Design and Implementation of RVV-Lite: A Layered Approach to the Official RISC-V Vector ISA

by

Caroline White

B.A.Sc., University of Waterloo, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Design and Implementation of RVV-Lite: A Layered Approach to the Official RISC-V Vector ISA

submitted by **Caroline White** in partial fulfillment of the requirements for

the degree of **Master of Applied Science** in **Electrical and Computer Engineering**

Examining Committee:

Guy Lemieux, Electrical and Computer Engineering
Supervisor

Miesko Lis, Electrical and Computer Engineering
Supervisory Committee Member

Andre Ivanov, Electrical and Computer Engineering
Supervisory Committee Member

Abstract

The open-source RISC-V Vector extension (RVV), whose specification was frozen in 2021, comprises over 400 instructions, with four integer and two floating-point data types. Its purpose is to accelerate applications in high-performance computing. Since it is the largest optional extension to the RISC-V ecosystem, it is prudent to assess the implementation cost of these instructions with respect to their value add, particularly in cost-sensitive embedded and domain-specific applications. Some instructions are never used by certain applications, while others create unique and potentially costly hardware requirements. They can instead be replaced by simpler instruction sequences. We propose RVV-Lite, a partitioning of RVV. This partitioning allows users to deploy a smaller implementation with a predefined subset of the instructions, which is often needed in embedded or domain-specific applications with limited area or power. The rationale behind the instruction groupings is explained, and implementation results are shown to help make informed choices about the cost of these incremental implementations. To simplify software management, we suggest reducing the number of possible configurations by subdividing primary instructions into 7 layers, while presenting 5 orthogonal extensions that can be added at any point. Instructions excluded from the primary and orthogonal instruction subsets are placed into a final layer of instructions, bridging the gap between RVV-Lite and RVV 1.0. With all layers and options, RVV-Lite consists of 280 instructions, resulting in the exclusion of 127 instructions from the Zve* embedded options proposed by the RVV specification. To demonstrate efficacy of this subset, we present cycle counts for common benchmarks and compare these, along with area and timing results, to other RISC-V Vector engines.

Lay Summary

RISC-V, an open-source computing instruction set, recently froze its specification for a vector instruction extension which adds parallel compute capabilities to conventional central processing units (CPUs). This allows for faster compute times when implementing applications like machine learning and computer vision, which contain many identical computations with little dependence between them. For embedded systems with limited space, like IoT devices, implementations of the full RISC-V Vector (RVV) specification would be large. Accordingly, we propose RVV-Lite, which eliminates less frequently used or very costly instructions, and subdivides the remaining instructions into 12 groups of instructions with related functionality: 7 are considered priority-ordered layers, and 5 are feature-enhancing optional extensions. This allows users to reap the benefits of the full RVV specification at a fraction of the cost.

Preface

The original ALU discussed in this thesis was designed by Farid Chalabi. The first attempt at an RVV-Lite prototype was implemented by Fredy Alves. Since their original contributions, most of the ALU implementation has been heavily modified, the RVV-Lite prototype was replaced by a new implementatoin, and the RVV-Lite layering approach has been redesigned several times in response to evolving implementation results.

Table of Contents

Appendices

List of Tables

List of Figures

Existing Terms

ASIC Application-Specific Integrated Circuit. An immutable, manufactured chip created for a specific purpose.

AVL Application Vector Length (software).

CFU Custom Functional Unit. A standardized port designed to interface between scalar host processors and accelerators.

DMA Direct Memory Access.

DSP Digital Signal Processing. On an FPGA, these blocks are used to perform multiplication and division quickly.

EEW Effective Element Width.

ELEN Maximum Element Length.

EMUL Effective Register Group Size.

EQ Equal.

FPGA Field Programmable Gate Array. A reconfigurable chip used to mimic functionality of logic circuits.

FPU Floating-Point Unit.

FXP Fixed-Point.

GEMM General Matrix Multiplication.

ISA Instruction Set Architecture. The set of instructions that a given computer processor is able to recognize and process.

LE Less than or Equal to.

LMUL Register Group Size.

LT Less Than.

LUT Lookup Table. One of the basic building blocks in an FPGA.

MAC Multiply-Accumulate.

MADD Multiply-Add.

NE Not Equal.

RVV RISC-V Vector Extension.

SEW Selected Element Width.

SIMD Single Instruction Multiple Data. A type of instruction that processes multiple data elements at the same time, in order to accelerate processing time.

VL Vector Length (in elements).

VLEN Vector register size (in bits).

VLMAX The maximum number of elements in a vector register group. VLMAX = LMUL * (VLEN/SEW).

Zve* 5 versions of RVV targeted to embedded applications. They include varying levels of datatype support and remove support for instructions requiring a full 64x64-bit multiplier.

New Terms

ADD Signed and unsigned addition.

ADDSUB Signed addition or subtraction.

ADDSUBU Unsigned addition or subtraction.

AGU Address Generation Unit.

Be Byte enable signal.

DATA_WIDTH Data width of the vector engine lanes.

DIVREM Signed and unsigned division and remainder operations.

EQ Equality operation (equal/not equal).

EW Element Width (8, 16, 32, 64).

FOP Floating-Point Operation (add/sub/mul/div/min/max).

GTE Greater Than operation (gt/gte).

IW Index Width (8, 16, 32, 64).

K Register group size (1, 2, 4, 8).

KEW Register group and EW pairings 1re8, 2re16, 4re32, 8re64.

LOP Logical Operation (and/or/xor).

LTE Less Than operation (lt/lte).

MAX Signed and unsigned maximum.

MCMP Mask Comparison (lt/lte/eq/ne).

MIN Signed and unsigned minimum.

MINMAX Signed min/max operations.

MINMAXU Unsigned min/max operations.

MOP Logical Mask Operation (and/nand/andnot/xor/or/nor/ornot/xnor).

MUL Signed and unsigned multiplication.

MULH Signed, unsigned, and signed-unsigned multiply-high instructions.

NYI Not Yet Implemented.

ROP Reduction Operation (sum, and, or, xor, maxu, max, minu, min).

S Signed.

SGT Set Greater Than operation (signed and unsigned).

SLT Set Less Than operation (signed and unsigned).

SUB Signed and unsigned subtraction.

U Unsigned.

VLD Vector Load.

VRegFile Vector Register File.

VSEW Vector engine SEW setting.

VST Vector Store.

Acknowledgements

I would like to thank my supervisor, Guy Lemieux, for introducing me to this problem, and providing guidance and support throughout the planning, development, and testing stages of this thesis project.

I would like to acknowledge the funding received through the NSERC Discovery Grant.

I would also like to thank my labmates, both past and present, for their debugging support and general camaraderie throughout this degree.

Chapter 1

Introduction

The RISC-V Vector Instruction Set (RVV) was developed with the intent of adding parallel compute capabilities to a popular open-source ISA. It was first frozen in September 2021, and contains over 400 individual instructions and six unique data types (8/16/32/64-bit integer, 32/64-bit floating point) [8].

Due to the number of supported instructions and data types, in addition to the vector extension being the largest RISC-V extension, implementation of the full specification with support for large vectors would be area- and therefore power-intensive. This is not ideal for embedded systems, which frequently have hard constraints on area and power usage. This is particularly true for embedded systems implemented on Field-Programmable Gate Arrays (FPGAs), as unlike their Application-Specific Integrated Circuit ASICs) counterparts their area constraints are immutable, and an increase in area can lead to a failed implementation instead of just an increased cost. While the RISC-V committee attempted to address this by modifying the specification for embedded systems (5 versions entitled *Zve**), even the smallest 32-bit version, containing no 64-bit or floating-point instructions, contains close to the 400 instructions in the original specification, many of which are not necessary for a general-purpose vector accelerator.

Clearly not all applications will require all 400 instructions, so there must be a way to reduce this and save area. However, selecting a static subset of instructions of the Zve* extensions could still lead to numerous instructions going unused, wasting area in embedded systems where area is critical. Creating a layered implementation would allow for much greater flexibility for designers that do not require all types of instructions for their applications and wish to save space on their chip (be it FPGA or ASIC).

Categorizing these layers as primary and optional further increases the number of possible configurations, making this type of layered implementation highly customizable.

This thesis proposes a reduced, layered version of the RISC-V Vector instruction set, tailored for FPGA-based embedded applications. The major contributions of this thesis are:

- A reduced, layered version of the RISC-V instruction set, called *RVV-Lite* v0.5. The instructions are divided into two categories: *primary*, and *optional*, with justification provided for each, as well as for the excluded or *extra* instructions.
- Unique notation to describe the above instruction set using minimal space, condensing the specification
- A complete implementation of the primary layers of this reduced set, along with one optional extentsion. We attach this vector engine, named Saturn-V, to its scalar host with a general-purpose CFU port, and show that it is smaller than comparable full Zve* implementations, without a significant reduction in performance.

Chapter 2 discusses RVV and the Zve* versions in more detail, introduces the CFU, and describes related vector processors and other works. Chapter 3 introduces the subsetting of instructions, as well as the notation that combines related instructions into a more condensed format. Chapter 4 describes Saturn-V (the FPGA-based implementation of RVV-Lite) with justification for various architectural decisions. Chapters 5 and 6 describe and justify the instructions that are in the reduced instruction set. Chapter 7 lists the last layer of instructions that are not in the primary layers or optional extensions, with explanations for why each is left for this final layer. Chapter 8 details the area and runtime results for the given implementation, and compares these numbers to other RVV processors. Finally, Chapter 9 summarizes the findings, and proposes future work.

Chapter 2

Background

This work is based on the RISC-V Vector extension, and incorporates ideas from other vector processing architectures like MXP [16] [15] and VESPA [18]. It also makes use of the Custom Functional Unit (CFU) specification [7], which defines a universal accelerator port.

2.1 RISC-V Vector Extension

The RISC-V Vector extension is composed of over 400 instructions, supports 4 integer (8-/16-/32-/64-bit) and 2 floating-point (single-/double-precision) data types, and up to 3 different operand combinations per instruction (Vector-Vector, Vector-Scalar, Vector-Immediate). To accelerate fractional computation in systems without floating-point support, RVV also supports fixed-point operations. It supports four rounding modes for these: round-to-nearest-even, round-to-nearest-odd, round-up, and round-down. Most of the instructions included are standard arithmetic and logic operations (e.g. add, subtract, multiply, AND/OR/XOR, etc), while others have more application-specific use cases (e.g. multiply-accumulate).

The specification requires that all RVV implementations have 32 vector registers, named $v0$ to $v31$, where each register stores VLEN bits, where VLEN is a power of 2 such that $128 \leq VLEN \leq 65536$. Each vector register can be split evenly into 8-, 16-, 32-, or 64-bit elements. To enable longer vectors for large data types, RVV includes register groupings of 1, 2, 4, or 8 registers. When used, register groups are treated as a singular continuous vector.

Mask vectors reuse these registers, but store one bit per element, packed contiguously regardless of element size or the size of the register grouping.

```
void rvv_rgb2luma(uint8_t *luma, uint32_t *rgb, const uint32_t image_width,
const uint32_t image_height)
{
    size_t vl;
    vuint32m4_t px;
    vuint16m2_t tmp, out_tmp, red, green, blue;
    vuint8m1_t  out;

    vl = vsetvl_e32m4 (image_width);

    for (int i = 0; i < image_height; i++){
        px      = vle32_v_u32m4(&rgb[i*image_width], vl);

        // narrowing shift
        red     = vnsrl_wx_u16m2(px, 16, vl);
        green   = vnsrl_wx_u16m2(px, 8, vl);
        blue    = vnsrl_wx_u16m2(px, 0, vl);

        red     = vand_vx_u16m2(red, 255, vl);
        out_tmp = vmul_vx_u16m2(red, 66, vl);

        green   = vand_vx_u16m2(green, 255, vl);
        tmp     = vmul_vx_u16m2(green, 129, vl);
        out_tmp = vadd_vv_u16m2(out_tmp, tmp, vl);

        blue    = vand_vx_u16m2(blue, 255, vl);
        tmp     = vmul_vx_u16m2(blue, 25, vl);
        out_tmp = vadd_vv_u16m2(out_tmp, tmp, vl);

        out_tmp = vadd_vx_u16m2(out_tmp, 128, vl);
        out     = vnsrl_wx_u8m1(out_tmp, 8, vl);

        vse8_v_u8m1 (&luma[i*image_width], out, vl);
    }
}
```

Figure 2.1: Mixed-width RGB2LUMA implementation, using RVV intrinsics

All arithmetic and logic operations have a masked option, and will read out the register $v0$ as the active mask register.

Unique to RISC-V are the configuration commands `vsetvl` and `vset{i}vli`, which set the vector length (VL) according to the requested application vector length (AVL), as well as the element width and register grouping size. Most vector processing architectures allow for loads and stores of different element sizes, but sign-extend elements in the register to operate at the same element width. RVV, on the other hand, allows processing of different element sizes [8] to provide increased parallelism, but this places more demand on the ALU design to be able to operate at different widths.

An example of an application using these mixed-width operations is the simple RGB to LUMA conversion in Figure 2.1. Here, the RGB input is 32-bit, and the expected luminance (LUMA) output is 8-bit. Because each of the components of the RGB input are 8-bit, we do not need full 32-bit precision for intermediate operations, and can instead use 16-bit values without losing precision. Thus, we use a narrowing shift instruction to extract each of the RGB values. Without floating-point support, we calculate the luminance value using fixed-point coefficients with 8 bits of precision, and then use a narrowing shift right by 8 before storing the output.

In other architectures, this would be accomplished by loading the bytes and coefficients into the vector registers as sign-extended words, and operating on them as so. This requires 2-4x the number of cycles to accomplish.

While this example contains only commonly used instructions, the full RVV specification contains many other instructions. Some of these, like the dedicated multiply-accumulate instruction, could improve the runtime of this example. However, when fully implemented, this extension would be too large for embedded applications with limited available area. Hence, modifications must be made to the specification to allow RVV-compliant embedded engines to be built.

2.1.1 Zve* Extension

The Zve* extensions [8] are RVV's approach to area-sensitive applications. They differ from the full specification in that some versions remove 64-bit support, and they omit a small number of hardware-intensive instructions.

There are 5 different versions in the Zve* extension which correspond to varying levels of support. The names are formatted as `Zve{32/64}{x/f/d}`, where the number corresponds to the maximum supported element width, and the end letter corresponds to the largest floating point data type supported (x = integer only, f = single-precision float, d = double-precision float).

Apart from differences in data type support, all Zve* extensions share some key omissions from the standard RVV extension. These omissions remove 64-bit support requirements for certain hardware-intensive operations. The removed instructions are the multiply-high and fixed-point multiply operations, which require a multiplier with a full 128-bit output. A multiplier of this size would consume a significant amount of extra hardware [8]. The extensions support all other vector integer and fixed-point instructions, as well as all load, store, and configuration instructions. Support is also provided for all reduction instructions, as well as all mask and permutation instructions.

Apart from the omission of a few instructions and certain datatypes, Zve* does not allow further subsetting of RVV. As a result, there are many instructions included in these versions of the vector extension that are likely to go unused, leading to wasted area. Thus, for embedded applications we must explore other ways to reduce the size of the instruction set and better tailor the implemented subset of instructions to suit a given system.

2.2 The CFU

The custom functional unit `(CFU)` specification [7] defines an interface designed specifically for connecting processors to application-specific hardware accelerators. It redirects instructions to the appropriate accelerator using

custom instruction encodings, and can help with differentiating between accelerators if there is any overlap in encodings. The CFU interface is responsible for accepting custom instructions from the host and keeping track of the status of any accepted instructions (i.e. forwarded to accelerator, stalled, etc). It is also responsible for forwarding any responses from the accelerator back to the host.

Based on the CFU interface specification, CFU-Playground [13], an FPGA-based framework for a RISC-V development environment specifically built to support experimentation around the VexRiscv RISC-V processor, particularly with regards to machine learning with Tensorflow. This environment is generated using LiteX [10], and allows users to connect a scalar host and accelerator via the CFU port plugin. The default board for this framework is the Arty FPGA, however it is easily adapted to other Xilinx boards, like the ZCU104 used for this thesis.

2.3 Related Work

Over the years there have been many vector and single-instruction-multiple-data (SIMD) processors and instruction sets. Many instructions sets and their processors that are currently in use have some sort of SIMD or vector processing capability. It follows that RISC-V would release their own version, and that CPU developers would eventually follow suit with implementations.

2.3.1 SIMD vs Vector Processing

SIMD processing is used to allow CPUs to perform the same operation on multiple elements of data using a single instruction. This is done by extending registers to hold multiple elements (either by creating SIMD registers or allowing registers to hold smaller elements) and adding more ALUs to operate on the number of required elements. This allows CPUs to complete a larger number of computations per clock cycle, reducing the overall latency of the complete set of operations. Most major chip makers have some

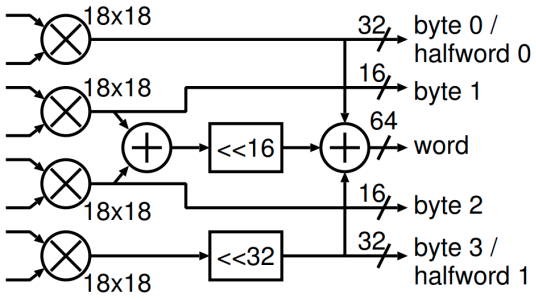

Figure 2.2: Fracturable 32-bit multiplier used in VEGAS. Source: [3]

support for SIMD instructions like MMX, SSE, AVX, or Neon [6].

