LEM: A Configurable RISC-V Vector Unit Based on Parameterized Microcode Expander

Zitao Fang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-150
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-150.html

May 19, 2022

LEM: A Configurable RISC-V Vector Unit Based on Parameterized Microcode Expander

by Zitao Fang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Krste Asanovic
Research Advisor

5/19/2022

(Date)

* * * * * * *

[illegible]

Professor Sophia Shao
Second Reader

5/18/2022

(Date)

Contents

Chapter 1

Abstract

The end of Moore's Law has motivated numerous innovations in computer architecture. The traditional approach of increasing frequency and numbers of transistors for general-purpose computation hardware are facing diminishing return, and we must turn to data-level parallelism and specialized hardware accelerator for performance growth. This thesis describes a simple RISC-V vector unit implementation based on a microcode expander. We found that even if we are only using one data path, we still get considerable performance improvement on some benchmark tests over scalar code. We also demonstrate that we can reuse existing hardware to implement custom instructions with minimum hardware overhead by mapping a DSL for accelerator generation onto the microcode expander.

Chapter 2

Introduction

Vector and SIMD instructions are powerful tools for programmers to exploit data-level parallelism. Among the three common types of parallelisms - instruction-level parallelism, data-level parallelism, and thread-level parallelism - data-level parallelism requires the least control complexity to achieve the same speedup. Not all algorithms exhibit data-level parallelism, and even if an algorithm does, programmers still need to rewrite the code for acceleration. However, reducing hardware complexity can lead to less area and less power, which leads to reduced computation costs.

Before vector and SIMD instructions were invented, programmers used loop unrolling to exploit data-level parallelism using the hardware designed for instruction-level parallelism, namely hardware pipelines. Different steps in the loop body are farther away, so we will spend less time waiting with the data hazard interlock.

Vector instructions are not the only way to continuous performance improvement in a post-Moore's Law era. Hardware accelerators built for specific algorithms are another popular type of architecture. Many exploit data-level parallelism to some degree, like 2D arrays for machine learning and accelerators for digital signal processing pipelines. It used to be difficult to find the right processor architecture to integrate and control accelerators, especially for tightly coupled ones. Most ISAs are not open, and their vendors don't provide interfaces for customization. But, the emergence of RISC-V changes this. It's an open instruction set, meaning anyone can use it without any restriction, and the modular design of RISC-V makes integrations of custom instruction extensions easier. RISC-V is the ideal ISA for academic research.

In the ADEPT/SLICE lab, we are currently building a new configurable core, Saturn-V, for DSP applications and hardware accelerator control. As a part of this effort, I built a microcode sequence expander, Loop Expanding Microsequencer (LEM), as an experimental interface for tightly coupled hardware accelerators. I created a RISC-V standard-compliant vector unit on top of LEM to demonstrate its ability to handle complex logic. I also worked with other people in the lab to map a domain-specific language to this program to demonstrate its flexibility for implementing arbitrary algorithms. I will dedicate the remainder of this thesis to these works.

Chapter 3

Background and Prior Works

3.1 RISC-V Instruction Set Architecture

RISC-V is an open instruction set architecture developed at UC Berkeley [1]. Contrary to most other proprietary instruction sets, anyone may build their hardware implementations based on this ISA without seeking licenses and paying royalties to any parties. This makes RISC-V an ideal platform for academic research. There have been many open source hardware implementations of RISC-V, including Rocket core and Berkeley Out-of-Order Machine (BOOM), They are both academic cores developed at UC Berkeley.

The RISC-V ISA has a modular design: the base instruction set only contains the most basic integer instructions necessary to support a minimal core (RV32I or RV64I for 32-bit and 64-bit machines, respectively), and designers are free to choose the ISA extensions to implement on top of it [2]. The most commonly implemented extensions include M (multiplication), A (atomic operation for synchronization), F (single FP), D (double FP), and C (compressed ISA). An architectural profile containing all these extensions, RV64GC, is required by most modern application OS that support RISC-V.

The RISC-V International foundation is in charge of the future development of the instruction set, and new extensions continue to be added into the standard for designers to choose from [3]. Some recent examples include vector (V) extension (which is the focus of this thesis), cryptographic (K), and bit manipulation (B) extension.

3.2 Rocket Core, RoCC Accelerators, and Chipyard

Rocket core is one of the earliest open-source RISC-V cores [4]. It has an in-order pipeline and supports the RV64GC instruction set. The core is written in Scala/Chisel and comes with a simple SoC generator framework, allowing users to easily parameterize the design and embed it in an SoC design.

Rocket chip provides an accelerator interface RoCC, and the designer can specify what instructions the Rocket decode unit should send to the accelerator for execution. Many accelerators designed in the ADEPT/SLICE lab use this accelerator interface to receive instructions from the core so that they can be invoked from regular RISC-V code. Some examples include Gemmini for matrix multiplication and machine learning [5], and Hwacha for vector instructions before V-extension [6].

Besides the RoCC interface, they are often connected to the L2 cache (or the level of cache shared by all cores) to access the memory system directly without using CPU time. RoCC instructions use the four reserved opcode slots for custom instructions in RISC-V and follow R-type (register type) instruction format. This design limits the number of arguments in a single instruction to two, and it only allows "macro"-instructions to be sent, hindering the development of closely coupled accelerators. In the case of Gemmini, one operation is often broken down into multiple configuration instructions, taking multiple cycles just to initiate an operation.

Chipyard is an SoC generator framework based on Rocket with more features [7]. It provides a clean interface for third-party core integration, more IP blocks and bus system support, FireSim FPGA-accelerated simulation flow, and workflow support for ASIC and FPGA. Most projects (including Saturn-V) in the SLICE lab now use Chipyard as their infrastructure.

