containing 57. Then different extensions can be added. Being RISC-V open source, these extensions have been the result of a contributive work of many people and each extension has experienced deep revisions and changes, to eventually freeze. The list of available Instruction Set Extension can be found updated in [10] and is summed up in Table 1.1.

Base	Version	Status	Description
RV32I	2.1	Ratified	32-bit
RV32E	1.9	Draft	32-bit, 16 Registers (embedded)
RV64I	2.1	Ratified	64-bit
RV128I	1.7	Draft	128-bit
Extension	Version	Status	Description
M	2.0	Ratified	Integer Multiplication and Division
A	2.1	Ratified	Atomic Intructions
F	2.2	Ratified	Single-Precision Floating-Point
D	2.2	Ratified	Double-Precision Floating-Point
Q	2.0	Ratified	Quad-Precision Floating Point
L	0.0	Draft	Decimal Floating-Point
C	2.0	Ratified	Compressed Instructions
B	0.0	Draft	Bit Manipulation
J	0.0	Draft	Dynamically Translated Languages
T	0.2	Draft	Transactional Memory
P	0.2	Draft	Packed-SIMD Instructions
V	0.9	Draft	Vector Operations

Table 1.1. Current RISC-V available extension and status

Naturally, among all these extentsion, our focus is on the Vector one, indicated with V in the table. RISC-V is a great fit for Vector Processing. Indeed, RISC-V is designed to support a large address space and has always had powerful Vector Extension as specific goal.

1.1.2 RISC-V Vector Extenstion

The first proposal of a Vector Extension for RISC-V has seen the light in June 2015. Since then, much work has been done to reach a 0.9 version, which has been released only in May 2020 [4]. Nevertheless, the VPU implemented here features the version 0.7.1 of the Vector Extension specifications.

The Vector Extension supports 32 Vector Registers, each one of *VLEN* bits, with Elements of maximum *ELEN* bits. Another important parameter is *VLMAX* which is the number of elements in a Vector and is equal to *VLEN/SEW* (**S**ingle **E**lement **W**idth). This Extension has been designed to allow the same binary code to work across variations in *VLEN*. *VLEN* and *ELEN* can be changed according to the target application. For example, Table 1.2 shows some possible implementations that can been targeted, according to the desired application.

Name	Issue Policy	Issue Width	VLEN[bits]	Datapath[bits]	$\frac{VLEN}{Datapath}$
Smallest	InO	1	32	32	1
Simple	InO	1	512	128	2
InO-Spatial	InO	4	512	512	1
OoO-Spatial	OoO	2-3	128	128	1
OoO-Temporal	OoO	2-3	512	128	4
OoO-Server	OoO	3-6	2048	512	4
OoO-HPC	OoO	3-6	16384	512	32

Table 1.2. Possible Vector Extension sizes ([11])

In order to save the 32-bit instruction encoding, the Vector Extension features a **vtype** register that contains the data of *vsew* (8,16,32,...,1024) and *vlmul*, which is the vector length multiplier. With the instruction *vsetvl* we can encode the information of *SEW* and *VLEN* and pass it to the Vector Architecture ([11]).

The RISC-V Vector Extension has some differences with respect to other existing SIMD extensions:

- **MMX, SSE, AVS, NEON, Altivec**: they have fixed register width in bits. Therefore, every time there is the need to increase *VLEN* a new set of instructions had to be designed and it needs to cope with edge cases.

- **ARM SVE**: differently from RISC-V, it is not agnostic on the value of inactive elements (i.e. all bits of the unvalid elements have to be set 1). Also, it uses masks instead of *vlr* for length.

The RISC-V Vector Extension is a really powerfool tool and even if it might turn out to be expensive to be developed, it has gained much interest in the last years and some implementations already exists at academic level ([12], [20], [22]). Still, while Vector Extension can be very useful, there is still much work to do to develop better software frameworks that can actually take advantage of the custom hardware.

1.2 Verification

In this Section we are going to present the other main ingredient of this Thesis: Functional Verification, Assertion-based Verification and its benefits.

Functional Verification is the task of verifying that the developed design respects the given specification, before the tape-out. Nowadays, Verification is a fundamental step in the Design Process and it is even more resource demanding than Design itself, which requires just between 30% and 40% of the project resources. This is because modern circuit's complexity and technology's shrinking increase the amount of needed work and reduce the productivity of the classical design $\leftrightarrow$ simulate $\leftrightarrow$ debug $\leftrightarrow$ cover loop, leading to a serious growth of the needed resources to verify the circuit and manage the project, as illustrated in Figure 1.1 ([16]).

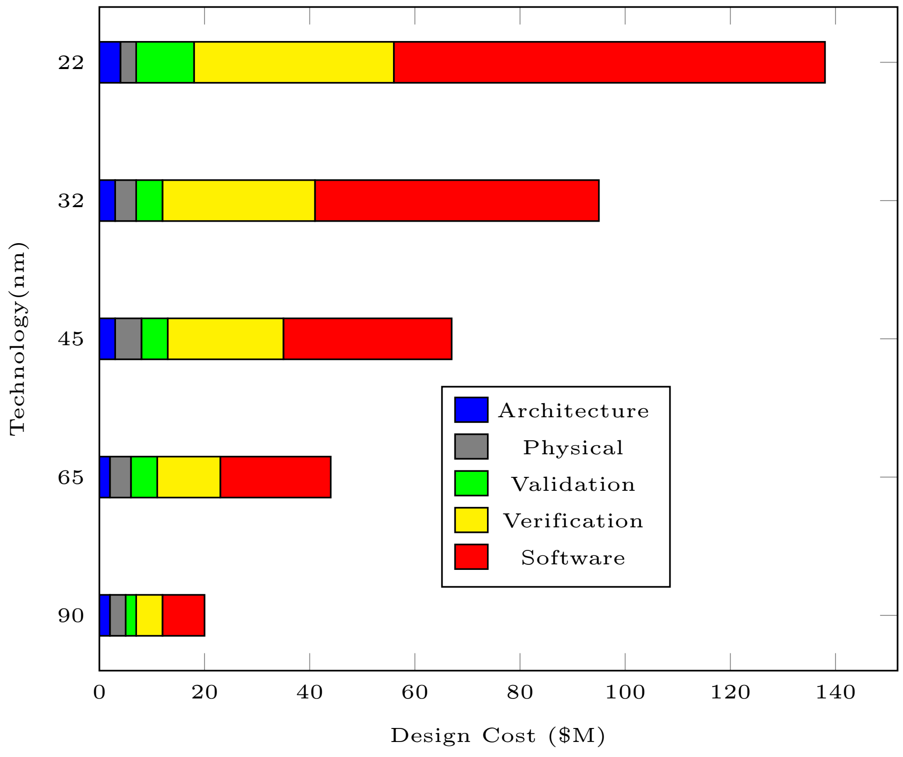


Figure 1.1. Design Cost

It is therefore possible to address the Verification as the bottleneck of the Design Process and every attempt to speed it up comes in useful, reducing the time-to-market. As a matter of fact, the entire Design has to be tested to avoid the realization of a faulty circuit that would cause incredibly high losses. This is why the whole process has to be as much efficient as possible. As it will be discussed later on, the typical functional errors appear due to:

- unspecified functionalities,
- conflicting requirements,
- and unimplemented features.

To find and correct the present Design bugs, two possible approaches can be identified: Dynamic Verification and Static Verification.

1.2.1 Dynamic Verification

Dynamic Verification (DV) is the Simulation-based approach: it is the most widely used approach and the only solution for complex systems verification. Surely, the whole DV process is expensive in terms of resources and time: it is the sum of 4 steps that require a significant effort by the Verification Engineers. The 4 phases are the following ([16]):

1. Development of the verification plan (i.e. defining what Functional Verification is going to be performed), of the testbench and of the tests.
2. Simulation of the tests.
3. Debugging, along with the Design Engineers, the found bugs after the comparison of the Design Under Test (DUT) with the reference architecure.
4. Check that the Design has been tested extensively and sufficiently. To do so, the most used metric is the *Coverage* that can be functional, code or System Verilog Assertion (SVA) coverage and indicates the development stage.

To reduce the time spent in these steps, many powerful EDA tools have been deployed, these tools allow a high level description of the problem, easing the whole process and they will be described in the next Chapter. However, apart from the high amount of work required to properly test a complex system, there are two other crucial limits in the Dynamic Approach: the first one is that simulations run *very* slowly (100 Hz-1kHz), while the second one is that no one can actually guarantee that the design has been *fully* tested, exhausting all verification possibilities for every given piece of logic, even with 100% coverage achieved. To cope with this last limit, Static Verification has been introduced.

1.2.2 Static Verification (Formal-based)

Static Verification is the second possible approach to perform Functional Verification. Static Verification tools try to check that the DUT is properly functioning demonstrating it by mathematical proofs rather than by brute-force simulations. Formal tools avoid any dependency on the user to find possible corner cases. When combined with Simulation-based Verification they prove their power as Verification tools.

Naturally, Static Verification is not a perfect fit for every kind of DUT. Given its high computational cost, it is particurarly effective for small size DUT that do not perform much arithmetic computation or that have a complex sequential nature.

Typically, the flow for Functional Static Verification is made of 3 steps ([16]):

1. Write the set of properties of the DUT that are necessary to be checked, in System Verilog Assertion (SVA) language (or other languages).
2. Provide the Formal Tool with the set of properties. At this point, the tool will derive a model for the RTL under test and apply *all possible* “stimuli” to verify that the properties will not fail under *any* circumstance.
3. If a bug is found, the tool will provide the input sequence that led to error to make the debugging possible.

Given a DUT, a property can be defined as the description of the relationships between some signals and how it evolves through time. To do so, System Verilog Assertions (SVA) is the most common language. For example, if we have a DUT that when receives a signal *req_i* equal to 1 has to return *grant_o* within a clock cycle, we can easily express it with the following property:

```
property handshake;
        req_i |-> ##1 grant_o;
endproperty
```

Once a property is defined, it can be **Asserted** or **Assumed**, making up an **Assertion** or an **Assumption** respectively.

The idea behind static verification is that, with simulations, the DUT is repetitively brought in the same states and more rarely in corner cases. Therefore, it tries to find all the possible states of the DUT and all the possible applicable input stimuli. If all the states are bug-free then the module is bug-free, otherwise or there are wrong assertions or bugs. This idea can be described with Figure 1.2: there is an initial state and the the whole set of states where the DUT can be, the goal of Static Verification is to find all of them and prove the absence of errors.

The outcomes of the Formal process on an assertion can be different (Figure 1.3):

- assertion **Proven**: it does not exist a legal stimulus that can cause the assertion to fail.

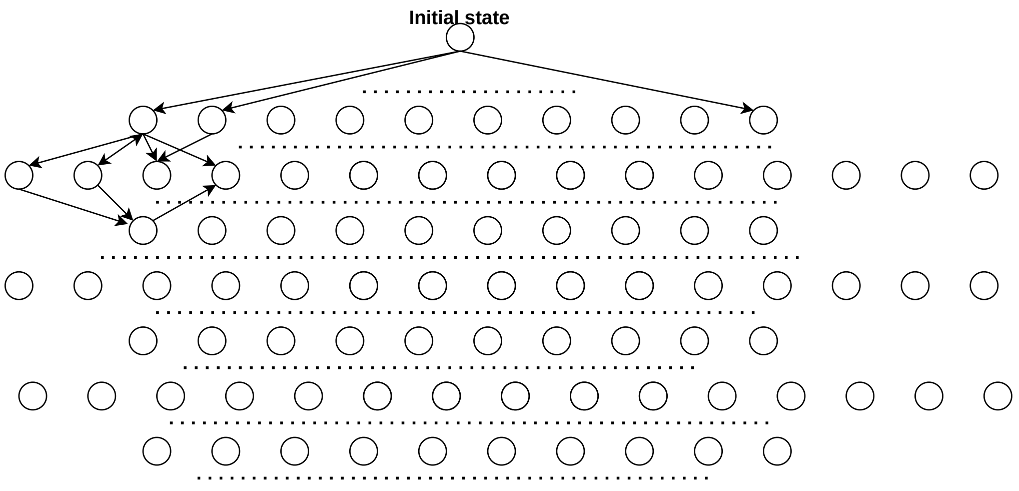


Figure 1.2. Formal Process

- assertion **Fails** : a counter example exists and there is an error hidden somewhere.
- assertion **Fails with warnings** : a counter example exists but not with primary inputs. It may be caused by unitialized registers or floating wires.
- assertion **Inconcludent**: the assertion is not proven neither fails. This may happen because the computational resources were not sufficient to give a result on the assertion. In this case a solution could be to perform the Formal Verification on a more powerful platform or to simplify the assertion.
- assertion **Vacuously proven**: the set of assumption on the signals do not allow the initial condition of the property to actually be triggered. For example, if we assume that signals `a` and `b` can never be 1 simultaneously and there is an assertion that states that when both `a` and `b` are 1 the output should be 0, that assertion is vacuosly proven.

The Static approach does not need any complex structure but it cannot satisfy every need: as said before, not every kind of DUT can be tested with static tools, due to the exponential growth of the mathematical complexity to find *all* possible stimuli. Nevertheless, Formal Verification comes in very handy in two phases of the Verification process (Figure 1.4):

- Bug hunting: it is the initial phase of the Verification, when the most part of bugs are found.
- Assurance: before tape-out, to ensure that the minimum functionalities that have to be work in every case are actually working.

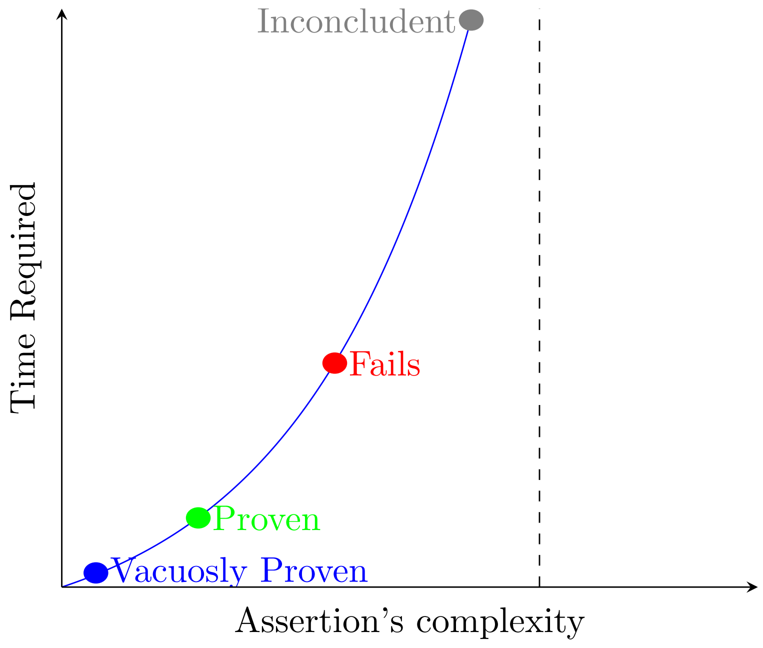


Figure 1.3. Possible Formal Outcomes

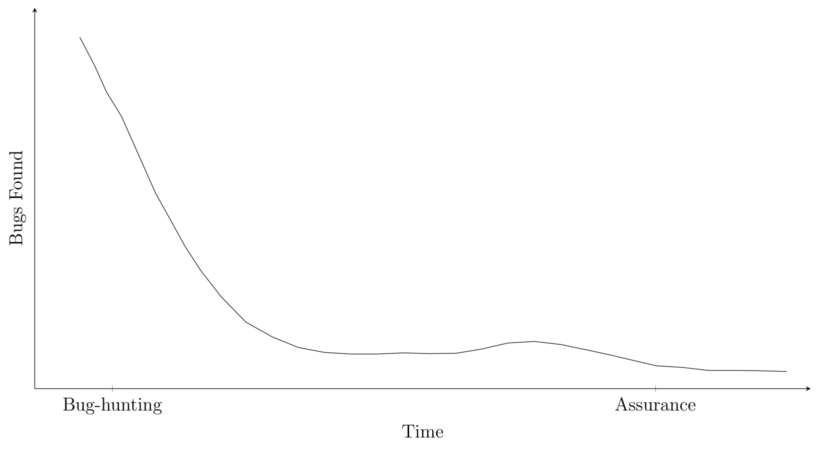


Figure 1.4. Bugs Found Against Time

1.2.3 Assertions and Assumptions for Functional Verification

Assertion-Based Verification is the core topic of this work. It is enabled by the presence of Assertions and Assumption and here we will describe what it is and what advantages it brings. Assertions and Assumptions are needed to perform Formal Verification and in Dynamic Verification they can significantly increase the **observability** of complex DUT, shortening the time to develop and to cover. How? The following quote of Alan Turing, reported here [17], contains the idea beyond this method:

Checking a large routine by Dr A. Turing:

"How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows. "

Image 1.5 describes Assertion-based Verification: given a complex system, made up of several submodules, it is possible to bind the Assertions and Assumptions on each single submodule. When an error against the top-level scoreboard is detected, this allows to see the error not only on the output of the DUT but also to trace it back to the malfunctioning submodule that causes the error. Also, Assertions are an objective measure of coverage since we will also check if and how many times they *passed* or *failed*. Moreover, it may happen that the Submodule Under Test (SUT) is working properly but it receives an unallowed input that may bring it to an unknown state, in this case, the assumptions would spot this bug and trace an error to the module that is driving the DUT.

Assertion-based Verification is an indispensable approach that has to be adopted when performing Verification: it can speed-up the Verification process up to 50% ([23]) and can find up to the 34% of bugs of a Device Under Test ([15]). In addition, it unlocks a robust Functional Verification scheme that is a hybrid between Static and Dynamic Verification, as illustrated in Image 1.6. In this work, we show an Assertion-based Verification Plan and the benefits that come from it.

However, there is a key difference between the classical Functional Verification described here and our work: we were not provided with precise and fixed specification of the studied submodules, so, along with the bug hunting we also had to write the specifications and, in case of false positives, correct our assertion and assumption thanks to a tight communication with the Design Team. This has brought several results:

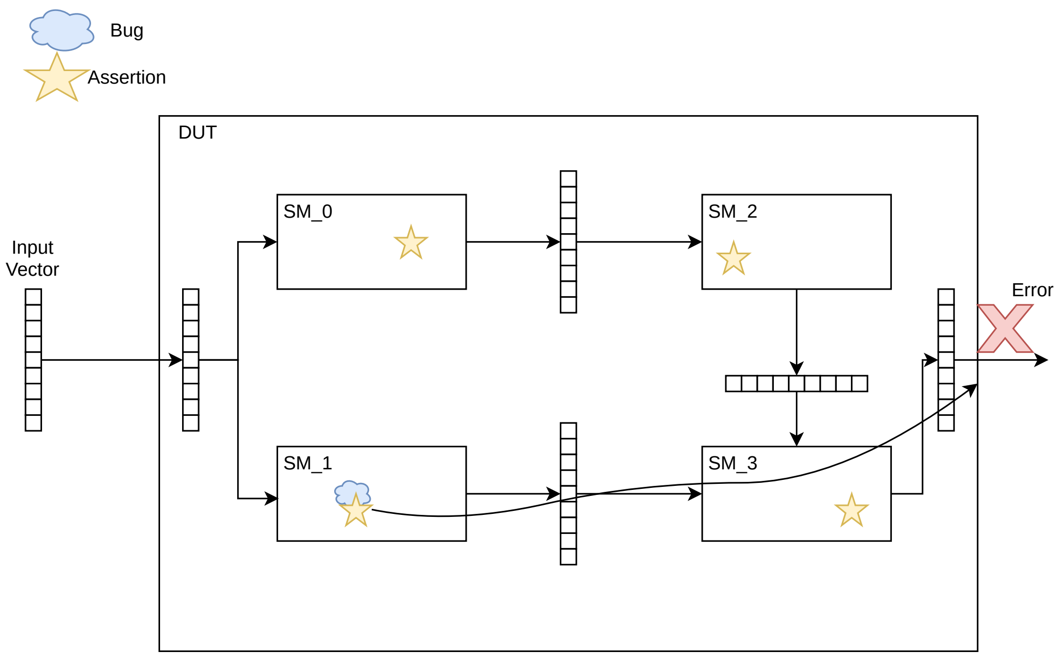


Figure 1.5. How Assertions improve observability

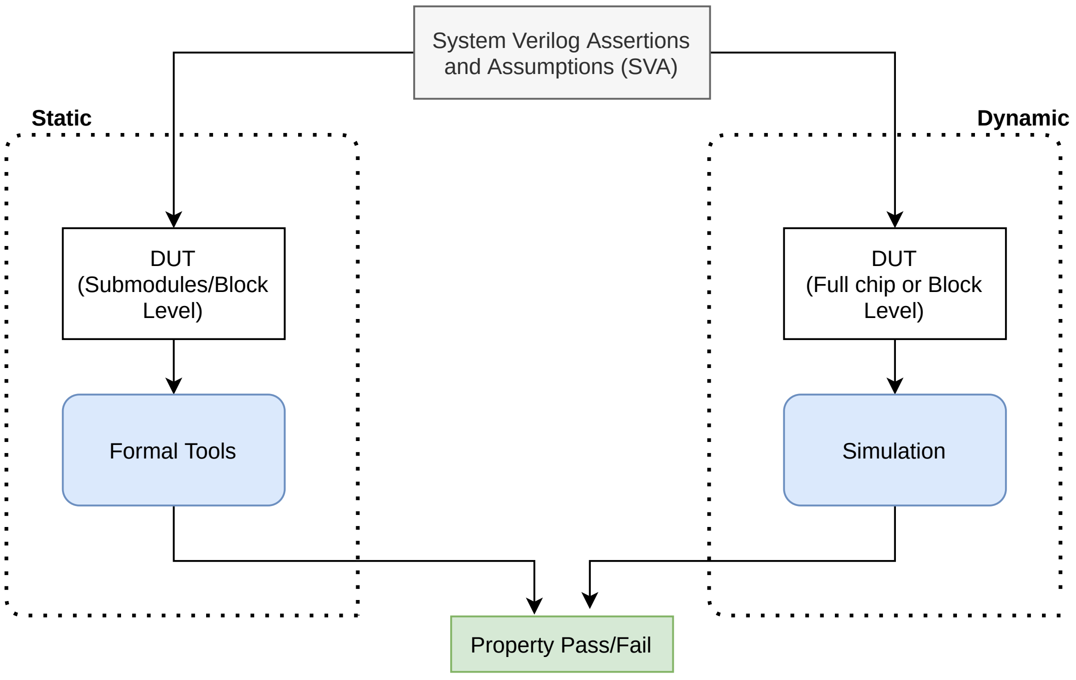

System Verilog Assertions and Assumptions (SVA)
Static
Dynamic
DUT (Submodules/Block Level)
DUT (Full chip or Block Level)
Formal Tools
Simulation
Property Pass/Fail

Figure 1.6. Hybrid Verification Approach

- we developed an Assertion-based Verification plan for each targeted submodule of the VPU,
- we developed "checker" modules filled with all the Assertions relative to each targeted submodules,
- we inserted these checkers in the tests of the whole RISC-V architecture,
- we developed specialized testbenches for the single DUTs and performed Static Verification on them,
- we provided reliable specifications and showed the actual advantages of this Hybrid approach for Functional Verification,
- as we will see in Chapter 4, during the time period where the Verification Plan was stable and working, assertions found the *36% of bugs* relative to the DUTs to which they were applied.

In the next chapter we will further describe in detail the architecture, the testbenches and the submodules.

Chapter 2

Context

This Chapter will be divided in three sections and will illustrate the framework in which this work has been carried out. The first one (2.1) will present the structure of the Vector Processing Unit(VPU) and will add a detailed description of its memory organization, which is crucial to understand its working. The second Section (2.2) will briefly illustrate the tested submodules and their role in the VPU. Finally, Section 2.3 will show how the simulations on the VPU were performed by the Verification Team.

2.1 The VPU

As said in Chapter 1, VPU have been closely identified with the Cray-1 Supercomputer which enables an extremely efficient implementation of a Vector Architecture. The basic conceptual structure behind the classical VPU is pretty simple and can be observed in Figure 2.1: the Vector Register File is distributed in more *Vector Lanes*. So, each Vector Lane contains a portion of each Vector Register, features the computing units and communicates with the memory subsytem. In particular, being N_LANES the number of present Lanes:

- each Vector Lane holds, for each Vector Register, $VLEN/N\_LANES$ bits and $VLEN/(N\_LANES \cdot SEW)$ elements.
- each Vector Lane features one (or more) deeply pipelined Functional Unit to process many elements at the same time.
- the Vector Lanes are equal between each other and communicate with the same memory subsystem.

Since the VPU under development targets HPC applications, as reported in Table 1.2, it has $VLEN = 16384$ and $N\_LANES = 8$ Vector Lanes with Datapath of 64 bits (that gives a total equal to $Datapath = 8 \cdot 64 = 512$).

