POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Thesis

Functional and Formal Verification on submodules of a Vector Processing Unit based on RISC-V V-extension

Supervisor:
Prof. Lavagno Luciano (PoliTo)

Cosupervisors:
Prof. Moll Echeto Francesc (UPC)
Dr. Palomar Oscar (BSC)

Candidate:
Guglielmi Vito Luca
ID: s265044

October 2020

Alla mia famiglia.

Abstract

This thesis was developed while working at Barcelona Supercomputing Center, a research center specialized in High Performance Computing and investigation in many fields, such as cloud computing, bioinformatics, material science and more.

Taking part to European Processor Initiative (EPI) project, the whole thesis aims to perform the verification process on a Vector Processing Unit (VPU). The implemented Vector Processing Unit is based on the RISC-V V-Extension, which is a set of specifications defining the Instructions Set Architecture (ISA) of a vector core. The V-Extension is currently on develop by the RISC-V foundation. This manuscript will refer to the versions 0.7.1.

The first chapter consist of an introduction of the needed concepts and of the context in which this thesis is been developed.

Then, in the second chapter, all the techniques used to verify functionally and formally this VPU are discussed.

All the results, such as found bugs or created material, are displayed in the third chapter. Moreover, analysing these results, the efficacy of the techniques used is evaluated. It is shown how formal and functional tools can be used to find bugs or to better define specification.

In the last chapter, it is possible to conclude that the techniques adopted produced the expected results showing significant improvements in the verification effort.

Contents

Chapter 1

Introduction

In this Chapter the RISC-V and V-Extension concepts relevant for this thesis are presented and discussed together with the verification notions. Furthermore, an overview on the context of this project is given, explaining the Vector Processing Unit (VPU) implemented and then showing the Universal Verification Methodologies (UVM) adopted.

Firstly, the RISC-V Instruction Set Architecture (ISA) is briefly illustrated, proceeding with an overview on all the major extensions provided by the RISC-V foundation. The Concepts section is then closed with some knowledge about the Verification process and its main tools.

Finally the theory is contextualized by the implemented VPU and on the used UVM structure. Those allow to better understand the contributions done in this thesis, contributions that are then explained into the next Chapter.

1.1 Concepts

1.1.1 RISC-V

RISC-V is an open, extensible and free ISA. It was initially only designed to support computer architecture research and education [1]. The project, began at the Berkeley University of California in 2010, published the first ISA User Manual in 2011.

The RISC-V ISA is implemented as a base integer ISA, but it is modular and

so supports many instruction encodings. This ISA is provided under open source licenses, and it is gaining a lot of popularity due to its open nature. Mainly there are two primary base integer variants, RV32I and RV64I, which provide 32-bit or 64-bit user-level address spaces respectively.

Extensions

RISC-V is designed to have a good customization which is the reason why it is provided with the possibility to be extended, but the base integer instructions cannot be redefined. There are two kinds of extensions: *standard* and *non-standard*.

- The *standard* ones need to be compatible with all the other standards and they should also aim to be generally useful.
- The *non-standard* ones can be highly specialized in some task and therefore they may conflict with other extensions.

For general development, as defined in the specification document [1], some of the standard extensions are predefined:

- "**I**" is the base integer extension and contains integer computational instructions, integer loads, integer stores, and control-flow instructions. It is mandatory for all RISC-V implementations.
- "**M**" is the standard integer multiplication and division extension, it allows to multiply and divide the values held in the integer registers.
- "**A**" is the standard atomic instruction extension. It is useful to have atomic instructions for inter-processor synchronization. In fact, with this extension is possible to read, modify, and write memory atomically.
- "**F**" is the standard single-precision floating-point extension. It adds floating-point registers, single-precision computational instructions, and single-precision loads and stores.
- "**D**" is the standard double-precision floating-point extension. Advantageous when the F extension is not enough, it expands the extension and adds double-precision computational instructions, loads, and stores.

- "**G**" is the denotation for an integer base plus these four standard extensions ("IMAFD").

- "**V**" is the standard Vector Extension. It is designed to add the possibility to perform vector operations, allowing performance and efficiency. The thesis will focus only on this extension.

The design philosophy of the RISC-V projects is based on modularity: the base ISA will not change over time, but new extensions will be available and new feature will be added. Which is a major feature because of the difficulty in finding general useful extension beyond the ones already existing. Therefore, it would not be convenient to constantly add new features to the base ISA and then have to keep track of that at a later time.

Base instructions

In Figure 1.1 it is possible to see how the base instruction are composed.

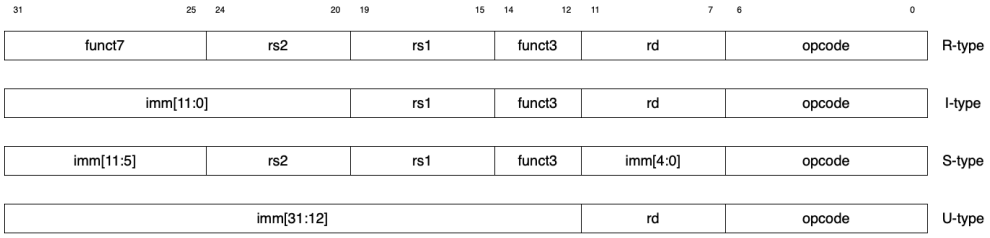

Figure 1.1. RISC-V base instruction formats [1]

It is important to notice that RISC-V is a load-store architecture, which means that only load and store operations can have access to the memory. It is a very convenient organization, because it reduces the average time-per-operation and guarantees a good functioning of the pipelined structure.

The instruction also supports signed byte and half word loads, which is very advantageous when working with signed byte and half word data types.

In Figure 1.2 it is possible to see how the load-store instructions are composed, and that the LOAD is an I-type op and the STORE is an S-type op.

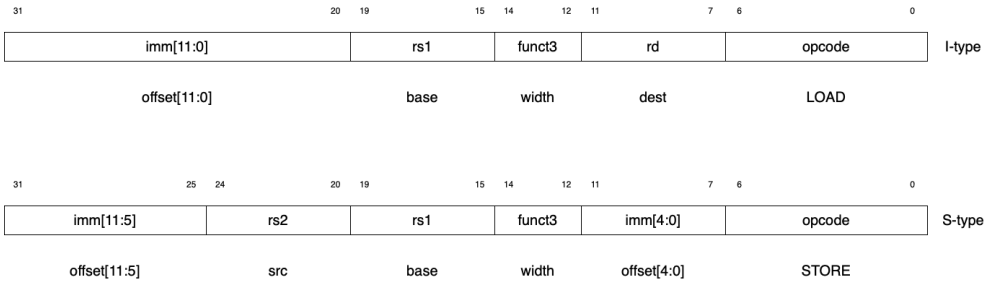

Figure 1.2. RISC-V load/store instruction formats [1]

1.1.2 Parallelism, Vectors and the V-Extension

During the last years, the parallel architecture is gaining inertia on the processors field. This is happening because the real world has parallel behaviour and so the hardware we use to compute simulations and calculus needs to be likewise [2].

Also, most of the paradigms that led to the decision of using a single core, are now changing quickly as the technology changes its needs. In fact, as the technology scales down to the nanometers, the power and the energy consumptions are becoming a problem. Also, the cost of a single transistor is significantly lower.

A solution to the efficiency problem are the VPU (Vector Processing Units) working with single instruction multiple data (SIMD), in this way it possible to maintain the binaries easy to program, but very powerful in term of data computation..

A famous example of a vector architecture is the Cray-1, presented in 1975. The Cray-1, a load/store architecture, was designed for Supercomputing and its major feature was to have a scalar mode along the vector one. This was advantageous because high performances are not always useful.

Time efficiency

As shown in Figure 1.3, with standard multithreading there will often be some empty thread along with the running ones, causing a loss in the efficiency.

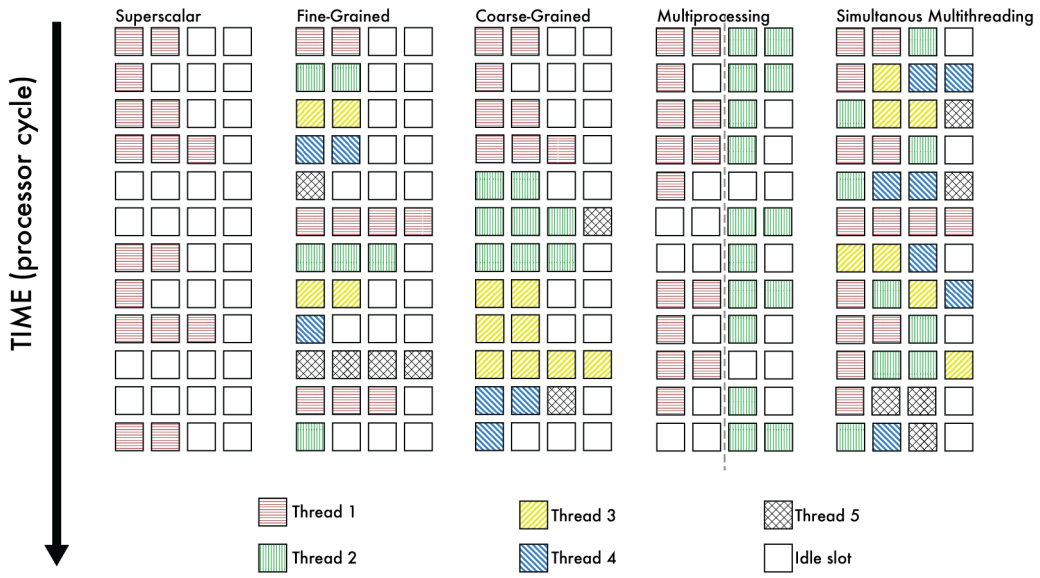

Figure 1.3. Multithreading Processor Clock Time usage [3]

Indeed the major advantage with the vector computation is the condensation of the processor usage. So that the usage will not be distributed as for standard multithreading processors, but it will have full usage for some cycles and none usage for others [3].

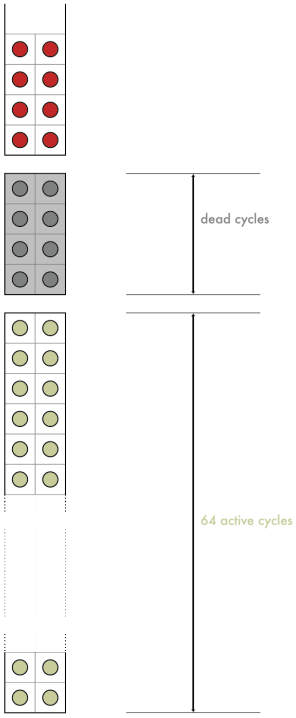

Figure 1.4. Vector Processor Processor Time usage [3]

It is shown in Figure 1.4 how the processor usage is condensed in some cycles and then empty for others. That happens because the vector operations need some clock cycles to start up. So based on the length of the pipeline, there will be some dead cycles. This means the latency increases, because each operation adds some dead cycles that need to be waited before a new operation can start.

But, this drawback can be used to increase the efficiency with modern low-power techniques. In fact, it was shown [4] that the Vector Accelerators, while *improving performances up to 10X*, can also *improve energy efficiency of 40-50% on loop kernels and 10-20% on larger program segments*. In Figure 1.5 it is possible to see how a vector needs to wait the previous one to be completed before starting, and so why the latency increases.

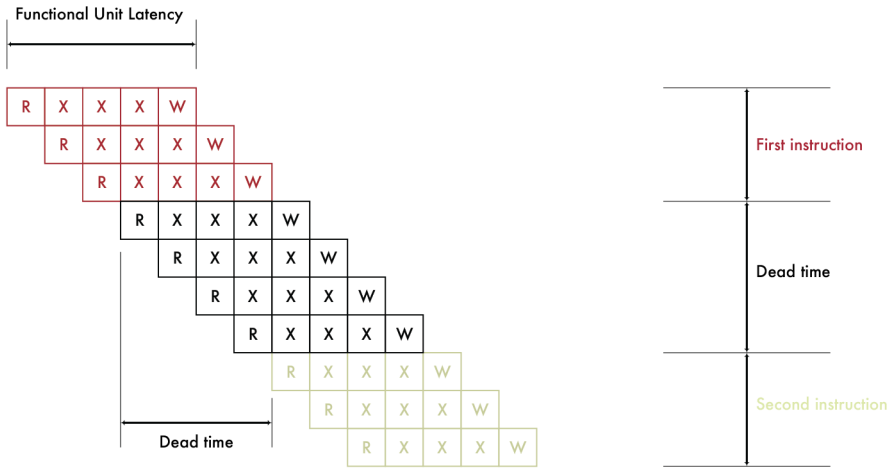

Figure 1.5. Latency penalty on vector processing units [3]

The design implementation that is been used for the EPI project and so in this thesis is the V-Extension (V stands for Vector), and it is in current development. The reference is the V-Extension 0.7.1.

The vector extension adds 32 vector registers, and 5 unprivileged CSRs (*vstart*, *vxsat*, *vxrm*, *vtype*, *vl*) to the base scalar RISC-V ISA [5]. There are also 8 vector predicate registers (vp0-vp7). The CSRs vectors define the configurations.

Privilege	Name	Description
URW	vstart	Vector start position
URW	vxsat	Fixed-point Saturate Flag
URW	vxrm	Fixed-Point Rounding Mode
URO	vl	Vector length
URO	vtype	Vector data type register

Table 1.1. RISC-V's CSRs

The datatypes and operations supported by the V-extension change based on the base scalar ISA [5].

The vector unit must be configured before being used. I.e. the active vector

length is held in the CSR *vl*, which can only hold values between 0 and MVL (Maximum Vector Length parameter) inclusive. The active vector length is usually written with the ***setvl*** instruction.

Vector instructions

Beside the base instructions that we can expect from a Vector Architecture (such as a move, add, xor and so on) it is also possible to find some useful operations related to the nature of the vector calculus [5]:

- **vectorial load/store**: are used to copy data between vector registers and memory. These instructions can be strided or indexed. The strided ones index the vector elements referring to a starting element and then adding (or subtracting) a certain stride to the base address. This kind of load/store is very fast, in particular in some special cases (as unit-strided or some optimized power of 2).

  The vector elements into the indexed ones are basically pointed with a vector of indexes, added to a base address. This process does really slows down the operation but allows to directly select the elements.

- **widening/narrowing**: these operation are used to increase or decrease the size of the vector's contents. In fact, they are very useful when performing operations that need to increase the result size (as example a multiplication between to integer at 32 bit needs to have 64 bit to not loose information). There are also few operations that require the inverse resizing, as for the narrowing.

- **rgather**: those are very particular operations which are very advantageous when manipulating vectors. They allow to index a vector using another vector as index. Therefore, various patterns are possible. The pseudo code representing this operation is the following:

  ```
  for (i=0; i<N; ++i)
    x[i] = y[idx[i]];
  ```

  where y is the starting vector, idx is the index vector and x is the result vector.

- **reduction**: finally, the reduction can perform an operation between a scalar and a vector and give a scalar as result (an easy example could be to calculate the maximum value between all the value contained in a vector and one scalar, the result would either be one of the element of the vector or the scalar).

Finally, all those operations can be *masked*. The masking is a common operation when there is branching or when complex patterns emerge. Normally one bit of the mask represent a whole word or byte into the vector. This bit indicates if the operation of the instruction should be performed for that specific element of the vector.

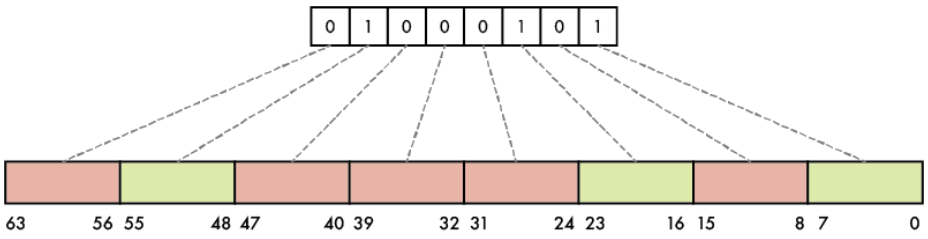

Figure 1.6. 8 bit mask, masking a 64 bit vector

In Figure 1.6 there is an example of an 8-bit vector masking a 64-bit vector. Each bit of the mask corresponds to 8 bit of the vector.

1.1.3 Verification

As the technology scales down, conflicting requirements for high performance, low-power and area arise. This lead to a complex design and to elevate the costs.

It is possible to see in Figure 1.7 how the cost for verification increases drastically with the shrinking of the technology node.

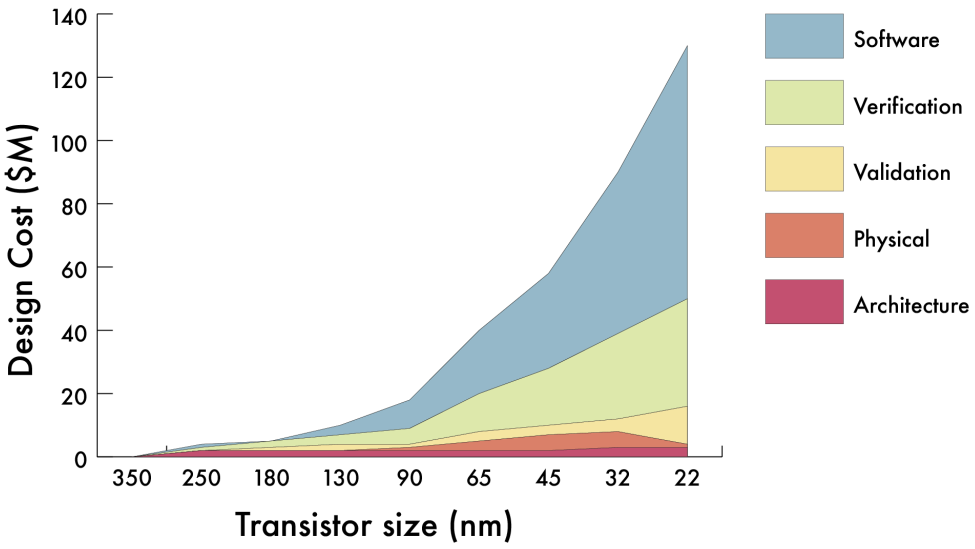

Figure 1.7. The design cost vs the technology node [6]

Verification is responsible to make sure that the design is in track with the specification. So, if the design complexity increases, the same does its verification process.

This is of course an issue for the time-to-market. In particular functional design verification takes 40–50% of the project resources. In other words, increasing the productivity of functional design verification and shorten the design / simulate / debug / cover loop is an essential task [6].

Moreover, the compounded complexity grows faster then the compounded productivity. This gap only means the verification needs to be faster and hence needs to implement more techniques. It is possible to see a study on the complexity/productivity gap in Figure 1.8

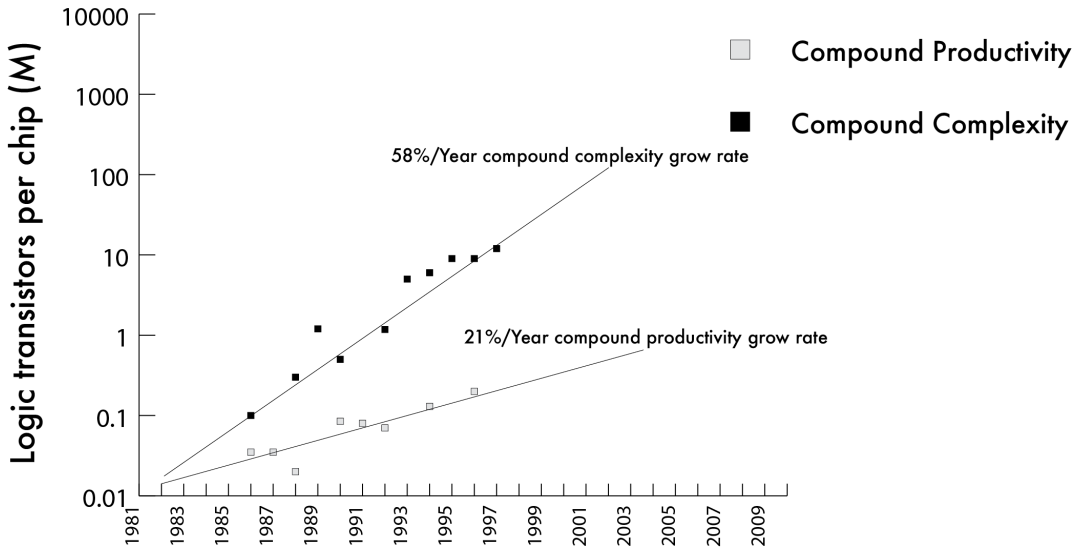

Figure 1.8. The gap between the complexity and the productivity [6]

This graph shows how the complexity rate is higher than the productivity rate during the years. This means that it is not possible to reach an equity only by increasing the number of tests or by adding some more checkers. More advanced techniques are needed to cope with the growing complexity.

In order to verify an RTL design, its specifications need to be stated as code. In this thesis all the tools implemented for the verification process are wrote in *SystemVerilog*. SystemVerilog is an hardware description language, born as an evolution of the Verilog language. Some extensions such as Objective-Oriented Programming elements and specific keywords for verification tools were added to it. The functionalities of a design can be stated by using the ***checkers***.

Checkers

They are mainly composed by *assertions* and *assumptions*, some really powerful statements used to define the constraints of the DUT's behaviour.

Assertions and assumptions are syntactically identical, but the first one refers to the output signals while the second one to the input signals. They are mainly composed by two parts which are both conditions, even though they act differently. The first part is a condition to *"fire"*, this means that when the condition is true the assertion, or assumption, starts checking for the second part [7].

The second part is another check on some condition, and can reveal the result of the check. At this point the possible results are *True* or *False*.

Both of them can be time consuming, and conditions on the edge are possible. It is also possible to create *properties* and *sequences* and then embed them into the assertions, in this way it is possible to reuse those parts.

Let's now see the basic structure of an assertion.

```
1 property property_1;
2     @(posedge clk_i)
3         a |-> b;
4 endproperty : property_1
5
6 assertion_1 : assert property ( disable_iff(rst) property_1) else $error("")
```

Basically if a is HIGH('1') then b is also expected to be HIGH('1').

The functionality is expressed into the property, then it is asserted into the assertion. It could be possible to recall tasks or processes, in this way it is possible to have more computational capabilities. It can also provide an error message to better identify the problem during simulations. When the signals into the *disable_iff* function are asserted (in the example above it is the *rst* signal) the assertion is disabled. This means that the assertion does now need to be re-fired with a new starting condition.

Functional verification

The functional verification needs to verify whether the functionalities described into the specifications are met into the design. In order to verify functionally, the functionalities need to be defined. This is a very important part of the process, as this translation is never perfect and often highlights some critical points into the specs, such as omissions or inaccuracies. The functional verification is performed using simulations, this means that time is required to simulate the design behaviour and to check it against its specifications.

In functional verification the assertions and the assumptions are implemented in the simulation process to verify to correctness of the RTL design.

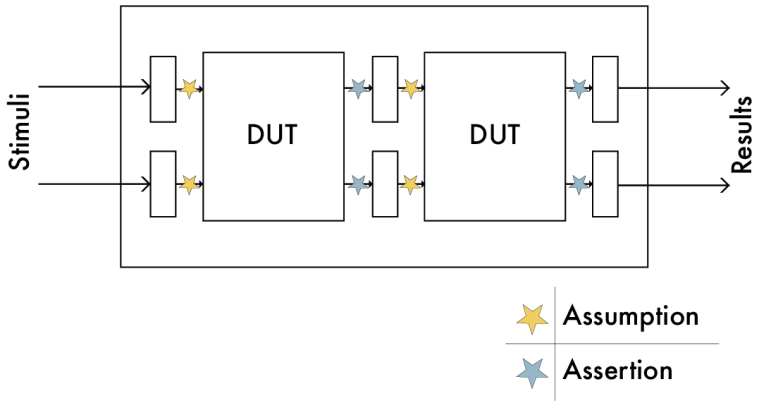

Figure 1.9. Assertions and assumptions into an RTL design

In Figure 1.9 it is possible to visualize how assertions and assumptions are used to verify functionally the RTL design. The assumptions are mainly checking the inputs and the assertions are mainly checking the outputs. It is possible to notice that the assertions for a DUT can be considered as assumptions for another one.

