A Pluggable Vector Unit for RISC-V Vector Extension

Vincenzo Maisto
Hensoldt Cyber GmbH, and University of Naples Federico II
Taufkirchen, Germany and, Naples, Italy
vincenzo.maisto@{hensoldt-cyber.de},{studenti.unina.it}

Alessandro Cilardo
University of Naples Federico II
Naples, Italy
acilardo@unina.it

***Abstract*—Vector extensions have become increasingly important for accelerating data-parallel applications in areas like multimedia, data-streaming, and Machine Learning. This interactive presentation introduces a microarchitectural design of a vector unit compliant with the RISC-V vector extension v1.0. While we targeted a specific core for demonstration, CVA6, our architecture is designed so as to ensure extensibility, maintainability, and re-usability in other cores. Furthermore, as a distinctive feature, we support speculative execution and precise vector traps. The paper provides an overview of the main motivation, design choices, and implementation details, followed by a qualitative and quantitative discussion of the results collected from the synthesis of the extended CVA6 RISC-V core.**

***Index Terms*—RISC-V, vector extension, open-source hardware, FPGA, UltraScale+**

I. Introduction

Data-parallel applications like multimedia, data-streaming and most recently Machine Learning and Deep Learning have an intrinsic data-level parallelism which scalar processors cannot exploit. All major ISAs feature packed-SIMD extensions, like ARM NEON [1] and Helium as well as state-of-the-art Intel AVX-512 [2]. Recently, Vector Length Agnostic SIMD extensions have been proposed by both ARM, with SVE [3] and RISC-V, with the V vector extension [4]. This new kind of extensions result in easier to design and more portable software. At the same time, vector extensions allow the ISA itself to be more extensible and forward-compatible.

RISC-V is an open-source, royalty-free ISA from RISC-V International [5]. Its licensing strategy has enabled the open-source philosophy to extend to the hardware domain. Great interest has been shown both by scholars and industry, so that the RISC-V ecosystem has gained a great momentum against competitors. Nowadays, research projects and industrial product using the ISA are countless. The V-extension aims to finally merge a Vector Length Agnostic execution paradigm within a general purpose ISA. The specification has finally reached stability and it is now ready for the ratification of version 1.0.

In this work, a re-usable and efficient implementation of the V-extension is presented. The proposed design is meant to be integrated as a functional unit in a scalar core. Our target core is CVA6 [6]. Nevertheless, our design remains independent of this choice. This approach is aimed to allow the microarchitecture to gain a broad use within the long-term evolution of the RISC-V architecture and open-source implementations. As a distinctive feature, the proposed unit supports out-of-order and *speculative* execution of vector instructions and *precise vector traps*. Furthermore, we show how our design can host arithmetic units from third parties with a low integration overhead.

This paper is structured as follows. Section II gives a high-level description of the designed architecture. Section III describes the microarchitecture of the vector unit. Section IV presents the synthesis results and compares our work to other solutions available from academia and industry. Section V concludes the paper and outlines our next steps.

II. Architecture

We target the Zve64x embedded profile of the vector specification [4]. We also choose not to support the Zvamo extension for the next future. One of our objectives was to come up with a design that can be easily dropped as a functional unit in the execution stage of a scalar core, regardless of its architectural features (in- or out-of-order, precise or non-precise exceptions) and micro-architectural details (scoreboarding, register renaming, etc). For that, dependencies with the scalar core are minimized, together with the necessary microarchitecture modifications. The scalar core handles instruction fetch, branch prediction, speculation, and the rest of the scalar semantics with its own optimizations. All of the vector semantics is, instead, handled by the vector unit.

Instruction data is fed from the scalar core to the vector unit through an *instruction queue*. Together with this data, the execution context of vector instructions also needs the values of vector Control and Status Registers (CSRs) sampled at issue-time. This queue provides synchronization, back-pressure, and decoupling with the scalar core and its implementation depends on its issue protocol. The vector unit is internally divided in three main stages: Sequencer, Execution stage, and Vector Write Back. The Sequencer accepts instructions from the queue and forwards them to the Execution stage, which hosts the Vector Register File (VRF) and the vector functional units. These units can execute out-of-order and interact with the VRF, reading and writing in SIMD fashion their operands. The datapath width is VLEN. Write back to the scalar microarchitecture is performed by the Vector Write Back Module. Figure 1 gives a high-level view of the vector unit.