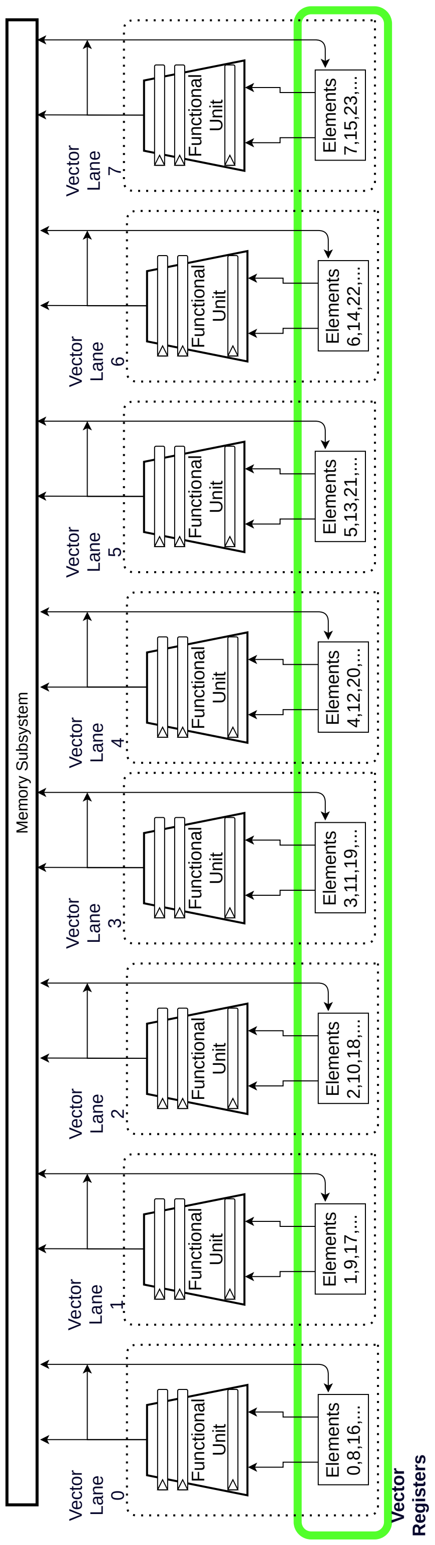


Figure 2.1. Vector Unit Structure

Naturally, the scheme in Figure 2.1 is just a conceptual scheme and misses many key features that we will see further in this Section: our VPU will work as a co-processor of a RISC-V scalar Core (called Avispado) and it is thought to support the RISC-V Vector Extension specifications, draft 0.7.1 ([4]), which requires a sophisticated and complex hardware. As expected, Avispado will perform the scalar operations and will issue different instructions to the VPU at the same time with a credit system. The instructions contained in the RISC-V Vector Extension can be categorized as follows:

- **Configuration-Setting Instructions**: these instructions will set the Control Status Register (CSR) containing the information on the Single Element Width (SEW) which, for our VPU, can be equal to 8,16,32 or 64. All the executed instructions will take into account the current SEW and operate according to it.
- **Vector Loads and Stores**: vectorial loads and stores will move data from and to the memory. The data will be sent by/to Avispado, which communicates with the external memory, in chunks of 512-bits.
- **Arithmetic Instructions**: it will support operations on Integers, Floating-Point and Fixed-Point elements.
- **Masked Operations**: Masked operations are a magic feature of the VPU. Basically, every instruction can be masked: when an operation is masked, a mask comes with it to indicate which elements of the vectors, that are being processed, are active. With this method, one can calculate the mask before than executing an operation and avoid a unefficient branchy execution. For example, if we need to execute the following code:

  ```
  for(i=0;i<64;i++)
          if(A[i]%2) C[i]=A[i]+B[i];
  ```

  with the masked operation we can first compute the mask based on $A[i]$ and then execute the whole $C[i] = A[i] + B[i]$; knowing which are the active elements.
- **Vector Reduction and Widening operations**: Reduction operations perform an operation that computes a scalar from a whole Vector like, for example, to find the maximum element in it. Widening instruction doubles the SEW of the elements of a Vector. An example of a reduction instruction could be: *vredsumu.vs vd,vs2,vs1,vm* which is equal to *vd[0] = sum(vs1[0],vs2[*])*, where vd is the destinatio vector, vs1 and vs2 are the two source vectors and vm is the mask vector that indicates all the vs2's active elements (denoted by *).
- **Vector Permutation operations** : these operations are the ones used to move elements around within the vector registers. We can move single elements

in a certain position of a vector (Scalar Move) and extract an element from a vector. Also, we can move whole vectors with three different operations:

- $vslideup : vd[i] = vs[i + rs1]$ where vd is the destination vector, vs the source vector and rs1 is a scalar value, whether being an immediate value or a element loaded from the memory.
- $vslidedown : vd[i] = vs[i - rs1]$
- $vrgather : vd[i] = vs1[vs2[i]]$

After this brief introduction to the RISC-V Vector Extension, it is possible to appreciate a high level scheme of the VPU structure. The VPU is made of several components: the work done during this thesis has been focused on the *Ring* modules which are the modules in charge of the communications between the Lanes, i.e. the components involved in Widening, Reduction and Permutation Operations.

The communication system is a circular system: every Lane can receive data coming from the previous Lane and send it only to the next one. This approach is not much reported in literature but was chosen to minimise the needed hardware. From the point of view of the Lanes, each Lane sees two modules called *Ring* (Data and Index) to which they send the data for the next Lane, and see other two sending the data from the previous one. In fact, no module called *Ring* exists, it rather is a heterogeneous combination of several submodules, each one with a specific task.

From Figure 2.2, it is possible to observe the structure of the VPU, with a particular emphasys on the Ring Modules, that will be illustrated in Section 2.2. Figure 2.2 shows how the VPU communicates just with Avispado and has no direct access to the memory. Avispado issues the instructions that are then decoded by the VPU, then the command signals are sent to the Vector Lanes and the Load and Store Management Units. When an operation for the Rings is issued, the VPU sends the data to them so they arrive to the destination Lane.

Before going further with the description of the Ring Modules, in the next Section, we have to focus on the VRF organization since it is a key point in the VPU understanding and it is worth to be explored.

As said before, the VPU features 8 Lanes, has $VLEN = 16384$ bits and $SEW = 8/16/32/64$ so each one of the 32 Vectors can have respectively *VLMAX=2048/ 1024/ 512/ 256*, according to the current SEW. Each VPU has a slice of the VRF: starting from Lane 0, each Lane takes 64 bits of the Vector. The *ELEMENT_ID* is defined as the position of the element in the Vector it belongs to. So, with SEW 64 we will have Lane 0 holding elements 0,8,16,...,248, Lane 1 holding 1,9,17,...,249

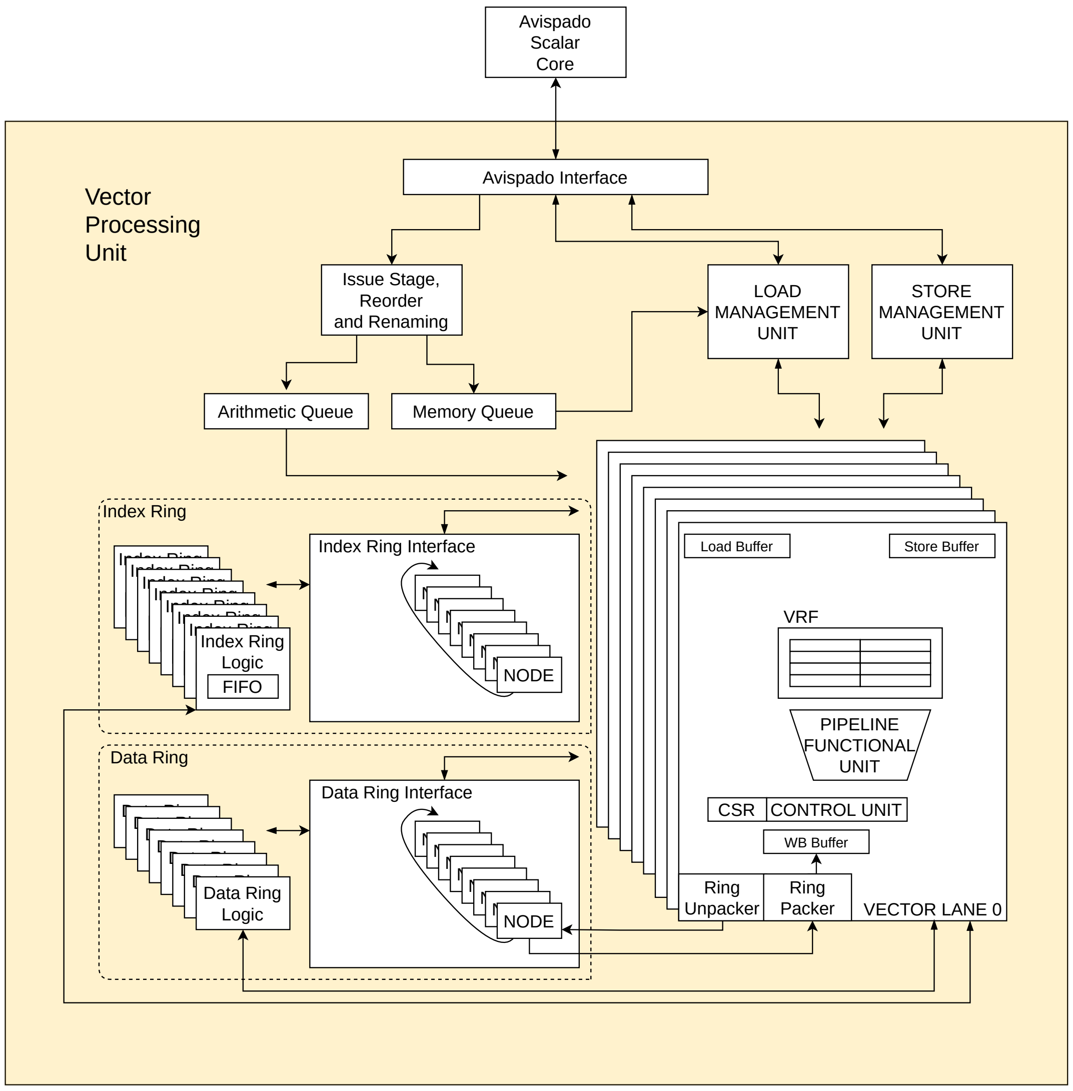


Figure 2.2. Vector Processing Unit

and so on, as reported in Table 2.1, where we have the Element ID of each element in each cell. As compared to Table 2.2 where we have the element distribution with SEW 32.

As far as the memory organization is concerned, each Vector Lane has a SRAM Memory of $10kB$ organized in 5 banks of $2kB$ each. Knowing the EL_ID, to address each element in the SRAM within a Vector Lane we need to compute 4 data:

1. **SubBank**: When $SEW \neq 64$ each bank can be naturally subdivided in $64/SEW$ subbanks, to correctly address the data.

VL0	VL1	VL2	VL3	VL4	VL5	VL6	VL7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
...	...	...	...	...	...	...	...
240	241	242	243	244	245	246	247
248	249	250	251	252	253	254	255

Table 2.1. Vector Distribution SEW 64

VL0		VL1		VL2		VL3		VL4		VL5		VL6		VL7	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511

Table 2.2. Vector Distribution SEW 32

2. **Bank**: Each Bank has rows of 64-bit width, every 64-bits we increment the number of the Bank by 1. Since the number of Bank is 5 we restart to count from 0 every 320 bits.

3. **Line**: Each Line contains 5 Banks, so, for each 5 64-bit chunks of width we increment the Number of the Destination Line.

4. **Lane**: it is the Destination Lane (0 in the examples).

Examples for an arbitrary Vector *v_i* can be found in Tables 2.3 and 2.4: here we see how the elements are disposed in Lane 0, according to the SEW. In each cell we see the corresponding *element_id*, the letter **B** indicates the bank, **SB** the subbank. Computing the addresses and correctly address the data is the main task of the *Ring* Modules and that is why it is so important to focus on the VRF organization in the VPU's internal memory. We can now go further and present these modules.

VL0					
B0	B1	B2	B3	B4	
0	8	16	24	32	Line 0
40	48	56	64	72	Line 1
80	88	96	104	112	Line 2
120	128	136	144	152	Line 3
160	168	176	184	192	Line 4
200	208	216	224	232	Line 5
240	248				Line 6

Table 2.3. SRAM LANE 0, SEW 64

VL0										
B0		B1		B2		B3		B4		
SB0	SB1	SB0	SB1	SB0	SB1	SB0	SB1	SB0	SB1	
0	1	16	17	32	33	48	49	64	65	Line 0
80	81	96	97	112	113	128	129	144	145	Line 1
160	161	176	177	192	193	208	209	224	225	Line 2
240	241	256	257	272	273	288	289	304	305	Line 3
320	321	336	337	352	353	368	369	384	385	Line 4
400	401	416	417	432	433	448	449	464	465	Line 5
480	481	496	497							Line 6

Table 2.4. SRAM LANE 0, SEW 32

2.2 Submodules

In this Section we will briefly describe the modules that have been studied and tested. The description reported here aims to locate the modules in VPU and to show their dependecies and roles. Indeed, this Section will not work as a collection of specifications, as it would be of difficult comprehension and would turn out to be useless. Also, we decided to include in the next Chapter 3 more precise descriptions of the modules.

Index Ring and Data Ring

There are two types of Ring: Index and Data. The Index Ring is used only when performing a *vrgather* ($vgather : vd[i] = vs1[vs2[i]]$) and it computes the destination address given the EL_ID coming from *vs2[i]*. For the other operations the Index Ring is never involved. Indeed, the Data Ring is responsible for the correct functioning of every Operation of Slide, Gather, Reduction and Widening.

It receives the data together with the destination address and sends it to the correct lane.

ring_unpacker

The ring_unpacker receives data from the Source Buffer of the Vector Lane and then, according to the current SEW, it unpacks it in a 64-bit data. So, for example, it receives 64 bits in input with $SEW = 16$, then it sends the 4 elements in a row to the *ring_if*.

ring_if

The *ring_if* is the module that manages the flow of data between the Lanes. Basically, is the one that receives the data and circulates them. Most importantly, it does not perform any elaboration on the data but it just handles the timing, controlling and communicating with all the other Ring components.

ring_node

The *ring_nodes* are in the *ring_if*. Each Lane sees 4 *ring_nodes*, 2 of the Index and 2 of the Data Ring. Each node receives data both from its Lane and from the previous node, then, if one of the two inputs is for the current lane it sets the bits to send the data to the *ring_packer*. (Figure 2.3).

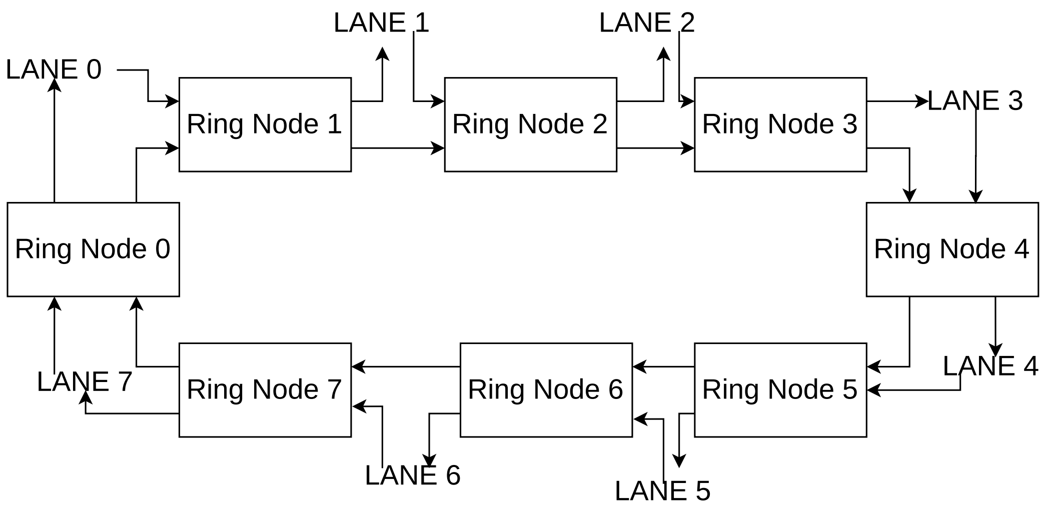


Figure 2.3. Ring Nodes System

ring_packer

The *ring_packer* is the last component of the ring involved in the operations. It performs the opposite operation of the unpacker: as the WB Buffer can receive the data for the VRF, we need to send the data already organized in 64-bit chunks.

ring_logic

The *ring_logic*, along with the *ring_node*, is the core of the Ring and it is the most error-prone. It has the task of computing the destination address. In case of a vgather, the *index_ring_logic* receives $vs2[i]$ containing the EL_ID and computes **Destination Lane**, **Destination Line**, **Destination Bank** and **Destination SubBank**, then it memorizes them in a FIFO of depth 10. Then, it sends all the computed addresses to its Vector Lane which will send to the Data Ring the *vs1[vs2[i]]* along with the computed address *vs2[i].* For all the other instructions, the Data Ring will receive all data in order from the source vector, from elemnt 0 to VLMAX and then easily compute the destination address, according to the current operation. With the computed addresses the *ring_if* will be able to send the data.

2.3 The UVM

This is the last Section that will introduce previous work and concepts. It is necessary to complete the description of the existing framework where the thesis has been carried out: here we will describe how the Verification of the whole VPU is being performed.

2.3.1 UVM

The Functional Verification for the VPU is carried out with the Simulation based approach and is performed with Universal Verification Methodology (UVM) which is a transaction-level methodology (TLM) designed to develop testbenches. It is implemented by an Object-Oriented Programming (OOP) languages, such as SystemVerilog, that provide a wide range of classes and libraries to easily develop re-usable and powerful tests. In particular, it provides libraries and models that enable the engineer to create any wanted structure and provides communication interfaces to connect the modules at transaction level.

Most importantly, UVM is a standard that codifies the best practices for efficient and exhaustive Verification. It is based on SystemVerilog Language and it provides an already defined hierarchy to test the DUT. The most basic UVM structure is the one in Figure 2.4. Naturally, the aim of this Section certainly is not to describe accurately the UVM language, which is a complete coding language and it needs books to be properly explained. Here, we would rather like to briefly present the classical UVM structure being it a fundamental part of the study that has been done. The components of the structure are the following:

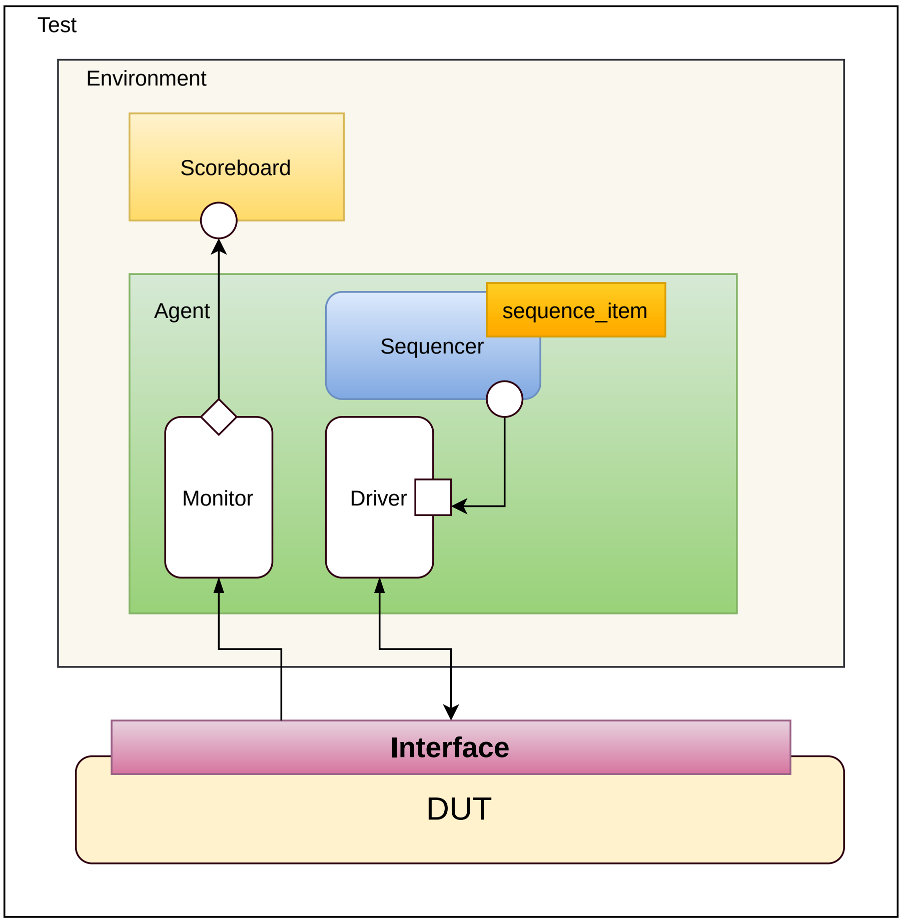


Figure 2.4. UVM Basic Structure

- **Test**: it is the top-level UVM component that instantiates the environments and configures it by overriding.
- **Environment**: it groups together all the components targeting the DUT. It defines the reusable component topology of the UVM Tests.
- **Agent**: The Agent contains most of modules of the UVM structure, such as:
  - the **Sequencer** that creates and sends the transactions to the Driver.
  - the **Driver** that takes the sequence_items and through a Virtual Interface sends them to the DUT.

  - the Virtual **Interface** that sends the Input and Output data to the Coverage **Monitor** that communicates with the Scoreboard, that naturally will check the DUT agains a reference architecture.

The extreme power of UVM is the re-usability and the ease of use: the reusability is unlocked by the possibility of overriding the classes of the instantianted object while the only thing we have to take care of is thinking of tests and not coding the structure, which is always standard and grants a high modularity. This drastically reduces the time to verify and why it is such a common standard.

2.3.2 UVM Structure

To test the whole RISC-V structure the Verification Team has developed a consistent UVM architecture, along with a huge effort for its automatizations and tests. The basic steps to verify the VPU are illustrated in Figure 2.5 and can be summarized as follows:

1. firstly, an assembly code has to be generated. To do so it is used a slightly modified version of riscv-dv ([7], [6]), which is a open-source constrained-random generator of instructions targeting RISC-V Verification. It is modified by the Verification Team to block the instructions not yet supported. Naturally, an assembly program could also be hand-written in case we want to launch specific directed tests.

2. Then, the assembly has to be compiled with the GCC compiler that targets RISC-V Vector Extension [5].

3. Once we have the binary, it is possible to send it to a modified version of Spike [8]. Spike works as a C library that can emulate a RISC-V architecture. In our case Spike accomplishes a double function: the first one is to execute the scalar instructions and then send the vector ones to the UVM. The second is to act as a Scoreboard for the vector instructions.

4. Finally, the UVM receives the vector instruction from Spike and issues them to the VPU, emulating the Avispado's protocol. Naturally, the UVM features the Drivers to do so, and the Monitors too to sample the VPU state and compare it with the Spike Results, when an instruction is executed.

This flow is automatized and runs every night to perform as many tests as possible on the VPU. Every execution is reported on the online repository and everyone can re-run the test locally to debug the errors found during the night. It is clear that this approach is essential to perform the functional verification of the VPU and it would not be possible to do otherwise. Also, this approach allows the Verification Team to

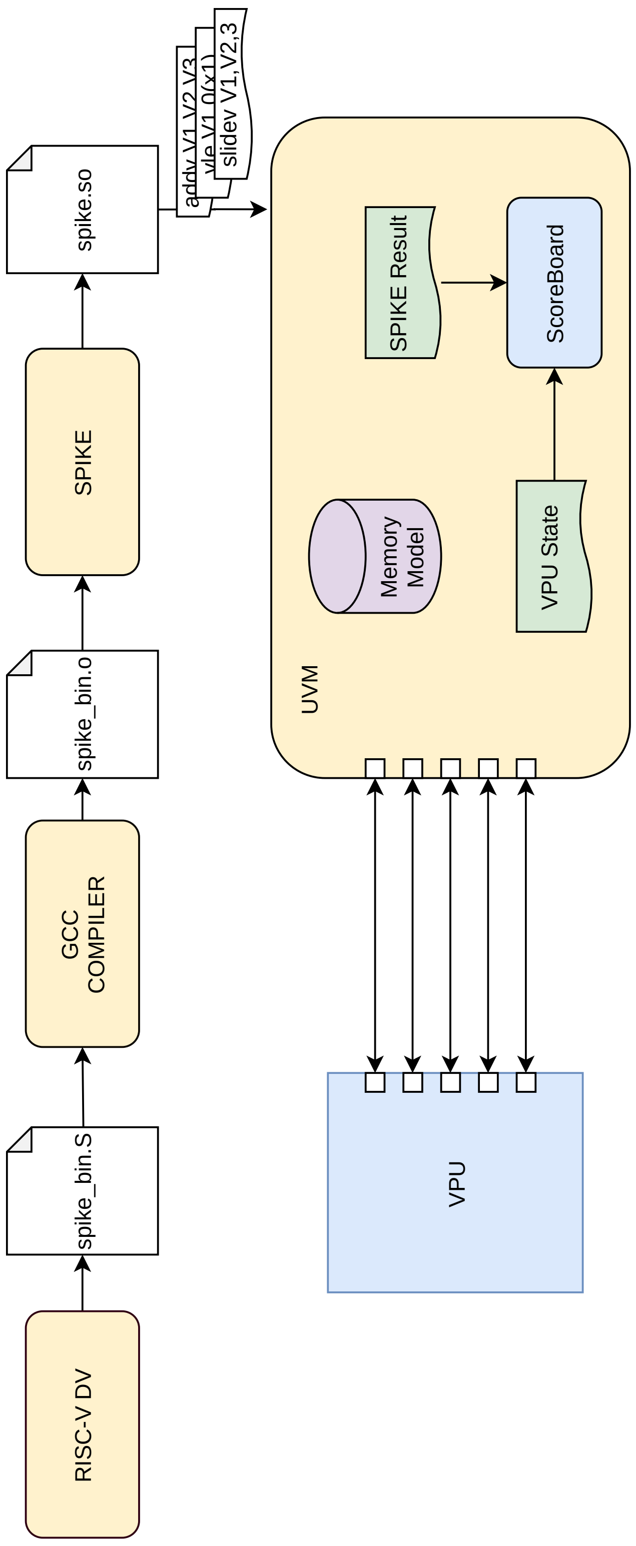


Figure 2.5. Verification steps for the VPU

work without any dependencies from the Design Team since the Vector Extension Specifications are fixed and do not depend on the VPU current implementation.

