A RISC-V "V" VP: Unlocking Vector Processing for Evaluation at the System Level

Manfred Schlägl Moritz Stockinger Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
manfred.schlaegl@jku.at moritz.stockinger@outlook.com daniel.grosse@jku.at

***Abstract*—In this paper we introduce the first free- and open-source SystemC TLM based RISC-V *Virtual Prototype* (VP) with support for the *RISC-V "V" Vector Extension* (RVV) Version 1.0.**

After an introduction to RVV, we present the integration of RVV and its 600+ instructions into an existing VP leveraging code generation for over 20k *Lines of Code* (LoC). Moreover, we describe the verification of the resulting VP using the *Instruction Sequence Generator* (ISG) *FORCE-RISCV* and the *Instruction Set Simulator* (ISS) *riscvOVPsim*.

Our case studies demonstrate the benefits of the RVV enhanced VP for system-level evaluation. We present non-vectorized and vectorized variants of two common algorithms which are executed on the VP with varying parameters. We show that by comparing the number of simulated execution cycles, we can derive valuable assessments for the design of RVV micro-architectures.

I. Introduction

The open-standard *Instruction Set Architecture* (ISA) RISC-V [1], [2] has the potential to democratize processor design, making it a disruptive force in the semiconductor industry. A key feature of RISC-V is its modularity which allows for a wide range of customization and specialization as well as to balance performance and power efficiency. Modularity is achieved by a variety of standard extensions that can be added to the base ISAs (RV32I for 32-bit Integer and RV64I for 64-bit Integer) to enhance its capabilities for specific tasks.

To significantly improve the performance for tasks such as image and video processing, audio processing, scientific simulations, and nowadays many AI algorithms, their *Data-Level Parallelism* (DLP) is exploited and *Single Instruction, Multiple Data* (SIMD) instruction set extensions have been developed. SIMD achieves this by performing the same operation on multiple data elements, which is called a ***Vector***, in parallel. The concept of SIMD was initially explored in the 1970s [3], gaining popularity in supercomputers developed by Cray, and these early endeavors are now recognized as vector architectures [4]. For approximately 25 years now, processors have included SIMD instructions categorized as multimedia extensions [5], which we refer to here as *classical SIMD*. Most of these classical SIMD extensions (e.g. Intel MMX, SSE, AVX, or ARM Neon) are designed to operate with fixed-size registers (vector length) as the underlying *Hardware* (HW) implementation is simplified. However, there are two main challenges of classical SIMD: (i) determining the ideal vector length is hard as it depends on the specific workload being targeted, and (ii) changing of the vector lengths means that the ISA extension has to be modified and thereby rendering previously compiled code obsolete.

We use a simple example to demonstrate the conceptual difference between classical SIMD (fixed vector length) and vector architectures (variable-length vectors). Fig. 1 shows the processing of large vectors in *stripmining* loops. In classical SIMD, the length of a vector is defined statically by the used vector register defined in the ISA. For example, the ARM *Advanced SIMD* (Neon) `q` registers can hold up to **16** elements with 8 bit. So in each iteration a fixed number of elements is processed. In contrast, in vector architectures the *Software* (SW) requests the needed vector length (**`reqlen`**) and the HW tells the available length (**`vl`**). In the *RISC-V "V" Vector Extension* (RVV) this is done with the `vsetvli` instruction. So in each iteration automatically the maximum available vector length (**`vl`**) is used. In summary, the advantage of vector architectures is that the capabilities of different HW can be fully exploited by the same SW binary.

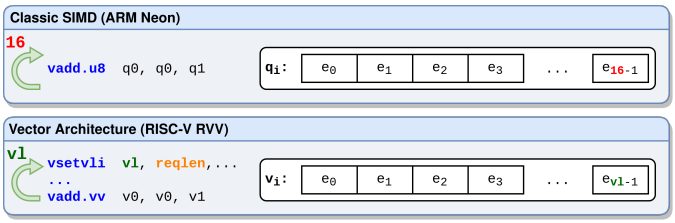

Fig. 1: Classic SIMD (ARM Neon) vs. Vector Architecture (RISC-V RVV)

*Virtual Prototype*s (VPs) are industry-proven and allow to parallelize HW and SW development as well as enable early architectural exploration. In essence, a VP is a high-level, executable model of the entire HW platform which runs unmodified production SW [6], [7] targeting the system level. In the current landscape, SystemC, a standardized class library for C++ (IEEE 1666, [8]), is the primary choice for developing VPs [9]. Based on the abstraction of communication details by leveraging *Transaction Level Modeling* (TLM) [10] orders of magnitude faster simulation in comparison to *Register Transfer Level* (RTL) is achieved [6].

