A Minimal RISC-V Vector Processor for Embedded Systems

Matthew Johns, Tom J. Kazmierski
School of Electronics and Computer Science
University of Southampton
Southampton, United Kingdom
mrj1g17@soton.ac.uk, tjk@ecs.soton.ac.uk

***Abstract*—This paper presents the first RISC-V vector processor design aimed at microcontrollers that uses the new RISC-V 'V' extension for vectors, part of the open-source RISC-V instruction set architecture (ISA). Being aimed at small embedded devices, it demonstrates a simpler method of parallel execution than traditional vector architectures to minimise logic. It has been synthesised for testing on an FPGA at 50MHz. Typical vector-compatible applications have been used as benchmarks. Performance has been improved by up to 5.8x for the demonstrated applications relative to a comparable scalar RISC-V processor, for an increase in FPGA resource utilisation of at most 2.6x.**

***Index Terms*—Microcontrollers, Vector processors, Accelerator architectures, Field programmable gate arrays**

I. Introduction

Vector processor architectures are prevalent in many application processors nowadays. However, the growth of technologies such as the Internet of Things (IoT) and autonomous vehicles will require ever-better image processing and machine learning capabilities in smaller packages. These could be provided by compact vector microcontrollers. The open-source RISC-V instruction set architecture (ISA) has a vector extension currently in development which suits this.

This project presents an embedded processor that implements a subset of this vector extension. It is novel for the following reasons:

1) To the authors' knowledge, no implementation of the RISC-V vector extension exists for microcontrollers. The only previous implementation of this extension is an application processor [1]. Furthermore, that implementation uses a coprocessor, whereas this design has vector functionality added as an integral part of the processor.
2) It shows the validity of using a subset of the instructions in the extension, giving performance gains whilst minimising complexity.
3) Comparisons are made between a scalar RISC-V processor and a vector one, rather than between different types of vector accelerators. This aims to show the usefulness of vector architectures in small embedded devices.

This paper is organised as follows. Section II describes current technologies. Section III proposes the new architecture. Section IV evaluates its performance. Lastly, Section V concludes the work.

978-1-7281-8928-4/20/$31.00 ©2020 IEEE

II. Background

RISC-V is an open-source instruction set architecture (ISA) originally developed by researchers at the University of California, Berkeley. The architecture specifies a base instruction set needed for compliance and multiple instruction extensions [2]. One of these is the 'V' extension for vectors, RV32V. This extension is currently not ratified, but a specification is under development on GitHub [3], with the latest stable release, at time of writing, being v0.9. Being so early in development, a vectorising compiler for this extension has not yet been completed.

RV32V is a vector ISA rather than a fixed-width Same Instruction Multiple Data (SIMD) ISA. Vectors are split into parts matching the capacity of the execution stage. Parts are processed in separate cycles until all parts have been operated upon [4]. Vector ISAs are more flexible than fixed-width SIMD for different vector lengths and element widths, producing code as much as 9x smaller for some applications [5].

The 'V' extension adds 32 vector registers to RV32I, v0-v31, as well as six control and status registers (CSRs). More than 160 instructions are added. However, not all instructions are worth implementing on all architectures. The specification in [3] states that although 'platform profiles' for RV32V have not yet been defined, it may be that embedded platforms implement only a subset of the instructions. For example, many embedded devices have no floating-point unit (FPU), prohibiting floating-point operations. The subset concept is developed in this design.

Arm and x86 architectures have used fixed-width SIMD for many years, with the former being the most useful comparison due to being another RISC architecture. Arm NEON was shown to offer a 5x speed-up for an edge detection algorithm [6] but is designed for application processors. For microcontrollers, Cortex-M4 and M7 cores have a DSP extension [7] using the general-purpose 32-bit registers. SIMD instructions operate on either four 8-bit values or two 16-bit values.