Vector processing uses this SIMD approach as a basic building block to process much longer streams of data. It defines new instructions to tell the CPU what element size and how many elements to process for each instruction. Since vectors may be longer than the available SIMD width, specifying the length allows the CPU to know how many elements it will need to process, and accordingly how many cycles it will need to process all input data. Because the vector length and SIMD width are no longer directly dependent on each other, it is possible to run the same vector code on processors with different SIMD widths.

2.3.2 MXP

MXP is a portable vector accelerator that supports multiple hosts (ARM, RISC-V, and others). It differs from RVV in that it has a scratchpad memory (which can be divided into any number of vectors of any length) instead of vector registers and register groupings. It also takes a relatively unique approach to processing mixed data widths, where it is able to use the same pipeline to process a vector of 8-/16-/32-bit elements using the same hardware [16]. This approach was initially inspired by the sub-word ALU capabilities in VEGAS [3], illustrated by the multiplier design in Figure 2.2.

For most operations, creating functional units that can process multiple element widths is straightforward, but for multiplication this requires im-

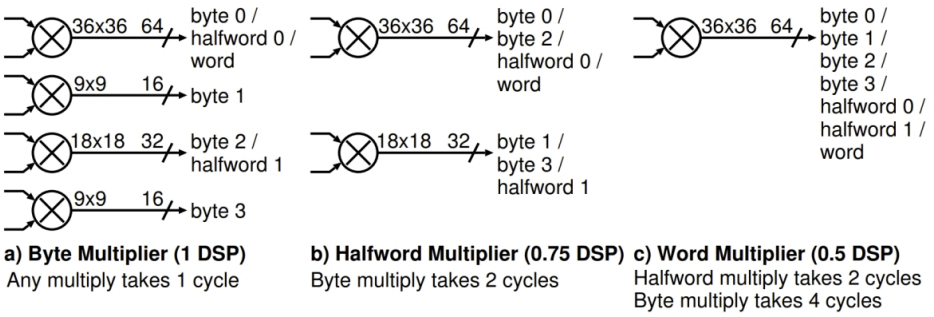

Figure 2.3: Fracturable 32-bit multiplier used in MXP. Source: [16]

plementation of a fracturable multiplier, in which four multipliers are used to compute either four 8x8-bit, two 16x16-bit, or one 32x32-bit product. In MXP this is done by breaking the operation down across two 9x9, one 18x18, and one 36x36 bit multiplier, as shown in Figure 2.3. Here, MXP provides three different implementations to adapt to the number of available digital signal processing (DSP) blocks. DSP numbers reported are for an Altera board, and not a Xilinx board. Implementation (a) takes the most space, but produces all results in the same cycle, (b) saves some space, but requires an extra clock cycle to produce all four byte multiplications, and (c), while even smaller than (b), takes two cycles for two halfword results, and four clock cycles for four byte multiplications.

For the implementation in this thesis, we adopt a fracturable multiplier approach more similar to VEGAS, and use eleven 18x18-bit multipliers, each taking one Xilinx DSP, to produce a 64-bit multiply result. This will be further explained in Chapter 4.1.2.

2.3.3 Hwacha

Hwacha [11] is a RISC-V-based vector instruction set that predates the official RVV instruction set. It has been taped out numerous times in ASIC technologies spanning from 16nm to 28nm. These implementations include an eight-core RISC-V processor capable of supporting up to 16 single-precision floating point operations per cycle [14] and a dual-core RISC-V

processor with one single-lane Hwacha vector accelerator per core [17]. The Hwacha vector co-processor depends on the RISC-V Rocket scalar core.

Because Hwacha pre-dates the actual RISC-V Vector specification, and because its implementations are all ASIC-based, results reported for these implementations are not directly relevant to the work in this thesis. However, it is important to recognize that Hwacha inspired the development of a standardized RISC-V vector specification.

2.3.4 ASIC-based RVV Implementations

Ara [2] is the only ASIC-based RVV implementation we are aware of, and it was based on version 0.5 of the specification, in 2019. The authors claim that Ara's micro-architecture is scalable and achieves 97% FPU utilization at 1GHz. It is tightly coupled with the 64-bit Ariane processor, which is based on RV64GC, a complete 64-bit version of the RISC-V instruction set architecture (ISA). Compared to Hwacha, Ara achieves higher normalized performance.

While there are also industry-announced RISC-V processors with 512-bit SIMD vector support from Si-Five (VIS7) [9] and Andes (NX27V) [4], not enough information is publicly available to determine the implementation, performance, or areas of their respective vector engines.

2.3.5 FPGA-based RVV Implementations

Arrow

Arrow is the oldest published FPGA-based RVV implementation, and was constructed based on version 0.9 of the specification [1]. Judging from its very small footprint of just under 500 lookup tables (LUTs), it likely is not a complete implementation of RVV0.9, and instead implements a small subset of the specification. Even so, Arrow achieves up to 78x speedup over a scalar RISC-V processor for a 4096-element vector addition or multiplication, and up to 51.2x speedup for a 4096-element vector max reduction. Additionally, it achieves 24.1x speedup for a 64x64 general matrix multiplication (GEMM), 50.4x speedup for a 512x512 GEMM, and 58.6x speedup

for a 4096x4096 GEMM. Arrow's register file is dual-banked, and capable of reading out up to two ELEN-bit words at once. This means Arrow can potentially support up to four ALU lanes. All results reported in their paper are for a single-issue, dual-lane implementation, where each ALU is able to independently process a vector instruction.

Vicuna

Vicuna is the only other open-source FPGA-based RVV implementation that is based on version 1.0 of the specification. It conforms to the Zve32x specification, meaning that it contains no floating-point support, nor does it support 64-bit elements. It also is intended for systems with a maximum VLEN of 2048 bits. It is currently only compatible with two scalar cores (Ibex and CV32E40X). It contains different "lanes" for different operations (ALU, memory, etc), and allows users to customize the widths of some of these pipelines.

The original release of Vicuna [12] was based on version 0.10 of the RVV specification. It was released in July 2021, and is significantly larger and somewhat slower than the current version of Vicuna at the time of writing this thesis. In fact, the current version is just over half the size of the original release. No updated area and cycle-count benchmark results have been published since its initial release. Accordingly, all area results reported for the latest version were measured using a custom compilation for the Xilinx ZCU104 board.

2.3.6 Subsetting Vector Instruction Sets

While we are not aware of any other third-party subsetting of the RISC-V vector instruction set, subsetting a vector instruction set was previously done with VESPA [19] at the University of Toronto in 2008. VESPA implements pseudo-subsetting by allowing each instruction in its vector coprocessor to be enabled or disabled independently. Disabling all instructions for a specific functional unit leads to the elimination of that unit in the final coprocessor.

They ran experiments on a 16-lane vector processor with a full memory

crossbar, 16KB cache and 64B cache lines. In doing so, they found that reducing the supported instruction set could reduce area usage by up to 58% for the IP_CHECKSUM benchmark, as the absence of multiplication instructions allowed for elimination of the entire functional unit. Similarly, the CONVEN and RGBCMYK benchmarks saw a 30% area reduction by eliminating multiply support. Another benchmark that saw notable improvement was the AUTCOR benchmark, which uses all functional ALU modules but doesn't require support for store instructions.

Another space-saving strategy explored was combining instruction elimination with elimination of unused vector lane width space. The combination of these two area reductions allowed the author to achieve clock speeds between 3 and 13% faster than the full vector coprocessor, with average area savings of 47% across all examined benchmarks.

This approach is very fine-grained, as instructions are not explicitly divided into groups of related functions, and are instead individually controlled. Their approach is also automated, with their custom tool analyzing a given application binary to determine the subset of instructions that are unused, and disabling support for them by default. While VESPA does provide a practical means for subsetting a vector instruction set, the author does not propose a standardized subset of instructions, nor does he propose an explicit layering of different functional groups. Subsetting and layering helps simplify software management.

Chapter 3

RVV-Lite Overview

RVV-Lite is a strict subset of the Zve* extensions of RVV. This means that all implemented versions of RVV-Lite, regardless of the selected layers and extensions, are forward compatible with a full Zve* implementation. By extension, all RVV-Lite implementations are also forward compatible with full RVV implementations. This means that any program that can run on an RVV-Lite implementation will be able to run on a full RVV implementation with no modifications.

Our implementation of RVV-Lite is named Saturn-V, and is fully described in Chapter 4. Our instruction layering scheme is constructed in a way that aims to minimize the area usage when synthesizing Saturn-V. Our specific instruction groupings were formed using area results from synthesizing for the ZCU104 board.

3.1 Instruction Subsetting

RVV-Lite is designed in a layered fashion to allow for implementations with varying levels of functionality. For implementations requiring fewer instructions, having fewer layers implies a smaller vector processor, leaving more area for other functional units on the chip.

The layers are categorized into 3 different sub-types: primary layers, optional extensions, and extra extensions. Within the Primary sub-type (A.X), some layers reuse prior layers in order to function correctly – for example the widening addition and subtraction layer (A.2) will require the core layer (A.1), as it has the addition and subtraction unit, and the slide-by-n layer (A.5) requires the slide-by-1 module from the core layer (A.1g). Each layer is grouped in such a way that most or all instructions will share a

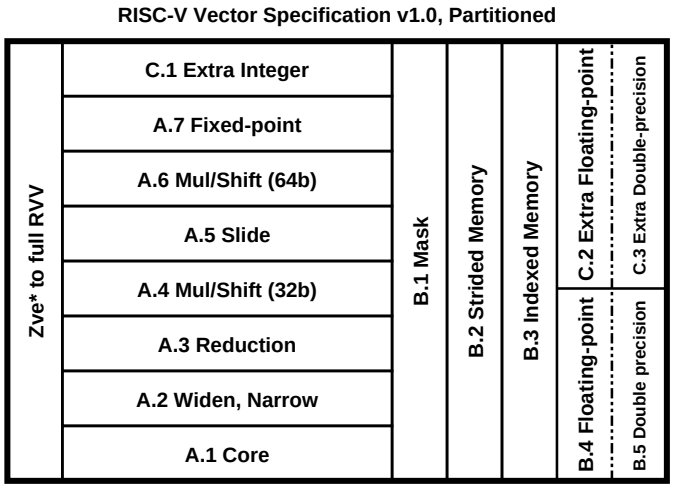

Figure 3.1: RVV-Lite layer structure

functional unit. Optional extensions (B.X) are separated in such a way that no instruction will have any dependencies outside of the core layer (A.1) of RVV-Lite, described in Chapter 5.1. Extra instructions (C.X) are intended to be excluded from most RVV-Lite implementations, as either their usage is very application-specific or their functionality can be easily replicated by a combination of other instructions, and they are therefore not worth the additional hardware cost for most implementations.

Chapters 5 through 7 detail the specific subset groups and justifies at a high level why layers are ordered as they are. In chapter 8 this specific choice of subsetting is defended using real area and performance results from a real RVV-Lite implementation described in Chapter 4. Figure 3.1 visualizes the intended layout of these different layers, to better explain what is meant by primary layers, and optional, and extra extensions. Layers organized vertically (A.1 to A.7 and C.1 to C.3) require all below layers. Layers organized horizontally can be implemented independently.

Subsequent chapters will contain a list of all primary layers, and optional and extra extensions, and will justify why each instruction is placed in each section. The notation used for instructions is in the format *instruction.type*. The *type* can be any combination of V, X, I, F, where $V = vv$ (two vector

operands), $X = vx$ (a vector-scalar operation), $X = vi$ (a vector-immediate operation), and $F = vf$ (a vector-float operation).

The following list summarizes other notation used:

- YY denotes up to 3 operand types: vs1/rs1/imm
- ZZ denotes up to 3 operand types: vs1/rs1/uimm
- rs1 denotes a register number from the X or F register set, i.e. X[rs1] or F[rs1]
- K denotes register group size: 1, 2, 4, 8
- EW/IW denotes element/index width: 8, 16, 32, 64
- KEW denotes register group/EW sizes: 1re8, 2re16, 4re32, 8re64
- L denotes index of last element, i.e. L = VL-1
- S/U denotes signed/unsigned
- LOP (logic-op): and, or, xor
- FOP (float-op): add, sub, mul, div, min, max
- MOP (mask-op): and, nand, andnot, xor, or, nor, ornot, xnor
- ROP (reduction-op): sum, and, or, xor, maxu, max, minu, min
- MCmp (mask-compare): seq, sne, sle, sleu
- ADD/SUB/MUL/MIN/MAX: add/sub/mul/min/max, addu/subu/mulu/minu/maxu
- MULH: mulh, mulhu, mulhsu
- ADDSUB{U}/MINMAX{U}: add{u}/min{u}, sub{u}/max{u}
- EQ/GTE/LTE: eq/gt/lt, ne/ge/le
- SLT/SGT: slt/sgt, sltu/sgtu

Beyond this list, any operation denoted using a $x\{y\}$ denotes that y is an option that can be added to x. In cases where only one option is listed, the two options are with or without y (i.e. $x\{y\}$ represents both x and xy). In cases where more than one option is listed, those are the only options (i.e. $x\{y/z\}$ means xy and xz). In cases where there are multiple options listed, operation chaining applies. This means that $x\{y/z\}\{a\}$ represents xy, xya, xz and xza. In cases where brackets are used on either side of an operation (e.g. $x\{y\} = z\{y\}$), this represents only $x = z$ and $xy = zy$.

3.2 Non-Instruction-Specific Restrictions

Beyond dividing the instructions into layers, there are other fundamental restrictions imposed in RVV-Lite that simplify implementation. Firstly, RVV-Lite does not include any ternary instructions to ensure simplicity in the design of the vector register file. FPGAs are generally able to support designs with dual-port BRAM, making a register file with two read ports and one write port simple to implement. By adding a third read port the design becomes more complex, as it introduces the need for additional data redundancy.

3.2.1 Configuration-Setting Instructions

In addition to configuring the vector length (VL), software must assign a selected element width (SEW) of 8/16/32/64 bits, and vector register group size (LMUL) of 1/2/4/8 registers. All three of these parameters (VL, SEW, LMUL) are set with a single `setvl` instruction. Assigning LMUL > 1 merges adjacent vector registers, thereby creating register groups, i.e. a register file with fewer but longer vector registers. Note that register groups must be aligned.

RVV allows these parameters to be set independently, and even allows fractional LMUL values. The current values of SEW and LMUL alter the behaviour of the register file, adding complexity and multiplexing logic particularly with instructions that must widen or narrow data.

Register groups were initially created to allow for longer vector registers to better amortize fixed performance overheads. In practise, FPGAs have large RAMs which allow sufficiently long vector registers, and the performance benefits of increasing register length show diminishing returns. Register groups are, however, required in order to implement widening instructions. Conversely, fractional LMUL values are needed to properly implement narrowing instructions when the length of the resulting vector is smaller than a full register. For example, using a narrowing instruction on a vector with SEW=16 LMUL=1 to an 8-bit result requires support for LMUL=1/2. Fractional LMUL values also allow for widening instructions where the destination is an individual register, rather than a register group.

To simplify things in RVV-Lite, the SEW/LMUL ratio must stay constant. This way, the maximum vector length stays constant regardless of the SEW value, as LMUL will change proportionally to SEW. This greatly simplifies register file design, particularly for widening and narrowing instructions, but it also restricts software to use a decreasing fixed number of registers as SEW increases. Though restrictive, requiring a constant SEW/LMUL modes helps simplify software; it ensures that VLMAX, the longest vector length (in elements) supported by the implementation, is constant across the SEW settings. The maximum number of elements per vector, VLMAX, is equal to $\frac{VLEN*LMUL}{SEW}$. As a result, keeping the SEW/LMUL ratio constant ensures that VLMAX doesn't change when the element size changes. We set this ratio to 8, which allows the best tradeoff between number of elements and available data widths: any lower, and elements are lost for larger SEW, any larger and we reduce the number of available elements for all SEW.

3.2.2 Register File Data Layout

With the constraint that SEW/LMUL stays constant at all times, there are a limited number of ways in which data can be arranged within in the register file. These arrangements, which are related to register file groupings, are shown in Figure 3.2. This is best explained using SEW=8 and SEW=16:

- When bytes are used, the group size is 1 and all bytes are stored consecutively within a single vector register.
- When halfwords are used, the group size is 2 and the first half of the vector elements are stored in an even-numbered register (V_n), while the second half (higher index values) are stored in an odd-numbered register (V_{n+1}).

Register groups must always be aligned to the size of the group, and the group name is determined by the lowest register number within the group. There is a conscious simplification/trade-off here in RVV-Lite, which is not present in RVV, where there are fewer vector registers available when larger element sizes are used. However, as the constraint does not introduce any new element widths or register group sizes it is still forward-compatible with RVV.

```
VLEN=128b, SEW=8b, LMUL=1

Byte     F E D C B A 9 8 7 6 5 4 3 2 1 0
Vn       F E D C B A 9 8 7 6 5 4 3 2 1 0

VLEN=128b, SEW=16b, LMUL=2

Byte     F E D C B A 9 8 7 6 5 4 3 2 1 0
V2*n       7   6   5   4   3   2   1   0
V2*n+1     F   E   D   C   B   A   9   8

VLEN=128b, SEW=32b, LMUL=4

Byte     F E D C B A 9 8 7 6 5 4 3 2 1 0
V4*n           3       2       1       0
V4*n+1         7       6       5       4
V4*n+2         B       A       9       8
V4*n+3         F       E       D       C

VLEN=128b, SEW=64b, LMUL=8

Byte     F E D C B A 9 8 7 6 5 4 3 2 1 0
V8*n                   1               0
V8*n+1                 3               2
V8*n+2                 5               4
V8*n+3                 7               6
V8*n+4                 9               8
V8*n+5                 B               A
V8*n+6                 D               C
V8*n+7                 F               E
```

Figure 3.2: Data layout and register groupings for SEW/LMUL=8

Restricting SEW/LMUL settings in RVV-Lite forces register grouping and thereby reduces the number of registers available for data sizes larger than a byte; there are only 16/8/4 registers for 16b/32b/64b data for SEW/LMUL=8. We hypothesize that this will simplify the register file design by reducing multiplexing for different SEW/LMUL settings, however verification of this is left for future work.