Contribution: We consider the open-source SystemC TLM based *RISC-V VP* introduced in [11], more specifically the derived GUI-enabled *GUI-VP* presented in [12]. The VP already comes with *Instruction Set Simulator*s (ISSs) for RV32 and RV64 and supports the RISC-V *IMAFDC* extensions. However, it lacks support for RVV, which was ratified 2021 in Version 1.0 [13]. In this paper, we **extend the *GUI-VP* with RVV Version 1.0 resulting in *RISC-V VP++*** which is available on GitHub[1]. The integration of RVV is a consequence of careful analysis of the RVV specification and the operation of the *GUI-VP*, which has to be understood to provide (i) efficient support for RVV in the RV32 and RV64 ISSs of the VP and, (ii) a solid functional verification of RVV

[1]https://github.com/ics-jku/riscv-vp-plusplus

in the VP. Leveraging code generation for over $20k$ of the total $23k+$ *Lines of Code* (LoC) required to integrate the $600+$ RVV instructions, the potential for implementation errors and maintenance overhead is greatly reduced.

The added benefit of the novel RVV enhanced VP for system-level evaluation is demonstrated in our case studies. We show that by integrating a simple, parametrizable instruction-accurate execution cycle model, and comparing the execution of non-vectorized with vectorized implementations, valuable assessments can be derived for the design of RVV micro-architectures.

Related Work: There are a number of open-source RISC-V simulators, such as *Spike* [14], *QEMU* [15], *RV8* [16] or *DBT-RISE* [17]. However, *RV8* and *DBT-RISE* do not provide support for RVV. *Spike* and *QEMU* are supporting RVV in Version 1.0, but their ISSs and platform parts are not implemented in SystemC, hence accuracy and granularity cannot be modeled following the SystemC standard.

Commercial VPs, like Synopsys Virtualizer or Mentor Vista, might also support RVV as well as fast and accurate timing models but their implementation is proprietary.

To the best of our knowledge, the VP presented in this paper is the only open-source, SystemC based VP that supports RVV and an instruction-accurate execution cycle model.

II. The RISC-V "V" Vector Extension (RVV)

The ratified Version 1.0 of RVV is specified in [13]. Essential parts added to the RISC-V programming model by RVV are: (i) 32 Vector registers, (ii) 7 *Control and Status Register*s (CSRs), and (iii) 624 instructions.

The 32 vector registers are used by RVV instructions for parallel computation. All registers can hold vectors with configurable element sizes and overall lengths up to `VLEN` bits. The concrete value of `VLEN` is not specified in the ISA, but can be chosen by the designer in powers of 2. For example, vector registers with a typical length of $\texttt{VLEN} = 128$ bit can hold vectors with up to 16 elements of 8 bit size, up to 8 elements of 16 bit size, or up to 4 elements of 32 bit size. The element size can be selected with special instruction (e.g. `vsetvli`).

A particularly noteworthy feature of RVV is *register grouping*, which allows 2, 4 or 8 vector registers to be combined to increase parallelism. For example, if a *grouping* of 8 is used, the 32 registers (`v0-v31`) with $\texttt{VLEN} = 128$ bits are combined to 4 registers (`v0,v8,v16,v24`) with $\texttt{VLEN} * 8 = 1024$ bits.

RVV adds 624 instructions, which can be categorized in configuration-, load/store- and processing operations.

Configuration instructions, such as `vsetvli` define the processing of follow-up instructions. They configure the element and vector lengths (`vtype` and `vl` CSRs).

With load/store instructions data is moved between memory and vector registers. Especially noteworthy is the support for *strided* and *indexed* load/stores that provide efficient handling of data structures in memory (e.g. loading matrix columns).

RVV offers a comprehensive set of instructions for integer, fixed point and floating point vector processing. Moreover, it comes with instruction for vector reductions (e.g. sum, min, max, ...) and permutations (e.g. move, slide, ...).

For operations that provide results wider than their operands there exist *widening* instruction variants which save the results in vectors with double the element length (e.g. multiplication of 16 bit elements to 32 bit elements). Correspondingly, there are *narrowing* variants for opposite cases.