Arm has also developed vector architectures. The Scalable Vector Extension is designed for high-performance computing (HPC) and has similarities with RV32V, such as leaving the vector register length implementation dependent [8]. Being

designed for HPC, the minimum length is 128 bits, likely too large for small microcontrollers. Arm also has the M-Profile Vector Extension (MVE), known as 'Helium' [9], designed to speed up processing by 4x for a 2x increase in datapath width. Multiple-issue arithmetic is avoided using a 'beat' system, where instructions that use separate blocks of hardware happen simultaneously.

RISC-V vector cores are not yet as common. The Hwacha decoupled accelerator [10] has been taped out multiple times, but predates work on the official RISC-V vector extension and uses its own instruction set. Hwacha was built around the open-source Rocket core, a 64-bit application processor. There is currently only one published implementation of the official 'V' extension: Ara [1], a tightly-coupled coprocessor, again connected to a 64-bit application core. Ara is based on RV32V v0.5. Application processors aim at higher performance and the typical lane-based vector architecture reflects this, but for microcontrollers an implementation like Ara requires too much support logic such as a despatcher and lane control logic, all of which add power and area whilst providing more performance than may be required.

III. Proposed Architecture

This section introduces the architecture of the processor. It is aimed at microcontrollers, where simplicity is the key to meeting area and power requirements. The processor uses the specification RV32E, which contains the same instructions as RV32I but with only 16 x-registers [2]. A subset of the instructions from RV32V v0.8 has been implemented to demonstrate the concept of a minimal implementation, comprising only the required arithmetic, memory, move and slide instructions. A block diagram of the architecture is shown in Fig. 1.

Previous designs have taken the form of coprocessors [1] [10], built around application processors more complex than those used in microcontrollers. This adds the complexity of interfaces between scalar and vector parts. Here, vector functionality is as tightly integrated into the processor as scalar functionality, avoiding such interfaces. The processor executes in-order and is single-stage. Rather than queuing multiple instructions into execution lanes, instructions are executed one at a time, removing the need for a scheduler. Therefore, it executes instructions in a SIMD manner, but with the software freedom of vectors. The lack of a pipeline structure gives the flexibility to execute instructions over multiple cycles depending on the operation and vector length without needing to stall or detect hazards. Pipelining could be added to increase the maximum frequency but in this case was not required to reach the target frequency.

A summary of design parameters is given in Table I.

TABLE I
Design parameters.

Clock frequency	50MHz
RAM size	192KiB
RAM port width	32b
Program memory size	1KiB
Supported element widths	8b, 16b, 32b
Vector register length (VLEN)	32b
Striping length (SLEN)	32b

A. Vector ALU

The traditional vector architecture comprises multiple identical lanes for parallel execution, each with the same width as the maximum data size [1] [10] [11]. In this 32-bit design, multiple 32-bit lanes would be required. This would consume a prohibitively large area of silicon for small microcontrollers, especially if each lane had a hardware multiplier. It would also require a large datapath to keep all lanes full. To counter this, a smaller solution is presented based on variable-width processing units (PUs), shown in Fig. 2. Each PU acts as an independent scalar integer ALU. This includes a multiplier, wired to enable single-cycle multiply-adds.

With the inputs fixed at 32 bits per operand, one cycle can run either a single 32-bit operation, two 16-bit operations, or four 8-bit operations. This does not require four 32-bit lanes; it needs only a single 32-bit PU, a single 16-bit PU, and two 8-bit PUs. The extent to which operations can be parallelised therefore depends on the size of the elements used. For instructions using vector register groups, one register is operated on per cycle, meaning arithmetic operations will last between 1 and 8 cycles. Although some PUs sit idle when the element width does not allow their use, the simplicity of this design is a major benefit for small devices.

Mapping logic is added to the inputs and outputs of the PUs to combine elements based on their width. For scalar instructions, the 32-bit PU is selected as the only output. This minimises the amount of dedicated hardware required for the vector extension. Fig. 2 also shows the 'slide mux', a combinational logic block used for vector slides. Vectors are shifted by the logical-shifter in the 32-bit PU, and the slide mux concatenates the new element onto the shifted data.

