2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Reusable Verification Environment for a RISC-V Vector Accelerator

Josue Quiroga, Junior Research Engineer, BSC, Barcelona, Spain (*josue.quiroga@bsc.es*)

Roberto Ignacio Genovese, Senior Research Engineer, BSC, Barcelona, Spain *(roberto.genovese@bsc.es*)

Ivan Diaz, Junior Research Engineer, BSC, Barcelona, Spain (*ivan.diaz@bsc.es*)

Henrique Yano, Junior Research Engineer, BSC, Barcelona, Spain (*henrique.yano@bsc.es*)

Asif Ali, Research Engineer, BSC, Barcelona, Spain (*asif.ali@bsc.es*)

Nehir Sonmez*, Established Researcher, BSC, Barcelona, Spain (*nehir.sonmez@bsc.es*)

Oscar Palomar, Established Researcher, BSC, Barcelona, Spain (*oscar.palomar@bsc.es*)

Victor Jimenez+, Junior Verification Engineer, Semidynamics, Barcelona, Spain (*victor.jimenez@semidynamics.com*)

Mario Rodriguez+, Verification Engineer, Codasip, Barcelona, Spain (*mario.rodriguez@codasip.com*)

Marc Dominguez+, Verification Engineer, Codasip, Barcelona, Spain (*marc.dominguez@codasip*.com)

***Abstract*—This paper presents a reusable verification environment developed for the verification of an academic RISC-V based vector accelerator that operates with long vectors. In order to be used across diverse projects, this infrastructure intends to be independent of the interface used for connecting the accelerator to the scalar processor core. We built a verification infrastructure consisting of a Universal Verification Environment (UVM) which is capable of validating the design performing co-simulation of the vector instructions. Moreover, we provided a set of tests and an automated test generation, simulation and error reporting infrastructure. This paper shares our experience on verifying a complex accelerator used in two distinct projects, with different interfaces.**

Keywords—verification; RISC-V; Vector Accelerator; UVM; Coverage; Random Binary Generation.

I. Introduction

RISC-V is an open-source Instruction Set Architecture (ISA), which, among others, has a vector extension (RVV). This extension includes the vectorized version of many arithmetic, logical and memory instructions along with vector-specific instructions such as reductions, scatter and gather operations. With RISC-V, novel efforts are contributing to the open source community. Groups like OpenHW [1] have designed and developed verification environments for many Parallel Ultra Low Power (PULP) group [2] with designs such as RI5CY, Ariane and Ibex. Moreover, using the vector extension instructions, the vector processor ARA [3] from the ETH Zürich (within the PULP Platform) was implemented in GlobalFoundries 22FDX FD-SOI technology and has a microarchitecture based on version 0.5 of the RISC-V vector extension ISA.

Creating a reusable and expandable verification environment for accelerators that are connected to different kinds of cores, without a standardized interface between them, is a big challenge. In this sense, we have developed an interface-agnostic base verification infrastructure, reusable for different projects with distinct scalar cores and versions of the vector accelerator, expandable with singular types of interface implementations.

Our main goal in this work is to functionally verify a vector accelerator unit that implements the RISC-V vector extension [4], used in diverse projects that have different scalar processor cores and is connected to them through different interfaces, from which it receives vector instructions. To meet these requirements, we used the Universal Verification Methodology (UVM), which is built under the premises of creating a modular, scalable and reusable verification environment. It uses object-oriented programming to foment the reusability of the different modules across projects.

*No longer affiliated to BSC.
+Contribution to this paper performed when affiliated to BSC.

The proposed UVM verification environment has two main parts:

- An interface-agnostic environment shared among projects which:
    - Generates vector instructions using an Instruction Set Simulator (ISS) mimicking a project-specific scalar core.
    - Compares results between the ISS and the DUT.
    - Provides a continuous integration environment with sanity checks, random test generation and coverage collection.
- For each particular project, there is a specific environment that implements the behavior of the interface communicating the ISS and the vector accelerator. Communication with the interface-agnostic environment is accomplished using polymorphism.