On the other hand, it is a sub-optimal approach: when an execution stops due to a mismatch between Spike and the VPU, the only clues that the Design Team has are the waveforms of the failing test and tracing back the error is accomplished mostly by hand, which it may result pretty hard and time consuming, given the VPU complexity and the length of the chain of components for each single instruction execution. That is why, along with this method, it is of crucial importance to focus also directly on the most critical and complex submodules of the VPU, to develop directed testbenches for them and to insert specific assertions directly within the VPU-level testing, as discussed in Chapter 1. In this way, when the error is detected and an assertion fails, it is much more easy to find the bug and correct it.

As said in Chapter 1, this kind of work needs to be performed by the Verification Team along with the Design Team and it is in what consists the work presented here. In the next Chapter, there will be illustrated more in depth the Submodules, the Assertion-based Verification Plan and its actuation, along with the Formal Verification that has been performed. Then, in the last Chapter we will show the the benefits coming from the implementation of this approach and already anticipated in Chapter 1.

Chapter 3

Methods

In this Chapter we are going to present the developed Verification Plan and its realization. Section 3.1 will work almost as a collection of specifications: we are going to present the Submodules that have already been introduced in Chapter 2, with a focus on their interfaces and handshakes, along with their waveforms, to provide the necessary description to understand the Verification Plan and to document the method followed in this work. The written specifications are 17 pages long and it is not necessary to attach them here to actually appreciate the work. In Section 3.2, we are going to present the Verification Plan: the checkers with the deployed assertions, the UVM and the directed tests.

3.1 The Submodules and Specifications

Writing the missing specification required a consistent effort of interfacing with the Design Team. Naturally, the process it has not been flawless: many times the assertions based on the unstable specifications gave false positive when reported to the designers. In that case the specifications and the assertions had to be modified while everything was working as expected. Nevertheless, writing specifications almost from scratch was a key exercise to think of the modules and allowed us to find reduntant signals or even bugs, without even the necessity of running tests.

It is important to recall that two Rings are present in the VPU: Data and Index. The Index one is involved only in the gather operation, which is defined as follows:

```
vrgather.v vd,vs1,vs2 # vd[i] = vs2[i]>VLMAX ? 0 : vs1[vs2[i]];
```

In this case, the Index Ring structure takes in input $vs2$, containing the destination ELEMENT_IDs, and computes the destination address for the VRF. Then, $vs1$ is sent by the Data Ring according to the new computed addresses. For the other Ring operations (Reduction, Widening, Permutations), the Index Ring is never involved as compared to the Data Ring which is always active.

3.1.1 Ring Interface

The Ring Interface is the component that handles the communications with the Lanes:

- every Lane sends to the Ring Interface the request to send some data to another Lane, along with the data and the destination address.

- after the request is accepted, the Ring Interface will send to the destination Lane the data that it received.

Thus, the Ring Interface, whether the data or the index, can receive two possible commands (Figure 3.1) from the Lanes:

1. a new request is issued: the signal `enable_i` is asserted, along with the information `length_i` that tells how many elements will be sent through the ring. Then, every time `req_i` is 1, it means that #`req_cnt_i` elements are sent together with their destination address. When the number of element sent is equal to the initially requested number, the operation ends.

2. an accumulation operation is issued: it is exactly like a normal request but without any counter on the number of elements.

When a data valid for a lane is in output, the corresponding bit of `we_o` is set to 1 and the data is sent to the corresponding Lane.

3.1.2 Ring Node

The ring node is a incredibly simple component within the ring interface. It executes just one operation, in one clock cycle: it receives data both from the previous ring and from its lane, when a valid data is in input it sets the output `we_o` to 1 in order to send it to the packer. This behaviour can be easily described with the Control Flow Diagram in Figure 3.2.

Even if this component appears to be simple, it results to generate a great amount of assertions and doubts. For example, what is it supposed to happen when both data in input are valid? Or, is this even something allowed to happen? What does it happen when a valid data is given along with a `stall_i` or a `kill_i`? As it is possible to imagine, writing the specifications and the assertion on a module forces you to think of corner cases from the first moment and to investigate all the possible input combination, to avoid writing down false assumptions.

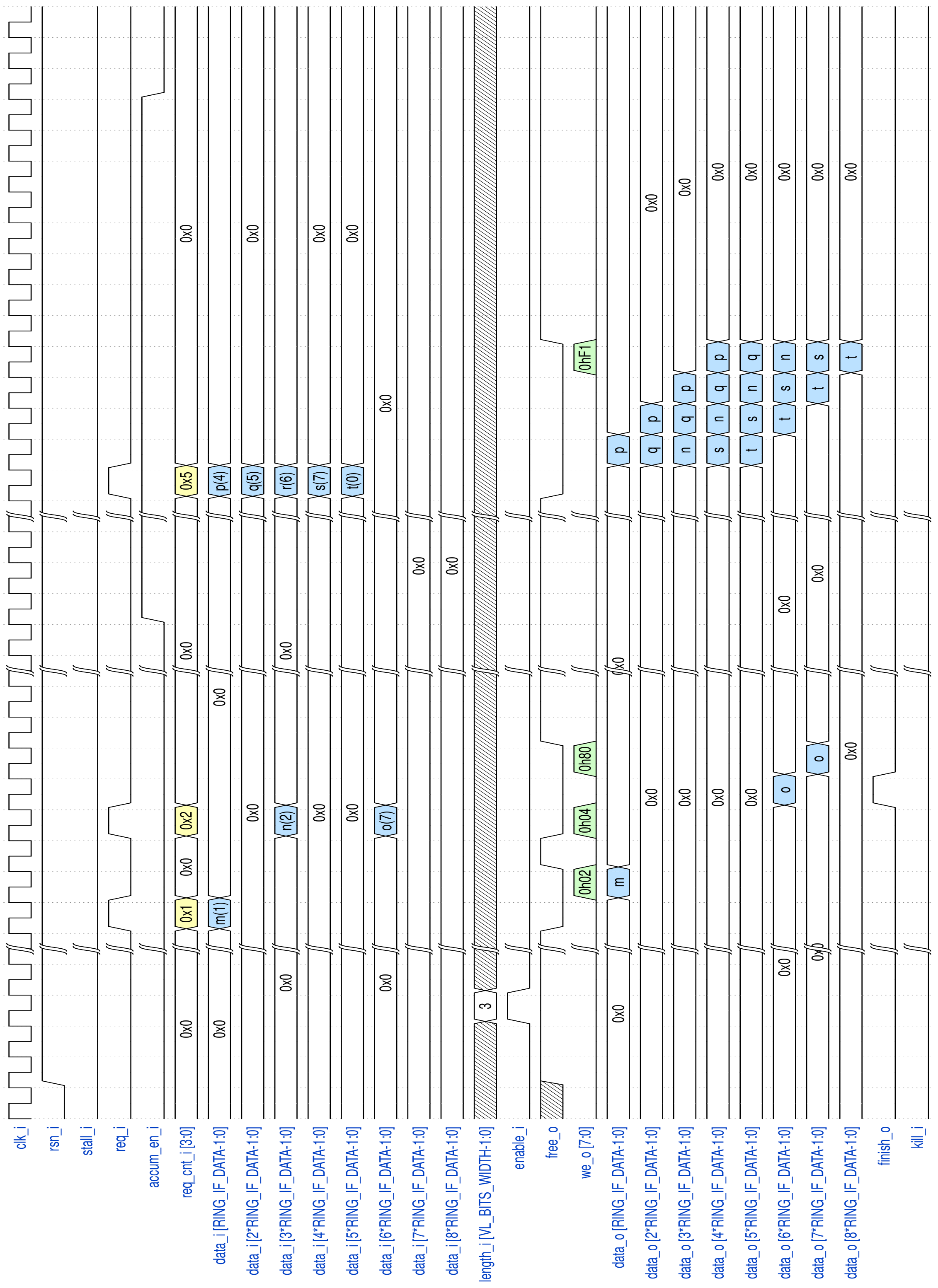
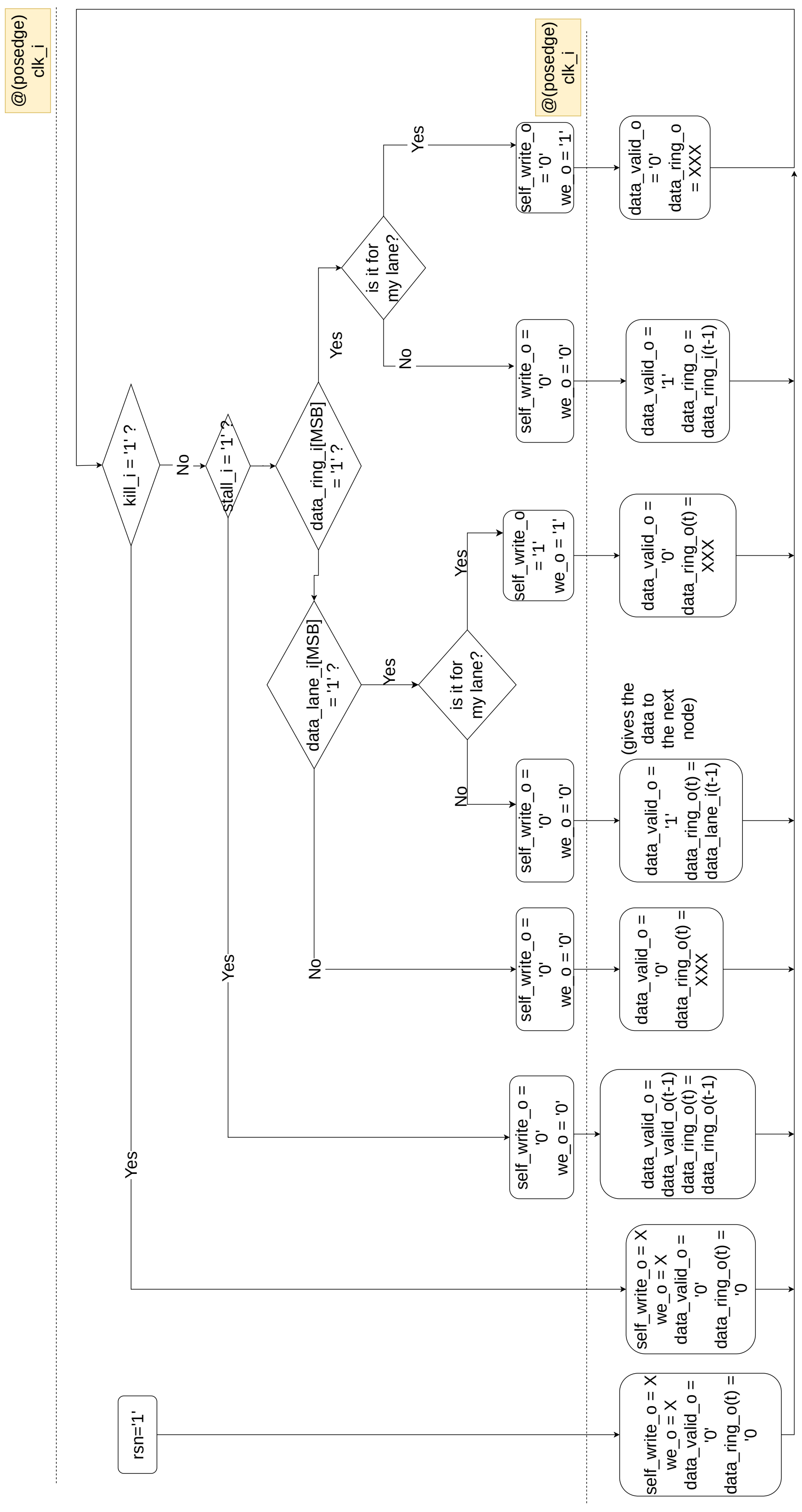

clk_i
rsn_i
stall_i
req_i
accum_en_i
req_cnt_i [3:0]
data_i [RING_IF_DATA-1:0]
data_i [2*RING_IF_DATA-1:0]
data_i [3*RING_IF_DATA-1:0]
data_i [4*RING_IF_DATA-1:0]
data_i [5*RING_IF_DATA-1:0]
data_i [6*RING_IF_DATA-1:0]
data_i [7*RING_IF_DATA-1:0]
data_i [8*RING_IF_DATA-1:0]
length_i [VL_BITS_WIDTH-1:0]
enable_i
free_o
we_o [7:0]
data_o [RING_IF_DATA-1:0]
data_o [2*RING_IF_DATA-1:0]
data_o [3*RING_IF_DATA-1:0]
data_o [4*RING_IF_DATA-1:0]
data_o [5*RING_IF_DATA-1:0]
data_o [6*RING_IF_DATA-1:0]
data_o [7*RING_IF_DATA-1:0]
data_o [8*RING_IF_DATA-1:0]
finish_o
kill_i
0x0
0x1
0x2
0x5
m(1)
n(2)
o(7)
p(4)
q(5)
r(6)
s(7)
t(0)
3
0h02
0h04
0h80
0hF1
m
n
o
p
q
s
t

Figure 3.1. Ring Interface



@(posedge) clk_i
kill_i = '1' ?
No
stall_i = '1' ?
data_ring_i[MSB] = '1' ?
Yes
is it for my lane?
Yes
self_write_o = '0' we_o = '1'
@(posedge) clk_i
data_valid_o = '0' data_ring_o = XXX
No
self_write_o = '0' we_o = '0'
data_valid_o = '1' data_ring_o = data_ring_i(t-1)
data_lane_i[MSB] = '1' ?
Yes
is it for my lane?
Yes
self_write_o = '1' we_o = '1'
data_valid_o = '0' data_ring_o(t) = XXX
No
self_write_o = '0' we_o = '0'
(gives the data to the next node)
data_valid_o = '1' data_ring_o(t) = data_lane_i(t-1)
No
self_write_o = '0' we_o = '0'
data_valid_o = '0' data_ring_o(t) = XXX
Yes
self_write_o = '0' we_o = '0'
data_valid_o = data_valid_o(t-1) data_ring_o(t) = data_ring_o(t-1)
Yes
self_write_o = X we_o = X data_valid_o = '0' data_ring_o(t) = '0
rsn='1'
self_write_o = X we_o = X data_valid_o = '0' data_ring_o(t) = '0

Figure 3.2. Ring Node

3.1.3 Index Ring Logic

The index ring logic is used only for gather instructions and it is provided with a FIFO of depth 10 to save different destination addresses of the data, which is organized as reported in Figure 3.3: is the concatenation of

- destination Lane, destination line, destination bank, destination sub bank.
- sending Lane, sending bank, sending sub bank.
- valid bit and zeroing bit.

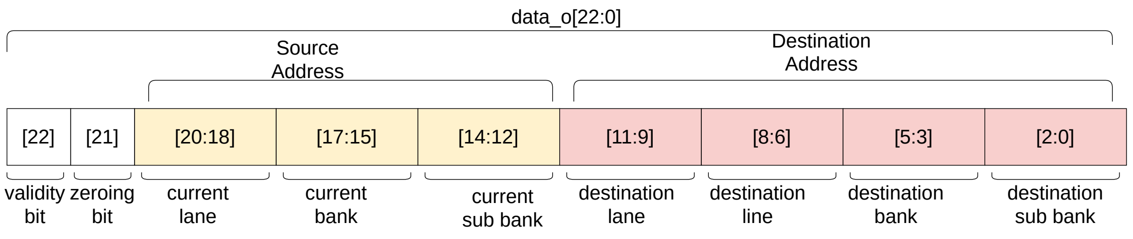


Figure 3.3. Address Composition

As reported in Figure 3.4, when a gather instruction is issued, `enable_i` and `start_i` are 1. Then every time a `send_i` arrives, the index ring logic receives `vs2[i]`, containing the `element_id` on `offset_i`. When `vs2[i]>VLMAX` the zeroing bit is set to 1. Given that information we can compute the destination address as follows:

```
el_id_dest = send_fifo[0].offset_fifo;
//update fifo
for (j = 0; j < FIFO_SIZE - 1; j++) begin
        send_fifo[j].offset_fifo = send_fifo[j + 1].offset_fifo;
        send_fifo[j].free = send_fifo[j + 1].free;
end

send_fifo[FIFO_SIZE-1].offset_fifo = 0;
send_fifo[FIFO_SIZE-1].free = 1;

dest_bank_int = (el_id_dest/(sub_banks*N_LANES))%N_BANKS;
dest_sub_bank_int = el_id_dest % sub_banks;
dest_line_int = el_id_dest/(sub_banks*N_LANES*N_BANKS);
dest_lane_int = (el_id_dest/sub_banks) % N_LANES;
```

Like in the case above, when the component increases just a bit the complexity, it is possible to develop a scoreboard with a high level of abstraction. In this way, we avoid to create errors due to the complexity of the code and we can implement a scoreboard that is much alike as possible as the specifications, without any bind to the actual RTL implementation.

3.1.4 Data Ring Logic

The Data Ring Logic computes the destination addresses for the Permutation instructions and for Widening and Reduction. These operations work on the whole source vector `vs1`, from element 0 to element VL_MAX: the Data Ring Logic computes the destination element_id, knowing the current element_id of the sender, according to the current instruction. Then, the destination address is computed with the equations showed for the Index Ring Logic. For each operation the handshake is different, in this way we make the Data Ring Logic aware of the current instruction. At the beginning of every instruction it receives `enable_i` and `start_i`, which works as a reset of the internal counter, and then 4 possible instructions can be issued:

- for the slide operation `type_i` tells whether it is a slide down or a slide up and it computes the destination as $el\_id_{dest} = el\_id_{source} + (-1)^{type\_i} \cdot offset\_i$.
- for the widening operation the `widening_i` signal is 1 and simply $el\_id_{dest} = el\_id_{source} \cdot 2$.
- for the reduction operation, `accum_en_i` is 1 and $el\_id_{dest} = 64/SEW * (LANE\_ID + 1)$.
- for the gather instruction, `rgather_op_i` is equal to 1 and the Data Ring Logic just gives in output the address computed from the Index Ring Logic which are passed through `rgather_lane_i`, `rgather_bank_i` and `rgather_sub_bank_i`.

3.1.5 Ring Data Unpacker

The Ring Data unpacker unpacks the 64-bit input data according to the current SEW. A new unpacking is requested with `new_req_i`, along with `elements_i` indicating the number of elements that will be given in output during the whole operation. Then, `req_i` is 1 and the 64-bit data arrives. Within the completion of a request the inputs must remain stable. Figure 3.6 shows what happens with 32-bits elements and 4 elements unpacked. It can also happen that the unpacker receives `finish_i` before having unpacked all the elements, in that case it has to forget the inflight request.

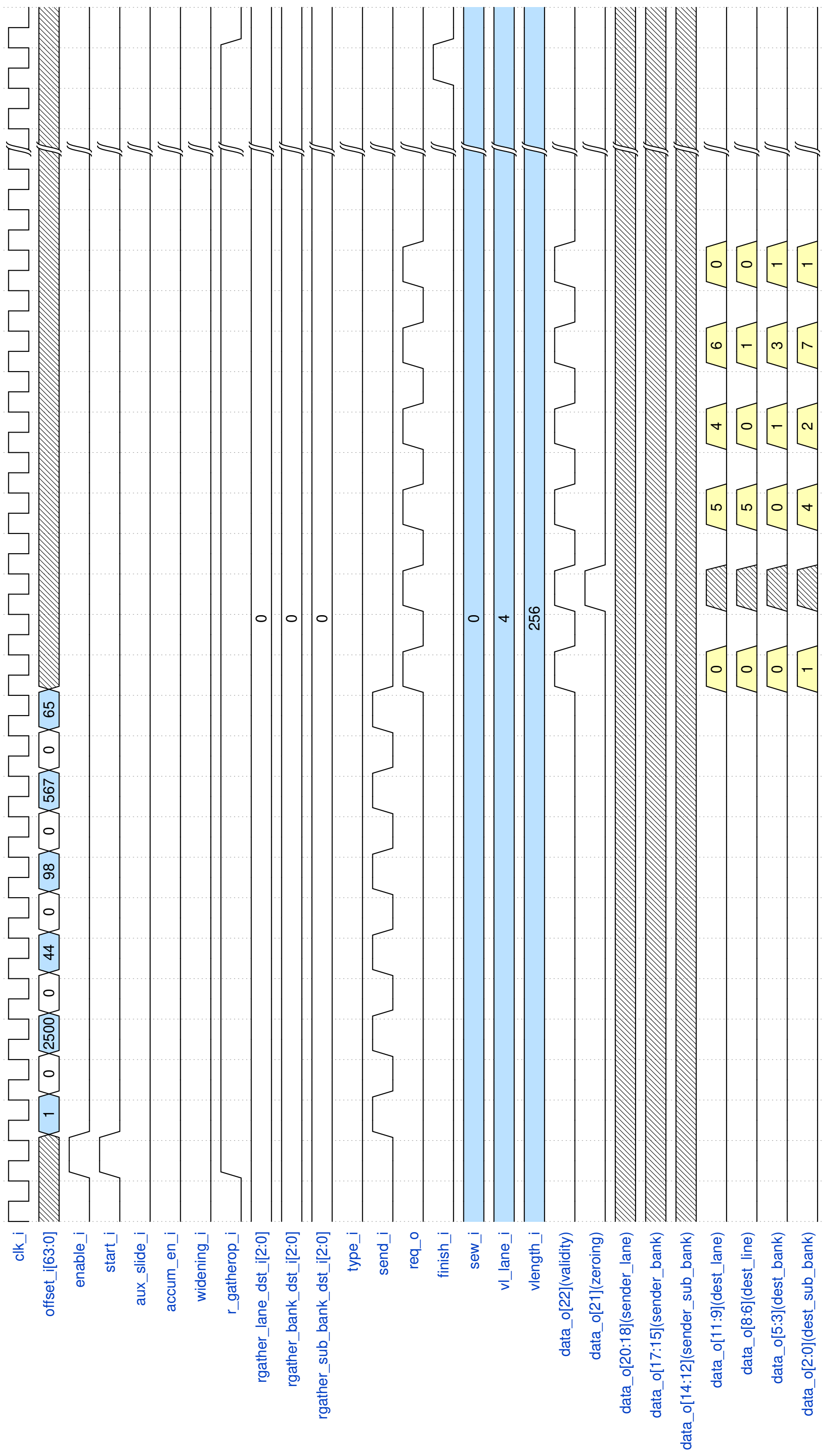


Figure 3.4. Index Ring Logic Gather Waveforms

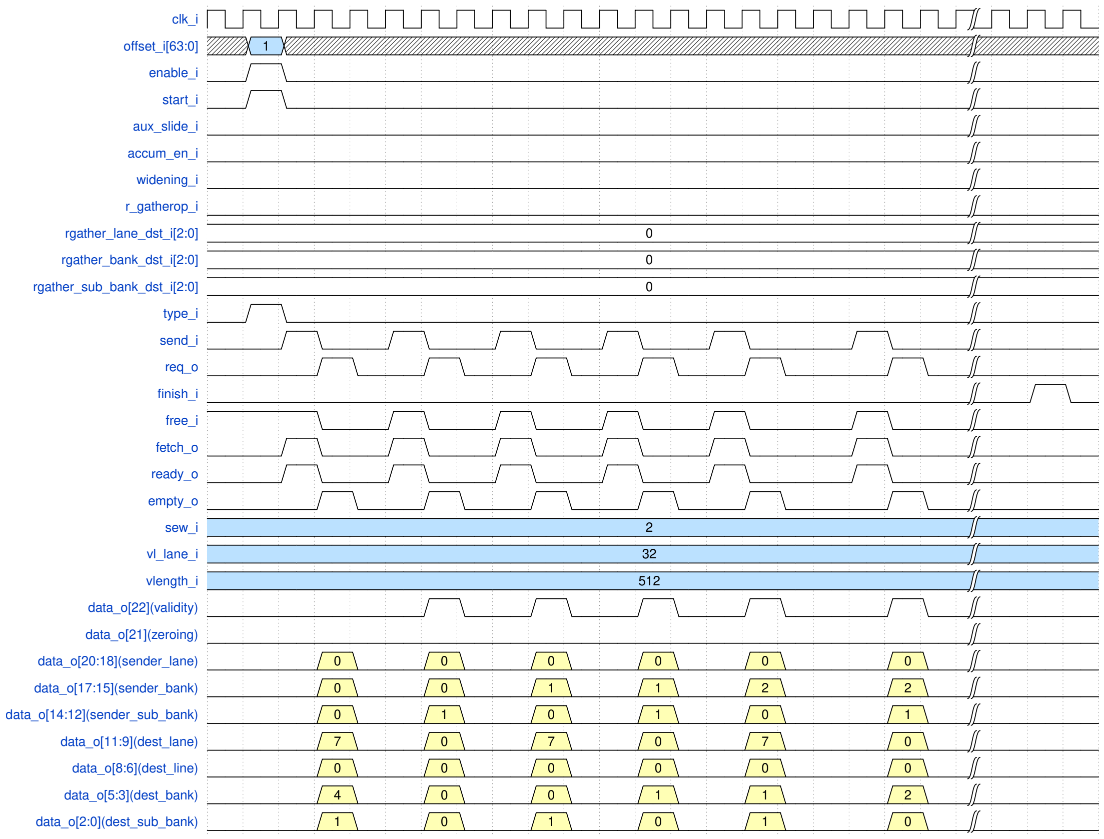


Figure 3.5. Data Ring Logic Slide

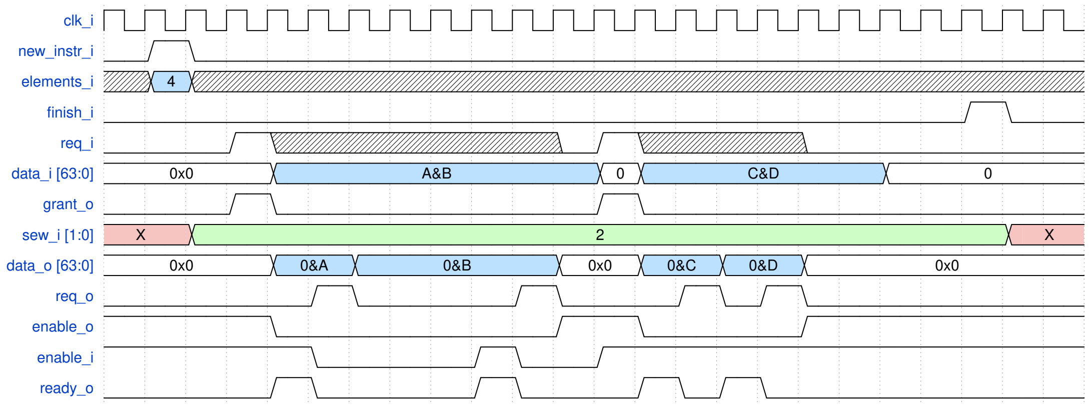


Figure 3.6. Ring Data Unpacker

3.1.6 Ring Data Packer

The Ring Data Packer accomplishes the opposite result of the unpacker. As Figure 3.7 shows, there are two possible patterns, apart from the possible different SEWs:

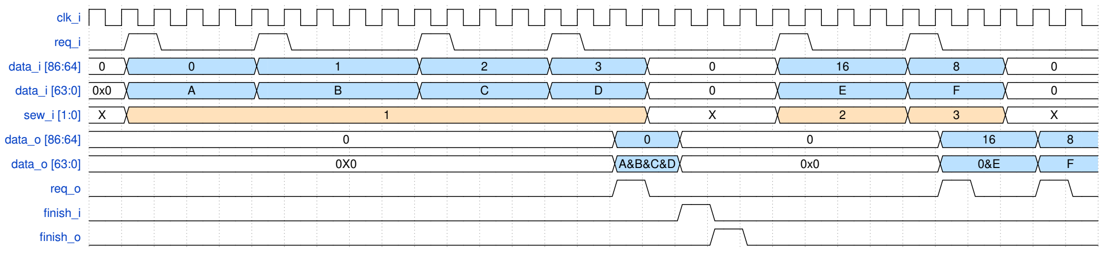


Figure 3.7. Ring Data packer

- 4 data of 16-bits are sent in a row to the packer, *the 4 data destination are 4 adjacent position in the VRF*. So, the packer will send a *unique* `req_o=1` when *all* the 4 data are received and will concatenate them in `data_o`.
- a 32-bit data is sent, and then a 64-bit data. The two data are neither adjacent nor fot the same destination: in this case, the ring data packer will understand that *the arrived elements are uncorrelated* and will send them both to the WB buffer in two distinct clock cycles.

3.2 Verification Plan

In this Section, we are going to present the Verification Plan that has been developed to test the Ring modules. A Verification Plan can be defined as a document that delineates what Functional Verification is going to be performed. As an example, it may consist in a spreadsheet defining the features that the DUT has and these may get translated in coverage metrics.

The implementation of our Verification Plan is described in Figure 3.8. For each submodule, we started with studying it and writing down its specifications and features. These features were contained in a spreadsheet and then translated and implemented in 3 different ways:

1. direct translation in **assertions**. At this point, all the assertions relative to the DUT were contained in the *"checker"* file.
2. the development of a **dedicated UVM** for the DUT.

3. the implementation of **direct binaries** to test the whole VPU, targeting the instructions that involve the Ring Module.

At this point we had to test the assertions that had been wrote down. To do this there were 3 methods:

1. insert the checkers in the symulations of the whole VPU, as discussed in Chapter 1 and 2 for the Assertion-based Verification.
2. insert the checker in the directed testbenches implemented just for the single DUT.
3. use the checkers to perform Formal Verification.

The 3 approaches can cause an assertion to fail or to pass. In case of a failing assertion we had to discuss it with the Design Team and in case it was a false positive, re-write the specifications, the checker, the drivers of the testbench and eventually re-think of the directed Tests. Naturally, writing the drivers and thinking of directed tests was itself an exercise to think of corner cases and to fix the specifications.

The rest of this Section will describe in detail the elements present in Figure 3.8. Firstly, we are going to present the Checkers and how they were inserted in the tests. Secondly, we are going to show the methods used for Dynamic Verification: direct bynaries and the dedicated UVM. Lastly, we are going to discuss about the Static Verification.

3.2.1 Checkers and Assertions

Once a checker was defined, it was inserted in all the tests in a straighforward manner: the **bind** method (reported in [21], [13] and [18]). Basically, to bind two modules between each other we can use two methods:

- insert the checker as a submodule of the DUT. This means that the DUT will contain the checker to pass all the signals to the assertions.
- the bind method.

The first one is highly unconvenient for the Verification Team as it requires it to write code in the modules that are developed from the Designers, who may even be reluctant to this idea. Instead, we can use the bind method that binds the checker to the DUT at compilation time, with few lines of code:

- for QuestaSim®, add to the line that calls the file of the DUT the option -mfcu -cuname along with the path to the checker `../src/rtl/*DUT*.sv -mfcu -cuname ../../unit-verification/*path to the checker*`

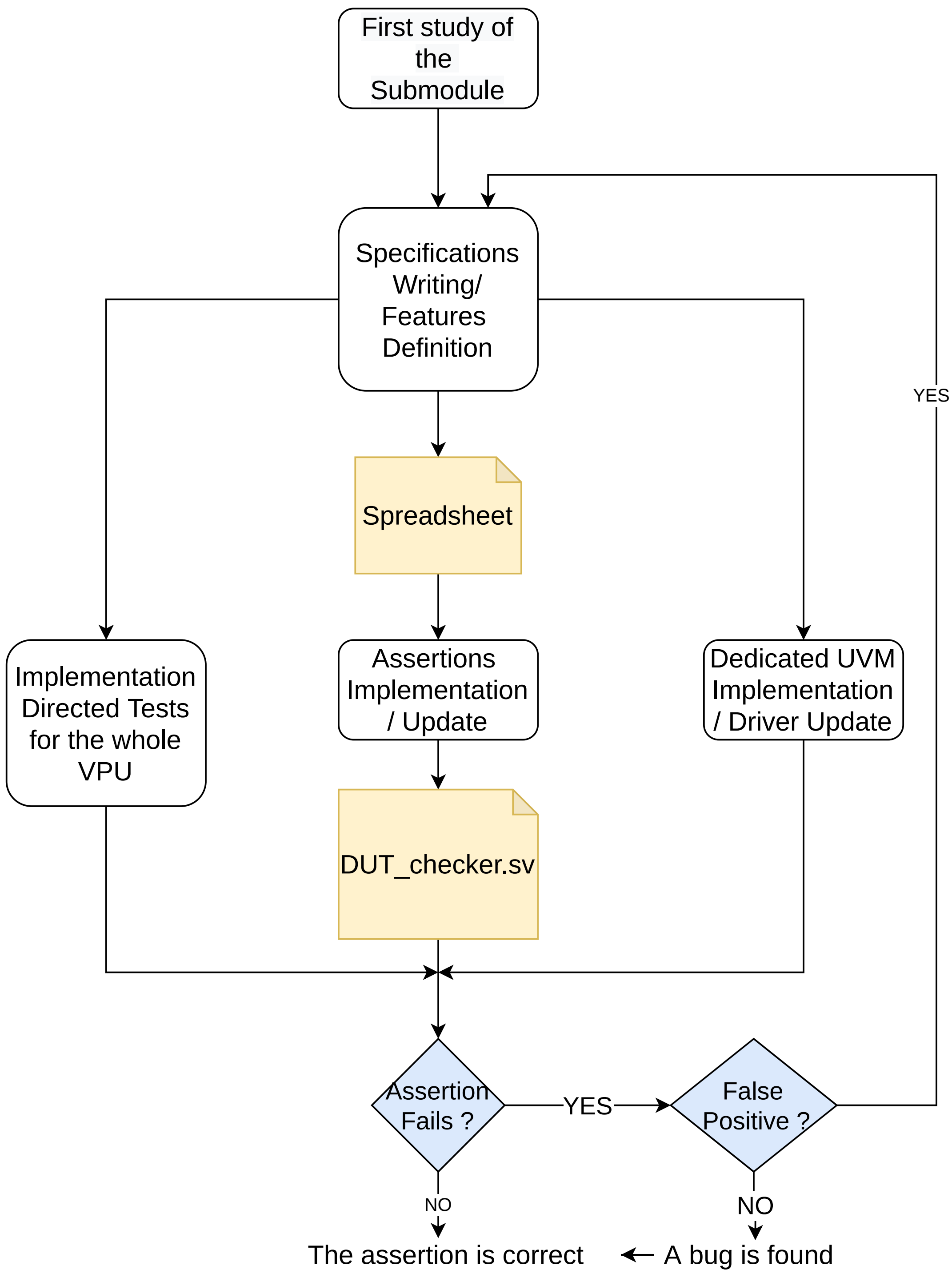

First study of the Submodule
Specifications Writing/ Features Definition
YES
Spreadsheet
Implementation Directed Tests for the whole VPU
Assertions Implementation / Update
Dedicated UVM Implementation / Driver Update
DUT_checker.sv
Assertion Fails ?
YES
False Positive ?
NO
NO
The assertion is correct
A bug is found

Figure 3.8. Work Flow

- include the checker too.

With this scheme, the two files are completely independent and we are able to insert the assertions in all the tests of the VPU, in all the test of the dedicated UVM and in the Formal tool too without any effort from the Designers and in a very elegant and efficient way.

As previously said, each checker contains all the assertions relative to a DUT. These assertions are directly wrote down from the spreadsheet containing all the features of a DUT. To make an example and to show the method followed, in 3.1, 3.2 and 3.3 we can find the spreadsheet (the Verification Plan) for the Ring Logic. Naturally, the aim of this Tables is not to describe all the assertions for the Ring Logic but it rather is to report a developed Verification Plan for a single module, indeed these Tables are very specific and hard to understand if the reader is not working on the project but they can briefly give the idea of the work carried out for each submodule.

In these Tables, two different categories of assertions and assumptions clearly emerge:

1. the ones targeting the handshake signals, the vast majority of properties:
    - **Assumptions** on the inputs, to check that the input signal arrive according to the specifications: for example, the signals that last one clock cycle have to last one clock cycle, the signals that cannot arrive if the DUT has a busy flag must remain to 0.
    - **Assertions** on the handshake signals, for example how the instructions are granted or similar.
2. **Assertions** on the computed output: *assertions from 1.0.12 to 1.0.19 and assertion 1.2.3* check not only the communication signals but that the computation of the output (in this case the calculated address) has been handled correctly. This Scoreboard is implemented using SystemVerilog tasks, and it is performed with high-level sequential code.

	ring_logic module	Name	
1.0.1	all outputs to 0 at reset and 1 clk cycles later	a_rsn_i	Assertion
1.0.2	no unknowns for the inputs when start_i is 1	a_start_i_known	Assumption
1.0.3	if rgather_op_i ==1 no unknowns for rgather_lane_dst_i and rgather_bank_dst_i	a_rgather_op_i _known	Assumption

1.0.4	When req_o ==1, check data_o for unknowns – if first bit (data_enable) is 1 then no unknowns for the rest of the bitvector	a_req_o_known	Assertion
1.0.5	req_o is a single cycle operation	a_req_o	Assertion
1.0.6	only gather or accum or widening can be = '1' at time	a_exclusive_input	Assumption
1.0.7	start_i is a one operation cycle, after it the offset_i is availble on the input	a_start_i_behav	Assumption
1.0.8	send_i lasts just one clock cycle	a_send_i	Assumption
1.0.9	data[22] == 0 && req_o == 0 1 the cycle after	a_kill_i	Assertion
1.0.10	when kill_i, the same cycle !send_i && !enable_i && !start_i;	a_kill_i_data_i	Assumption
1.0.11	when rsn_i , inputs to 0	a_rsn_i_data_i	Assumption
1.0.12	check if data_o[14:12] corresponds to the bank of the current sender	a_data_o_bank _in	Assertion
1.0.13	check if data_o[17:15] corresponds to the sub_bank of the current sender	a_data_o_sub_ bank_in	Assertion
1.0.14	check if data_o[20:18] corresponds to the lane of the current sender	a_data_o_lane _in	Assertion
1.0.15	check if the data_o[2:0] corresponds to the sub_bank of the address destination	a_data_o_sub _bank_out	Assertion
1.0.16	check if the data_o[11:9] corresponds to the lane of the address destination	a_data_o_lane _out	Assertion
1.0.17	check if the data_o[5:3] corresponds to the bank of the address destination	a_data_o_bank _out	Assertion
1.0.18	check if the data_o[8:6] corresponds to the line of the address destination	a_data_o_line _out	Assertion
1.0.19	check data_o[22] which has to be 1 for the indx_ring_logic doing a gather with vs1[i]>VL_MAX, otherwise 0	a_data_o _zeroing	Assertion

Table 3.1: Ring Logic Assertions

	data_ring_logic module	Name	
1.1.1	with (gather_op='1' or accum_en='1' or widening_i='1') , after send_i ='1', data_o on the out the next cycle	a_data_after_send	Assertion
1.1.2	after send_i='1' with ready_o = '1', req_o = 1 (one req for each send probably)	a_req_after_send	Assertion
1.1.3	when req_o rises fetch_o fells (in the same cycle)	a_fetch_on_req	Assertion

1.1.4	when req_o rises ready_o fells (in the same cycle)	a_ready_on_req	Assertion
1.1.5	fetch_o ='1' when send_i ='1' and ready_o = '1'	a_fetch_on_send	Assertion
1.1.6	no send_i ='1' if fetch_o ='0' and ready_o = '1'	a_send_on_fetch	Assumption
1.1.7	when req_o rises, free_i fells in the same cycle	a_free_on_req	Assumption
1.1.8	when enable_i rises (for a clock cycle) the same cycle offset_i is valid (no unkonwn) in input	a_offset_known	Assumption
1.1.9	if gather_op='1' and send_i='1', the same cycle no unknown on rgather_*	a_data_gather	Assumption
1.1.10	no enable_i='1' when accum_en_i='1'	a_data_accum_en	Assumption
1.1.11	no send_i when free = '0'	a_data_send_free	Assumption
1.1.12	if gather == '0' && accum_en == '0' && widening == '0' && enable == '1' ==>type_i and offset and aux_slide not unknown	a_data_slide_i	Assumption

Table 3.2: Data Ring Logic Assertions

	indx_ring_logic module		
1.2.1	fetch_o falls when send_i rises	a_index_fetch _send	Assertion
1.2.2	if gather_op='1' and send_i='1', the next cycle rgather_* = '0'	a_index_gather	Assumption
1.2.3	between the rise of gather_op_i and its fall, the number of send_i is equal to the number of req_o (i.e. no data corruption)	a_index_send_req	Assertion

Table 3.3: Index Ring Logic Assertions

All this collection of assertions and assumption extensively checks the correct behaviour of the module and, as expected, are significately useful, especially when inserted in the high-level VPU testing. Moreover, we can notice how we can generate up to 31 properties for the Data Ring Logic and 22 for the Index Ring Logic, 7 of which work performing arithmetic and sequential scoreboarding. It is intuitive to see how focusing on a submodule can increase the number of control points within the VPU and increase the controllability and observability of the whole system.

3.2.2 Dynamic Functional Verification

Here we are going to show the different ways we had to simulate the Ring Modules. Basically, there were the tests of the whole VPU and the dedicated UVM.

FV in the whole UVM

As reported in Section 2.1, many tests are performed every day. Each test generated by riscv-dv contains hundreds of randomly generated assembly instructions and extensively tests the VPU. In this framework, the checkers were binded to the different DUTs. Given the nature of our Assertions and Assumptions and the unstability of the specifications, within the VPU none of them was marked as a **$fatal** that would stop the execution. Even in the last steps of the work, when the specifications were stable and fixed and no false positives happened, the assertions and assumptions were marked as **$errors** which are reported in the simulation report file. This difference means that the firing assertions had to be checked "manually" by us and then discussed with the Design Team.

All the checkers relative to the ring contain a total of 98 assertions and assumptions and resulted to give a significant contribution to the debugging at a low cost in terms of computational resources: profiling the resources used througout a whole simulation, this value was always below 5%, meaning that they do not appreciably slow down the execution. Simultaneously, as we will report in the next Chapter 4, from the moment the checkers were stable, they found 36% of reported bugs concerning the Ring Modules.

Dedicated UVM

Developing the Dedicated UVM has been another core work of the last six months. In this case it turned out to be a concrete way to fix the attention on the modules and think of them, finding corner cases and defining the specifications, rather than to find bugs. This process reached the peak when coding the drivers. The classical way for doing this was the following:

- individuation of the possible different operations that the DUT can perform.
- understanding of how these operations are issued to the DUT and writing down the waveforms.
- writing the driver in the UVM structure.

For example, we can look at the code for the driver of the ring_data_unpacker: we know that it is a simple component that can receive just one operation (called "req_op") but it is necessary to find how the signals actually behave. Therefore, we first need to write down the waveforms (that are already presented in Figure 3.6) and then we can code the driver: we randomly generate the number of elements that we want to send (`elements_i`) and we send it to the unpacker with `new_instr_i`. At this point we randomize `data_i` and set `req_i` to 1. Finally, we wait for `enable_o` to send other data to the unpacker. We repeat this operation as many times as request by the value fixed by `elements_i` and `sew_i`. Eventually, we set `finish_i` to 1 for a clock cycle.

```
task drive(ring_data_unpacker_sequence_item command);
        int n;
        @(posedge ru_if.clk_i);
        //sync reset
        if(command.op == req_op) begin
                ru_if.op =              req_op;
                ru_if.rsn_i =           1'b1;
                ru_if.kill_i =          1'b0;
                ru_if.new_instr_i =     1'b1;
                ru_if.elements_i =      command.elements_i;
                ru_if.finish_i =        1'b0;
                ru_if.data_i =          command.data_i;
                ru_if.req_i =           1'b0;
                ru_if.enable_i =        1'b0;
                ru_if.sew_i =           command.sew_i;
                -> ru_if.new_input;
                @(posedge ru_if.clk_i);
                ru_if.new_instr_i =     1'b0;

                // wait for the first enable_o = 1
                while(!ru_if.enable_o) begin
                        @(posedge ru_if.clk_i);
                end

                // repeat the unpack sequence at most command.elements_i/(2**(3-command.sew_i)) times
                n = command.elements_i/(64/(2**(3+command.sew_i)));
                repeat(n) begin // wait for enable_o to give a new data
                        // load 64-bit chunk of data
                        randomize(command.data_i);
                        ru_if.data_i =          command.data_i;
                        ru_if.req_i =           1'b1;
                        ru_if.enable_i =        1'b1;
                        @(posedge ru_if.clk_i);
                        ru_if.req_i =           1'b0;
                        // now a grant_o has been given, we can unpack the data
                        while(!ru_if.enable_o) begin
                                @(posedge ru_if.clk_i);
                                ru_if.enable_i = $random;
                        end
                        // the unpacking ended, we can give a new data
                        end

                        @(posedge ru_if.clk_i);
                        while(ru_if.req_o) begin
                                        @(posedge ru_if.clk_i);
                        end
                        while(!ru_if.enable_o) begin
                                        @(posedge ru_if.clk_i);
                        end
                        @(posedge ru_if.clk_i);
                        ru_if.finish_i =        1'b1;
                        @(posedge ru_if.clk_i);
                        ru_if.finish_i =        1'b0;
                end

                if(command.op == rsn_op) begin
                        ru_if.op =              rsn_op;
                        ru_if.rsn_i =           1'b0;
                        ru_if.kill_i =          1'b0;
                        ru_if.new_instr_i =     1'b0;
                        ru_if.elements_i =      command.elements_i;
                        ru_if.finish_i =        1'b0;
                        ru_if.data_i =          command.data_i;
                        ru_if.req_i =           1'b0;
                        ru_if.enable_i =        1'b0;
                        ru_if.sew_i =           0;
                        -> ru_if.new_input;

                end
endtask : drive
```

Most importantly, finding a way to correctly drive the DUT was not just a theoretical exercise but played a key role in the definition of the specifications and in finding reduntant signals or missing or bad-defined features, which may be properly classified as finding bugs.

The architecture of the UVM is illustrated in Figure 3.9. The direction chosen was to have a unique environment for all the ring modules, to avoid redundancy since we decided that having a complete UVM for the simplest components like the node or the packer would have been not efficient for our purposes. Thus, in the environment are instantiated as many Agents as the Submodules and each Agent can be active or passive, this means when we are testing a module, its Agent is the only one Active, i.e. featuring a sequencer and a Driver, while all the other ones cannot drive anything. As desired, this structure is fully re-usable and allows to easily create new tests, insert new operations or override the Drivers. This structure has been organized to work with a simple python3 bash command:

```
python3 compile_and_sim_unit.py –u ring –D *name of the
   ring submodule* –t *name of the test*
```

So far, for every unit, a constrained test has been developed that is able to correctly stimulate all the assertions. However, we did not fill the coverage and scoreboard modules: even in this UVM we used just the checkers to control that the DUT was working properly. This is because we gave much more importance to the High-Level VPU testing and also because this dedicated approach may be optimal only on the final stages of the Architecture design, when small directed tests have to be performed in case the testing the VPU did not complete all the coverage on the component.

Test Plan

A part from the directed tests on the single Submodules, we decided to implement hand-written assembly code to stimulate the whole Ring within the VPU in order to have directed tests to check the basic functionalities of the Ring. This has been actuated with the implementation of 11 binaries addressing corner cases for the different types of operations that the ring performs. The tests were the following:

- `vrgather (vd[i] = vs2[i]>VL_MAX ? 0 :  vs1[vs2[i]])`: for the gather instruction all the corner cases are obtained by modifying the content of `vs2`.

  1. `vd[i]=vs1[VLMAX-i];` to reverse a whole vector.
  2. `if(i%2) vs2[i]=i; else vs2[i]>VL_MAX;` to alternate zeroing (`if ( i%2 ) vd[i] = vs1[i]; else vd[i]=0;`).
  3. `if(i%2) vd[i]=vs1[i+1]; else vd[i]=vs1[i-1];` with SEW 64, to alternate the element among the Lanes. In this case, element 0 will go to element 1 and viceversa, 2 to 3 and 3 to 2, and so on alternating the source and destination Lane.
  4. `if(i%2) vd[i]=vs1[i+1]; else vd[i]=vs1[i-1];` with SEW 32. In this case, `vs2` is equal to the case before, to alternate the elements among the subbanks. Therefore, no element will change Lanes from the departure to the arrival.
  5. `vs2[i]>VLMAX` to have zeroing on all elements.
  6. `vs2[i]=3` to have the same element distributed to the whole destination vector.