B. Vector Registers

Rather than have an individual vector register file (VRF) for each lane as in [1], this design avoids replication by using a single central VRF. The vector register length matches the datapath width, simplifying the transfer of data to the ALU as one register is transferred per cycle. Write-enable signals divide each register into four bytes, giving write control over each element. The VRF uses four 32-bit read ports and two 32-bit write ports. This allows widening instructions to take place at the same rate as single-width instructions. The ALU output to the VRF write ports is 64-bit to facilitate this.

The specification requires 32 vector registers. However, 16 are capable of supporting operations on all three element widths, plus the maximum vector register group of 8 registers. In this design, expanding the VRF from 16 to 32 registers doubled not only the number of registers used, but also the amount of combinational logic. This design therefore implements only 16 vector registers. Although the smaller number partially limits performance, gains were still observed.

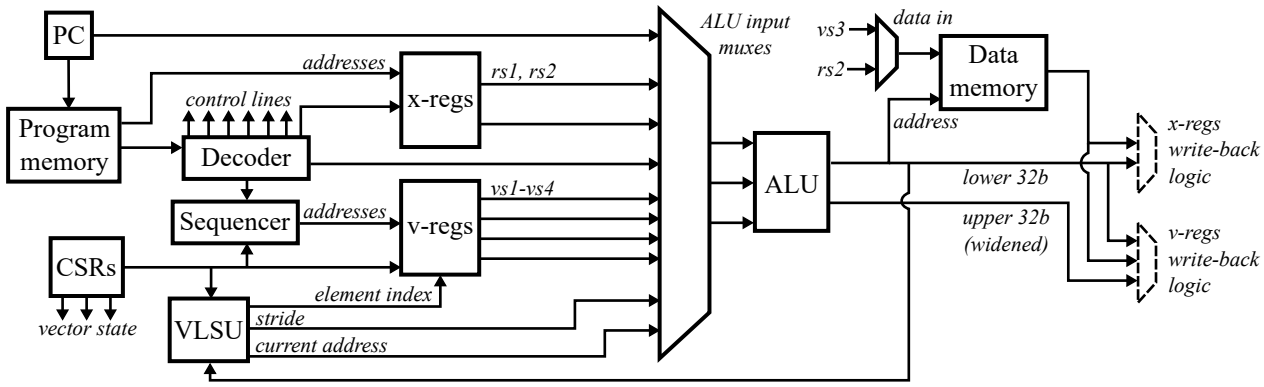

Fig. 1. Block diagram of the vector processor.

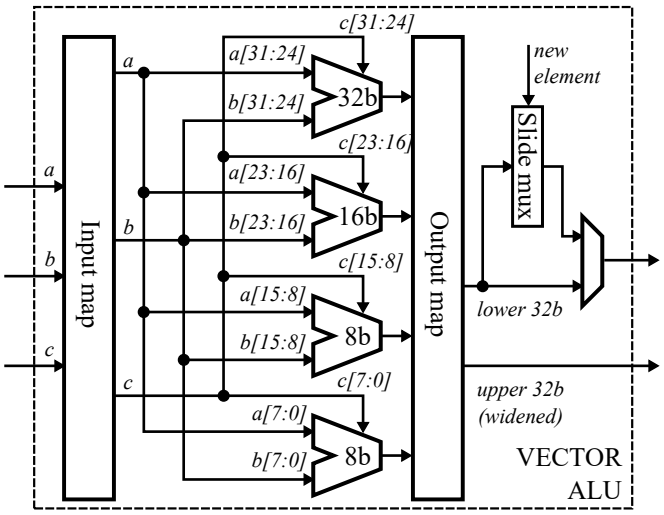

Fig. 2. Block diagram of the vector ALU, showing the mapping of inputs into the processing units (PUs) for 8-bit elements. Elements going into wider PUs are sign- or zero-extended as appropriate.