3.3 Microcode Machine

Microcode has a very long history as a processor control scheme. Although it is slower and bigger than hardcoded control units, it offers more flexibility for hardware design while maintaining ISA compability. Being able to use software compiled for earlier machines is crucial for a machine's success, otherwise it will lose access to the software collection and often the ecosystems of previous machines and deemed too expensive to switch to by users. This architectures can be seen in IBM System/360 series in 1960s and 1970s, where every machine in the series has compatible ISA but very different hardware realization [8]. Microcode bridges these differences and create a uniform software interface for them. Besides, a programmable microcode table allows bug-fixing after product release and helps avoid recalls in some cases, which can be catastrophic like Intel's DIV bug [9]. This design is still used on modern CISC processors (in particular x86) to reduce control and data path complexity.

3.4 Vector Unit

Vector architecture was first popularized by the Cray-I archtiecture and remained as the mainstream architecture for supercomputers for a long time [10]. Cray-I has a iconic loveseat-shape for its extensive cooling system, a modular design that uses only four type of ASIC to build hundreds of logic boards, and new vector architecture that lowers the length threshold for efficient vector operations over scalar instructions. The machine employs vector registers instead of performing memory-to-memory vector operation, whose latency became unacceptable due to the diverging processor and memory speed. The team also proposed the vector chaining mechanism, which can be viewed as register bypassing for vector. With chaining, later vector instructions can be issued with only intermediate results, while other functional units (most likely the memory) may still be computing the remaining output portion of the previous instructions.

Later vector units has more sophiscated design to address memory bottleneck, which is still one of the most important limiting factor for computation. The T0 vector unit, developed in UC Berkeley in 1990s, is a MIPS-based vector unit with 8 vector lanes [11]. The vector unit has separated memory pipeline connected through buses, which allows computations to be overlapped with memory operations for the next loop iteration. T0 issues and executes instructions in order. A later RISC-V based vector unit that predates the vector extension, Hwacha, takes a step further and includes a runahead unit for memory instructions in the frontend and prefetch the data once it detects one [6]. Hwacha's micro-ops expanders (”sequencers”) also implements some forms of out-of-order execution to further eliminate dead cycles.

RISC-V vector instructions set takes a different approach to reduce instructions and register file complexity. There are only 32 registers with implementation-defined length, and user may set the vector element width and vector length to be operated on by setting control registers (CSRs in RISC-V terminology). These options will change vector instructions' behaviors. At most 8 registers can be grouped together and treated as a single register by vector instructions. Any trap during vector execution stop it immediately and save the current element index to a CSR `vstart` for later resume. The vector instruction set uses strip-mining programming model: users execute vector instructions inside a loop, and the loop will "chop off" a segment of input vectors that has been processed by the current iteration, whose length is usually the maximum vector length in the current architecture settings. This process continues until we reach the tail, for which we will execute the same instruction sequence but with a shorter `vl`. This architecture greately reduces the number of instructions due to different vector configurations and simplifies the vector register design.

3.5 Saturn-V Core

Saturn-V core is a configurable core targetting control cores for tightly coupled hardware accelerators and DSP applications. Before the implementation of the vector unit, the core supports RV64GC architecture with an optional B (Bit-manipulation) extension. The default configuration contains two scalar datapaths with only one FPU on the first datapath. All data path may issue scalar memory instructions, but only one of them can be active in a given cycle. Although the frontend is capable of issuing two instructions per cycle, any floating point instructions assigned to the second lane will need to stall until the first lane is cleared so that it can be issued to an FPU. The datapath is classical 5-stage pipeline design, and we can configure the total number of lanes as well as the number of FP lanes.

3.6 Systolic Array Accelerator and Einstein DSL

For operations on 2D data structures, an accelerator based on systolic array is a more efficient way to exploit data-level parallelism than 1D vector units. Such architectures consist of many functional units arranged in 2D matrix.

Gemmini is an example of RISC-V-based systolic array accelerators [5]. The accelerator targets GEMM kernels with simple activation functions and a convolution unroller so that it can be used in machine learning applications. It runs as a separated hardware accelerator, and the only control signal connection is RoCC interface despite being connected to the cache system for data access.

To apply hardware acceleration to more kernel types, the Gemmini group is developing the Einstein DSL, a Scala-based language that describes an algorithm running on 2D systolic arrays. The project is still in its early stages. The target of this project is to generate hardware designs directly from the algorithms described in the DSL, allowing agile design for systolic array accelerators.

Chapter 4

Parameterizable Microcode Sequence Expander

4.1 Overview

The microcode expander was originally a part of the vector implementation, and the early design doesn't allow reuse for other instructions. As the first step toward a high-performance RISC-V vector unit, we implemented a micro-ops unroller for vector instructions on a single scalar datapath so that they can be expanded and executed over multiple cycles using existing scalar datapaths. We soon realized that we could exploit this pattern to implement custom vector operations or even scalar operations that need to be broken down into multiple steps (like cryptographic/K extension of RISC-V), so we extracted the unroller core to exposed its interface for other operations.

The expander contains two counters to track the unrolling progress, and a flag register to indicate whether the expander is busy. During circuit elaboration, the module reads the list of all *decoder extensions* from its config and instantiates them inside the expander. The expander operates as follows:

1. The expander sends a new instruction to all of its decoder extensions, which would indicate whether they recognize the instructions. If none of them does, the expander outputs the fetch bundle as is and repeat this step.

2. If any extension recognizes the instruction, the expander will block the input instruction stream from the frontend. The extension will

start generating and outputting a sequence of control signal bundles. The cycle when new instructions are recognized is called the "prepare phase", and we initialize the pipeline in this step. At the same time, the outer loop counter is reset to the initial value provided by the extension, and the inner loop counter is reset to 0. The busy flag is raised.