II. Related Work

In recent years, the necessity for a verification process from different teams working with RISC-V has motivated different approaches in the industry, academia and research centers. At different DVCON chapters (Americas, China, India, and Europe), papers and presentations[5-10] using methodologies like UVM and open-source solutions have been published proposing various guidelines and tools in order to improve the performance of the verification process inside teams in charge of this task. Focusing on reusability and flexibility, A.S. Carretero et al presented [11] a UVM verification architecture and methodology with OOP foundations (successfully verified a Xilinx RFSoC DFE) in order to achieve: 1) A flexible testbench that allowed support of Unit-level, Top-level and System-level DUT's, 2) Maintainable and reusable coding that facilitated reuse, 3) Reduced debug for the testbench and 4) Maximum reuse of tests at the different integration levels.

In the specific case of the verification of a RISC-V Accelerator using the vector extension ISA, C.Li et al propose [12] a hybrid verification solution for a Vector Core using a UVM-based verification environment. This environment has a random instruction generation flow that feeds the DUT with a fetch agent. Moreover, exceptions are taken into account and travers all the possible exceptions that the RISC-V vector extension raises with higher requirements for the instruction's contextual relevance. In order to handle data hazards (inter and intra RAW/WAW/WAR) they use assertions to explore instructions' retirement and longest stall cycles.

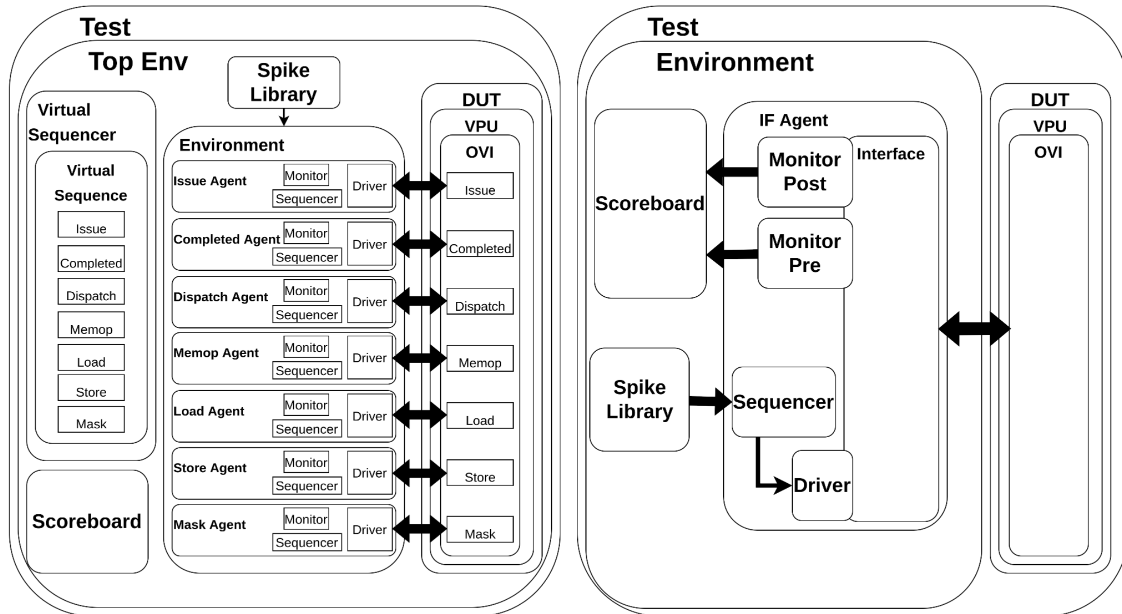
Our verification environment approach, similar to [12], uses RISCV-DV [13] an open-source random instruction generator for test generation and can be adapted for newer versions of the ISA, in case the RTL implements them. In addition, our environment has an adaptive structure for different projects that, for example, could have different interfaces to communicate the Vector Core to the Scalar Processor (e.g. for accessing memory) and/or different microarchitecture specifications (e.g. for configuration registers). Our adaptive environment is meant to have a flexible configuration structure to face the different existing projects and, possibly, future ones. With this, we have a maintainable and reusable environment for a Vector Processor Unit using RISC-V vector extensions.