Finally, *masked* variants exist for many RVV instructions. These variants additionally consider the vector register `v0`, which is interpreted as a bit mask. This mask can be used to specify the element indices on which the operation is to have an effect. With this it is for example possible to add only some elements of two vectors.

III. Integrating RVV in GUI-VP

In this section, we present the integration of RVV in the ISSs of *GUI-VP*. We first give an overview of the essential components for processing instructions in the ISSs. After that, we discuss each newly introduced component in detail.

Fig. 2 presents the concept of the RVV integration in the ISS instruction processing. Already existing components for processing *IMAFDC* instructions are shown in gray. Components added for RVV in this integration process are highlighted in blue and purple. The components marked as blue are automatically generated, each for the *RV32* and the *RV64* ISS. The components marked as purple are implemented generically for both ISSs manually.

On the left of Fig. 2, a new instruction is entering the *Instruction Decoding* (InsDec) stage. InsDec extracts the various *Field Values* (e.g. register addresses, immediate values, shift amounts, ...) of the instruction according to the *Instruction Encoding*. In addition, InsDec determines a unique *Opcode* from the *Opcode Table* that assigns the instruction to a specific operation to be executed. Finally, the *Opcode* and *Field Values* are passed on to the next stage, the *Operation Selection* (OpSel).

In the middle of Fig. 2, the OpSel stage can be seen. In this stage, the operation to be executed is selected on the basis of the *Opcode*. The *Opcode* further determines the required *Field Values* for an operation. The selected operation with its parameters is then passed on to *Execution* (Exec).

On the right of Fig. 2, we see the Exec stage with all additional parts necessary for the execution (*Registers*, *CSRs*). Exec receives the parametrized operation from OpSel and performs all the steps necessary to execute the operation, e.g. permission checks, register and memory read/writes, computations, compares, branches, jumps, ...

With this, we have introduced the essential components of the ISS instruction processing. Next, in Section III-A, Section III-B and Section III-C, we present the main stages and the newly introduced blue and purple RVV components of Fig. 2 in more detail. Finally, in Section III-D, we focus on the automated code generation used for the blue components.

A. Instruction Decoding (Fig. 2, left)

In this stage, we introduce the two new components: (i) *RVV Encoding* which extends the *Instruction Encoding* and (ii) *RVV Opcodes* that extend the *Opcode Table*.

Instruction Encoding is realized with C++ functions that extract all *Field Values* defined in the instructions formats of RISC-V. To support RVV we generate new functions to extract 9 new *Field Values* introduced by RVV, e.g. the *masking* bit `vm`, or `zimm[10:0]` for `vsetvli`.

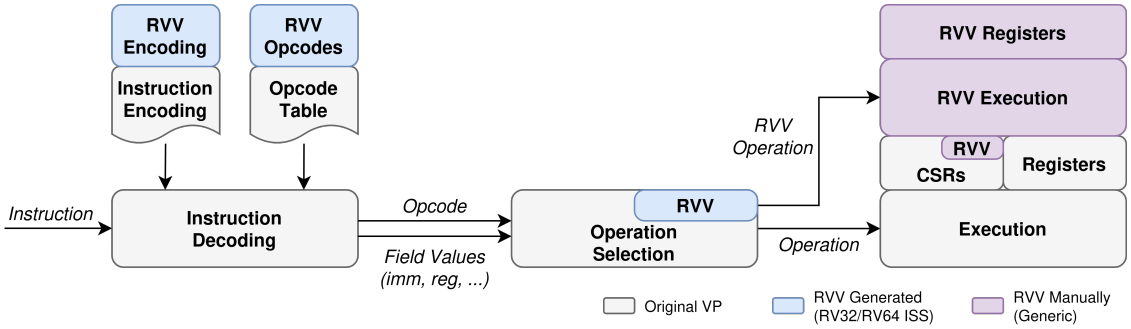

Fig. 2: Conceptual Overview: RVV Integration in the *GUI-VP* ISSs

Regarding *Instruction Encoding* and *Opcode Table*, there is the decoding of instructions and their mapping to an *Opcode*. Decoding and mapping is implemented as a decision tree (C++ multi-nested switch case) in which individual bits and bit fields of the instruction are examined successively until an operation is identified. The decision tree finally provides a corresponding *Opcode* from the *Opcode Table*.