For readability of assembly code, we suggest new `vtype` descriptors of `d8`, `d16`, `d32` and `d64` to represent `e8m1`, `e16m2`, `e32m4`, and `e64m8`, respectively. Hence, the only valid encodings of SEW and LMUL in `vtype` are 000000, 001001, 010010, 011011. The same 3 bit pattern is repeated, which allows

for some minor logic optimization.

3.2.3 Element Masking

RVV-Lite also leaves all inactive elements unchanged when writing back to the vector register file. This is referred to as "undisturbed" masking in the full RVV specification, while full RVV includes "agnostic" masking (allowing overwriting) as well. RVV-Lite uses only undisturbed masking to ensure that developers can rely on elements retaining their value when they are not currently being updated, but may need to be accessed later.

3.3 Layer Area Breakdown

Table 3.1 shows the results of implementing each RVV-Lite layer in Saturn-V using Vivado 2020.2. This implementation uses a DATA_WIDTH of 64 bits, as it is the minimum engine width that can naturally support 64-bit element operations. The vector register file size is set to the RISC-V maximum of 64kB (32 x 16,384-bit vector registers) to illustrate the maximum demand on BRAM resources; since FPGAs generally have many large on-chip Block and Ultra RAMs, we can easily support a register file of this size.

Area results in this table reflect the current state of RVV-Lite v0.5. These and prior results were used to motivate the current instruction layering described in subsequent chapters.

Though our target board, the ZCU104, has an onboard FPGA with over 500 000 LUTs, the layers were designed in such a way that even on smaller FPGAs like the Artix-7 35T [5], which has roughly 20 000 LUTs, additional instruction groupings do not consume more than 5% of the available LUT resources. The target size for an instruction grouping is therefore half of this (about 500 LUTs). This is done in an effort to reserve space for other onboard accelerators and reduce overall power consumption.

Table 3.1: Implementation breakdown for Saturn-V (64b SIMD width, 64kB register file / VLEN=16,384)

Configuration	**LUT**	**BRAM**	**URAM**	**DSP**	**FMAX (MHz)**
A.1 Core Instructions, decomposed:					
a) Load/Store/Cfg	1,220	20	2	0	177.2
b) And/Or/Xor/Add/Sub/ID	2,584	36	2	0	181.7
c) Min/Max.	2,630	36	2	0	183.3
d) Basic Mask	3,155	37	2	0	178.6
e) Vector Move	3,237	37	2	0	175.4
f) Whole Reg Mov/Ld/St	3,251	37	2	0	185.4
g) Vector Slide1	3,835	37	2	0	160.1
A.2 Widening Add/Sub	4,070	37	2	0	177.1
A.3 Reduction	4,523	37	2	0	162.1
A.4 Mul/Shift:					
a) Mul/ShiftL (8/16/32b)	4,833	37	2	8	172.1
b) MulH/ShiftR (8/16b)	4,857	37	2	8	183.8
c) MulH/ShiftR (32b)	5,165	37	2	8	166.5
d) Widening Mul, Narrowing Shift	5,647	37	2	8	167.3
A.5 Slide-by-N	5,932	37	2	8	187.3
A.6 Mul/Shift (64b)	6,273	37	2	11	161.0
A.7 FXP	6,523	37	2	11	169.4
B.1 Mask (+A.7)	7,164	37	2	11	172.0

Chapter 4

Implementation Design

To ensure the subsetting of instructions is based on sound rationale for saving area, we implement Saturn-V, an RVV-Lite v0.5 compliant processor that can attach to any RISC-V processor that implements a standard CFU port. The name Saturn-V comes after the layered, multi-stage rocket used by NASA to put humans on the moon. Saturn-V supports all base layers, as well as the additional mask instructions layer. Mask instructions are the only implemented optional extension, which exists to guide the division of masked instructions between the core instruction layer (A.1) and optional extension (B.1).

Saturn-V attaches to the VexRiscv processor using a CFU port [7], which redirects all vector instructions from the scalar host. The CFU uses a valid/ready protocol to signal the start and end of instructions. Use of the general-purpose CFU port means that Saturn-V is not tightly coupled to any particular RISC-V implementation, which differs from other RVV implementations.

4.1 Processor Design

Saturn-V is structured as in Figure 4.1. In this figure, there are three address generation units (AGUs) connected to the ALU and generating source addresses a and b (Aa and Ab) and destination address c (Ac) based on the current SEW setting (VSEW). There are two additional AGUs in the direct memory access (DMA) engine for the vector load (VLD) and vector store (VST) operations. The vector register file (VRegFile) is able to support two vector reads and one vector write per cycle. The data associated with these operations are denoted Da, Db, and Dc respectfully. The bytes that are

written to the vector register file are determined by the byte enable signal, Be.

It is capable of receiving up to 1 instruction per cycle from the CFU ("CFU Host" in the diagram), if the vector length matches the data width (DATA_WIDTH). For each additional packet of vector data, the latency will increase by 1 cycle. Upon receiving a valid instruction the processor holds it in a register and decodes the source(s), destination, and operation type in order to detect hazards between operations. Based on the current vector length (VL), this stage determines the number of data packets that will be processed for a given instruction: $\lceil \frac{VL*SEW}{DATA\_WIDTH} \rceil$ for ALU and memory operations, $\lceil \frac{VL*SEW/8}{DATA\_WIDTH} \rceil$ for mask operations, and $\lceil \frac{VLEN}{DATA\_WIDTH} \rceil$ for whole register operations. Once the instruction is able to move forward with no hazards, a counter is set to ensure the next instruction is stalled until the current instruction has finished reading or generating data inputs for the memory or ALU pipeline stages that follow.

To enhance the throughput of the processor, independent memory load and ALU instructions can operate in parallel. The bottleneck happens in the decode stage, because only one instruction is able to read data from the register file at once. By separating these operations, the processor is able to send memory read requests and proceed to process other vector instructions until the memory has returned. Since memory response time is not deterministic, data packets returned by a memory read are stored in a FIFO queue until the full vector has been returned, and are then passed to the register file once the processor has finished processing any independent ALU instructions. In the event that an ALU instruction depends on the result of an outstanding load operation, the pipeline will stall until the result has been returned from memory. Once the register file has been written to, the end of the instruction triggers a valid output signal, which tells the scalar host to send the next vector instruction.

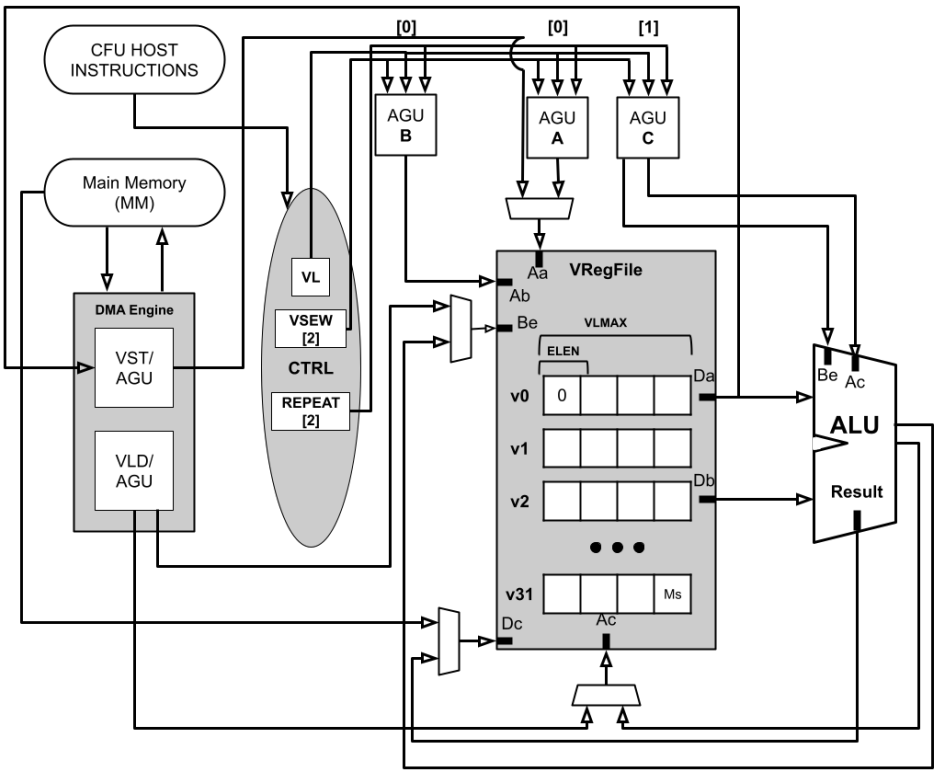

Figure 4.1: Internal Saturn-V micro architecture

4.1.1 Configuration-Setting Instructions

As mentioned in Section 3, we use a constant SEW/LMUL ratio of 8 to ensure the maximum number of available elements per vector (VLMAX) stays constant. To simplify the design, the configuration unit reads only the SEW value to determine which settings should be applied, as the encodings for SEW and LMUL are identical when our chosen ratio of 8 is used. In the event that an incoming instruction attempts to use an unsupported SEW/LMUL, the value of SEW will be used to determine both SEW and LMUL.[1]

4.1.2 ALU Architecture

The vector ALU has one 64-bit lane that can operate on any element size (8/16/32/64 bits). One of the key challenges is designing an ALU that can change the element size on an instruction-by-instruction basis (potentially every cycle). Figure 4.2 illustrates the basic internal structure of the ALU lanes. As each functional unit has a unique enable signal, no two units will produce an output in the same cycle. For this reason, we are able to OR the outputs together instead of implementing a multiplexer.

For and/or/xor, the design is trivial because the same logic is used regardless of element size. However, addition, multiplication, sliding, and widening require special handling described below.

For add/subtract, a single 80-bit adder is used. In case of subtraction, the second input operand is simply translated to twos-complement. Each 8-bit chunk of the input is sign extended by 1, and a 'dummy' bit is inserted between each pair of 9-bit operands to force carry chain behaviour with the following configurations: carry generation (creates two narrow adders, one with carry-in set, by setting both dummy bits), carry kill (creates two narrow adders, one with carry-in clear, by clearing both dummy bits), and carry propagation (wider adder by forcing dummy bits to opposite phase).

[1] When an unsupported setting is used, RVV requires setting an illegal type bit, vill, and throwing an illegal-instruction exception on the next vector instruction. RVV-Lite does not require exceptions, so vill is not required.

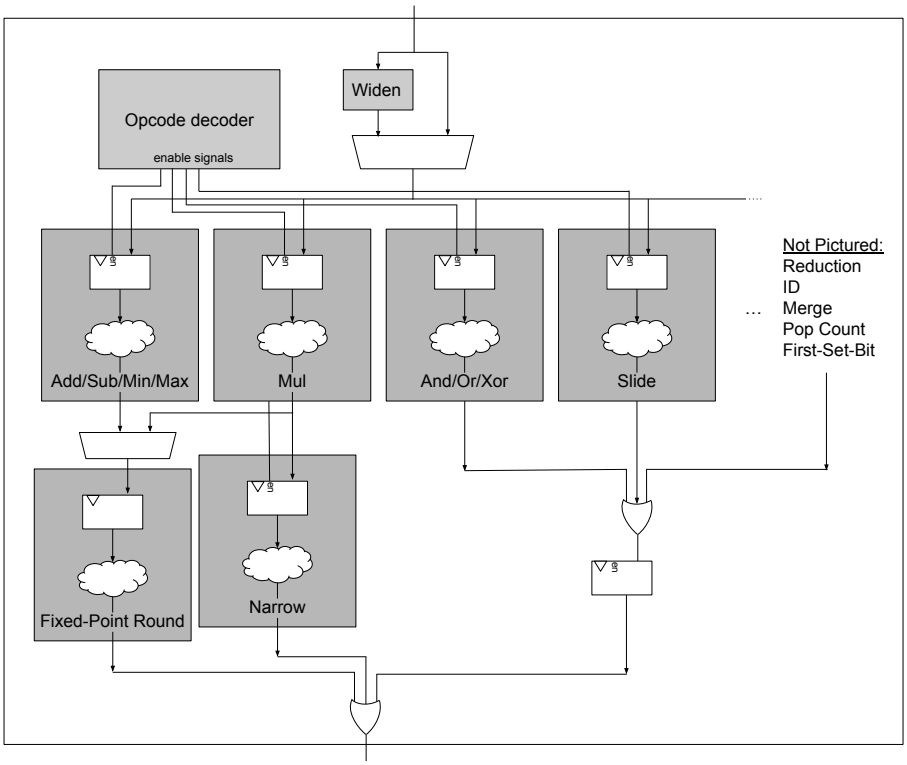

Figure 4.2: ALU structure

This could potentially have been done with a 72-bit adder, but this was not investigated as it would save minimal hardware (likely 8 LUTs; one per output bit).

For the narrow adder configurations, since the forced dummy bit values are known, it is also possible to check the carry-out status of the preceding narrow adder.

This differs from the approach taken by Arrow [1] and Vicuna [12], both of which insert multiplexers to break the carry chain every 8 bits. Breaking carry chains with arbitrary logic can hurt performance and area, since extra LUTs are required (on some FPGA architectures) before and/or after each 8b adder to access the carry chain signal.

For `MINMAX` and `MINMAXU` instructions, the same adder block is used: one operand is subtracted from the other, and the appropriate operand is selected based on the sign of the result.

The slider must shift at byte granularity or higher in increasing powers of 2 to the maximum supported element size in bytes. For example, sliding a vector of half words by 1 is equivalent to sliding a vector of bytes by 2. With 64-bit element support, slide-by-1 and slide-by-N for $N \leq \lceil \frac{DATA\_WIDTH}{8} \rceil$ require similar area. Shifting beyond 64-bits is done with address control and a layer of flip-flops, which increases the area for the slide-by-N instruction as it introduces multiplexing across time slots.

Widening is associated with multiple operations (add, subtract, multiply), so to implement these instructions we widen the upper and then lower halves of the inputs to double the input width. This data is then passed into the appropriate arithmetic unit, along with the resulting `EEW`.

Narrowing is in essence data truncation, so it uses its own datapath, tacked on to the output of the multiplier unit. While it could be merged with the shifter, we wanted to have a very lightweight narrowing implementation that did not directly depend upon any multiplier logic, in order to allow possible support for other types of narrowing instructions.

Reduction operations use a reduction tree, where the first stage reduces $\frac{DATA\_WIDTH}{SEW}$ elements to $\frac{DATA\_WIDTH}{2SEW}$, the second to $\frac{DATA\_WIDTH}{4SEW}$, and so forth. Because this shares no operators with the rest of the RVV instruc-

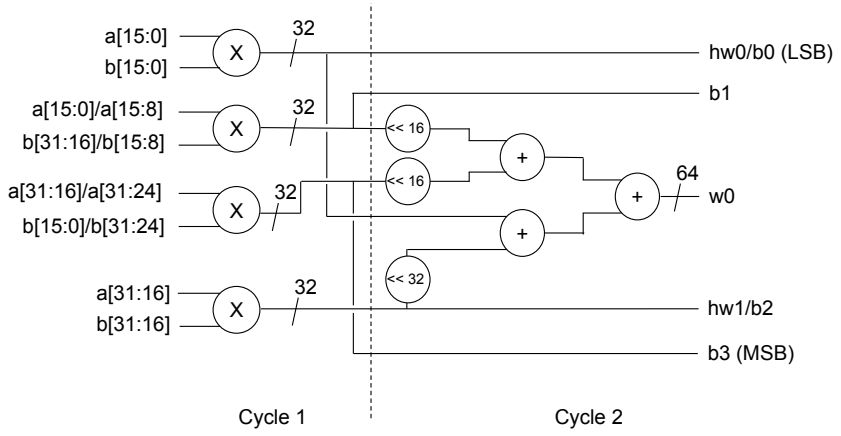

Figure 4.3: Internal structure of Saturn-V's 32-bit multiplier

tion set, reduction is given its own functional unit.

Masked operations can be split into 2 types. The first generates masks, and is essentially an extension to the Min/Max sub-layer. Masks are generated based on equality or inequality comparisons (e.g. less/greater than), and the Min/Max block already performs these comparisons. The second takes mask vector(s) as inputs and manipulates them to generate either a mask output or scalar output (in the case of population count and find-first-set-bit).

64-bit Multiplier Architecture The multiplier is made fracturable as VEGAS [3] and MXP [16] were for operands up to 32 bits, as previously discussed in Section 2.3.2. The implementation of this fracturable multiplier is the same as in VEGAS, as it uses partial products to calculate the 32-bit output, instead of using MXP's approach of fitting differently sized multipliers within the same DSP block. Figure 4.3 illustrates this flow.

In Figure 4.3, the two 32-bit inputs are separated into two 16-bit chunks, and the partial sums for all four combinations of half of input a with half of input b are calculated.