When 'platform profiles' are added to the RV32V specification before ratification, it may be useful to allow smaller devices to use a smaller number of vector registers, similar to how RV32E allows a smaller number of x-registers than RV32I.

The need for vector registers to be read or written over multiple cycles requires separate logic to determine register addresses in each cycle. This is provided by the sequencer module. The effect depends on the instruction. For example, during arithmetic operations, addresses are taken from the instruction and incremented each cycle. However, in memory operations, the next register is only moved to once the previous one has been filled with elements, meaning addresses do not change every cycle.

C. Vector Memory Operations

The memory port has been constrained to 32 bits. Being aimed at microcontrollers, there is no memory controller. Data is loaded and stored when needed before execution continues. The memory system is designed for the FPGA used, allowing single-cycle accesses. Under these conditions, a load takes two cycles: one to calculate the address and one to read. By using vector load instructions, elements are loaded in batches, allowing loads to be chained. When an element is being read, the ALU is used to calculate the address for the next element. Therefore, when reading N elements, $N+1$ cycles are used, rather than $2N$ for scalar loads. The logic for controlling this is the Vector Load/Store Unit (VLSU). It counts the elements and holds the calculated addresses, whilst sending the current element index to the vector registers to set the write-enable signals. It also selects the offset for each address, either the element width in bytes or the stride specified by the instruction. The ALU is reused for calculating addresses rather than a dedicated adder inside the VLSU, sparing a 32-bit adder.

IV. Evaluation

A. Methodology

Three benchmark applications suitable for vector optimisation are used to measure the performance of the processor:

1) The conversion of an RGB image to greyscale by right-shifting and summing the individual colour components.
2) Filtering a greyscale image with an edge detection filter.
3) An integer matrix multiplication of two 120x120 matrices of 8-bit elements.

1) and 2) each use a 256x256 image. 2) applies a 3x3 filter F to an image I to produce a filtered image Q. The value of any filtered pixel with position (x, y) is given by (1).

$$Q(x,y) = \sum_{i=1}^{3}\sum_{j=1}^{3} I(x+i-2, y+j-2)F(i,j) \quad (1)$$

Due to the current lack of a C compiler supporting the RISC-V 'V' extension, programs were written in assembly and optimised for the architecture.

For comparison, these operations have also been run on a baseline scalar processor with RV32E instructions. This processor uses a similar architecture to the proposed vector processor except without all vector functionality. As such, all instructions are single-cycle except memory loads, which

TABLE II
BENCHMARK EXECUTION TIMES IN MILLISECONDS, WITH SPEED-UP FOR THE VECTOR PROCESSOR RELATIVE TO THE BASELINE.

Benchmark	Scalar	Vector	Speed-up
Greyscale	21.0	7.7	2.7x
Filter	115.8	35.6	3.2x
Matrix multiply	347.9	59.9	5.8x

again take two cycles. Its frequency is the same as the vector processor and a scalar multiply instruction has been added from the 'M' extension to give comparable results. Execution times for each benchmark are measured in an RTL simulation of the processor in the ModelSim software.

To verify the processor is synthesisable, these operations have been tested by synthesising the design on an Intel Cyclone V 5CSEMA5F31C6 FPGA, mounted on the Terasic DE1-SoC board. The processor was run on the FPGA with a clock frequency of 50MHz. Only on-chip memory was used. Quartus Prime 18.1.0 SJ Lite Edition was used for synthesis.

B. Benchmark Performance

The execution times of each of the benchmark applications on the vector processor and the baseline scalar reference processor are shown in Table II. There is considerable speed-up when using this vector processor compared to the baseline.