Formal verification

The formal verification can be done in different ways: *theorem proving*, that tries to prove the equivalence between specifications and design by using mathematical reasoning; *equivalence checking*, useful when performing optimization to the design, trying to demonstrate that the various versions are mathematically equivalent; and *model checking*, which try to find counter example on the behaviour of the design, and, if a counterexample exists, the Formal tool provides an example of the specific case to demonstrate the falsity. It is performed using *assumptions* to assume the design behaviour and then using *assertions* to test it.

As already mentioned, the assertions and the assumptions in simulations are treated equally . This is not true in the formal verification [8]. In order to be performed, the formal assertions verification only needs to instantiate the RTL file and the checkers file. Then it uses the assumptions as drivers for the RTL, and finally it tries to prove wrong the assertions using mathematical simplifications. Hence, here it is possible to see how the assumptions work differently. If a property is assumed it can never be proved wrong. Only assertions need to

be checked. The formal tool, in this project, was only used to perform model checking. The tool used is provided by Mentor, and it is called Questa Propcheck. There are different possible outcomes occurring while verifying:

- **vacuous proof**: the case is vacuously proven when the assertion is always true given the assumptions. This means probably there is some mistake into the assumptions or the assertions;

- **proof**: starting at the initial state, no legal stimulus exists that causes in the assertion to be violated. It is normally proven in the first 10-100 cycles;

- **firing**: a counterexample to the assertion exist, this means that there is a bug in design, or the specifications are not well defined;

- **firing with warnings**: a counterexample to the assertion exist, but it does not use primary inputs. This means that there are uncontrolled internal values that cause a fail;

- **inconclusive**: the formal analysis timed out before demonstrating the assertion. This could be also meaning the complexity is too high to be elaborated with math tools;

In Figure 1.10 are graphically summarized all the possible outcomes previously exposed. It is shown how the time required to prove an assertion grows exponentially with the depth of the analysis. There are also illustrated the various "zones" in which the result is produced.

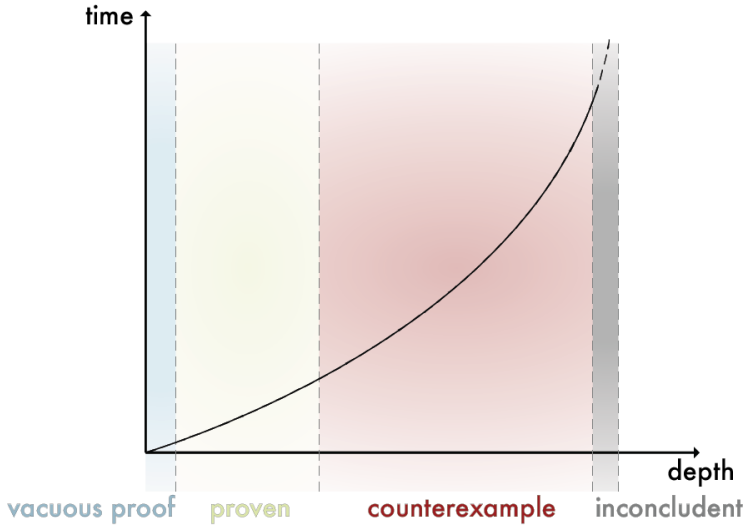

Figure 1.10. Growing complexity and expected results in formal verification.

Coverage

The whole process of verification does also need to have a direction. Coverage is employed to do it.

Coverage is very useful and can be performed on functionalities or code:

- functionality: it measures the cover of all the functionalities stated by the specifications. This approach makes it possible to assure that all the requirements are met. But this also means that there is no information about unused RTL.
- code: it measures the cover of the RTL code. This means that it is possible to acknowledge whether there is unused logic and possibly some branch of the flow. Although, this implies that there is no check about the functionalities implemented.

The main goal is to take the coverage to 100%. But at the same time it is easy to fool the coverage, because it depends on how the cases are taken into account.

In order to use all of these techniques there is the need of a structure to contain and control them. This structure is designed following the ***Universal Verification Methodology*** (UVM). The UVM is thought to verify circuits

designs, and it is provided of Object-Oriented Programming elements, in order to enhance reusability. Later on this Chapter the particular case of this work is discussed.

1.2 Context

Prior to start, it is important to highlight that the VPU is designed to be the co-processor of a scalar core named Avispado. This core is under development by SemiDynamics, which works together with the BSC at the EPI project. In order to implement and test the VPU a simulator of the Avispado core was used, it is called Spike. It was initially developed to simulate RISC-V core, but for this project it was extended to simulate also the Vector-Extension. So, Spike is simulating both the core and the VPU, to use it as a scoreboard. In Figure 1.11 it is possible to see the interface between Avispado and the VPU for all the instructions.

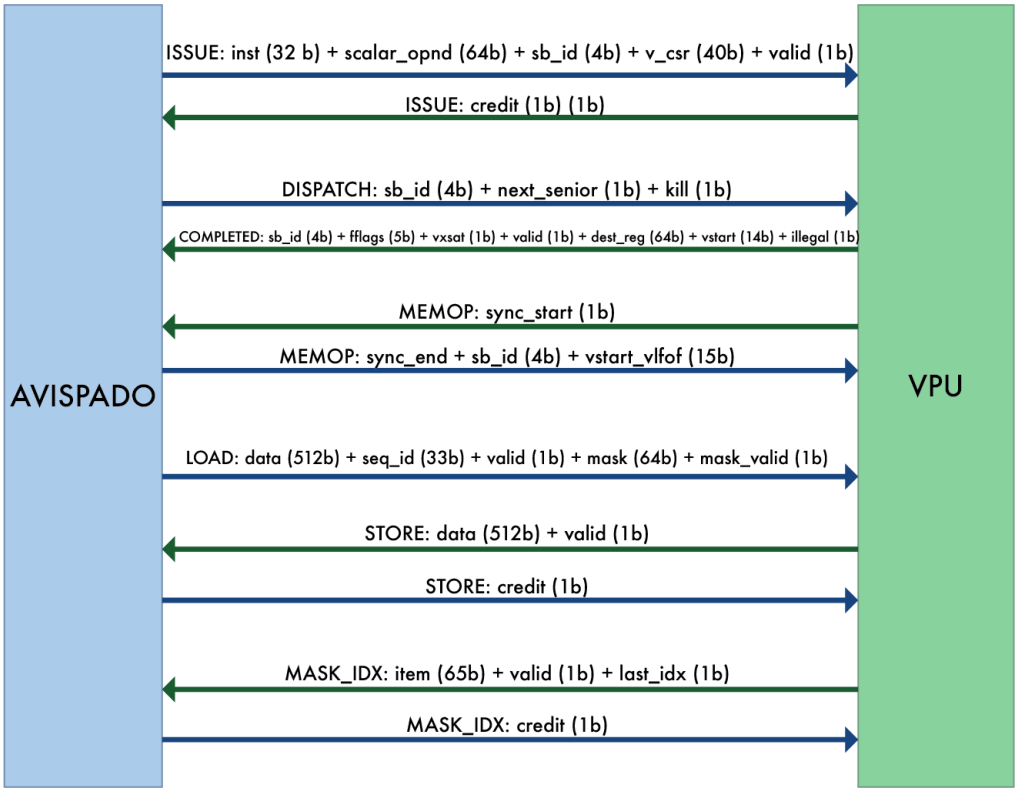

Figure 1.11. Avispado-VPU interface

The instructions are issued with a credit system, whose number corresponds to the depth of the issue queue of the VPU. All the informations about the interface are reported into the paper published by SemiDynamics [9]

1.2.1 The VPU

In Figure 1.12 it is possible to find a simplified scheme of the implemented VPU.

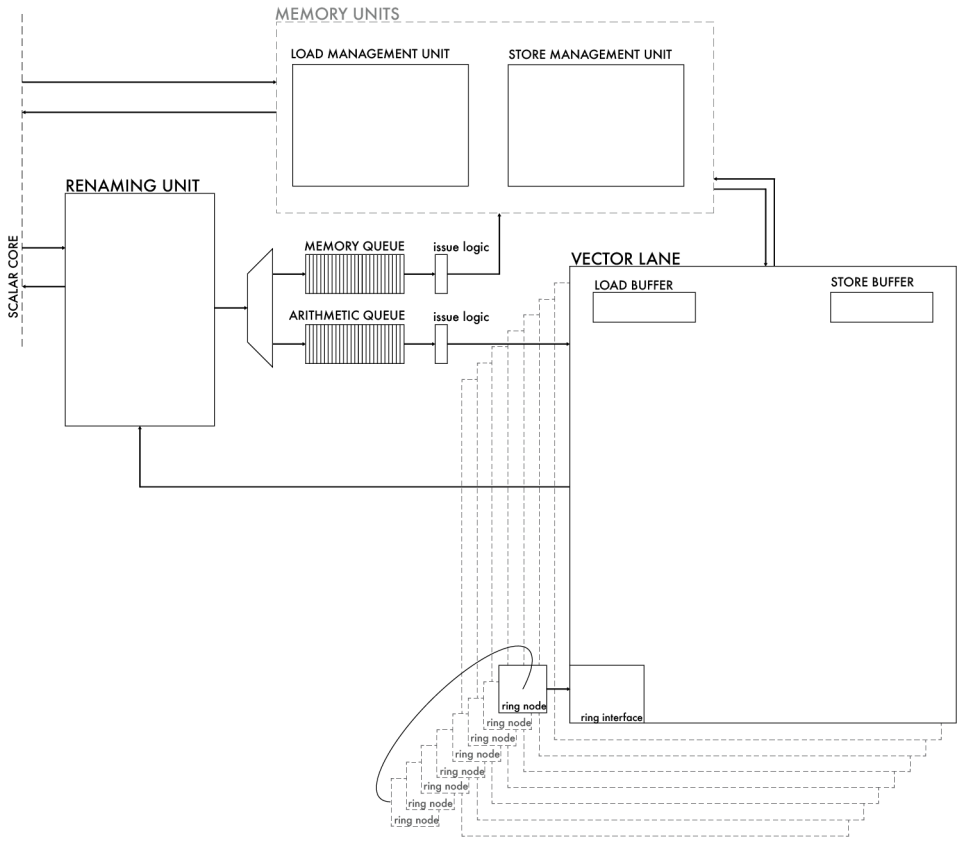

Figure 1.12. EPI project's VPU

It is important to notice that the VPU implemented is a decoupled architecture, this means that two operations can be executed at once only in case they are not both memory operations nor arithmetic ones.

The main elements composing the VPU are:

- **Renaming Unit**: the scope of this unit is to remove false dependencies due to the naming of the registers. This is possible because of the virtual

memory. The processor only uses logical registers, which in this component they are mapped to physical registers using the Free Register List (FRL). Write-to-write dependencies are removed if possible, in order to avoid Data Hazards.

- **Queues**: the VPU is a decoupled vector architecture, this means that the arithmetic instructions and the memory instructions are buffered in different queues. In Figure 1.12 it is possible to see how the two queues are independent in the issue stage. The vector lane is capable of performing only one arithmetic instruction at time, so the issue queue must wait until the previous instruction finishes its execution, but simultaneously a memory instruction can be performed. The whole issue stage is composed by the queue and the issue logic. The number of entries in the issue queues is parameterized.

- **Memory Units**: as mentioned before, the only instructions that can access memory are the load and the store operations. There are different units working on memory instructions: The *load management unit*, the *store management unit* and the *index and mask unit*. As the name suggest their core task is respectively to load operations, to store operations and to mask the operations and provide the indexes. The possible addressing modes are strided or indexed.

- **Vector Lane**: the Vector Lane is the core of the VPU. It is composed by different vector processing lanes. This is a very common technique that allows to improve performance and scalability. In an ideal multi-lane vector architecture all the lanes are working simultaneously and thus efficiently, with the cost of more hardware to control the synchronization.

  One of the most important submodules of the Vector Lane is the Vector Register File (VRF). This is designed with only one read/write port, this because it is important to limit the area usage, and so to increase the operating frequency. It is necessary to have a buffer in order to avoid streaming problems and bubbles inside the pipe.

  This solution implies the existence of some cost in terms of latency, due to the starting of a new instruction. Inside the Vector Lane the Write-Back

Buffer (WB) and the Load Buffer (LB) are important as well. Those buffers store the data until the VRF line is complete. There is also the Store Buffer (SB) which holds the data read from the register file and then sends it to the Store Unit. Eventually when an instruction is completed the physical register are freed.

The VPU can be configured with different numbers of lanes from 1 up to 8, the default value is 8.

- **Lane Interconnection**: due to the range of the Lanes, which can be any value between 1 and 8, it is important to have a system capable of synchronizing all of them. There are also some operations which require to use multiple lanes at once to perform the execution, so an unidirectional ring intercommunication is implemented between the lanes.

It is also important to point out how the memory is organized in order to understand how some operations work.

The vectors are distributed into the different lanes. Indeed it is possible to say the Vector are sliced into the lanes, this means that an efficient organization is needed in order to understand where to put the right data.

According to the RISC-V V-extension, vector elements can have different sizes. The parameter that describes the size of an element is the Standard Element Width (SEW). This is determined by the Control and Status Register (CSR) named *vsew*. The maximum supported SEW into the EPI project is 64 bit. Also the maximum VLEN is equal to 16384 bit, so 2kB.

The number of elements a vector registers holds is given by VLEN/SEW, according to the Table 1.2.1:

SEW	ELEMENTS
64	256
32	512
16	1024
8	2048

In Figure 1.13 it is possible to see the structure used for SEW = 64 bit as examples.

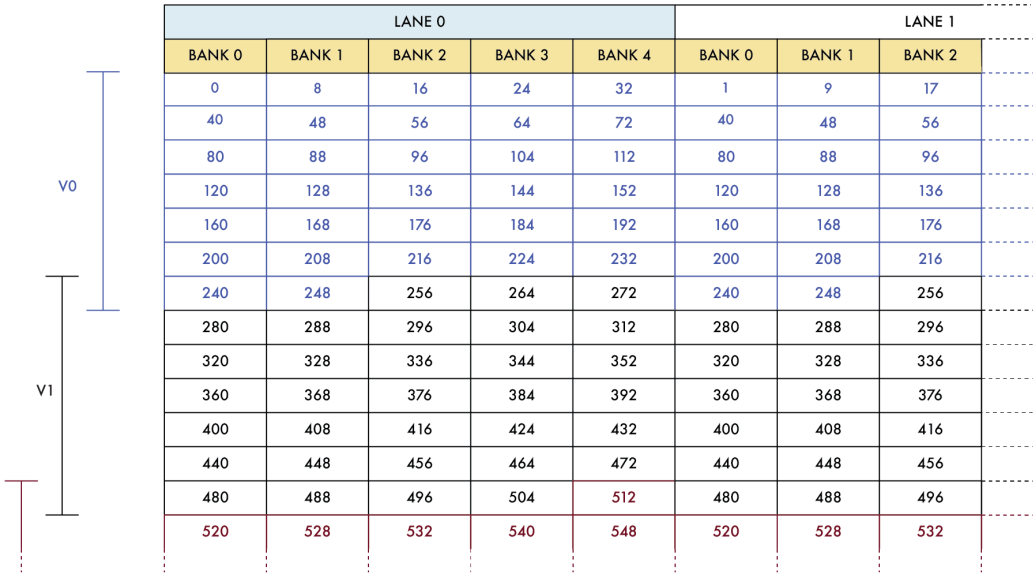

Figure 1.13. VRF with SEW = 64

Notice that the sub-banks are more than 1 only when the SEW is not equal to 64.

In Figure 1.14 it is possible to identify the sub-banks as subdivision of a bank.

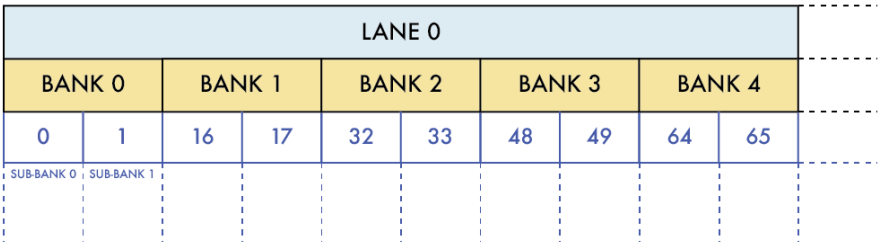

Figure 1.14. VRF with SEW = 32

1.2.2 The UVM

As mentioned before, the UVM is the solution to the standardization problem of the verification. In fact, it is a transaction-level methodology (TLM) designed for testbench development. It is a class library that facilitates writing configurable and reusable code [6].

The testbenches created are designed to be reusable, as a result less code and more production is achieved.

This is mainly obtained with the polymorphism. It means an object can be used to define a reusable class. Operating with this technique an object can be used as a super-class and other sub-classes can be created.

It works following an hierarchy method, and every component can only operate with the components above it. In this way multiple sub-components can be instantiated.

Generic UVM

In Figure 1.15 it is possible to see an example of a standard UVM.

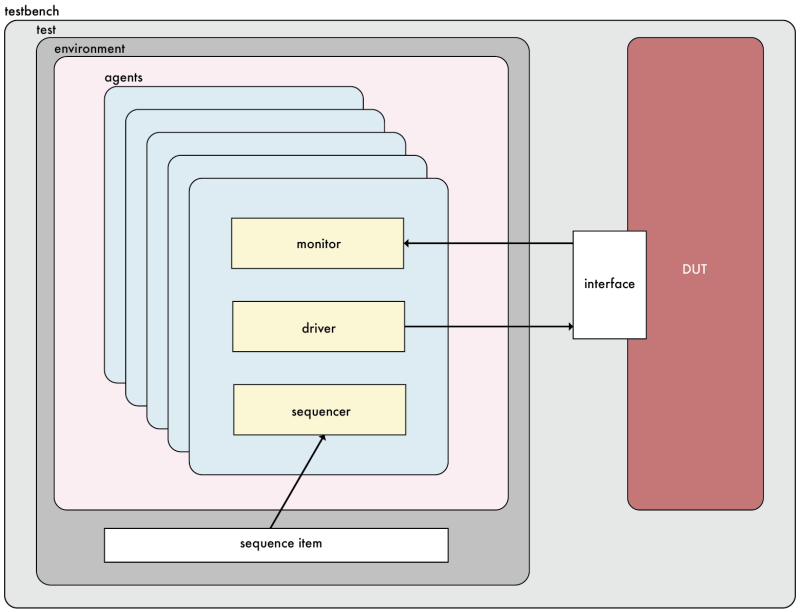

Figure 1.15. A generic UVM

Let's proceed analyzing some elements of a standard UVM and its implementations in the EPI project.

- **Testbench**: typically, it instantiates the DUT (Design Under Test) and the UVM Test Class. Also it is responsible to configure the connections between them. There are different ways to communicate into an UVM and in this module is possible to choose them. It is also important to say that the Testbench instantiates the Test dynamically at run-time, enabling to compile it once and run many Tests.

- **Test**: this is the top-level component into an UVM testbench. Typically a base test class is defined to instantiate the top-level environment and then it is extended to the specific case.

- **Environment**: it is a component that defines the environment of a test. It aims to define all the agents and the scoreboards. The top-level environment instantiates and configures the reusable verification IP and defines its default configurations based on the application of the test.

  Typically there is a different environment for each interface of the DUT.

- **Sequence Item**: it is the fundamental lowest denominator object in the UVM hierarchy. It can also be defined as a transaction and it is the smallest data transfer that can exist in a UVM. It can include variables and constraints.

- **Sequence**: it is generated by the environment using the sequence item. It is an ordered collection of transactions. Mostly it can impose some constraint to the variables generated into the sequence item.

- **Agent**: it is one of the most important components of the UVM. It groups together all the components that are dealing with the DUT and so a specific DUT interface. Normally there is a different agent for every interface or DUT. This allows specific sequencers for specific stimulus. It can also be active or passive depending on its action on the DUT: if it sends signals to stimulate the DUT it is considered active, and otherwise passive.

  It contains:

  - **Driver**: it is the component responsible for the communication between the UVM and the DUT at pinlevel. It receives the sequences from the sequencer and then converts them into signals, following the interface protocol. This action is observed by another component, the (command) monitor.

    It can be turned off when the agent is defined as passive. In this way there is no other component sending signals to the DUT.

  - **Sequencer**: it controls the requests and the responses between the driver and the sequence item. So it is a controller.

  - **Monitor**: it observes the outputs of the DUT at pinlevel. Then transforms those signals into transactions for the analysis. On a larger stage, those transactions are very likely then compared with the expected outputs. This is normally done in the scoreboard. It can perform internally also some processing. The signals the monitor is observing could be monitored into the driver, but this would mean violating the modularity choices for the UVM.

- **Scoreboard**: it is a checker for the outputs of the DUT. It compares the transactions obtained by the monitors against a predicted result. There are different ways to generate a predicted result and so a scoreboard, it often uses a C/C++ model, but it is possible to use also other languages as well.

It is important to point the UVM works in different phases. It is possible to macro-divide the process in 3 phases: the *build phase*: here the components are constructed from the top, in this phase an important sub-phase is present, the *connect sub-phase*: here all the components are connected upwards; the *run phase*: in this phase the simulation is ran, and finally the *cleanup phase*: here all the results are checked and reported.

In Figure 1.16 it is illustrated a simple scheme representing all the phases.

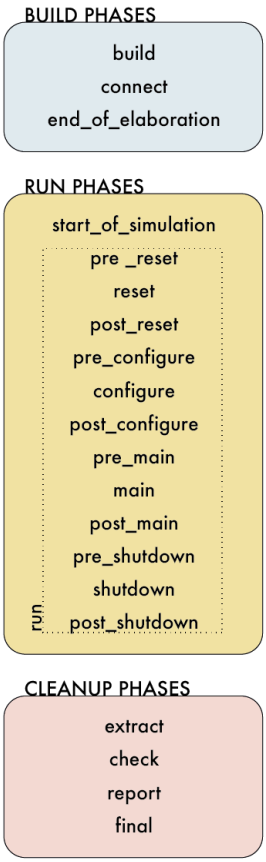

Figure 1.16. UVM phases

In the EPI project two different UVM structures were implemented. One to test all the VPU (and its interface with the scalar core) and another one to test the submodules. This approach was chosen because it is not always possible to test all the corner cases with an UVM testing the whole VPU.

Let's proceed with a brief analysis of those two structures.

Main UVM

The main UVM is interfaced with Avispado and it is composed following the standard structure. It was mainly used to support the use of a scoreboard (Spike) and then to implement then some checkers and some coverage controls.

The majority of the automatic tests are made with this structure, and they are scripted to run all the night and to produce a valid dataset to then find and fix the bugs. Those tests are created using random RISC-V vector instructions, with some constraints that allow the tests to be valid.

Using a simulator for Avispado as input, the driver is a little bit different from a standard one. Indeed, the input are not controlled at pinlevel but vector instructions are sent to the core, which will then produce a correct input for the VPU.

Those instructions are ultimately compared with with the scoreboard and a result is produced.

Submodules UVM

Another structure was necessary due to the different interface, and so to different drivers. It is not always trivial to create a driver for a submodule, because the handshake processes can be very complex, thus it would be possible to introduce some error in the verification structure.

As an UVM structure is useful when driving an monitoring all the signals of a module, this technique was used for the Load Management Unit but not for the Load Buffer. This because the handshake process for the Load Buffer is complex, and so would have been the UVM, with the possibility to introduce some errors. The structure of the LMU's UVM is represented in figure 1.17.

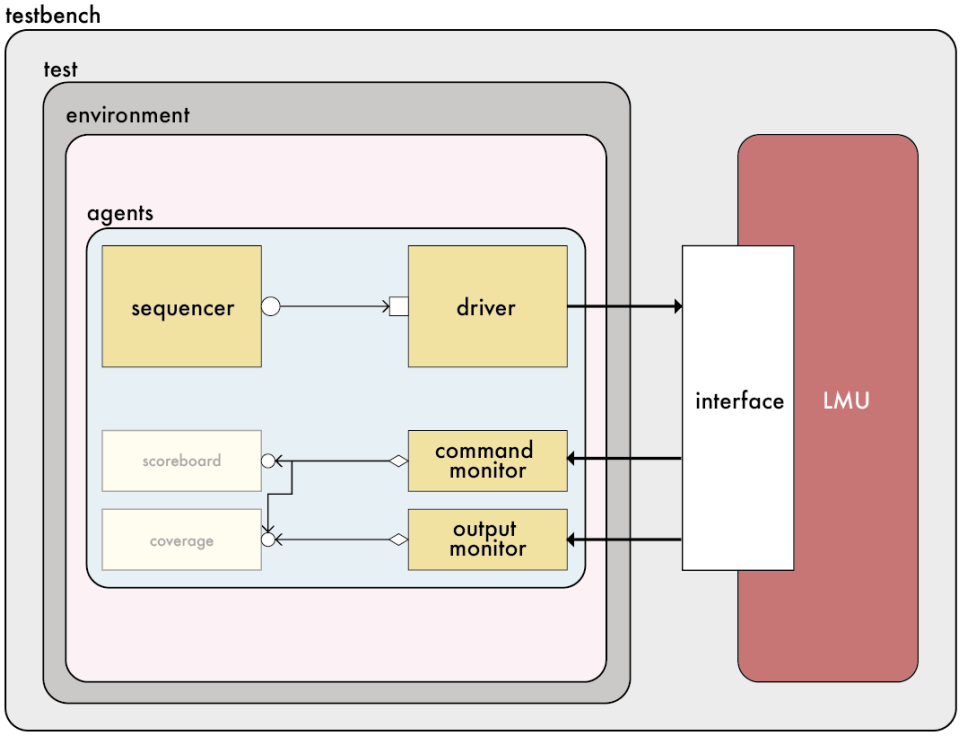

Figure 1.17. A small UVM for the submodules

It is possible to observe the scoreboard and the coverage are disabled in this application. In reality a scoreboard was implemented for each of them, but was handles with a different methodology discussed later on the next Chapter.