For larger inputs, three 32-bit multipliers are combined to create a 64-bit multiplier, combining the outputs in a similar fashion to the way $w0$ is

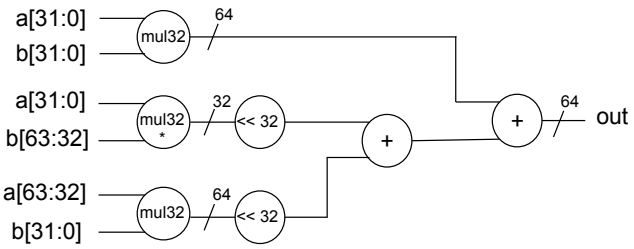

Figure 4.4: Saturn-V's 64-bit multiplier composed of 32-bit multipliers

	64-bit	128-bit (+ vmulh variants)	128-bit (+ vmulh, vsmul)
DSP	11	16	16
LUT	911	1198	1202

Table 4.1: Area of 64-bit multipliers with varying output sizes

generated. This is illustrated in Figure 4.4. In the figure, mul32 refers to a full implementation of Figure 4.3, and mul32* indicates a multiplier that produces only one 32-bit product (requiring 3 16-bit multipliers), and not the other 8-/16-bit outputs.

Because Zve* does not support 64-bit multiplication and shifting operations that produce the upper 64-bits of output, this implementation calculates only the lower 64 bits of the 64x64-bit multiplication. It therefore does not support the 64-bit version of the `vsmul` instruction, or the signed and unsigned `vmulh` variants. A comparison of this implementation and the version supporting one or both of these instructions is shown in Table 4.1.

A 33% increase in area and five additional DSPs to support only 2 extra instructions is a large additional area cost for an embedded processor requiring minimal area usage. This supports Zve*'s exclusion of these instructions.

The shifter re-uses the multiplier hardware; specifically, a shift-right for an element size of SEW uses the upper SEW output bits of each multiplication result.

To support 64-bit right shifts using a multiplier with a 64-bit output, we adopt two different strategies based on the shift amount. To calculate

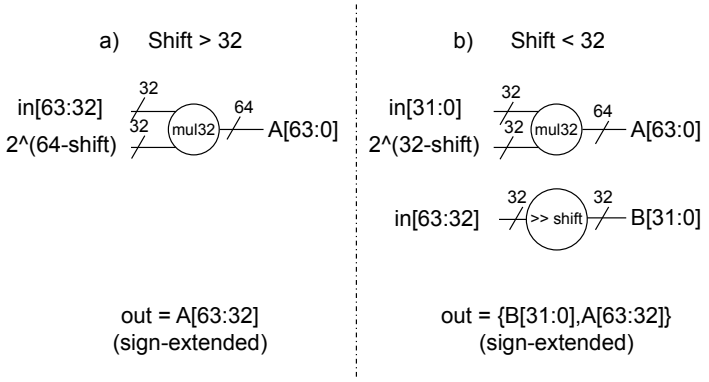

Figure 4.5: 64-bit right shift using 64-bit multiplier hardware

the lower 32 bits of the output, the lower 32 bits of the input along with $2^{64-shift}$ are sent to the multiplier for shifts greater than or equal to 32 and $2^{32-shift}$ for shifts less than 32. For shifts smaller than 32 bits, the top 32 bits of the input are shifted right using new shifter hardware, and the result is concatenated to the upper 32 bits of the multiplier output. Figure 4.5(a) shows the flow for shifts greater than 32, while Figure 4.5(b) illustrates the flow for shifts less than 32.

Shift-left uses a similar approach, but passes in the elements and the shift amount raised to the power of 2, and reads the result from the lower bits of the output corresponding to the SEW. It requires no special hardware to do this, as a left shift operation is equivalent to a multiplication by a power of 2.

In the case of a fixed-point scaling shift right where the shift amount is >= 32, the bottom 32 bits are OR-reduced to produce a single bit and passed into the multiplier to ensure no bits are lost for the fixed point rounding step when the rounding mode is round-to-nearest-even or round-to-nearest-odd.

Slide-By-N Design Instead of leaving all slide-by-N hardware to the ALU itself, this implementation instead leaves at most a slide-by-$[N\%\frac{DATA\_WIDTH}{SEW}]$ to the ALU, to minimize multiplexing and shifting requirements. To handle shifts longer than the engine width it will read or write to the vector register

file at a different starting offset. Sliding by an arbitrary amount involves two stages, a major stage and a minor stage. The major stage is done by adjusting either the read offset of the source or write offset of the result by $N/\frac{DATA\_WIDTH}{SEW}$. The minor stage is done within the ALU, using the existing slide-by-1 hardware (for the different element sizes 8/16/32), for slides up to 7 bytes as required by $N\%\frac{DATA\_WIDTH}{SEW}$.

Chapter 5

Primary Instruction Layers

The following instructions are separated into layers which are ordered by how frequently instructions are expected to be used, as well as their area cost. Higher-numbered layers may require one or more of the preceding layers in order to function (e.g. 64-bit multiplication requires 32-bit multiplication to be implemented). By doing this, implementations are able to share some functional units across instruction groupings, in addition to the existing functional unit sharing within each layer, thus saving additional area. Ordering instructions in this manner ensures that programs that are created for smaller RVV-Lite engines are always forward compatible with both larger engines and full RVV engines.

Although somewhat arbitrary, we aim to keep layers between 200 and 1000 LUTs in size. This can be verified by referencing the area results in Table 3.1.

5.1 Core Layer

The first layer of instructions, called the 'core' instructions, are the smallest grouping of instructions that comprise an RVV-Lite-compliant vector engine. This subset includes basic memory load and store operations, configuration instructions, basic arithmetic and logic operations, a handful of mask set instructions, and some vector permutation instructions. This core layer is intended to be implemented together, but is further subdivided below as it exceeds our 1000 LUT target.

The full core instruction set contains just over 84 instructions, comes in at a total of just over 3800 LUTs, 37 BRAMs, and 2 URAMs, and runs at a maximum frequency of 160MHz. Though it greatly exceeds our target

upper bound of 1000 LUTs, each sub-layer is below the threshold. This layer may be further divided in future if any sub-layers are determined to be unnecessary for a basic general-purpose vector engine. However, for now it remains as a single layer, and rationale for the composition of each sub-layer is given below.

5.1.1 Load, Store, and Configuration

In the following instructions `VL` refers to the *vector length* of a given instruction (set in the vector coprocessor), `AVL` refers to the *application vector length* (the currently requested vector length in software), `EEW` refers to the current *effective element width* of the processor (i.e. the currently stored element width setting), and `EW` refers to the element width of a particular instruction (8/16/32/64 bits).

```
vsetvli  rd,rs1,vtypei      # rd=VL=f(AVL), AVL=rs1, new vtype
vsetivli rd,uimm,vtypei     # rd=VL=f(AVL), AVL=uimm, new vtype
vsetvl   rd, rs1, rs2       # rd=VL=f(AVL), AVL=rs1, vtype=rs2
vleEW.v   vd,(rs1),vm       # vector load, EEW=EW
vseEW.v   vs3,(rs1),vm      # vector store, EEW=EW
```

At the most basic level, any RVV-Lite implementation is required to support basic memory load and store instructions, as well as configuration instructions. Specifically, it supports vector loads and stores corresponding to a configurable vector length (VL). This allows the vector engine to read and write up to VLMAX data elements of 8-, 16-, 32-, or 64-bits to and from the vector register file. The configuration instructions, namely `vsetvl` and `vset{i}vli`, allow users to set the VL and selected element width (SEW) of the engine. By supporting only instructions in this sub-layer, implementations can effectively treat the RVV-Lite engine as an interface between custom vector instructions, the vector register file, memory, and the scalar host processor, without taking any additional space for an otherwise unused vector ALU.

The first sub-layer comes in at just over 1200 LUTs, which includes roughly 800 LUTs dedicated to the memory queue required to connect the vector processor to the AXI-Lite port in CFU-Playground. The structure of this connector is illustrated in Figure 5.1. With burst mode support,

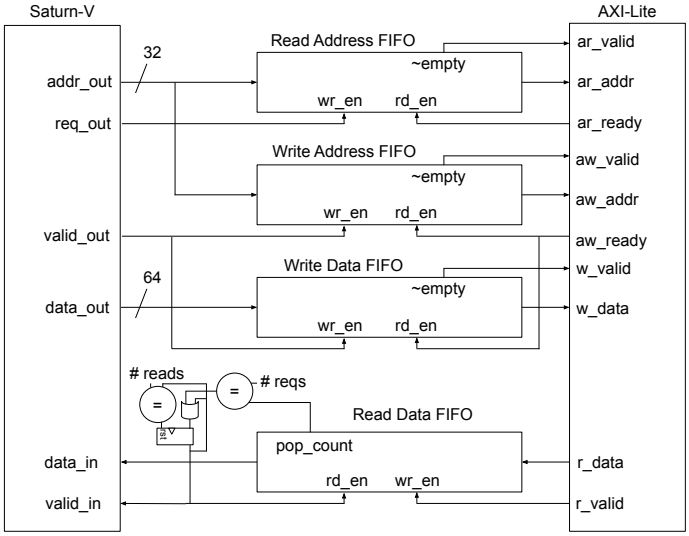

Figure 5.1: Memory queue between Saturn-V and AXI-Lite bus

we expect this sub-layer to shrink in size, as the size of the incoming data queue is reduced. This happens because currently the number of cycles required to send load requests is unpredictable, as control of the bus is yielded after each data packet is returned, and so the connector must wait for a new ready signal from the AXI port for every 64 bits. As a result, a continuous stream of incoming data is not guaranteed for load requests, and as a safety precaution we wait until all outgoing memory load requests have returned values before streaming data to the vector engine. For this reason, the incoming data queue is sized to accommodate a full vector of 64-bit elements. With burst mode support the size required to buffer this incoming data would be significantly less, as there is a guarantee that there will be consecutive data packets returned. This layer also includes only 20 BRAMs and 2 URAMs, of which 16 BRAMs make up the vector register file, and the other 4 BRAMs and 2 URAMs comprise the data and address buffers between the memory port and the AXI-Lite bus.

5.1.2 Logic, Addition, Subtraction, and ID

In this section, `ADDSUB` refers to the signed addition and subtraction operations, and `LOP` refers to the logical operations AND, OR, and XOR.

```
vid.v       vd,vm           # vd[i] = i
vLOP.VXI    vd,vs2,YY,vm    # vd[i] = vs2[i] LOP YY
vADDSUB.VXI vd,vs2,YY,vm    # vd[i] = vs2[i] ADDSUB YY
vrsub.XI    vd,vs2,YY,vm    # vd[i] = YY - vs2[i]
```

Beyond basic memory capabilities, the core layer includes basic logic and arithmetic operations. Specifically, it includes both signed and unsigned addition and subtraction operations, as well as AND, OR, and XOR logic operations. This allows users to vectorize a number of simple programs. Additionally, this sub-layer includes the vector element index `vid` instruction, which sets the value of each element to its corresponding index in the vector. This instruction provides unique functionality to the vector engine, but is not included in the first sub-layer as it requires dedicated hardware, and its functionality can be replaced by a load from memory if needed.

This is the first sub-layer to require the ALU. All functional unit modules are padded to have the same pipeline depth. The modules introduced include a 64-bit adder, which is fracturable to allow it to produce eight byte-wide results, four 16-bit results, two 32-bit results, or one 64-bit result. They also include a logic unit module with three basic logic operations (and/or/xor). Scalar and immediate values are sign-extended and replicated for each element to fill the 64-bits. The final module included implements the vid function, which generates a unique number between 0 and VLMAX-1 (2047 in this case) for each 8-, 16-, 32-, or 64-bit element. Thus, the sub-layer adds approximately 1400 LUTs. It also adds 16 new BRAMs, as the system requires data redundancy in order to provide a second read port on the vector register file.

5.1.3 Min/Max

In this set, `MINMAX{U}` refers to `MINMAX` (signed minimum and maximum) and `MINMAXU` (unsigned minimum and maximum) operations.

```
vMINMAX{U}.VX  vd,vs2,YY,vm    # vd[i] = MINMAX{U}(vs2[i], YY)
```

Signed and unsigned minimum and maximum instructions are included in the core layer as well, as they introduce basic comparison capabilities to the vector ISA. The hardware required reuses the existing subtraction functional unit, but introduces an additional multiplexer. These instructions, along with the basic logic and arithmetic operations, enable users to vectorize simple image filters, and other common vector applications.

These instructions are implemented in the addition module of the ALU. This unit introduces a few multiplexers to select the minimum or maximum element, as well as to select the appropriate functional unit's output in the addition module, and so the layer is only about 50 LUTs in size.

5.1.4 Basic Mask

Here, `MCMP` refers to the *mask comparison* operations `LT` (less than), `LE` (less than or equal), `EQ` (equal), and `NE` (not equal). `MOP` refers to the logical *mask operations* `AND`, `NAND`, `ANDNOT`, `XOR`, `OR`, `NOR`, `ORNOT`, and `XNOR`. Note that `SLT` and `SGT` are additional comparisons for less than and greater than, but they are notationally separate from the `MCMP` group because they have fewer operand types.

```
vmMCMP.VXI  vd,vs2,YY,vm    # vd.m[i] = (vs2[i] MCMP YY)
vmSLT.VX    vd,vs2,YY,vm    # vd.m[i] = (vs2[i] < YY)
vmSGT.XI    vd,vs2,YY,vm    # vd.m[i] = (vs2[i] > YY)
vmMOP.mm    vd,vs2,vs1      # vd.m[i] = MOP(vs2.m[i],vs1.m[i])
```

To enable users to selectively operate on elements, basic masking operations are included. Mask comparison operations (`MCMP`, `SLT`, and `SGT`) instructions allow users to operate selectively on only certain individual elements of a given vector without the need for indexed load/store operations. Inclusion of mask logic instructions (`vm{and/or/xor/not}`) allows for more complex conditional operations. The instructions are functionally equivalent to a vector logic operation with an LMUL of 1, meaning they incur minimal additional hardware cost by reusing the existing logic unit.

This sub-layer creates a shadow mask register used to store the vector mask value in register $v0$. This shadow register is implemented as a BRAM, increasing the overall BRAM count in the system by one. This is done in

order to enable masked instructions without introducing a third read port to the vector register file. Additionally, it adds equality comparison logic (x == 0) to the min/max unit mentioned above, as well as byte-enable signal generation logic to replicate mask values for element widths greater than 1 byte. To compute the mask results, an 8:1 multiplexer is required to select the appropriate result of the available mask comparison logic operations (eq, neq, lt, ltu, lte, lteu, gt, gtu). As well, because mask vectors are stored in vector registers, though they are 1/8 the size, address generation logic to repeat the same register id and offset between 8 and 64 times (based on SEW) is required. These components together result in an area cost of 500 LUTs.

5.1.5 Vector Move

```
vmv.v.VXI vd,YY     # vd[i] = YY, XI modes: integer splat
vmv.x.s   rd,vs2    # x[rd] = vs2[0], scalar copy (vs1=0)
vmv.s.x   vd,rs1    # vd[0] = x[rs1], scalar copy (vs2=0)
```

The most basic vector permutation instruction, the vector move (`vmv`) is included to both easily copy data across registers without requiring redundant memory reads (using `vmv.{v.v/v.i/v.x}`), and to quickly extract or set the first element of a vector (using `vmv.{x.s/s.x}`). Including these operations in the core set is vital as exclusion would require additional memory operations, significantly reducing the overall acceleration capability of any vector processor. Implementation of these instructions also requires little extra hardware, as a move operation can be easily implemented as an AND or OR with identical source operands. These logic paths, if correctly implemented, introduce little in the way of extra hardware or power.

In Saturn-V, the move instruction is implemented as an OR operation where both inputs are the same. The five vector move instructions included in this sub-layer only introduce about 80 LUTs, largely due to the need to replicate the same input for both source operands for different types of move instructions.

5.1.6 Whole Register Move/Load/Store

For this grouping, `EMUL` refers to the *effective register grouping*, `K` refers to the register grouping sizes (1/2/4/8), and `KEW` refers to the register group and element width pairings `1re8`, `2re16`, `4re32`, and `8re64`.

```
vmvKr.v   vd,vs2    # whole-vec. reg. group copy EMUL=K
vlKEW.v v8,(a0)     # whole reg EMUL=K, VLMAX/EW elements, ignores VL
vsKr.v  v8,(a1)     # whole reg EMUL=K, VLMAX bits, ignores VL
```

Whole register load, store, and move instructions are equivalent to their partial register counterparts using a VL equal to VLMAX. As a result, these are easy to include in the core layer, as they require few additional logic paths to implement, but allow users to use entire registers without firing additional configuration instructions, thereby reducing program cycle counts and improving speedup. These instructions also allow the operating system to save and restore state, as sometimes elements beyond the current VL will hold meaningful values.

These instructions are already supported logically because of the first sub-layer and the move sub-layer. They simply require a signal indicating that the processor should assume VL=VLMAX for all instructions. Thus, these instructions add fewer than 20 LUTs.

5.1.7 Slide-by-1

```
vslide1up.vx   vd,vs2,rs1,vm # vd[i+1]=vs2[i], vd[0]=X[rs1]
vslide1down.vx vd,vs2,rs1,vm # vd[i]=vs2[i+1], vd[L]=X[rs1]
```

Slide-by-1 instructions are included in the core layer, as they are required for any program with a sliding window. A prime example of this is a filter or signal processing application.

These instructions require multiplexing logic to move each byte of the input to any other place in the output. This is done because the same unit processes 8-, 16-, 32-, and 64-bit slides, and does this by enabling each in a different stage. This introduces just under 600 LUTs, as each of these slides is performed in a different pipeline stage and requires its own multiplexing

hardware. To support the slide-by-1 instruction, the input signal is one-hot encoded (one bit for each slide size).

Though the area usage of this sub-layer is large relative to some other sub-layers, its function cannot be imitated by any other combination of core layer instructions apart from a load from the original address offset by 1. For this reason, slide-by-1 is left in the core layer to improve power and overall performance.

