Assessing RISC-V Vector Extension for Machine Learning

Master's thesis in Embedded Electronic System Design

Johan Hellström and Marwan Ghamlouch

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Assessing RISC-V Vector Extension for Machine Learning

Johan Hellström and Marwan Ghamlouch

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of omputer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Assessing RISC-V Vector Extension for Machine Learning
Johan Hellström and Marwan Ghamlouch

Supervisor: Per Larsson-Edefors, CSE
Advisors: Göran Bilski & Tryggve Mathiesen, AMD
Examiner: Lena Peterson, CSE

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: AI generated image of an FPGA circuit board

Typeset in LaTeX
Gothenburg, Sweden 2023

Assessing RISC-V Vector Extension for Machine Learning
Johan Hellström and Marwan Ghamlouch
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This report presents a partial design and implementation of a soft RISC-V vector extension on a field-programmable gate array (FPGA) based on the most recent and ratified specification (v1.0), with the aim to investigate the suitability of RISC-V vector processor extensions for machine learning applications.

The results were obtained by creating a matrix multiplication benchmarking program compiled in GCC and modifying configurations that altered the behavior of the designed prototype. The configurations that could be altered were vector length and whether or not forwarding from the execute stage was enabled. We also implemented our design in a synthesis tool (Vivado) in order to estimate resource usage, power consumption and timing.

From our prototype we were able to find that we could, for our benchmarking program, improve the performance by up to 5.4 relative to a scalar RISC-V processor, but at the cost of a notable resource usage and power increase.

In conclusion, we believe that vector extension is suitable for machine learning applications because of the achievable performance increase, however the design should be heavily optimized to reduce the resource utilization to capitalize on this.

Keywords: RISC-V, ISA, ISA extension, vector, processor, machine learning.

Acknowledgments

This thesis was a huge undertaking that could not have been done without the continuous and thorough support of others. We can not thank enough our academic supervisor Per Larsson-Edefors for helping us set and maintain realistic ambitions through out our work, and our industry supervisors Göran Bilski and Tryggve Mathiesen (and the AMD office at large!) for providing invaluable technical guidance every step of the way to accomplish this thesis.

Marwan would like to extend his gratitude to his family for being a great source of inspiration for what can be achieved in life, and for all the work and effort they spent to help me be in a position to take on this opportunity. He would like to also thank his Olof friends for their continuous support, and his best friend for always being by his side and supporting him through this journey.

Johan would like to also thank his family and friends for helping him with various issues and pushing him to achieve his goals. He would also like to thank them for giving feedback on several aspects of the thesis.

Johan Hellström and Marwan Ghamlouch, Gothenburg, 2023-06-26

Contents

Glossary **xi**

1 Introduction **1**
1.1 Background . . . 1
1.2 Gap . . . 2
1.3 Problem Statement . . . 2
1.4 Limitations . . . 2
1.5 Thesis Outline . . . 3

2 Theory **5**
2.1 Field-Programmable Gate Arrays (FPGAs) . . . 5
2.2 Vector Instructions . . . 5
2.2.1 Vector Registers . . . 6
2.2.2 Vector Lanes . . . 6
2.2.3 Vector Masking . . . 6
2.3 Machine Learning . . . 7
2.3.1 Matrix Multiplication . . . 7
2.3.2 Convolution . . . 8
2.3.3 ReLU Activation Function . . . 8
2.4 Digital Signal Processing . . . 9
2.5 Multiplication in Hardware . . . 10
2.5.1 Wallace Tree . . . 10
2.5.2 Multiplier Grouping . . . 12
2.6 Pipeline Hazards . . . 12
2.6.1 Forwarding . . . 12
2.7 Processor Performance Metrics . . . 13

3 Methods **15**
3.1 Overview . . . 15
3.2 Materials . . . 16
3.3 Benchmarks . . . 16
3.4 Testbench . . . 17

4 Prototype **19**
4.1 Overview . . . 19
4.2 Pipeline . . . 20

4.2.1 OF Stage . . . 20
4.2.2 EX Stage . . . 21
4.2.3 MEM Stage . . . 22
4.2.4 WB Stage . . . 22
4.2.5 Synchronization . . . 23
4.3 Scalar Input Expander . . . 24
4.4 Arithmetical and Logic Unit (ALU) . . . 25
4.4.1 Zero/Sign Extender . . . 25
4.4.2 Adder/Subtractor and Comparator . . . 25
4.4.3 Multiplier . . . 26
4.4.4 Shifter . . . 28
4.4.5 Intra-lane Reduction . . . 28
4.5 Memory Management . . . 29
4.6 Lanes . . . 33
4.7 Hazard Detection . . . 33
4.7.1 Forwarding . . . 34
4.8 Masking . . . 35