Chapter 2

Contributions

2.1 Submodules and Specifications

The main focus of this thesis will be on two important submodules in charge of the load instructions:

Load Management Unit and *Load Buffer*.

2.1.1 Load Management Unit

This module is in charge of handling the load operations. It receives cache lines from Avispado containing the data for the load. Then, the LMU rearranges the data to distribute it to the vector lanes. The load operation can be strided or indexed. Since the VPU is able to work out-of-order and with two loads in parallel, an ID system is necessary, so each instruction comes with an ID. In particular the load operation comes with the signal *seq_id_i*. This signal identifies the load and contains all the determinant informations about it.

The main elements of this submodules are:

- **Shifter**: it is useful to have the first bit in other position than the MSB or the LSB position;

- **Compactor**: it is useful to compact all the valid elements. When the stride is equal to one the compactor is not used;

- **Aligner**: it is in charge to align correctly the elements for the output (lane, bank and sub-bank).

The main parameters defining this submodule are:

- **MEM_DATA_WIDTH**: width of the chunk of data received from Avispado. The standard value is *512*.
- **SEQ_ID_WIDTH**: width of the *seq_id_i* that identifies the data coming from Avispado. The standard value is *33*.
- **MAX_NUMBER_ELEMENTS**: maximum number of elements that can be encoded in the chunk of data received (64 when SEW = 8 bit). The standard value is *64*.
- **MAVISPADO_LOAD_MASK_WIDTH**: Indicates the maximum number of mask bit that are received with the data. Every bit of the mask represents a byte into the data. The standard value is *64*.
- **NUM_LANES**: number of lanes. The standard value is *8*.

Interface

Signal	Description
load_granted_i	a load is granted, it will have a certain sew_i and stride_i
load_granted_sb_id_i	the id for the issued load, can be issued up to 2 loads
indexed_load_granted_i	the granted load is indexed
load_sync_end_i	indicates if the load is ended
load_sync_end_sb_id_i	indicates the load id of the ended load
load_data_valid_i	indicates if the data in load_data_i bus is valid
load_data_i	data received from Avispado
seq_id_i	the sequence id (described below)
mask_valid_i	validity of mask_i signal
mask_i	mask bit to mask load_data_i
sew_i	identifies the size of each vector element
stride_i	stride indicated in bytes

Table 2.1. Input signals to the LMU

Signal	Description
load_data_o	output data sent to the lanes
load_dvalid_o	indicates if the data in load_data_o bus is valid
mask_o	mask bit to mask load_data_o it is needed also for not masked inst.
element_ids_o	identifies each element sent in load_data_o
sb_id_o	identifies the instruction
vstart_self_o	identifies the first valid element in the chunk of data received in load_data_i
vstart_next_o	identifies the last valid element in the chunk of data received in load_data_i
min_element_id_idx_o	index of the first valid element in elements_ids_o of each lane.

Table 2.2. Output signals to the LMU

In the Tables 2.1 and 2.2 are reported respectively the input and the output signals of the LMU, together with their description.

Sequence ID

The sequence id ID is necessary as the memory system does not guarantee the in-order arrival of elements. Hence, this signal contains all the informations needed to correctly elaborate the data.

The *seq_id_i* is composed by:

- **seq_id_i[4:0]** = *v_reg*, identifies the logical vector register in which the data should be written;
- **seq_id_i[15:5]** = *el_id*, identifies the lowest valid element id contained in the chunk of data being transmitted;
- **seq_id_i[21:16]** = *el_off*, identifies the offset in the chunk of data being transmitted;
- **seq_id_i[28:22]** = *el_count*, identifies the number of valid elements being transmitted. Masked elements are valid elements;
- **seq_id_i[32:29]** = *sb_id*, scoreboard id of the load instruction that requested the data.

Handshake

The handshake protocol with the different components is as important as the data manipulation. Following, there is a simple example with the stride value equal to 1 and offset equal to 0 (meaning that there is no manipulation on the input data). The procedure for the handshake is the following:

1. Up to 2 load are granted by memory queue with the *load_granted_i* signal. The SEW and stride information of the relative instruction are passed;
2. Avispado sends the data and the seq_id_i with its *seq_id_i*;
3. The next clock cycle, the data is at the output of the LMU towards the Vector Lane;
4. When a load is finished, another load can be granted.

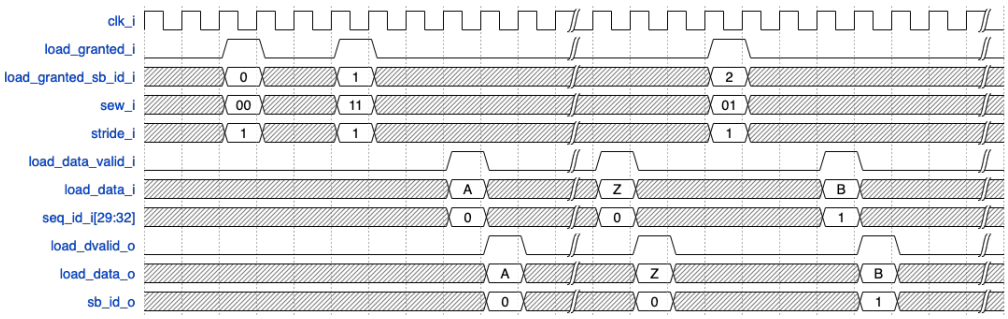

Figure 2.1. Timing Diagram unit-strided load for the LMU

In Figure 2.1 it is possible to see an example of an unit-strided load. This is a simple load, and its behaviour is discussed in the following section.

Strided Load

In this case all the valid elements are separated by a constant stride. If the stride is equal to 1, then it is called unit-strided load. Figure 2.2 shows how the LMU works in this case. The parameters (defined in the *seq_id_i*) are defining the elements to consider.

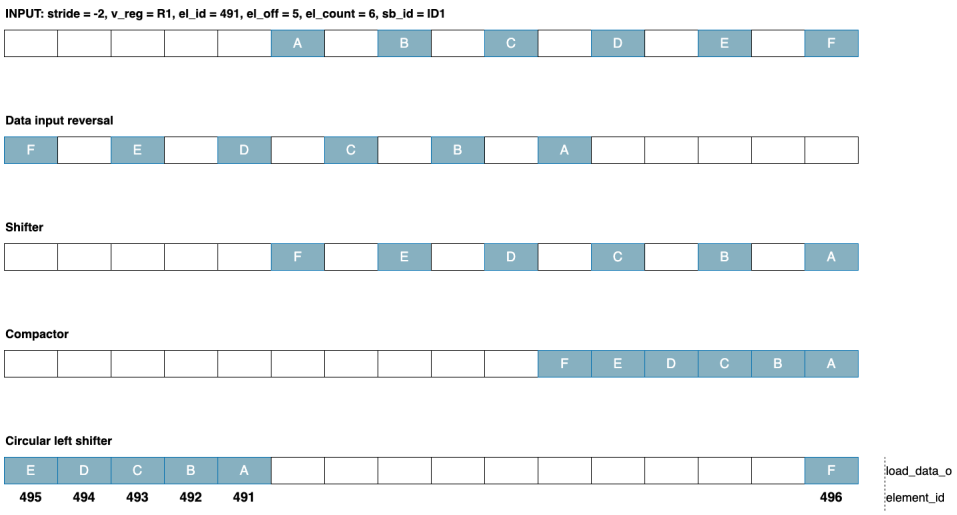

Figure 2.2. Strided load handled by the LMU

Indexed Load

It is also possible to load values in an indexed way. By design, it was chosen that only one valid element at time can be sent, and so *el_count* needs to be equal to 1. If the indexed load is requested with many valid elements, the cache line is sent multiple times with only one valid element each time. In Figure 2.3 it is possible to see an example of a simple indexed load.

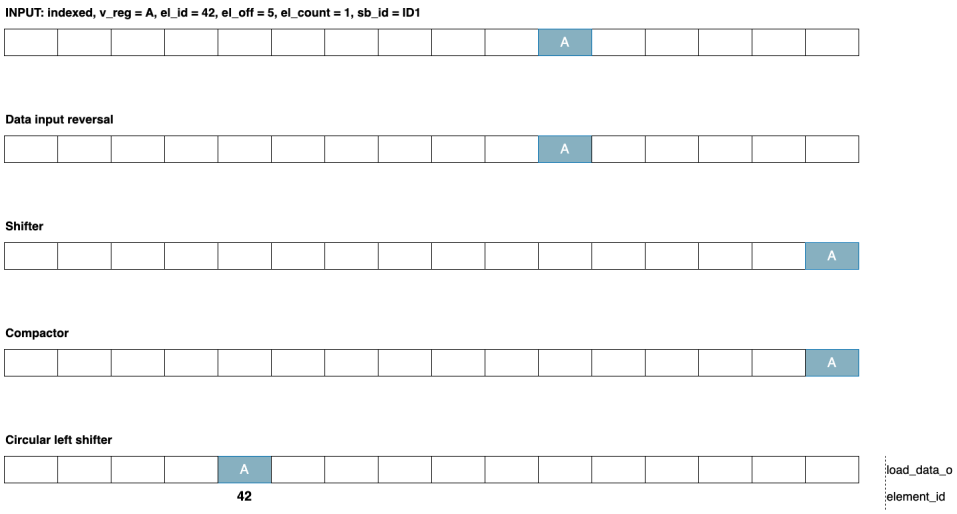

Figure 2.3. Indexed load handled by the LMU

Masked Load

All the load operations can be masked. This operation does not change the number of valid elements, but if an elements is masked, at the end of the process it will not be sent. In Figure 2.4 it is possible to see an example of the simple strided-load shown before, but this time, with a mask.

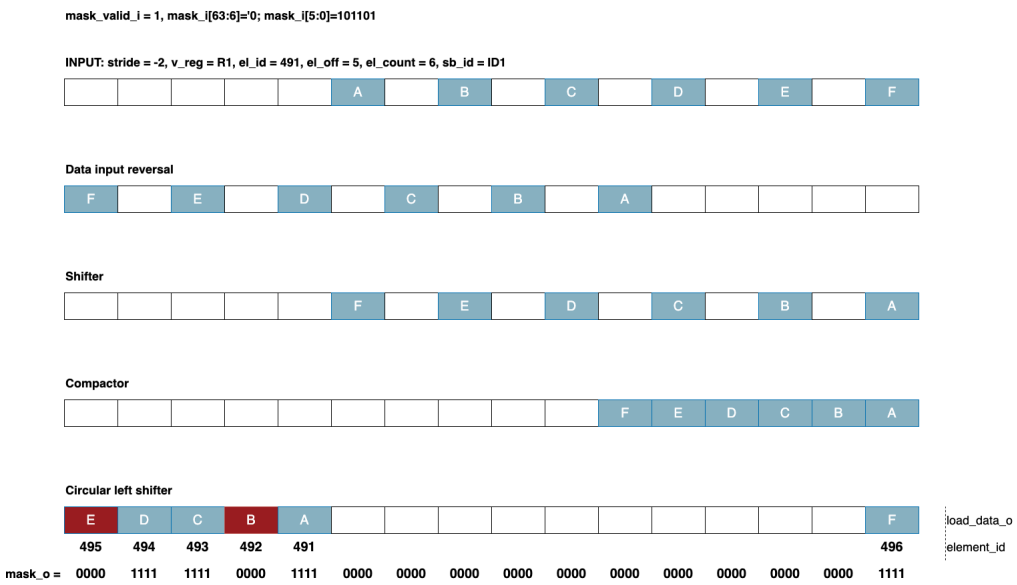

Figure 2.4. Strided load, with mask, handled by the LMU

2.1.2 Load Buffer

The second submodule that is mainly in charge of the load operation is the Load Buffer. Its function is writing the data sent by Avispado to the Vector Register File. The data can come from different instructions inflight, and the Buffer will always try to optimize and group the data to write.

The LMU receives full cache lines of 512 bit from Avispado and forwards them to the corresponding Load Buffer for each lane (64 bit max per lane), depending on the *seq_id_i*. The position of the data into the LB will be determined by the element ID. Up to two loads can be inflight, but their associated cache lines can arrive out-of-order, although the implementation will be parameterized to accept N loads in flight.

Interface

The Load Buffer is connected with a lot of modules. From the *Vector Lane* it receives the clock and the reset and communicates if there is a load inflight; with the *LMU* it exchanges all the data; from *Avispado* it knows if the operation has terminated; the requested handshake is handled with the *Memory Queue*; the data goes to the *Vector Register File*, synchronizing with the internal *Finite State Machine*; finally it can commit the result with the *Commit Unit*.

A simple connection structure is represented in Figure 2.5.

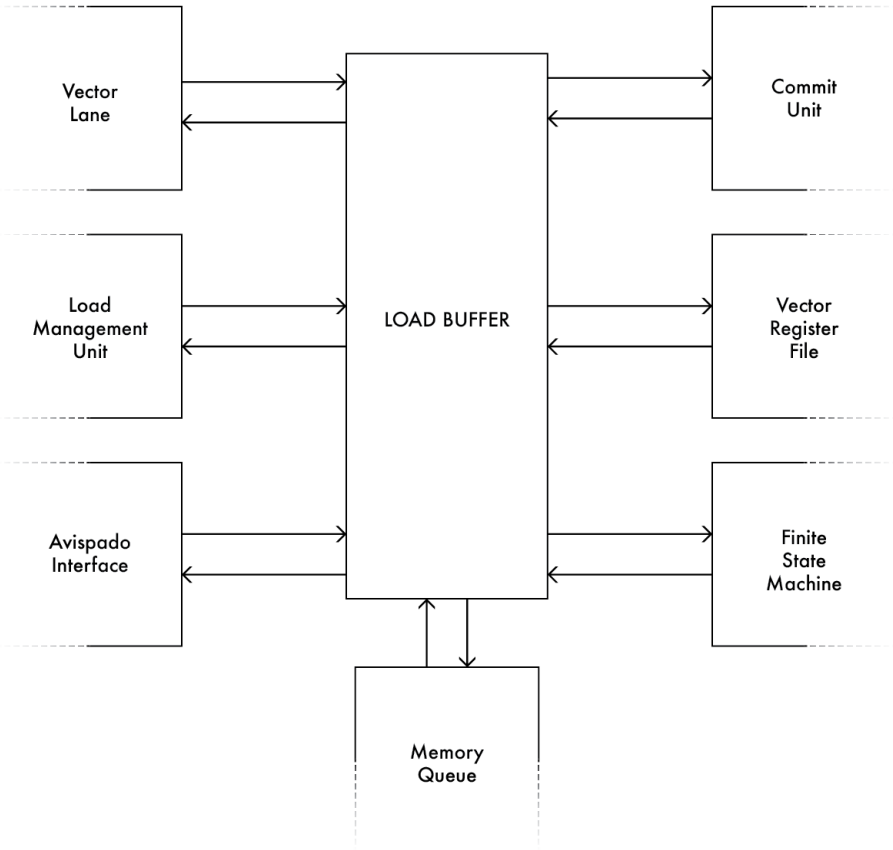

Figure 2.5. Load Buffer's interfaces

Structure

There is a Load Buffer of each lane and there are three layers to buffer the elements. The layers are divided by the concept of Element Group. In fact, the

elements cannot be disposed in every combination, but every element, based on its element ID, has an exact location (based also on the SEW). The position of the elements follows the structure of the VRF explained in the first Chapter.

In Figure 2.6 it is possible to see a general connection between the LMU and the various Load Buffers.

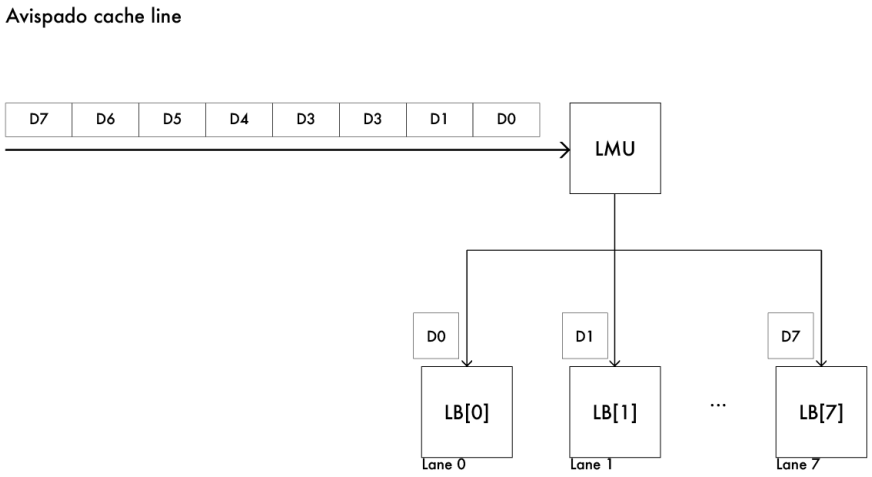

Figure 2.6. Data through the LB

In Figure 2.7 there is an example of the disposition of the elements of 64 bit. It is also important to say that the number of bits for each element group stays the same, this means that the number of elements depends on the SEW.

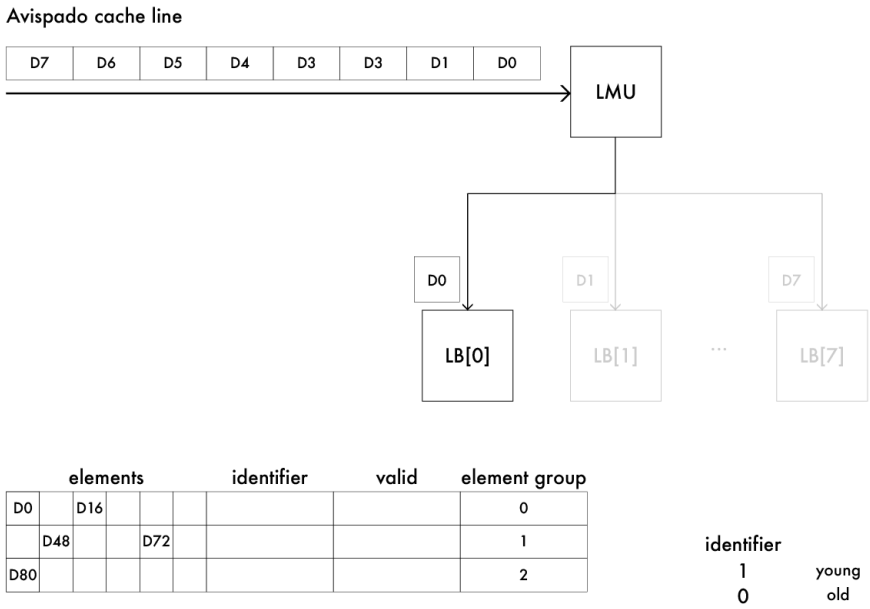

Figure 2.7. General case for the LB

The Load Buffers are mainly composed of 4 parts:

- *Elements*: are the values stored into the buffer. The elements are always 512 bit long, so there are 5 elements with SEW = 64, 10 elements with SEW = 32 and so on;
- *Identifier*: identifies the elements coming from the young or the old load;
- *Valid*: identifies if the elements are valid;
- *Element Group*: identifies the group of elements into the Vector Register File.

Retry

There are cases in which three buffers are not enough to store all the elements. It is possible that more than three elements try to occupy the same position and thus causing a problem. This case is handled with a retry mechanism: one of

the elements is discarded and a new request is made to Avispado, to notify the retry. It is important to identify the element to discard when there is a retry, whether it is a new or an old one.

There are 4 possible cases:

1. if the incoming data comes from the *young load* and the buffer does only contain *data from the same load*, the data with the highest element group will be discarded;

2. if the incoming data comes from the *young load* and the buffer contains *data from both the oldest and youngest loads*, the incoming data will be discarded;

3. if the incoming data comes from the *old load* and the buffer only contains *data from the same load*, the data with the highest element group will be discarded;

4. if the incoming data comes from the *old load* and the buffer contain *data from both the oldest and youngest loads*, the data inside the buffer, the one from the young load will be discarded.

Flow

In Figure 2.8 it is possible to see a simple scheme of the LB's behaviour.

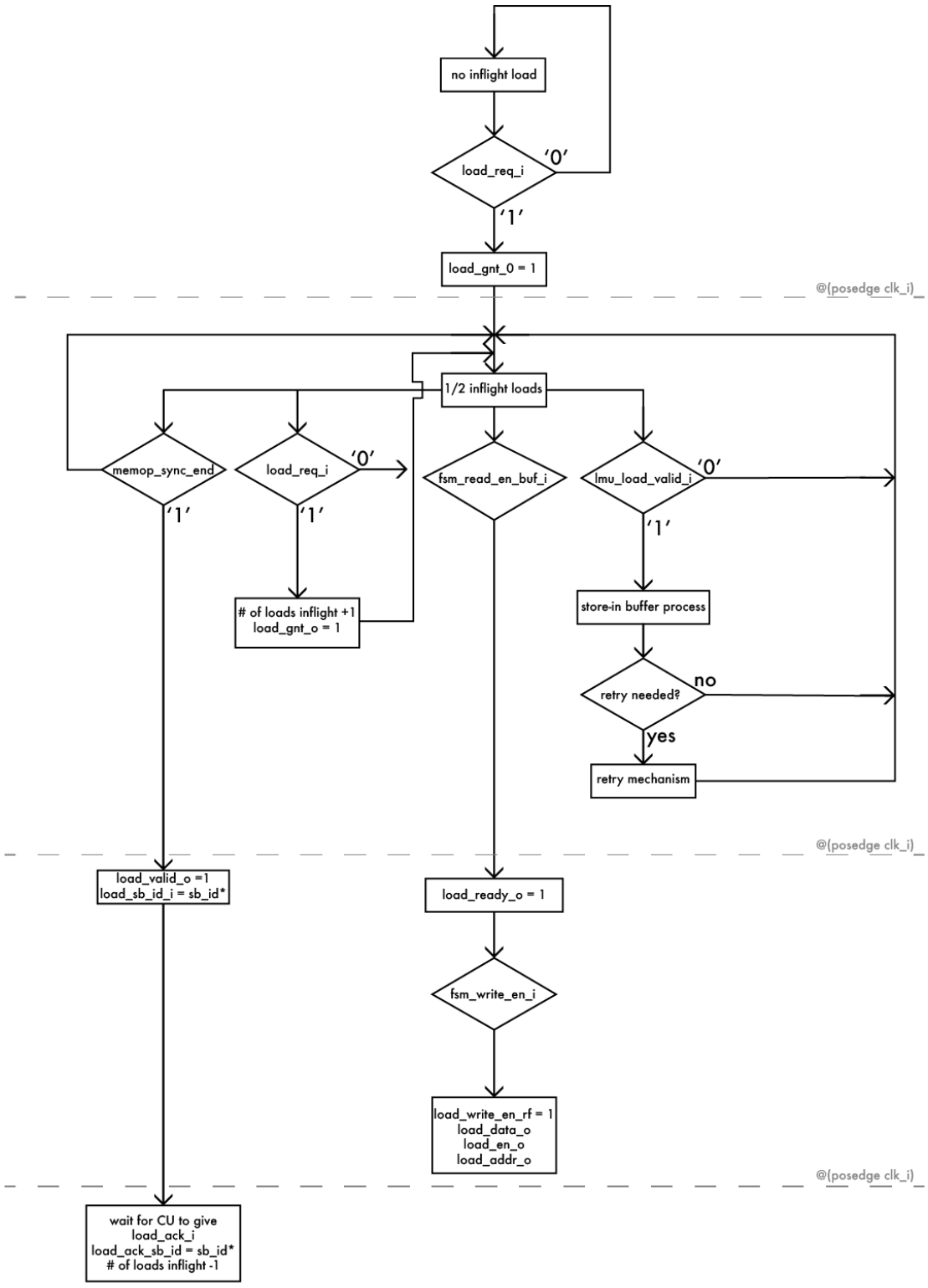

Figure 2.8. Working flow of the Load Buffer

Following the scheme it is possible to understand the main working flow of the Load Buffer. When the Load Buffer receives a request, it is granted if the number of loads inflight is less than two. For each request the LB can receive the data, activating the retry mechanism if necessary, or it can receive a *memop_sync_end* and then continue the process to finish the load. During this process it is possible that *fsm_read_en_buf_i* is asserted by the internal Finite State Machine of the Vector Lane, thus the LB can write the data in output.