III. Application

3.1 Previous verification environment

A previous verification environment had already been developed for stimulating and testing a previous version of the vector unit, but it was not reusable for other projects. In particular, the interface implementation was not encapsulated nor independent of the other parts of the verification infrastructure and it was difficult to maintain, extend and reuse. This environment had a different UVM agent for each channel of the interface, as shown in Figure 1, which involved massive inter-process communication.

Figure 1. Old UVM Environment (Left) and New UVM Environment (Right).



3.2 New verification environment

Following UVM's premise [14] of creating a modular, scalable and reusable verification environment, the one we built has a commonly shared base infrastructure that is independent of the interface communicating the scalar core and the vector unit. This infrastructure includes:

- Random test generation.
- Integration and co-simulation with an ISA simulator (which mimics the scalar core, sending vector instructions). The one we are currently using is Spike [15].
- Results comparison instruction-by-instruction between the vector accelerator and the simulator.
- A continuous integration environment, for running sanity checks, tests regressions and coverage results gathering.

The implementation of the interface [16] between the UVM environment and the vector unit is project dependent, but the base environment has already proven to work for two different ones: the European Processor Initiative (EPI), which uses the Open Vector Interface (OVI) [17] shown in Figure 4 and 5, and eProcessor EuroHPC project, which uses a custom interface, shown in Figure 6. These two projects have different scalar cores and different requirements for communicating with the accelerator. In particular, for EPI, memory access is done in the scalar core and data is transmitted to the vector unit through OVI. For eProcessor, on the other hand, memory is accessed directly by the vector unit (using the AMBA CHI protocol), and memory disambiguation checks need to be done using the scalar core-accelerator interface, in order to comply with the weak memory ordering model. Another difference between the accelerators is how vector configuration CSRs are handled: in EPI these are handled in the scalar core and transmitted through OVI and in eProcessor the accelerator handles them itself. Moreover, the accelerators for each project support different versions of the RVV ISA extension. For handling these, the infrastructure supports using distinct Spike versions to serve as reference models (for driving vector instructions and for results comparison).

3.2.1 Golden reference model - Spike

The golden reference model in the verification environment is used to mimic the scalar core, send vector instructions to the vector accelerator, run these same vector instructions and compare the obtained results with those of the DUT.

The base verification environment supports the use of any ISA simulator as a reference model by declaring a set of pure virtual methods in a wrapper class that should be implemented by the desired ISS. The wrapper class is then overridden by the one implementing these methods in the build phase of the UVM test, by using the factory override capabilities: *set_type_override_by_type(iss_wrapper::get_type(), spike::get_type());*

Communication with Spike ISS is accomplished by using SystemVerilog's Direct Programming Interface (DPI), as Spike is implemented in C++. The imported functions, shown in Figure 2, are called in the class extending the ISS wrapper, that is, the one implementing the pure virtual methods.

Figure 2. Imported Spike ISS functions though DPI.

```
1.  import "DPI-C" function void spike_setup(input longint argc, input string argv);
2.  import "DPI-C" function int  run_and_inject(input int instr, output iss_state_t iss_state);
3.  import "DPI-C" function int  exit_code();
4.  import "DPI-C" function void set_tohost_addr(input longint tohost_addr, input longint fromhost_addr);
5.  import "DPI-C" function void get_memory_data(output longint mem_element, input longint mem_addr);
6.  import "DPI-C" function void start_execution();
7.  import "DPI-C" function int  set_memory_data(input int unsigned data, input longint unsigned address, input int size);
8.  import "DPI-C" function void do_step(input int unsigned n);
9.  import "DPI-C" function void spike_set_external_interrupt(int mip_val);
10. import "DPI-C" function int  spike_run_until_vector_ins(inout iss_state_t iss_state);
```

For each specific project, a different version of Spike is used, as they support different versions of the vector extension. To support this in a seamless way, Spike's library file is project dependent and a different branch of the ISS repository is used and maintained to generate this file accordingly. For vector extension 0.7.1 [4], changes were made to the original Spike repository as it implemented a later version of the ISA.

3.2.2 ISA tests

We created a suite of ISA tests that could give the possibility to confirm and simulate the basic functionality of the vector instructions described at the RISC-V Vector Extension Specs version 0.7.1.