To add support for RVV, we extend (i) the *Opcode Table* with the *RVV Opcodes* and (ii) the decision tree for the new instructions (*RVV Encoding*) and *Opcodes* (*RVV Opcodes*).

B. Operation Selection (Fig. 2, middle)

As described before, this stage gets the *Opcode* and all *Field Values*. The selection is realised as a case distinction on the *Opcode* (C++ switch case). For each case, the necessary *Field Values* are extracted. The execution of the operation is then performed in the Exec stage.

To support RVV, the case distinction is extended with the *RVV Opcodes*. Also, for RVV, the extraction of *Field Values* is done in the Exec stage. The advantage of this is that automated code generation can also be applied to this stage.

C. Execution (Fig. 2, right)

The Exec stage, is where the actual execution of instruction takes place. It receives the parameterized operation from the OpSel step and performs all the steps necessary to execute the operation. The Exec stage also implements all registers and CSRs the instructions interact with.

To integrate RVV we add 7 new *RVV CSRs*, 32 *RVV Registers* and 624 instructions, as presented in Section II.

The *RVV CSRs* are added to the existing list of CSRs of the ISS. All CSR instructions can automatically access them. RVV also adds bits to the existing CSRs `misa` and `mstatus`. A constant set bit is added to `misa` to indicate support for RVV. In `mstatus`, the enable and dirty bits for RVV are added.

The *RVV Registers* are accessible from RVV instructions only and are efficiently implemented as C++ arrays of `VLEN` bits. `VLEN` is configurable and by default set to 512 bits.

The major part of the integration is the implementation of the 624 RVV instructions. Many of the instructions are variants of the same operations but with slightly different behavior (e.g. *widening*). This high level of regularity is exploited to produce an efficient implementation with high reuse.

The examples in Listing 1 demonstrate the basic idea. The listing shows the pseudocode implementation of three arithmetic RVV instructions. All implementations follow the same basic scheme: They use a generic `vLoop` function that implements the iteration over the `vl` elements. The parameters of `vLoop` are (i) the actual operation to execute, (ii) `elem_sel_t`, and (iii) `param_sel_t`. The `elem_sel_t` parameter indicates the widening (`x`/`w`) and signedness (`u`/`s`) of the three involved vectors. The `param_sel_t` parameter defines whether one source operand is a vector, immediate or scalar.

```
1  vadd.vi:
2    vLoop(vAdd(), elem_sel_t::xxxsss, param_sel_t::vi)
3  vsub.vx:
4    vLoop(vSub(), elem_sel_t::xxxsss, param_sel_t::vx)
5  vwaddu.wv:
6    vLoop(vAdd(), elem_sel_t::wwxuuu, param_sel_t::vv)
```

Listing 1: Implementation Examples of Arithmetic Instructions (Pseudocode)

`vadd.vi`, in Line 1 of Listing 1, adds an immediate to a vector. It uses `vLoop` to execute `vAdd` on non-widened vectors (`xxx`), with signed values (`sss`) and an immediate as operand (`vi`). `vsub.vx`, in Line 3, subtracts a scalar from a vector. It uses `vLoop` to execute `vSub` on non-widened vectors (`xxx`), with signed values (`sss`) and a scalar as operand (`vx`). Finally, `vwaddu.wv`, in Line 5, performs an unsigned add of a vector to a widened vector. It uses `vLoop` to execute `vAdd` on two widened and one non-widened vector (`wwx`), with unsigned values (`uuu`) and a vector as operand (`wv`).

These simple examples show that although the instructions are different in detail, they can be realised using the same basic scheme. The same also applies to other RVV instructions.

However, there are also many aspects of RVV that are significantly more complex. Our implementation includes multiple variants of `vLoop` and many more operations like `vAdd`. In total, the manual RVV implementation for the Exec stage contains over 2,500 LoC.

D. Automated Code Generation (Fig. 2, blue)

In this section, we present the automated code generator which is used to generate the blue components shown in Fig. 2. Code generation includes, *RVV Encoding* and *RVV Opcodes* of the InsDec stage (Section III-A), and RVV in the OpSel stage (Section III-B).