2.2 Verification Plan

In order to have a better approach with the verification of a design, it is important to define a Verification Plan. There are many ways to write it. In this specific case, the Verification Team was working along with the Design Team to produce better specifications and not only to test the completed design. This means that the Verification Plan is done following the general rules to have a good value in future, but it is not entirely defined a priori.

Defining the approach to have does also mean defining the tools that will be used, in fact, many different of them were created, such as a *Test Plan*, an *UVM* and the *checkers*.

2.2.1 Test Plan

In order to have a good coverage of the cases, and good simulations to find bugs, it is very important to have a test plan.

The test plan defines all the different cases to test for a submodule or for the entire VPU. This means trying to find all the different corner cases stimulating the DUT. Whenever possible, a good test plan only includes different sets of stimulus, in this way an easy implementation is possible. But this situation does not cover all the possible cases. For instance, not every case could be tested in a submodule of the VPU only modifying the inputs from the scalar core. Which means that sometimes it is necessary to create some modified settings, to create the correct environment for the test.

It was created a test plan to stress the load operations. Those are affecting a lot of submodules of the VPU, but the focus will be on the Load Buffer and on

the LMU, mainly. Let's now analyze some of the created tests.

Test on consecutive elements

The first test planned was a simple test about consecutive elements, this means that all the elements are sent in order. Of course this is a simple case, and it is thought to test if all the chain to the effective load are working. In fact, it is a regression test.

The only constraint meant to be in this test is the sequentiality of the elements. An example is the one reported in Figure 2.9.

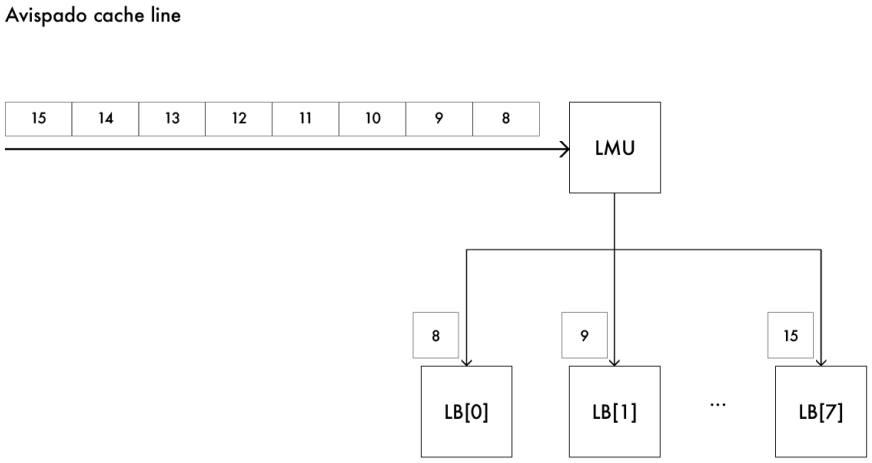

Figure 2.9. Sequential inputs from Avispado

Test on random values

The second kind of test to implement is the random test. This will stress the ability to use different elements ID and different loads at the same time. In this way all the handling for the positions, calculated based on the load and on the element ID, is stressed. It was created a special modality to constrain the randomness to the possible values. This was implemented in the UVM using the configurations for the sequence.

An example of random inputs from the same load is the one represented in Figure 2.10.

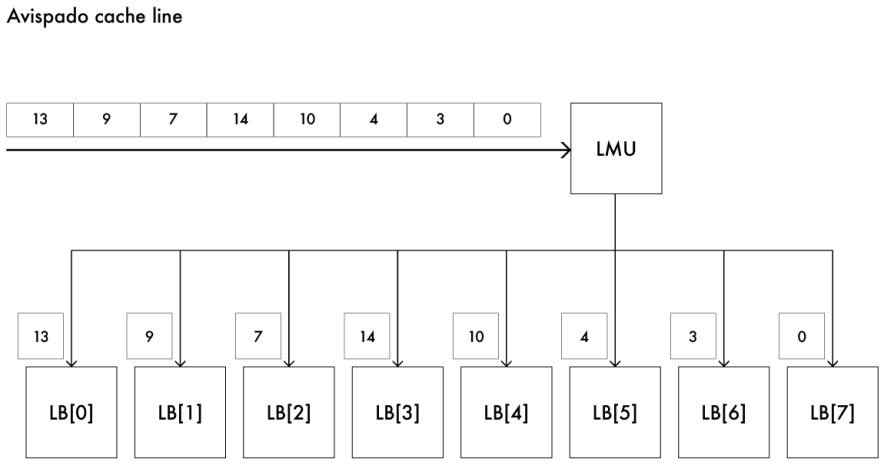

Figure 2.10. Random inputs from Avispado

It is also possible to have inputs from two loads, as reported in Figure 2.11.

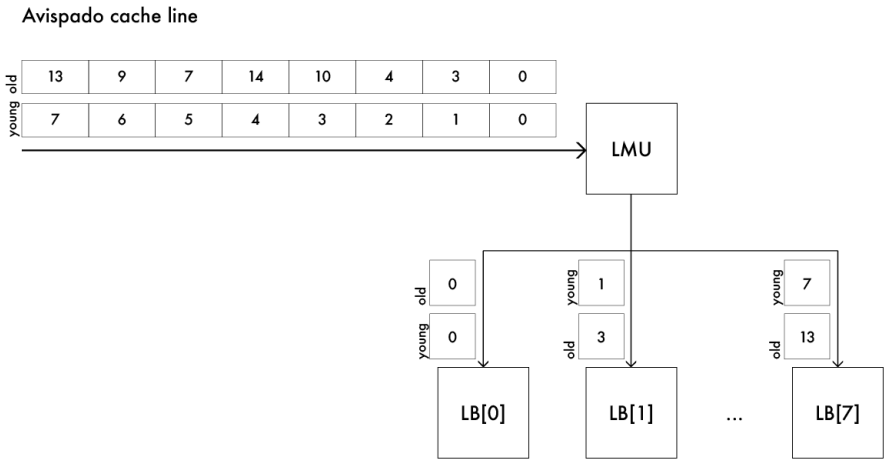

Figure 2.11. Random inputs from two loads

Test on splitted elements

A very interesting case is revealed when studying the positions. Let's take into account a line of elements sent by the LMU to the Load Buffer. In this case the SEW needs to be different from 64, so the case will use SEW = 32. The data sent will be sequential, starting from 65 to 80. In this way it is possible to test if the output is disposed correctly and so if the Load Buffer for the Lane[0] stores 80-65 as values, in this order.

The situation explained is the one reported in Figure 2.12. It does also show how the input is splitted in the LB.

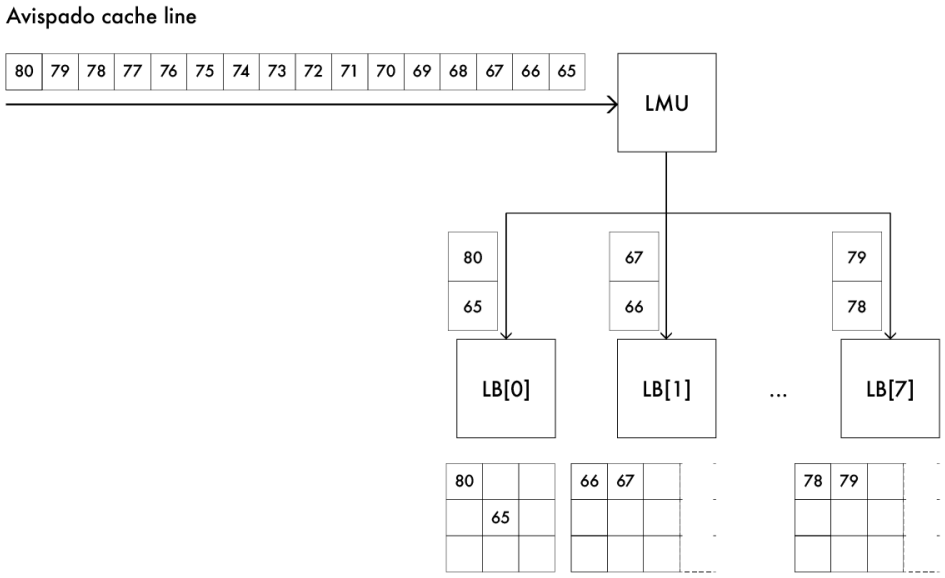

Figure 2.12. Input splitted into the LB

For reference let's analyze the binary used for this specific case.

```
#define __riscv_xlen 32
#include "test_macros.h"

.globl _start
.section .text

INIT_TEST(e32, 512)

la x1, init_region
addi x1, x1, 0x3c
vle.v v0, 0(x1)


END_TEST

# Initializes 4 registers (256 elements of 64 bit each one)
# TODO Do a macro that allocates N registers...
RVTEST_DATA (
        .dword 0x0000000200000001, 0x0000000400000003; \
        .dword 0x0000000600000005, 0x0000000800000007; \
        .dword 0x0000000A00000009, 0x0000000C0000000B; \
        .dword 0x0000000E0000000D, 0x000000100000000F; \
        .dword 0x0000001200000011, 0x0000001400000013; \
        .dword 0x0000001600000015, 0x0000001800000017; \
        .dword 0x...
        .dw...
```

The test is initialized with a macro, defining the SEW = 32 and the size in bit = 512. Then the data (the one that follows the instructions) is loaded into a register as address, and is summed an offset of *0x3c*, equal to 60 in decimal. This is the exact value needed to have the first sent element id = 65. As the number of elements will be 512/32 = 16 the element ids will go from 65 to 80.

Test on retries

The last test typology is about the retry mechanism. This occurs when all the three lines are filled with an element in the same position, and then a fourth elements arrives, this means that the Load Buffer does not have another position for one of the elements, thus one of the elements needs to go back to the sender and a new request is made.

Considering SEW = 64, there are different versions about this test: it can be a simple in-order retry with elements 0-40-80-120-... as inputs, it can be out-of-order as 0-40-120-80-..., or it can be with two different loads.

The easier example is represented in Figure 2.13. 0-40-80 are already filling the spots where 120 will try to fit. So a retry is needed, discarding 120.

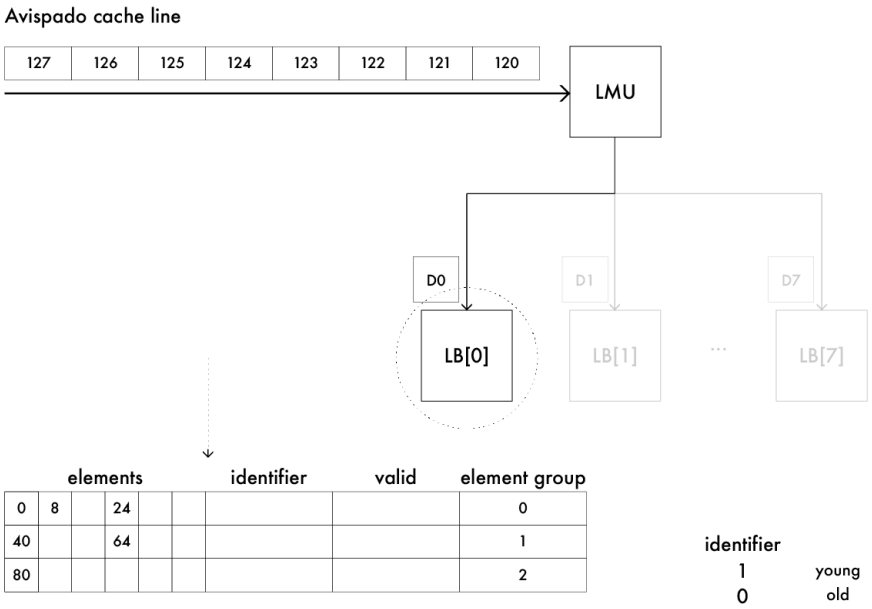

Figure 2.13. Stimulated retry

The out-of-order example is represented in Figure 2.14. 0-40-120 are already filling the spots where 80 will try to fit. So a retry is needed, discarding 120, and taking 80.

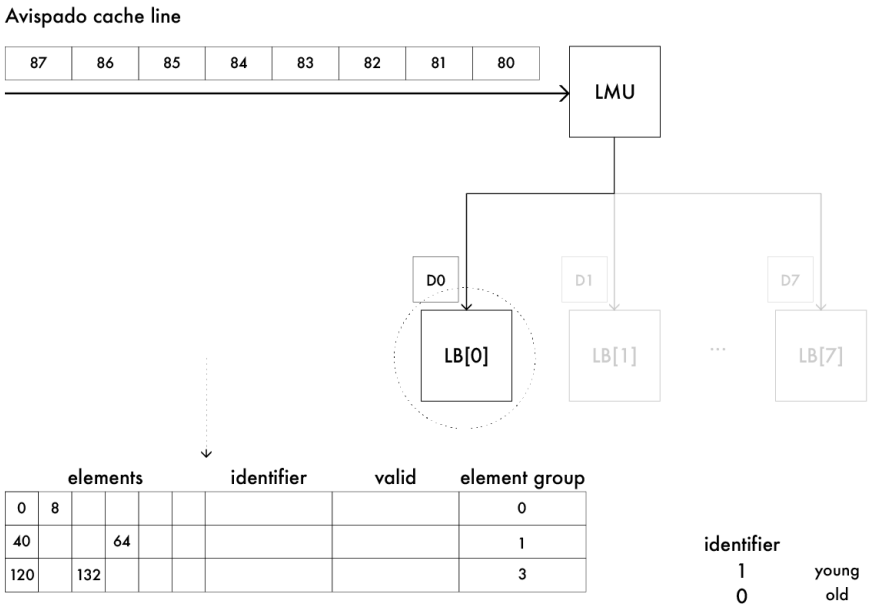

Figure 2.14. Stimulated retry with out-of-order values

2.3 Verification Process

The test executed are just the base for the verification. The tests can only define some functionalities, but a complete functional verification is required to test all of them. The normal approach starts with defining some functionalities and then comparing them to the design. This of course is the crucial part, because it is not always obvious how a specific case works. Also, there are different ways to test these functionalities.

2.3.1 Scoreboard

As seen into the test plan, an easy way to test the results is the scoreboard, thus to have a predicted value and then to compare it against the calculated one. Before the work of this thesis started, the UVM for the Avispado interface and a scoreboard for the VPU results were already implemented by the BSC

verification team. Let's briefly analyze how the scoreboard works and why it was useful to develop other tests.

Spike

Spike is a RISC-V ISA Simulator. For this specific project it was extended to support the vector extension. Spike has two functions, the first one is to simulate the Avispado core, producing stream of vector instructions and memory addresses. Those are feed to the VPU using the UVM. The second function is to produce a result to be compared with the one produced by the VPU. A simple scheme is represented in Figure 2.15.

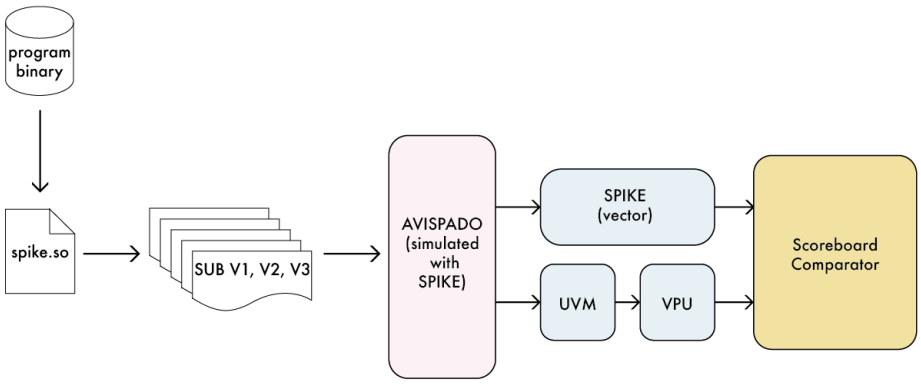

Figure 2.15. Simulation path

This kind of tool is very useful because it can give an idea of the general result and at the same time can give hints about the working condition of the VPU. Of course it is just a first step, because often it is hard to find a bug only using the comparison with the final result.

Load Management Unit's scoreboard

The aim of this thesis was principally based on testing the load operations, so a scoreboard for the LMU was created.

Recalling the LMU working model, the module first takes into account the stride, reversing the data in case of negative stride, then the data is positioned correctly again, then it is compacted considering the stride, and finally it is aligned to match with correct ID. In high level it was done using vectors, and a couple of steps were merged together. It is possible to see below the pseudo code representing the implementation in this way.

```
1        x=new[N_ELEMENTS][SEW];
2        y=new[N_ELEMENTS][SEW];
3
4        //first assignment
5        x[N_ELEMETS][SEW] = load_data_i[N_ELEEMTS*SEW]
6
7        //consider the stride
8        if(stride<0 && !is_indexed)
9            y[i] = x[-i];
10       else
11           y[i] = x[i];
12
13
14       //from # of bytes to # of elements
15       local_stride = mod(local_stride * 8 / SEW);
16
17       //concatenate the data according to stride
18       x[N_ELEMENTS-1-i] = y[(N_ELEMENTS-1-i*local_stride-OFFSET)%N_ELEMENTS];
19
20       //shift the data according to EL_ID
21       k = EL_ID % N_ELEMENTS;
22
23       y[(N_ELEMENTS-1-(k+i))%N_ELEMENTS] = x[(N_ELEMENTS-1-i)]
```

Also the mask was predicted in a similar way, thus the masking step was then implemented with a simple check on the mask to have the correct result. In this way it is possible to have an exact result for the LMU, so when a test fails, it is always possible to check if the LMU performed correctly with the data it received in input.

Load Buffer's scoreboard

For the Load Buffer the behaviour was really complicated, so a simplified version of a scoreboard was implemented. Hence, it does not work on the correct result operation-per-operation, because the LB has three layer of deepness and does always try to optimize the output. So, to implementing all the rules about the output would mean to create a similar structure with an high risk to create a bug in the verification model.

The idea was that the data observed as input would eventually become the

output data, placing in the correct position. This of course is not conflicting with the concept of retry. In fact, the retry mechanism only delays the data. It was implemented as an assertion to take advantage of the time correlation.

A pseudo version of this checker is represented below.

```
// Pseudo checker
if( lmu_dvalid_load_i) {

    load_data_o[get_el_bank(el_id_i, sew_i)*SEW + SEW-1 -: SEW] == data_i;
}


// This function computes the bank of an element
function [BANK_IDX_SZ-1:0] get_el_bank([EL_ID_WIDTH-1:0] el_id,
            [SEW_WIDTH-1:0] sew);

    case (sew[1:0])
        SEW64: get_el_bank=((el_id/N_LANES)%N_BANKS);
        SEW32: get_el_bank=(((el_id>>1)/N_LANES)%N_BANKS);
        SEW16: get_el_bank=(((el_id>>2)/N_LANES)%N_BANKS);
        SEW8:  get_el_bank=(((el_id>>3)/N_LANES)%N_BANKS);
    endcase
endfunction
```

Using the *element_id* and the corresponding bank (based on the SEW), it is possible to predict the result for a specific location and compare it with the calculated one. In this pseudo code the time handling is not very visible, but the *if* statement will wait until the result goes as output.

2.3.2 Checkers

In order to connect the checkers to the whole structure the binding method was used: basically, when a file containing the checker is created, it is compiled with a bind option to connect it to an RTL file. In this way the files can be kept separate and the Design Team and the Verification Team can work together on different files [10].

```
vlog $PATH_LB/LoadBuffer.sv -mfcu -cuname $PATH_LBC/LoadBuffer_checker.svh
```

In the code above the compiling line for the LoadBuffer RTL is reported, with the binding option recalling the checker, in order to compile it as well. Then, as the following code illustrates, the bind statement is written into the checker file. This statement will connect all the in/out ports between the DUT and the checker.

```
bind LoadBuffer LoadBuffer_checker bind_LoadBuffer_checker (.*);
```

Load Management Unit's assertions

The main focus was on the calculated result and on the handshake with the other components. All the functionalities to be checked were define in the verification plan, then a list of assertions was used to produce a good checker. The list of all the checkers can be found in Appendix.

name	functionality
a_el_count	when load_data_valid_i then $seq\_id[28:22](el\_count) \leq \frac{(n\_elements - el\_offset)}{(stride*8/sew)}$
a_stride _i	$stride \cdot 8$ can be one of the following $\pm(sew, 2 \cdot sew, 4 \cdot sew)$ with SEW = $2^{(3+sew)}$ and not indexed load
a_load_granted_i_known	when load_granted_i, not unknown the following : SEW, granted_sb_id
a_load_data_i_known	when data_valid_i, not unknown seq_id_i, load_data_i
a_sb_id_o	when load_data_valid_i , next clock cycle sb_id_o(t) = seq_id_i[29:32](t-1)
a_load_dvalid_o_known	when load_dvalid_o, outputs known
a_dvalid_o	after load_data_valid_i, next clock cycle, load_dvalid_o
a_vstart_o	correct v_start_next_o v_start_self_o, according to load_data_o
a_granted_ids	can not be granted 2 loads with the same id at once
a_indexed_instr	when the load is an indexed load, el_count=1 and mask_valid_i = 0
a_load_sync_end_i	for every load_granted_i there is eventually the corresponding load_sync_end_i
a_load_grant_beh	when num_load_inflight = 2, load_grant_i = 0
a_load_sync_end_beh	when num_load_inflight = 0, load_sync_end_i = 0
a_load_req_sync_end	load_sync_end_i cannot be simultanous with load_granted_i
a_el_count_sew_i	when $stride\_i \neq (1, 2, 4) \cdot sew\_i$ (in bytes), el_count has to be = 1
a_sb_correct	when load_dvalid_i, if it is given a sb_id not present in the fifo, load_dvalid_o will be 0
a_data_o	correct load_data_o, clk cycle after load_dvalid_i (stride/mask/indx)
a_mask_o	correct mask_o, according to load_data_o
a_el_ids_o	correct element_ids_o
a_min_el_id_idx_o	correct min_el_id_idx_o
a_rsn_o	when rsn_i load_dvalid_o = 0 and mask_o = '0
a_rsn_i	when rsn_i load_data_valid_i = 0 and load_granted_i = 0

Table 2.3. Assertions on LMU

As it is possible to see in Table 2.3, there are different assertions possible for a single submodule. Here they are separated by colour distinguishing the different "categories". Indeed, the red ones are checking the basic signals behaviour and the handshakes. An example is **a_dvalid_o**:

```
property p_dvalid_o;
        @(posedge clk_i)
        load_data_valid_i && sb_correct |-> ##1 load_dvalid_o ;
endproperty : p_dvalid_o



a_dvalid_o : assert property( disable iff(!rsn_i || kill_i) p_dvalid_o )
        else $error('LMU did not compute any output');
```

- **load_data_valid_i** is the input necessary to fire the assertion;
- **sb_correct** is the result of a task, that calculates if the *scoreboard_id* is a valid one;
- | - > is the separation between the firing condition and the checking one. It is not time consuming (the time consuming version would be | = >), so the check for the second condition starts immediately;
- **##1** command refers to 1 *clock cycle(s)* delay;
- **load_dvalid_o** asserted is the expected output.