Our design supports *speculative execution* of vector instructions and *precise vector traps*. This potentially results in performance improvements in case of vector instructions which can generate exceptions. Such instructions can execute

978-3-9819263-6-1/DATE22/©2022 EDAA

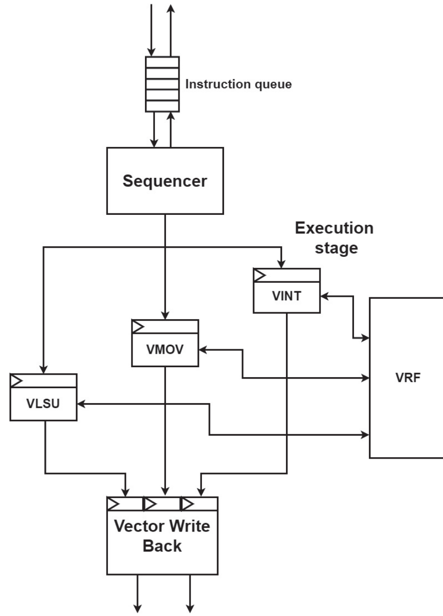

Fig. 1. Block Diagram of the vector unit.

for a long time before trapping (e.g., a page fault), and with support for speculation more recent vector operations do not have to stall. Precise vector traps extend the RISC-V exception model with vector semantics thanks to the precise setting of the `vstart` CSR. Such feature allows Supervisor-mode software to easily identify the exception cause and avoid spurious address translations and memory accesses in case of demand-paging. The internal microarchitecture of the vector unit is a microprogrammed SIMD pipeline. This choice was preferred to a lane-based architecture, with the aim of minimizing pressure on the VRF, simplifying data allocation and instruction sequencing, and having a uniform execution model for vector instructions. This simplification resulted also in an easy-to-extend sequencing logic, which made it possible to implement different kinds of vector instruction classes incrementally, e.g., single-width, mixed-width, reductions, cross-element operations, etc.

A. Extending a Scalar RISC-V Core

This subsection focuses more specifically on the features that a scalar core must support and the modifications that are necessary to integrate the vector unit. This integration requires the scalar core to decode and handle vector instruction dependencies with the baseline architecture and, hence, allocate, read, and write scalar registers. Furthermore, there are also syntactic dependencies from and to vector CSRs with vector instructions which need to be enforced. Also, an execution and a write-back port must be provided. Finally, memory interactions of the vector unit and the baseline core need to be synchronized.

Our target scalar core is CVA6, v4.2 [7], a single-issue in-order RV64GC implementation, supporting Sv39 virtual memory model, partial use of register renaming, and out-of-order write-back. A ReOrder Buffer (ROB) is extended with some scorebording functionalities, becoming the heart of the architecture. Minimising microarchitectural and source code impact was our key requirement. The first two stages of CVA6's pipeline were left untouched since they had no interaction with the vector extension. The *Decoding* stage was extended with a *Vector Pre-Decoder* in parallel with the original logic. Here, for vector instructions, only those details useful to the scalar core are decoded, while the rest of the decoding is delegated to the vector unit. Most of the extensions involved the *ROB*. This data structure handles the Issue, Read Operands, Write-Back, and Commit phases in CVA6's pipeline. *Operand forwarding* was updated for scalar operands (integer and floating-point registers) to be forwarded from and to vector instructions. Stalls were introduced for vector instructions in case of vector CSRs update operations. The *Write-Back* logic was modified by adding a new write-back port with additional metadata necessary to update the vector CSRs. The *Commit* logic was extended to feed the vector unit with details about retiring and trapping instructions (e.g., for Configuration instructions or partial execution of vector operations). The *CSR register file* was enlarged with the new CSRs, which each vector instruction can now update at commit-time. Additional fields for V-extensions were enabled in the `MSTATUS` and `MISA` CSRs. The modifications to the memory architecture are not discussed in this paper.