5 Results **37**
5.1 CPI and Speedup . . . 37
5.2 Hardware Utilization . . . 40

6 Conclusion **43**
6.1 Challenges . . . 43
6.2 Future Work . . . 44

Bibliography **45**

A Testbench Programs **I**
A.1 Matrix Multiplication . . . I

Glossary

AI Intelligence exhibited by a machine (something artificial). 1

CPI A quotient between the number of cycles and the number of instructions. 13, 37, 38, 39

CSR Special registers containing control and status information. 21, 22, 36

DSP Process a digital signal. 5, 9, 41

EEW A segment of load and store instructions which determines the element width of the data. 29, 30, 31

EX A pipeline stage used to execute the instruction currently in progress. 20, 21, 22, 23, 35, 37, 38, 40, 42

FPGA An integrated circuit that can be reconfigured. In this project, it is used to implement a soft processor. 1, 2, 5, 9, 16, 26, 37

HDL A type of language where the user describes the behavior of the hardware. 5

ISA An abstract model of a computer that defines the design such as instructions and capabilities. 1, 6

LSB The binary 1s place of an integer. 24

LUT A hardware component used to implement arbitrary functions. 5, 9, 41

MEM A pipeline stage used to load from and store to main memory. In this project, multi-cycle instructions iterate in this stage until they have completed. 20, 22, 23

ML Methods to make machines learn by the means of providing test data for the machine to train on, in order to find trends. 1, 7

mux A hardware component used to push one of many inputs to the output. 22, 24, 25, 28, 30, 32, 33

OF A pipeline stage used to fetch operands from memory. Also commonly referred to as instruction decode (ID). 20, 22, 23, 34

RAW A type of data hazard where data is read after writing, but has not yet been written to the register file before it is supposed to be read. 12, 33, 34

RTL A design abstraction modeling how signals are connected between registers. 33

RVV Vector extension specified for the RISC-V ISA. 2, 6, 16, 19, 20, 21, 29, 34, 35, 36, 40, 41, 43, 44

SEW The current element width, stored in a register. 24, 25, 26, 28, 30, 31, 32

SIMD Computer architecture design where a single instruction can instruct the CPU to perform an operation on multiple data. 1, 6

SISD Computer architecture design where a single instruction performs an operation on a single piece of data. 5

VHDL A hardware description language. 5, 35

VL The number of vector elements to be updated by an instruction. 32, 36

VLEN The length of a vector register. 20, 33, 37, 38, 39, 40, 42, 43, 44

VRF Register file containing all vector registers. 20, 22, 23, 30, 32, 36, 44

WAR A type of data hazard where data is written after reading. Not a concern in in-order execution pipelines. 12, 33

WAW A type of data hazard where data is written after writing. Not a concern in in-order execution pipelines. 12, 33

WB A pipeline stage used to write data back to a register file. 20, 22, 44

XRF Register file containing all scalar registers. 20, 22

1

Introduction

Over the last decades, the world has greatly benefited from the ubiquity of performant and scalable computing. However, with the boom in artificial intelligence (AI) and machine learning (ML) [1], a demand has emerged for even more performant computers and energy efficiency in those two domains. As the marginal performance gain from CPUs and GPUs is decreasing [2], industry experts believe the next wave of computing will be domain-specific computing [3], with a focus on computer architecture being tailored to the intended application [4]. Hence, with the growing need for innovating at the computer-architecture level, more attention is being given to a recent open-source instruction set architecture (ISA) named RISC-V (pronounced *risk-five*) [5]. The reason behind this is the low barrier of entry with a small core set of instructions and a permissive license. Furthermore, RISC-V endorses a modular approach where the standard base ISA is small but more functionality can be added through extensions.

Given that AI and ML heavily rely on single instruction, multiple data (SIMD) operations, vector processor extensions are often utilized to run such applications. Therefore, for our thesis project we aim to investigate the suitability of RISC-V vector processor extensions for machine learning, through partial design and implementation of a soft RISC-V vector extension on a field-programmable gate array (FPGA), and based on the most recent RISC-V specification (v1.0) [6].

1.1 Background

There are multiple prior implementations of soft RISC-V vector-extended processors, each however has a set of trade-offs the developers had to make [7]–[11]. Certain trade-offs are due to the nature of FPGAs having limited resources, however most have been adapted to their intended application. For instance, in the work presented in [7], only a subset of the vector ISA was implemented, and the complexity of the design was reduced by integrating vector functionality into the processor itself (as opposed to having a vector co-processor paired with a scalar processor, saving on interfaces between the two). These changes were made as the design was meant to eventually target micro-controllers.