The light blue ones are scoreboard-like assertions, indeed are checking the scoreboard_id, the data in output and also the masks. They use the results of the tasks to check the values, it is possible to see an example in **a_el_ids_o**where it shown how an assertion uses the result of a task.

```
1  //task
2  task compute_ids(int unsigned SEW, int unsigned EL_COUNT,
3              int unsigned EL_ID, int unsigned N_ELEMENTS);
4
5      int unsigned f_v_e, i, j;
6
7      // f_v_e is the index of the first 11-bit-element
8      // of computed_el_id_o where we have to put an EL_ID
9          f_v_e = EL_ID % N_ELEMENTS; // find the first valid element
10         f_v_e = f_v_e * SEW / 8 ; // multiply it by the number of bytes per el
11
12         // initialize to 0
13         for(j=0;j<MAX_NUMBER_ELEMENTS;j++) begin
14             computed_el_id_o[j] = '0;
15         end
16
17         for(j=0;j<EL_COUNT;j++) begin
18                 for(i=0;i<SEW/8;i++) begin
19                         computed_el_id_o[(f_v_e+i+(j*SEW/8))%MAX_NUMBER_ELEMENTS]
20                             = EL_ID+j;
21                 end
22         end
23
24 endtask : compute_ids
25
26 // property
27 property p_el_ids_o;
28         bit [MAX_NUMBER_ELEMENTS-1:0][EL_ID_WIDTH-1:0] buffer_ids;
29         @(posedge clk_i)
30         (load_data_valid_i  && sb_correct, buffer_ids = computed_el_id_o)|->
31         ##1 element_ids_o == buffer_ids;
32 endproperty : p_el_ids_o
33
34 //assertion
35 a_el_ids_o : assert property( disable iff(!rsn_i || kill_i) p_el_ids_o )
36         else $error('mismatch in element_ids_o');
```

It is possible to see that the property uses the value *computed_el_id_o*, a global variable. It is important to say that the task *compute_ids* is not called by the property but from another external process not displayed here. Finally, the yellow ones are just checking the conditions for the reset and the expected output.

It is also important to notice that some of those *are not assertions*, but *assumptions*. This means that they are conditions on the input and not on the outputs. There is no difference for the Functional Verification, unlike with Formal Verification.

Load Buffer's assertions

A similar approach was adopted for the Load Buffer. Having a lot of interfaces the LB has also a lot of handshakes, and so assertions. For this reason, in the Table 2.4 are only reported some categories containing all the assertions for an interface.

name	functionality
AVISPADO IF	this is the interface with the scalar core
a_mem_sync_end_i	if mem_sync_end_i is high, memop_sb_id_i must be a known value
a_valid_after_sync_end	for every memop_sync_end_i (with a valid memop_sb_id_i) there will be a load_valid_o (with correct load_sb_id_o)
a_memop_sync_end_num	if num of memop_sync_end_i = num of load_ack_i +2, then only memop_sync_end with a sb_id different from the ones stored
a_load_ack_i_num	if the num of load_ack_i is in excess, we assume this ack is for a different operation
a_unique_request	there can not be a request with the same sb_id of an inflight load
LMU INTERFACE	from this interface the LB receive the cache lines
MQ INTERFACE	from this interface the load are requested/granted
FSM INTERFACE	from this interface comes the enable to write on the VRF
CU INTERFACE	to this interface the ids for and ended load are sent
VRF INTERFACE	this is the interface to load the data
a_load_inflight	load_inflight_o is checked, it can be 1 when the number of load inflight is 1 or 2 and it can be 0 otherwise
a_load_ready_o	when load_inflight_o = 1 and lmu_dvalid_load_i = 1, if there is fsm_read_en_lbuf_i, the next clock cycle there is load_ready_o = 1
a_data_corruption	check that each element that enter will eventually exit in correct position
VL INTERFACE	from this interface the clk and the rst are recived

Table 2.4. Assertions on LB

A good example for the handshake is *a_mem_sync_end_i*. This is an assumption on the inputs, but will be treated as an assertion.

```
property p_mem_sync_end_i;
        @(posedge clk_i)
        memop_sync_end_i |-> !$isunknown(memop_sb_id_i);
endproperty : p_mem_sync_end_i


a_mem_sync_end_i: assume property(disable iff(!rsn_i||kill_i) p_mem_sync_end_i)
        else $error('unknown memop sd id on memop_sync_end_i asserted');
```

The property is using the system function ***$isunknown***, this function returns if the value passed is unknown ('*X*' or '*Z*'). So basically when *memop_sync_end_i* is asserted, the value of its scoreboard id must be known.

Then *a_data_corruption* is the scoreboard like discussed before, and finally there are the reset's ones. The assertions are reported into the Appendix.

2.3.3 Drivers

The other tool used to test the submodules is the Driver. It is a file that defines the stimuli and their order. A very specific order of signals defines an operation. It is then possible to create the situation in which the DUT will work as expected. It is important to notice that the driver is enabled only when the UVM is configured as ACTIVE.

Load Management Unit

This is the only submodule on which was developed a driver, due to the reduced complexity. In fact, the driver can be very complex when different handshakes are present.

In the next page it is possible to see one of the operations implemented into the driver, as example.

```
if(command.op == gnt_op) begin
    // randomizations
    SEW = command.sew_i;
    SEW = (2**(3+SEW))/8; // in number of bytes , for the stride in bytes
    randomize(command.stride_i)
        with {command.stride_i inside{1*SEW,2*SEW,4*SEW,-1*SEW,-2*SEW,-4*SEW,
        command.n*SEW};};

    randomize(command.load_granted_sb_id_i)
        with{!(command.load_granted_sb_id_i inside
        {fifo[0].sb_id, fifo[1].sb_id});};

    // check how many inflight load
    if ( !fifo[0].done && !fifo[1].done ) begin
            @(posedge lmu_if.clk_i) ;
            lmu_if.load_data_valid_i = 0;
            lmu_if.load_sync_end_i=1;
            lmu_if.load_sync_end_sb_id_i=fifo[command.which_load].sb_id;
            fifo[command.which_load].done = 1;
    end

    // driving
    @(posedge lmu_if.clk_i);
    lmu_if.load_sync_end_i=0;
    lmu_if.load_sync_end_sb_id_i='0;
    @(posedge lmu_if.clk_i);
    lmu_if.load_sync_end_i=0;
    lmu_if.load_sync_end_sb_id_i='0;
    lmu_if.op = gnt_op;
    lmu_if.rsn_i = 1'b1;
    lmu_if.kill_i = 1'b0;
    lmu_if.load_granted_i = 1;
    lmu_if.indexed_load_granted_i = command.indexed_load_granted_i;
    lmu_if.load_granted_sb_id_i= command.load_granted_sb_id_i;
    lmu_if.load_data_valid_i = 0;
    lmu_if.load_data_i = '0;
    lmu_if.seq_id_i = '0;
    lmu_if.mask_valid_i = 1'b0;
    lmu_if.mask_i = '0;
    lmu_if.sew_i = command.sew_i;
    lmu_if.stride_i = command.stride_i;
    // update the fifo
    if ( fifo[1].done ) begin
            fifo[1].SEW = command.sew_i;
            fifo[1].STRIDE = command.stride_i;
            fifo[1].sb_id = command.load_granted_sb_id_i ;
            fifo[1].is_indexed = command.indexed_load_granted_i;
            fifo[1].done = 0;
    end
    else if( fifo[0].done ) begin
            fifo[0].SEW = command.sew_i;
            fifo[0].STRIDE = command.stride_i;
            fifo[0].sb_id = command.load_granted_sb_id_i;
            fifo[0].is_indexed = command.indexed_load_granted_i;
            fifo[0].done = 0;
    end
    @(posedge lmu_if.clk_i);
    lmu_if.load_granted_i = 0;
    -> lmu_if.new_input;
end
```

The operation implemented is the granting of a load request. The code is mainly divided in three parts:

- the *first* one makes some randomization for the data while it checks if the driver is following the assumptions about the inputs (in this case on the loads inflight);
- the *second* part is the driving part, all the inputs are well defined, and also some clock cycles need to be waited sometimes. All the value assigned with ***command*** have been randomized, but constrained to valid values;
- the *third* part is just the updating of an internal FIFO, useful to be in sync with the LMU and to not fail the assumptions.

Let's now move on to the results of the functional and formal analysis.

Chapter 3

Results and Considerations

In this chapter the results and the material created during all the work of this thesis are discussed. It is important to highlight that the results are not the only parameter defining whether a good verification work was done or not. In fact, building the infrastructure and create documentation are important tasks that every verification engineer will eventually do. Even if these tasks are not always directly correlated to some found bug, they will produce results in long term.

3.1 Found Bugs

The thesis was developed while the design and the specifications were still in progress and so unstable. To synchronize all the work it was used the GitLab structure, creating issues and discussing the problems. A well defined issue was then assigned to a design engineer to fix it.

The majority of the bugs were found on the Load Management Unit. Let's now analyze some of them.

Indexed load bug

This issue was found using the Formal tools and trying to define better specifications for the Load Management Unit.

The problem was the following: The LMU used the *sequence_id*, in particular the *el_count* field, to identify if the load was strided or indexed. In particular, *el_count* contains the number of valid elements being loaded; when this value

was *el_count = 1* the load was considered as indexed. The issue was occurring because the indexed load should have ignored the stride value, but this was not the case. In fact, the load would not have ignored the stride, as every operation was actually considered as a strided one.

The best solution for this problem was to create a new signal, *is_indexed*, to identify an indexed load. The stride will be always ignored when this signal is asserted, and the value of *el_count* must be equal to 1.

Load id bug

This issue regards the load ids. When a load was finished, the *load_sync_end_i* signal was sent to the LMU to notify the operation was concluded. It was sent without an identifier to understand which of the two possible loads was finished, so a FIFO was used to end them in order. This was an issue because the loads should have the possibility to end out-of-order, so a new signal was created: *load_sync_end_sb_id_i*. This signal is the id that identifies the load to be finished. This issue was found analyzing the specifications to create the Test Plan.

Out-of-order load bug

This issue regards the out-of-order loads. In the Load Management Unit there is a FIFO handling the load ids for the two possible inflight loads. The FIFO was freed of an element only when another one was put as input. But it could have been happened that an already inflight load was into the FIFO, i.e. with *id = 1*, and another load with the same id, issued. Having the same id, the configurations for the second load would have been overwritten.

This issue was found analyzing the result of two consecutive loads, with same id, but one was strided and another one was indexed. The second load was not considered as indexed, failing the condition that says that *el_count = 1*. This was the case of a typical bug finding flow: first, a big test fails during a load. Then, it is examined using the assertions on the load modules. Exploring the waveform is then possible to find the bug.

After kill bug

This bug is related to the validity of the output of the LMU. When there is a kill, the entire instruction needs to be killed. But, it could be possible to receive valid data for a killed load. This happens because the kill mechanism needs some time to stop the operation. The LMU should ignore the input from a killed instruction, but it did not happen in this case. Indeed, the validity of the data in output was assumed every time the data was valid in input. This caused the simulation to timeout on the next instruction. This was happening because the kill was not issued correctly, so the starting point for the next instruction was not clean.

This issue can represent a valid example of why the formality on the assertions in very important. Consider the following code:

```
property p_dvalid_o;
@(posedge clk_i)
load_data_valid_i && sb_correct |-> ##1 load_dvalid_o ;
        endproperty : p_dvalid_o

a_dvalid_o : assert property( disable iff(!rsn_i || kill_i) p_dvalid_o )
else $error('LMU did not compute any output');
```

In this case, the assertion is testing if there is *load_dvalid_o* when the input is valid. But it does not test if *load_dvalid_o* has always a valid *load_data_valid_i* a clock cycle before. The property as reported is not able to spot the error in this issue, but knowing where the problem is, it is possible to use the correctness of this assertion to understand the bug.

Two loads bug

This issue was spotted both in the LMU and in the LB. The modules received two loads with the same id. This led to an assumption error, as every load has an unique id. In reality the problem was not in the Load Management Unit nor in the Load Buffer, but in the memory queue. In fact, this modules is the one supposed to hand the ids for each load. This issue was found very late in the verification process because was depending on a trigger enabled by the kill.

For reference in Figure 3.1 there is the interface of a simulation, in particular on the waveforms. It is possible to see two loads issue with the same id.

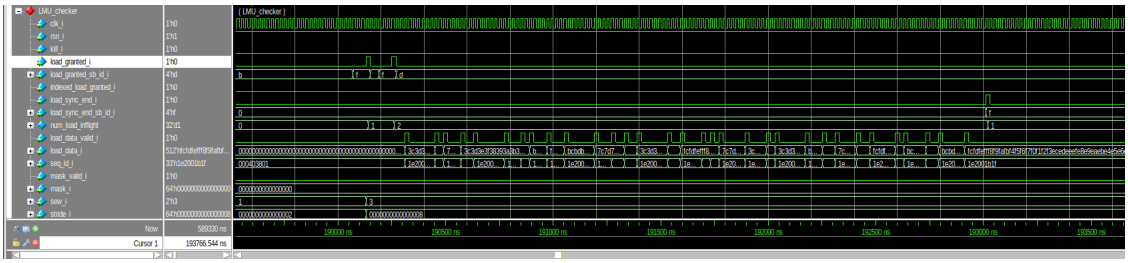

Figure 3.1. Two loads issued with the same id

3.2 Material Created

Beside than the found bugs, during the thesis, some material has been created. In particular regarding the structure to handle the test of the submodules.

UVM

A complete UVM was created, to drive and test the Load Management Unit. It may have different tests with different sequences and so different constraints.

```
class lmu_constrained_test extends lmu_random_test;
    `uvm_component_utils(load_management_unit_constrained_test)

        function void build_phase(uvm_phase phase);
                lmu_sequence_item::type_id::
                  set_type_override(lmu_constrained_sequence_item::get_type());
                super.build_phase(phase);
        endfunction

        function new(string name, uvm_component parent);
                super.new(name,parent);
        endfunction : new


endclass : lmu_constrained_test
```

In this code the base_test is extended and then it is overridden into the build_phase the sequence_item.

```
class lmu_constrained_sequence_item extends lmu_sequence_item;
        `uvm_object_utils(lmu_constrained_sequence_item)

        constraint few_rst{op dist{rsn_op:=1,gnt_op:=5,load_op:=9,kill_op:=2};}

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

endclass : lmu_constrained_sequence_item
```

In the sequence new constraints are applied. In this case the probability for the operation are defined. This kind of structure is very reusable and expandable, furthermore, during the whole verification process it can be improved.

Specifications

Some documentation was created on missing specifications. This work required a looped check on the behaviour obtained by the RTL design. The specifications and the design were still being developed, this does also mean that the documentation should have been updated. But this was not always the case, so a constant check (helped by the formal tool) helped in maintaining the documentation correct.

Verification plan

Some parts of the verification plan were already created when the work of this thesis started. However, a lot of changes were upcoming when starting the creation of the checkers, so important modifications were done to the verification plan. The assertion reported into the Appendix are following the verification plan implemented.

Test plan

The test plan for the loads was entirely created during the work of this thesis. First, it required some confidence with the load operation, then it was possible to update the simple cases and fill it with interesting corner cases. In order to give validity to the test plan it was important to create an easy way to run those tests. For that reason some configurable modalities were set.

Modalities

The last contribution was to create the modalities for the implemented tests. These modalities are the way to stimulate the VPU modifying some settings into the UVM. This allows to order the data in specific ways or to force some mechanism inside the VPU. In the code below it is possible to see a couple of them.

```
//(for elegance there is also a subtraction in case of overflow,
//in order to start from the smallest index possible).
//The range_id is NOT getting smaller as the overflow occurs,
//because we do not want to cause more retries, so we put the index fixed to 0

else if (m_cfg.seq_id_mode == RETRY_ID) begin
    if(loop_ooo[load_index] <= 3 & loop_ooo[load_index] > 0) begin
        range_id[load_index] = c_lines_per_group -1 ;
        seq_id_index[load_index] = (seq_id_index[load_index] +
            range_id[load_index])%(inflight_loads[load_index].seq_ids.size());

    end
    else seq_id_index[load_index] = 0;
        loop_ooo[load_index] += 1;

end


//Here we want to stimulate 0-40-120-80 id seq,
//so we choose the correct cycle value and manipulated the range.
//then the cycle after the range returns to be 4 and stale.

else if (m_cfg.seq_id_mode == RETRY_OOO_ID) begin
    if(loop_ooo[load_index] == 0) begin
        range_ooo_id[load_index] = c_lines_per_group - 1;
        seq_id_index[load_index] = 0;

    end else begin
        if(loop_ooo[load_index] == 3)
            range_ooo_id[load_index] = 2*range_ooo_id[load_index] + 1;
        if(loop_ooo[load_index] == 4)
            range_ooo_id[load_index] = -(c_lines_per_group);
        if(loop_ooo[load_index] == 5)
            range_ooo_id[load_index] = c_lines_per_group -1;
        seq_id_index[load_index] = (seq_id_index[load_index] +
            range_ooo_id[load_index])%(inflight_loads[load_index].seq_ids.size());
        if(loop_ooo[load_index] > 5)
            seq_id_index[load_index] = 0;
    end
    loop_ooo[load_index] += 1;
end
```

It is possible to see the handling of the element id sent by Avispado. In this case the order of the ids is manipulated, however, this will not cause an error as the final result will be the same. In this way it is possible to stimulate the retry mechanism sending all the elements for the same position into the Load Buffer. There is also another version for the out-of-order retry. Those modalities can be configured for each test, and so randomized. In this way, they can be implemented in the automatic tests.

3.3 Considerations

During the first couple of months of work, we have better defined the specifications and developed the tools to test the design. The results were still poor, because the verification effort is slow in the first phases, when the tools are not stable nor complete. A nightly testing method was already being created by the BSC Verification Team, using the servers to run up to 50 long simulations per night. The number of instructions was variable and depended on the complexity. When the tools, such as assertions and assumptions, were ready, they were implemented into the night run system. This allowed to ensure the assumptions validity and to test the design against the assertions. Some results were coming up, but it was still not enough. The assertions and assumptions were tested using formal tools. A scoreboard was implemented for the LMU and scoreboard-like assertions were implemented for the LB. This helped a lot, because a general result for the submodule was then available. This meant that it was easier to discriminate whether the problem was in the logic part or into the interface one.

A relevant part of the bugs found during the period of this thesis was discovered using directly the checkers. The 40% of the total load-related problems were spotted with the contribution of this work. Almost the 80% of those errors were found using assertions, the rest of them studying and formally testing the specifications. Considering only the Load Management Unit, up to the 56% of the errors were spotted using the UVM structure, the assertions and the specification review. About 45% of the bugs were found only using checkers. The expected value was about the 35% [11]. In this case the number was higher because of the scoreboard implemented as an assertion.

In Figure 3.2 it is represented the distribution of the errors found by the Verification Team, the Design Team and the personal contribution.

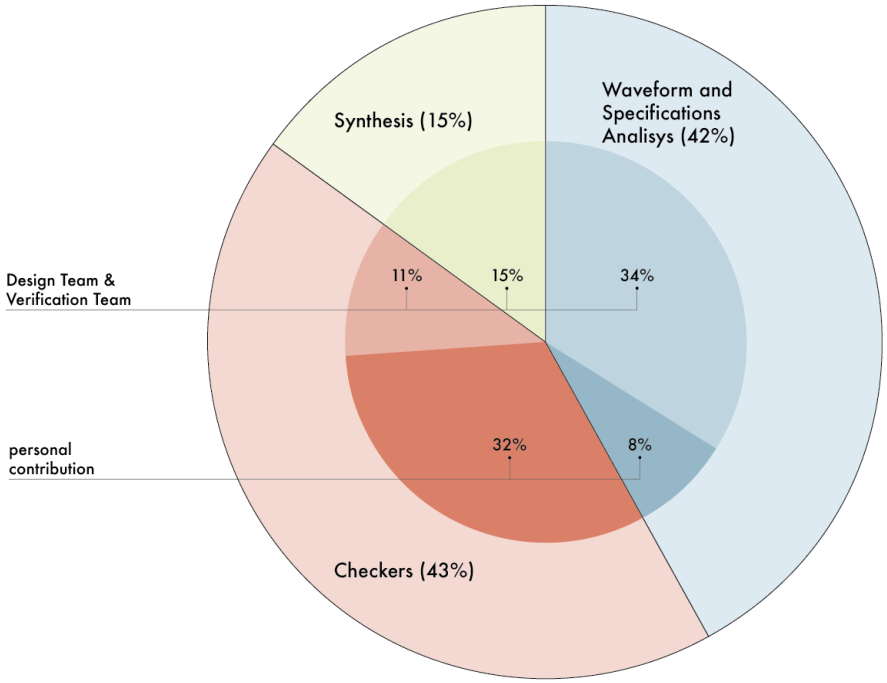

Figure 3.2. Found bugs

All the bugs found can be grouped in three categories: *Synthesis*, these are problems related to unconnected wires uninstantiated logic; *Waveform and specifications analysis*, these are all the bugs found simply analyzing the waveform or the specifications, tracing back the error, starting from a wrong result of the VPU; the *Checkers*, in this category there are all the bugs directly found using assertions and assumptions.

As displayed, not all the "checkers" errors were found using the tools created in this work, because some of them were spotted using assertions on the interfaces developed by the Verification Team.

Chapter 4

Conclusions

4.1 Conclusions

This thesis has shown all the efforts to verify the behaviour of some DUTs, in particular for the load operations.

Analyzing the results obtained with the Load Management Unit, it is possible to understand why it is crucial to have a good Verification Plan. It was used to test all the specifications and in this way to test their robustness. A detailed Verification Plan enables to find some bugs in many different cases and, moreover, it was helping defining an high level model, in order to eventually create a scoreboard. In particular when the design is simple the usage of the scoreboard becomes very useful. In this thesis the scoreboard was created for the LMU but not for the LB, otherwise it would have required a verification process on the scoreboard itself. Scoreboard-like assertions were then created to test the results. However, these assertions are not testing the difficult cases, so it is very hard for them to find an hidden bug. But it is still possible to improve them, creating a complete scoreboard. In fact, all the code created, was created following the philosophy of re-usability and modularity. When performing some long random tests, some blind spots were still present, because the randomness needs a lot of time to reach all the corner cases. That is the reason why a Test Plan was necessary, as well as its capability to define the design behaviour in difficult cases.

The Formal tool has been used with the sole purpose of assuring the assumptions and the assertions were formulate correctly. Even though it was too early to apply different usage of this tool in this thesis, it is important to mention that there is the chance to use it for other aims, such as an optimization of the model using mathematical equivalence.

As a final consideration an estimation on the bugs found was done: *~40% of the load related problems were spotted with the work done in this thesis.* In particular, *up to the 55% of the errors on the Load Management Unit were spotted using the UVM structure, the assertions and the specification review.* This confirms the importance of advanced techniques for the verification process. Moreover, all the tools created are still in use in the current development of the EPI project. This will allow further discoveries on bugs and missing specifications.

4.2 Future developments

Regarding the verification process, there is still some space for future developments. In particular the next steps would be to complete the Verification Plan and add assertions and assumptions for all the submodules. It would also be useful to have a scoreboard for each submodule and to complete the one for the Load Buffer. Formal verification could also be used to test all the VPU and then optimize the design in order to have less logic and so less errors. Finally, it would be more productive to implement more coverage controls. This will allow to better guide the process and to avoid blind spots.

Considering the project in its entirety, it is now important to catch up with the specifications that are being updated by RISC-V. In this way it will be possible to provide an innovative VPU.

List of Figures

List of Tables

Bibliography

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html.

[2] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[3] K. Asanović, "Lecture on vectors l15," Feb 2020. https://inst.eecs.berkeley.edu/~cs152/sp20/.

[4] C. Lemuet, J. Sampson, J. Francois, and N. Jouppi, "The potential energy efficiency of vector acceleration," in *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, IEEE, 2006.

[5] R.-V. Foundation, "Risc-v "v" vector extension specifications," Tech. Rep. 0.7.1-20190610-Workshop-Release, RISC-V Foundation, 2019. https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1.

[6] A. B. Mehta, *ASIC/SoC functional design verification*. Springer, 2018.

[7] A. B. Mehta, *SystemVerilog Assertions and Functional Coverage*. Springer, 2016.

[8] R. Drechsler, *Formal System Verifcation*. Springer, 2018.

[9] SemiDynamics, "Avispado-vpu interface specification," tech. rep., SemiDynamics, 2020. https://github.com/semidynamics/

`OpenVectorInterface`.

[10] D. Smith, "Using bind for class-based testbench reuse with mixed-language designs," *Doulos*, 2009.

[11] M. Kantrowitz and L. M. Noack, "Verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor," *Digital Equipment Corporation*, 2013.

