Functional Verification of a RISC-V Vector Accelerator

Victor Jimenez*, Mario Rodriguez†, Marc Dominguez†, Josep Sans*, Ivan Diaz‡, Luca Valente§, Vito Luca Guglielmi¶, Josue Quiroga‡, Roberto Genovese‡, Nehir Sonmez‡, Oscar Palomar‡ and Miquel Moreto‡
*name.surname@semidynamics.com †name.surname@codasip.com ‡name.surname@bsc.es §name.surname@unibo.it
¶vitoluca95guglielmi@gmail.com

Validation Team
Computer Sciences Department
Barcelona Supercomputing Center

***Abstract*—We present the functional verification efforts for an academic RISC-V based vector accelerator, successfully taped-out in the context of the European Processor Initiative. For our novel RISC-V based decoupled vector accelerator, we built a verification infrastructure consisting of a UVM environment, performing step by step co-simulation of all vector instructions, using the Spike instruction set simulator as a reference model. Furthermore, for validating this complex design connected to a scalar core using a custom interface, we provided automated constrained-random test generation, simulation and error reporting, and CI/CD infrastructure. We found 3005 errors during this process and reached 95.79% functional coverage.**

***Index Terms*—verification, RISC-V, vector accelerator, UVM, coverage, random binary generation**

1. Introduction

Many open source and research hardware projects have emerged in the past decade, in which the main objective was to tape out an entire processing system [8, 9, 4]. To this end, a significant effort in design verification must be made in order to avoid fabricating a prone-to-fail design. However, academic designs are typically not verified at the industrial-grade, often due to a lack of resources and experience, and different needs than the industry. Meanwhile, open-source ISAs such as RISC-V favor collaboration between research and industrial entities, also providing independence from non-European computing technologies.

The European Processor Initiative (EPI)[1] is a project that embraces this idea, being conceived to create the first European processor and accelerators. Many partners are involved in its development, for example, BSC developed the Vector Accelerator that will be directly connected to a scalar RISC-V core designed by SemiDynamics, while the top-level integration of the test chip is done by EXTOLL and the tape out is coordinated by Fraunhofer.

RISC-V is an open-source *Instruction Set Architecture* (*ISA*) [13], which among others, has a vector extension, currently in version 1.0 [3]. This extension includes the vectorized version of many arithmetic, logical and memory instructions, along with vector-specific instructions such as reductions, and scatter and gather operations. Additionally, the *RISC-V Vector extension* (*RVV*) is vector length agnostic and supports different element widths via dedicated configuration registers. Our main goal in this project was to verify our novel, decoupled vector accelerator functionally, which implemented version 0.7.1 of the *RVV* and was connected to the scalar processor core via the ***Open Vector Interface* (*OVI*)** [12].

With RISC-V many groups have appeared in the open source world contributing to the community with their projects. Groups like *OpenHW* have designed and developed verification environments [10] for many designs as *Parallel Ultra Low Power* (*PULP*) designs such as *RI5CY*, *Ariane* and *Ibex*.

The main contributions of this paper are:

- Description of an industrial grade verification approach, with *UVM* testbench, reference model, assertions and coverage, for a modern RISC-V vector accelerator.
- Implementation of a common UVM testbench for a novel interface design and a large-scale RTL project.
- Result comparison of each completed vector instruction against the reference model via co-simulation of constrained-random binaries and C programs.
- Automated testing and regression infrastructure to reach high levels of functional/code coverage, of up to 95.79%.

2. Background

The *Vector Processing Unit* (*VPU*) is based on ISA Vector extension 0.7.1v [2], has eight vector lanes, supporting large vectors of up to a maximum vector length of 256 elements of 64 bits each (16Kb total). It has 32 logical and 40 physical vector registers. Each lane has one Fused Multiply Accumulate (FMA) unit capable of calculating two double-precision operations per cycle, for a total maximum throughput of 16 DFlops/cycle. It supports 64 and 32-bit floating-point vector operations, as well as 64, 32, 16 and 8-bit integer vector operations. Memory operations have limited out of order capability, mostly between arithmetic and memory operations.

The *VPU* has all eight vector lanes connected to the memory operation units, the inter-lane ring and the instruction queues, which serialize the instructions arriving from the scalar core to which the *VPU* is connected (Figure 1). The scalar core is in charge of executing scalar instructions and sending the vector instructions to the *VPU*. Memory accesses for the vector memory operations are also performed by the core, through *OVI*.

This work has been done by the authors in the time they were affiliated to Barcelona Supercomputer Center.