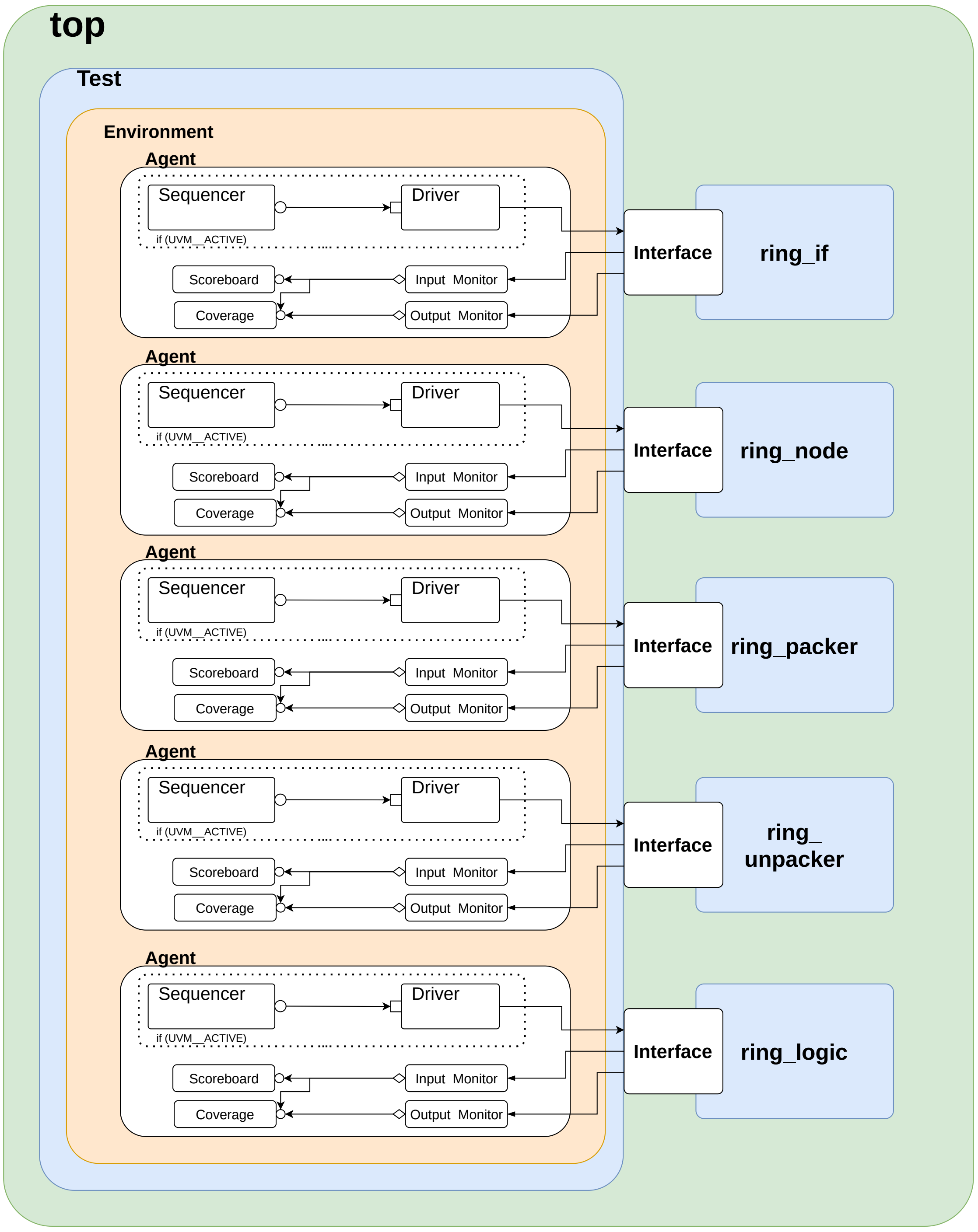


Figure 3.9. Dedicated UVM Structure

- `slide`

  1. `vslideup vd,vs1,VLMAX` in this case the result will be `vd[VLMAX]=vs1[0]` while the other elements will not be valid.
  2. `vslidedown vd,vs1,VLMAX` in this case the result will be `vd[0]=vs1[VLMAX]` while the other elements will not be valid.
  3. `vslidedown vd,vs1,5` with SEW=8 it is a stressing case for the Ring and stress much the packer and unpacker.

- `reduction` a standard reduction with the max operator is `vredmax.vs vd, vs2, vs1, vm` which means vd[0] = max(vs1[0] , vs2[*]) given * every active element.

  1. `vs2` with values in ascendent order.
  2. `vs2` with valus in ascendent order as they were unsigned, but the reduction is signed.

All those tests, which are rapid tests with few instructions have been integrated in the sanity check tests collection and are performed every night as regression tests. Anyway, it was very interesting to go in depth and see the whole instruction's flow from Avispado to the ring and it required a clear view of the VPU, the VRF management and its organization.

3.2.3 Static Functional Verification

Formal Verification is the last method we applied for the Verification of the Ring and it turned out to be a extremely powerful tool for the Ring components. As said in Chapter 1, the main advantages of Formal Verification is that it does not require to think of corner cases nor a complex structure, indeed, it just needs the DUT and the checker module. On the other hand, it is higly resource consuming given the exponential nature of the problem.

This method was applied in the early stages of the work and turned out to be very useful for simple components as the node, the packer and unpacker. In those cases, it spotted many missing assumptions. Indeed, when an assertion failed it was often because of a missing assumption that was immediately added. On the other hand, it turned out to be not so powerful when coming to the ring_logic module, which is the most complex studied here. This is because, the scoreboard is written in `tasks` and every time the tool sees a task, it treats the module as a black-box. Therefore the only assertions that could have been tested where the ones on the communication signals and handshakes.

As said in Section 1.2, the Formal approach is not the perfect fit for the current developing stage of the project: it is optimal for the first *bug-hunting* phase and for the last *assurance* phase. However, it turned out to be more useful in checking the assertions and the assumptions: we used it in the first phase of the work and it allowed to find several missing assumptions and writing down the specifications, thanks obviously to a confront with the Design Team. In the last part of the work, it did not find any bug: indeed the only bugs found, as we will see in the next Chapter, were due *Assumptions* failing, which it is not possible to happen in the Formal tools (for they do not violate Assumptions), and errors on the Assertions based on the tasks.

Nevertheless, we have created a ready-to-use Formal Structure that will be used before the tape-out on the submodules, to check their basis functionalities and that can be launched with a simple script command:

```
python3 compile_and_verify.py -u ring -D *name of the submodule*.
```

In this Chapter, the implemented Verification Plan has been described while in the next one we are going to show the results that it has achieved, highlighting some of the bug founds, the material created and the final consideration on the work.

Chapter 4

Contributions and Discussion

In this Chapter we are going to show the main contributions made by the work presented and discuss them. Two main contributions can be identified by this Assertion-based Verification: the found bugs and the material created. In the first Section we are going to present the Reported Bugs and comment them, while in the second Section we are going to report the Material Created. Finally, there will be the Discussion Section.

4.1 Reported Bugs

At the very beginning of this work, the Ring Modules were almost stable modules: this means that they encountered only minor changes in the interfaces and all the needed functionalities were already featured. Moreover, the studying of the modules, of the VPU and of UVM took a great portion of time, along with the Verification Plan development. Nevertheless, we were able to find different functional bugs which can be categorized in two types:

- Bugs on **Assertions**: bugs on assertions are bugs existing due to a error in the submodule under test. This means that given the same input, the checker and the DUT return a different output value.
- Bugs on **Assumptions**: bugs on assumptions are bugs existing due to an error in the module driving the DUT. In this case, the input is not allowed to be as it is and causes the DUT to fail.

In this Section we are going to show some of the found bugs, of both types. These bugs were all found during the symulation tests of the whole VPU and the presence of the checkers actually eased and speeded up the targeting of the bugs.

4.1.1 Bugs on Assumptions

Naturally, this kind of bugs can be found only in the simulation of the whole VPU since the dedicated UVM is written to properly drive the DUT and naturally this is a great advantage in the Verification flow. An example of this happening is the following: the *ring_data_unpacker*, as reported in Chapter 3, is issued with an operation with the signal `new_instr_i` along with `elements_i` that tells of how many elements will have been sent in output at the end of the operation. So, as the ring_data_unpacker receives 64 bits at a time, every time it receives some valid data (`req_i = 1`), at most 64/SEW valid elements are processed and will eventually exit from the unpacker. This behaviour means that the number reported from `elements_i` has to be greater or equal than the number of `req_i` $\cdot 64/SEW$ (not exactly equal because `finish_i` may arrive without caring that all the requested elements have actually been unpacked). This means that this property can be **assumed**:

```
property p_elements_i_req_i;
 int elements;
 @(posedge clk_i)
 new_instr_i ##0 (1, elements = elements_i/(2**(3-sew_i)) + 1) |->
 first_match((1, elements = elements - (req_i && enable_o)) [*0:$]
     ##0 finish_i) |->
 elements >= 0;
endproperty : p_elements_i_req_i
```

What happened is that the unpacker received `elements_i = 2` and then five `req_i`, as showed in Figure 4.1. Having this assumption failing allowed the Design

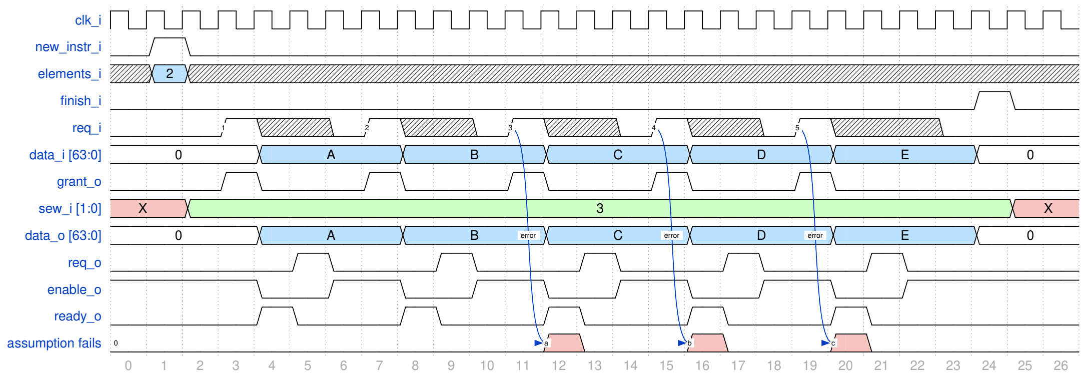


Figure 4.1. Ring Data Unpacker with error

Team to trace back the error directly on the Source Buffer. This error appeared for the execution of a masked reduction: in this case the reduction wrapper did not update the internal register containing the mask, when the `source_buffer` to whom it has to send the data, was not available to read them (with `source_buffer_ready_q = 0`).

Most importantly, this bug was found in a test that was not failing on a Ring operation. This kind of bugs are called *silent bugs*: or the scoreboard of Spike was giving the same results of the VPU and the tests were executing correctly, or failing during a later executed instruction, because the bug leaves the VPU in an incorrect state. In this case this simple assumption was able to spot a silent error that would have come up only later and would have been much more difficult to identify, since the source of the error is very distant from its effect.

In cases like this the power of Assertion-based Functional Verification is clear and bright for everyone. Also, this highlights how Verification is everything but a trivial task: even when good specs are available, performing a double check of the DUT against different reference architectures is an optimal approach to find more bugs that would not be found otherwise.

4.1.2 Bugs on Assertions

The developed checkers found two major bugs on the *kill* mechanism for the Data Ring Logic when performing a gather and here we are going to describe them. Basically, the `kill_i` signal is sent by Avispado when the inflight instructions have to be killed and works as a reset of the VPU. The correct working mechanism of the gather in the `data_ring_logic` is described in Figure 4.2. The `start_i`, `rgather_op_i` and `enable_i` signals tell us that a gather operation has come to the Ring Logic. Then, the destination (`rgather_lane_i`, `rgather_bank_i`, `rgather_sub_bank_i`) is sent along with the `send_i` signal. At this point, the Data Ring Logic puts the data in output `data_o` and signals it with the signal `req_o`.

As said in Chapter 3, we can observe two kind of assertions, and therefore two kinds of errors:

- errors in the handshake,
- mismatches with the scoreboard.

and we experienced both of them for the Ring Logic. The assertion of the first type that fails is the following:

```
//1.2.3 vp
property p_index_send_req;
 int num_send;
 @(posedge clk_i)
 ($rose(rgather_op_i), num_send = 0) |->
 first_match( (1, num_send = num_send + send_i - req_o) [*0:$] ##0
     $fell(rgather_op_i) ) |->
 num_send == 0;
endproperty : p_index_send_req
```

basically, it **asserts** that, while `r_gather_op_i=1`, the number of `req_o` is equal to the number of `send_i`. This simply means that no data corruption has occurred: indeed, if the number of received element is different to the number of sent elements, it means that the Ring is or losing or creating information and neither of them is good.

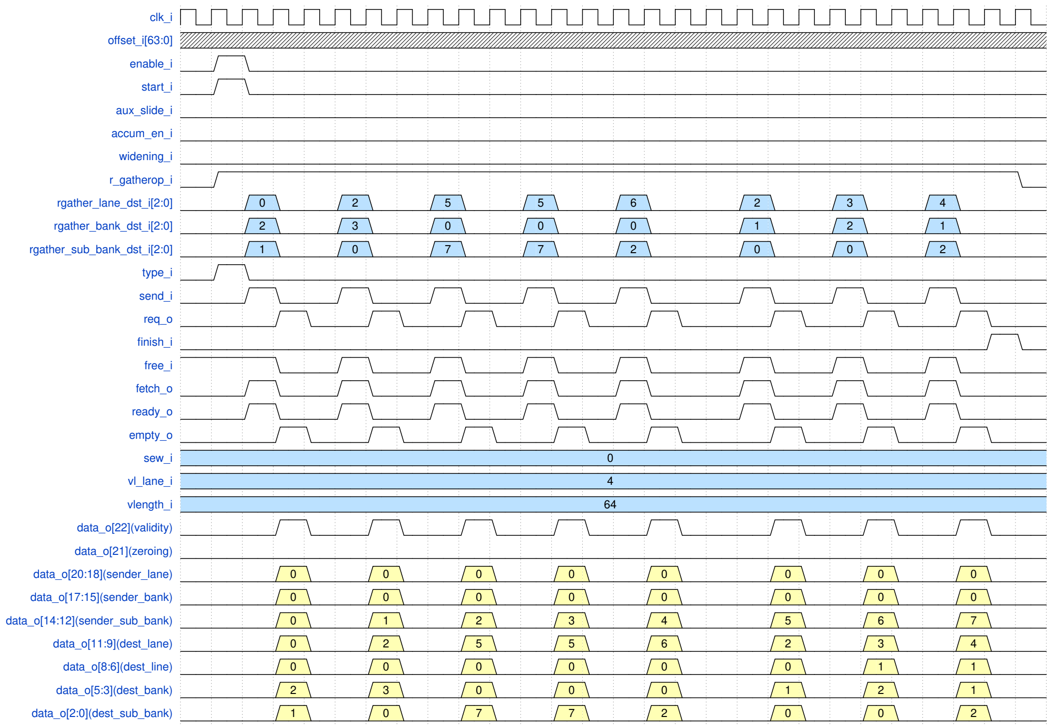


Figure 4.2. Data Ring Logic Gather

The assertions failing relative to the second type are the following two:

```
//1.0.12 vp
property p_data_o_bank_in;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |->
 data_o[17:15] == current_bank_int;
endproperty : p_data_o_bank_in

//1.0.13 vp
```

```
property p_data_o_sub_bank_in;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |->
 data_o[14:12] == current_sub_bank_int;
endproperty : p_data_o_sub_bank_in
```

These 2 assertions check that the computed source bank and sub_bank are the same for our scoreboard and for the DUT. As showed in Figure 4.2, the Data Ring Logic sends the data starting from element number 0 to element VL_MAX, and then it increments for each `send_i` that arrives. So, the source addresses have to be correct according to the current `element_id`.

The 3 assertions presented above where failing basically due to the same reason: a bad handling of the `kill_i` signal. Basically, after the kill signal, the reset of the internal counters due to `start_i` was not executed properly. As a consequence the Data Ring Logic started sending the elements not from `element_id = 0` but `=1` and the number of `req_i` and `send_i` was not equal, as depicted in Figure 4.3, causing the simulation to time-out.

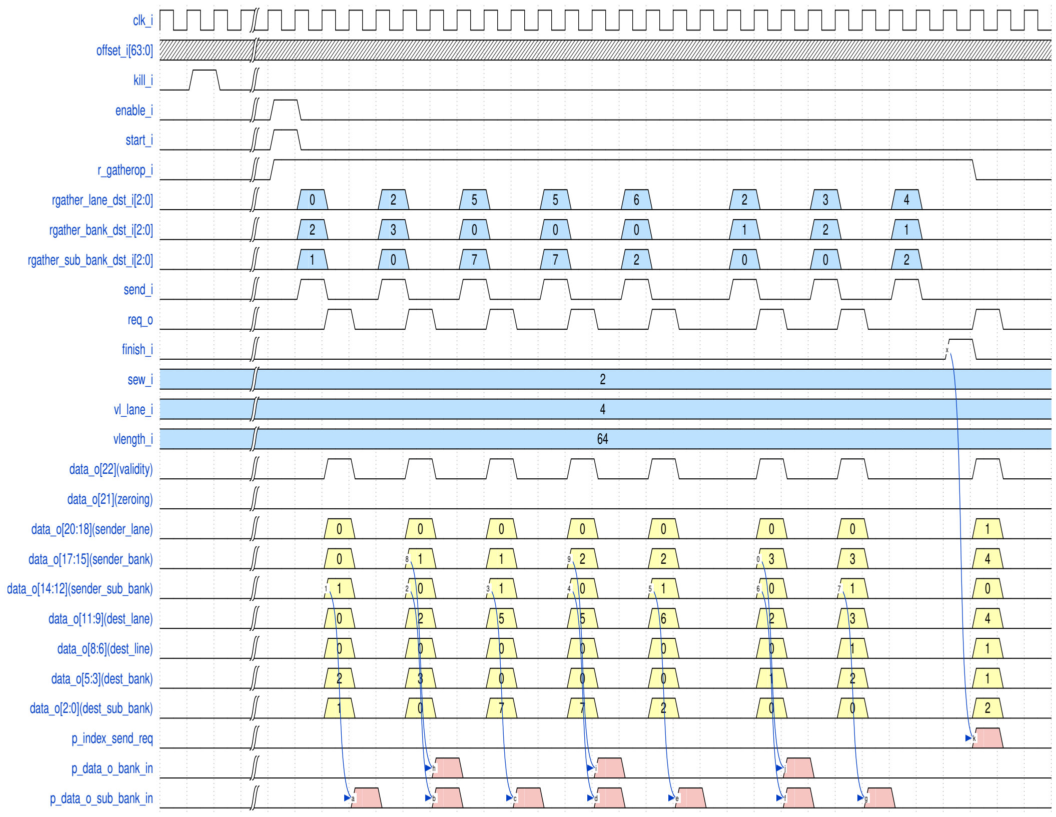


Figure 4.3. Data Ring Logic Gather with error

Even in this case, it was a *silent bug*: the simulation report of the VPU was correctly saying that the execution stopped due to a time-out error during a load operation. In fact, the time-out was due to the *rgather* operation performed right before than the Load: certainly the presence of the checker played a key role in finding this bug in a straightforward way.

The first 4 months of work were dedicated to the studying of the checkers, the UVM the VPU and to the implementation of the whole Verification Plan described in the previous chapter. Eventually, in the last two months, among the 3000 tests run, 4 major bugs, out of the total 11 bugs concerning the Ring, were found thanks to the checkers. This means that the 36% of bugs that were found since when the Verification Plan's structure was functioning, were found by the checkers. Figure 4.4 illustrates the sources of the remaining 64% of bugs: 18% of them were reported due to a Synthesis error, which means errors due to floating wires and oversized signals, the other 46% was spotted without the contribution of the assertions: they were spotted from the Design Team by looking at failing simulation and tracing back the error to the Ring from the waveforms.

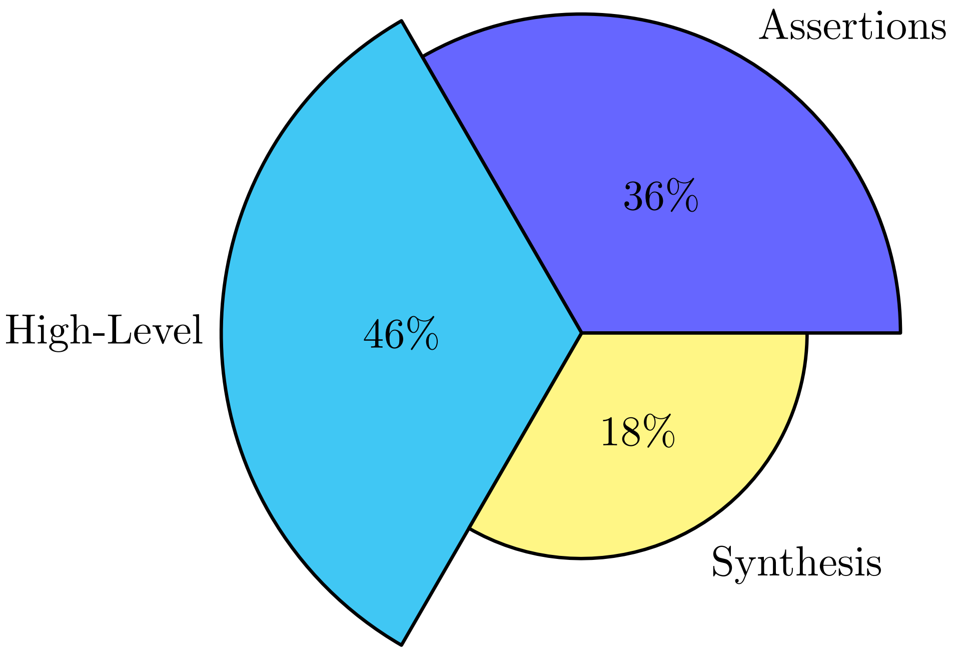


Figure 4.4. Percentage Found Errors

This percentage is very significative: naturally, it does not mean that these bugs would have not been found eventually otherwise, but it indicates the power of this approach and the actual speed-up we had thanks to the checkers. This percentage is in line with many studies that have been carried out to support this thesis, for example, the first paper done in this direction claimed [15] that 34% of the found bugs in the checking of the DECchip 21164 Alpha microprocessor, were found by assertion. Also, [23] describes how the assertion-based approach can successfully address the Functional Verification of a chip-multi-threaded microprocessor and reports that in that project it achieved a reduction of the debug time by more than 50%.

Many of the assertions could have been developed also from the Design team. Indeed, the developed assertions are *low-level* assertions that check the base functionalities of the DUT and can be easily developed by the designer within the DUT, and may result as practical as the ones in the checkers. This is especially valid for the assumptions, which are supposed to be crucial for the Designer. As a matter of fact, the connection with the Design Team was intense and higly profitable. However, the Design Team was missing the time and chance to do so, therefore, this work was carried out in order to give a proper structure and organization and to create a applicable Verification Plan for the Ring modules that would not have been made otherwise.

4.2 Material Created

Apart from the found bugs and the advantages showed in the previous Section, the presented work has produced a considerable amount of material and experience that will be able to turn out useful for the Verification Team.

First of all, it provided the creation of the 17 pages of missing technical specifications for the Ring Modules. The specification produced are low-level specifications that freeze the functionalities of the Ring Modules to the moment they were wrote down. Naturally, these modules can change and have actually changed throughout the last 6 months. Though, now it is possible to say that these modules are stable and only minor changes to improve the performances may occur, as a consequence, the specifications result to be accurate and a solid starting point even in the case of changes. This part of the work can be addressed as a fundamental step, indeed, just the process of thinking of components and writing down the functional specifications is doing Verification itself. Also, we provided the 'checker' modules, containing a total of 98 assertions and assumptions and the implementation of the directed binaries for the whole VPU.

Secondly, this project carried out the first consistent work on Formal Verification in the Verification Team, along with the basic documentation and scripting needed to ease further attempts (especially before the tape-out that is planned in the next months). As said in 3, Formal Verification does not need any complex structure, still we developed the needed basic documentation. Also, we implemented the bind method to insert the checkers in the tests of the whole VPU and to perform the Formal Verification: naturally, as no low-level Verification was made until that moment, this method was never used in the whole project until then.

Lastly, the creation of a dedicated, complete and re-usable UVM for the Ring and for each module in it. The hope is that it will come in useful in the final steps of the Verification process: so far the coverage-driven process is only at VPU-level and no low-level coverage exists. Hopefully, this work will be done in the next phase of the Verification and the UVM will be needed to reach 100% of coverage.

4.3 Discussions

The work carried out brought many advantages on the Verification for the Ring Modules and, most importantly, it highlighted the utility of Assertion-based Verification that was before missing due to lack of time, people and specifications on the submodules. Certainly, developing from scratch a whole VPU targeting HPC applications and generating a complete Verification structure is a complex task that requires much time. However, this approach showed much potential and showed its usefullness to perform a high-quality Verification. Indeed, before this work, the only possible approach to individuate bugs was the testing of the VPU against Spike and tracing back the error from the waveforms.