Appendix A

Appendix

A.1 Checkers

A.1.1 Load Buffer

```
1  //`define FORMAL // please comment this when performing the Functional
       Verification
2
3  import EPI_pkg::*;
4
5  localparam TADDR_WIDTH_LOC = 8;
6  localparam BANK_IDX_SZ = (N_BANKS == 1)?1:$clog2(N_BANKS);
7  localparam MAX_LB_DELAY = $;
8
9  bind LoadBuffer LoadBuffer_checker bind_LoadBuffer_checker (.*);
10
11
12
13 module LoadBuffer_checker
14         (
15         input                                   clk_i,
16         input                                   rsn_i,
17         input                                   load_req_i,
18         input [SB_WIDTH-1:0]                    load_sb_id_i, // IT IS
               POSSIBLE THAT WE HAVE TO MANAGE TWO LOADS IN PARALLEL
19         input                                   load_masked_i,
20         input [SEW_WIDTH-2:0]                   load_sew_i,
21         input                                   lmu_dvalid_load_i,
22         input [DATA_PATH_WIDTH-1:0]             lmu_data_i,
23         input [MASK_BANK_WORD-1:0]              lmu_enable_i,
24         input [SB_WIDTH-1:0]                    lmu_sb_id_i,       //
               sb_id corresponding to the data received in lmu_dvalid_load_i
```

```
25          input [$clog2(MASK_BANK_WORD)-1:0]              lmu_min_el_id_i,
                        // element_id corresponding to the data received in
                lmu_dvalid_load_i
26          input [MASK_BANK_WORD-1:0][EL_ID_WIDTH-1:0]     lmu_el_id_i,          //
                 element_id corresponding to the data received in lmu_dvalid_load_i
27          input                                           fsm_read_en_lbuf_i,
                // this signal comes from main FSM WB state (in order to be able to
                 read data from the buffers in MEM state)
28          input                                           fsm_mem_write_en_i,   //
                 this signal indicates that we're in FSM MEM state so we can write
                in the VRF
29          input [VADDR_WIDTH-1:0]                         load_addr_i,
30          input [1:0][TADDR_WIDTH_LOC-1:0]                load_addr_trans_i,
31          input                                           memop_sync_end_i,
32          input [SB_WIDTH-1:0]                            memop_sb_id_i, // sb_id
                 of the memop_sync_end, identifies which load operation is
                finishing!
33          input                                           load_write_en_rf_o,   //
                 write enable to the register file
34          input                                           load_gnt_o,
35          input                                           load_valid_o,
                            // indicates that the load operation was completed
                successfully
36          input [SB_WIDTH-1:0]                            load_sb_id_o,
37          input [VADDR_WIDTH-1:0]                         load_valid_addr_o,
38          input                                           load_masked_o,
39          input                                           load_ready_o,
40          input reg [N_BANKS*DATA_PATH_WIDTH-1:0]         load_data_o,
41          input reg [N_BANKS-1:0][MASK_BANK_WORD-1:0]     load_data_en_o, // TODO
                 TBD the final interface
42          input [VADDR_WIDTH-1:0]                         load_addr_o,
43          input                                           load_inflight_o,
44          input [1:0][TADDR_WIDTH_LOC-1:0]                load_addr_trans_o,
45          input                                           kill_i,
46          input                                           load_ack_i,
47          input [SB_WIDTH-1:0]                            load_ack_sb_id_i,
48          input [CSR_VLEN_START-1:0]                      vstart_self_i,
49          input [CSR_VLEN_START-1:0]                      vstart_next_i,
50          input reg [CSR_VLEN_START-1:0]                  vstart_self_o,
51          input reg [CSR_VLEN_START-1:0]                  vstart_next_o,
52          input reg                                       retry_self_o,
53          input reg                                       retry_next_o,
54          input reg [SB_WIDTH-1:0]                        retry_sb_id_o
55          );
56
57  ////////////////////////////////////////////////////
58  ////////////////////// variables ///////////////////////
59  ////////////////////////////////////////////////////
60  typedef struct {
61          bit [SB_WIDTH-1:0] sb_id;
62          bit done;
```

```
63          bit [SEW_WIDTH-2:0] sew;
64          bit asked_for_retry;
65          bit now_granted;
66  } granted_load;
67
68
69  granted_load fifo[1:0];
70  int num_gnt;
71  int num_load_completed;
72  int num_ended;
73  bit [SB_WIDTH-1:0] sb_id0;
74  bit [SB_WIDTH-1:0] sb_id1;
75  bit ng_0;
76  bit ng_1;
77  bit done0;
78  bit done1;
79
80
81  ////////////////////////////////////////////////////
82  ////////////////////processes///////////////////////
83  ////////////////////////////////////////////////////
84
85  always_comb begin
86          assign ng_0 = fifo[0].now_granted;
87          assign ng_1 = fifo[1].now_granted;
88          assign sb_id0 = fifo[0].sb_id;
89          assign sb_id1 = fifo[1].sb_id;
90          assign done0 = fifo[0].done;
91          assign done1 = fifo[1].done;
92  end
93
94  initial begin
95          fifo[0].done = 1;
96          fifo[0].sb_id = 20; // we have to initialize it to a value that cannot
                be in input, to avoid to read the wrong SEW and STRIDE from the
                wrong fifo cell
97          fifo[0].sew = 0;
98          fifo[0].asked_for_retry = 0;
99          fifo[0].now_granted = 0;
100         fifo[1].done = 1;
101         fifo[1].sb_id = 20;
102         fifo[1].sew = 0;
103         fifo[1].asked_for_retry = 0;
104         fifo[1].now_granted = 0;
105 end
106
107
108
109 //for each load_gnt_o there is a fell(load_req_i)
110 always @(negedge clk_i or negedge rsn_i) begin
111
```

```
112         if ( !rsn_i || kill_i) begin
113                 num_gnt = 0;
114                 num_load_completed = 0;
115                 num_ended = 0;
116                 fifo[0].done = 1;
117                 fifo[0].sb_id = 20;
118                 fifo[0].sew = 0;
119                 fifo[0].asked_for_retry = 0;
120                 fifo[0].now_granted = 0;
121                 fifo[1].done = 1;
122                 fifo[1].sb_id = 20;
123                 fifo[1].sew = 0;
124                 fifo[1].asked_for_retry = 0;
125                 fifo[1].now_granted = 0;
126         end
127         else    begin
128                 fifo[0].now_granted = 0;
129                 fifo[1].now_granted = 0;
130                 if ( load_gnt_o ) begin
131                         if ( fifo[0].done ) begin
132                                 fifo[0].sb_id = load_sb_id_i;
133                                 fifo[0].done = 0;
134                                 fifo[0].sew = load_sew_i;
135                                 fifo[0].asked_for_retry = 0;
136                                 fifo[0].now_granted = 1;
137                         end
138                         else if ( fifo[1].done ) begin
139                                 fifo[1].sb_id = load_sb_id_i;
140                                 fifo[1].done = 0;
141                                 fifo[1].sew = load_sew_i;
142                                 fifo[1].asked_for_retry = 0;
143                                 fifo[1].now_granted = 1;
144                         end
145                 end
146                 if ( load_ack_i ) begin
147                         if ( (fifo[0].sb_id == load_ack_sb_id_i) && !fifo[0].
                                done ) begin
148                                 fifo[0].done = 1;
149                                 num_gnt = num_gnt - 1 ;
150                                 num_load_completed = num_load_completed - 1 ;
151                                 num_ended = num_ended - 1 ;
152                         end
153                         else if ( (fifo[1].sb_id == load_ack_sb_id_i) && !fifo[
                                1].done ) begin
154                                 fifo[1].done = 1;
155                                 num_gnt = num_gnt - 1 ;
156                                 num_load_completed = num_load_completed - 1 ;
157                                 num_ended = num_ended - 1 ;
158                         end
159                 end
160                 if ( retry_self_o || retry_next_o ) begin
```

```
161                         if ( (fifo[0].sb_id == retry_sb_id_o) && !fifo[0].done
                                ) begin
162                             fifo[0].asked_for_retry = 1;
163                         end
164                         else if ( (fifo[1].sb_id == retry_sb_id_o) && !fifo[1].
                                done ) begin
165                             fifo[1].asked_for_retry = 1;
166                         end
167                 end
168
169                 num_gnt = num_gnt + load_gnt_o ;
170                 num_load_completed = num_load_completed + load_valid_o ;
171                 if ( memop_sync_end_i && ( ( (memop_sb_id_i==fifo[0].sb_id) &&
                        !fifo[0].done ) | ( (memop_sb_id_i==fifo[1].sb_id) && !fifo
                        [1].done ) ) ) begin
172                                     num_ended = num_ended + 1 ;
173                 end
174
175         end
176
177 end
178
179
180
181 ////////////////////////////////////////////////////
182 ////////////////////properties//////////////////////
183 ////////////////////////////////////////////////////
184
185  `ifdef FORMAL
186
187         //vp0.0.0
188         property p_load_req_i_sync;
189                 @(negedge clk_i)
190                 1 |=> @(posedge clk_i)  $stable(load_req_i);
191         endproperty :  p_load_req_i_sync
192
193         //vp 0.0.1
194         property p_memop_sync_end_i_sync;
195                 @(negedge clk_i)
196                 1 |=>  @(posedge clk_i) $stable(memop_sync_end_i);
197         endproperty :  p_memop_sync_end_i_sync
198
199         //vp0.0.2
200         property p_load_ack_i_sync;
201                 @(negedge clk_i)
202                 1 |=>  @(posedge clk_i) $stable(load_ack_i);
203         endproperty :  p_load_ack_i_sync
204
205 `endif
206
207 //AVISPADO IF//////////////////
```

```
208 //vp 1.0.1
209 property p_mem_sync_end_i;
210         @(posedge clk_i)
211         memop_sync_end_i |-> !$isunknown(memop_sb_id_i);
212 endproperty : p_mem_sync_end_i
213
214 //vp 1.0.2
215 property p_valid_after_sync_end;
216         bit [SB_WIDTH-1 : 0] valid_id;
217         @(posedge clk_i)
218         ( memop_sync_end_i && ( ((memop_sb_id_i==fifo[0].sb_id) && !fifo[0].
                done) || ((memop_sb_id_i==fifo[1].sb_id) && !fifo[1].done)) ) ##0
                (1, valid_id = memop_sb_id_i) |-> ##[1:$] load_valid_o && (
                load_sb_id_o == valid_id);
219 endproperty : p_valid_after_sync_end
220
221 //vp 1.0.3
222 property p_memop_sync_end_num;
223         @(negedge clk_i)
224         num_ended == 2 |-> ##1 !memop_sync_end_i || ( memop_sync_end_i && ( ( (
                memop_sb_id_i!=fifo[0].sb_id) || ( (memop_sb_id_i==fifo[0].sb_id)
                && fifo[0].done ) )&& ( (memop_sb_id_i!=fifo[1].sb_id) || ( (
                memop_sb_id_i==fifo[1].sb_id) && fifo[1].done) ) ) ) ;// implement
                the memory to store the ids plz
225 endproperty : p_memop_sync_end_num
226
227 //vp 1.0.4
228 property p_load_ack_i_num;
229         @(posedge clk_i)
230         num_ended == 0 |-> !load_ack_i || ( load_ack_i && ( ( (load_ack_sb_id_i
                !=fifo[0].sb_id) || ( (load_ack_sb_id_i==fifo[0].sb_id) && fifo[0].
                done ) )&& ( (load_ack_sb_id_i!=fifo[1].sb_id) || ( (
                load_ack_sb_id_i==fifo[1].sb_id) && fifo[1].done) ) ) ) ;//
                implement the memory to store the ids plz
231 endproperty : p_load_ack_i_num
232
233 //vp 1.0.5
234 property p_unique_request;
235         @(posedge clk_i)
236         load_req_i |-> ( (load_sb_id_i!=fifo[0].sb_id) || ( (load_sb_id_i==fifo
                [0].sb_id) && (fifo[0].now_granted || fifo[0].done) ) ) && ( (
                load_sb_id_i!=fifo[1].sb_id) || ((load_sb_id_i==fifo[1].sb_id) && (
                fifo[1].now_granted || fifo[1].done) ) )  ;
237 endproperty : p_unique_request
238
239 //LMU IF////////////////////////
240 //vp 1.1.1
241 property p_lmu_valid_load_i;
242         @(posedge clk_i)
243         lmu_dvalid_load_i |->
```

```
244         !$isunknown(load_req_i) && !$isunknown(load_sb_id_i) && !$isunknown(
                load_masked_i) && !$isunknown(load_sew_i) && !$isunknown(
                lmu_dvalid_load_i) && !$isunknown(lmu_enable_i) && !$isunknown(
                lmu_sb_id_i) && !$isunknown(lmu_min_el_id_i) && !$isunknown(
                lmu_el_id_i) && !$isunknown(fsm_read_en_lbuf_i) && !$isunknown(
                fsm_mem_write_en_i) && !$isunknown(load_addr_i) && !$isunknown(
                load_addr_trans_i) && !$isunknown(memop_sync_end_i) && !$isunknown(
                memop_sb_id_i) && !$isunknown(load_ack_i) && !$isunknown(
                load_ack_sb_id_i) && !$isunknown(vstart_self_i) && !$isunknown(
                vstart_next_i);
245
246 endproperty : p_lmu_valid_load_i
247
248 //vp 1.1.2
249 property p_enable_coherence;
250         int sew;
251         @(posedge clk_i)
252         (lmu_dvalid_load_i, sew=load_sew_i) |->  is_coherent(lmu_enable_i,sew);
253 endproperty : p_enable_coherence
254
255
256 //MQ IF//////////////////////////
257 //vp 1.2.1
258 property p_load_request_i;
259         @(posedge clk_i)
260         load_req_i |-> !$isunknown(load_sb_id_i) && !$isunknown(load_masked_i)
                && !$isunknown(load_sew_i) && !$isunknown(load_addr_i) && !
                $isunknown(load_addr_trans_i) && !$isunknown(lmu_min_el_id_i) && !
                $isunknown(lmu_el_id_i);
261 endproperty : p_load_request_i
262
263 //vp 1.2.2
264 property p_req_after_gnt;
265         @(posedge clk_i)
266         $fell(load_req_i) |-> $fell(load_gnt_o);
267 endproperty : p_req_after_gnt
268
269 //vp 1.2.3
270 property p_gnt_after_req;
271         @(posedge clk_i)
272         $rose(load_req_i) |-> $rose(load_req_i)[=1] ##0 load_gnt_o;
273 endproperty : p_gnt_after_req
274
275
276 //vp 1.2.4
277 property p_gnt_beh;
278         @(posedge clk_i)
279         load_gnt_o |-> ##1 (!load_gnt_o) || (load_gnt_o && num_gnt == 1) ; //(!
                load_gnt_o || num_gnt == 1) ;//or ##1 !load_gnt_o;
280 endproperty : p_gnt_beh
281
```

```
282 //vp 1.2.5
283 property p_num_gnt;
284         @(posedge clk_i)
285         load_gnt_o || load_ack_i |-> num_gnt < 3 ;
286 endproperty : p_num_gnt
287
288
289
290 //vp 1.2.6
291 property p_num_req_end;
292         @(posedge clk_i)
293         (num_gnt == 0) |-> !load_ack_i || ( load_ack_i && ( load_ack_sb_id_i!=
                fifo[0].sb_id || (load_ack_sb_id_i==fifo[0].sb_id && fifo[0].done)
                )  && ( load_ack_sb_id_i!=fifo[1].sb_id || (load_ack_sb_id_i==fifo[
                1].sb_id && fifo[1].done) )  ) ;
294 endproperty : p_num_req_end
295
296 //FSM IF////////////////////////
297 //vp 1.3.1
298 property p_read_write_en;
299         @(posedge clk_i)
300         fsm_read_en_lbuf_i || fsm_mem_write_en_i |-> !(fsm_read_en_lbuf_i &&
                fsm_mem_write_en_i);
301 endproperty : p_read_write_en
302
303 //vp 1.3.2
304 property p_write_after_read;
305         @(posedge clk_i)
306         fsm_read_en_lbuf_i ##1 load_ready_o |-> fsm_mem_write_en_i;
307 endproperty : p_write_after_read
308
309
310 //CU IF////////////////////////
311 //vp 1.4.1
312 property p_cu_ack;
313         bit [SB_WIDTH-1:0] ID;
314         @(posedge clk_i)
315         load_valid_o ##0 (1, ID = load_sb_id_o) |-> ##[1:$] load_ack_i && (
                load_ack_sb_id_i == ID);
316 endproperty : p_cu_ack
317
318 //vp 1.4.2
319 property p_cu_ack_num;
320         @(posedge clk_i)
321         load_valid_o || load_ack_i |-> num_load_completed < 3 ;
322 endproperty : p_cu_ack_num
323
324
325 //vp 1.4.3
326 property p_num_load_completed;
327         @(posedge clk_i)
```

```
328         (num_load_completed == 0) |-> !load_ack_i || ( load_ack_i && (
                load_ack_sb_id_i!=fifo[0].sb_id || (load_ack_sb_id_i==fifo[0].sb_id
                 && fifo[0].done) )  && ( load_ack_sb_id_i!=fifo[1].sb_id || (
                load_ack_sb_id_i==fifo[1].sb_id && fifo[1].done) )  ) ;
329 endproperty : p_num_load_completed
330
331
332 //VRF IF////////////////////////
333 //vp 1.5.1
334 property p_output_data_known;
335         @(posedge clk_i)
336         load_write_en_rf_o |-> !$isunknown(load_addr_o) && !$isunknown(
                load_addr_trans_o) && !$isunknown(load_data_o) && !$isunknown(
                load_data_en_o);
337 endproperty : p_output_data_known
338
339 //vp 1.5.2
340 property p_write_en_rf;
341         @(posedge clk_i)
342         fsm_mem_write_en_i && load_ready_o |-> load_write_en_rf_o;
343 endproperty : p_write_en_rf
344
345 //vp 1.5.3
346 property p_load_valid_addr;
347         bit [VADDR_WIDTH-1:0] load_addr;
348         @(posedge clk_i)
349         ($rose(load_req_i), load_addr = load_addr_i) |-> ##[1:$] $rose(
                load_valid_o) && load_valid_addr_o == load_addr;
350 endproperty : p_load_valid_addr
351
352
353
354 //VL IF////////////////////////
355 //vp 1.6.1
356 property p_load_inflight_o;
357         @(negedge clk_i)
358         $rose(load_inflight_o) || $fell(load_inflight_o) |-> (load_inflight_o
                && num_gnt>0) || (!load_inflight_o && num_gnt==0);