There are three main sources of this performance improvement. The first is the use of SIMD instructions to parallelise programs. The benchmark programs use 8-bit elements, meaning parts of the calculation operated at a 4x speed-up. With multiply-add instructions also taking a single cycle, in some places this would be an 8x speed-up over the baseline processor, where separate multiply and add instructions were required. Secondly, the more efficient memory loads vastly reduced the number of cycles spent retrieving vectors from memory. Thirdly, especially in the filter algorithm, the large amount of space provided by the vector registers and the ability to slide vectors aided in the reuse of data already loaded, thus reducing the overall number of load instructions needed.

These performance results show the value of vector processing for certain applications, even with a small number of vector instructions and a minimal implementation.

C. FPGA Resource Utilisation

The overhead for adding this vector functionality to the processor is shown by the resource utilisation values in Table III. The DSP blocks were used only by the ALU. Of the Adaptive Logic Modules (ALMs), approximately 30% each were used by the ALU and vector registers. These modules both required large amounts of combinational logic for mapping each element to the correct position.

The overall increase in resource usage is at most 2.6x compared to the baseline. Whilst this may remain prohibitive for the smallest or lowest-power designs, the performance improvements that result will be enough to justify the area overhead in some situations. In the tested applications, the factor increase in performance beats the factor increase in area.

TABLE III
FPGA RESOURCE UTILISATION OF THE VECTOR PROCESSOR AND SCALAR BASELINE PROCESSOR.

Resource type	Scalar	Vector
Adaptive Logic Modules	1055	2751
Registers	739	1372
M10K RAM Blocks	257	257
DSP Blocks	2	5
Average Interconnect Usage	3.1%	5.7%

V. CONCLUSION

This paper presented a RISC-V vector processor design, believed to be the first implementation for microcontrollers of the RISC-V 'V' extension, RV32V. It is also the first example of vector functionality being tightly integrated into the processor. It is a compact design yet demonstrates performance gains in the tested applications, thus showing the advantages of vector processing in microcontrollers. It also shows the validity of only using a subset of instructions from the vector extension. This design has been synthesised on an FPGA and operates correctly at 50MHz. Compared to a scalar baseline processor, measured performance gains ranged from a 2.7x speed-up to a 5.8x speed-up for the vector processor. This came at an increase in FPGA resource utilisation of no more than 2.6x.

REFERENCES

[1] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI," *IEEE Trans. VLSI Syst.*, vol. 28, no. 2, pp. 530–543, Feb. 2020.

[2] A. Waterman and K. Asanovic, *RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Version 20190608-Base-Ratified*, (2019). [Online]. Available: https://riscv.org/specifications

[3] RISC-V. "Working draft of the proposed RISC-V V vector extension". GitHub. [Online]. Available: https://github.com/riscv/riscv-v-spec

[4] Y. G. et al., "A vector coprocessor architecture for embedded systems," in *Proc. 2011 Int. SoC Design Conf.*, 2011, pp. 195–198.

[5] D. Patterson and A. Waterman. "SIMD Instructions Considered Harmful". ACM SIGARCH. [Online]. Available: https://www.sigarch.org/simd-instructions-considered-harmful

[6] K. Z. et al., "A high performance real-time edge detection system with NEON," in *Proc. 2017 IEEE 12th Int. Conf. ASIC (ASICON)*, pp. 847–850.

[7] T. Lorenser. (2016) "The DSP capabilities of ARM Cortex-M4 and Cortex-M7 Processors (white paper)". Arm. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/white-paper-dsp-capabilities-of-cortex-m4-and-cortex-m7

[8] N. S. et al., "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar./Apr. 2017.

[9] T. Grocutt. "Making Helium: Why not just add Neon? (1/4)". Arm. [Online]. Available: https://community.arm.com/developer/research/b/articles/posts/making-helium-why-not-just-add-neon

[10] Y. L. et al., "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *Proc. ESSCIRC 2014 - 40th Eur. Solid State Circuits Conf. (ESSCIRC)*, pp. 199–202.

[11] C. E. Kozyrakis and D. A. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov./Dec. 2003.