[1]https://www.european-processor-initiative.eu/accelerator/

 DOI 10.1109/MDAT.2022.3226709

This article has been accepted for publication in IEEE Design & Test. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2022.3226709

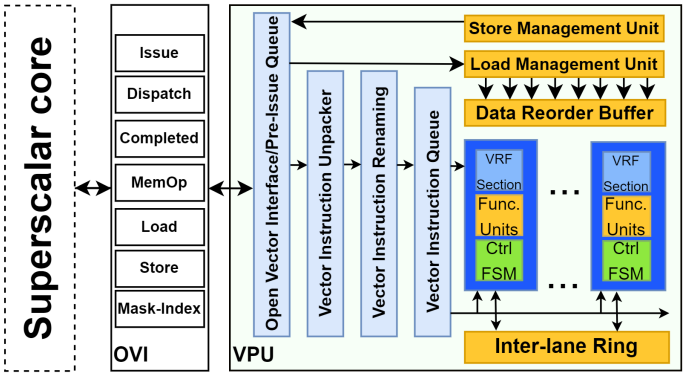

Fig. 1. *VPU* is connected to the Scalar Core through *OVI*.

OVI contains the following sub-interfaces:

- ISSUE: through which the core sends the request along with the instruction, configuration values and scalar input.
- DISPATCH: all issued instructions are either confirmed or killed. This enables speculative issue of vector instructions.
- COMPLETED: thorugh which the *VPU* notifies the instruction has been completed, together with metadata and scalar output.
- MEMOP: start and finish signals of a memory operation are sent using this interface.
- LOAD: used by the core to send the load data and metadata.
- STORE: *VPU* sends the data of store operations to the core through this interface.
- MASK-INDEX: through which the *VPU* sends the vector content to generate the addresses of masked and indexed memory instructions.

At each vector instruction, many sub-interfaces must be considered given that some of them can change the instruction's behavior or should be looked to retrieve results. ISSUE, STORE and MASK-INDEX interfaces use a credit system for handshaking between the *VPU* and the scalar core.

The *VPU* was developed in collaboration with another partner, UniZagreb, which was in charge of developing the Floating Point Unit submodule and its verification.

3. Design Verification Methodology

We constructed a set of tools and utilities around the *design under test* (*DUT*) that facilitated the detection of errors. The tools we developed had to be easy to share with partners as some verification efforts are shared and reusable for the next-generation designs. To meet these requirements, we used the *Universal Verification Methodology* (*UVM*) [1], which is built under the premises of creating a modular, scalable and reusable verification environment.

At first, we considered verifying individually each *VPU* submodule, stimulating them with constrained-random techniques with several *UVM* environments. As this approach implied an unbearable amount of effort for our team and the final specifications were not ready for all the submodules, we decided to focus at the interface level (*OVI*) that already had well-defined specifications.

Once we built the *UVM* that drives instructions to the *DUT*, we needed a way to evaluate the results of the *VPU*. For that purpose, we used a *UVM* scoreboard that compares these results with the ones from the reference model, a software that predicts how the design should behave based on the inputs. Our reference model accepts instructions as an input and generates the expected results. We decided to use the *RISC-V ISA* simulator *Spike* [11] for co-simulation in our *UVM* environment.

Even if detecting a mismatch in the result of an instruction is crucial for our job, it may not point out the cause of the error. We also added *SystemVerilog* assertions to improve observability.

4. Design Verification Infrastructure

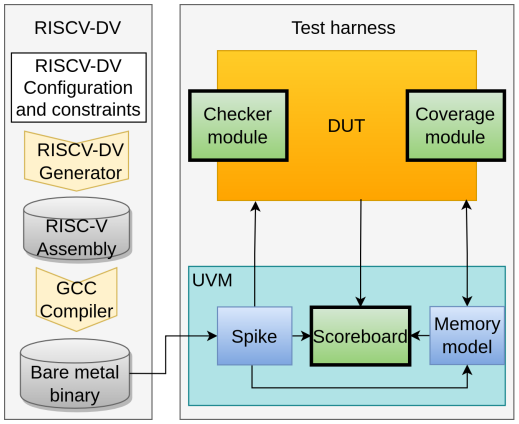

Fig. 2. Verification Environment Overview

The verification environment we implemented is shown in Figure 2 and explained in the following sub-sections.

A. UVM

Our environment is composed of the *UVM* top module, which instantiates the *UVM* environment. As we have different semi-independent sub-interfaces, we created one agent for each specific sub-interface. For example, at the issue sub-interface, there is an agent, which contains a sequencer, a driver and a monitor connected to the virtual interface.