359 endproperty : p_load_inflight_o
360
361 //vp 1.6.2
362 property p_load_ready_o;
363         @(posedge clk_i)
364          load_inflight_o && lmu_dvalid_load_i |-> ##[0:$] fsm_read_en_lbuf_i
                 ##1 load_ready_o;
365 endproperty : p_load_ready_o
366
367 //vp 1.6.3
368         //vp 1.6.3.0
369         property p_data_corruption_64;
370                 logic [DATA_PATH_WIDTH-1:0] data_i;
```

```
371                 logic [SEW_WIDTH-2:0] sew_i;
372                 logic [EL_ID_WIDTH-1:0] el_id_i;
373                 logic [MASK_BANK_WORD-1:0] enable_i;
374                 logic [SB_WIDTH-1:0] sb_id_i;
375
376                 @(posedge clk_i)
377
378                 (lmu_dvalid_load_i, data_i = lmu_data_i, sb_id_i = lmu_sb_id_i,
                        el_id_i = lmu_el_id_i[0], enable_i = lmu_enable_i) ##0
379                 ((lmu_sb_id_i == fifo[0].sb_id && !fifo[0].done, sew_i = fifo[0
                        ].sew) or (lmu_sb_id_i == fifo[1].sb_id && !fifo[1].done,
                        sew_i = fifo[1].sew)) ##0
380                 sew_i == 2'b11
381                 |-> first_match( (1) [*0:$] ##0 ( (load_write_en_rf_o && (
                        load_data_o[get_el_bank(el_id_i, sew_i)*64 + 63 -: 64] ==
                        data_i)) ||
382                 !enable_i[0] ||
383                 (load_ack_i && (sb_id_i==load_ack_sb_id_i)) ||
384                 ( (lmu_sb_id_i == fifo[0].sb_id && fifo[0].asked_for_retry && !
                        fifo[0].done) ||
385                 (lmu_sb_id_i == fifo[1].sb_id && fifo[1].asked_for_retry && !
                        fifo[1].done) )))
386                 |-> !(load_ack_i && (sb_id_i==load_ack_sb_id_i));
387         endproperty : p_data_corruption_64
388
389
390
391         //vp 1.6.3.1
392         property p_data_corruption_32(i_32);
393                 logic [DATA_PATH_WIDTH-1:0] data_i;
394                 logic [SEW_WIDTH-2:0] sew_i;
395                 logic [MASK_BANK_WORD-1:0][EL_ID_WIDTH-1:0] el_id_i;
396                 logic [MASK_BANK_WORD-1:0] enable_i;
397                 logic [SB_WIDTH-1:0] sb_id_i;
398
399                 @(posedge clk_i)
400
401                 (lmu_dvalid_load_i, data_i = lmu_data_i, sb_id_i = lmu_sb_id_i,
                         el_id_i = lmu_el_id_i[i_32*4], enable_i = lmu_enable_i)
                        ##0
402                 (((lmu_sb_id_i == fifo[0].sb_id) && (fifo[0].done == 0), sew_i
                        = fifo[0].sew) or ((lmu_sb_id_i == fifo[1].sb_id) && (fifo[
                        1].done == 0), sew_i = fifo[1].sew)) ##0
403                 sew_i == 2'b10
404                 |-> first_match( (1) [*0:$] ##0 ( ( load_write_en_rf_o && ( (
                        load_data_o[get_el_bank(el_id_i, sew_i)*64 + 63 -: 32] ==
                        data_i[i_32*32 + 31 -: 32]) ||
405                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 31 -: 32] ==
                        data_i[i_32*32 + 31 -: 32]) )) ||
406                 !enable_i[i_32*4] ||
407                 (load_ack_i && (sb_id_i==load_ack_sb_id_i)) ||
```

```
408                 ( (lmu_sb_id_i == fifo[0].sb_id && fifo[0].asked_for_retry && !
                        fifo[0].done) ||
409                 (lmu_sb_id_i == fifo[1].sb_id && fifo[1].asked_for_retry && !
                        fifo[1].done) )))
410                 |-> !(load_ack_i && (sb_id_i==load_ack_sb_id_i));
411
412         endproperty : p_data_corruption_32
413
414         //vp 1.6.3.2
415         property p_data_corruption_16(i_16);
416                 logic [DATA_PATH_WIDTH-1:0] data_i;
417                 logic [SEW_WIDTH-2:0] sew_i;
418                 logic [MASK_BANK_WORD-1:0][EL_ID_WIDTH-1:0] el_id_i;
419                 logic [MASK_BANK_WORD-1:0] enable_i;
420                 logic [SB_WIDTH-1:0] sb_id_i;
421
422                 @(posedge clk_i)
423
424                 (lmu_dvalid_load_i, data_i = lmu_data_i, sb_id_i = lmu_sb_id_i,
                         el_id_i = lmu_el_id_i[i_16*2], enable_i = lmu_enable_i)
                        ##0
425                 (((lmu_sb_id_i == fifo[0].sb_id) && (fifo[0].done == 0), sew_i
                        = fifo[0].sew) or ((lmu_sb_id_i == fifo[1].sb_id) && (fifo[
                        1].done == 0), sew_i = fifo[1].sew)) ##0
426                 sew_i == 2'b01
427                 |-> first_match( (1) [*0:$] ##0 ( ( load_write_en_rf_o && (
                              (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 63 -: 16
                        ] == data_i[i_16*16 + 15 -: 16]) ||
428                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 47 -: 16] ==
                        data_i[i_16*16 + 15 -: 16]) ||
429                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 31 -: 16] ==
                        data_i[i_16*16 + 15 -: 16]) ||
430                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 15 -: 16] ==
                        data_i[i_16*16 + 15 -: 16]) ) ) ||
431                 !enable_i[i_16*2] ||
432                 ( load_ack_i && (sb_id_i==load_ack_sb_id_i)) ||
433                 ( (lmu_sb_id_i == fifo[0].sb_id && fifo[0].asked_for_retry && !
                        fifo[0].done) ||
434                 (lmu_sb_id_i == fifo[1].sb_id && fifo[1].asked_for_retry && !
                        fifo[1].done)))
435                 |-> !(load_ack_i && (sb_id_i==load_ack_sb_id_i));
436
437         endproperty : p_data_corruption_16
438
439         //vp 1.6.3.3
440         property p_data_corruption_8(i_8);
441                 logic [DATA_PATH_WIDTH-1:0] data_i;
442                 logic [SEW_WIDTH-2:0] sew_i;
443                 logic [MASK_BANK_WORD-1:0][EL_ID_WIDTH-1:0] el_id_i;
444                 logic [MASK_BANK_WORD-1:0] enable_i;
445                 logic [SB_WIDTH-1:0] sb_id_i;
```

```
446
447                 @(posedge clk_i)
448
449                 (lmu_dvalid_load_i, data_i = lmu_data_i, sb_id_i = lmu_sb_id_i,
                         el_id_i = lmu_el_id_i[i_8], enable_i = lmu_enable_i) ##0
450                 (((lmu_sb_id_i == fifo[0].sb_id) && (fifo[0].done == 0), sew_i
                        = fifo[0].sew) or ((lmu_sb_id_i == fifo[1].sb_id) && (fifo[
                        1].done == 0), sew_i = fifo[1].sew)) ##0
451                 sew_i == 2'b00
452                 |-> first_match( (1) [*0:$] ##0 ( ( load_write_en_rf_o && (
                             (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 63 -: 8]
                         == data_i[i_8*8 + 7 -: 8]) ||
453                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 55 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
454                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 47 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
455                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 39 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
456                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 31 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
457                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 23 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
458                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 + 15 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ||
459                 (load_data_o[get_el_bank(el_id_i, sew_i)*64 +  7 -: 8] ==
                        data_i[i_8*8 + 7 -: 8]) ) ) ||
460                 !enable_i[i_8] ||
461                 ( load_ack_i && (sb_id_i==load_ack_sb_id_i))  ||
462                 ( (lmu_sb_id_i == fifo[0].sb_id && fifo[0].asked_for_retry && !
                        fifo[0].done) ||
463                 (lmu_sb_id_i == fifo[1].sb_id && fifo[1].asked_for_retry && !
                        fifo[1].done) )))
464                 |-> !(load_ack_i && (sb_id_i==load_ack_sb_id_i));
465
466
467         endproperty : p_data_corruption_8
468
469
470
471 //RSN IF//////////////////////
472 //vp 1.7.1
473 property p_rsn_output;
474         @(posedge clk_i)
475         !rsn_i |-> !load_write_en_rf_o && !load_gnt_o && !load_valid_o && !
                load_inflight_o && retry_self_o == '0 && retry_next_o == '0;
476 endproperty : p_rsn_output
477
478 //vp 1.7.2
479 property p_rsn_input;
480         @(posedge clk_i)
481         !rsn_i |-> lmu_enable_i == '0 && !load_req_i && !fsm_read_en_lbuf_i;
```

```
482 endproperty : p_rsn_input
483
484 //vp 1.7.3
485 property p_kill_input;
486         @(posedge clk_i)
487         kill_i |-> !load_req_i;
488 endproperty : p_kill_input
489
490
491
492 ////////////////////////////////////////////////////
493 ////////////////////assert & assume/////////////////////
494 ////////////////////////////////////////////////////
495
496 //FORMAL ASSUMPTION///////////////
497
498 `ifdef FORMAL
499
500         //vp 0.0.0
501         a_load_req_i_sync :             assume property( disable iff (!rsn_i ||
                kill_i) p_load_req_i_sync) else $error("load_req_i changed on the
               negedge");
502
503         //vp 0.0.1
504         a_memop_sync_end_i_sync :       assume property( disable iff (!rsn_i ||
                kill_i) p_memop_sync_end_i_sync) else $error("load_req_i changed
               on the negedge");
505
506         //vp 0.0.2
507         a_load_ack_i_sync :             assume property( disable iff (!rsn_i ||
                kill_i) p_load_ack_i_sync) else $error("load_req_i changed on the
               negedge");
508
509 `endif
510
511 //AVISPADO IF//////////////////
512 //vp 1.0.1
513 a_mem_sync_end_i :              assume property (disable iff (!rsn_i || kill_i)
        p_mem_sync_end_i) else $error("unknown memop scoreboard id on
       memop_sync_end_i asserted");
514
515 //vp 1.0.2
516 a_valid_after_sync_end :        assert property (disable iff (!rsn_i || kill_i)
        p_valid_after_sync_end) else $error("no load_valid after a sync_end");
517
518
519 //vp 1.0.3
520 a_memop_sync_end_num :          assume property (disable iff (!rsn_i || kill_i)
        p_memop_sync_end_num) else $error("too much memop_sync_end ");
521
522
```

```
523 //vp 1.0.4
524 a_load_ack_i_num :              assume property (disable iff (!rsn_i || kill_i)
        p_load_ack_i_num) else $error("too much memop_sync_end ");
525
526 //vp 1.0.5
527 a_unique_request :              assume property (disable iff (!rsn_i || kill_i)
        p_unique_request) else $error("requested an already issued load");
528
529 //LMU IF////////////////////////
530 //vp 1.1.1
531 a_lmu_valid_load_i :            assume property (disable iff (!rsn_i || kill_i)
        p_lmu_valid_load_i) else $error("unknown input on mem_dvalid_load_i
       asserted");
532
533 //vp 1.1.2
534 a_enable_coherence :            assume property (disable iff (!rsn_i || kill_i)
        p_enable_coherence) else $error("wrong lmu_enable_i to LB");
535
536 //MQ IF////////////////////////
537 //vp 1.2.1
538 a_load_request_i :              assume property (disable iff (!rsn_i || kill_i)
        p_load_request_i) else $error("unknown load_inputs on load_req_i asserted"
       );
539
540 //vp 1.2.2
541 a_req_after_gnt :               assume property (disable iff (!rsn_i || kill_i)
        p_req_after_gnt) else $error("req was 1 after a gnt");
542
543 //vp 1.2.3
544 a_gnt_after_req :               assert property (disable iff (!rsn_i || kill_i)
        p_gnt_after_req) else $error("no gnt after a req");
545
546 //vp 1.2.4
547 a_gnt_beh :                     assert property (disable iff (!rsn_i || kill_i)
        p_gnt_beh) else $error("gnt lasted more than 1 clock cycle");
548
549 //vp 1.2.5
550 a_num_gnt :                     assert property (disable iff (!rsn_i || kill_i)
        p_num_gnt) else $error("a load_gnt_o was asserted with no enought requests
       ");
551
552 //vp 1.2.6
553 a_num_req_end :                 assume property (disable iff (!rsn_i || kill_i)
        p_num_req_end) else $error("too many mem_sync_ends or too many load_req_i"
       );
554
555
556 //FSM IF////////////////////////
557 //vp 1.3.1
558 a_read_write_en :               assume property (disable iff (!rsn_i || kill_i)
        p_read_write_en) else $error("read and write enables at the same time");
```

```
559
560 //vp 1.3.2
561 a_write_after_read :            assume property (disable iff (!rsn_i || kill_i)
        p_write_after_read) else $error("no write after a read");
562
563
564 //CU IF//////////////////////////
565 //vp 1.4.1
566 a_cu_ack :                      assume property (disable iff (!rsn_i || kill_i)
        p_cu_ack) else $error("wrong handshake of load_valid with load_ack");
567
568 //vp 1.4.2
569 a_cu_ack_num :                  assert property (disable iff (!rsn_i || kill_i)
        p_cu_ack_num) else $error("load_ack without a load_valid_o");
570
571 //vp 1.4.3
572 a_num_load_completed :          assume property (disable iff (!rsn_i || kill_i)
        p_num_load_completed) else $error("to many load_ack_i");
573
574 //VRF IF////////////////////////
575 //vp 1.5.1
576 a_output_data_known :           assert property (disable iff (!rsn_i || kill_i)
        p_output_data_known) else $error("unknown outputs on load_write_en_rf_o");
577
578 //vp 1.5.2
579 a_write_en_rf :                 assert property (disable iff (!rsn_i || kill_i)
        p_write_en_rf) else $error("load_write_en_rf_o when a valid load_write was
        possible");
580
581 //vp 1.5.3
582 a_load_valid_addr :             assert property (disable iff (!rsn_i || kill_i)
        p_load_valid_addr) else $error("load addr is not matching");
583
584
585 //VL IF////////////////////////
586 //vp 1.6.1
587 a_load_inflight_o  :            assert property (disable iff(!rsn_i || kill_i)
       p_load_inflight_o) else $error("wrong load_inflight_o");
588
589 //vp 1.6.2
590 a_load_ready_o :                assert property (disable iff(!rsn_i || kill_i)
       p_load_ready_o) else $error("ready_o = 0 when a data was into the buffer
       during the write");
591
592 //vp 1.6.3
593 generate
594         //1.6.3.0
595         a_data_corruption_64 :  assert property (disable iff (!rsn_i || kill_i)
              p_data_corruption_64) else $error("data_i didn't pass trought the
             LB");
596
```

```
597         //1.6.3.1
598         for (genvar i_32 = 0; i_32 < 2; i_32++)
599                 a_data_corruption_32 :  assert property (disable iff (!rsn_i ||
                         kill_i) p_data_corruption_32(i_32)) else $error("data_i
                        didn't pass trought the LB");
600         //1.6.3.2
601         for (genvar i_16 = 0; i_16 < 4; i_16++)
602                 a_data_corruption_16 :  assert property (disable iff (!rsn_i ||
                         kill_i) p_data_corruption_16(i_16)) else $error("data_i
                        didn't pass trought the LB");
603         //1.6.3.3
604         for (genvar i_8 = 0; i_8 < 8; i_8++)
605                 a_data_corruption_8 :   assert property (disable iff (!rsn_i ||
                         kill_i) p_data_corruption_8(i_8)) else $error("data_i didn
                        't pass trought the LB");
606
607 endgenerate
608
609 //RSN IF//////////////////////
610 //vp 1.7.1
611 a_rsn_output :                  assert property (p_rsn_output) else $error("
        wrong outputs on reset");
612
613 //vp 1.7.2
614 a_rsn_input :                   assume property (p_rsn_input) else $error("
        wrong inputs on reset");
615
616 //vp 1.7.3
617 a_kill_input :                  assume property (p_kill_input) else $error("
        wrong inputs on kill");
618
619
620
621 ////////////////////////////////////////////////////////
622 ////////////////////// tasks ///////////////////////////
623 ////////////////////////////////////////////////////////
624
625
626 // This function computes the bank of an element
627 function [BANK_IDX_SZ-1:0] get_el_bank([EL_ID_WIDTH-1:0] el_id, [SEW_WIDTH-1:0]
         sew);
628     case (sew[1:0])
629         SEW64: get_el_bank=((el_id/N_LANES)%N_BANKS); // 0, 1 ... 7 go to bank
                0, 8 to 15 to bank 1 ... until 40-47 that go again to bank 0, etc.
630         SEW32: get_el_bank=(((el_id>>1)/N_LANES)%N_BANKS); // 0 to 15 go to
                bank 0, 16 to 31 to bank 1 ... until 80-95 that go again to bank 0,
                 etc.
631         SEW16: get_el_bank=(((el_id>>2)/N_LANES)%N_BANKS); // 0 to 31 go to
                bank 0, 32 to 63 to bank 1 ... until 160-191 that go again to bank
                0, etc.
```

```
632         SEW8:  get_el_bank=(((el_id>>3)/N_LANES)%N_BANKS); // 0 to 63 go to
                bank 0, 64 to 128 to bank 1 ... until 320-383 that go again to bank
                 0, etc.
633     endcase
634 endfunction
635
636 // This function checks the coherence of lmu_enable_i
637 function bit is_coherent([MASK_BANK_WORD-1:0] lmu_enable_i, [SEW_WIDTH-1:0] sew
    );
638     is_coherent = 0;
639     case (sew[1:0])
640         SEW64: if( (lmu_enable_i == '0) || (lmu_enable_i == '1) ) is_coherent =
                1;
641         SEW32:  if ( ( (lmu_enable_i[3:0] == '0) || (lmu_enable_i[3:0] == '1) )
                && ( (lmu_enable_i[7:4] == '0) || (lmu_enable_i[7:4] == '1) ) )
               is_coherent = 1;
642
643         SEW16:  if ( ( (lmu_enable_i[1:0] == '0) || (lmu_enable_i[1:0] == '1) )
                && ( (lmu_enable_i[3:2] == '0) || (lmu_enable_i[3:2] == '1) ) && (
                (lmu_enable_i[5:4] == '0) || (lmu_enable_i[5:4] == '1) ) && ( (
               lmu_enable_i[7:6] == '0) || (lmu_enable_i[7:6] == '1) ) )
               is_coherent = 1;
644         SEW8: is_coherent=1;
645     endcase
646 endfunction
647
648
649
650
651
652
653 endmodule: LoadBuffer_checker
```