Manually creating code for *RVV Encoding*, *RVV Opcodes* and operation selection for 624 RVV instructions would be very tedious and error-prone. This is compounded by the fact that the code would have to be written twice, equivalently each for the *RV32* and *RV64* ISS. Since the generated components also contain many repetitive code patterns, the use of automated code generation pays off several times over.

```
Command("OPIVV", [
  "f6",  "f6",  "f6",  "f6",  "f6",  "f6",  "vm", "vs2",
 "vs2", "vs2", "vs2", "vs2", "vs1", "vs1", "vs1", "vs1",
 "vs1",    0,     0,     0,  "vd",  "vd",  "vd",  "vd",
  "vd",    1,     0,     1,     0,     1,     1,    1],
"[6:0],[14:12],[31:26]"),
```

Listing 2: *Command Table* Entry for Vector Instruction Format *OPIVV*

```
PreLimOp(
  "vadd.vv", "OPIVV",
  "vLoop(vAdd(), elem_sel_t::xxxsss, param_sel_t::vv);")
```

Listing 3: *Operation Table* Entry for `vadd.vv`

The code generator is implemented in Python and takes two inputs: (i) a *Command Table*, that describes the *Vector Instruction Formats* as defined in RVV spec Section 5 [13], and (ii) a *Operation Table*, that describes the 624 RVV instructions, where each entry refers to an entry in the *Command Table*.

An example of an entry in *Command Table* is shown in Listing 2. The entry describes the *OPIVV* instruction format, that is used for example for `vadd.vv`. The first subentry, `OPIVV`, defines the instruction format name. The second subentry is a list of 32 elements, which describe the meaning of the individual bits of the instruction format. `f6` describes the 6 bit wide function field, `vm` the mask bit, `vs2` the second vector source register, etc. Constant values like `0` and `1` describe fixed bit values to identify the instruction format. The third subentry describes, which bit ranges of a concrete instruction are used to determine the actual operation (*Opcode*). In this example, it contains the constant bit values and the `f6` field.

An example of an entry in *Operation Table* that describes the `vadd.vv` instruction is shown in Listing 3. The first subentry denotes the instruction name. The second subentry, `OPIVV`, is the reference to the instruction format entry in the *Command Table* (Listing 2). The third subentry already contains a fragment of the code to be generated. More details on this we already introduced in Section III-C above (Listing 1).

The code generator iterates over all 624 instructions described in the *Operation Table*. For each entry the reference to *Command Table* is resolved. Having now all needed information for each instruction, the generator creates corresponding code fragments for all blue components shown in Fig. 2. This includes *RVV Encoding* and *RVV Opcodes* in the InsDec stage, and the RVV handling in the OpSel stage.

For the *RV32* and *RV64* ISS, over $10,400$ lines of code are generated each. This totals to over $20,800$ lines of generated code. Compared to the about $2,500$ lines of hand-written code for the Exec stage, this is a significant saving in terms of effort.

IV. Verifying RVV

This section presents the verification of the RVV enabled *RISC-V VP++* presented in Section III. We first give a conceptual overview of the verification chain. After that, we discuss each step of the chain in detail. Finally, we discuss the achived verification coverage.

The conceptual overview of the verification chain is shown in Fig. 3, with *RISC-V VP++* as *Simulator under Test* (SuT) highlighted in purple. We use an *Instruction Sequence Generator* (ISG) to generate RVV test cases as executable (ELF) files. Each test case is fed into the *Reference Simulator*, the SuT and in the *Coverage Analysis* simulator. The *Reference Simulator* and SuT create traces of memory and register accesses. The *Coverage Analysis* simulator computes and accumulates the achieved coverage.

The generated traces are then stripped from irrelevant information and compared in the *Post-processing & Comparison* step. The final results of the comparisons and the coverage achieved are then compiled in a *Verification Report*.

We will now discuss each step of the chain in detail.

A. The Instruction Sequence Generator: FORCE-RISCV

As ISG we use *FORCE-RISCV* provided by the *OpenHW Group* [18]. At time of writing, it was the only available free- and open-source ISG with support for RVV Version 1.0.

FORCE-RISCV provides a comprehensive and highly configurable framework for RISC-V test generation. It employs randomization to autonomously select instructions, registers, addresses, and data for generating tests. The provided API grant significant control over the generation process. It can be used in test templates to specify the instructions to be tested and to define associated constraints.

For our verification, we use the pre-defined RVV test template contained in *FORCE-RISCV*. However, two modifications are necessary: First, the RVV test template contains the vector *Atomic Memory Operations* (AMO) instructions, which are not part of RVV Version 1.0. The RVV test template is modified to exclude these instructions.