In another work presented in [8], a systolic array-based vector unit was developed and four custom instructions were added. The developers made these design changes in an effort to improve ML inference on edge devices.

In [9], a RISC-V vector processor for acceleration of machine learning algorithms was developed. The developers introduced a vector memory subsystem consisting of a control unit and an array of buffers, which they used as an intermediate between the vector lanes and the AXI Full master controller in order to not stall the vector as data is being loaded. This hints to the frequent memory accesses anticipated in machine-learning algorithms.

The developers of a RISC-V based SIMD co-processor [10], added an auxiliary processing unit (APU) interface as they intend their design to be embedded into existing system-on-chip (SoC) configurations.

In [11] a vector extension with speculative execution was implemented, with the goal of integrating the design as a functional unit in a scalar core.

1.2 Gap

As discussed above, multiple implementations of RISC-V vector extension (RVV) have been created. However, to our knowledge, no comparisons between the implementations have been made. For that reason, it is of interest to investigate how different implementations, as well as our own, compare. Since the RVV specification has been ratified fairly recently, few comparisons between a non-RVV and an RVV implementation exist for the latest specification.

In this project we are aiming to add RVV support to an existing RISC-V core provided to us by AMD, and that has no comparisons with RVV added to it.

Furthermore, no conclusions regarding the suitability of vector extensions for machine learning purposes have been drawn in previous work. The aim of this thesis is to provide such conclusions.

1.3 Problem Statement

The aim of this thesis is to assess RVV's suitability for machine learning, which we aim to do through implementing a RISC-V vector-extended soft processor on an FPGA. The implemented soft processor is created based on version 1.0 of the RVV specification [6] and tested using the metrics described in Section 3.3.

1.4 Limitations

Because of the time limit of the thesis project, we must impose certain limitations on what avenues we can explore. For instance, the floating-point vector instructions would take considerable time to implement, but would not give any new insights of the suitability of vector extension with regards to machine learning. Instead, we will create benchmarking tools written in C and compiled with GCC [12]. By doing this it will be possible to see what instructions we would be required to implement in order to test our design, thus reducing the workload substantially.

Another limitation we made was to only support the set of instructions that our benchmarking programs required (see Section 3.3). This meant that, for instance, neither division nor register grouping was supported.

Besides the limitations set at the beginning of the project, additional limitations were later added due to the limited time we had. Notably, instead of testing several testbenching programs to evaluate the performance of our prototype we narrowed our focus to just one program: matrix multiplication. This meant that we would get fewer results, but since we would still be able to produce relevant results we deemed this to be a reasonable additional limitation.

1.5 Thesis Outline

In Chapter 2 all relevant background knowledge necessary to follow the thesis is presented. Chapters 3 and 4 explain how the thesis was carried out, the problems we faced and how we solved them. The results are shown in Chapter 5, and finally conclusions are drawn in Chapter 6.

2

Theory

In this chapter, necessary theory behind concepts used or discussed in the following chapters are introduced. First, we introduce FPGAs in Section 2.1, the concept of vector instructions in Section 2.2, and machine learning and some of its internal operations in Section 2.3. In Section 2.4 we introduce digital signal processing, multiplication in hardware in Section 2.5. Finally, we discuss pipeline hazards in Section 2.6 and processor metrics in Section 2.7. The reader is expected to have basic knowledge about computer architecture (most notably pipelining), as can be gained from [13].

2.1 Field-Programmable Gate Arrays (FPGAs)

An FPGA is an integrated circuit that can be configured by an end-user. They are commonly used to implement what is called soft-processors which allow the end-user to reiterate and improve their design in short time period and at a low-cost, as the circuit layout and manufacturing is taken care of, compared to implementing a hard-processor that could contain hardware bugs with no way of fixing it after manufacturing, thus making the entire processor virtually unusable.

The FPGA can be reconfigured by uploading a new bit-stream containing the configuration options for the components on the board, such as look-up tables (LUTs), digital signal processing (DSP) blocks and registers. The bitstream is usually generated by vendor-specific tools, for instance Vivado [14] which is used in this project, and described in hardware description languages (HDLs). The HDL used in this project is very high speed integrated circuit program hardware description language (VHDL).