3. In the loop, we increment the loop counters and stop when they reach the limit provided by the extension during the prepare phase. We supply the counter values to the extension, which will generate the control signal bundle (micro-op) for the operation. The extension may also branch on the inner loop counter to simulate microcode execution.

The outer loop in this architecture targets different elements in the array, and the inner loop should be used as a micro-pc. In a later version, we will unroll the outer loop and spread them over multiple vector lanes.

4.2 Interface

The `UnrollerExtensionIO` interface is the standard interface through which the microcode expander communicates with the decoder module. It can be inherited to add additional pins (usually for CSR) so that additional signals can pass through the expander layer into the decoder extensions. The decoder extension should generate micro-op bundles based on the information from the expander (counter values, busy flag, etc.), and there is no restriction on how to implement it. In addition, upon receiving a new instruction, the decoder module needs to output the starting and ending counter values, whether the decoder module recognizes the instruction, and whether it is an illegal instruction.

A Saturn-V micro-op bundle is supplied to decode modules as a template, and the decoder modules need to augment the bundle with decoded control signals. The expander will select the bundle from the currently active decoder module and output it to the vector register file. Both the expander and decoder modules have knowledge of Saturn data path and control scheme, therefore

Chapter 5

RISC-V Vector Unit

5.1 Overview

RISC-V vector extension contains over a hundred instructions with diverse functionalities. To handle the complexity, we use the microcode expander to generate control signals for the scalar pipelines so that we can focus on functional units and decode logic. We modified the integer and floating point ALU to support additional operations not required by the scalar instruction set. We need to integrate a microcode expander equipped with vector extension, a vector register file, and a scoreboard for data hazard detection to the data path. We bundle them into a single module so that we can minimize code change to the scalar datapath. Internally, the vector extension is divided into five submodules to handle different groups of instructions. The interface is very similar to the microcode expander level so that users can add custom vector instructions easily. Below is a diagram of Saturn-V data path after integrating the vector unit.

5.2 Vector Register File

The vector register file is a simple flip-flop-based register model that is very inefficient for synthesis in terms of area. It also has large numbers of read and write ports, complicating any future conversion to an SRAM-based design to increase data density and reduce wiring costs. For every vector lane, there are

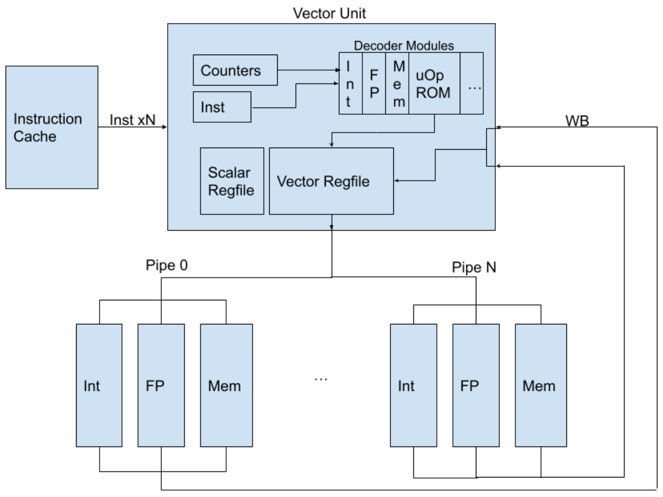


Figure 5.1: High-Level Abstraction of Satun-V Datapath with Vector Unit

three read ports (for FMA) and two write ports (one extra for long-latency op). Therefore, register banking and scheduling must be implemented before the conversion. However, this model provides a basic behavior model for the rest of the vector unit to target, and we will at least have a working model to refer to when we implement the SRAM register file.

The vector regfile control signals and request bundle are defined in `VectorRegfileUOP`. This big bundle can be roughly divided into four parts: read and write request, mask and permutation operation control signals, immediate generate controls and hazard checking signals. We will describe each part below.

5.2.1 Register Data Request

The same register data request format is used by both reads and writes, and it can be divided into two small bundles: `VectorRegReq` and `OperandManip`. The first bundle specifies which vector register and *double word* to read data. he second bundle specifies how to extract data from the double word. This design is useful when we need to extract data from scalar register inputs (in which case the vector request bundle is invalid but we still want to perform some data processing and extraction). Some additional bit manipulation instructions in the cryptographic (K) extension need it, such as packing operations that pack the lowest bytes from several registers into one register [3].

Some vector instructions require one or two operands in some micro-ops to be replaced by some constants: for example, we need to add 1 (before ALU pre-shift) to the memory address in the accumulator for vector unit-stride memory operations. We define a few constants in the vector request bundle to indicate what special constant we want to use for the operand.

It is not feasible to have the same accumulator-based design as the integer data path for FP reduction instruction because FPU is multi-cycle. We need a "reduction queue" for feedback data from FPU so that it can handle multi-cycle latency. We also include an extra bit flag in the vector register request bundle to indicate the use of the FP reduction queue. In the source operand request, a set `reduce` bit means the FPU will need to wait for the feedback data from the queue and then use it to replace the operand. In the destination writeback request, this bit indicates that we need to put the data into the reduction queue so that the next operation can take it.

The index in the vector register request is 64-bit aligned, and the register address in the request bundle may be different from the address in the instruction when `LMUL > 1`. Similarly, the index in the operand manipulation bundle is byte-addressed with alignment based on the element width. To simplify the process of data request generation, we define some functions to fill in the request fields by shifting the element index from the microcode expander and breaking them down.