5.2 Widening Addition and Subtraction

```
vwADDSUB{U}.VX   vd,vs2,YY,vm  # vd[i] = vs2[i] ADDSUB{U} YY    (no 64b)
```

The next subset includes widening addition and subtraction instructions. These instructions allow applications to effectively cast between consecutively sized datatypes and prevent overflow in integer arithmetic operations. These operations are not essential to basic applications, but can reduce memory footprint for vector programs while possibly improving performance and power through lower bandwidth demands. Their implementation can be quite simple, as the ALU only needs to resize the inputs to be the same element width as the output before passing data through the existing add/subtract unit. These instructions require a new widening unit in the ALU, as well as new logic in the AGU to generate the same address and offset for two cycles; the first cycle is used to widen the lower half of the data, while the second widens the upper half.

This results in an additional 250 LUTs. Because these instructions require additional hardware without introducing unique operations, these are not considered core instructions. However their memory savings are enough to motivate placing these in the second layer of RVV-Lite, and their overall footprint is well below 500 LUTs, allowing the layer to not be further subdivided.

5.3 Reduction

The reduction operations in this section are referred to using `ROP`. These instructions include the `SUM`, `MINMAX`, and `LOP` operations.

```
vredROP.vs vd,vs2,vs1,vm    # vd[0]=ROP(vs1[0],vs2[*])
```

Because of the need for cross-lane communication, reductions are not easily replaced by any other vector operations. The reduction operations included in this layer of RVV-Lite are the sum, and/or/xor, and (un)signed minimum and maximum operations. Though each of these operations exists in the core layer, none of these instructions include any cross-lane communication between elements, meaning reduction operations require significant additional hardware in order to be fully implemented for all element widths. Without a vredsum instruction, for example, one does a `vadd` with half of the elements in one operand and half in the other, reduces VL to half its size, and repeats the operation recursively until a single element is left. This mimics a reduction tree with a depth of $log_2(N)$ for N elements. Including dedicated reduction instructions allows the opportunity to minimize cycle count in the vector ALU, if the operation is pipelined correctly. This reduction in runtime motivates the inclusion of such instructions early in the RVV-Lite specification, though the additional hardware requirements prevent it from being considered as a core instruction. The reduction instructions require dedicated reduction trees, which do not share functional units with any other ALU modules in the vector unit. As a result, their area cost is relatively large at 450 LUTs for 8 additional instructions, however it is still below the 1000 LUT threshold and does not need to be further subdivided.

5.4 Multiplication and Shifting

To round out the basic arithmetic abilities of the RVV-Lite, integer multiply and shift capabilities must be included. Vector acceleration of these operations is essential for acceleration of common image and data processing operations like pixel representation conversion and neural network layer processing (namely matrix multiplication and convolution operations). However,

these operations are area- and therefore power-intensive, and are thus not included until the fourth layer of RVV-Lite.

This layer is subdivided into four sub-layers, each of which introduces unique hardware and contributes to the overall area of roughly 1100 LUTs and 8 DSPs. These sub-layers are evaluated separately, but are ultimately intended to be implemented as a single grouping.

5.4.1 8/16/32-bit Multiply and Shift Left

```
vmul.VX    vd,vs2,YY,vm     # vd[i] = LSB(vs2[i] * YY) (8/16/32b)
vsll.VXI   vd,vs2,ZZ,vm     # vd[i] = vs2[i] << YY     (8/16/32b)
```

The first and most foundational grouping of multiply and shift instructions are the basic multiply-low and left shift operations. The multiply-low instructions take two inputs and return the lower half of the multiplier output. A 32-bit multiplier, where only the low 32-bit result is needed, can be built using three 16-bit multipliers using the simple distributive property

$$(A * 2^{16} + B)(C * 2^{16} + D) = AC * 2^{32} + (AD + BC) * 2^{16} + BD \quad (5.1)$$

The left-shift instructions, being a multiplication by a power of 2, can operate using the same multiplier hardware using one input (as the power of 2) generated from the shift-amount operand. Of the products calculated in Equation 5.1, only the latter three are within the scope of a multiply-low operation, and thus the multiplier hardware required to add the result AC to the outputs of the other multipliers is never implemented. However, the result is still calculated as a part of the 16-bit element multiplication.[2]

This sub-layer introduces two 32-bit multipliers, each of which is capable of producing one 32-bit, two 16-bit, or four 8-bit results. Due to the structure of the fracturable multiplier, the module introduces four DSP blocks for each 32-bit multiplier. The layer also introduces just over 300 LUTs in multiplexing and addition logic.

[2] In addition to the three 16-bit multipliers, one 8-bit multiplier is also needed to compute the four 8-bit multiplications. In the next sub-layer, this fourth multiplier will become a full 16-bit multiplier

5.4.2 8/16-bit Multiply High and Shift Right

Here, `MULH` refers to the all three `mulh` operations: signed, unsigned, and signed-unsigned.

```
vsr{l/a}.VXI   vd,vs2,ZZ,vm # vd[i] = vs2[i] {>}>> YY   (8/16b)
vMULH.VX   vd,vs2,YY,vm     # vd[i] = MSB(vs2[i] * YY)  (8/16b)
```

This next sub-group includes only the 8- and 16-bit versions of the multiply-high and right shift instructions. It again makes use of the multipliers from the previous sub-layer, while requiring additional hardware to extract the upper 8-/16-bits of a given operation. The shift right operation is implemented using the multiply-high logic paths, as the operation

$$A[Y-1:0] >> X$$

is functionally equivalent to

$$\{A[Y-1:0] * 2^{Y-X}\}[2Y-1:Y]$$

Accordingly, these functions can be implemented together with very little overhead. By re-using the fracturable 32-bit multiplier, these operations are nearly free, requiring about 24 LUTs.

5.4.3 32-bit Multiply High and Shift Right

```
vsr{l/a}.VXI   vd,vs2,ZZ,vm # vd[i] = vs2[i] {>}>> YY  (32b)
vMULH.VX   vd,vs2,YY,vm     # vd[i] = MSB(vs2[i] * YY) (32b)
```

The 32-bit versions of these instructions require the use of a full 32x32-bit multiplier, with a 64-bit output. Thus, all results calculated in equation 5.1 must be implemented, requiring extra adder hardware to compute the full 64-bit result. Because the higher 32 bits are not used in any previous sub-layer, the logic required to calculate and route those results resides entirely in this sub-layer. It requires roughly 300 LUTs, which is comparable to the initial multiplication unit logic.

5.4.4 Widening Multiply and Narrowing Shift

The sub-grouping includes the widening signed and unsigned multiply-low operations (`MUL`), as well as the signed-unsigned version.

```
vwMUL.VX   vd,vs2,YY,vm     # vd[i] = vs2[i] * YY     (8/16/32b)
vwmulsu.VX vd,vs2,YY,vm     # vd[i] = vs2[i] S*U YY   (8/16/32b)
vnsrl.wX   vd,vs2,x0,vm     # vd[i] = vs2[i]          (16/32/64b)
```

The final grouping of multiply and shift instructions implemented are the widening and narrowing versions of the aforementioned instructions. These instructions are the last subset of 8-/16-/32-bit instructions examined, as they require the full 64-bit width of the 32x32-bit multiplier to be routed to a single output, which introduces additional multiplexing hardware not required for previous groupings.

While widening multiplications are low in cost, as the ALU can re-use the widening unit from the widening addition and subtraction layer, narrowing is a new concept; it is not a shift-right, rather it selects the lower half of each element from the multiplier and generates the appropriate byte-write enable signal to write to either the top or bottom half of the appropriate offset in the vector register file. It also requires additional logic for the destination register address generation unit in order to write to the same register two cycles in a row. This is done with a different byte-enable signal so no data is overwritten, but is required because narrowing produces an output that is half the size of the input. Thus, data from two consecutive clock cycles will need to be written to the same offset in the destination vector register.

Together, these instructions introduce just under 500 LUTs, making them the largest sub-layer in the 32-bit multiply and shift grouping. Due to its size, it is possible that future iterations will separate these instructions into their own layers, much like the widening add/subtract layer is separate from the add/subtract sub-layer of the core layer.

5.5 Slide-by-N

```
vslideup.XI   vd,vs2,ZZ,vm  # vd[i+ZZ] = vs2[i]
vslidedown.XI vd,vs2,ZZ,vm  # vd[i] = vs2[i+ZZ]
```

When the vector engine has a DATA_WIDTH less than or equal to the largest supported element width (64-bits), the slide-by-N instruction can be implemented with virtually no overhead. However, for slides or DATA_WIDTH larger than 64 bits extra multiplexing hardware is required to ensure elements are inserted in both the right data lane and the right index. Additionally, for slides amounts greater than the DATA_WIDTH, extra logic is required to advance the pointer in the AGU before reading from the vector register file and decrement the slide amount to be within the bounds of the slide-by-1 hardware.

This layer requires roughly 300 LUTs, which are mostly from the read offset generation logic. For data widths larger than 64, additional LUTs will be required to shift bytes spatially to any byte offset beyond 64 bits within the full datapath width, as those operations are not supported in the slide-by-1 hardware.

5.6 64-bit Multiplication and Shifting

```
vmul.VX  vd,vs2,YY,vm       # vd[i] = LSB(vs2[i] * YY)
vsll.VXI vd,vs2,ZZ,vm       # vd[i] = vs2[i] << YY
vsr{l/a}.VXI vd,vs2,ZZ,vm   # vd[i] = vs2[i] {>}>> YY
```

The final grouping of multiplication and shifting instructions supported are the 64-bit multiply-low, shift left, and shift right operations. These instructions require additional multiplier hardware; results would come from either one 64x64-bit multiplier with a 128-bit wide output or four 32-bit multipliers, arranged using distributive properties where

$$(A * 2^{32} + B)(C * 2^{32} + D) = AC * 2^{64} + (AD + BC) * 2^{32} + BD \quad (5.2)$$

Regardless of implementation, adding 64-bit multiplication capabilities requires significant hardware. Zve* does not include the 64-bit multiply-high instructions, as doing so would require a large multiplier with 128-bit output to be implemented. Without this 128-bit output multiplier, implementing a 64-bit right shift (for shifts less than 32) is less straightforward, but can

be done by multiplying the lower 32 bits by $2^{32-shift}$, shifting the upper 32 bits by $shift$, and concatenating the results.

Because 64-bit multipliers are large, and are not required for 32-bit processors, these instructions are placed in the second-last layer. However, because a complete implementation of the primary layers should support all four data widths, these instructions are not considered optional.

We do not need to support multiply-high instructions for this layer, and so we are able to support these instructions using fewer than 250 LUTs. This is a similar area cost to the 32-bit multiply instruction sub-layers. However, three additional DSPs are also required to calculate the third 32-bit partial-product. We reduce the additional area gain for shift-right operations by implementing the 64-bit shifts using a 32x32-bit multiplier and some additional shifting logic for the upper 32 bits. This avoids the need to use a full 128-bit multiplier to achieve the same result. The hardware is illustrated in Figure 4.5.

5.7 Fixed-Point Operations

```
vaADDSUB{U}.VX  vd, vs2, YY, vm # round_US(vs2[i] ADDSUB{U} YY, 1)
vsmul.VX  vd, vs2, YY, vm       # vd[i]=clip(round_S(vs2[i]*YY,SEW-1)) (no 64b)
vssr{l/a}.VXI vd, vs2, ZZ, vm   # vd[i]=round_{U/S}(vs2[i],ZZ)
```

The final primary layer contains the fixed-point saturating multiply/shift and averaging add instructions. The saturating multiplication instruction is only supported for up to 32-bit elements, as Zve* does not require 64-bit support for this instruction.

Fixed-point instructions introduce the need for an averaging unit in the addition unit, bitwise-or logic to determine the rounding for the multiplication results, as well as new output multiplexing logic in order to extract the correct bits from the multiplication and shifting output. This layer requires roughly 250 LUTs to implement.

Chapter 6

Optional Extensions (not layered)

The following groupings of instructions are included as optional extensions in RVV-Lite v0.5. They are not dependent on any layers other than the core layer (A.1) described in Chapter 5. Their implementation is not necessary to form a complete RVV-Lite processor. Implementation results for the additional mask extension are included, however other extensions were separated due to functional differences and have not yet been implemented.

6.1 Additional Mask Extension

```
vlm.v       vd, (rs1)          # ld mask of ceil(vl/8) bytes
vsm.v       vs3,(rs1)          # st mask of ceil(vl/8) bytes
vadc.VXIm   vd,vs2,YY,v0       # vd[i]=vs2[i]+vs1[i]+v0.m[i]
vmadc.VXm   vd,vs2,YY,v0       # vd.m[i]=cout(vs2[i]+vs1[i]+v0.m[i])
vmadc.VXI   vd,vs2,YY          # vd.m[i]=cout(vs2[i]+vs1[i])
vsbc.VXm    vd,vs2,YY,v0       # vd[i]=vs2[i]-vs1[i]-v0.m[i]
vmsbc.VXm   vd,vs2,YY,v0       # vd.m[i]=brrw(vs2[i]-vs1[i]-v0.m[i])
vmsbc.VX    vd,vs2,YY          # vd.m[i]=brrw(vs2[i]-vs1[i])
vcpop.m     rd,vs2,vm          # x[rd] = sum(vs2.m[i]), count bits
vfirst.m    rd,vs2,vm          # x[rd] = idx_of_first_one(vs2.m)
vmerge.VXIm vd,vs2,YY,v0       # vd[i] = v0.m[i] ? YY : vs2[i]
```

This optional grouping of mask instructions include the mask load and store instructions, add-with-carry and subtract-with-borrow, mask population count, find-first-set-bit, and mask merge instructions. These instructions require unique multiplexer and bit-counting hardware, as well as additional pipeline registers to carry forward the appropriate output address,

which lead to the additional 650 LUTs for three unique ALU modules, as well as additional logic in the add/subtract unit from the core layer.

There are two types of carry/borrow instructions: those that write a vector data register with the sum/difference, and those that write a vector mask primaryd on carry/borrow at the MSB. All of these operations require extra adders in the ALU add unit, and require additional logic paths to indicate whether the output of a given operation is a data vector or a mask vector, increasing the size of the add unit. These instructions are not considered part of the core layer, as they do not add much functionality. The carry- and borrow-out instructions can be replicated by a widening add followed by a narrowing shift right, and the carry- and borrow-in instructions can be replicated by two consecutive addition or subtraction instructions. However, introduction of these instructions does improve throughput of these operations by about 2x, and thus they are included in the optional mask extension.

The merge instruction also introduces a unique new datapath, thereby introducing additional area. Its functionality can be replaced by two OR instructions, where the first input is ORed with a 0 vector using a mask and the second input is ORed with the first result using the mask's inverse. This operation, however, requires twice the cycles to set each mask, and subsequently takes at least twice the number of cycles to produce the result after the masks are set. Accordingly, this operation creates potential for significant speedup over any RVV-Lite implementation without it, and it is therefore considered essential enough to be included in the optional mask extension.

The population count and find-first-set-bit instructions introduce unique operations to the vector unit, and cannot easily be replaced by other combinations of instructions. However, they are not frequently used instructions and are therefore considered part of the optional extension.

Also included are the mask load and store instructions. These are functionally equivalent to a memory instruction with LMUL=1, as the encoding uses the same width as a memory transaction for an SEW of 8, however extra logic is required to ensure only 1/8 the width of the register is ac-

cessed at once. Though this extra logic is not in itself area-intensive, it is not considered necessary as part of the primary layers as its functionality can be easily replicated by one of the aforementioned memory transactions at no area cost.

6.2 Strided Memory

```
vlseEW.v  vd,  (rs1), rs2, vm   # strided ld, EEW=EW
vsseEW.v  vs3, (rs1), rs2, vm   # strided st, EEW=EW
```

These instructions load elements separated by a gap of $rs2$ elements into vd from the address in $rs1$ (i.e. a stride of zero loads the same element repeatedly, a stride of one is a regular load, a stride of two would load every other element, etc). Strided memory instructions create unique datapath requirements for memory address generation, requiring extra logic to implement. This layer, and these instructions, are considered optional as they would greatly benefit applications that frequently perform these sorts of data accesses, like those that would use RVV-Lite to process image filters, neural networks, or tiled matrix multiplication. However, for applications that rarely require this, like a multi-level perceptron, the extra required area does not benefit the user enough to justify its inclusion in the primary instruction layers. In these situations, it is more beneficial to have the user organize their data contiguously, in a struct-of-arrays format, such that strided loads and stores are not required.

6.3 Indexed Memory

```
vl{u/o}xeiIW.v  vd,  (rs1), vs2, vm # {un}ordered, indexed load
vs{u/o}xeiIW.v  vs3, (rs1), vs2, vm # {un}ordered, indexed store
```

Like strided memory accesses, both ordered and unordered indexed memory accesses introduce unique datapath requirements to the memory address generator that are not otherwise used. Indexed accesses introduce the unique need for a byte-enable signal when reading or writing from memory to ensure the correct elements are written to the vector. While this additional

functionality can benefit applications that access data that is not stored in a predictable manner, most conventional benchmarks and applications will not use this instruction, or will use it infrequently. While supporting indexed loads will result in a worst case runtime of one memory transaction per element, not supporting it may result in even more cycles used by the scalar processor to rearrange the data. This tradeoff between extra memory hardware and cycle savings is wholly application-dependent, and accordingly these are considered optional instructions and are not included in the primary layer.

6.4 Floating-Point

In RVV, vector floating-point instructions are optional, but if included they require floating-point support in the scalar core. To remain forward-compatible, RVV-Lite must follow the same policy. To save area, an RVV-Lite vector processor is expected to share use of the scalar host's floating-point unit (FPU) as the scalar host is not expected to use its FPU 100% of the time. Implementation of any floating-point unit requires significant hardware, as floating-point numbers require calculation of both a new exponent and a new mantissa for every arithmetic operation.