Second, the test case generator assumes a 48 bit address space. Since the address range of *RV32* is limited to 32 bits, the generator is modified accordingly.

The test set that *FORCE-RISCV* generates contains $6,300$ test cases with over $250,000$ vector instruction invocations.

B. Reference Simulator: Handcar

As a *Reference Simulator*, we use the tracing simulator *Handcar* that comes with *FORCE-RISCV*.

Handcar is based on the widespread *Spike* [14] simulator and has been extended to generate trace output. For each instruction executed, all memory and register accesses (reads and writes) are logged as a separate line of the trace.

FORCE-RISCV is configured to automatically call *Handcar* for each test case during generation. As consequence, the creation of the test set, presented in Section IV-A, provides *Reference Traces* for all $6,300$ test cases.

C. Simulator under Test: RISC-V VP++

Our RVV enabled *RISC-V VP++* forms the SuT. However, to allow a later comparison, *RISC-V VP++* is extended to provide trace output similar to the *Reference Simulator Handcar*. For each instruction executed, all memory and register accesses are logged as a separate line of the trace.

A top-level script automates the execution of the $6,300$ test cases on the SuT after generation. The resulting intermediate outputs are thus $6,300$ *Ref-* and *SuT Traces*.

D. Post-processing & Comparison

The *Post-processing & Comparison* step is implemented in Python and takes the $6,300$ *Ref-* and *SuT Traces* as input.

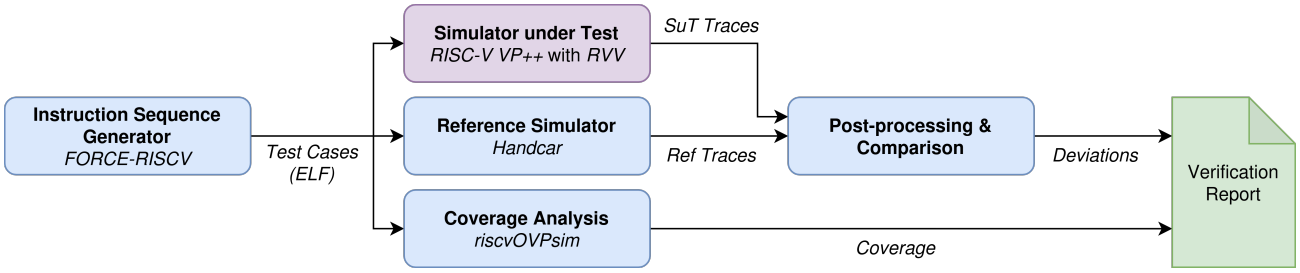

Fig. 3: Conceptual Overview: RVV Verification Chain

The traces produced by *Handcar* cannot be used for comparison without *Post-processing*: First, *Handcar* traces contain superfluous register reads that do not correspond to the function of actual instruction. The reason is most likely a bug in *Handcar* that inadvertently outputs some implementation details of the underlying *Spike* simulator. Since reads in our case do not affect the HW execution state, they are completely removed by the *Post-processing*.

Second, in *Handcar* traces, memory and register accesses to the same destination can occur multiple times with different values and in different order. This holds also true for the individual elements of the RVV registers. Also here, we suspect that *Handcar* outputs too much detail of the underlying *Spike* simulator in the trace. Since only the last write to a destination determines the value of the destination after the instruction, the *Post-processing* removes all writes to a particular destination except the last one.

After *Post-processing* the $6,300$ *Ref-* and *SuT Traces* are compared. All deviations are included in the *Verification Report* in a human readable form (html) to facilitate debugging.

E. Coverage Analysis

Finally, to evaluate our verification, we determine the test coverage. For this, we use the free, but closed-source RISC-V ISS *riscvOVPsim* from *Imperas* [19].

riscvOVPsim supports RISC-V *IMAFDCSUE* and RVV, and computes comprehensive *functional* coverage metrics.

Each of our $6,300$ test cases is fed into *riscvOVPsim* by the top-level automation script. The simulator determines the coverage of each test case and accumulates the results. The overall result is then included in the *Verification Report*.

In total, we get $26,936$ out of $33,076$ basic coverage points for RVV, which corresponds to a coverage of **81.44%**. Although this value is already promising, there is still potential for improvement. However, they are left for future work and could for example include: (i) increasing the number of generated test cases, (ii) defining more sophisticated *FORCE-RISCV* test templates for RVV, (iii) integrating fuzzing techniques in *FORCE-RISCV*, or (iv) investigating the application of metamorphic testing methods [20].