5.2.2 Write request

For long-latency instructions (mostly memory operations), we need a very compact representation of the register request. The `HellaCache` interface used by Saturn-V only allows us to track a memory request by sending an integer (tag), and we will lose access to the micro-op bundle when the response arrives. Therefore, every vector register write port accepts an "RD tag" that can be decoded into a complete data request inside the vector register file. The vector unit codebase also provides utility functions to generate and decode a tag from a data request and a function to calculate the tag width. All long-latency functional units use this tag format to store write-back requests with their associated data.

5.2.3 Mask Register

The default vector mask register, `v0` is duplicated in our vector register file to reduce the number of read ports for instructions with a mask input. Otherwise, a masked vector fused multiply-add will need four read ports (mask, operand 1-3) per lane. The mask bit is stored in `uop_mask` field inside the micro-op bundle, although this field has a different meaning in later stages (described below). Some instructions use mask inputs other than `v0`, where every micro-op reads one bit at a time (in contrast to mask logical operations, for which we read 64 bits from both input mask registers every cycle). For these instructions, the vector register file can use one of the input registers instead of `v0` to produce `uop_mask`.

Many vector operations produce mask output. This means for every element in the input register, we output one mask bit. The mask bit is stored in `uop_mask` after the execution pipeline stage. Although the vector register can only overwrite an entire byte at a time, we implemented single-bit writeback with a 64-bit register buffer that provides all unused bits in the destination double word. In the mask generation mode, the buffer will be filled with the value in the destination register (1) after each write or (2) at the beginning of the instruction so that the inactive output bit can be preserved. We produce the writeback double word by substituting the bit at the writeback location with the incoming bit. This double word will be written into the flip-flop array as well as the buffer.

Some bit operations in the vector extension have mask register output, but we can generate 64 bits every cycle. The mask logical operations mentioned above are such instructions. However, their output values still need to be masked bit-by-bit if `vm` is false. To support these operations, the vector register file can perform bitwise selection (or merge) on the writeback data and the original value in the destination register element (again, stored in the 64-bit buffer) according to the bit values in `v0`.

5.2.4 Immediate Generation

The immediate generation logic is located inside the vector register file. The current version of vector extension only uses the literal value of `RS1` field as an immediate for Input 1. Most of the instructions require the 5-bit immediate to be sign-extended to `SEW`, then we will either zero or sign extend the `SEW`-bit operand to 64 bits using `OperandManip` logic. Some instructions in the specification do not sign-extend the 5-bit immediate to `SEW`, hence we have an `imm_unsigned` flag in the vector micro-op bundles to mark them.

We also need to generate immedate for `vsetvl` instructions, which has a long immediate encoded in the space after the `RS1` field in the instruction.

The logic for immediate generation is located in the method `create_imm_gen()`, and users may add their own immediate generation logic by extending the class and overriding this function.

5.2.5 Hazard checking

Our vector unit has a simple scoreboard-based hazard checking mechanism. The decoder will set a bitmap `sboard_check` to indicate what vector register the instruction intends to read *or write* during the prepare phase. The scoreboard entry for a particular register is a counter, whose value indicates how many issued operations need to write to the register. We will only allow the prepare micro-ops to proceed if all the scoreboard entries it requests have the value 0 to avoid RAW and WAW (for long-latency operations) hazard.

The scoreboard entries are incremented when a micro-op is issued into the pipeline and start reading the vector register. The counter will be decremented at the writeback stage, or if the micro-op is flushed, immediately at the stage where it becomes invalid.

5.2.6 Permutation Operations in the Register File

The permutation instructions in the vector extension rearrange data in a vector register. These rearrangements are non-trivial to implement, so we provide a set of index registers in the vector register file and a narrow index adder for them. These registers can be used as indices for vector memory requests or directly outputted into the data path as an operand. Data can be loaded into these registers from architectural register reads or the index adder output. The index adder takes inputs from either one of the index registers, the mask bit from the vector register file, or a value supplied by the decoder. Besides, we have an index comparator that takes the same input data type as the index adder, and its output can be used to replace `uop_mask`. All control signals for these components are defined in the register micro-op bundle. We implemented all vector permutation instructions using these components, and users may use them to implement their custom permutation instructions.

5.3 ALU and FPU Modification

In RISC-V vector extension, many instructions require special functional units typically not available in a scalar datapath. This section provides an overview of the extended ALU and FPU components we added to the datapath and the changes to accommodate them.

5.3.1 Integer ALU

We used a two-step approach for integer ALU extension. We want to minimize the changes to the Rocket ALU because it is well-tested and too compact to add complex logic. However, some necessary changes can only be implemented by changing the Rocket ALU directly (like adding more bits to the adder). We modified the vanilla Rocket ALU (into `CustomALU`) to incorporate the following changes:

1. Adder with overflow output (required by add-with-carry family and fixed-point add, signedness of input needed)
2. Shifter with shift-out bits (for rounding)
3. Optionally, rotation support (used in B-extension only)

We then added additional logic to a module that wraps the Rocket ALU (`ExtendedALU`) to implement the following:

1. Fixed point arithmetic for adder and shifter (clipping and rounding).
2. Population count, first-set-bit count, and set-before-first bit operation (`PriorityEncoder(OH)`, with `in3` as the bit mask).
3. Output selection based on a control bit (implemented as a single mux, controlled by `ctrl_bit` mux which also serves as the output mask).
4. Additional Logic to (1) exchange operands, (2) negate Operand 2 after exchange, (3) negate output, and (4) shift Operand 2 by at most 3 bits for address generation.

Vector and bit manipulation extensions have many instructions that share the same functional unit, so we also implement all bit manipulation instructions here. However, due to a lack of test cases, we have to disable all logic used only by the B-extension since they are not verified. Below is a diagram of the logic.