2.2 Vector Instructions

Typical scalar processors apply one instruction to only one piece of data, which is referred to as single instruction, single data (SISD). This reduces a processor's resource usage (such as area) as the processor prioritizes accommodating different types of operations and executing them in a reasonable amount of time. Because of this ability to handle different types of operation, there is an associated decoding overhead with every instruction, as the processor determines which operands to read and what operation needs to happen. Although the decoding overhead is not

significant per instruction, it quickly adds up if the programs running (discussed further in Section 2.3) require repeating the same operation on multiple pieces of data.

The existence of programs that run repetitive pieces of code on multiple data has led to the development of vector processors that aim to operate on vectors of data rather on just single pieces of data, so an instruction is decoded just once for handling multiple pieces of data. With that aim in mind, vector processors also introduce some other features to speed up operating on multiple pieces of data such as parallelization with vector lanes and handling larger operands with vector registers.

However, not all code is suitable for vector processors and even in highly vectorizable code and applications there is almost always some scalar code that needs to run, so it is reasonable to have a scalar processor that can offload vectorizable applications to a vector processor whenever needed. In that regard it is common to have vector extensions which assist scalar processors and expand their ISA to have more SIMD or vector instructions, such as RVV.

2.2.1 Vector Registers

One feature of using a vector processor is that the registers that hold the operands for vector operations are extended to hold more bits and fit more elements, for instance, a 128-bit vector register could hold 16 8-bit elements or eight 16-bit elements. This gives us the advantage of reducing the program size as one instruction can now address a larger number of elements, and also reduces the decoding overhead per piece of data.

2.2.2 Vector Lanes

Another advantage offered by vector processors is parallelizing operations. This is achieved by creating vector lanes which are almost identical pipelines running in parallel, with each lane being fed only certain elements from the vector register. This technique works for operations that handle independent data elements, but not for operations that operate on elements within the same vector.

2.2.3 Vector Masking

It is common to operate on multiple pieces of data, but we often also want to take action on data elements based on some condition. Vector processors utilize vector masks to determine what elements of a vector register to modify or leave untouched, where the vector masks themselves can also be a result of some vector operation. For example, the ReLu function mentioned in Section 2.3.3 is described as $y = max(x, 0)$. So we can first create a mask by comparing all elements of a vector register `X` to 0, if the element is less than 0, we set the corresponding mask register bit to 0, else we set it to 1 - this can be done with one instruction (`vmslt`). Afterwards we can use the newly created mask along with a move function to move `r0` (a scalar register permanently set to the value 0) into a vector register `X`, where now only elements

with a corresponding mask bit of 0 will be set to 0, while other elements will remain untouched.

2.3 Machine Learning

Machine learning (ML) is a class of programs where a computer is not explicitly programmed, but is instead 'trained' on large data sets to produce a desired outcome. During training, pieces of data are put through the computer program meanwhile a set of weights are tweaked with every piece of data until the program produces the desired outcome. After the program has been 'trained', the weights obtained during training are fixed and are used when running the program again on new pieces of data. The process of using pre-computed weights on new pieces of data is referred to as inference.

During inference, the program might apply the weights on the input data through different operations, with the more common operations being matrix multiplication, convolution [15] and activation functions.

2.3.1 Matrix Multiplication

Matrix multiplication is straightforward to perform; a row of the first matrix is multiplied piece-wise with every column in the second matrix, with all the products resulting from one row-column multiplication accumulated together to form one new element in the output matrix. The i^{th} row with j^{th} column produce the element in the i^{th} row and j^{th} column in the output matrix.

Listing 2.1 shows the code for a 2D matrix multiplication in C.

```
void mat_mul(int * a, int m1, int n1, int * b, int m2,
             int n2, int * result) {
 for (int i = 0; i < m1; i++) {
   for (int j = 0; j < n2; j++) {
     for (int k = 0; k < m2; k++) {
       result[i*n2 + j] += a[i*n1 + k] * b[j + k*m2];
     }
   }
 }
}
```

Listing 2.1: 2D matrix multiplication implementation in C.

However, we note that the memory access pattern of matrix b in the innermost for loop matrix is not contiguous if the matrix was stored in a row major form. This problem can be overcome if we transpose the second matrix when storing it. The code for this is shown in Listing 2.2.

This makes the memory access pattern for both matrices mostly contiguous and reduces the cost of this operation. More importantly however, the multiply and

```
void mat_mul_transpose_b(int * a, int m1, int n1, int * b, int m2,
                         int n2, int * result) {
  // a is a m1xn1 matrix
  // b is a transpose of a m2xn2 matrix
  // (so m2xn2 are the dimensions of b _before_ the transpose)
  for (int i = 0; i < m1; i++) {
    for (int j = 0; j < n2; j++) {
      for (int k = 0; k < m2; k++) {
        result[i*n2 + j] = a[i*n1 + k] * b[j*m2 + k];
      }
    }
  }
}
```

Listing 2.2: 2D matrix multiplication with the second matrix transposed.

accumulate line of code is now more suitable for vectorization as, for example, an entire row and an entire column could be loaded with one instruction each, and then operated on together with a multiply instruction (`vmul`) followed by reduction sum (`redsum`).