First, we grouped the instructions by type, as in the different chapters of the specifications: Vector Loads and Stores, Vector Integer Arithmetic Instructions, Vector Fixed-Point Arithmetic Instructions, Vector Floating-Point Instructions, Vector Reduction Operations, Vector Mask Instructions and Vector Permutation Instructions. After grouping them by type, we generate the tests by sub-types of instructions, again as in the specifications. For example, in the group of Vector Arithmetic Instructions (Chapter 12 in the specs), we created a test for Vector Single-Width Integer Add and Subtract (Subchapter 12.1), another test for Vector Widening Integer Add/Subtract (Subchapter 12.2), and so on for all the subchapters.

The main characteristics we took into account into account to configure the tests are:

1) Single Element Width (SEW) and Vector Length (VL) - We tests the 4 different SEWs: 8, 16, 32, and 64 bits with its maximum VL: 2048, 1024, 512, and 256 for EPI (long vectors). For eProcessor VL: 1024, 512, 256, and 128 because the VPU has half Vector Lanes (4 Lanes) than those located in EPI (8 Lanes).
2) Unmasked and Masked: Each operation has the possibility of being masked. For this, a special vector register needs to hold the active(1)/inactive(0) bits that would give the mask for the given operation. All instructions in each test considered their unmasked and masked version using, for both, the different SEWs and VLs.

3.2.3 RISCV-DV randomly generated tests

RISCV-DV is an open-source SystemVerilog/UVM based random instruction generator for RISC-V developed by Google which we use in our environment to generate randomly constrained binaries. With this tool configuration options we generate different types of tests for our projects and, in some cases, we have modified the tool itself in order to fit our needs.

These aforementioned configuration options are called targets, and these include information about the core for which the tests are being generated, such as the list of unsupported instructions, vector extension parameters

or implemented CSRs; and configurations on the structure of the test to be generated, such as number of instructions, enabling of some kinds of instructions, generation of branches, generation of trap and exception handlers or the boot mode.

This tool has a wide range of configuration possibilities without the need for modification of the targets mentioned above. However, we needed to add a few modifications to make the tool fit our needs completely. This is why we actually have different versions of the tool for the different projects. These versions have small modifications and they all branch out from a main version which hosts all the major changes that we implemented that involve all the versions and the changes coming from the upstream repository of the tool.

The first modification is the addition of a directed instruction stream to correctly generate vector memory instructions. This part is meant to be modified, since the directed instruction stream mechanism is made to be able to insert chunks of predefined instructions inside the randomized code, so we made use of it to generate memory instructions with a correct range of indexes, strides and directions.

Another great part of the modifications are the addition of several new options so that we could change some things of the generation of the binaries with a simple run-time option. Some of these include the enabling and disabling of the generation of some types of instructions, such as reductions, narrowing and widening or vector floating point; and data pattern for the data page generation.

Finally, the other part of the modifications involve the change of some parts of the code, such as several constraints regarding SEW and VL, changing the initialization or ending code, which involved modifying some of the hardcoded instructions and also adding new instructions and new mechanisms to allow, for example, the generation of vsetvli instructions randomly throughout the code.

3.2.4 Base verification environment - vpu-dv

The base interface-agnostic verification environment provides a set of UVM components to generate vector instructions, send them to the DUT, receive its results and compare them with those of the reference model. Some of these classes are virtual and meant to be extended, implementing methods specific for the project to be verified. In particular, one of the most important ones, is the *protocol_base_class*, which is instantiated in the driver and monitors of the base environment, extended in the project specific environment and needs to implement the methods depicted in Figure 3, according to the specifications of the interface connecting the scalar core and the vector unit:

Figure 3. *protocol_base_class* virtual methods.