The RVV floating-point instructions can be partitioned into two groups. The first group, which will be described in this optional extension, are vectorized versions of the RISC-V scalar floating-point instructions. This first group is included in this optional extension. The second group are new instructions with no scalar equivalent, and therefore would require additional hardware to support them. This second group is not included in RVV-Lite, and is instead relegated to Appendix C as part of the extra extension. The remainder of this section discusses the first group, presenting the instructions as sub-extensions with common features.

6.4.1 Type-casting, Move, Merge, and Slide

```
vmfEQ.VF      vd,vs2,YY,vm      # vd[i] = (vs2[i] EQ YY)
vfsgnj.VF     vd,vs2,YY,vm      # vd[i]={sgn(YY),abs(vs2[i])}
```

```
vfsgnjn.VF    vd,vs2,YY,vm      # vd[i]={~sgn(YY),abs(vs2[i])}
vfsgnjx.VF    vd,vs2,YY,vm      # vd[i]={sgn(YY*vs2[i]),abs(vs2[i])}
vfclass.v     vd,vs2,vm         # vd[i] = classify( vs2[i] )
vfcvt.{xu/x}.f.v     vd,vs2,vm  # float to {u}int
vfcvt.rtz.{xu/x}.f.v vd,vs2,vm  # float to {u}int (round to zero trunc.)
vfcvt.f.{xu/x}.v     vd,vs2,vm  # {u}int to float
vfmerge.vfm   vd,vs2,rs1,v0     # vd[i] = v0.m[i]?f[rs1]:vs2[i]
vfmv.v.f      vd,rs1            # vd[i] = f[rs1] (float splat)
vfmv.f.s      rd,vs2            # f[rd] = vs2[0] (rs1=0)
vfmv.s.f      vd,rs1            # vd[0] = f[rs1] (vs2=0)
vfslide1up.F   vd,vs2,YY,vm     # vd[i+1]=vs2[i],vd[0]=F[YY]
vfslide1down.F vd,vs2,YY,vm     # vd[i]=vs2[i+1],vd[L]=F[YY]
```

This most basic sub-extension includes floating-point instructions that require no additional hardware, as well as instructions that allow the existing integer ALU to be used for floating-point calculations.

Type-casting instructions (i.e. float to signed and unsigned integers and vice versa) are included in the lowest-level sub-group as they allow users to reuse the existing vector integer ALU for data that must be stored as a floating-point value.

The floating-point version of the set-mask-if-equal instruction is included as this is a zero-cost operation, as the functionality and instruction encoding is nearly identical to the integer version. Two floating-point values will only be equal if all bits are equal, which is functionally equivalent to the equivalent integer operation. Set not equal, while encoded differently than its integer counterpart, requires the same hardware and is thus virtually zero-cost. Other mask setting inequality operations are not included in this first sub-extension, as their calculation requires comparison of both the mantissas and exponents, creating additional logic requirements that are not supported in other layers.

Slide-by-1, merge, and move instructions are included in this set as their implementation is zero-cost. These instructions operate in exactly the same fashion as their 32-bit integer instruction counterparts, and their instruction encoding is identical, making their implementation free.

6.4.2 Basic Arithmetic and Mask Logic

```
vfADDSUB.VF   vd,vs2,YY,vm      # vd[i] = ADDSUB( vs2[i], YY )
vfrsub.vf     vd,vs2,rs1,vm     # vd[i] = f[rs1] - vs2[i]
vfMINMAX.VF   vd,vs2,YY,vm      # vd[i] = MINMAX( vs2[i], YY )
vmfGTE.vf     vd,vs2,rs1,vm     # vd[i] = (vs2[i] GTE f[rs1])
vmfLTE.VF     vd,vs2,YY,vm      # vd[i] = (vs2[i] LTE YY)
vfredMINMAX.vs vd,vs2,vs1,vm    # vd[0] = fMINMAX(vs1[0],vs2[*])
vfred{u/o}sum.vs  vd,vs2,vs1,vm # vd[0] = {un}ord_fsum(vs1[0],vs2[*])
```

This sub-extension includes the floating-point versions of the basic integer arithmetic instructions and arithmetic reduction instructions in the first three core sub-layers of RVV-Lite, including signed and unsigned add, subtract, min, and max values. These instructions are included here as, like their integer counterparts, they are fundamental in vectorized programs. Notably, the widening addition and subtraction instructions are excluded from this set, as they require specific support for double-precision floating-point numbers, which is not always guaranteed in a scalar host with a single-precision FPU.

This sub-group also includes the remaining inequality mask-set operations that are excluded from the first sub-group (set less than, set greater than, set less than or equal, set greater than or equal). While they require additional hardware to compare both mantissa and exponent, much like their integer counterparts their inclusion rounds out basic floating-point masking and allows users to selectively operate on data.

6.4.3 Multiplication

```
vfmul.VF      vd,vs2,YY,vm      # vd[i] = fmul( vs2[i], YY )
```

As in the integer instruction subset, floating-point multiplication is included in the optional floating-point extension after basic arithmetic instructions. Implementations that do not require multiplication are not required to implement this instruction, though those that choose to use the scalar FPU can do so easily.

6.4.4 Division

```
vfdiv.VF      vd,vs2,YY,vm      # vd[i] = fdiv( vs2[i], YY )
vfrdiv.vf     vd,vs2,rs1,vm     # vd[i] = f[rs1] / vs2[i]
```

Floating-point division is included in this sub-extension of floating-point instructions, as any scalar host with an FPU that natively supports floating-point division will already support this instruction.

6.4.5 Square Root

```
vfsqrt.v      vd,vs2,vm      # vd[i] = sqrt( v2[i] )
```

The final sub-group contains the square root operation. It is included here, as its functionality cannot easily be replicated by other floating-point instructions, but the calculation can be offloaded to the scalar FPU.

6.5 Double-Precision

The RVV double-precision extension requires the scalar host to support double-precision; for forward compatibility, RVV-Lite does the same. As with single-precision, the double-precision vector instructions can reuse a double-precision scalar FPU.

The following instructions all require double-precision floating-point support, and are grouped separately from the 32-bit floating-point instructions for that reason. Each of these instructions requires components of one or more operation in the previous group to function correctly, and thus any implementation that chooses to include this extension is required to implement the single-precision extension as well.

6.5.1 Widening Arithmetic Instructions

```
# widening floating-point, 2SEW = SEW op SEW (6 instr.)
vfwADDSUB.VF vd, vs2, YY, vm    # vs2[i] ADDSUB YY
vfwmul.VF    vd, vs2, YY, vm    # vs2[i] * YY
# widening floating-point, 2SEW = 2SEW op SEW (4 instr.)
vfwADDSUB.wVF  vd, vs2, YY, vm  # vs2[i] ADDSUB YY
```

This section includes the floating-point equivalents of the widening addition, subtraction, and multiplication instructions already included in the primary instruction layers. These instructions should be included in any processor which supports all widening integer instructions and has a double-precision FPU.

Also included are the versions of the widening addition and subtraction operations which take arguments of SEW and 2*SEW as input, and return a 2*SEW result. These operations are included because support for double-precision addition and subtraction is relatively straightforward.

6.5.2 Widening and Narrowing Cast Instructions

```
# floating-point conversion, widening (7 instr.)
vfwcvt.{xu/x}.f.v  vd, vs2, vm      # SEW float to 2SEW {u}int
vfwcvt.rtz.{xu/x}.f.v vd, vs2, vm   # SEW float to 2SEW {u}int (round to zero trunc.)
vfwcvt.f.{xu/x/f}.v  vd, vs2, vm    # SEW {{u}int/float} to 2SEW float
# floating-point conversion, narrowing (8 instr.)
vfncvt.{xu/x}.f.w vd, vs2, vm       # 2SEW float to SEW {u}int
vfncvt.rtz.{xu/x}.f.w vd, vs2, vm   # 2SEW float to SEW {u}int (round to zero trunc.)
vfncvt.f.{xu/x/f}.w vd, vs2, vm     # 2SEW {{u}int/float} to SEW float
vfncvt.rod.f.f.w vd, vs2, vm        # 2SEW float to SEW float (round to odd)
```

As with the integer to/from float cast instructions, support for all RVV widening signed/unsigned integer to float, widening float to signed/unsigned integer, narrowing signed/unsigned integer to float, and narrowing float to signed/unsigned integer instructions are included.

6.5.3 Widening reductions

```
# floating-point reductions, widening, 2SEW = 2SEW + SEW[*] (2 instr.)
vfwred{u/o}sum.vs vd, vs2, vs1, vm  # vd[0] = {un}ord_fsum(vs1[0],vs2[*])
```

Both ordered and unordered widening sum-reduction instructions are included in this floating-point extension, as their implementation simply requires that the second input is first cast to double-precision, and is then passed to the FPU as normal.

Chapter 7

Extra Extension

Extra instructions are grouped here for simplicity, and are not required to be implemented together. Implementations are not expected to need these instructions, and should be able to exclude them entirely. Reasoning for each instruction in this category is expanded upon below.

7.1 Extra Integer

The following instructions are all integer operations, and do not require support for any other datatypes.

7.1.1 Fault-only-first-load

```
vleEWff.v    vd, (rs1), vm  #    EEW=EW unit-stride fault-only-first load
```

This instruction is used to set the application vector length by reading until an invalid address is accessed. Per the RVV spec, use of this instruction is generally unsafe, as it can be used to quickly determine valid addresses [8]. In embedded applications which may have safety-critical components, or those which store or access sensitive data, security is highly important. As a result, implementation of this instruction is not recommended if security is of concern, as it can be easily avoided by simply programming using loops with finite bounds.

7.1.2 Segmented Memory Load and Store

```
vlseg<nf>e<eew>.v     vd, (rs1), vm            # Seg load template
vsseg<nf>e<eew>.v    vs3, (rs1), vm            # Seg store template
vlseg<nf>e<eew>ff.v   vd, (rs1), vm            # Fault-only-first seg load
```

```
vlsseg<nf>e<eew>.v    vd, (rs1), rs2, vm        # Strided segment load
vssseg<nf>e<eew>.v   vs3, (rs1), rs2, vm        # Strided segment store
vl{u/o}xseg<nf>ei<eew>.v  vd, (rs1), vs2, vm    # {Un}ord. idx seg load
vs{u/o}xseg<nf>ei<eew>.v vs3, (rs1), vs2, vm    # {Un}ord. idx seg store
```

Segmented loads are used to load and store data of equal sizes into consecutively numbered vector registers. They are used to unpack an array-of-structs into distinct registers, assuming the struct of elements are all the same size. This gives programmers the freedom to use an array-of-structs instead of the more vectorization-friendly struct-of-arrays. However, their functionality can be replaced by multiple strided loads.

7.1.3 Widening and Narrowing Mixed-Width Arithmetic

```
vw{add/sub}{u}.wVX  vd, vs2, YY, vm     # vd[i] = vs2[i] {+/-} YY    (no 64b)
vnsrl.wVXI vd, vs2, ZZ, vm              # vd[i] = vs2[i] >>> ZZ (rs1!=0, no 8b)
vnsra.wVXI vd, vs2, ZZ, vm              # vd[i] = vs2[i] >> ZZ (no 8b)
```

For these instructions, the operands are 2*SEW and SEW, and the result is either 2*SEW (widening) or SEW (narrowing). Implementation of these instructions requires reading each data packet of the narrower operand twice (once for each half), and widening it in order to successfully operate with the wider operand. This requires additional multiplexing hardware in the widening unit, as well as the address generation units, and can be easily replaced by an instruction that widens the second operand, followed by an addition/subtraction/shoft instruction at the wider element width. For this reason, they are not considered essential and are not included with the primary widening and narrowing instructions.

In the extra instruction extensions, the addition and subtraction instructions are grouped separately from the narrowing instructions, for the same reasons that they are separated in the primary layers.

7.1.4 Sign-Extended Widening Instructions

```
v{z/s}ext.vfD vd, vs2, vm           # vd[i] = {z/s}ext( vs2[i] )
```

These instructions widen a value to 2/4/8-times the input width, and sign extend the result. These instructions are not included by default, as support for widening from the input SEW to 4*SEW or 8*SEW is not otherwise supported, and would require additional area equal to at roughly double the area required to implement the initial widening add/subtract unit (200 LUTs). As a result, though these operations can be useful they can be replaced by chained widening instructions (i.e. sequential single- to double-width widening instructions), followed by signed multiplication (for negative values), and therefore may not be worth the additional area cost for applications that use these types of operations infrequently.

7.1.5 Division and Remainder

`DIVREM` refers to the signed and unsigned division and remainder instructions.

```
vDIVREM.vVX vd, vs2, YY, vm     # vd[i] = vs2[i] DIVREM YY
```

Integer division and remainder operations are notoriously hardware- and cycle-heavy, and are selcomd pipelined. Though their use cannot be readily replaced by a combination of other instructions, it is not recommended that applications requiring limited hardware and reasonable clock speeds implement such an instruction unless absolutely necessary. For this reason, these instructions are not included as part of the primary layers. If division is required, use of a vectorized software routine is recommended.

7.1.6 Multiply-Accumulate/Add (MAC/MADD)

```
vmacc.VX  vd, YY, vs2, vm       # vd[i] = +(YY * vs2[i]) + vd[i]
vnmsac.VX vd, YY, vs2, vm       # vd[i] = -(YY * vs2[i]) + vd[i]
vmadd.VX  vd, YY, vs2, vm       # vd[i] = +(YY * vd[i]) + vs2[i]
vnmsub.VX vd, YY, vs2, vm       # vd[i] = -(YY * vd[i]) + vs2[i]
vwmacc{u}.VX vd, YY, vs2, vm    # vd[i] = +(YY * vs2[i]) + vd[i]
vwmaccsu.VX vd, YY, vs2, vm     # vd[i] = +(S(YY) * U(vs2[i])) + vd[i]
vwmaccus.vx vd, rs1, vs2, vm    # vd[i] = +(U(x[rs1]) * S(vs2[i])) + vd[i]
```

All instructions in this group require three read ports to exist in the vector register file, which introduces not only additional multiplexing and address

generation hardware, but potentially another redundant copy of the vector data, depending on the memory structure used. In the case of an FPGA-based implementation such as this one, it requires an additional 16 BRAM blocks for a 16kB vector register file.

As these instructions can be easily replaced by a multiplication (signed, unsigned, or mixed) followed by an addition instruction, they are not essential to implement for most embedded applications.

7.1.7 Fixed-Point Saturating Arithmetic

```
vsADD.VXI vd, vs2, YY, vm       # vd[i] = saturate( vs2[i] + YY )
vsSUB.VX  vd, vs2, YY, vm       # vd[i] = saturate( vs2[i] - YY )
```

These instructions require specialized overflow detection hardware, which is not included for any other primary instruction, and requires multiplexing hardware to insert the maximum value in the case of overflow. These would most commonly be used in accumulators.

7.1.8 Clipped Narrowing

```
vnclip{u}.wVXI vd, vs2, vs1, vm    # vd[i] = clip{u}(roundoff_{S/U}(vs2[i], ZZ))
```

Signed and unsigned rounding narrow and clip instructions are not included in the primary layers, as they are used only with narrowing fixed point values. Because all included fixed point instructions in the primary layers will only grow in size, there is little need for such an instruction, though its function is not easily replaced by any other primary instruction.

7.1.9 Widening Reduction

```
vwredsum{u}.vs  vd, vs2, vs1, vm    # 2*SEW = 2*SEW + sum({s/z}ext(SEW))
```

These instructions are equivalent to a sign-extended sum-reduction at SEW with x[vs2]=0 followed by a cast to 2*SEW and an add with the expected x[vs1]. Alternately, this can be replaced by a widening add with 0 and a sum-reduction at 2*SEW. Because reduction operations require a dedicated

reduction tree, implementation of these instructions would require significant additional hardware (potentially 400+ LUTs for an FPGA), and as such they are not considered essential for embedded applications.

7.1.10 Mask Sum

```
vmsbf.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k< i]) ? 1 : 0
vmsif.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k<=i]) ? 1 : 0
vmsof.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k< i] && vs2.m[i]) ? 1:0
viota.m    vd, vs2, vm          # vd.m[i] = sum( vs2.m[ all k<i] )
```

These instructions set masks to all 1s before a given index and all 0s afterward. In order to do this without these instructions, one could simply use the `vfirst.m` instruction to determine the location of the first bit and change the VL accordingly to replicate the behaviour of the first two instructions. After doing so, applications could run any masked instruction to replicate the `vmsof` instruction, or could run the existing `vpopc` instruction to replicate the behaviour of `viota.m`.

Adding dedicated computations for these instructions would require additional multiplexers at the output of the `vfirst` module in order to send 1s or 0s to the output depending on whether the first 1 has been found. As with the previous unique mask instructions, this would greatly increase area with little payoff.

7.1.11 Data Movement (Gather/Compress)

```
vrgather{ei16}.vv vd,vs2,vs1,vm # vd[i]=(vs1[i]>=VLMAX)?0:vs2[vs1[i]];
vrgather.XI vd, vs2, ZZ, vm     # splat: vd[i]=(x[rs1]>=VLMAX)?0:vs2[ZZ] (splat)
vcompress.vm vd, vs2, vs1       # vd[i] = vs2[enum(vs1[*])], vs1[*] is a mask
```

Gather and compress operations, like indexed load operations, require unique multiplexing hardware in order to selectively move elements from one vector into another. These operations can be avoided by using the existing supported mask operations, indexed loads/stores, or organizing data differently to achieve the same results.