Each virtual sequence creates interface-specific transactions that are sent to the corresponding interface. When the driver gets the transaction, it stimulates the corresponding sub-interface with the incoming transaction values. As the virtual sequence does not know when the transaction is driven, we also have a specific monitor that captures the interface state and sends it back to the virtual sequence through the sequencer. The corresponding virtual sequence gets the transaction and reacts to it, producing a new stimulus [7].

All seven sub-interfaces are unique in the environment and constantly communicate with each other (e.g., masked to load and issue to dispatch sub-interfaces). To keep them in synchronization, we use *UVM* events, which are capable of transmitting

This article has been accepted for publication in IEEE Design & Test. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2022.3226709

data along with the event trigger. This feature eased the virtual sequences inter-communication.

The high degree of dependence between the sub-interfaces complicated the constrained-random stimulus generation. Therefore, we decided to randomize only the instructions fed to the issue sub-interface and made all the other sub-interfaces react according to the instructions driven.

B. RISCV-DV

RISCV-DV [6] is a *SystemVerilog/UVM* based open-source instruction generator for *RISC-V* processor verification developed by Google. *RISCV-DV* generates random *RISC-V* assembly tests, which we used to provide vector instructions to test the *VPU*. *RISCV-DV* implemented a later *RVV* than 0.7.1, so we developed and adapted the parts we needed to fit.

The major additions we did to *RISCV-DV* were:

- Generation of *vsetvli* instructions through the code and modification of the generation of memory operations to allow the change of element width and vector length.
- An option to select the initialization pattern of the data pages.
- Constraining of the memory addresses accessed by the test to avoid memory exceptions, specially for vector memory indexed instructions.
- Adapting to the 0.7.1 *RVV*.

Additionally, since some of the design modules were in development for most of the verification process, we had to initially blacklist many of the instructions from our generated tests, to get functional tests at each iteration. When a significant number of errors were fixed, we gradually removed instructions from the blacklist, until all implemented instructions were enabled.

C. Spike

In our environment, *Spike* has two main roles: 1) As a scalar core, executing scalar instructions and providing the vector ones to the *UVM* in program order, and 2) As a golden/reference model to check the correctness of the *DUT* results.

To fulfill these two functions, we performed several modifications to *Spike*:

- Definition of functions to call *Spike* in *SystemVerilog* using *Direct Programming Interface* (*DPI*).
- Creation of a method that resumes the simulation until a vector instruction is executed, the reference results are returned to the *UVM* to compare against *VPU* results.
- Functions to read from *Spike*'s memory.
- A function to force the result of reductions into *Spike* to avoid execution divergence in unordered floating-point reductions.

When a vector instruction is found, *Spike* provides the instruction, the results and other relevant data to the *UVM*. The instruction is then packed as a transaction and sent to the issue agent. It arrives at the *VPU*, it's executed, and the reference model results are compared to those generated by the *VPU*. Also, some changes were done to accommodate *Spike* to the 0.7.1 *RISC-V* vector specification:

- The implementation of the vector tail zeroing, replaced by a different policy after version 0.7.1.
- Instruction decoding to follow the 0.7.1 specification.
- The requirements of *Vector Context Status* (*VCS*) fields in *mstatus*.

Once the instruction is fed using the issue agent, the *UVM* follows the *VPU* instruction execution flow. This involves the stimulation/observation of two interfaces: a) DISPATCH where we must confirm or discard the execution of each instruction sending this information in instruction order, and b) COMPLETED at the end of execution of a confirmed instruction, the completed monitor will observe a flag being set and will create a transaction.

With this *UVM* setup, we could run simple instructions, which helped in the first stages of the verification process to see that our design was not stalling. However, this doesn't assert the result or the execution of the instruction went well. So, we introduced a *UVM* component that checks the correctness of the results, the scoreboard.

D. Scoreboard

It's connected to the completed monitor and when an instruction finishes, a method that compares both results is executed. The issue monitor would send the transaction to the scoreboard and the reference model, but we directly take the information coming from *Spike*, as we need it to feed the instruction to the *VPU*.

The most interesting results from instruction execution are not always seen as outputs of the *VPU*. In *OVI*, the COMPLETED sub-interface includes a scalar output and some flags. In the general case, the *VPU* will write the result of the instruction in one physical vector register. At instruction completion time, the registers are accessed to get the result of the instruction.