Naturally, testing the VPU against Spike is the first capital step required to verify the VPU. Also, since the specifications of the VPU are provided by RISC-V, the Verification Team has a perfect reference to develop an error-free scoreboard. However, this approach is not optimal from the Verification point of view: having only a VPU-level testing makes hard the debugging for the Design Team. It is possible to find an optimal point between VPU-level testing and low-level testing: apparently, the complexity arises for the Verification Team when looking on the single submodules, in fact, the whole Verification process can significatively take advantage from it and became more and more effective and efficient. Naturally, the low-level approach requires much effort and time and, at some point, the work done to develop the submodules checkers and dedicated UVM may be wasted time. Conceptually, we can plot the Verification efficiency against the low-level approach as in figure 4.5. Here, the word *efficiency* indicates the potential speed-up in finding bugs thanks to the low-level approach. Without low-level approach the Verification is sub-optimal since the observability and controllability are not sufficient, on the other hand, it is possible to reach over-controllability and over-observability, saturating the efficiency, reduntantly verifying modules that are already error-free.

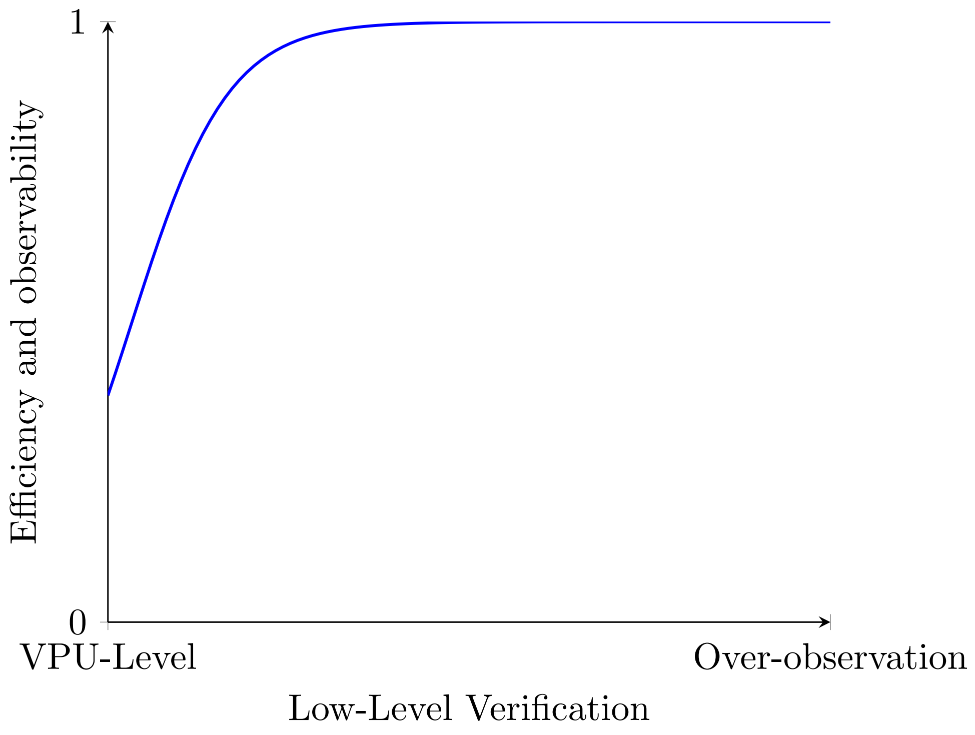


Figure 4.5. Verification Efficiency

Finally, it has been learned that the only activity of thinking about the DUT is the first step of the Verification and maybe is the most important one: when writing an assertion, it is crucial to have understood the DUT and if a specification is missing or if there are conflicting behaviours, those can be spotted even without Dynamic or Static Verification.

Chapter 5

Conclusions

5.1 Conclusions

This work focused on Assertion-Based Verification on the Ring Submodules of the VPU. The first step consisted in developing a Verification Plan aiming to extensively verify each one of the Ring components and that has led to the *creation of detailed specifications, creation of a dedicated UVM and dedicated binaries.*

The studying of the modules started with the studying of Vector Processing and the RISC-V Vector Extension Instruction Set. Then, we focused on the VPU, its organization and on the operation stimulating the targeted submodules and finally we fixed our attention exclusively on the Ring modules. This top-down approach turned out to be fundamental to properly verify submodules in a complex system.

At the beginning, the Assertion-based Verification was quite a mistery to me and I also proceeded attempt after attempt but eventually the results were in line with the expectations, or even better. Indeed, we were able to find different bugs on the Ring Modules, and produce experience and material, that will hopefully turn out useful for the Verification Team.

The Assertion-Based approach has emerged throughout the work as a key factor to obtain a reliable Verification of a complex system such as the VPU: since the moment when the Verification plan was functioning, it was able to find the *36% of bugs* concerning the Ring Modules.

As a matter of fact, this work confirms the importance of Assertions in improving observability and controllability of complex DUTs and it is a necessary method to cope with complex systems that have to be verified not only at System-level but at the same level where the code is "hand-written" by the Designers.

5.2 Future Directions

The Verification Team for the VPU was born just a year ago and it has made a huge work to create a solid Verification infrastructure from scratch.

Stemming from this work, three main tasks can be addressed as the next ones:

1. Increment the effort on the Formal Verification of the Submodules.
2. Extend the bind method to insert the coverage on the submodules, to check if the VPU-Level testing is actually covering all the corner cases.
3. Extend the Assertion-based Verification to all the necessary VPU's modules.

Appendix A

Checkers

A.1 ring_if

```
import EPI_pkg::*;

bind ring_if ring_if_checker #(
        .LANES(LANES),
        .WIDTH(WIDTH),
        .DATA(DATA)) bind_ring_if_checker (.*);

module ring_if_checker
    #(
    parameter LANES = 8,
    parameter WIDTH = 672,
    parameter DATA = 84
    )(

    input                       clk_i,
    input                       rsn_i,
    input                       stall_i,
    input                       req_i,
    input                       accum_en_i,
    input [3:0]                 req_cnt_i, // THIS IS USED TO TELL HOW MANY VALID
        ELEMENTS THE INTERFACE HAS RECEIVED (COUNTS THE PROCESSED ELÆŔMENTS)
    input [WIDTH-1:0]           data_i, // From quick calculations: data_enable
        (1-bit), lane_id (3-bit), line_id (5-bit), bank_id (3-bit), data (64-bit)
    input [VLEN_WIDTH-1:0]      length_i, // CHANGED TO HOLD THE MAXIMUM NUMBER OF
         ELEMENTS (2048?) -> MIGHT BE CHANGED TO 12 BITS
    input                       free_o,
    input [LANES-1:0]           we_o,
    input [WIDTH-1:0]           data_o,    // Data for each lane (64-bit per lane)
         -> To be added: bits for the line offset, data enable,
    input                       finish_o,    // bank_id for the position in the wb
         buffer (maybe internally solved)
    input                       kill_i,
    input                       enable_i,
    input                       wb_ack_i
    );

//could be useful
wire data_valid_o[7:0];
logic data_v_o;
```

```

assign data_valid_o = {data_o[DATA-1], data_o[2*DATA-1], data_o[3*DATA-1], data_o[
    4*DATA-1], data_o[5*DATA-1], data_o[6*DATA-1], data_o[7*DATA-1], data_o[8*DATA
    -1]};

assign data_v_o = data_o[DATA-1] || data_o[2*DATA-1] || data_o[3*DATA-1] || data_o
    [4*DATA-1] || data_o[5*DATA-1] || data_o[6*DATA-1] || data_o[7*DATA-1] ||
    data_o[8*DATA-1];


//////////////////////////////////////////////////////
//////////////////////processes//////////////////////
//////////////////////////////////////////////////////

int counter;

always @(posedge clk_i)
        begin
                if ((rsn_i == 1'b0) || (kill_i == 1'b1)) begin
                        counter = 0;
                end

                else if (enable_i == 1'b1) begin
                        counter = counter + 1;
                end

                else if (finish_o == 1'b1) begin
                        counter = counter - 1;
                end

                else begin
                        counter = counter;
                end

        end


//////////////////////////////////////////////////////
/////////////////////properties//////////////////////
//////////////////////////////////////////////////////


//2.01 vp
property p_rsn_i;
        @(posedge clk_i)
        !rsn_i |-> data_o == '0 && we_o == '0 && free_o ##1 data_o == '0 && we_o
            == '0 && free_o;
endproperty : p_rsn_i

//2.02 vp
property p_rsn_i_data_i;
        @(posedge clk_i)
        !rsn_i |-> !kill_i && !stall_i && data_i == '0 && req_cnt_i == '0 && !
            req_i && !enable_i && length_i == '0 && !accum_en_i;
endproperty : p_rsn_i_data_i

//2.03 vp
property p_free_o_high;
        @(posedge clk_i)
        req_i |-> ##[1:$] free_o;
endproperty : p_free_o_high

//2.04 vp
```

```
property p_free_o_low;
        @(posedge clk_i)
        req_i |-> !free_o;
endproperty : p_free_o_low

//2.05 vp
property p_free_o_stable;
        @(posedge clk_i)
        free_o ##1 !req_i |-> free_o;
endproperty : p_free_o_stable

//2.06 vp
property p_free_o_timeout;
        int i = 9;
        @(posedge clk_i)
        ( req_i, i = 9 ) |-> first_match(( 1 , i = i - !stall_i ) [*0:$] ##0
            free_o ) |-> i >= 0;
endproperty: p_free_o_timeout

//2.07 vp
property p_finish_o_enable_i;
        @(posedge clk_i)
        enable_i |-> enable_i [=1] ##1 finish_o ##1 !finish_o;
endproperty : p_finish_o_enable_i

//2.08 vp
property p_stall_i;
        logic[WIDTH-1:0] past_data_o;
        @(posedge clk_i)
        ( stall_i, past_data_o = data_o ) |-> !we_o ##1 data_o == past_data_o;
endproperty : p_stall_i

//2.09 vp
property p_kill_i;
        @(posedge clk_i)
        kill_i |-> ##1 data_o == '0 && we_o == '0 && free_o;
endproperty : p_kill_i

//2.10 vp
property p_data_i_known;
        @(posedge clk_i)
                req_i |->
                !(( data_i[DATA-1] && $isunknown(data_i[DATA-1:0])) || (data_i[2*
                    DATA-1] && $isunknown(data_i[2*DATA-1:1*DATA])) ||
                (data_i[3*DATA-1] && $isunknown(data_o[3*DATA-1:2*DATA])) || (
                    data_i[4*DATA-1] && $isunknown(data_i[4*DATA-1:3*DATA])) ||
                (data_i[5*DATA-1] && $isunknown(data_i[5*DATA-1:4*DATA])) || (
                    data_i[6*DATA-1] && $isunknown(data_i[6*DATA-1:5*DATA])) ||
                (data_i[7*DATA-1] && $isunknown(data_i[7*DATA-1:6*DATA])) || (
                    data_i[8*DATA-1] && $isunknown(data_i[8*DATA-1:7*DATA] )));
endproperty : p_data_i_known

//2.11 vp
property p_data_o_known;
        @(posedge clk_i)
                data_v_o |->
                !(( data_o[DATA-1] && $isunknown(data_o[DATA-1:0])) || (data_o[2*
                    DATA-1] && $isunknown(data_o[2*DATA-1:1*DATA])) ||
                (data_o[3*DATA-1] && $isunknown(data_o[3*DATA-1:2*DATA])) || (
                    data_o[4*DATA-1] && $isunknown(data_o[4*DATA-1:3*DATA])) ||
                (data_o[5*DATA-1] && $isunknown(data_o[5*DATA-1:4*DATA])) || (
                    data_o[6*DATA-1] && $isunknown(data_o[6*DATA-1:5*DATA])) ||
                (data_o[7*DATA-1] && $isunknown(data_o[7*DATA-1:6*DATA])) || (
                    data_o[8*DATA-1] && $isunknown(data_o[8*DATA-1:7*DATA] )));
endproperty : p_data_o_known

//2.12 vp
property p_req_i_on_free_o;
        @(posedge clk_i)
        !free_o |=> !$rose(req_i);
endproperty : p_req_i_on_free_o

//2.13 vp
property p_req_i_one_cycle;
        @(posedge clk_i)
        $rose(req_i) |-> ##1 !req_i;
endproperty : p_req_i_one_cycle

//2.14 vp
property p_kill_i_data_i;
        @(posedge clk_i)
        kill_i |-> !req_i && !enable_i;
endproperty : p_kill_i_data_i

//2.15 vp
property p_enable_i_no_accum;
        @(posedge clk_i)
        $rose(enable_i) |-> first_match(1 [*1:$] ##0 ( finish_o || !$stable(
            accum_en_i) )) |-> !accum_en_i;
endproperty : p_enable_i_no_accum

//2.16 vp
property p_accum_no_enable_i;
        @(posedge clk_i)
        accum_en_i |-> !enable_i;
endproperty : p_accum_no_enable_i

//2.17 vp
property p_accum_no_finish_o;
        @(posedge clk_i)
        accum_en_i |-> !finish_o;
endproperty : p_accum_no_finish_o

//2.18 vp
property p_data_i_valid_req;
        @(posedge clk_i)
        ( data_i[DATA-1] || data_i[2*DATA-1] || data_i[3*DATA-1] || data_i[4*DATA
            -1] ||
        data_i[5*DATA-1] || data_i[6*DATA-1] || data_i[7*DATA-1] || data_i[8*DATA
            -1] ) |-> req_i;
endproperty : p_data_i_valid_req

//2.19 vp
property p_enable_i;
        @(posedge clk_i)
        enable_i |-> ##1 !enable_i;
endproperty : p_enable_i

//2.20 vp
property p_enable_i_finish_o;
        @(posedge clk_i)
        $fell(enable_i) |=> first_match(1 [*0:$] ##0 (enable_i || finish_o)) ##0 (
            finish_o && !enable_i);
endproperty : p_enable_i_finish_o

```

```
//2.21 vp
property p_req_i_beh;
        @(posedge clk_i)
        req_i |-> (counter == 1) or accum_en_i;
endproperty : p_req_i_beh

//2.22 vp
property p_accum_req_i;
        @(posedge clk_i)
        $rose(accum_en_i) |-> accum_en_i && !req_i;
endproperty : p_accum_req_i

//2.23 vp
property p_rsn_i_beh;
        @(posedge clk_i)
        $fell(rsn_i) |-> ##1 !rsn_i;
endproperty : p_rsn_i_beh


////////////////////////////////////////////////////////
//////////////////assert & assume////////////////////
////////////////////////////////////////////////////////


//2.01 vp
a_rsn_i :                 assert property( p_rsn_i ) else $error("bad output for the
     rsn state");

//2.02 vp
a_rsn_i_data_i :          assume property( p_rsn_i_data_i ) else $error("bad input
    for the rsn state");

//2.03 vp
a_free_o_high :           assert property( disable iff(!rsn_i || kill_i)
    p_free_o_high ) else $error("free_o doesn't return to 1 after (at most) 8
    clock cycle from req_i = '1'");

//2.04 vp
a_free_o_low :            assert property( disable iff(!rsn_i || kill_i)
    p_free_o_low ) else $error("free_o doesn't go to 0 when req_i is 1 ");

//2.05 vp
a_free_o_stable :         assert property( disable iff(!rsn_i || kill_i)
    p_free_o_stable ) else $error("free_o goes to 0 with no req_i asserted");

//2.06 vp
a_free_o_timeout :        assert property( disable iff(!rsn_i || kill_i)
    p_free_o_timeout ) else $error("free_o timeout");

//2.07 vp
a_finish_o_enable_i :     assert property( disable iff(!rsn_i || kill_i)
    p_finish_o_enable_i ) else $error("finish_o goes to 1 with no enable_i
    asserted");

//2.08 vp
a_stall_i :               assert property( disable iff(!rsn_i || kill_i) p_stall_i )
     else $error("ban handle for the stall_i");

//2.09 vp
a_kill_i :                assert property( disable iff(!rsn_i) p_kill_i ) else
    $error("ban handle for the kill_i");

//2.10 vp
```

```
a_data_i_known :         assume property( disable iff(!rsn_i || kill_i)
    p_data_i_known ) else $error("data_i unknown when data_i valid asserted");

//2.11 vp
a_data_o_known :         assert property( disable iff(!rsn_i || kill_i)
    p_data_o_known ) else $error("data_o unknown when data_o valid asserted");

//2.12 vp
a_req_i_on_free_o :      assume property( disable iff(!rsn_i || kill_i)
    p_req_i_on_free_o ) else $error("req_i asserted without free_o = '1'");

//2.13 vp
a_req_i_one_cycle :      assume property( disable iff(!rsn_i || kill_i)
    p_req_i_one_cycle ) else $error("req_i asserted for more than one cycle");

//2.14 vp
a_kill_i_data_i :        assume property( disable iff(!rsn_i) p_kill_i_data_i )
    else $error("invalid inputs on kill_i");

//2.15 vp
a_enable_i_no_accum :    assume property( disable iff(!rsn_i || kill_i)
    p_enable_i_no_accum ) else $error("accum signal during and enable op");

//2.16 vp
a_accum_no_enable_i :    assume property( disable iff(!rsn_i || kill_i)
    p_accum_no_enable_i ) else $error("accum signal during and enable");

//2.17 vp
a_accum_no_finish_o :    assume property( disable iff(!rsn_i || kill_i)
    p_accum_no_finish_o ) else $error("accum signal during and enable");

//2.18 vp
a_data_i_valid_req :     assume property( disable iff(!rsn_i || kill_i)
    p_data_i_valid_req ) else $error("no req_i with data valid");

//2.19 vp
a_enable_i :             assume property( disable iff(!rsn_i || kill_i) p_enable_i
    ) else $error("enable_i lasted more than 1 clock cycle");

//2.20 vp
a_enable_i_finish_o :    assume property( disable iff(!rsn_i || kill_i)
    p_enable_i_finish_o ) else $error("there were 2 enable_i without a finish_o");

//2.21 vp
a_req_i_beh :            assume property( disable iff(!rsn_i || kill_i) p_req_i_beh
     ) else $error("there was a req_i without enable or accum_en");

//2.22 vp
a_accum_req_i :          assume property( disable iff(!rsn_i) p_accum_req_i ) else
    $error("req_i in the same cycle accum_en_i rose");

//2.23 vp
a_rsn_i_beh :            assume property( p_rsn_i_beh ) else $error("rsn_i lasted
    less than 1 clock cycle");

endmodule: ring_if_checker // ring_interface
```

A.2 ring_node

```
import EPI_pkg::*;

```

```
bind ring_node ring_node_checker #(
        .LANES(LANES),
        .WIDTH(WIDTH),
        .LANE_ID(LANE_ID)) bind_ring_node_checker (.*);

module ring_node_checker
    #(
    parameter LANES = 8,
    parameter WIDTH = 84,
    parameter LANE_ID = 3'b000
    )(

    input               clk_i,
    input               rsn_i,
    input               stall_i,
    input               kill_i,
    input               start_i,
    input [WIDTH-1:0]   data_lane_i,
    input [WIDTH-1:0]   data_ring_i,
    input               data_valid_o,
    input [WIDTH-1:0]   data_ring_o,
    input               we_o,
    input               self_write_o
    );

wire[2:0] curr_lane;
assign curr_lane = LANE_ID;



//////////////////////////////////////////////////////
/////////////////////properties//////////////////////
//////////////////////////////////////////////////////




//3.01 vp
property p_rsn_i_outputs;
        @(posedge clk_i)
        !rsn_i |-> !data_valid_o && data_ring_o == '0 ##1 !data_valid_o &&
            data_ring_o == '0;
endproperty : p_rsn_i_outputs

//3.02 vp
property p_self_write_o;
        @(posedge clk_i)
        $rose(start_i) && data_lane_i[WIDTH-1] && ( curr_lane == data_lane_i[WIDTH
            -12:WIDTH-14] ) && !stall_i |->
        (self_write_o && we_o) ;
endproperty : p_self_write_o

//3.03 vp
property p_we_o;
        @(posedge clk_i)
        data_ring_i[WIDTH-1] && ( curr_lane == data_ring_i[WIDTH-12:WIDTH-14] ) &&
             !stall_i |-> we_o ;
endproperty : p_we_o

//3.04 vp
property p_data_lane_propagation();
        logic [WIDTH-1:0] local_var;
        @(posedge clk_i)
        (start_i && !stall_i,local_var=data_lane_i) ##0 data_lane_i[WIDTH-1] && (
            curr_lane != data_lane_i[WIDTH-12:WIDTH-14] ) |-> !we_o && !
            self_write_o ##1  (data_ring_o == local_var) && data_valid_o;
endproperty : p_data_lane_propagation

//3.05 vp
property p_data_ring_propagation;
        logic [WIDTH-1:0] local_var;
        @(posedge clk_i)
        (data_ring_i[WIDTH-1] && !stall_i,local_var=data_ring_i) ##0 ( curr_lane
            != data_ring_i[WIDTH-12:WIDTH-14] )
        |-> !we_o && !self_write_o ##1 data_valid_o && data_ring_o == local_var;
endproperty : p_data_ring_propagation

//3.06 vp
property p_invalid_data;
        @(posedge clk_i)
        !data_ring_i[WIDTH-1] && !data_lane_i[WIDTH-1] && !stall_i |-> !we_o && !
            self_write_o  ##1 !data_valid_o ;
endproperty : p_invalid_data

//3.07 vp
property p_stall_i; //data ring
        logic[WIDTH-1:0] local_data_ring_o;
        logic local_data_valid_o;

        @(posedge clk_i)
        ( stall_i, local_data_ring_o = data_ring_o, local_data_valid_o =
            data_valid_o ) |->
        !we_o && !self_write_o |->  first_match(1 [*1:$] ##0 !stall_i)  |->
            data_valid_o == local_data_valid_o && data_ring_o == local_data_ring_o
            ;
endproperty : p_stall_i

//3.08 vp
property p_kill_i;
        @(posedge clk_i)
        kill_i |-> ##1 !data_valid_o && data_ring_o == '0;
endproperty : p_kill_i

//3.09 vp
property p_rsn_data_i;
        @(posedge clk_i)
        !rsn_i |-> !stall_i && !kill_i && !start_i && !data_lane_i[WIDTH-1] && !
            data_ring_i[WIDTH-1];
endproperty : p_rsn_data_i

//3.10 vp
property p_kill_data_i;
        @(posedge clk_i)
        kill_i |-> !stall_i && !start_i && !data_lane_i[WIDTH-1] && !data_ring_i[
            WIDTH-1];
endproperty : p_kill_data_i

//3.11 vp
property p_start_data_lane_i;
        @(posedge clk_i)
        start_i |-> !data_ring_i[WIDTH-1];
endproperty : p_start_data_lane_i


//3.12 vp
property p_rsn_i_beh;
```

```
        @(posedge clk_i)
        $fell(rsn_i) |-> ##1 !rsn_i;
endproperty : p_rsn_i_beh

/////////////////////////////////////////////////////////
///////////////////assert & assume/////////////////////
/////////////////////////////////////////////////////////




//3.01 vp
a_rsn_i_outputs :                assert property( p_rsn_i_outputs ) else $error("bad handle for the rsn signal");

//3.02 vp
a_self_write_o :                 assert property( disable iff(!rsn_i || kill_i) p_self_write_o ) else $error("didn't self_write_o when supposed to");

//3.03 vp
a_we_o :                         assert property( disable iff(!rsn_i || kill_i) p_we_o ) else $error("no write enable when supposed");

//3.04 vp
a_data_lane_propagation :        assert property( disable iff(!rsn_i || kill_i) ( p_data_lane_propagation or p_stall_i) ) else $error("after a start and a valid data_lane_i, with the data for another lane, data_lane_i was not propagated to the output the following clock cicle");

//3.05 vp
a_data_ring_propagation :        assert property( disable iff(!rsn_i || kill_i) ( p_data_ring_propagation or p_stall_i) ) else $error("wrong data_ring_o when valid data_ring_i the previous clock");

//3.06 vp
a_invalid_data :                 assert property( disable iff(!rsn_i || kill_i) ( p_invalid_data or p_stall_i) ) else $error("bad handle for invalid data");

//3.07 vp
a_stall_i :                      assert property( disable iff(!rsn_i || kill_i) p_stall_i ) else $error("bad handle for the stall signal");

//3.08 vp
a_kill_i :                       assert property( disable iff(!rsn_i) p_kill_i ) else $error("bad handle for the kill signal");

//3.09 vp
a_rsn_data_i :                   assume property( p_rsn_data_i ) else $error("no reset of inputs data when reset");

//3.10 vp
a_kill_data_i :                  assume property( disable iff(!rsn_i) p_kill_data_i ) else $error("no reset of inputs data when kill");

//3.11 vp
a_start_data_lane_i :            assume property( disable iff(!rsn_i || kill_i) p_start_data_lane_i ) else $error("no valid data lane on start");

//3.12 vp
a_rsn_i_beh :                    assume property( p_rsn_i_beh ) else $error("rsn_i lasted less than 1 clock cycle");

endmodule: ring_node_checker // ring_node
```