A.1.2 Load Management Unit

```
1  `define FUNCTIONAL // comment this when doing formal verification, (Quest CDC
       does not support task)
2
3  import EPI_pkg::*;
4
5  bind load_management_unit load_management_unit_checker #(
6          .MEM_DATA_WIDTH(MEM_DATA_WIDTH),
7          .SEQ_ID_WIDTH(SEQ_ID_WIDTH),
8          .MAX_NUMBER_ELEMENTS(MAX_NUMBER_ELEMENTS),
9          .AVISPADO_LOAD_MASK_WIDTH(AVISPADO_LOAD_MASK_WIDTH),
10         .NUM_LANES(NUM_LANES)) bind_load_management_unit_checker (.*);
11
12
13
14 module load_management_unit_checker
15         #(
```

```
16          parameter MEM_DATA_WIDTH = 512,
17          parameter SEQ_ID_WIDTH = 33,
18          parameter MAX_NUMBER_ELEMENTS = 64,
19          parameter AVISPADO_LOAD_MASK_WIDTH = 64,
20          parameter NUM_LANES = 8
21          )(
22          input                                                   clk_i,
23          input                                                   rsn_i,
24          input                                                   kill_i,
25          input                                                   load_granted_i,
26          input [SB_WIDTH-1:0]
                load_granted_sb_id_i,
27          input
                indexed_load_granted_i,
28          input                                                   load_sync_end_i
                ,
29          input [SB_WIDTH-1:0]
                load_sync_end_sb_id_i,
30          input
                load_data_valid_i,
31          input [MEM_DATA_WIDTH-1:0]                              load_data_i,
32          input [SEQ_ID_WIDTH-1:0]                                seq_id_i,
33          input                                                   mask_valid_i,
34          input [AVISPADO_LOAD_MASK_WIDTH-1:0]                    mask_i,
35          input [SEW_WIDTH-2:0]                                   sew_i,      //
                SEW (00:8 - 01:16 - 10:32 - 11:64)
36          input signed [XREG_WIDTH-1:0]                           stride_i,   //
                stride in bytes
37          input [MEM_DATA_WIDTH-1:0]                              load_data_o,
38          input                                                   load_dvalid_o,
39          input [AVISPADO_LOAD_MASK_WIDTH-1:0]                    mask_o,
40          input [MAX_NUMBER_ELEMENTS-1:0] [EL_ID_WIDTH-1:0]       element_ids_o,
41          input [SB_WIDTH-1:0]                                    sb_id_o,
42          input [CSR_VLEN_START-1:0]                              vstart_self_o,
43          input [CSR_VLEN_START-1:0]                              vstart_next_o,
44          input [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]
                min_element_id_idx_o
45          );
46
47  ////////////////////////////////////////////////////
48  ////////////////////  variables  ///////////////////
49  ////////////////////////////////////////////////////
50
51  typedef struct {
52          longint signed STRIDE;
53          int unsigned SEW;
54          int unsigned sb_id;
55          bit is_indexed;
56          bit done;
57  } granted_load;
58
```

```
59
60  granted_load fifo[1:0];
61  bit is_indexed;
62  bit sb_correct;
63  int unsigned SEW;
64  longint signed STRIDE;
65  int unsigned EL_COUNT;
66  int unsigned OFFSET;
67  int unsigned EL_ID;
68  int unsigned N_ELEMENTS;
69  int unsigned num_load_inflight;
70
71  bit [MEM_DATA_WIDTH-1:0]  computed_data_o;
72  bit [MEM_DATA_WIDTH-1:0]  x_display0;
73  bit [MEM_DATA_WIDTH-1:0]  x_display1;
74  bit [MEM_DATA_WIDTH-1:0]  x_display2;
75  bit [AVISPADO_LOAD_MASK_WIDTH-1:0] m_result;
76  bit [MAX_NUMBER_ELEMENTS-1:0][EL_ID_WIDTH-1:0] computed_el_id_o;
77  bit [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]         c_min_element_id_idx;
78  bit [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]         c_min_element_id_idx_1;
79  bit [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]         c_min_element_id_idx_2;
80  bit [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]         c_min_element_id_idx_3;
81  bit [MEM_DATA_WIDTH-1:0] big_mask;
82  bit [AVISPADO_LOAD_MASK_WIDTH-1:0] valid_bytes;
83  bit [NUM_LANES-1:0] valid_mins;
84
85  ////////////////////////////////////////////////////
86  ////////////////////processes///////////////////////
87  ////////////////////////////////////////////////////
88
89
90  initial begin
91          sb_correct = 1;
92          computed_data_o = '0;
93          fifo[0].sb_id = 20; // we have to initialize it to a value that cannot
                be in input, to avoid to read the wrong SEW and STRIDE from the
                wrong fifo cell
94          fifo[1].sb_id = 20;
95          fifo[1].done = 1;
96          fifo[0].done = 1;
97          num_load_inflight = 0;
98  end
99
100
101 // sew and stride fifo
102 always @(negedge clk_i)
103         begin
104         if ( !rsn_i || kill_i ) begin
105                 fifo[1].SEW = '0;
106                 fifo[1].STRIDE = '0 ;
107                 fifo[1].sb_id = 20 ;
```

```
108                 fifo[1].done = 1;
109                 fifo[0].SEW = '0;
110                 fifo[0].STRIDE = '0;
111                 fifo[0].sb_id = 20;
112                 fifo[0].done = 1;
113         end
114
115         if ( load_sync_end_i ) begin
116                 if ( load_sync_end_sb_id_i == fifo[1].sb_id ) begin
117                         fifo[1].done = 1;
118                 end
119                 if ( load_sync_end_sb_id_i == fifo[0].sb_id ) begin
120                         fifo[0].done = 1;
121                 end
122         end
123
124         if ( load_granted_i ) begin
125                 if ( fifo[1].done ) begin
126                         fifo[1].SEW = sew_i;
127                         fifo[1].STRIDE = stride_i;
128                         fifo[1].sb_id = load_granted_sb_id_i ;
129                         fifo[1].is_indexed = indexed_load_granted_i;
130                         fifo[1].done = 0;
131                 end
132                 else if( fifo[0].done ) begin
133                         fifo[0].SEW = sew_i;
134                         fifo[0].STRIDE = stride_i;
135                         fifo[0].sb_id = load_granted_sb_id_i;
136                         fifo[0].is_indexed = indexed_load_granted_i;
137                         fifo[0].done = 0;
138                 end
139         end
140
141         // check number of load_inflight
142         if ( !rsn_i || kill_i) begin
143                 num_load_inflight = 0;
144         end
145         else begin
146                 num_load_inflight = num_load_inflight + load_granted_i - (
                        load_sync_end_i && ( ( load_sync_end_sb_id_i == fifo[0].
                        sb_id ) | ( load_sync_end_sb_id_i == fifo[1].sb_id ) ) );
147         end
148
149         `ifdef FUNCTIONAL
150
151                 // sb_id check and start computing data
152                 if ( load_data_valid_i ) begin
153                         check_sb_correct();
154                 end
155
156
```

```
157                 // big_mask calculation
158                 if ( load_dvalid_o ) begin
159                         compute_big_mask_o();
160                 end
161
162         `endif
163
164 end
165
166
167
168
169 ////////////////////////////////////////////////////
170 ////////////////////properties//////////////////////
171 ////////////////////////////////////////////////////
172
173
174 // 1.0.1  vp
175 property p_el_count;
176         int SEW0, SEW1, OFFSET;
177         longint signed STRIDE0, STRIDE1;
178         @(posedge clk_i)
179
180         ( load_data_valid_i && sb_correct,
181         OFFSET = seq_id_i[21:16],
182         SEW0 = 2**(3+fifo[0].SEW),
183         STRIDE0 = (fifo[0].STRIDE > 0) ? fifo[0].STRIDE : -fifo[0].STRIDE,
184         SEW1 = 2**(3+fifo[1].SEW),
185         STRIDE1 = (fifo[1].STRIDE > 0) ? fifo[1].STRIDE : -fifo[1].STRIDE
186         )
187         |->
188         ( (seq_id_i[32:29] == fifo[0].sb_id) && ( seq_id_i[28:22] <= (
              MEM_DATA_WIDTH/(STRIDE0*8) - OFFSET*SEW0/(STRIDE0*8)) ) ) or
189         ( (seq_id_i[32:29] == fifo[1].sb_id) && ( seq_id_i[28:22] <= (
              MEM_DATA_WIDTH/(STRIDE1*8) - OFFSET*SEW1/(STRIDE1*8)) ) ) or
190         ( seq_id_i[28:22] == 1 ) ;
191 endproperty : p_el_count
192
193
194 // 1.0.2 vp
195 property p_stride_i;
196         int SEW0, SEW1;
197         longint signed STRIDE0, STRIDE1;
198         @(posedge clk_i)
199         ( load_data_valid_i &&
200         ( ( (seq_id_i[32:29] == fifo[0].sb_id) && !fifo[0].is_indexed && !fifo[
              0].done) ||
201           ( (seq_id_i[32:29] == fifo[1].sb_id) && !fifo[1].is_indexed && !fifo[
                1].done) ),
202         SEW0 = (2**(3+fifo[0].SEW))/8,
203         SEW1 = (2**(3+fifo[1].SEW))/8,
```

```
204         STRIDE0 = (fifo[0].STRIDE > 0) ? fifo[0].STRIDE : -fifo[0].STRIDE,
205         STRIDE1 = (fifo[1].STRIDE > 0) ? fifo[1].STRIDE : -fifo[1].STRIDE
206         )
207         |->
208         ( ( (seq_id_i[32:29] == fifo[0].sb_id) && !fifo[0].done) && !(STRIDE0 %
                SEW0) ) ||
209         ( ( (seq_id_i[32:29] == fifo[1].sb_id) && !fifo[1].done) && !(STRIDE1 %
                SEW1) );
210 endproperty : p_stride_i
211
212
213 // 1.0.3 vp
214 property p_load_granted_i_known;
215         @(posedge clk_i)
216         load_granted_i |-> !$isunknown(sew_i) && !$isunknown(
              load_granted_sb_id_i) ;
217 endproperty : p_load_granted_i_known
218
219 // 1.0.4 vp
220 property p_load_data_i_known;
221         @(posedge clk_i)
222         load_data_valid_i |-> !$isunknown(load_data_i) && !$isunknown(seq_id_i)
                && !$isunknown(mask_valid_i) && !$isunknown(mask_i) ;
223 endproperty : p_load_data_i_known
224
225 // 1.0.5 vp
226 property p_sb_id_o;
227         int sb ;
228         @(posedge clk_i)
229         (load_data_valid_i && sb_correct, sb=seq_id_i[32:29]) |-> ##1 sb_id_o
              == sb;
230 endproperty : p_sb_id_o
231
232 // 1.0.6 vp
233 property p_load_dvalid_known;
234         @(posedge clk_i)
235         load_dvalid_o |->       !$isunknown(load_data_o) && !$isunknown(mask_o)
                && !$isunknown(element_ids_o) && !$isunknown(vstart_self_o)  && !
              $isunknown(vstart_next_o) &&                                    !
              $isunknown(min_element_id_idx_o);
236 endproperty : p_load_dvalid_known
237
238 // 1.0.7 vp
239 property p_dvalid_o;
240         @(posedge clk_i)
241         load_data_valid_i && sb_correct |-> ##1 load_dvalid_o ;
242 endproperty : p_dvalid_o
243
244 // 1.0.8 vp
245 property p_vstart_o;
246         int unsigned count, id;
```

```
247         @(negedge clk_i)
248         (load_data_valid_i   , count=seq_id_i[28:22], id=seq_id_i[15:5])|-> ##1
                ( (vstart_self_o == id) && (vstart_next_o == count + id) );
249 endproperty : p_vstart_o
250
251 // 1.0.9 vp
252 property p_granted_ids;
253         @(negedge clk_i)
254         load_granted_i |-> ( ( load_granted_sb_id_i != fifo[0].sb_id && !fifo[0
                ].done ) || fifo[0].done ) && ( ( load_granted_sb_id_i != fifo[1].
                sb_id && !fifo[1].done ) || fifo[1].done );
255 endproperty : p_granted_ids
256
257 // 1.0.10 vp
258 property p_indexed_instr;
259         @(posedge clk_i)
260         load_data_valid_i && ( ( (seq_id_i[32:29] == fifo[0].sb_id) && fifo[0].
                is_indexed ) || ( (seq_id_i[32:29] == fifo[1].sb_id) && fifo[1].
                is_indexed ) ) |-> seq_id_i[28:22]==1 ;
261 endproperty : p_indexed_instr
262
263 // 1.0.11 vp
264 property p_load_sync_end_i;
265         bit [SB_WIDTH-1:0] sb_id_granted;
266         int counter;
267         @(posedge clk_i)
268         ( load_granted_i , sb_id_granted = load_granted_sb_id_i, counter = 3 )
                |-> ##[1:$] load_sync_end_i && sb_id_granted ==
                load_sync_end_sb_id_i;
269 endproperty : p_load_sync_end_i
270
271 // 1.0.12 vp
272 property p_load_grant_beh;
273         @(posedge clk_i)
274         num_load_inflight == 2 |-> ##1 !load_granted_i;
275 endproperty : p_load_grant_beh
276
277
278 // 1.0.13 vp
279 property p_load_sync_end_beh;
280         @(posedge clk_i)
281         num_load_inflight == 0 |-> ##1 !load_sync_end_i || ( load_sync_end_i &&
                 (load_sync_end_sb_id_i!=fifo[0].sb_id || (load_sync_end_sb_id_i==
                fifo[0].sb_id && fifo[0].done) ) && (load_sync_end_sb_id_i!=fifo[1]
                .sb_id || (load_sync_end_sb_id_i==fifo[1].sb_id && fifo[1].done) )
                );
282 endproperty : p_load_sync_end_beh
283
284 // 1.0.14 vp
285 property p_load_req_sync_end;
286         @(posedge clk_i)
```

```
287         load_granted_i |-> !load_sync_end_i || ( load_sync_end_i &&
                load_sync_end_sb_id_i!=load_granted_sb_id_i );
288 endproperty : p_load_req_sync_end
289
290 // 1.0.15 vp
291 property p_el_count_sew_i;
292         int SEW0, SEW1, OFFSET;
293         longint signed STRIDE0, STRIDE1;
294         @(posedge clk_i)
295         ( load_data_valid_i && sb_correct,
296         OFFSET = seq_id_i[21:16],
297         SEW0 = (2**(3+fifo[0].SEW))/8,
298         STRIDE0 = (fifo[0].STRIDE > 0) ? fifo[0].STRIDE : -fifo[0].STRIDE,
299         SEW1 = (2**(3+fifo[1].SEW))/8,
300         STRIDE1 = (fifo[1].STRIDE > 0) ? fifo[1].STRIDE : -fifo[1].STRIDE
301         ) ##0
302         ( (seq_id_i[32:29]==fifo[0].sb_id) && !fifo[0].is_indexed && !fifo[0].
              done && ( STRIDE0!=SEW0 && STRIDE0!=2*SEW0 && STRIDE0!=3*SEW0 &&
              STRIDE0!=4*SEW0 ) ||
303           (seq_id_i[32:29]==fifo[1].sb_id) && !fifo[1].is_indexed && !fifo[1].
                done && ( STRIDE1!=SEW1 && STRIDE1!=2*SEW1 && STRIDE1!=3*SEW1 &&
                STRIDE1!=4*SEW1 ) )
304         |->
305         ( seq_id_i[28:22] == 1 ) ;
306 endproperty : p_el_count_sew_i
307
308 `ifdef FUNCTIONAL
309         // 1.1.1 vp
310         property p_sb_correct;
311                 @(posedge clk_i)
312                 load_data_valid_i && !sb_correct |-> ##1 !load_dvalid_o;
313         endproperty : p_sb_correct
314
315         // 1.1.2 vp
316         property p_data_o;
317                 bit [MEM_DATA_WIDTH-1:0] data_out;
318                 @(posedge clk_i)
319                 (load_data_valid_i && sb_correct, data_out = computed_data_o)
                        |-> ##1 (load_data_o & big_mask) == (data_out & big_mask);
320         endproperty : p_data_o
321
322         // 1.1.3 vp
323         property p_mask_o;
324                 bit [MEM_DATA_WIDTH-1:0] mask_out;
325                 @(posedge clk_i)
326                 (load_data_valid_i && sb_correct, mask_out = m_result)|-> ##1
                        mask_o == mask_out;
327         endproperty : p_mask_o
328
329         // 1.1.4 vp
330         property p_el_ids_o;
```

```
331                 bit [MAX_NUMBER_ELEMENTS-1:0][EL_ID_WIDTH-1:0] buffer_ids;
332                 @(posedge clk_i)
333                 (load_data_valid_i  && sb_correct, buffer_ids =
                        computed_el_id_o)|-> ##1 element_ids_o == buffer_ids;
334         endproperty : p_el_ids_o
335
336         // 1.1.5 vp
337         property p_min_element_id_idx_o;
338                 bit [NUM_LANES-1:0][$clog2(MASK_BANK_WORD)-1:0]
                        b_min_ids,b_min_ids_1,b_min_ids_2,b_min_ids_3;
339                 bit [NUM_LANES-1:0] b_valid_mins;
340                 int SEW0, SEW1;
341                 bit [3:0] sb_id;
342                 @(posedge clk_i)
343                 (load_data_valid_i && sb_correct, b_min_ids =
                        c_min_element_id_idx, b_min_ids_1 = c_min_element_id_idx_1,
                         b_min_ids_2 = c_min_element_id_idx_2, b_min_ids_3 =
                        c_min_element_id_idx_3, SEW0 = fifo[0].SEW, SEW1 = fifo[1].
                        SEW, sb_id=seq_id_i[32:29], b_valid_mins=valid_mins)|-> ##1
344 ( sb_id == fifo[0].sb_id && SEW0==3 ) || ( sb_id == fifo[1].sb_id && SEW1==3 )
        || (
345 ( ( min_element_id_idx_o[0]==b_min_ids[0] || min_element_id_idx_o[0]==
        b_min_ids_1[0] || min_element_id_idx_o[0]==b_min_ids_2[0] ||
        min_element_id_idx_o[0]==b_min_ids_3[0] ) || !b_valid_mins[0] ) &&
346 ( ( min_element_id_idx_o[1]==b_min_ids[1] || min_element_id_idx_o[1]==
        b_min_ids_1[1] || min_element_id_idx_o[1]==b_min_ids_2[1] ||
        min_element_id_idx_o[1]==b_min_ids_3[1] ) || !b_valid_mins[1] ) &&
347 ( ( min_element_id_idx_o[2]==b_min_ids[2] || min_element_id_idx_o[2]==
        b_min_ids_1[2] || min_element_id_idx_o[2]==b_min_ids_2[2] ||
        min_element_id_idx_o[2]==b_min_ids_3[2] ) || !b_valid_mins[2] ) &&
348 ( ( min_element_id_idx_o[3]==b_min_ids[3] || min_element_id_idx_o[3]==
        b_min_ids_1[3] || min_element_id_idx_o[3]==b_min_ids_2[3] ||
        min_element_id_idx_o[3]==b_min_ids_3[3] ) || !b_valid_mins[3] ) &&
349 ( ( min_element_id_idx_o[4]==b_min_ids[4] || min_element_id_idx_o[4]==
        b_min_ids_1[4] || min_element_id_idx_o[4]==b_min_ids_2[4] ||
        min_element_id_idx_o[4]==b_min_ids_3[4] ) || !b_valid_mins[4] ) &&
350 ( ( min_element_id_idx_o[5]==b_min_ids[5] || min_element_id_idx_o[5]==
        b_min_ids_1[5] || min_element_id_idx_o[5]==b_min_ids_2[5] ||
        min_element_id_idx_o[5]==b_min_ids_3[5] ) || !b_valid_mins[5] ) &&
351 ( ( min_element_id_idx_o[6]==b_min_ids[6] || min_element_id_idx_o[6]==
        b_min_ids_1[6] || min_element_id_idx_o[6]==b_min_ids_2[6] ||
        min_element_id_idx_o[6]==b_min_ids_3[6] ) || !b_valid_mins[6] ) &&
352 ( ( min_element_id_idx_o[7]==b_min_ids[7] || min_element_id_idx_o[7]==
        b_min_ids_1[7] || min_element_id_idx_o[7]==b_min_ids_2[7] ||
        min_element_id_idx_o[7]==b_min_ids_3[7] ) || !b_valid_mins[7] )
353 ) ;
354         endproperty : p_min_element_id_idx_o
355
356 `endif
357
358 // 1.2.1 vp
```

```
359 property p_rsn_o;
360         @(posedge clk_i)
361         !rsn_i |-> ##1 !load_dvalid_o && mask_o == '0;
362 endproperty : p_rsn_o
363
364 // 1.2.2 vp
365 property p_rsn_i;
366         @(posedge clk_i)
367         !rsn_i |-> !load_data_valid_i && !load_granted_i;
368 endproperty : p_rsn_i
369
370
371 ////////////////////////////////////////////////////
372 ////////////////////assert & assume/////////////////////
373 ////////////////////////////////////////////////////
374
375 // 1.0.1 vp
376 a_el_count :                    assume property( disable iff(!rsn_i || kill_i)
        p_el_count ) else $error("Avispado sent a EL_COUNT greater than allowed");
377
378 // 1.0.2 vp
379 a_stride_i :                    assume property( disable iff(!rsn_i || kill_i)
        p_stride_i ) else $error("Avispado sent an invalide stride");
380
381 // 1.0.3 vp
382 a_load_granted_i_known :        assume property( disable iff(!rsn_i || kill_i)
        p_load_granted_i_known ) else $error("MQ sent unknown sew or sb_id");
383
384 // 1.0.4 vp
385 a_load_data_i_known :           assume property( disable iff(!rsn_i || kill_i)
        p_load_data_i_known ) else $error("MQ sent unknown sew or sb_id");
386
387 // 1.0.5 vp
388 a_sb_id_o :                     assert property( disable iff(!rsn_i || kill_i)
        p_sb_id_o ) else $error("wrong sb_id in output");
389
390 // 1.0.6 vp
391 a_load_dvalid_known :           assert property( disable iff(!rsn_i || kill_i)
        p_load_dvalid_known ) else $error("an LMU output is unknown");
392
393 // 1.0.7 vp
394 a_dvalid_o :                    assert property( disable iff(!rsn_i || kill_i)
        p_dvalid_o ) else $error("LMU did not compute any output");
395
396 // 1.0.8 vp
397 a_vstart_o :                    assert property( disable iff(!rsn_i || kill_i)
        p_vstart_o ) else $error("wrong vstart_*");
398
399 // 1.0.9 vp
400 a_granted_ids :                 assume property( disable iff(!rsn_i || kill_i)
        p_granted_ids ) else $error("unallowed sb_id_i to the LMU");
```

```
401
402 // 1.0.10 vp
403 a_indexed_instr :               assume property( disable iff(!rsn_i || kill_i)
        p_indexed_instr) else $error("issued an indexed with el_count!=1 or with
        mask");
404
405 // 1.0.11 vp
406 a_load_sync_end_i :             assume property( disable iff(!rsn_i || kill_i)
        p_load_sync_end_i) else $error("it didn't arrive a load_sync_end_i that was
         supposed to");
407
408 // 1.0.12 vp
409 a_load_grant_beh :              assume property( disable iff(!rsn_i || kill_i)
        p_load_grant_beh) else $error("issued 3 load_granted_i to the LMU without
        load_sync_end");
410
411 // 1.0.13 vp
412 a_load_sync_end_beh :           assume property( disable iff(!rsn_i || kill_i)
        p_load_sync_end_beh) else $error("ended a load not actually issued");
413
414 // 1.0.14 vp
415 a_load_req_sync_end :           assume property( disable iff(!rsn_i || kill_i)
        p_load_req_sync_end) else $error("issued a new instruction simultaneously
        with a sync_end");
416
417 // 1.0.15 vp
418 a_el_count_sew_i :              assume property( disable iff(!rsn_i || kill_i)
        p_el_count_sew_i ) else $error("Avispado sent a EL_COUNT!=1 with a stride
        !=+-1/2/3/4*sew");
419
420
421 `ifdef FUNCTIONAL
422
423         // 1.1.1 vp
424         a_sb_correct :                  assert property( disable iff(!rsn_i ||
                kill_i) p_sb_correct ) else $error("LMU sent a output as dvalid_o =
                 1, for a non-valid load");
425
426         // 1.1.2 vp
427         a_data_o :                      assert property( disable iff(!rsn_i ||
                kill_i) p_data_o ) else $error("wrong load_data_o");
428
429         // 1.1.3 vp
430         a_mask_o :                      assert property( disable iff(!rsn_i ||
                kill_i) p_mask_o ) else $error("wrong mask_o");
431
432         // 1.1.4 vp
433         a_el_ids_o :                    assert property( disable iff(!rsn_i ||
                kill_i) p_el_ids_o ) else $error("mismatch in element_ids_o");
434
435         // 1.1.5 vp
```

```
436         a_min_element_id_idx_o :        assert property( disable iff(!rsn_i ||
                kill_i) p_min_element_id_idx_o) else $error("wrong
                min_element_id_idx_o");
437
438 `endif
439
440 // 1.2.1 vp
441 a_rsn_o :                               assert property(p_rsn_o) else $error("
        rsn did not work");
442
443 // 1.2.2 vp
444 a_rsn_i :                               assume property(p_rsn_i) else $error("
        rsn input");
445
446 ////////////////////////////////////////////////////////
447 ////////////////////// tasks ///////////////////////////
448 ////////////////////////////////////////////////////////
449 `ifdef FUNCTIONAL
450
451         task check_sb_correct();
452
453
454                 if ( seq_id_i[32:29] == fifo[0].sb_id && !fifo[0].done ) begin
455                         SEW = 2**(3+fifo[0].SEW) ;
456                         STRIDE = fifo[0].STRIDE;
457                         is_indexed = fifo[0].is_indexed;
458                         sb_correct = 1;
459                 end
460                 else if ( seq_id_i[32:29] == fifo[1].sb_id && !fifo[1].done )
                        begin
461                         SEW = 2**(3+fifo[1].SEW) ;
462                         STRIDE = fifo[1].STRIDE;
463                         is_indexed = fifo[1].is_indexed;
464                         sb_correct = 1;
465                 end
466                 else begin
467                         sb_correct = 0;
468                 end
469                 if (sb_correct) begin
470                         EL_COUNT = seq_id_i[28:22];
471                         OFFSET = seq_id_i[21:16];
472                         EL_ID = seq_id_i[15:5];
473                         N_ELEMENTS = MEM_DATA_WIDTH/SEW;
474
475                         compute_data_o(SEW,STRIDE,EL_COUNT,OFFSET,EL_ID,
                                N_ELEMENTS,is_indexed);
476                         compute_mask_o(SEW,EL_COUNT,EL_ID,N_ELEMENTS);
477                         compute_ids(SEW,EL_COUNT,EL_ID,N_ELEMENTS);
478                         compute_c_min_element_id_idx(SEW,EL_COUNT,EL_ID,
                                N_ELEMENTS);
479
```

```
480                 end
481
482
483         endtask : check_sb_correct
484
485         task compute_data_o(int unsigned SEW, longint signed STRIDE, int
                unsigned EL_COUNT, int unsigned OFFSET, int unsigned EL_ID, int
                unsigned N_ELEMENTS, bit is_indexed);
486                 bit x[][];
487                 bit y[][];
488                 int unsigned j;
489                 int unsigned k;
490                 longint signed local_stride;
491
492                 local_stride=STRIDE;
493
494                 // step zero: sample the input, one byte at a time (
                        SystemVerilog requires a constant width)
495                 x=new[N_ELEMENTS];
496                 y=new[N_ELEMENTS];
497
498                 foreach(x[i]) begin
499                         x[i]=new[SEW];
500                         y[i]=new[SEW];
501                 end
502
503
504                 for(k=0; k < N_ELEMENTS; k++) begin
505                         for(j=0;j < SEW; j++) begin
506                                 x[N_ELEMENTS-1-k][SEW-j-1] = load_data_i[k*SEW+
                                        j];
507                         end
508                 end
509
510                 x_display0 = {>>{x}}; //debug signal
511
512                 // first step: take the stride into account
513
514                 if(!is_indexed && local_stride<0) begin
515                         foreach(x[i]) begin //if it's strided with negative
                                stride, reverse the input
516                                 y[N_ELEMENTS-1-i]=x[i];
517                         end
518                 end
519                 else if (is_indexed || (!is_indexed && local_stride>=0) ) begin
520                         foreach(x[i]) begin //if it's indexed or strided with
                                positive stride, do nothing
521                                 y[i]=x[i];
522                         end
523                 end
524
```

```
525                 x_display1 = {>>{y}}; //debug signal
526
527                 if (local_stride<0) local_stride = ( -1 ) * local_stride;
528                 local_stride = local_stride * 8 / SEW; // normalize from #of
                        bytes to # of elements
529
530
531                 // second step: concatenate the data according to stride
532                 // mask is not taken into account as there is no reason to out
                        the output bits to 0: what we compare are the two outputs
                        in AND with the mask_o
533                 // initialize x to 0
534                 for(j=0;j<N_ELEMENTS;j++) begin
535                         for(k=0;k<SEW;k++) begin
536                                 x[j][k] = 0;
537                         end
538                 end
539
540
541                 for (j=0; j<EL_COUNT; j++) begin
542                                 x[(N_ELEMENTS-1-j)] = y[(N_ELEMENTS-1-j*
                                        local_stride-OFFSET)%N_ELEMENTS];
543                 end
544
545                  x_display2 = {>>{x}}; //debug signal
546
547                 // third step: shift the data according to EL_ID
548                 k = EL_ID % N_ELEMENTS;
549                 for (j=0; j<N_ELEMENTS; j++) begin
550                         y[(N_ELEMENTS-1-(k+j))%N_ELEMENTS] = x[(N_ELEMENTS-1-j)
                                ]; // "%n_elements" automatically manages the
                                underflow
551                 end
552
553                 /* we do the iterazion on all y to clean it from the old values
                         computed in step 1, otherwise we could do it on only
                        EL_COUNT but with a further previous initialization to 0*/
554
555                 computed_data_o = {>>{y}};
556
557
558         endtask : compute_data_o
559
560
561         task compute_mask_o(int unsigned SEW, int unsigned EL_COUNT, int
                unsigned EL_ID, int unsigned N_ELEMENTS);
562
563                 bit m_op [][];
564                 int unsigned m, n, k;
565
566                 m_op = new[N_ELEMENTS];
```

```
567
568                 foreach(m_op[k])
569                         m_op[k] = new[AVISPADO_LOAD_MASK_WIDTH/N_ELEMENTS];
570
571                 //init to 0
572                 for(m = 0; m < N_ELEMENTS; m++) begin
573                         for(n = 0; n < AVISPADO_LOAD_MASK_WIDTH/N_ELEMENTS; n
                            ++)
574                             m_op[m][n] = 1'b0;
575                 end
576
577                 //assignment valid values to 1
578                 k = 0;
579                 for(m = EL_ID; m < (EL_ID + EL_COUNT); m++) begin
580                         if ( !mask_valid_i || (mask_valid_i && mask_i[k]) )
                            begin
581                             for(n = 0; n < AVISPADO_LOAD_MASK_WIDTH/
                                N_ELEMENTS; n++) begin
582                                 m_op[N_ELEMENTS-1-(m%(N_ELEMENTS))][n]
                                    = 1'b1;
583                             end
584                         end
585                         k++;
586                 end
587
588                 m_result = {>>{m_op}};
589
590         endtask : compute_mask_o
591
592
593         task compute_ids(int unsigned SEW, int unsigned EL_COUNT, int unsigned
                EL_ID, int unsigned N_ELEMENTS);
594
595                 int unsigned f_v_e, i, j;
596
597                 // f_v_e is the index of the first 11-bit-element of
                        computed_el_id_o where we have to put an EL_ID
598                 f_v_e = EL_ID % N_ELEMENTS; // find the first valid element
599                 f_v_e = f_v_e * SEW / 8 ; // multiply it by the pace (number of
                         bytes per element)
600
601                 // initialize to 0
602                 for(j=0;j<MAX_NUMBER_ELEMENTS;j++) begin
603                         computed_el_id_o[j] = '0;
604                 end
605
606                 for(j=0;j<EL_COUNT;j++) begin
607                         for(i=0;i<SEW/8;i++) begin
608                             computed_el_id_o[(f_v_e + i + (j*SEW/8))%
                                MAX_NUMBER_ELEMENTS] = EL_ID + j;
609                         end
```

```
610             end
611
612         endtask : compute_ids
613
614
615         task compute_big_mask_o();
616             int j, k;
617             bit bm[][];
618
619
620             bm=new[MEM_DATA_WIDTH/8];
621
622             foreach(bm[i]) begin
623                 bm[i]=new[8];
624             end
625
626             //init to 0
627             for(k=0; k < MEM_DATA_WIDTH/8; k++) begin
628                 for(j=0;j < 8; j++) begin
629                     bm[k][j] = 0;
630                 end
631             end
632
633
634             //generate the mask at 512 bit
635             for(k=0; k < MEM_DATA_WIDTH/8; k++) begin
636                 if(mask_o[k]) begin
637                     for(j=0; j < 8; j++) begin
638                         bm[MEM_DATA_WIDTH/8 -k -1][j] = 1;
639                     end
640                 end
641             end
642
643             big_mask = {>>{bm}};
644
645         endtask: compute_big_mask_o
646
647
648         task compute_c_min_element_id_idx(int unsigned SEW, int unsigned
                EL_COUNT, int unsigned EL_ID, int unsigned N_ELEMENTS);
649
650             bit m_op [][];
651             int unsigned m, n, k, min_el_id, f_v_e;
652             int j,i;
653
654             // computing the mask, as if the instruction was not masked
655             f_v_e = EL_ID % N_ELEMENTS; // find the first valid element
656             f_v_e = f_v_e * SEW / 8 ; // multiply it by the pace (number of
                    bytes per element)
657
658             for(j=0;j<MAX_NUMBER_ELEMENTS;j++) begin // initialize to 0
```

```
659                     valid_bytes[j] = '0;
660             end
661
662             for(j=0;j<EL_COUNT;j++) begin // put to 1 the valid bits
663                     for(i=0;i<SEW/8;i++) begin
664                             valid_bytes[(f_v_e + i + (j*SEW/8))%
                                    MAX_NUMBER_ELEMENTS] = 1'b1;
665                     end
666             end
667
668             // computing min_el_id_idx
669             foreach(valid_mins[i]) valid_mins[i]=0;
670             foreach(c_min_element_id_idx[i]) c_min_element_id_idx[i]=0;
671
672             for(j=0;j<N_LANES;j++) begin
673                     c_min_element_id_idx[j]=0;
674                     min_el_id = 2057;
675                     for(i=0;i<8;i++) begin
676                             if(valid_bytes[j*8+i]) begin
677                                     if (computed_el_id_o[j*8+i]<min_el_id)
                                            begin
678                                             min_el_id=computed_el_id_o[j*8+
                                                    i];
679                                             c_min_element_id_idx[j]=i;
680                                     end
681                                     valid_mins[j]=1;
682                             end
683                     end
684             end
685
686
687             if(SEW==32) begin
688                     for(j=0;j<8;j++) begin
689                             c_min_element_id_idx_1[j]=c_min_element_id_idx[
                                    j]+1;
690                             c_min_element_id_idx_2[j]=c_min_element_id_idx[
                                    j]+2;
691                             c_min_element_id_idx_3[j]=c_min_element_id_idx[
                                    j]+3;
692                     end
693             end
694
695             if(SEW==16) begin
696                     for(j=0;j<8;j++) begin
697                             c_min_element_id_idx_1[j]=c_min_element_id_idx[
                                    j]+1;
698                             c_min_element_id_idx_2[j]=c_min_element_id_idx[
                                    j];
699                             c_min_element_id_idx_3[j]=c_min_element_id_idx[
                                    j]+1;
700                     end
```

```
701                 end
702
703                 if(SEW==8) begin
704                         for(j=0;j<8;j++) begin
705                                 c_min_element_id_idx_1[j]=c_min_element_id_idx[
                                        j];
706                                 c_min_element_id_idx_2[j]=c_min_element_id_idx[
                                        j];
707                                 c_min_element_id_idx_3[j]=c_min_element_id_idx[
                                        j];
708                         end
709                 end
710
711         endtask : compute_c_min_element_id_idx
712
713 `endif
714
715 endmodule: load_management_unit_checker
```