Significant hardware is required for the `vrgather.vv` and `vrgatherei16.vv` instructions, as there is no guarantee of index ordering or consistent separation between elements. Similarly, for `vcompress.vm`, though ordering is guaranteed, there is no guarantee of which indices will be enabled.

7.2 Extra Floating-Point

The instructions in this section all require either an FPU in the scalar host processor with single-precision support, or a dedicated floating-point vector ALU. However, the operations required by the instructions in this section have no counterpart in the RISC-V floating-point extension (named F).

7.2.1 MAC Operations

```
# floating-point MAC, SEW = SEW * SEW +/- SEW (16 instr.)
vf{n}macc.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {-/+} vd[i]
vf{n}msac.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
vf{n}madd.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {-/+} vs2[i]
vf{n}msub.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {+/-} vs2[i]
```

As with the integer versions of this operation, implementation of these instructions may require a third vector register file read port. As RISC-V has no MAC instruction, the scalar host's FPU is not expected to natively support these instructions, thus their inclusion would require new FPU hardware.

7.2.2 7-bit Accurate Estimations

```
vfrsqrt7.v  vd, vs2, vm         # vd[i] = 1.0 / sqrt( v2[i] )
vfrec7.v    vd, vs2, vm         # vd[i] = 1.0 / v2[i]
```

Standard RISC-V floating-point units do not natively support these operations, thus their inclusion would require separate support. Though potentially useful, the reciprocal square root operation can be replaced by the normal square root followed by an inverse operation ($\frac{1}{x}$). This results in no degradation in accuracy and it can reuse the existing FPU without requiring

any additional hardware. The use of this operation is not frequent enough in most applications to justify the hardware required to included it in a general purpose vector unit.

7.3 Extra Double-Precision

```
# floating-point MAC, widening, 2SEW = SEW*SEW +/- 2SEW (8 instr.)
vfw{n}macc.VF  vd, YY, vs2, vm  # vd[i] = {-/+} (YY * vs2[i]) {-/+} vd[i]
vfw{n}msac.VF vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
```

Like their 32-bit float counterparts, these double-precision MAC operations require a third read port and new functional hardware to function, as the RISC-V double-precision extension ("D"), has no MAC instruction. For this reason, they are excluded from the initial double-precision group.

Chapter 8

Experimentation and Results

The experiments run for this thesis were predominantly done to verify that the implemented version of RVV-Lite is smaller than other full RVV implementations and is capable of running benchmarks in either a comparable number of cycles, or a shorter overall runtime. The following sections state and analyze results that verify each of these.

8.1 Benchmark Performance

The benchmarks in this section were implemented prior to the original design of RVV-Lite. This benchmark set includes AXPY[3] (a * x + y), general matrix multiplication (GEMM), 2D FIR filter, RGB to LUMA pixel conversion, and median image filter. The first two benchmarks were chosen largely to show that lack of support for a unified multiply-accumulate instruction does not significantly detract from the accelerator's performance. The FIR filter incorporates a majority of the supported layers. In contrast, the RGB benchmark uses fewer layers and has no MAC operations. The median filter uses only the core layer. Table 8.1 shows a summary of the different layers required for each listed benchmark.

These benchmarks were initially intended to benchmark RVV, and not RVV-Lite. They were not designed with our final layering sequence in mind, so not all layers are included in our benchmark set. However, it is easy to include these missing layers. For example, a transposed implementation of a GEMM would make use of the reduction sum instruction from layer A.3 in place of the add instruction.

[3]This benchmark is the integer version of the commonly used SAXPY (single-precision a*x + y) benchmark. There is no formal name for the integer version.

Table 8.1: Summary of layers required for each benchmark

Benchmark	A.1	A.2	A.3	A.4	A.5	A.6 [a]	A.7	B.1
AXPY	✓			✓		✓		
GEMM	✓			✓		✓		
2D FIR	✓			✓	✓	✓	✓[b]	
RGB2LUMA	✓	✓		✓				
MEDIAN	✓							

[a] 64-bit multiplication would only be required for the 64-bit version of these benchmarks

[b] Fixed-point may be required here, as traditionally this benchmark is done with either fixed- or floating-point values

8.2 Comparison to Related Work

Table 8.2 shows the benchmark results for SEW=8 in terms of cycles and overall time (using calculated FMax) to show that Saturn-V is a viable vector accelerator. Benchmarks with "(no ld)" indicate results when benchmarks are run assuming memory latency of 1 cycle per incoming data packet. This is done by replacing load operations with a vid instruction. While the results show that it is not the fastest implementation that exists when load operations are included, the memory system is not the primary focus of this implementation.

It is important to note that all numbers reported for Vicuna are from the creator's own paper, which focuses on the initially released version from July 2021. When disassembled GEMM and AXPY benchmarks from Vicuna's repository are compared to our own versions, the major differences are that Vicuna is able to process the MAC instruction, and due to their shorter VLEN (128 for their compact version) they use an LMUL of 4. Apart from this, the GEMM benchmarks both use one row of the non-transposed input matrix B, multiply with a scalar element from input matrix A (moving across one row at a time), and add to the same row of output C. We do not have access to the benchmarks for Arrow, and as a result we cannot confirm that our benchmarks use the same data access pattern.

While Saturn-V takes a higher number of cycles to compute a general matrix multiply when loaded from memory than Vicuna does, this is largely in part due to the high latency associated with memory load transactions.

Table 8.2: Comparison of cycle count for benchmarks on Saturn-V and Related Work (64b data width versions)

Vector Engine	Benchmark	Size	Cycles	FMax (MHz)	T (ms)
Saturn-V	2D FIR	256^2, 4^2 taps	7,353,565	172.0	42.8
Saturn-V	RGB2LUMA	1024^2	17,104,061	172.0	99.4
Saturn-V	MEDIAN	$1k * 128$	5,338,474	172.0	31.0
Saturn-V	AXPY	65536	601,024	172.0	3.49
Saturn-V	GEMM	256^2	32,018,107	172.0	186.9
Saturn-V	GEMM	512^2	253,092,877	172.0	1,471.5
Saturn-V	GEMM	1024^2	2,032,177,435	172.0	11,815.0
Saturn-V	AXPY (no ld)	65536	61,588	172.0	0.4
Saturn-V	GEMM (no ld)	256^2	4,270,125	172.0	24.8
Saturn-V	GEMM (no ld)	512^2	33,728,548	172.0	196.1
Saturn-V	GEMM (no ld)	1024^2	268,830,079	172.0	1,563.0
Vicuna 1	AXPY	65536	108,985	77.7	1.4
Vicuna 1	GEMM	256^2	4,758,824	77.7	61.2
Vicuna 1	GEMM	1024^2	256,277,942	77.7	3,298.3
Arrow	GEMM	512^2	120,000,000	112.0	1071.42

Currently, Saturn-V is connected to CFU-Playground, which supports only AXI-Lite and Wishbone interfaces at this time. Because CFU-Playground does not yet support AXI4 ports with burst mode, the latency for memory operations is significantly higher, as each packet must be sent individually. In CFU-Playground, this can result in up to 10 cycles[4] between outgoing read requests. When receiving data, all data must be received before it is passed to the processor. This creates a bottleneck. Beyond this, Vicuna allows caching of data, which can reduce memory load latency.

To show the impact that a faster memory system would have, Table 8.2 also includes versions of the GEMM and AXPY benchmarks in which all load operations are replaced by instructions with single-cycle latency per data packet (vid and vmv). Results using these versions show that the difference in cycle counts is largely due to to the latency of the load operations.

When comparing to Arrow [1], it is important to remember that we are not sure of the exact instruction subset that is implemented. Given the use of the matrix multiply benchmark, it is quite possible that the MAC

[4]this number was measured in simulation

Table 8.3: Implementation breakdown for Saturn-V (32b SIMD width, 512B register file / VLEN=128)

Configuration	LUT	BRAM	DSP	FMAX (MHz)
A.1 Core Instructions (32b)	1,523	3	0	214.9
A.2 Widening Add/Sub (32b)	2,198	3	0	217.4
A.3 Reduction (32b)	2,370	3	0	217.2
A.4 Mul/Shift: (32b)	2,907	3	4	230.6
A.5 Slide-by-N (32b)	2,979	3	4	214.6
A.7 FXP (32b)	3,074	3	4	195.5
B.1 Mask (+A.7) (32b)	3,382	3	4	202.8
A.7+B.1+VexRiscv (VLMAX=128) (32b)	12,957	25	11	202.8

instruction is included in this implementation. Because this instruction combines two instructions, it initially seems to make sense that the execution time for the same matrix size would be larger on Saturn-V, as it must run two instructions for every MAC operation. However when load instruction latency is removed from the equation our implementation is much faster than Arrow in both cycle count and run time.

To more accurately compare to Vicuna, the only other FPGA-based RVV 1.0 implementation we are aware of, we synthesized Saturn-V with a maximum element width of 32 and a VLEN of 128. To ensure no 64-bit calculations were included, the engine width for Saturn-V was set to 32. Each of the layers in Table 8.3 is roughly half the size of its 64-bit counterpart, which is expected.

Table 8.4: Comparison of Saturn-V configurations to other RVV implementations

Configuration	DATA_WIDTH	ELEN	LUT	BRAM	DSP	FMAX (MHz)
A.7 FXP	32	32	3,074	3	4	195.5
B.1 Mask (+A.7)	32	32	3,382	3	4	202.8
Vicuna 1	32	32	9,136	0	4	67.1
Vicuna 2	32	32	4,652	0	5	84.6
A.7 FXP	64	32	4,806	6	8	217.2
B.1 Mask (+A.7)	64	32	5,343	6	8	220.4
Vicuna 1	64	32	9,464	0	8	77.7
Vicuna 2	64	32	8,581	0	10	72.0
A.7 FXP	64	64	5,651	6	11	173.9
B.1 Mask (+A.7)	64	64	6,349	6	11	167.2
Arrow	64	64	474	32	-	112.0

Table 8.4 directly compares the version of Saturn-V containing all primary layers (A.1 through A.7) as well as the version including the optional mask extension to Vicuna. As expected, the reduced instruction set of RVV-Lite is smaller than both full implementations of Vicuna when compiled with the same VLEN and 32-bit data width. Additionally, Saturn-V is able to operate at a much higher clock speed on the same FPGA device.

The second section in the table compare the results for a 64-bit data width. All Saturn-V and Vicuna rows use a VLEN of 128 bits, as this is the default setting for the latest versions of Vicuna. Saturn-V is able to fit in a reasonably small area, and due to aggressive pipelining of the ALU is able to operate at a significantly higher clock speed than both versions of Vicuna when targeted to the same Xilinx board (ZCU104).

Chapter 9

Conclusion

RVV was initially designed as a monolithic block of instructions, without regard for how much area would be required by the total sum. This is the first work to formally examine reduced versions of the RVV extension to save area. Based on the results in Chapter 8, it is clear that dividing the instructions in RVV into layers is an effective way to reduce the area of a vector engine without losing performance, regardless of the chosen maximum element size and data width of the vector engine.

9.1 Recommendations

When implementing RVV-Lite, we recommend implementing only the required layers in order to save as much area as possible. Similarly, implementing only the required optional extensions is recommended, as this will minimize overall area use.

When implementing RVV-Lite, it is also recommended that 64-bit support not be included when connected to a 32-bit host CPU, as 64-bit support can greatly increase area, which was covered in Chapter 8.

9.2 Strengths and Limitations

This work reduces the existing RISC-V instruction set to a smaller overall area than other existing vector engines implementing RVV 1.0. While the benchmark performance is not state-of-the-art when memory latency is included, the overall performance is enough that reducing and layering this instruction set can feasibly improve performance of vectorizable applications in embedded systems without requiring significant area.

Currently, Saturn-V is connected to CFU-Playground, which has a lot of area overhead because it is a general purpose CPU. Accordingly, the overall system size is much larger than it might otherwise be with a more specialized RISC-V CPU. Additionally, the CFU-Playground environment is built using LiteX, which supports only AXI-Lite and Wishbone interfaces at this time. As a result, memory transaction latency is as high as 10 cycles per 64-bit package for memory requests, because AXI-Lite has no support for burst transactions. For this same reason, incoming packets are currently stored until the last memory request has been sent, in order to ensure no data is lost. While this guarantees predictable memory transfers, it significantly increases memory transaction time, and reduces overall performance as a result.

9.3 Future Work

Future work will include testing Saturn-V with other scalar RISC-V hosts, including those with 64-bit element support. In order to do this, other processors which do not currently implement the CFU port will need to be modified to include one. Unlike other RVV implementations, there will be no need to modify the vector processor itself in order to support other scalar hosts.

RVV-Lite also needs further tuning to ensure it is an optimal subsetting of the RVV specification. After examining the area results in Table 3.1, it may be prudent to consider dividing larger layers (like the core layer) into separate layers. As well, after examining which layers are most commonly used for benchmarks in Table 8.2, the ordering of layers may be changed to better suit these and other applications. However, the existing Saturn-V implementation should be further optimized before new decisions are made regarding the structure of RVV-Lite.

In order to improve memory transaction latency we intend to implement AXI4 support for the memory queue in order to support burst transactions. This will significantly improve memory transaction latency, and will allow for a significantly smaller incoming data queue. Incoming data after requests

are sent will be able to be transferred directly to the register file, since data packets will arrive in a burst instead of arriving up to 10 cycles apart.

We also intend to implement the remaining optional floating point, double-precision floating-point, and advanced memory extensions. The floating-point instructions will be implemented by adding a second FPU, and redirecting each half of the 32-bit data from the vector unit to one of the two FPUs. This can be done because the FPU is not expected to be in use by the scalar host 100% of the time. The double-precision instructions require only the main FPU for vector units with a data with of 64, as the FPU will already accept and produce 64-bit results. The advanced memory instructions will require address generation hardware and multiplexers in order to request the appropriate addresses and select the appropriate bytes of data that are returned in each 64-bit-wide packet.

Finally, future work will include optimizing the existing ALU instructions to reduce area and improve overall performance to make a competitive vector processor.

Bibliography

[1] Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen A. R. Saghir. Arrow: A RISC-V vector accelerator for machine learning inference. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2021.

[2] Matheus A. Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI. *CoRR*, abs/1906.00478, 2019.

[3] Christopher Chou, Aaron Severance, Alex Brant, Zhiduo Liu, Saurabh Sant, and Guy Lemieux. VEGAS: Soft vector processor with scratch-pad memory. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, pages 15–24, 01 2011.

[4] Mike Demler. Andes plots RISC-V vector heading. *The Linley Group Microprocessor Report*, 2020.

[5] Digilent. Arty A7 reference manual.

[6] Erik Engheim. Advantages of RISC-V vector processing over x86 style SIMD. *ITNEXT*, 2022.

[7] Jan Gray, Tim Vogt, Tim Callahan, Charles Papon, Guy Lemieux, Maciej Kurc, and Karol Gugala. Introducing composable custom extensions and custom function units for RISC-V, 2022.

[8] RISC-V International. RISC-V "V" vector extension, 2021.

[9] Aakash Jani. SiFive brings vectors to S-series, 2021.

[10] Florent Kermarrec, Sébastien Bourdeauducq, Jean-Christophe Le Lann, and Hannah Badier. LiteX: an open-source SoC builder and library based on Migen Python DSL, 2020.

[11] Albert Ou, Quan Nguyen, Yunsup Lee, and Krste Asanovic. A case for MVPs : Mixed-Precision Vector Processors. In *International Workshop on Parallelism in Mobile Platforms (PRISM)*, June 2014.

[12] Michael Platzer and Peter Puschner. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan Green, Pete Warden, Timothy Ansell, and Vijay Janapa Reddi. CFU Playground: Full-stack open-source framework for tiny machine learning (tinyML) acceleration on FPGAs. 01 2022.

[14] Colin Schmidt, John Wright, Zhongkai Wang, Eric Chang, Albert Ou, Woorham Bae, Sean Huang, Vladimir Milovanović, Anita Flynn, Brian Richards, Krste Asanović, Elad Alon, and Borivoje Nikolić. An eight-core 1.44-GHz RISC-V vector processor in 16-nm FinFET. *IEEE Journal of Solid-State Circuits*, 57(1):140–152, 2022.

[15] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft vector processors with streaming pipelines. In *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2014.

[16] Aaron Severance and Guy G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10. IEEE, 2013.

[17] John Charles Wright, Colin Schmidt, Ben Keller, Daniel Palmer Dabbelt, Jaehwa Kwak, Vighnesh Iyer, Nandish Mehta, Pi-Feng Chiu, Stevo Bailey, Krste Asanović, and Borivoje Nikolić. A dual-core RISC-V vector processor with on-chip fine-grain power management in 28-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2721–2725, 2020.

[18] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008.

[19] Peter Yiannicouras. *FPGA-Based Soft Vector Processors*. PhD thesis, University of Toronto, 2009.

Appendix A

Base Instruction Layers

This summarizes the instructions in RVV-Lite v0.5, which subdivides the RISC-V Vector Specification into 7 layers of instructions, 5 optional components, and 3 extra layers. The notation below, which is unique to this work, condenses the RVV-Lite specification.