To check whether these results are correct or not, we include the destination vector register value in the information that we extract from *Spike*.

One particular case that we found is with reduction instructions, more specifically, the floating-point ones. The *VPU* uses a different reduction algorithm than *Spike*, which is allowed by the *RVV* specification. This situation caused two problems; to begin with, we got a mismatch sometimes when executing these instructions when they were actually correct according to the rounding mode and algorithm used. The second problem was that, even if we knew that the mismatch had been a false positive, the result remained wrong in the *Spike* vector registers. These values could later be used in other instructions and cause mismatches even if the instruction was executed correctly. We have created an independent reference model in C for the unordered reductions that implements the same exact reduction algorithm as the DUT. The VPU result is compared in these cases against the reduction reference model instead of spike, and if there is a match, the value is injected into the register in spike.

E. Memory Operations

Memory operations are one of the most delicate parts of our design. The *VPU* does not have direct access to memory, so it reads and writes data through the scalar core using the memop, load, store and mask interfaces, which require plenty of inter sub-interface communication.

For load operations, we need the data inside memory before the instruction is executed. Therefore, along with the rest of the instruction information from *Spike*, we get the data that should be read. This data is written in a memory model, based on the one from the *OpenTitan* project [9], which is accessed in the corresponding addresses, and the *Spike* data is sent through the load sub-interface of the *VPU*.

For store operations, we need the data in memory before the instruction executes to check masked operations, to detect undesired writes. In a store operation, the *VPU* sends the data, which is stored in the memory model. Later on, these values will be read and compared with the *Spike* ones.

Masked memory operations involve particular outgoing transactions from the VPU, which sends the masks or indexes. These are needed to execute the instruction on the environment side and compare with the ones in *Spike*. This comparison helped to detect the origin of the issue in many of the memory instructions errors that were mask related.

One interesting case that *OVI* presents is what we call *retries*. Retries occur when the *VPU* cannot handle all the loaded cache lines the scalar core sent. If this happens, the instruction will complete specifying a *vstart* value,representing the first element that it could not write in the vector registers, indicating that the instruction did not finish and that it must be re-executed. This implies killing the instructions after the retrying one and re-issuing the instruction starting from the element corresponding to the stored *vstart* value. Additionally, memory exceptions for multiple in-flight loads and stores, were randomized, causing failing instructions to be killed. An added complexity was calculating the correct *vstart* value, deciding between the lowest received index, mask or data chunk.

Retries implied plenty of changes and were one of the primary sources of errors in the *VPU*. We wanted to have the possibility to increase the chance of causing retries to the *VPU* which was randomized using *UVM* configuration objects.

F. Assertions

One of the critical points of the *VPU* is the interface, so we decided to run down the *OVI* specifications and write System Verilog Assertions (*SVA*) [5] that check that it is behaving as expected, implementing more than 50. At the early stages of the *UVM* testbench development, they helped to identify bugs in the *VPU*, as well as problems in the *UVM* stimulation. Most of the asserted properties were targeted to the memory-related sub-interfaces and ensured that the *OVI* specifications were strictly followed at any point of the project.

G. Coverage

We defined and implemented a functional coverage plan. We mostly checked things that could be directly observed in the *VPU* interface, like instructions, execution parameters and values in the memory sub-interfaces. This way, we could ensure that we executed all the instructions in all the possible ways in the Accelerator. Furthermore, we gathered coverage metrics of certain internal modules.

For instructions coverage and their different formats in Spec 0.7.1v [2] (Vector Length, Single Element Width, Rounding Modes, Masks, etc.) we developed a set of of ISA tests, that quickly tested the key configurations, in parallel with RISCV-DV random tests for further stress. We also implemented functional coverage for testing diverse loads and stores scenarios, not only for Vector Length, but also considering the organization of the register file. Loads and their possible retries, as explained in Section E, with different vstart values were covered by directed tests that were added to the regressions suite in order to check these scenarios were still valid on every RTL change.

In addition to functional coverage, we recorded assertions usage (active/passed) and code coverage of the simulations run by continuous integration, which was used to generate and run tests, and to collect coverage metrics.

H. CI Infrastructure

Our Continuous Integration (*CI*) infrastructure is built using the open-source *CI* server *Jenkins*, where we created a set of pipelines that interact with each other to have the most error-free design possible.