V. Case Studies

In this section, we demonstrate the benefits of the RVV enhanced *RISC-V VP++* for system-level evaluation. First, we add a simple, parametrizable execution cycle model for RVV in the VP (Section V-A). We then vary two parameters of this model corresponding to different micro-architectural implementations of RVV. For each parameter combination, we run two common algorithms in non-vectorized and vectorized variants (Section V-B and Section V-C). Finally, we show that by comparing the number of simulated execution cycles, we can derive valuable assessments for the design of RVV micro-architectures (Section V-D).

A. RVV Execution Cycle Model

The *GUI-VP* from [12] already provides a simple instruction accurate timing model that just assigns each instruction a fixed number of execution cycles. In this work, we extend this model since the number of cycles per RVV instruction is not always fixed. In fact, the number of execution cycles depends on the RVV register length `VLEN` and the number of lanes for parallel processing and hence both become parameters.

`VLEN`, as already described, is the length of the 32 RVV registers in number of bits. The number of *Lanes*, models the number of hidden functional elements that can be used for parallel operations by the HW. Each *Lane* can process one bit. For example, if we add two 32 bit wide vector registers, RVV with 16 *Lanes* will take twice as many execution cycles as RVV with 32 *Lanes*.

B. Algorithm 1: PNG Average

The first algorithm considered in our case study is one of five filter types used in the decompression of the common image format *Portable Network Graphic* (PNG), namely *PNG Average* [21]. *PNG Average* is applied to pixel rows of an image. For each pixel in a row, the algorithm calculates the mean of the left (previous pixel) and upper neighbor (pixel in previous row) and subtracts this mean from the current pixel. The motivation behind this is to reduce the actual values of pixels by estimated values and thus to improve the compression rate.

The non-vectorized and vectorized implementation variants are adaptations from RVVRadar. RVVRadar is a open-source framework available on GitHub to help programmers in vectorizing algorithms for RVV and was introduced in [22]. It already contains several algorithms and implementations for RISC-V and RVV. However, the vectorized variant of *PNG Average* provided by RVVRadar contains an additional optimization compared to the non-vectorized variant, which eliminates an additional read in the processing loop. Since this would distort our experimental results in favor of the vectorized variant, the optimization is removed for our experiments.

C. Algorithm 2: MAC_32_16_16

The second algorithm considered in our case study is a *Multiply-Accumulate* (MAC) operation. MAC operations

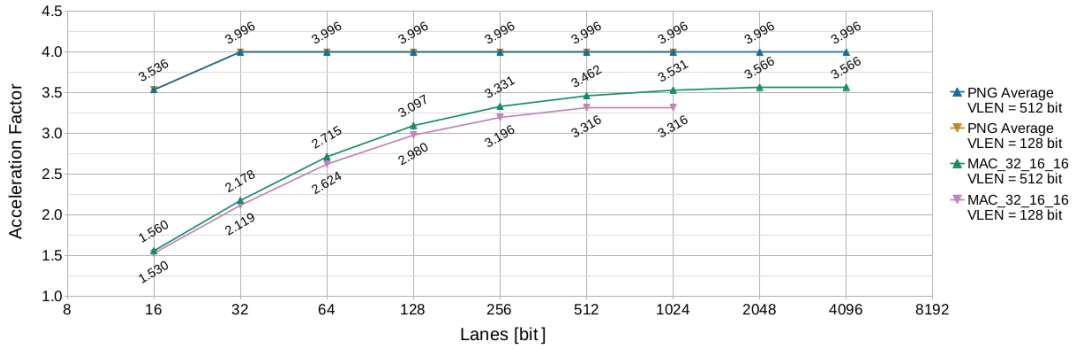

Fig. 4: Acceleration of vectorized compared to non-vectorized Implementations under varying RVV Execution Cycle Model Parameters

combine multiplication and addition and are often found in signal processing algorithms.

The concrete variant *MAC_32_16_16* considered in our experiments multiplies two vectors with 16 bit elements and adds the result in-place to another vector with 32 bit elements.

Similar to *PNG Average* presented in Section V-B, the non-vectorized and vectorized implementation variants for *MAC_32_16_16* are adaptations from RVVRADAR.