III. Detail of the vector unit microarchitecture

This section describes in more detail the microarchitectural level of our vector unit. We present the single submodules and analyze the most interesting challenges we addressed.

A. Sequencing stage

The Sequencing stage takes as input one instruction per cycle through the data structure prepared by the scalar core in the instruction queue. Here, vector instructions are completely decoded. In case of encoding exceptions, write-back to the scalar core is requested to Vector Write-Back (see Section III-D). Vector configuration instructions execute in this module right after decoding, since they do not need to interact with the VRF and do not need any sequencing. Also, for these instructions, write-back can be immediately requested. Furthermore, since these instructions update those vector CSRs which actually configure the behaviour of the hardware, no vector instruction can be accepted by the instruction queue until a pending configuration operation retires and the vector CSRs are updated. The rest of the instructions are sequenced and dispatched to the vector functional units in the Execution stage. Each vector instruction is sequenced into EMUL sequential micro-operations. Each of these is a SIMD operation, working at the single vector register granularity. Instructions are, then, dispatched to the functional units in the next stage. Back-pressure from these units and configuration hazards (see Section III-B2) are taken into account.

B. Vector Register File

The VRF is a multi-banked SRAM memory, arbitrated by an allocator against the requests of each functional unit. The

allocator acts as a lock server and implements a *custom locking protocol* designed to track data dependencies between micro-operations.

1) Locking Protocol: Data hazards handling is based on the *two-phase shared lock protocol*, applied at the micro-operation level. In order to allow *chaining* from different functional units, each vector register group operand cannot be locked at once and each vector register has to be released as soon as it is no longer needed. To ensure that no micro-operation from a more recent instruction overruns the ones executing on a slower functional unit, each micro-operation has to also acquire the locks for the next one, so as to stall the faster functional unit. Please note that the sequentiality between micro-operations of the same instructions is used to enforce stalls in case of data hazards.

It follows that each micro-operation *i* reads its own source vector registers, acquires the locks for micro-operation *i+1*, and releases its read locks only once both of the former operations have completed. Once write-back is performed, the write lock can be released. There are only few exceptions where chaining has to be disabled. Some instructions need to lock a whole vector register group at once because of irregular data dependencies (e.g., `vslide{1}downs.vx` and `vrgather{ei16}.vv`) and to release each lock when it is no longer necessary. Re-startable instructions are not supposed to execute on elements whose indices are less than `vstart`. Therefore, the corresponding micro-operations can skip the reading and writing of their operand, but the locks have to be acquired and released following the protocol to ensure consistency. The same reasoning also applies for reductions. For such operations, only one element in the vector register group `vs1` has to be actually read and only one has to be written for the group of `vd`. Nevertheless, all the registers in the group have to be locked and unlocked as usual.

2) Configuration Hazards: Together with data hazards also configuration hazards have to be handled. Such hazards can occur when an older instruction is executing with a $runningEMUL \geq 1$ on one of its operands and a more recent instruction: 1) has an $EMUL < runningEMUL$; 2) has an overlap with a vector register group operand of the older instruction; 3) the overlap is not on the first vector register of the group. This situation is detected by the dispatching logic in the Sequencing stage and the new instruction is stalled until the older one completes. This is not the optimal solution, but we consider it the best trade-off between design complexity and performance.