- NYI denotes a layer that is Not Yet Implemented
- VXIF denotes up to 4 operand modes: V=vv, X=vx, I=vi, F=vf
- YY denotes up to 3 operand types: vs1/rs1/imm
- ZZ denotes up to 3 operand types: vs1/rs1/uimm
- rs1 is from X or F register set, i.e. X[rs1] or F[rs1]
- K denotes register group size: 1, 2, 4, 8
- EW/IW denotes element/index width: 8, 16, 32, 64
- KEW denotes reg group/EW sizes: 1re8, 2re16, 4re32, 8re64
- L denotes index of last element, i.e. L = VL-1
- S/U denotes signed/unsigned
- LOP (logic-op): and, or, xor
- FOP (float-op): add, sub, mul, div, min, max
- MOP (mask-op) and, nand, andnot, xor, or, nor, ornot, xnor
- ROP (reduction-op): sum, and, or, xor, maxu, max, minu, min
- MCMP (mask-compare): seq, sne, sle, sleu
- ADD/SUB/MUL/MIN/MAX: add/sub/mul/min/max, addu/subu/mulu/minu/maxu
- MULH: mulh, mulhu, mulhsu
- ADDSUB{U}/MINMAX{U}/EQ/GTE/LTE: add{u}/min{u}/eq/gt/lt, sub{u}/max{u}/ne/ge/le
- SLT/SGT: slt/sgt, sltu/sgtu

A.1 84 Core (8/16/32/64b)

```
a) vsetvli      rd,rs1,vtypei       # rd=VL=f(AVL), AVL=rs1, new vtype
a) vsetivli     rd,uimm,vtypei      # rd=VL=f(AVL), AVL=uimm, new vtype
a) vsetvl       rd, rs1, rs2        # rd=VL=f(AVL), AVL=rs1, vtype=rs2
a) vleEW.v      vd,(rs1),vm         # vector load, EEW=EW
a) vseEW.v      vs3,(rs1),vm        # vector store, EEW=EW
b) vid.v        vd,vm               # vd[i] = i
b) vLOP.VXI     vd,vs2,YY,vm        # vd[i] = vs2[i] LOP YY
b) vADDSUB.VXI  vd,vs2,YY,vm        # vd[i] = vs2[i] ADDSUB YY
b) vrsub.XI     vd,vs2,YY,vm        # vd[i] = YY - vs2[i]
c) vMINMAX{U}.VX vd,vs2,YY,vm       # vd[i] = MINMAX{U}(vs2[i], YY)
d) vmMCMP.VXI   vd,vs2,YY,vm        # vd.m[i] = (vs2[i] MCMP YY)
d) vmSLT.VX     vd,vs2,YY,vm        # vd.m[i] = (vs2[i] < YY)
d) vmSGT.XI     vd,vs2,YY,vm        # vd.m[i] = (vs2[i] > YY)
d) vmMOP.mm     vd,vs2,vs1          # vd.m[i] = MOP(vs2.m[i],vs1.m[i])
e) vmv.v.VXI    vd,YY               # vd[i] = YY, XI modes: integer splat
e) vmv.x.s      rd,vs2              # x[rd] = vs2[0], scalar copy (vs1=0)
e) vmv.s.x      vd,rs1              # vd[0] = x[rs1], scalar copy (vs2=0)
f) vmvKr.v      vd,vs2              # whole-vec. reg. group copy EMUL=K
f) vlKEW.v      vd,(a0)             # whole reg EMUL=K, VLEN/EW elements, ignores VL
f) vsKr.v       vd,(a1)             # whole reg EMUL=K, VLEN bits, ignores VL
g) vslide1up.vx     vd,vs2,rs1,vm   # vd[i+1]=vs2[i], vd[0]=X[rs1]
g) vslide1down.vx   vd,vs2,rs1,vm   # vd[i]=vs2[i+1], vd[L]=X[rs1]
```

A.2 4 Widen Add/Sub (8/16/32b)

```
vwADDSUB{U}.VX   vd,vs2,YY,vm      # vd[i] = vs2[i] ADDSUB{U} YY
```

A.3 8 Reduction (8/16/32/64b)

```
vredROP.vs vd,vs2,vs1,vm        # vd[0]=ROP(vs1[0],vs2[*])
```

A.4 38 Mul/Shift (mostly 8/16/32b)

```
a) vmul.VX      vd,vs2,YY,vm    # vd[i] = LSB(vs2[i] * YY)  (8/16/32b)
a) vsll.VXI     vd,vs2,ZZ,vm    # vd[i] = vs2[i] << YY      (8/16/32b)
b) vsr{l/a}.VXI vd,vs2,ZZ,vm    # vd[i] = vs2[i] {>}>> YY   (8/16b)
b) vMULH.VX     vd,vs2,YY,vm    # vd[i] = MSB(vs2[i] * YY)  (8/16b)
c) vsr{l/a}.VXI vd,vs2,ZZ,vm    # vd[i] = vs2[i] {>}>> YY   (32b)
c) vMULH.VX     vd,vs2,YY,vm    # vd[i] = MSB(vs2[i] * YY)  (32b)
d) vwMUL.VX     vd,vs2,YY,vm    # vd[i] = vs2[i] * YY       (8/16/32b)
d) vwmulsu.VX   vd,vs2,YY,vm    # vd[i] = vs2[i] S*U YY     (8/16/32b)
d) vnsrl.wX     vd,vs2,x0,vm    # vd[i] = vs2[i]            (16/32/64b)
```

A.5 4 Slide-by-N (8/16/32/64b)

```
vslideup.XI   vd,vs2,ZZ,vm      # vd[i+ZZ] = vs2[i]
vslidedown.XI vd,vs2,ZZ,vm      # vd[i] = vs2[i+ZZ]
```

A.6 11 Multiply/Shift (64b)

```
vmul.VX  vd,vs2,YY,vm       # vd[i] = LSB(vs2[i] * YY)
vsll.VXI vd,vs2,ZZ,vm       # vd[i] = vs2[i] << YY
vsr{l/a}.VXI vd,vs2,ZZ,vm   # vd[i] = vs2[i] {>}>> YY
```

A.7 16 Fixed-point (8/16/32/64b)

```
vaADDSUB{U}.VX  vd, vs2, YY, vm # round_US(vs2[i] ADDSUB{U} YY, 1)
vsmul.VX  vd, vs2, YY, vm       # vd[i]=clip(round_S(vs2[i]*YY,SEW-1)) (no 64b)
vssr{l/a}.VXI vd, vs2, ZZ, vm   # vd[i]=round_{U}S(vs2[i],ZZ)
```

Appendix B

Optional Features (not layered)

B.1 21 Mask (8/16/32/64b)

```
vlm.v       vd, (rs1)          # ld mask of ceil(vl/8) bytes
vsm.v       vs3,(rs1)          # st mask of ceil(vl/8) bytes
vadc.VXIm   vd,vs2,YY,v0       # vd[i]=vs2[i]+vs1[i]+v0.m[i]
vmadc.VXm   vd,vs2,YY,v0       # vd.m[i]=cout(vs2[i]+vs1[i]+v0.m[i])
vmadc.VXI   vd,vs2,YY          # vd.m[i]=cout(vs2[i]+vs1[i])
vsbc.VXm    vd,vs2,YY,v0       # vd[i]=vs2[i]-vs1[i]-v0.m[i]
vmsbc.VXm   vd,vs2,YY,v0       # vd.m[i]=brrw(vs2[i]-vs1[i]-v0.m[i])
vmsbc.VX    vd,vs2,YY          # vd.m[i]=brrw(vs2[i]-vs1[i])
vcpop.m     rd,vs2,vm          # x[rd] = sum(vs2.m[i]), count bits
vfirst.m    rd,vs2,vm          # x[rd] = idx_of_first_one(vs2.m)
vmerge.VXIm vd,vs2,YY,v0       # vd[i] = v0.m[i] ? YY : vs2[i]
```

B.2 4 Strided Memory (8/16/32/64b) - NYI

```
vlseEW.v  vd,  (rs1), rs2, vm   # strided ld, EEW=EW
vsseEW.v  vs3, (rs1), rs2, vm   # strided st, EEW=EW
```

B.3 8 Indexed Memory (8/16/32/64b) - NYI

```
vl{u/o}xeiIW.v  vd,  (rs1), vs2, vm # {un}ordered, indexed load
vs{u/o}xeiIW.v  vs3, (rs1), vs2, vm # {un}ordered, indexed store
```

B.4 48 Floating-point (32b) - NYI

```
a) vmfEQ.VF      vd,vs2,YY,vm       # vd[i] = (vs2[i] EQ YY)
a) vfsgnj.VF     vd,vs2,YY,vm       # vd[i]={sgn(YY),abs(vs2[i])}
a) vfsgnjn.VF    vd,vs2,YY,vm       # vd[i]={~sgn(YY),abs(vs2[i])}
a) vfsgnjx.VF    vd,vs2,YY,vm       # vd[i]={sgn(YY*vs2[i]),abs(vs2[i])}
a) vfclass.v     vd,vs2,vm          # vd[i] = classify( vs2[i] )
```

```
a) vfcvt.{xu/x}.f.v  vd,vs2,vm      # float to {u}int
a) vfcvt.rtz.{xu/x}.f.v vd,vs2,vm   # float to {u}int (round to zero trunc.)
a) vfcvt.f.{xu/x}.v  vd,vs2,vm      # {u}int to float
a) vfmerge.vfm   vd,vs2,rs1,v0      # vd[i] = v0.m[i]?f[rs1]:vs2[i]
a) vfmv.v.f      vd,rs1             # vd[i] = f[rs1] (float splat)
a) vfmv.f.s      rd,vs2             # f[rd] = vs2[0] (rs1=0)
a) vfmv.s.f      vd,rs1             # vd[0] = f[rs1] (vs2=0)
a) vfslide1up.F   vd,vs2,YY,vm      # vd[i+1]=vs2[i],vd[0]=F[YY]
a) vfslide1down.F vd,vs2,YY,vm      # vd[i]=vs2[i+1],vd[L]=F[YY]
b) vfADDSUB.VF   vd,vs2,YY,vm       # vd[i] = ADDSUB( vs2[i], YY )
b) vfrsub.vf     vd,vs2,rs1,vm      # vd[i] = f[rs1] - vs2[i]
b) vfMINMAX.VF   vd,vs2,YY,vm       # vd[i] = MINMAX( vs2[i], YY )
b) vmfGTE.vf     vd,vs2,rs1,vm      # vd[i] = (vs2[i] GTE f[rs1])
b) vmfLTE.VF     vd,vs2,YY,vm       # vd[i] = (vs2[i] LTE YY)
b) vfredMINMAX.vs vd,vs2,vs1,vm     # vd[0] = fMINMAX(vs1[0],vs2[*])
b) vfred{u/o}sum.vs  vd,vs2,vs1,vm  # vd[0]={un}ord_fsum(vs1[0],vs2[*])
c) vfmul.VF      vd,vs2,YY,vm       # vd[i] = fmul( vs2[i], YY )
d) vfdiv.VF      vd,vs2,YY,vm       # vd[i] = fdiv( vs2[i], YY )
d) vfrdiv.vf     vd,vs2,rs1,vm      # vd[i] = f[rs1] / vs2[i]
e) vfsqrt.v      vd,vs2,vm          # vd[i] = sqrt( v2[i] )
```

B.5 35 Double-precision (requires B.4) - NYI

```
# widening floating-point, 2SEW = SEW op SEW (6 instr.)
vfwADDSUB.VF vd, vs2, YY, vm    # vs2[i] ADDSUB YY
vfwmul.VF    vd, vs2, YY, vm    # vs2[i] * YY
# widening floating-point, 2SEW = 2SEW op SEW (4 instr.)
vfwADDSUB.wVF  vd, vs2, YY, vm  # vs2[i] ADDSUB YY
# floating-point conversion, widening (7 instr.)
vfwcvt.{xu/x}.f.v  vd, vs2, vm      # SEW float to 2SEW {u}int
vfwcvt.rtz.{xu/x}.f.v vd, vs2, vm   # SEW float to 2SEW {u}int (round to zero trunc.)
vfwcvt.f.{xu/x/f}.v  vd, vs2, vm    # SEW {{u}int/float} to 2SEW float
# floating-point conversion, narrowing (8 instr.)
vfncvt.{xu/x}.f.w vd, vs2, vm       # 2SEW float to SEW {u}int
vfncvt.rtz.{xu/x}.f.w vd, vs2, vm   # 2SEW float to SEW {u}int (round to zero trunc.)
vfncvt.f.{xu/x/f}.w vd, vs2, vm     # 2SEW {{u}int/float} to SEW float
vfncvt.rod.f.f.w vd, vs2, vm        # 2SEW float to SEW float (round to odd)
# floating-point reductions, widening, 2SEW = 2SEW + SEW[*] (2 instr.)
vfwred{u/o}sum.vs vd, vs2, vs1, vm  # vd[0] = {un}ord_fsum(vs1[0],vs2[*])
```

Appendix C

Extra Instructions

C.1 101 Extra Integer (requires A.7) - NYI

```
# (group a) fault-only-first load (4 instr.)
vleEWff.v    vd, (rs1), vm  #    EEW=EW unit-stride fault-only-first load
# (group b) segment load/store (32 instr.)
vlseg<nf>e<eew>.v     vd, (rs1), vm             # Seg load template
vsseg<nf>e<eew>.v    vs3, (rs1), vm             # Seg store template
vlseg<nf>e<eew>ff.v   vd, (rs1), vm             # Fault-only-first seg load
vlsseg<nf>e<eew>.v    vd, (rs1), rs2, vm        # Strided segment load
vssseg<nf>e<eew>.v   vs3, (rs1), rs2, vm        # Strided segment store
vl{u/o}xseg<nf>ei<eew>.v  vd, (rs1), vs2, vm    # {Un}ord. idx seg load
vs{u/o}xseg<nf>ei<eew>.v vs3, (rs1), vs2, vm    # {Un}ord. idx seg store
# (group c) wide+narrow ops, 2SEW = 2SEW op SEW (8 instr.)
vwADDSUB{U}.wVX  vd, vs2, YY, vm                # vd[i] = vs2[i] ADDSUB[U} YY    (no 64b)
# (group d) sign extension, widening from SEW/D to SEW, D = 2/4/8  (6 instr.)
v{z/s}ext.vfD vd, vs2, vm                       # vd[i] = {z/s}ext( vs2[i] )
# (group e) narrowing-shift, SEW = 2SEW >> SEW (6 instr.)
vnsrl.wVXI vd, vs2, ZZ, vm                      # vd[i] = vs2[i] >>> ZZ (rs1!=0, no 8b)
vnsra.wVXI vd, vs2, ZZ, vm                      # vd[i] = vs2[i] >> ZZ (no 8b)
# (group f) divide/remainder, DIVREM denotes div, divu, rem, remu (8 instr.)
vDIVREM.vVX vd, vs2, YY, vm     # vd[i] = vs2[i] DIVREM YY
# (group g) MAC and MADD, widening MAC (15 instr.)
vmacc.VX  vd, YY, vs2, vm       # vd[i] = +(YY * vs2[i]) + vd[i]
vnmsac.VX vd, YY, vs2, vm       # vd[i] = -(YY * vs2[i]) + vd[i]
vmadd.VX  vd, YY, vs2, vm       # vd[i] = +(YY * vd[i]) + vs2[i]
vnmsub.VX vd, YY, vs2, vm       # vd[i] = -(YY * vd[i]) + vs2[i]
vwmacc{u}.VX vd, YY, vs2, vm      # vd[i] = +(YY * vs2[i]) + vd[i]
vwmaccsu.VX vd, YY, vs2, vm     # vd[i] = +(S(YY) * U(vs2[i])) + vd[i]
vwmaccus.vx vd, rs1, vs2, vm    # vd[i] = +(U(x[rs1]) * S(vs2[i])) + vd[i]
# (group h) fixed-point, COST? needs: overflow detect, mux (5 instr.)
vsADD.VXI vd, vs2, YY, vm       # vd[i] = saturate( vs2[i] + YY )
vsSUB.VX  vd, vs2, YY, vm       # vd[i] = saturate( vs2[i] - YY )
# (group i) narrowing, CLIP denotes clip and clipu (6 instr.)
vnCLIP.wVXI vd, vs2, vs1, vm    # vd[i] = clip(roundoff_US(vs2[i], ZZ))
# (group j) widening reductions (2 instr.)
vwredsum{u}.vs  vd, vs2, vs1, vm    # 2*SEW = 2*SEW + sum({s/z}ext(SEW))
```

```
# (group k) mask (4 instr.)
vmsbf.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k< i]) ? 1 : 0
vmsif.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k<=i]) ? 1 : 0
vmsof.m    vd, vs2, vm          # vd.m[i] = (!vs2.m[ all k< i] && vs2.m[i]) ? 1:0
viota.m    vd, vs2, vm          # vd.m[i] = sum( vs2.m[ all k<i] )
# (group l) data movement (5 instr.)
vrgather{ei16}.vv vd,vs2,vs1,vm # vd[i]=(vs1[i]>=VLMAX)?0:vs2[vs1[i]];
vrgather.XI vd, vs2, ZZ, vm     # splat: vd[i]=(x[rs1]>=VLMAX)?0:vs2[ZZ] (splat)
vcompress.vm vd, vs2, vs1       # vd[i] = vs2[enum(vs1[*])], vs1[*] is a mask
```

C.2 18 Extra Floating-point (requires B.4) - NYI

```
# floating-point MAC, SEW = SEW * SEW +/- SEW (16 instr.)
vf{n}macc.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {-/+} vd[i]
vf{n}msac.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
vf{n}madd.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {-/+} vs2[i]
vf{n}msub.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {+/-} vs2[i]
# floating-point estimations, 7b accuracy (2 instr.)
vfsqrt7.v   vd, vs2, vm         # vd[i] = 1.0 / sqrt( v2[i] )
vfrec7.v    vd, vs2, vm         # vd[i] = 1.0 / v2[i]
```

C.3 8 Extra Double-precision (requires C.2) - NYI

```
# floating-point MAC, widening, 2SEW = SEW*SEW +/- 2SEW (8 instr.)
vfw{n}macc.VF  vd, YY, vs2, vm  # vd[i] = {-/+} (YY * vs2[i]) {-/+} vd[i]
vfw{n}msac.VF vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
```