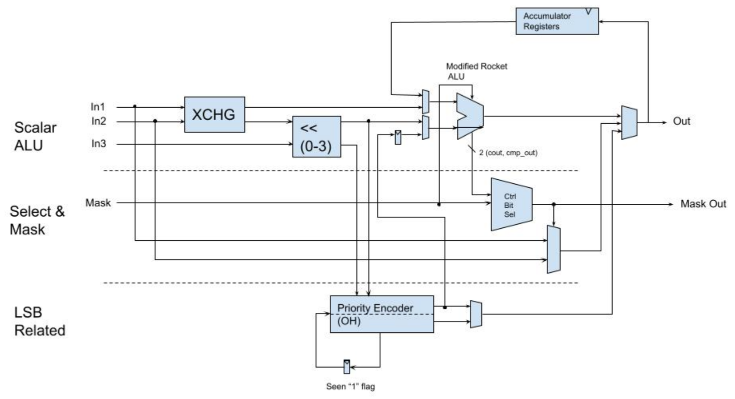


Figure 5.2: Simplified Diagram of Scalar ALU

Accumulator Register and "Seen 1" Flag

To support reduction and stride memory operations, we have "accumulator registers" in the extended ALU. The number of accumulators are configurable. They are controlled by two signals: `use_acc` and `save_acc`, both in `AccumulatorReq`. When `save_acc` is high, we save the current output of the ALU and store it into the specified accumulator register. When `use_acc` is high, Input 1 of the ALU will be replaced by the accumulator value.

To support any bitwise vector instructions that involve the first set bit, we also have a `reg_seen_1` flag register to the ALU so that the ALU knows if it has seen a set bit before. It is often that case that whether the ALU has seen a set bit will affect the instruction behavior.

All the states in the ALU can be replayed: all subsequent pipeline stage will keep a copy of the register values before they are changed when the micro-op is at the execution stage. These values will be recovered to the ALU if a replay occurs.

5.3.2 Integer Multiplier

To support integer fused multiply-add instructions and some fixed-point multiplication instructions, we also modified the Rocket multiplier used in Saturn-V, in the same two-step fashion. We built the multiplier by modifying Rocket's pipelined multiplier to add the third input port for addition after multiplication. The same fixed-point round logic as in the integer ALU is used in the outer module to support fixed-point multiplication instructions.

5.3.3 FPU Enhancement and Reciprocal / Square Root Lookup Table

Compared to the scalar FPU, the FPU in the vector lane requires a feedback queue from its output to its inputs. We need it to implement floating point reduction instructions, where the queued data is used to replace the input with a set FP reduce flag. See the section for vector register requests for more details.

This vector unit also contains two floating point lookup tables (`FPLUT`) to approximate reciprocal and square roots, since we have two such instructions in the vector extension. They accept FP data in Hardfloat's recoded format in different lengths. The lookup table is not working when this thesis is completed, but I will try to fix the problem in a later version.

5.3.4 Common Decode Logic

`VectorUnrollerExtension` contains the common decode logic for all vector instructions. It also serves as the interface between individual vector decode submodule and the expander. It has the following functions:

1. Register overlap checking (some forms of overlaps between source and destination are not allowed)

2. Register group alignment checking

3. Generation of scoreboard bitmap for hazard checking

5.3.5 Decode and Control Logic for Different Vector Instructions

Because the number of instructions we need is large, we created five decode modules for each (self-defined) category of vector instructions:

1. Memory operations
2. Integer arithmetic
3. FP arithmetic
4. Permutation of vector elements and instructions with scalar destination register
5. `vsetvl` handling

They are connected to the common decode logic through an extended version of the interface between the microcode expander and the vector common decode logic, augmented with vector-specific information like input register checking. One may easily define custom vector operations by leveraging the existing infrastructure. To implement an operation, we just need to implement another decode module and add it to the list of modules to be instantiated inside the vector decoder extension.

For vector integer and FP arithmetic instructions, we have a special decode table format to compress the number of columns in the table, coated heavily in syntactic sugar. This is because most instructions only use a very small subset of all control signal fields, and there are so many control fields that a plain decode table for them will be impossible to read.

Our decode table is organized as follows: there are three types of decode table rows, and each row contains a tranformation on the decode bundle, which is a small bundle that contains the decode table output. This bundle is initially all zeros. All table row objects check the input instruction to see if it matches their patterns. If yes, the row object will apply its transformation onto the decode bundle. "Region" rows match the instruction's `funct6` against a `BitPat`, which can be used to match multiple vector instructions close to each others on the instruction listing in the specification. "Line" rows requires exact matches, and they are the only type of rows that will signal a decode table hit. "Subline" rows match `funct5` field (in the location of `rs1` field) beside `funct6`, and it contains a smaller decode table running similar logic to apply transformation. They are most commonly used for unary instructions, since several unary instructions often share the same `funct6` space and are distinguished by their `funct5` values. Besides, every decode table rows will also check the `funct3`, which encodes the operand type information, to determine if it is a match since some combinations of operand types are illegal for some instructions.

Our decode table is organized as follows: there are three types of decode table rows, and each row contains a transformation on the decode bundle, which is a small bundle that contains the decode table output. This bundle is initially all zeros. All table row objects check the input instruction to see if it matches their patterns. If yes, the row object will apply its transformation onto the decode bundle. ”Region” rows match the instruction’s `funct6` against a `BitPat`, which can be used to match multiple vector instructions close to each other on the instruction listing in the specification. ”Line” rows require exact matches, and they are the only type of rows that will signal a decode table hit. ”Subline” rows match `funct5` field (in the location of `rs1` field) beside `funct6`, and it contains a smaller decode table running similar logic to apply the transformation. They are most commonly used for unary instructions since several unary instructions often share the same `funct6` space and are distinguished by their `funct5` values. Besides, every decode table row will also check the `funct3`, which encodes the operand type information, to determine if it is a match because some combinations of operand types are illegal for some instructions.