D. Results

The results of our experiments are summarized in Fig. 4. The X-axis shows the number of *Lanes* the RVV is configured with. The Y-axis shows the acceleration factor, which is the ratio of the execution cycles consumed by the non-vectorized to the vectorized variants. The exact values of the acceleration factor can be found at the data points of the curves. The four curves show the two algorithms *PNG Average* and *MAC_32_16_16*, and the two RVV configurations for `VLEN`.

Please note that for experiments in which the number of *Lanes* is greater than `VLEN`, *register grouping* has to be considered. Grouped vectors can contain up to $\texttt{VLEN} \times 8$ bits. Consequently, we also use up to $\texttt{VLEN} \times 8$ *Lanes* which results in 1024 for $\texttt{VLEN} = 128$ and 4096 for $\texttt{VLEN} = 512$.

For ***PNG Average*** we see that the maximum acceleration of **3.996** is reached with 32 *Lanes*. Further acceleration with a larger `VLEN` or more *Lanes* is not possible. The explanation for this behavior is that in *PNG Average* parallel processing can only be applied to the color channels within a pixel. This is because the result of one pixel is dependent on the result of the previous pixel. For four channel RGBW color the paralellization is therefore limited to a factor of four.

For ***MAC_32_16_16***, we see a maximum acceleration of **3.566** with `VLEN` 512 and 2048 *Lanes*. The acceleration does not increase linearly with `VLEN` or the number of *Lanes*. This can be explained by the fact that memory accesses dominate over computations in *MAC_32_16_16* and that memory accesses cannot be arbitrarily parallelized. For example, in our model, memory accesses are limited to 64 bits. Therefore, the total cost in execution cycles of loading more than 64 bits with a single instruction (vectorized) or with multiple instructions (non-vectorized) is the same.

It is evident that RVV yields a substantial speedup. For RVV micro-architectures, we can state (i) that larger registers (`VLEN`) or more computational units (*Lanes*) do not automatically lead to a linear increase in performance, and (ii) that special attention should be paid to the memory interface. Overall, both case studies clearly demonstrate the benefits of the RVV enabled *RISC-V VP++* for system-level evaluation.

VI. Conclusions and Future Work

In this paper, we presented *RISC-V VP++*, the first free- and open-source SystemC TLM based VP with support for RVV. We described how RVV, with its **624** instructions was integrated into an existing VP. We also presented our verification chain for the enhanced VP and achieved a functional coverage of **81.44%** according to *riscvOVPsim*. Finally, in our case studies, we demonstrated that (i) RVV can be used to significantly speed up parallizable algorithms and (ii) that our RVV enhanced VP can provide valuable assessments for evaluations at the system level.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.
[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.
[3] M. Flynn, "Very high-speed computing systems," *IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.
[4] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," in *ICS*, 1998, p. 425–432.
[5] R. Lee, "Multimedia extensions for general-purpose processors," in *SiPS*, 1997, pp. 9–23.
[6] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.
[7] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.
[8] "IEEE standard for standard SystemC language reference manual." [Online]. Available: https://doi.org/10.1109/ieeestd.2012.6134619
[9] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.
[10] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009. [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf
[11] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.
[12] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *GLSVLSI*, 2023, pp. 599–605.
[13] "RISC-V V vector extension," https://github.com/riscv/riscv-v-spec, 2022.
[14] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.
[15] "QEMU a generic and open source machine emulator and virtualizer," https://www.qemu.org.
[16] "RV8," https://rv8.io.
[17] "DBT-RISE," https://github.com/Minres/DBT-RISE-Core.
[18] "FORCE-RISCV RISC-V instruction sequence generator (isg)," https://github.com/openhwgroup/force-riscv.
[19] "riscvOVPsim Imperas RISC-V instruction set simulator (iss)," https://www.imperas.com/riscvovpsim-free-imperas-risc-v-instruction-set-simulator.
[20] C. Hazott, F. Stögmüller, and D. Große, "Verifying embedded graphics libraries leveraging virtual prototypes and metamorphic testing," in *ASP-DAC*, 2024.
[21] "Portable network graphics (png) specification (second edition)," https://www.w3.org/TR/PNG.
[22] L. Klemmer, M. Schlägl, and D. Große, "RVVRadar: a framework for supporting the programmer in vectorization for RISC-V," in *GLSVLSI*, 2022, pp. 183–187.