3) Support for Speculative Execution: Some of the available V-extension implementations only execute vector instructions when they are not speculative, causing significant performance loss. Therefore, we identified speculative execution as a key requirement. We chose to support speculation via *register renaming* and *checkpointing,* discarding roll-back techniques as we consider them too complex for vectors. Each vector register is composed by multiple copies, where only one holds the non-speculative architectural state, while the others are either invalid or hold a speculative value. Speculative execution is useful in two cases: exceptions and branch prediction. In case of an exception in a precise architecture, all the instructions following, in program order, the offending one have to be flushed, while all those preceding it have already retired. Therefore, a single flush signal is necessary. In case of branch prediction, instead, a selective flush logic, similar to OVI's [8] kill signal, is necessary. The reason is that there may be pending instructions, preceding the branch in program order, which must not be flushed. Since CVA6 currently has a speculating window of zero instructions in case of branch prediction, this feature is not supported yet. This is the only tight dependency between CVA6 and the vector unit so far.

C. Execution Stage

This stage hosts the vector functional units and the VRF. It has a regular structure and has been designed for extensibility. Each functional unit is composed by three modules common to all of them, micro-operation queue, SIMD Read Operands, and SIMD Write-Back, and a single function-specific entity, SIMD Execute. The first three modules are encapsulated into an opaque wrapper, which offers a simpler and unified interface to the SIMD Execute module. This design approach was aimed to allow third parties to develop arithmetic units which can, then, be integrated without any overhead caused by the complexity of the vector unit design.

The *micro-operation queue* is an FSM controller holding the data prepared by the Sequencing logic and issuing single micro-operations to the *SIMD Read Operand* module. The latter is responsible for preparing the operands for the current micro-operation following the locking protocol. The prepared data is issued to the SIMD Execution module, which can then send the result to *SIMD Write-Back*. The latter performs the write requests to the VRF and is optimized to minimize interactions with it. The whole process is repeated for each micro-operation, until completion, when a write-back to the scalar architecture is needed (see subsection III-D). Currently, the design features three parallel vector functional units, which can execute and write back out-of-order:

1) Vector Integer Unit: performs all the integer and fixed-point computations and integer reductions;

2) Vector Permutation Unit: responsible for all the permutation and mask instructions;

3) Vector Load/Store Unit: executes the vector memory access instructions and supports virtual memory. It requires a single external MMU and the number of translation requests is minimized for each instruction. Memory requests are composed along its internal pipeline and sequenced to the memory interface. There is no Vector Store to Vector Load internal forwarding, due to the complexity of vector address overlap, even with simple unit-stride instructions.

D. Vector Write Back

This module is the last stage of the vector pipeline. It is a static priority, combinatorial allocator, responsible for communicating with the Write-Back ports of the scalar microarchitecture. The interface is not strictly dependent on CVA6's Write Back interface. A configurable number of Write-Back ports assigned to the vector unit is allocated against the requests

from the vector functional units and the Sequencing stage. Together with scalar write-back data, commands to update the contents of the vector CSRs are also provided, (e.g., set `vstart` or set `vl` and `vtype` fields for vector configuration instructions).

IV. Evaluation

This section evaluates the synthesis results on a Zynq UltraScale+ device, namely XCZU9EG, with the Xilinx Vivado IDE. The synthesized architecture features an extended version of CVA6 with the vector unit as well as a prototypical scratchpad memory, which is equivalent to a cache having hit probability of 1 in case of vector memory accesses providing a maximum bandwidth of VLEN/2 bits per cycle, and an external SRAM memory. The vector unit was described in SystemVerilog HDL for a total of 10k LOCs, while the extension of CVA6 required less than 400 LOCs.

Synthesis results for different values of VLEN are presented in Table I.

TABLE I
Synthesis report for various VLEN on XCZU9EG

VLEN	LUT	FF	max freq. (MHz)
0	39.5k	21.7k	112
128	72.4k	27.8k	100
256	94.8k	31.2k	84
512	136.5k	37.9k	75

From this table we observe a linear trend for LUTs and FFs and some moderate impact of the extension size on the estimated maximum frequency. VLEN=0 denotes a different configuration, where CVA6 does not integrate the vector unit.

We optimized an internally developed reference implementation of the XTS-AES cipher [9] relying on the RISC-V V-extension. The resulting implementation requires a VLEN$\geq$256 instance of the vector unit and makes massive use of permutation instructions, like `vrgatherei16.vv`. We were able to perform a block-level vectorized elaboration of 64 128-bit blocks observing a 2.9x speed-up in encryption for VLEN=256, and 4.1x for VLEN=512. This configuration runs the scalar and vectorized versions of the cipher at the same frequency.