A.3 ring_logic

```
import EPI_pkg::*;

localparam FIFO_SIZE = 10;

bind ring_logic ring_logic_checker #(
        .LANE_ID(LANE_ID),
        .SLIDE_WIDTH(SLIDE_WIDTH),
        .DATA_OUT_WIDTH(DATA_OUT_WIDTH),
        .IS_INDEX_RING(IS_INDEX_RING),
        .N_LANES(N_LANES),
        .N_BANKS(N_BANKS),
        .SIZE(SIZE),
        .MAX_VLEN(MAX_VLEN)) bind_ring_logic_checker (.*);

module ring_logic_checker
        #(
        parameter LANE_ID = 3'd0,
        parameter SLIDE_WIDTH    = 64,  // Scalar value from the core
        parameter DATA_OUT_WIDTH = 20,
        parameter IS_INDEX_RING = 0, // this is used to distinguish between the data and index rings (may be updated in future versions)
        parameter N_LANES = 8,
        parameter N_BANKS = 5,
        parameter SIZE = 10,
        parameter MAX_VLEN = 256 // CHANGE THIS PARAMETER FOR OTHER LENGHTS (maybe needed only for grouping??)
        )(
        input clk_i,
        input rsn_i,
        input [SLIDE_WIDTH-1:0] offset_i,
        input enable_i, // USED TO SAVE ALL THE SETTINGS AND AVOIDING THE STRANGE THINGS WITH OP INFLIGHT AND SO ON
        input start_i,
        input aux_slide_i,
        input widening_i, // USED FOR THE SPECIAL CASE OF SLIDE FOR WIDENING, IN ORDER TO CORRECTLY CREATE THE SOURCES
        input rgather_op_i,
        input [2:0] rgather_lane_dst_i,
        input [2:0] rgather_bank_dst_i,
        input [2:0] rgather_sub_bank_dst_i,
        input type_i, // USED TO CHECK WHETHER IS A SLIDE UP OR DOWN (IF 0 SLIDE UP, IF 1 SLIDE DOWN)
        input send_i,
        input finish_i,
        input free_i,
        input accum_en_i,
        input [DATA_OUT_WIDTH-1:0] data_o,
        input fetch_o,
        input req_o,
        input empty_o,
        input ready_o,
        input kill_i,
        input [VLEN_WIDTH-1:0] vlength_i, // TOTAL VECTOR LENGTH
        input [VLEN_WIDTH-1:0] vl_lane_i, // ELEMENTS PER LANE
        input [1:0] sew_i      // STANDARD ELEMENT WIDTH
        );



```

```
typedef struct {
        int offset_fifo;
        bit free;
        bit zeroing;
} send_fifo_t;

send_fifo_t send_fifo[FIFO_SIZE-1:0];
bit is_index_ring;
int current_bank_int;
int current_sub_bank_int;
int sub_banks;
bit op_inflight;
bit op_started;
bit slide_ud;
int dest_bank_int;
int dest_sub_bank_int;
int dest_line_int;
int dest_lane_int;
int el_id_int;
int signed offset_int;
int unsigned el_id_dest;
int sew_int;
int i;
int rgather_lane_dst_int;
int rgather_bank_dst_int;
int rgather_sub_bank_dst_int;
int data_zeroing;

assign is_index_ring = IS_INDEX_RING;

initial begin
        current_sub_bank_int = 0;
        current_bank_int = 0;
        op_inflight = 0;
        op_started = 0;
        dest_bank_int = 0;
        dest_sub_bank_int = 0;
        dest_line_int = 0;
        dest_lane_int = 0;
        el_id_int = 0;
        offset_int = 0;
        el_id_dest = 0;
        sew_int = 0;
        data_zeroing = 0;
end

//FIFO ON SEND_I
always @(negedge clk_i or negedge rsn_i) begin
        if (~rsn_i) begin
                foreach (send_fifo[i]) begin
                        send_fifo[i].offset_fifo = '0;
                        send_fifo[i].free = 1;
                        send_fifo[i].zeroing = 0;
                end
        end
        else if (kill_i) begin
                foreach (send_fifo[i]) begin
                        send_fifo[i].offset_fifo = '0;
                        send_fifo[i].free = 1;
                        send_fifo[i].zeroing = 0;
                end
        end
        //fills the fifo searching for the first free space in the fifo
```

```
    else if (send_i) begin
            for (i = 0; i < FIFO_SIZE; i++) begin
                if (send_fifo[i].free) begin
                    send_fifo[i].offset_fifo = ( offset_i > ( MAX_VLEN * sub_banks) ) ? LANE_ID*sub_banks : offset_i;// offset_i[11:0] - vlength_i : offset_i[11:0];
                    send_fifo[i].free = 0;
                    send_fifo[i].zeroing = ( offset_i > ( MAX_VLEN * sub_banks) ) ? 1 : 0;
                    break;
                end
            end


    end

end




always @(posedge clk_i or negedge rsn_i) begin
    if (~rsn_i) begin
        current_sub_bank_int = 0;
        current_bank_int = 0;
        op_inflight = 0;
        op_started = 0;
        dest_bank_int = 0;
        dest_sub_bank_int = 0;
        dest_line_int = 0;
        dest_lane_int = 0;
    end
    else if (kill_i) begin
        current_sub_bank_int = 0;
        current_bank_int = 0;
        op_inflight = 0;
        op_started = 0;
        dest_bank_int = 0;
        dest_sub_bank_int = 0;
        dest_line_int = 0;
        dest_lane_int = 0;
    end
    else if (accum_en_i || rgather_op_i || widening_i) begin
        op_inflight = 1;
        op_started = 1;
        if (widening_i & enable_i) begin
            current_sub_bank_int = 0;
            current_bank_int = 0;
            dest_sub_bank_int = 0;
            dest_line_int = 0;
            dest_lane_int = (LANE_ID*2) % N_LANES;
            dest_bank_int = (LANE_ID*2) / N_LANES;
        end
        if (accum_en_i & start_i) begin
            current_sub_bank_int = 0;
            current_bank_int = 0;
            dest_sub_bank_int = 0;
            dest_bank_int = 0;
            dest_line_int = 0;
            dest_lane_int = (LANE_ID + 1) % N_LANES;
        end
        if(rgather_op_i && enable_i) begin
```

```
                        current_sub_bank_int = 0;
                        current_bank_int = 0;
                end
                /*if(start_i) begin
                        current_sub_bank_int = 0;
                        current_bank_int = 0;
                end */
        end

        else if (start_i) begin
                op_inflight = 1;
                op_started = 1;
                dest_bank_int = (el_id_dest/(sub_banks*N_LANES))%N_BANKS;
                dest_sub_bank_int = el_id_dest % sub_banks;
                dest_line_int = el_id_dest/(sub_banks*N_LANES*N_BANKS);
                dest_lane_int = (el_id_dest/sub_banks) % N_LANES;
                if(enable_i) begin
                        current_sub_bank_int = 0;
                        current_bank_int = 0;
                end
        end


        if ((op_started && finish_i) || !op_started) begin
                op_inflight = 0;
                if(!accum_en_i && !rgather_op_i && !widening_i) begin
                        current_sub_bank_int = 0;
                        current_bank_int = 0;
                        dest_bank_int = 0;
                        dest_sub_bank_int = 0;
                        dest_line_int = 0;
                        dest_lane_int = 0;
                end
        end
        if (op_inflight) begin
                if(req_o) begin
                        current_sub_bank_int = current_sub_bank_int + 1;
                        if(current_sub_bank_int == (sub_banks)) begin
                                current_bank_int = current_bank_int + 1;
                                current_bank_int = current_bank_int % N_BANKS;
                                current_sub_bank_int = 0;
                        end

                        if (widening_i) begin
                                widening_op();
                        end

                        else if (rgather_op_i) begin

                        end

                        else if (accum_en_i) begin
                                reduction_op();
                        end
                        else begin
                                slide_op();
                        end





                end
        end
end



always @(negedge clk_i) begin
        if(enable_i) begin
                sub_banks = 2**(3-sew_i);
                slide_ud = !type_i;
                el_id_int = LANE_ID*sub_banks; // set to the first valid element of the lane
                offset_int = type_i ? (-offset_i) : offset_i;
                el_id_dest = el_id_int + offset_int;
                if (el_id_dest >= '1) el_id_dest = vlength_i - (2**32 - el_id_dest);
                sew_int = sew_i;
        end

        if(rgather_op_i && op_inflight) begin
                if (!is_index_ring && start_i) rgather_op_data();
                if (is_index_ring && req_o) rgather_op_index();

        end
end

/////////////////////////////////////////////////////
///////////////////properties///////////////////////
/////////////////////////////////////////////////////


//1.0.1 vp
property p_rsn_i;
        @(posedge clk_i)
        !rsn_i |-> data_o == '0 && !ready_o && !fetch_o && !req_o && empty_o ##1 data_o == '0 && !ready_o && !fetch_o && !req_o && empty_o;
endproperty : p_rsn_i

//1.0.2 vp
property p_start_i_known;
        @(posedge clk_i)
        start_i |->     !$isunknown(offset_i) && !$isunknown(enable_i) && !$isunknown(start_i) && !$isunknown(aux_slide_i) && !$isunknown(widening_i) &&
                        !$isunknown(rgather_op_i) && !$isunknown(type_i) && !$isunknown(send_i) && !$isunknown(finish_i) && !$isunknown(free_i) &&
                        !$isunknown(accum_en_i) && !$isunknown(vlength_i) && !$isunknown(vl_lane_i) && !$isunknown(sew_i);
endproperty : p_start_i_known

//1.0.3 vp
property p_rgather_op_i_known;
        @(posedge clk_i)
        rgather_op_i |-> !$isunknown(rgather_lane_dst_i) && !$isunknown(rgather_bank_dst_i) && !$isunknown(rgather_sub_bank_dst_i);
endproperty : p_rgather_op_i_known

//1.0.4 vp
property p_req_o_known;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> !$isunknown(data_o);
endproperty : p_req_o_known

```

```
//1.0.5 vp
property p_req_o;
        @(posedge clk_i)
        $rose(req_o) |-> ##1 $fell(req_o);
endproperty : p_req_o

//1.0.6 vp
property p_exclusive_input;
        @(posedge clk_i)
        ($rose(rgather_op_i) or $rose(widening_i) or $rose(accum_en_i)) |-> (
            rgather_op_i ^ widening_i ^ accum_en_i) && !(rgather_op_i &&
            accum_en_i && widening_i); // (!rgather_op_i || widening_i ||
            accum_en_i ) && (rgather_op_i || !widening_i || accum_en_i) && (
            rgather_op_i || widening_i || !accum_en_i);
endproperty : p_exclusive_input

//1.0.7 vp
property p_start_i;
        @(posedge clk_i)
        $rose(start_i) |-> ##1 $fell(start_i);
endproperty : p_start_i

//1.0.8 vp
property p_send_i;
        @(posedge clk_i)
        send_i |-> ##1 !send_i;
endproperty : p_send_i

//1.0.9 vp
property p_kill_i;
        @(posedge clk_i)
        kill_i |-> ##1 !data_o[22] && !req_o;
endproperty : p_kill_i

//1.0.10 vp
property p_kill_i_data_i;
        @(posedge clk_i)
        kill_i |-> !enable_i && !start_i;
endproperty : p_kill_i_data_i

//1.0.11 vp
property p_rsn_i_data_i;
        @(posedge clk_i)
        !rsn_i |-> offset_i == '0 && vlength_i == '0 && vl_lane_i == '0 && !
            widening_i && !send_i && !rgather_op_i && !aux_slide_i && !accum_en_i
            && !start_i;
endproperty : p_rsn_i_data_i

//1.0.12 vp
property p_data_o_bank_in;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[17:15] == current_bank_int;
endproperty : p_data_o_bank_in

//1.0.13 vp
property p_data_o_sub_bank_in;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[14:12] ==
            current_sub_bank_int;
endproperty : p_data_o_sub_bank_in

//1.0.14 vp
property p_data_o_lane_in;
```

```
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[20:18] == LANE_ID;
endproperty : p_data_o_lane_in

//1.0.15 vp
property p_data_o_sub_bank_out;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[2:0] == dest_sub_bank_int;
endproperty : p_data_o_sub_bank_out

//1.0.16 vp
property p_data_o_lane_out;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[11:9] == dest_lane_int;
endproperty : p_data_o_lane_out

//1.0.17 vp
property p_data_o_bank_out;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[5:3] == dest_bank_int;
endproperty : p_data_o_bank_out

//1.0.18 vp
property p_data_o_line_out;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[8:6] == dest_line_int;
endproperty : p_data_o_line_out

//1.0.19 vp
property p_data_o_zeroing;
        @(posedge clk_i)
        req_o && data_o[DATA_OUT_WIDTH-1] |-> data_o[21] == data_zeroing;
endproperty : p_data_o_zeroing

//1.1.1 vp
property p_data_after_send;
        @(posedge clk_i)
        (rgather_op_i or widening_i or accum_en_i) and send_i |-> ##1 !$isunknown(
            data_o);
endproperty : p_data_after_send

//MOD (ADDED READY_O)
//1.1.2 vp
property p_req_after_send;
        @(posedge clk_i)
        send_i && ready_o && !kill_i |-> ##1 req_o;
endproperty : p_req_after_send

//1.1.3 vp
property p_fetch_on_req;
        @(posedge clk_i)
        $rose(req_o) |-> !fetch_o;
endproperty : p_fetch_on_req

//1.1.4 vp
property p_ready_on_req;
        @(posedge clk_i)
        $rose(req_o) && !accum_en_i |-> $fell(ready_o);
endproperty : p_ready_on_req

//MOD (ADDED READY_O)
//1.1.5 vp
property p_fetch_on_send;
```

```
        @(posedge clk_i)
        send_i && !accum_en_i && ready_o |-> fetch_o;
endproperty : p_fetch_on_send

//MOD (ADDED READY_O)
//1.1.6 vp
property p_send_on_fetch;
        @(posedge clk_i)
        !fetch_o && !accum_en_i && ready_o |-> !send_i;
endproperty : p_send_on_fetch

//1.1.7 vp
property p_free_on_req;
        @(posedge clk_i)
        $rose(req_o) |-> $fell(free_i);
endproperty : p_free_on_req

//1.1.8 vp
property p_offset_known;
        @(posedge clk_i)
        $rose(enable_i) |-> !$isunknown(offset_i);
endproperty : p_offset_known

//1.1.9 vp
property p_data_gather;
        @(posedge clk_i)
        rgather_op_i && $rose(send_i) |-> ( !$isunknown(rgather_lane_dst_i) && !
            $isunknown(rgather_bank_dst_i) && !$isunknown(rgather_sub_bank_dst_i)
            );
endproperty : p_data_gather

//1.1.10 vp
property p_data_accum_en;
        @(posedge clk_i)
        $rose(enable_i) |-> accum_en_i == 1'b0;
endproperty : p_data_accum_en

//1.1.11 vp
property p_data_send_free;
        @(posedge clk_i)
        $rose(send_i) |-> free_i == 1'b1;
endproperty : p_data_send_free

//1.1.12 vp
property p_data_slide_i;
        @(posedge clk_i)
        !rgather_op_i && !accum_en_i && !widening_i && $rose(enable_i) |-> ( !
            $isunknown(offset_i) && !$isunknown(type_i) && !$isunknown(aux_slide_i
            ) );
endproperty : p_data_slide_i

//1.2.1 vp
property p_index_fetch_send;
        @(posedge clk_i)
        $rose(send_i) |-> $fell(fetch_o);
endproperty : p_index_fetch_send

//1.2.2 vp
property p_index_gather;
        @(posedge clk_i)
        rgather_op_i && $rose(send_i) |-> ( rgather_lane_dst_i == '0 &&
            rgather_bank_dst_i == '0 && rgather_sub_bank_dst_i == '0) ;
endproperty : p_index_gather
```

```

//1.2.3 vp
property p_index_send_req;
        int num_send;
        @(posedge clk_i)
        ($rose(rgather_op_i), num_send = 0) |->  first_match( (1, num_send =
            num_send + send_i - req_o) [*0:$] ##0 $fell(rgather_op_i) ) |->
            num_send == 0;
endproperty : p_index_send_req

///////////////////////////////////////////////////////
///////////////////assert & assume////////////////////
///////////////////////////////////////////////////////


//1.0.1 vp
a_rsn_i :                assert property( p_rsn_i ) else $error("bad output for the
     rsn state");

//1.0.2 vp
a_start_i_known :        assume property( disable iff(!rsn_i || kill_i)
    p_start_i_known ) else $error("unknown inputs on start_i asserted");

//1.0.3 vp
a_rgather_op_i_known :   assume property( disable iff(!rsn_i || kill_i || !
    is_index_ring) p_rgather_op_i_known ) else $error("unknown rgather inputs on
    rgather_op_i asserted");

//1.0.4 vp
a_req_o_known :          assert property( disable iff(!rsn_i || kill_i)
    p_req_o_known ) else $error("unknown data_o on req_o asserted with data_o
    valid");

//1.0.5 vp
a_req_o :                assert property( disable iff(!rsn_i || kill_i) p_req_o )
    else $error("req_o lasted more than 1 clock cycle");

//1.0.6 vp
a_exclusive_input :      assume property( disable iff(!rsn_i || kill_i)
    p_exclusive_input ) else $error("more operation at once");

//1.0.7 vp
a_start_i :              assume property( disable iff(!rsn_i || kill_i) p_start_i )
     else $error("start_i asserted for more than one cycle");

//1.0.8 vp
a_send_i :               assume property( disable iff(!rsn_i || kill_i) p_send_i )
    else $error("send_i asserted for more than one cycle");

//1.0.9 vp
a_kill_i :               assert property( disable iff(!rsn_i) p_kill_i ) else
    $error("wrong output when kill_i");

//1.0.10 vp
a_kill_i_data_i :        assume property( disable iff(!rsn_i) p_kill_i_data_i )
    else $error("input not zero when kill_i");

//1.0.11 vp
a_rsn_i_data_i :         assume property( disable iff(!rsn_i) p_rsn_i_data_i ) else
     $error("input not zero when rsn_i");

//1.0.12 vp
```

```
a_data_o_bank_in :      assert property( disable iff(!rsn_i || kill_i || ( rgather_op_i && !is_index_ring) || !op_inflight) p_data_o_bank_in ) else $error("wrong current_bank on data_o");

//1.1.13 vp
a_data_o_sub_bank_in :  assert property( disable iff(!rsn_i || kill_i || ( rgather_op_i && !is_index_ring) || !op_inflight) p_data_o_sub_bank_in ) else $error("wrong current_sub_bank on data_o");

//1.0.14 vp
a_data_o_lane_in :      assert property( disable iff(!rsn_i || kill_i || ( rgather_op_i && !is_index_ring) || !op_inflight) p_data_o_lane_in ) else $error("wrong current_lane on data_o");

//1.0.15 vp
a_data_o_sub_bank_out : assert property( disable iff(!rsn_i || kill_i || !op_inflight) p_data_o_sub_bank_out ) else $error("wrong dest_sub_bank on data_o");

//1.0.16 vp
a_data_o_lane_out :     assert property( disable iff(!rsn_i || kill_i || !op_inflight) p_data_o_lane_out ) else $error("wrong dest_lane on data_o");

//1.0.17 vp
a_data_o_bank_out :     assert property( disable iff(!rsn_i || kill_i || !op_inflight) p_data_o_bank_out ) else $error("wrong dest_bank on data_o");

//1.0.18 vp
a_data_o_line_out :     assert property( disable iff(!rsn_i || kill_i || ( rgather_op_i && !is_index_ring) || !op_inflight) p_data_o_line_out ) else $error("wrong dest_line on data_o");

//1.0.19 vp
a_data_o_zeroing :      assert property( disable iff(!rsn_i || kill_i ) p_data_o_zeroing ) else $error("wrong zeroing bit on data_o");

//1.1.1 vp
a_data_after_send :     assert property( disable iff(!rsn_i || kill_i || is_index_ring) p_data_after_send ) else $error("no data_o after a send_i on a op request");

//1.1.2 vp
a_req_after_send :      assert property( disable iff(!rsn_i || kill_i || is_index_ring) p_req_after_send ) else $error("no req_o after a send_i on a op request");

//1.1.3 vp
a_fetch_on_req :        assert property( disable iff(!rsn_i || kill_i || is_index_ring) p_fetch_on_req ) else $error("fetch_o is not falling when req_o is rising");

//1.1.4 vp
a_ready_on_req :        assert property( disable iff(!rsn_i || kill_i || is_index_ring) p_ready_on_req ) else $error("ready_o is not falling when req_o is rising");

//1.1.5 vp
a_fetch_on_send :       assert property( disable iff(!rsn_i || kill_i || is_index_ring) p_fetch_on_send ) else $error("when send_i is '1' fetch_o is not '1'");

//1.1.6 vp
```

```
a_send_on_fetch :        assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_send_on_fetch ) else $error("when fetch_o is '0' send_i is
    not '0'");

//1.1.7 vp
a_free_on_req :          assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_free_on_req ) else $error("when req rose free didn't fall");

//1.1.8 vp
a_offset_known :         assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_offset_known ) else $error("offset was not known when enable
    was = '1'");

//1.1.9 vp
a_data_gather :          assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_data_gather ) else $error("unknown input to data ring logic
    for rgather");

//1.1.10 vp
a_data_accum_en :        assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_data_accum_en ) else $error("accum_en and enable_i ='1'
    contemporary");

//1.1.11 vp
a_data_send_free :       assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_data_send_free ) else $error("send_i when the data ring logic
     was not free");

//1.1.12 vp
a_data_slide_i :         assume property( disable iff(!rsn_i || kill_i ||
    is_index_ring) p_data_slide_i ) else $error("unknown instruction for slide");

//1.2.1 vp
a_index_fetch_send :     assert property( disable iff(!rsn_i || kill_i || !
    is_index_ring) p_index_fetch_send ) else $error("fetch_o didn't fall when
    send_i to index");

//1.2.2 vp
a_index_gather :         assume property( disable iff(!rsn_i || kill_i || !
    is_index_ring) p_index_gather ) else $error("rgather_* != '0 when sending
    gather request to index");

//1.2.3 vp
a_index_send_req :       assert property( disable iff(!rsn_i || kill_i || !
    is_index_ring) p_index_send_req ) else $error("data corruption in rgather
    indx_ring_logic");

//////////////////////////////////////////////////////
///////////////////////tasks/////////////////////////
//////////////////////////////////////////////////////



task widening_op ();
        dest_sub_bank_int = dest_sub_bank_int + 1;
        if (dest_sub_bank_int == sub_banks/2) begin
                dest_lane_int = (dest_lane_int + 1) % N_LANES;
                if (dest_lane_int == ((LANE_ID*2 + 2) % N_LANES)) begin
                        dest_bank_int = (dest_bank_int + 2);
                        dest_lane_int = (LANE_ID*2) % N_LANES;
                        if (dest_bank_int >= N_BANKS) begin
                                dest_bank_int = dest_bank_int % N_BANKS;
                                dest_line_int = (dest_line_int +1 ) % (MAX_VLEN*
                                    sub_banks/(N_BANKS*N_LANES));
                        end

                end

                dest_sub_bank_int = 0;
        end



endtask : widening_op



task rgather_op_index();
        int j;
        el_id_dest = send_fifo[0].offset_fifo;
        data_zeroing = send_fifo[0].zeroing;

        //update fifo
        for (j = 0; j < FIFO_SIZE - 1; j++) begin
                send_fifo[j].offset_fifo = send_fifo[j + 1].offset_fifo;
                send_fifo[j].free = send_fifo[j + 1].free;
                send_fifo[j].zeroing = send_fifo[j + 1].zeroing;
        end

        send_fifo[FIFO_SIZE-1].offset_fifo = 0;
        send_fifo[FIFO_SIZE-1].free = 1;
        send_fifo[FIFO_SIZE-1].zeroing = 0;

        dest_bank_int = (el_id_dest/(sub_banks*N_LANES))%N_BANKS;
        dest_sub_bank_int = el_id_dest % sub_banks;
        dest_line_int = el_id_dest/(sub_banks*N_LANES*N_BANKS);
        dest_lane_int = (el_id_dest/sub_banks) % N_LANES;

endtask : rgather_op_index


task rgather_op_data();
        dest_bank_int = rgather_bank_dst_i;
        dest_sub_bank_int = rgather_sub_bank_dst_i;
        dest_lane_int = rgather_lane_dst_i;
        dest_line_int = 0;
endtask : rgather_op_data

task reduction_op();
        current_sub_bank_int = 0;
        current_bank_int = 0;
        dest_sub_bank_int = 0;
        dest_bank_int = 0;
        dest_line_int = 0;
        dest_lane_int = (LANE_ID + 1)%N_LANES;
endtask : reduction_op


task slide_op();
        el_id_int = el_id_int + 1;

        if ( (el_id_int%(2**(3-sew_int))) == 0 ) el_id_int = el_id_int-sub_banks +
             (sub_banks*N_LANES);

        el_id_dest = el_id_int + offset_int;

        if (el_id_dest >= vlength_i) el_id_dest = vlength_i - (2**32 - el_id_dest)
            ;

        dest_bank_int = (el_id_dest/(sub_banks*N_LANES))%N_BANKS;
        dest_sub_bank_int = el_id_dest % sub_banks;
        dest_line_int = el_id_dest/(sub_banks*N_LANES*N_BANKS);
        dest_lane_int = (el_id_dest/sub_banks) % N_LANES;
endtask : slide_op


endmodule: ring_logic_checker // ring_logic
```