5.3.6 Vector CSR control

RISC-V vector standard defines a few CSRs. We include a module to handle any changes to their values and supply their values to the vector decoder. Besides, it also (1) decodes and changes `vconfig` for `vsetvl`-family instructions, (2) changes vl and sends terminate signal to the unroller for `vle<eew>ff` (so that the unroller stops issuing instructions without triggering exception), and (3) handles precise trap and set `vstart` accordingly.

Chapter 6

Custom Microcode Operation through Einstein DSL

As a part of the CS252A class project, we explored the feasibility of using the microcode expander to drive complex functional units. One of the potential use cases is to feed data from a scratchpad memory into a systolic array with complex access patterns, as Gemmini control unit currently does.

Due to our time constraints, we could not use the vanilla Einstein DSL for this project since it targets 2D systolic arrays. Instead, we created a modified DSL version for 1D vectors so that we can more easily map it onto our vector decode unit. We believed that we could implement custom operations on the microcode expander and vector unit with minimal overhead, so we implemented a small SpMV kernel using the simplified DSL to demonstrate its feasibility.

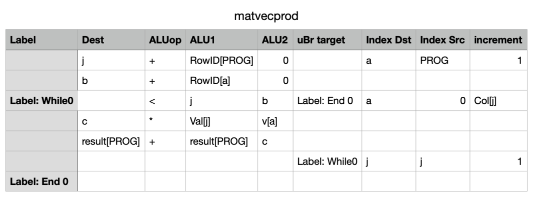
6.1 Simplified DSL Syntax

The simplified DSL uses the SIMT programming model similar to those used in OpenCL and CUDA, although the kernels are executed in series for different elements at this time due to hardware limitations. The language syntax is similar to Python. We believe that this design can also be used in a future implementation with multiple vector lanes.

A kernel defined in this DSL describes a series of scalar operations needed to produce one vector element output. The DSL has a few keywords that the kernel can use to access the counter indices from the microcode expander to determine what vector elements to read and write, similar to `threadIdx` and `blockIdx` in CUDA. The kernel may also define temporary variables, which will be mapped to either accumulator registers or vector index registers depending on their data type. Index registers are shorter registers (with the maximum value smaller than `VLMAX`) that can be used as indices to access vector registers. They are located in the vector register file. The current implementation does not have any restrictions on control structures (if-else and loops) in the kernel. Because we only have one vector lane at this time, all control structures behave the same as their counterparts in software running on a single hardware thread. In future versions, we will apply masking to resolve branch and variable-length loop for multi-thread scenarios; this means we will issue micro-ops for both `if` and `else` at all time and mask away unused branch. A code sample is provided in the following section.

The simplified DSL uses the SIMT programming model similar to those used in OpenCL and CUDA, although the kernels are executed in series for different elements at this time due to hardware limitations. The language syntax is similar to Python. We believe that we can also use this design in a future implementation with multiple vector lanes.

A kernel defined in this DSL describes a series of scalar operations needed to produce one vector element output. The DSL has a few keywords that the kernel can use to access the counter indices from the microcode expander to determine what vector elements to read and write, similar to `threadIdx` and `blockIdx` in CUDA. The kernel may also define temporary variables, which will be mapped to either accumulator registers or vector index registers depending on their data type. Index registers are shorter registers (with the maximum value smaller than `VLMAX`) that can be used as indices to access vector registers. They are located in the vector register file. The current implementation does not have any restrictions regarding control structures (if-else and loops) in the kernel. Because we only have one vector lane at this time, all control structures behave the same as their counterparts in software running on a single hardware thread. In future versions, we will use instruction masking to resolve branch and variable-length loops for multi-thread scenarios; this means we will issue micro-ops for both `if` and `else` at all times and mask away unused branches. A code sample is provided in the following section.



matvecprod

Label	Dest	ALUop	ALU1	ALU2	uBr target	Index Dst	Index Src	increment
	j	+	RowID[PROG]	0		a	PROG	1
	b	+	RowID[a]	0				
Label: While0		<	j	b	Label: End 0	a	0	Col[j]
	c	*	Val[j]	v[a]				
	result[PROG]	+	result[PROG]	c				
					Label: While0	j	j	1
Label: End 0								

Figure 6.1: SpMV Complied Microcode Table

6.2 SpMV Algorithm

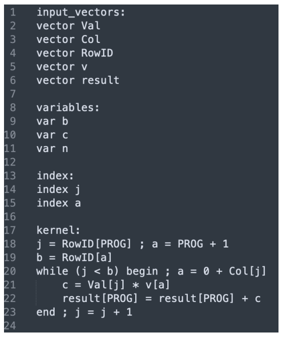
We implemented the standard SpMV algorithm with the DSL, then compiled them into a microcode table. We allow arbitrary microcode branching, so there is no limit on how much time we will spend on one output element. It's the programmer's responsibility to write microcode free of infinite loops.

The custom instruction that invokes the algorithm treats the two input vector registers as two-field structures in the same way as vector segmented memory operations do. The register group (as defined by the current vector configuration) following the one specified by the instruction operand field will also be an input vector. We arrange the parameter fields so that no micro-op needs to take two inputs from the same input register (group). This design limits the input matrix length and width to (`MAXVL` - 1) and the number of nonzeros to `MAXVL` so that all data can be loaded into the register file.

Because we only have limited time for this project, we couldn't build a fully automated build flow from DSL to RTL. The hardware decoder module was manually implemented while referring to the generated microcode table because we had to resolve structural hazards and optimize the microcode. We also needed to manually determine what to include in the decoder output bundle. We hope that a future compilation flow can recognize hardware structural hazards and minimize the decode bundle size when generating decode table entries similar to those described in 5.3.5.