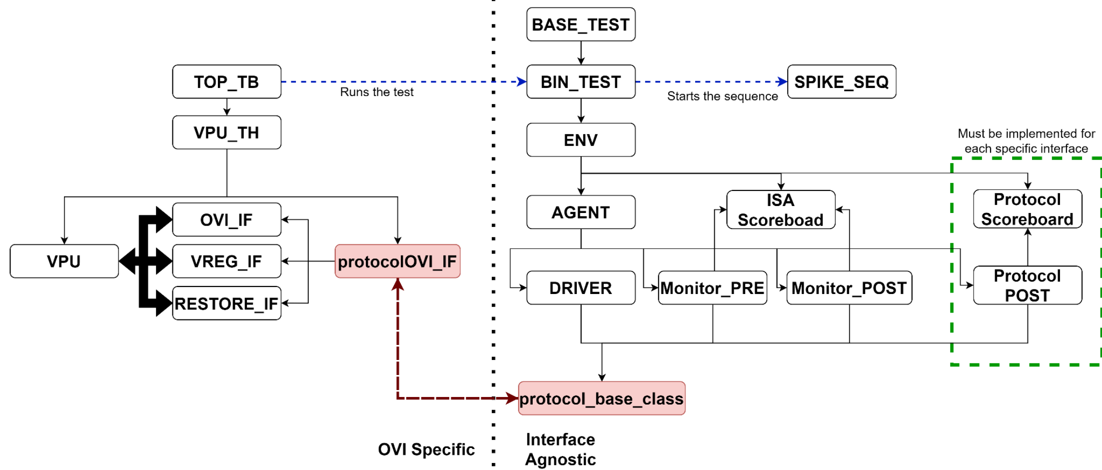
```
1.  pure virtual task do_protocol(); //Runs the specific protocol of the interface to stimulate the DUT
2.  pure virtual task wait_for_clk(int unsigned num_cycles = 1); //Waits for as many num_cycles cycles of the interface clock
3.  pure virtual function drive (ins_tx req); //Pushes the instruction inside the transaction into the pending instructions queue
4.  pure virtual function bit new_ins_tx(); //Returns whether or not there are new instructions received from the driver
5.  pure virtual function iss_state_t monitor_pre(); //Returns the first pending instruction received from the driver
6.  pure virtual function bit new_dut_tx(); //Returns whether or not there are new completed instructions
7.  pure virtual function dut_state_t monitor_post(); //Returns the first pending completed instruction's result
8.  pure virtual function bit new_protocol_tx(); //Returns whether or not there are new completed instructions
9.  pure virtual function protocol_instr_t monitor_protocol(); //Returns the first pending completed instruction
10. pure virtual function protocol_instr_t next_infl_instr(); //Returns the first inflight instruction
```

In particular, Figure 4 shows the connection between the base verification environment and the specific one for the EPI project.

Regarding version control (done using git), we have a repository for the base verification environment which needs to clone, inside of it, the project specific and the tests ones. Continuous integration is done using Gitlab pipelines. The base verification environment provides a shared infrastructure for the following pipelines:

- Sanity check: run on every commit to the repository to check the correct compilation of the environment.
- ISA tests: run a set of tests aimed to test every instruction in the ISA specifications.

Figure 4. Interface-Agnostic Base Environment with OVI implementation for EPI.

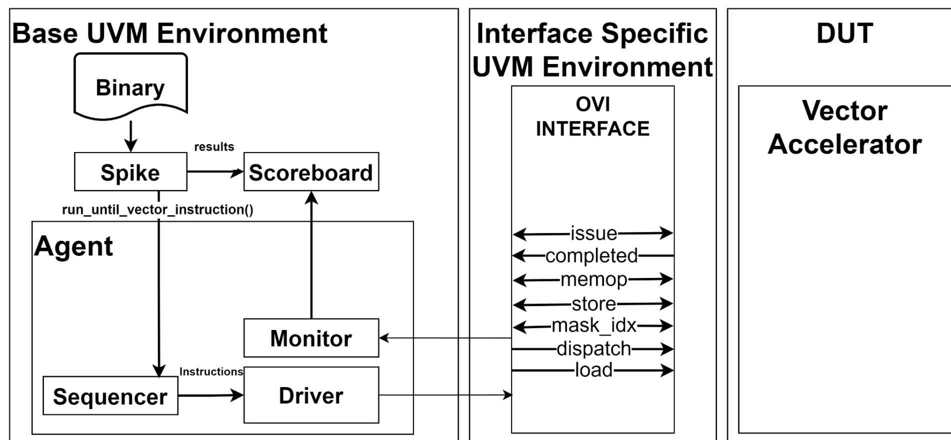


- Random tests: run a set of randomly generated tests created on each run using RISCV-DV.
- Merge request: run ISA tests and random tests on branches different than develop, when a merge request to it is done.
- Small regression: run a small set of tests generated by ranking random tests' contribution to coverage results. Run daily.
- Complete regression: run a large set of tests generated by ranking random tests' contribution to coverage results. Run weekly.
- Failed tests: run previously failing tests to check if these pass with new modifications.