A.4 ring_node

```
import EPI_pkg::*;

bind ring_node ring_node_checker #(
        .LANES(LANES),
        .WIDTH(WIDTH),
        .LANE_ID(LANE_ID)) bind_ring_node_checker (.*);

module ring_node_checker
    #(
    parameter LANES = 8,
    parameter WIDTH = 84,
    parameter LANE_ID = 3'b000
    )(

    input                   clk_i,
    input                   rsn_i,
    input                   stall_i,
    input                   kill_i,
    input                   start_i,
    input [WIDTH-1:0]       data_lane_i,
    input [WIDTH-1:0]       data_ring_i,
    input                   data_valid_o,
    input [WIDTH-1:0]       data_ring_o,
    input                   we_o,
    input                   self_write_o
    );

wire[2:0] curr_lane;
assign curr_lane = LANE_ID;



/////////////////////////////////////////////////////
/////////////////////properties/////////////////////
/////////////////////////////////////////////////////




//3.01 vp
property p_rsn_i_outputs;
        @(posedge clk_i)
        !rsn_i |-> !data_valid_o && data_ring_o == '0 ##1 !data_valid_o &&
            data_ring_o == '0;
endproperty : p_rsn_i_outputs
```

```

//3.02 vp
property p_self_write_o;
        @(posedge clk_i)
        $rose(start_i) && data_lane_i[WIDTH-1] && ( curr_lane == data_lane_i[WIDTH
            -12:WIDTH-14] ) && !stall_i |->
        (self_write_o && we_o) ;
endproperty : p_self_write_o

//3.03 vp
property p_we_o;
        @(posedge clk_i)
        data_ring_i[WIDTH-1] && ( curr_lane == data_ring_i[WIDTH-12:WIDTH-14] ) &&
             !stall_i |-> we_o ;
endproperty : p_we_o

//3.04 vp
property p_data_lane_propagation();
        logic [WIDTH-1:0] local_var;
        @(posedge clk_i)
        (start_i && !stall_i,local_var=data_lane_i) ##0 data_lane_i[WIDTH-1] && (
            curr_lane != data_lane_i[WIDTH-12:WIDTH-14] ) |-> !we_o && !
            self_write_o ##1  (data_ring_o == local_var) && data_valid_o;
endproperty : p_data_lane_propagation

//3.05 vp
property p_data_ring_propagation;
        logic [WIDTH-1:0] local_var;
        @(posedge clk_i)
        (data_ring_i[WIDTH-1] && !stall_i,local_var=data_ring_i) ##0 ( curr_lane
            != data_ring_i[WIDTH-12:WIDTH-14] )
        |-> !we_o && !self_write_o ##1 data_valid_o && data_ring_o == local_var;
endproperty : p_data_ring_propagation

//3.06 vp
property p_invalid_data;
        @(posedge clk_i)
        !data_ring_i[WIDTH-1] && !data_lane_i[WIDTH-1] && !stall_i |-> !we_o && !
            self_write_o  ##1 !data_valid_o ;
endproperty : p_invalid_data

//3.07 vp
property p_stall_i; //data ring
        logic[WIDTH-1:0] local_data_ring_o;
        logic local_data_valid_o;

        @(posedge clk_i)
        ( stall_i, local_data_ring_o = data_ring_o, local_data_valid_o =
            data_valid_o ) |->
        !we_o && !self_write_o |->  first_match(1 [*1:$] ##0 !stall_i)  |->
            data_valid_o == local_data_valid_o && data_ring_o == local_data_ring_o
            ;
endproperty : p_stall_i

//3.08 vp
property p_kill_i;
        @(posedge clk_i)
        kill_i |-> ##1 !data_valid_o && data_ring_o == '0;
endproperty : p_kill_i

//3.09 vp
property p_rsn_data_i;
        @(posedge clk_i)
        !rsn_i |-> !stall_i && !kill_i && !start_i && !data_lane_i[WIDTH-1] && !
            data_ring_i[WIDTH-1];
endproperty : p_rsn_data_i

//3.10 vp
property p_kill_data_i;
        @(posedge clk_i)
        kill_i |-> !stall_i && !start_i && !data_lane_i[WIDTH-1] && !data_ring_i[
            WIDTH-1];
endproperty : p_kill_data_i

//3.11 vp
property p_start_data_lane_i;
        @(posedge clk_i)
        start_i |-> !data_ring_i[WIDTH-1];
endproperty : p_start_data_lane_i


//3.12 vp
property p_rsn_i_beh;
        @(posedge clk_i)
        $fell(rsn_i) |-> ##1 !rsn_i;
endproperty : p_rsn_i_beh

/////////////////////////////////////////////////////////
///////////////////assert & assume////////////////////
/////////////////////////////////////////////////////////




//3.01 vp
a_rsn_i_outputs :                assert property( p_rsn_i_outputs ) else $error("
    bad handle for the rsn signal");

//3.02 vp
a_self_write_o :                 assert property( disable iff(!rsn_i || kill_i)
    p_self_write_o ) else $error("didn't self_write_o when supposed to");

//3.03 vp
a_we_o :                         assert property( disable iff(!rsn_i || kill_i)
    p_we_o ) else $error("no write enable when supposed");

//3.04 vp
a_data_lane_propagation :        assert property( disable iff(!rsn_i || kill_i) (
    p_data_lane_propagation or p_stall_i) ) else $error("after a start and a valid
     data_lane_i, with the data for another lane, data_lane_i was not propagated
    to the output the following clock cicle");

//3.05 vp
a_data_ring_propagation :        assert property( disable iff(!rsn_i || kill_i) (
    p_data_ring_propagation or p_stall_i) ) else $error("wrong data_ring_o when
    valid data_ring_i the previous clock");

//3.06 vp
a_invalid_data :                 assert property( disable iff(!rsn_i || kill_i) (
    p_invalid_data or p_stall_i) ) else $error("bad handle for invalid data");

//3.07 vp
a_stall_i :                      assert property( disable iff(!rsn_i || kill_i)
    p_stall_i ) else $error("bad handle for the stall signal");

//3.08 vp
```

```
a_kill_i :                        assert property( disable iff(!rsn_i) p_kill_i )
    else $error("bad handle for the kill signal");

//3.09 vp
a_rsn_data_i :                    assume property( p_rsn_data_i ) else $error("no
    reset of inputs data when reset");

//3.10 vp
a_kill_data_i :                   assume property( disable iff(!rsn_i) p_kill_data_i
     ) else $error("no reset of inputs data when kill");

//3.11 vp
a_start_data_lane_i :             assume property( disable iff(!rsn_i || kill_i)
    p_start_data_lane_i ) else $error("no valid data lane on start");

//3.12 vp
a_rsn_i_beh :                     assume property( p_rsn_i_beh ) else $error("rsn_i
    lasted less than 1 clock cycle");

endmodule: ring_node_checker // ring_node
```

A.5 ring_packer

```
import EPI_pkg::*;

bind ring_data_packer ring_data_packer_checker #(
        .RING_LOGIC_DATA(RING_LOGIC_DATA),
        .DATA_WIDTH(DATA_WIDTH),
        .DATA_IN_WIDTH(DATA_IN_WIDTH),
        .MAX_N_SUB_BANKS(MAX_N_SUB_BANKS),
        .DATA_CTRL_WIDTH(DATA_CTRL_WIDTH)) bind_ring_data_packer (.*);

module ring_data_packer_checker
    #(
    parameter RING_LOGIC_DATA = 20,
    parameter DATA_WIDTH    = 64,
    parameter DATA_IN_WIDTH = RING_LOGIC_DATA+DATA_WIDTH,
    parameter MAX_N_SUB_BANKS = MIN_SEW,
    parameter DATA_CTRL_WIDTH = DATA_WIDTH/MAX_N_SUB_BANKS
    )(
    input clk_i,
    input rsn_i,
    input kill_i,
    input [DATA_IN_WIDTH-1:0] data_i,
    input req_i,
    input free_i,
    input finish_i,
    input [DATA_IN_WIDTH-1:0] data_o,
    input req_o,
    input finish_o,
    input [1:0] sew_i,
    input widening_i,
    input [MAX_N_SUB_BANKS-1:0] valid_sub_banks_o
    );

//////////////////////////////////////////////////////
////////////////////sequences/////////////////////////
//////////////////////////////////////////////////////

//auxiliary seq to 6.0.8
sequence s_req_o_1;
```

```
        ##1 !req_o;
endsequence : s_req_o_1

//auxiliary seq to 6.0.8
sequence s_req_o_2;
        ##1 req_o ##1 !req_o;
endsequence : s_req_o_2


//////////////////////////////////////////////////////
/////////////////////properties//////////////////////
//////////////////////////////////////////////////////


//6.0.1 vp
property p_data_i_known;
        @(posedge clk_i)
        req_i |-> !$isunknown(data_i);
endproperty : p_data_i_known

//6.0.2 vp
property p_sew_i_known;
        @(posedge clk_i)
        req_i |-> !$isunknown(sew_i);
endproperty : p_sew_i_known

//6.0.3 vp
property p_sew_i_stable;
        int n_req_i;
        @(posedge clk_i)
        ($rose(req_i) && !kill_i, n_req_i = 2**(3-sew_i)) |=> first_match(( 1 ,
            n_req_i = n_req_i - req_i ) [*0:$] ##0 !$stable(sew_i) || $rose(req_o)
             ) |-> $stable(sew_i);
endproperty : p_sew_i_stable

//6.0.4 vp
property p_finish_o;
        @(posedge clk_i)
        finish_i |-> finish_i[=1] ##1 finish_o;
endproperty : p_finish_o

//6.0.5 vp
property p_req_i_max;
        int n_req_i;
        @(posedge clk_i)
        ($rose(req_i), n_req_i = 2**(2-sew_i)) |=> first_match(( 1 , n_req_i =
            n_req_i - req_i ) [*0:$] ##0 n_req_i <= 0 || $rose(req_o) ) |->
            n_req_i >= 0;
endproperty : p_req_i_max

//6.0.6 vp
property p_data_o_known;
        @(posedge clk_i)
        req_o |-> !$isunknown(data_o);
endproperty : p_data_o_known

//6.0.7 vp
property p_req_i;
        @(posedge clk_i)
        req_i |-> ##1 !req_i;
endproperty : p_req_i

//6.0.8 vp
```

```
property p_req_o;
        @(posedge clk_i)
        $rose(req_o) |-> s_req_o_1 or s_req_o_2;
endproperty : p_req_o

//6.0.9 vp
property p_finish_i;
        @(posedge clk_i)
        finish_i |-> ##1 !finish_i;
endproperty : p_finish_i

//6.0.10 vp
property p_finish_i_finish_o;
        @(posedge clk_i)
        $fell(finish_i) |-> first_match(1 [*0:$] ##0 ( finish_i || finish_o ) )
            ##0 ( finish_o && !finish_i )   ;
endproperty : p_finish_i_finish_o

//6.0.11 vp
property p_req_i_on_free;
        @(posedge clk_i)
        free_i |-> !req_i;
endproperty : p_req_i_on_free


///////////////////////////////////////////////////////
//////////////////assert & assume////////////////////
///////////////////////////////////////////////////////


//6.0.1 vp
a_data_i_known          :       assume property( disable iff(!rsn_i || kill_i)
    p_data_i_known ) else $error("unknown data_i on req_i");

//6.0.2 vp
a_sew_i_known           :       assume property( disable iff(!rsn_i || kill_i)
    p_sew_i_known ) else $error("unknown sew_i on req_i");

//6.0.3 vp
a_sew_i_stable          :       assume property( disable iff(!rsn_i || kill_i)
    p_sew_i_stable ) else $error("unstable sew_i during a req");

//6.0.4 vp
a_finish_o              :       assert property( disable iff(!rsn_i || kill_i)
    p_finish_o ) else $error("after a finish_i there wasn't a finish_o");

//6.0.5 vp
a_req_i_max             :       assume property( disable iff(!rsn_i || kill_i)
    p_req_i_max ) else $error("more req_i than expected");

//6.0.6 vp
a_data_o_known          :       assert property( disable iff(!rsn_i || kill_i)
    p_data_o_known ) else $error("unknown data_o on req_o");

//6.0.7 vp
a_req_i                 :       assume property( disable iff(!rsn_i || kill_i)
    p_req_i ) else $error("req_i lasted more than 1 clock cycle");

//6.0.8 vp
a_req_o                 :       assert property( disable iff(!rsn_i || kill_i)
    p_req_o ) else $error("req_o lasted more than 2 clock cycle");

//6.0.9 vp
```

```
a_finish_i              :       assume property( disable iff(!rsn_i || kill_i)
    p_finish_i ) else $error("finish_i lasted more than 1 clock cycle");

//6.0.10 vp
a_finish_i_finish_o     :       assume property( disable iff(!rsn_i || kill_i)
    p_finish_i_finish_o ) else $error("two finish_i without a finish_o between
    them");

//6.0.11 vp
a_req_i_on_free         :       assume property( disable iff(!rsn_i || kill_i)
    p_req_i_on_free ) else $error("req_i when free_i was = 1");

endmodule: ring_data_packer_checker
```

A.6 ring_unpacker

```
import EPI_pkg::*;

bind ring_data_unpacker ring_data_unpacker_checker #(
        .DATA_WIDTH(DATA_WIDTH),
        .IS_INDEX_RING(IS_INDEX_RING),
        .MAX_N_SUB_BANKS(MAX_N_SUB_BANKS),
        .DATA_CTRL_WIDTH(DATA_CTRL_WIDTH)) bind_ring_data_unpacker (.*);

module ring_data_unpacker_checker
        #(
    parameter DATA_WIDTH    = 64,
    parameter IS_INDEX_RING = 0, // this is used to distinguish between the data
        and index rings (may be updated in future versions)
    parameter MAX_N_SUB_BANKS = 8,
    parameter DATA_CTRL_WIDTH = DATA_WIDTH/MAX_N_SUB_BANKS
    )(
    input clk_i,
    input rsn_i,
    input kill_i,
    input new_instr_i,
    input [ELEMENTS_WIDTH_LANE-1:0] elements_i,
    input [DATA_WIDTH-1:0] data_i,
    input req_i,
    input grant_o,
    input [DATA_WIDTH-1:0] data_o,
    input req_o,
    input enable_o,
    input enable_i,
    input [1:0] sew_i,
    input ready_o,
    input finish_i
    );



//////////////////////////////////////////////////////
/////////////////////properties//////////////////////
//////////////////////////////////////////////////////


//5.0.1 vp
property p_data_i_known;
        @(posedge clk_i)
        req_i && enable_o |-> !$isunknown(data_i);
endproperty : p_data_i_known
```

```

//5.0.2 vp
property p_sew_i_known;
        @(posedge clk_i)
        req_i && enable_o |-> !$isunknown(sew_i);
endproperty : p_sew_i_known

//5.0.3 vp
property p_data_i_stable;
        @(posedge clk_i)
        $rose(grant_o) |=> first_match((1) [*0:$] ##0 $rose(enable_o) || !$stable(
            data_i)) |-> $stable(data_i);
endproperty : p_data_i_stable

//5.0.4 vp
property p_req_i_grant_o;
        @(posedge clk_i)
        req_i |-> ##[1:$] grant_o;
endproperty : p_req_i_grant_o

//5.0.5 vp
property p_req_o_num;
        int n_req_o;
        @(posedge clk_i)
        $rose(grant_o) ##0 (1, n_req_o = 2**(3-sew_i)) |=> first_match((1, n_req_o
             = n_req_o - $fell(req_o) ) [*0:$] ##0 $rose(grant_o) ) |-> n_req_o >=
             0;
endproperty : p_req_o_num

//5.0.6 vp
property p_enable_i_ready_o;
        @(posedge clk_i)
        !enable_i |-> !ready_o;
endproperty : p_enable_i_ready_o

//5.0.7 vp
property p_grant_o_req_i_num;
        int n_req_i;
        @(posedge clk_i)
        $rose(new_instr_i) ##0 (1, n_req_i = elements_i/(2**(3-sew_i)) + 1) |=>
            first_match( (1, n_req_i = n_req_i - (req_i && grant_o) ) [*0:$] ##0
            finish_i ) |-> n_req_i >= 0;
endproperty : p_grant_o_req_i_num

//5.0.8 vp
property p_req_o_ready_o;
        @(posedge clk_i)
        ready_o |-> ready_o[=1] ##1 req_o;
endproperty : p_req_o_ready_o

//5.0.9 vp
property p_req_o;
        @(posedge clk_i)
        req_o |-> ##1 !req_o;
endproperty : p_req_o

//5.0.10 vp
property p_ready_o;
        @(posedge clk_i)
        ready_o |-> ##1 !ready_o;
endproperty : p_ready_o

//5.0.11 vp
```

```
property p_data_o_known;
        @(posedge clk_i)
        req_o || ready_o |-> !$isunknown(data_o);
endproperty : p_data_o_known

//5.0.12 vp
property p_sew_i_stable;
        @(posedge clk_i)
        $rose(new_instr_i) |=> first_match((1) [*0:$] ##0 $rose(finish_i) || !
            $stable(sew_i)) |-> $stable(sew_i);
endproperty : p_sew_i_stable

//5.0.13 vp
property p_valid_data_o;
        @(posedge clk_i)
        req_o |-> (sew_i == 0 && data_o[DATA_WIDTH-1:8]=='0) ||  (sew_i == 1 &&
            data_o[DATA_WIDTH-1:16]=='0) || (sew_i == 2 && data_o[DATA_WIDTH-1:32]
            =='0) || sew_i==3;
endproperty : p_valid_data_o

//5.0.14 vp
property p_elements_i_known;
        @(posedge clk_i)
        new_instr_i |-> !$isunknown(elements_i);
endproperty : p_elements_i_known

//5.0.15 vp
property p_elements_i_req_o;
        int elements;
        @(posedge clk_i)
        new_instr_i ##0 (1, elements = elements_i) |->  first_match((1, elements =
             elements - $fell(req_o)) [*0:$] ##0 finish_i) |-> elements >= 0;
endproperty : p_elements_i_req_o

//5.0.16 vp
property p_finish_after_instr;
        int elements;
        @(posedge clk_i)
        new_instr_i |-> new_instr_i[=1] ##1 finish_i;
endproperty : p_finish_after_instr

//5.0.17 vp
property p_grant_o;
        int elements;
        @(posedge clk_i)
        req_i && enable_o |-> grant_o;
endproperty : p_grant_o

//5.0.18 vp
property p_elements_i_req_i;
        int elements;
        @(posedge clk_i)
        new_instr_i ##0 (1, elements = elements_i/(2**(3-sew_i)) + 1) |->
            first_match((1, elements = elements - (req_i && enable_o)) [*0:$] ##0
            finish_i) |-> elements >= 0;
endproperty : p_elements_i_req_i


/////////////////////////////////////////////////////////
///////////////////assert & assume/////////////////////
/////////////////////////////////////////////////////////


```

```
//5.0.1 vp
a_data_i_known          :       assume property( disable iff(!rsn_i || kill_i)
    p_data_i_known ) else $error("unknown data_i on enable_i and req_i");

//5.0.2 vp
a_sew_i_known           :       assume property( disable iff(!rsn_i || kill_i)
    p_sew_i_known ) else $error("unknown sew_i on enable_i and req_i");

//5.0.3 vp
a_data_i_stable         :       assume property( disable iff(!rsn_i || kill_i)
    p_data_i_known ) else $error("unstable data_i on enable_o = 0");

//5.0.4 vp
a_req_i_grant_o         :       assert property( disable iff(!rsn_i || kill_i)
    p_req_i_grant_o ) else $error("no grant_o for a req_i");

//5.0.5 vp
a_req_o_num             :       assert property( disable iff(!rsn_i || kill_i)
    p_req_o_num ) else $error("wrong number of valid req_o(s)");

//5.0.6 vp
a_enable_i_ready_o      :       assert property( disable iff(!rsn_i || kill_i)
    p_enable_i_ready_o ) else $error("ready_o = 1 with enable_i = 0");

//5.0.7 vp
a_grant_o_req_i_num     :       assert property( disable iff(!rsn_i || kill_i)
    p_grant_o_req_i_num ) else $error("wrong number of valid req_i(s)");

//5.0.8 vp
a_req_o_ready_o         :       assert property( disable iff(!rsn_i || kill_i)
    p_req_o_ready_o ) else $error("req_o was not = 1 after a ready_o");

//5.0.9 vp
a_req_o                 :       assert property( disable iff(!rsn_i || kill_i)
    p_req_o ) else $error("req_o didn't last 1 clock cycle");

//5.0.10 vp
a_ready_o               :       assert property( disable iff(!rsn_i || kill_i)
    p_ready_o ) else $error("ready_o didn't last 1 clock cycle");

//5.0.11 vp
a_data_o_known          :       assert property( disable iff(!rsn_i || kill_i)
    p_data_o_known ) else $error("unknown data_o on req_o or ready_o");

//5.0.12 vp
a_sew_i_stable          :       assume property( disable iff(!rsn_i || kill_i)
    p_sew_i_stable ) else $error("sew_i unstable during the same req_i");

//5.0.13 vp
a_valid_data_o          :       assert property( disable iff(!rsn_i || kill_i)
    p_valid_data_o ) else $error("misplacement in data_o");

//5.0.14 vp
a_elements_i_known      :       assume property( disable iff(!rsn_i || kill_i)
    p_elements_i_known ) else $error("unknonw elements_i when new_instr_i was
    asserted");

//5.0.15 vp
a_elements_i_req_o      :       assume property( disable iff(!rsn_i || kill_i)
    p_elements_i_req_o ) else $error("wrong number of req_o");

//5.0.16 vp
```

```
a_finish_after_instr    :       assume property( disable iff(!rsn_i || kill_i)
    p_finish_after_instr ) else $error("no finish_i after new_instr_i");

//5.0.17 vp
a_grant_o               :       assert property( disable iff(!rsn_i || kill_i)
    p_grant_o ) else $error("grant_o didn't raise when req_i and enable_o were
    asserted");

//5.0.18 vp
a_elements_i_req_i      :       assert property( disable iff(!rsn_i || kill_i)
    p_elements_i_req_i ) else $error("too many req_i for the given elements_i");


endmodule: ring_data_unpacker_checker
```

Bibliography

[1] Mentor verification academy. (accessed on august 2020). https://verificationacademy.com/.

[2] Risc-v cores and soc overview. (accessed on august 2020). https://github.com/riscv/riscv-cores-list.

[3] Risc-v history. (accessed on august 2020). https://riscv.org/risc-v-history/.

[4] Risc-v specifications. (accessed on august 2020). https://github.com/riscv/riscv-v-spec.

[5] Risc-v vector extension gcc binutils v0.7.x. (accessed on august 2020). https://github.com/riscv/riscv-binutils-gdb/tree/rvv-0.7.x.

[6] Riscv-dv presentation. (accessed on august 2020). https://content.riscv.org/wp-content/uploads/2019/12/12.10-16.10b-Open-Source-Verification-Platform-for-RISC-V-Processors.pdf.

[7] Riscv-dv repository. (accessed on august 2020). https://github.com/google/riscv-dv.

[8] Spike risc-v isa simulator. (accessed on august 2020). https://github.com/riscv/riscv-isa-sim.

[9] V for vector: software exploration of the vector extension of risc-v. (accessed on august 2020). https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/.

[10] K Asanovic A. Waterman. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. 2019.

[11] K Asanovic. International conference on supercomputing, workshop on risc-v and openpower. (session 4, accessed on august 2020). https://www.youtube.com/watch?v=6mXxNbKM3Qs&feature=youtu.be.

[12] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64, 2020.

[13] Clifford E Cummings. Systemverilog assertions-bindfiles & best known practices for simple sva usage. SNUG, 2016.

[14] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: Past, present and future, 1998.

[15] M. Kantrowitz and L. M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *33rd Design Automation Conference Proceedings, 1996*, pages 325–330, 1996.

[16] Ashok B. Mehta. *ASIC/SoC Functional Design Verification.* Springer, Cham, 2018.

[17] F. L. Morris and C. B. Jones. An early program proof by alan turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[18] David Rich. The missing link: the testbench to dut connection. *Fremont, CA: Design and Verification Technologies Mentor Graphics*, 9, 2013.

[19] Ray Salemi. *The UVM Primer: An Introduction to the Universal Verification Methodology.* Boston Light Press, 2013.

[20] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores. *IEEE Transactions on Computers*, 2020.

[21] Doug Smith. Using bind for class-based testbench reuse with mixedlanguage designs. 2009.

[22] Ryo Taketani and Yoshinori Takeuchi. Configurable processor hardware developing environment for risc-v with vector extension.

[23] Babu Turumella and Mukesh Sharma. Assertion-based verification of a 32 thread sparcâĎć cmt microprocessor. In *Proceedings of the 45th Annual Design Automation Conference*, DAC âĂŹ08, page 256âĂŞ261, New York, NY, USA, 2008. Association for Computing Machinery.