Re-running a scalar XTS-AES implementation at 112MHz on CVA6 alone and comparing it with the vectorized execution described above, we still observed a 2.2x speed-up for VLEN=256 and 2.7x for VLEN=512.

A. Discussion

Our design is not tightly coupled with a scalar core or designed from scratch together with it, unlike other works [10]–[14]. It requires both minimal source code and microarchitectural impact on the scalar core. This makes our design easily re-usable in different cores. Differently from previous works in academia [10] and in industry [11], our design supports speculative execution of vector instructions and precise vector traps. The internal microarchitecture of the vector unit is a micro-programmed SIMD pipeline, similarly to SiFive's DLEN concept [12] [13] and Alibaba's two 128-bit Vector Engines [14]. This choice was aimed to minimize pressure on the VRF and to solve one the main issues observed in a previous work [10], i.e. cross-lane operations. The latter are expected to be very frequent in use-cases like classical and post-quantum cryptography. Our design treats such operations no differently than other operations, causing no unnecessary stalls. Furthermore, our implementation is also more efficient than SiFive's for `vrgather` and `vcompress`, as the whole SIMD datapath is used on average, while in [12], [13] only one element per cycle can be computed, regardless of the effective element width (EEW). Finally, the SIMD architecture presented here allowed us to rely on a single external MMU, instead of having a TLB for each lane, as in previous work [15].

V. Conclusions and Future Work

In this paper, we presented and evaluated a RISC-V based vector unit. Our design approach allows a scalar core to easily integrate the vector unit with minimal overhead. We support speculative and out-of-order execution of vector instructions with precise vector traps. We described the most interesting microarchitectural challenges we addressed in our work and the solution we designed.

The presented microarchitecture was designed for extensibility from the ground up. Therefore, it already offers the opportunity for enhancements such as: 1) leveraging the extensibility of the design to easily integrate an array of scalar floating-point units to support vector operations; 2) extend our register renaming logic to also resolve data hazards more efficiently; 3) extend our flushing logic with selective flushes for more aggressive speculation. As a future work, we will also investigate an integrated scalar/vector memory architecture, which is the primary bottleneck in vector processing. To reduce dependencies with the scalar core, we plan a L1 D$ bypass to a vector-aware and re-usable L2$ with an AXI or TileLink compatible interface.

References

[1] ARM®, "Introducing NEON™", 2009
[2] Intel®, "Intel® 64 and IA-32 Architectures Software Developer's Manual", 2019.
[3] Nigel Stephens et al.:, "The ARM Scalable Vector Length", 2017.
[4] "RISC-V 'V' Vector Extension", https://github.com/riscv/riscv-v-spec/.
[5] RISCV-V International, www.riscv.org
[6] Florian Zaruba and Luca Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology", 2019.
[7] "CVA6 v4.2", https://github.com/openhwgroup/cva6/releases/tag/v4.2.0
[8] SemiDynamics Technology Services SL, Roger Espasa, Pedro Marcuello, Alberto Moreno and Sebastiano Pomata, "AVISPADO - VPU Interface, Version 1.03", 2021.
[9] IEEE Computer Society, "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices", 2018
[10] Matheus Cavalcante et al., "Ara: A 1 GHz+ Scalable and Energy-Efficient RISC—V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI", 2019.
[11] Mike Demler, "Andes plots RISC-V Vector heading", 2020.
[12] SiFive, "SiFive P270 Core Complex Manual, 21G2.01.00", 2021
[13] SiFive, "SiFive X280 Core Complex Manual, 21G2.01.00", 2021
[14] C. Chen et al., "Xuantie-910: Innovating Cloud and Edge Computing by RISC-V", IEEE Hot Chips 32 Symposium (HCS), 2020
[15] Roger Espasa et al., "Tarantula: A Vector Extension to the Alpha Architecture", 2002.