Functional and code coverage collection is done by merging coverage databases from these pipelines and an HTML report with its results is generated and accessed via web. Functional coverage covergroups and coverpoints have been defined for the inner modules of the vector accelerator, and each specific project has its own functional coverage definition for supported vector instructions and the interface communicating the core with the accelerator.

3.2.5 EPI verification environment - epac-vpu-dv

Figure 5. Simplified Diagram of the Verification Environment with EPI's OVI Interface.



One of the specific projects that makes use of the proposed verification environment is the vector accelerator for the European Processor Initiative (EPI). This accelerator communicates with a core through the Open Vector Interface (OVI). The implementation of this interface extends from the protocol_base_class and implements its

pure virtual functions. In particular, the *do_protocol()* function calls a task for each subinterface of OVI (issue, dispatch, completed, memop, load, store and mask_idx). These tasks read a queue containing vector instructions, which is filled by the *drive()* method of the interface, called by the base environment's driver. Completed instructions are put in another queue, which is read by the environment's monitor by calling the *monitor_post()* method.

The epac-vpu-dv also implements a monitor and a scoreboard for checking the correct behavior of the interface. In particular, for memory operations, it checks mask, indexes, and values to load/store to/from memory in order to compare them against the ISS.

This project's Spike version performs memory reads and writes, as needed for communicating with the vector unit, which relies on the scalar core for performing these operations, as well as for handling CSRs.

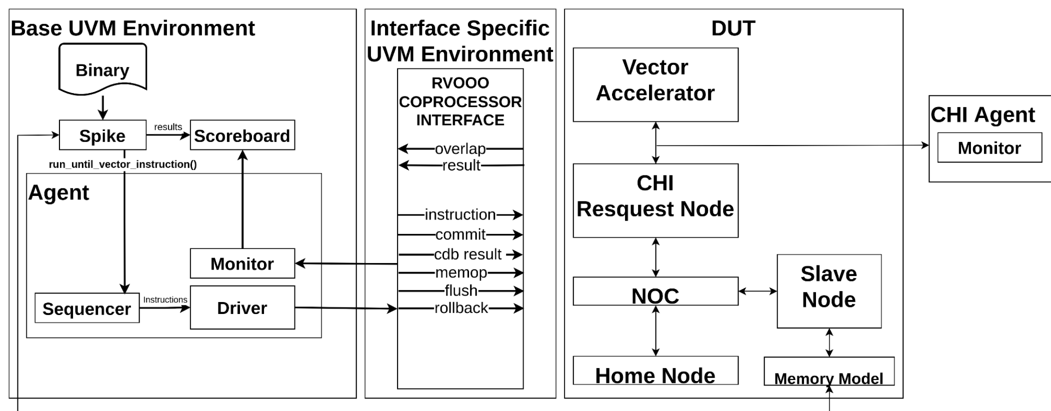
Regarding the environment compilation and simulation scripts, the repository has files included in the base environment's makefile, which define the specific project's RTL and verification git repositories to be cloned.

For coverage, apart from the covergroups defined in the base verification environment for the RTL inner modules, the epac-vpu-dv instantiates covergroups for instructions supported by this project and the OVI interface.

3.2.6 eProcessor verification environment - eprocessor-vpu-dv

Another project using the split verification infrastructure is the vector accelerator being used for eProcessor. The particularity of this accelerator is that it has the capability of accessing memory directly, without needing to send requests to the core. Memory access is done using the AXI CHI protocol. The interface communicating the core and the vector accelerator needs to send information to detect and avoid memory aliasing between these two. This functionality is implemented in the interface extending the base_protocol_class.

Figure 6. Simplified Diagram of the Base Verification Environment with eProcessor's Interface Implementation.