We implemented the following pipelines:

1) New tests: Generates random tests with *RISCV-DV*, compiles the *DUT*, executes the binaries and does a classification between passed and failed tests, separated later into two directories. The first ones are used to create a regression set and the others are kept for debugging and checking until the error is fixed.
2) Retry: For each change in the main branch of the *DUT* repository, the set of failed tests is re-executed and classified again in passed and failed.
3) Selection: Every day at midnight, if the number of tests classified as passed is above a certain threshold, tests are ranked by the collected coverage and we create two sets of regressions, a large one, and a small one.
4) Regressions: When there is a change in the *DUT*, which is candidate to be merged, we execute the small set of regressions to check the correctness of these changes. Also, once per week, the large set is executed to ensure that recent changes do not break known-good tests.

We used GitLab for version control and as a way to track down issues in our environment. We also documented the code and all these features so anyone in the project could run a simulation and added this documentation as guides and tutorials using the Wiki feature from GitLab.

5. Experimental Results

This environment was in use for about a year. In this time, we have managed to find many errors and provide helpful information to the *RTL* team to debug these. Apart from that, we have provided *CI* pipelines that allowed both teams (RTL design and verification) to test new features and find new errors.

This article has been accepted for publication in IEEE Design & Test. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2022.3226709

The errors we have encountered in the *VPU* include design issues and instruction mismatches. Some of these were caused by specification problems. When an error was found, the necessary information to reproduce it (e.g. binary, faulty instruction) was provided. Furthermore, a table summarizing the active errors was used to help focussed debugging effort (e.g. detecting multiple erroneous tests with the same instruction mnemonic, vector length, element width). Once the cause of the error was tentatively fixed, regressions were run before the changes could be merged.

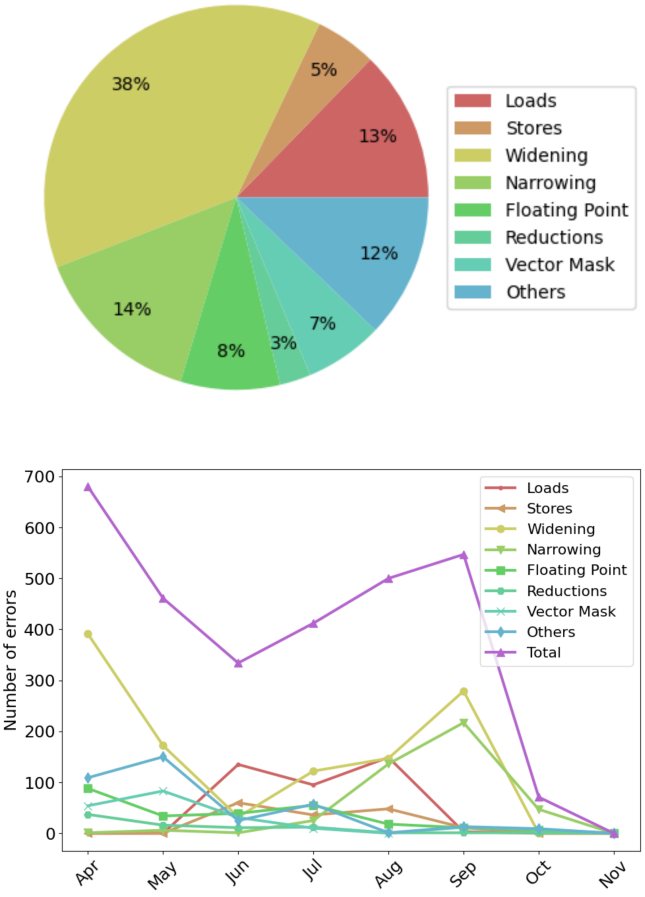

Fig. 3. Errors and number found per month

In Figure 3, the faulting instruction distribution between the random tests that failed in our daily simulations is shown in the pie chart. We can see the type of instructions that failed the most: memory, narrowing and widening vector instructions. We ran 24 tests every night between April and July, which were increased to 50 tests between August and the end of November which marked the *RTL* freeze before the chip tape out. Each of these tests contained approximately 500 vector instructions, we managed to find a total of 3005 errors, and the instructions above constitute around 70% of them.

In Figure 3, we can see the number of errors per month. Around April, when we first set up the environment and the *CI* pipeline, we had many errors. In that starting phase, the environment was not complete and the design had many issues, causing different kinds of instructions to fail. Additionally, some instructions were not implemented and had to be blacklisted, which artificially made the number of errors decrease during this phase.

Once the *RTL* team fixed more errors and finished implementing the missing features, we started whitelisting instructions, which caused an increase of errors found between June and September. During this period, we also managed to execute several vectorized micro-benchmarks like *SpMV*, *Matmul* or *axpy*. While errors increased temporarily, we were also fixing more of them, and this decrease can be seen at the final phase of the plot.