6.3 Hardware Modification

When we implemented the SpMV algorithm, we had to add some hardware structures to Saturn-V to support it. For most algorithms that don't need specialized functional units, the only hardware overhead is the custom encode table and additional accumulator and index registers (configurable to



```
input_vectors:
vector Val
vector Col
vector RowID
vector v
vector result

variables:
var b
var c
var n

index:
index j
index a

kernel:
j = RowID[PROG] ; a = PROG + 1
b = RowID[a]
while (j < b) begin ; a = 0 + Col[j]
    c = Val[j] * v[a]
    result[PROG] = result[PROG] + c
end ; j = j + 1
```

Figure 6.2: SpMV DSL Source

minimize waste). All other hardware components are needed by the standard vector extension. We have to change the RTL architecture to expose more control signals in the vector register file for customization, but these changes also simplified our vector extension. Since they are not on the critical path of our current implementation, we believe that the RTL changes for our DSL implementation do not incur high hardware costs.

Chapter 7

Implementation and Result

The implementation is done on the vanilla version of Saturn-V core. We use the Chipyard SoC generator [7] framework as our infrastructure to reduce debug time and shorten our feedback cycle. To reduce the implementation complexity, we used only one data path to unroll vector instruction while blocking all other lanes if the microcode expander is busy. We modified the scalar data path to insert our vector unit and implement additional operations required by the vector ISA. All works are done in Chisel, and they will be released with a free and open-source license along with the rest of Saturn-V project in the future.

7.1 Assembly Tests Functionality

When we finished this vector unit, the RISC-V vector extension standard had only been ratified for six months. Given the relatively short time since the ratification, there were no assembly test suites accessible to us that provide sufficient coverage for such a big instruction set. Therefore, we decided to use an incomplete test suite provided by the RISC-V International Open-Source lab in China, which is still under development, so that we could at least partially verify the functionality of our vector unit. We also worked with the team and provided feedback and bug reports to improve the test suite. Please note that by the time I finish this thesis, the following cases have not been covered by the test suite:

1. Long `vl` and $EMUL \neq 1$: the vector test cases we used are designed in the same way as scalar tests and focus on the correctness of element-

wise operation. Except for a few instructions where we need multiple elements to check correctness, all test cases hard $EMUL \neq 1$. `vsetvl` also had no tests to check its corner cases.

2. Masked instructions are not covered except for a few where masks are required to check correctness (like `vadc` which uses masks as an ALU input).

3. Cases with vector register overlapping and alignment (when $LMUL > 1$ and for `vlxseg`) were not checked.

4. Nonzero `vstart` values, which means we need to resume operations after an interrupt when executing a vector instruction, are not checked.

5. Tests for instructions that depend on `VLMAX` are not correctly implemented due to a design flaw (not detecting `VLMAX`).

Some of the problems (like masked instructions and long `vl` cases) were partially covered by the benchmark tests. Also, vector FP reciprocal and square-root reciprocal estimation operations along with the fixed point rounding instructions were not working when I finished this thesis due to limited time. I will continue to try to fix these bugs if possible.

7.2 Benchmark Result

FLOP/S (floating point operations per second) is the most important metric for the performance of a vector unit. Although not all vector instructions specified in the ISA are working now, we are still able to run a few benchmark tests that give us some insights into its performance. We select three benchmark tests to run so that we can compare our design against scalar code: DGEMM (matrix size 43), DGEMV (row size 50, column size 100), and conv2d-depthwise (4 channels, 3x3 filter, 56x56 output). These kernels represent the operations with different computational densities and access patterns.

7.2.1 Analysis

Because our current architecture only uses one datapath for vector operation, there is no real parallelism in the vector unit. We expected that the

Table 7.1: Benchmark Performance

	mcycle	minstret	FLOP/cycle
vec-dgemm	136436	124859	1.17
vec-dgemv	36257	20627	0.27
vec-conv2d-depth	816940	658829	0.28
dgemm	274326	273721	0.58
dgemv	45913	30528	0.22
conv2d-depth	1497828	1264443	0.15

performance of these benchmarks to be similar to their scalar version. However, it turns out that the vector performance has around 100% improvement over the scalar version, and in some cases, vector code has significantly denser floating point operations. Since the scalar kernels we use are not exact translations from their vector versions, we may have suboptimal scalar code that causes the "performance improvement". However, it is equally likely that the vector unit does eliminate the dead cycles caused by data and control hazards by unrolling loops more aggresively.

7.3 Preliminary Synthesis Result

We synthesized the Saturn-V core with the vector unit using Hammer VLSI toolchain [12], which itself is a part of the Chipyard SoC framework. The target technology is ASAP7, and our frequency goal is 1GHz. The frequency goal hasn't been met, and the current critical path has 1533ns latency. Most delays are due to the long logical paths in the ALU and can be reduced by rearranging ALU components. Below is a table of all major modules and their areas when we have the following configuration: 2-issue, 64 bits, 1 FPU lane, RV64GCV with 256 bits vector length, and custom DSL support.

The vector register file is a flip-flop-based design and is not practical for place-and-route. In the current post-synthesis design, the vector register file occupies 1/3 of the total area.

Besides replacing flip-flops with SRAM, we can also optimize the area by changing the current decode table organization by allowing unused decode bundle fields to be any values instead of 0.