Regarding Spike, modifications were made in order to support FP8, FP16 and AI instructions, and, as CSRs are handled in the vector unit (contrary to EPI), instructions for reading and writing them need to be sent through the protocol interface (like vsetvl/vsetvli/csrr/csrw).

IV. Preliminary Results

Encapsulating the interface implementation and creating a base interface-agnostic environment, which can be used for distinct projects, gave us the following advantages:

- An extendable, reusable and portable environment to work with.
- In comparison with the old verification environment, which had substantial interprocess communication due to the implementation of the interface channels in several agents, the new

environment (which implements these functionalities in just one interface) compilation and simulation times are faster.

- The ability to uncover more bugs, previously not detected using the old environment, as a result of a better implementation (less constraints on test cases generation) and bigger test regressions.

The environment infrastructure has allowed us to drastically improve the development of verification environments for different and newer versions of the vector accelerator. For example, for eprocessor-vpu-dv, along with RTL development, the verification environment interface implementation helped both teams identify bugs in the DUT and the verification environment, running individual tests. In particular, we're now developing the second version of the EPI vector unit (epac2-vpu-dv), which uses version 2.0 of the OVI protocol. By just developing the implementation of this protocol, and using the vpu-dv base verification environment, we're already able to begin verifying the new vector accelerator, along with the RTL development, finding bugs sooner and faster.

V. Conclusions

In this paper, we have described the implementation of a reusable verification environment targeting a RISC-V vector accelerator. The environment has already been used for two distinct projects implementing different scalar cores and their corresponding interface with the vector unit, and is planned to be used in other projects to come, avoiding the overhead of developing a new verification infrastructure by using an interface-agnostic common base environment.

Acknowledgment

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) and No 956702 (eProcessor) . 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_N1618737 is also co-funded by MCIN/AEI /10.13039/501100011033 and by the UE NextGenerationEU/PRTR

References

[1] OpenHW Group, Website: https://www.openhwgroup.org/

[2] PULP Platform, Website: https://pulp-platform.org/

[3] Cavalcante, M., et al, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI" IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019.

[4] RISCV, "Vector Extension 0.7.1 Specs", https://github.com/riscv/riscv-v-spec/releases/download/0.7.1/riscv-v-spec-0.7.1.pdf

[5] E. Karabulut, et al, "A comprehensive Verification Platform for RISC-V based Processors" DVCON EU, Oct 2020.

[6] W. W. Chen,et al, "A Methodology to Verify Functionality, Security, and Trust for RISC-V Cores" DVCON EU, Oct 2020.

[7] S. Davifmann, and L. Moore. "Introduction to the 5 levels of RISC-V processor verification" DVCON U.S., presentation, Feb 2022.

[8] M. Chupilko, et al, "Open Source Solution for RISC-V Verification" DVCON EU, presentation, Oct 2019.

[9] N. Tusinschi, "RISC-V Integrity: A Guide for Developers and Integrators" DVCON Europe, presentation, Oct 2019.

[10] M. Zachariasova, et al, "UVM-based Verification of a RISC-V Processor Core Using a Golden Predictor Model and a Configuration Layer" DVCON United States, Feb 2018.

[11] A. Sanz Carretero, et al, "Testbench flexibility as a foundation for success" DVCON Europe, Oct 2021.

[12] C. Li, Y. Feng, and L. Li, "A Hybrid Verification Solution to RISC-V Vector Extension" DVCON U.S., presentation, Feb 2022.

[13] RISCV-DV, Website: https://github.com/google/riscv-dv

[14] Glasser, Mark. "Open verification methodology cookbook". Springer Science & Business Media, 2009.

[15] Spike, Website: https://github.com/riscv-software-src/riscv-isa-sim

[16] Rich, David. "The missing link: the Testbench to DUT connection." Fremont, CA: D&V Technologies Mentor Graphics 9, 2013.

[17] R. Espasa, P. Marcuello, A. Moreno, S. Pomata, "OVI: Open Vector Interface Specification" Semidynamics, https://github.com/semidynamics/OpenVectorInterface/blob/master/open_vector_interface_spec.pdf, Dec 2021.