At the end of this testing period, which we called "Night runs", we were not getting any more errors. This was a huge advancement, but we wanted to provide more tests, so we developed a new set of testing pipelines. These combined ran around 600 tests every day. We used them for collecting coverage numbers and finding bugs, especially since the development of the *VPU* still continued with new features.

TABLE I
FUNCTIONAL COVERAGE PER DESIGN UNIT

Design Unit	Coverage	Design Unit	Coverage
OVI/Pre-issue Queue	**91.95%**	Data Reorder Buffer	**88.09%**
Instruction Unpacker	**100.00%**	Ctrl FSM	**92.64%**
Instruction Renaming	**100.00%**	Functional Units	**100.00%**
Instruction Queue	**100.00%**	Vector Register File	**92.37%**
Store Management	**87.50%**	Inter-lane Ring	**99.18%**
Load Management	**100.00%**	Vector Lane	**93.61%**
Item/Mask Management	**100.00%**		

Table I summarizes the functional coverage achieved, for an average of 95.79%. Regarding code coverage (average of 72.64%) we have achieved 90.90% in Statements and a 49.83% in Toggles. We know that we are not driving the design appropriately in some cases, which were difficult to add in the environment, and that is reflected in the coverage numbers. Additionally, the lower code coverage numbers can indicate some unused data structures or conditions in the *RTL*.

6. Conclusions and Related Work

In this paper, we have described the implementation of a verification environment targeting an academic *RISC-V* based vector accelerator, which was successfully taped-out in the context of a European project. The environment is a reusable and extendable *UVM* environment, which implements the protocol between the *OVI* and the *VPU* and checks the correctness of the completed instructions.

Additionally, assertions and coverage were developed to observe better and extract metrics to know how well the verification process was done. Moreover, this environment is complemented by creating random binaries using the *RISCV-DV* generator and the *CI* infrastructure, which plays an essential role in code health/maintainability and coverage closure. Thanks to our automated constrained-random test generation, simulation and error reporting and CI/CD infrastructure, through this process, we found 3005 errors and reached 95.79% of functional coverage.

Regarding the environment, we have learnt that the implementation of the communication divided among several agents complicates the maintenance, extension and performance. A way to cope with these issues would be a single agent that produces the stimulus. Doing all the interface interaction using a single module could simplify the sub-interfaces communication and possible expansions of the design and the environment, which we will follow as future work.

7. Acknowledgements

This research has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 (European Processor Initiative) and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project, PCI2022-132935 is also co-funded by MCIN/AEI /10.13039/501100011033 and by the UE NextGenerationEU/PRTR.

Bibliography

[1] Accelera. *Universal Verification Methodology standard reference*. https://www.accellera.org/downloads/standards/uvm. [Online; accessed October 20, 2022].

[2] Amid, Asanovic, et al. *Specifications - RISC-V Vector extension 0.7.1*. https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1. [Online; accessed October 20, 2022].

[3] Amid, Asanovic, et al. *Specifications - RISC-V Vector extension 1.0*. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0. [Online; accessed October 20, 2022].

[4] Matheus Cavalcante et al. "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI". In: *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 28.2 (2019), pp. 530–543.

[5] Cerny, Eduard, et al. *SVA: The power of assertions*. Springer International Publishing, 2015.

[6] Google. *Random instruction generator RISCV-DV*.

[7] Mark Litterick, Jeff Montesano, and Taruna Reddy. "Mastering Reactive Slaves in UVM". In: SNUG. 2015.

[8] lowRISC. *lowRISC*. https://github.com/lowrisc/. [Online; accessed October 20, 2022].

[9] lowRISC. *OpenTitan project*. https://github.com/lowrisc/opentitan. [Online; accessed October 20, 2022].

[10] OpenHW. *PULP-Platform Simulation Verification*. https://core-v-docs-verif-strat.readthedocs.io/en/latest/pulp_verif.html. [Online; accessed October 20, 2022].

[11] RISC-V. *Spike repository*. https://github.com/riscv-software-src/riscv-isa-sim. [Online; accessed October 20, 2022].

[12] Semidynamics. *Open Vector Interface specifications*. https://github.com/semidynamics/OpenVectorInterface. [Online; accessed October 20, 2022].

[13] A. Waterman and K. Asanović. *Specifications - RISC-V International*. https://riscv.org/technical/specifications/. [Online; accessed October 20, 2022]. 2019.