Table 7.2: Selected Post-Synthesis Area for Vector Unit (area in μ^2)

Instance	Cells	Cell Area	Net Area	Total Area
core	174246	331929	149596	481525
-vector_unit	59530	119647	48166	167813
–regfile	44830	101741	41337	143078
–unroller	10354	11986	4109	16095
—VectorUnrollerExtension	8836	9233	3390	12623
—-VecDecPermutationModule	1334	1507	612	2119
—-VecDecIntegerModule	1199	1427	558	1985
—-VecDecFPModule	1163	1433	535	1968
—-VecDecMemoryModule	941	976	321	1297
—-VecDecVSETVLModule	508	419	22	441
–vsboard	3501	5324	2599	7923
-alu	5231	7562	4233	11795
–scalarALU	2174	2895	1636	4531

7.4 DSL Synthesis and Performance Result

We ran synthesis and some benchmark tests for the DSL implementation. The synthesis results and benchmark performance data are listed below. The custom decoder module is not on the critical path. Most of the area increase is due to the additional index and accumulator registers. The performance of DSL implementation is worse than their scalar code, mostly because the DSL algorithm must load the entire 4x16 double word memory region into the vector registers even if some of them are not used. We can observe this from the constant latency for different input sizes. We believe that setting `vl` before loading every field could eliminate the extra memory access and bring the total latency down and closer to the scalar version.

Partition	**Area** (um^2)
Decoder	850
Saturn-V Core	492072
Saturn-V Core (original)	481525

Table 7.3: Area Breakdown

Matrix Size/Input Non-Zeros	*uOp* Expander (cycles)	Scalar (cycles)
12x12/15 nnz	2156	948
14x14/10 nnz	2229	902
12x4/7 nnz	2107	739
5x13/11 nnz	2216	593

Table 7.4: Custom vs Scalar Performance Comparison

Chapter 8

Future Work

Saturn-V is an ongoing project, and LEM is only a prototype now. It can be used as an RTL model for vector units at this time, but it can do more if we keep working on it.

Vector Unit with Multiple Lane and Compact Register File The vector unit can only issue microcode bundles to a single datapath to simplify initial implementation, but a vector unit, by design, should exploit data-level parallelism and allows multiple micro-ops to be issued at the same time. The easiest way to do this is to add multiple ports to the expander and the vector register file, duplicating the entire micro-op bundle except for register requests, since different vector lanes should work on different data streams.

As mentioned in previous sections, the vector register file is extremely large and occupies almost half of the entire core area. To build a synthesizable vector unit for ASIC, we must replace flip-flops with SRAM and implement scheduling logic to address the extra latency and structural hazards.

Systolic Array and Complex Functional Unit Control The LEM microcode expander can be used to issue control signal sequences to complex functional units. This is particularly useful for systolic arrays since the LEM expander is designed to handle parallel data streams. By augmenting the Einstein DSL microcode compiler, we should be able to describe a systolic array with custom interconnects for non-GEMM 2D algorithms and utilize the LEM infrastructure to control the sequence generation, as we have shown in the vector DSL implementation.

Implementing Other RISC-V Extension Currently, only the vector extension is implemented on LEM. Although the functional units for bit-manipulation extension are already in place, the decoder unit is not, and the arithmetic logic is not tested. Implementing a fully-functional bit manipulation unit would not take long.

Also, although not yet vectorized, the cryptographic extension has a lot of instructions that can be divided into multiple steps. We should be able to build a very small cryptographic accelerator by reusing LEM infrastructure. Even if we can build a faster accelerator with a compact, decoupled design, the LEM-based cryptographic implementation can still serve as a reference model and help verifying the compact unit.

Chapter 9

Conclusion

There is no more free lunch after the end of Moore's Law. We can no longer scale up the frequency and numbers of transistors for the same architecture and expect to get as much performance improvement as we did in the past. We have to explore new hardware architectures, particularly for algorithm acceleration and parallel hardware. The Saturn-V core and LEM microcode expander is the first step for a RISC-V vector unit and also for closely coupled hardware accelerators. Although it is only a functional model now, LEM still proves the capability and flexibility of the microcode expander architecture as a building block for accelerators and vector units.

Chapter 10

Acknowledgement

The Saturn-V project is led by Prof. Krste Asanovic, who is my advisor and gave me a lot of advice on architecture design for this project. Jerry Zhao, Albert Ou, and Daniel Grubb work on RTL and vector kernel design, and they provided feedback for the design and helped me integrate the vector unit into Saturn-V. Max Banister and Charles Hong worked with me to explore OS support for heterogeneous cores (since Saturn-V SoC will be a multi-core heterogeneous system), which influenced some of my design decisions for the vector unit.

Prashanth Ganesh and Hasan Genc, both advised by Prof. Sophia Shao, worked on the Einstein DSL project and helped me map the language onto LEM. Prof. John Wawrzynek was teaching the CS 252A class where we developed the DSL mapping project, and he also gave us feedback and comments on the direction of our project.

Shenwei Hu and other members in the RIOS lab provided an assembly test suite for this project, which was extremely helpful and significantly shortened the time to test the vector unit.

Bibliography

[1] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[3] *Specification - risc-v international*, `https://riscv.org/technical/specifications/`, Accessed: 2022-05-16.

[4] K. Asanović, R. Avizienis, J. Bachrach, *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[5] H. Genc, A. Haj-Ali, V. Iyer, *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv preprint arXiv:1911.09925*, vol. 3, p. 25, 2019.

[6] Y. Lee, "Decoupled vector-fetch architecture with a scalarizing compiler," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2016. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-117.html`.

[7] A. Amid, D. Biancolin, A. Gonzalez, *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020, ISSN: 1937-4143. DOI: `10.1109/MM.2020.2996616`.

[8] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the ibm system/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87–101, 1964.

[9] D. Price, “Pentium fdiv flaw-lessons learned,” *IEEE Micro*, vol. 15, no. 2, pp. 86–88, 1995.

[10] R. M. Russell, “The cray-1 computer system,” *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[11] K. Asanovic, *Vector microprocessors.* University of California, Berkeley, 1998.

[12] E. Wang, “Hammer: A platform for agile physical design,” M.S. thesis, EECS Department, University of California, Berkeley, 2020. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-28.html`.