2.3.2 Convolution

The convolution operation is similar to matrix multiplication however all the elements of the first matrix, often called a filter or a kernel, are element-wise multiplied with an equally sized sub-matrix of the second matrix. After that all the products are accumulated and result in one new element in the output matrix. Figure 2.1 shows the algorithm calculating the second output element using a 3×3 filter.

In a scalar processor this would likely require a load, multiply, and add instruction for each pair of elements in the two matrices, while in a vector processor, a load and a multiply-accumulate (one instruction) would be needed for every n elements (where n would depend on the size of the vector register).

2.3.3 ReLU Activation Function

In neural networks - a type of machine learning model - it is common to have a activation functions that introduce non-linearity to the network, with one common activation function being the ReLU function that has a simple operation of $y = max(x, 0)$.

In a scalar processor, we would likely have to load, compare and then set or store the value of y (which could use a branch jump) for each element, however in a vector processor we can use vector masks to ascertain which elements to set to 0 and which to keep untouched. Hence we are able to handle conditional logic for multiple elements at a time before operating on them.

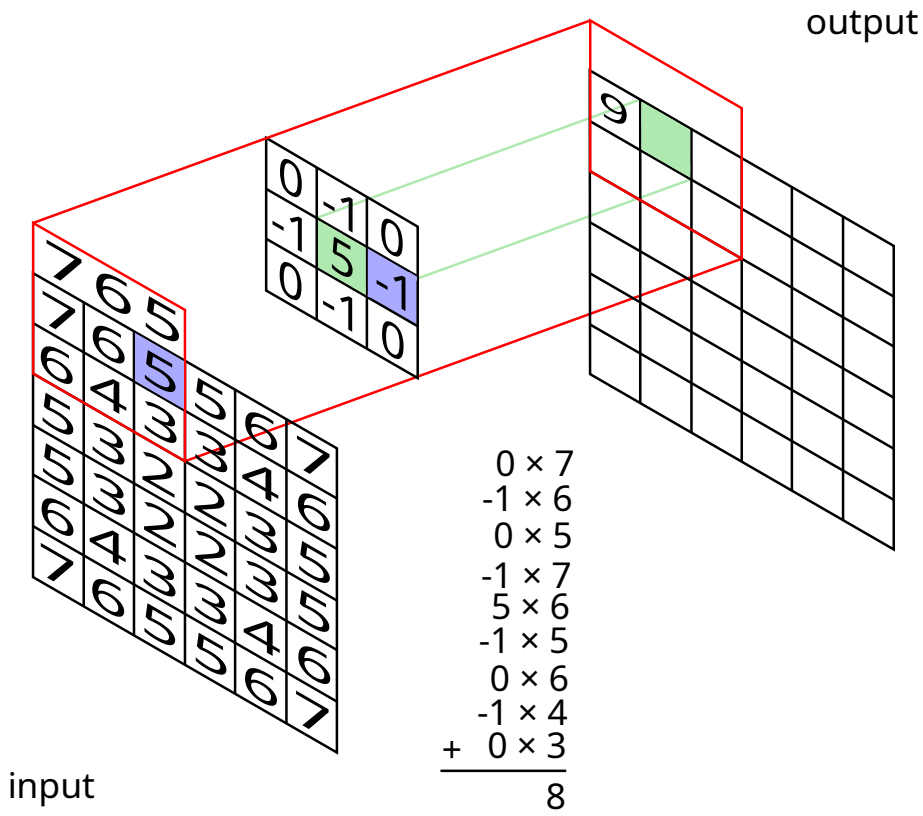


Figure 2.1: A diagram showing the computation of one element in 2D Convolution from [16].

2.4 Digital Signal Processing

DSP is the concept of processing a signal, usually by applying a filter to the signal [17]. On an FPGA board, DSP blocks can be used to implement both filters as well as arithmetical operations. Using these blocks to implement arithmetical operations is of interest, since implementing the same arithmetical operations using LUTs will have a larger delay and be more resource demanding. On the development board used, we have a DSP block called DSP48 [18], where 48 refers to the input bit-width and its block diagram illustrates how it can be used to implement arithmetical functions (see Figure 2.2).

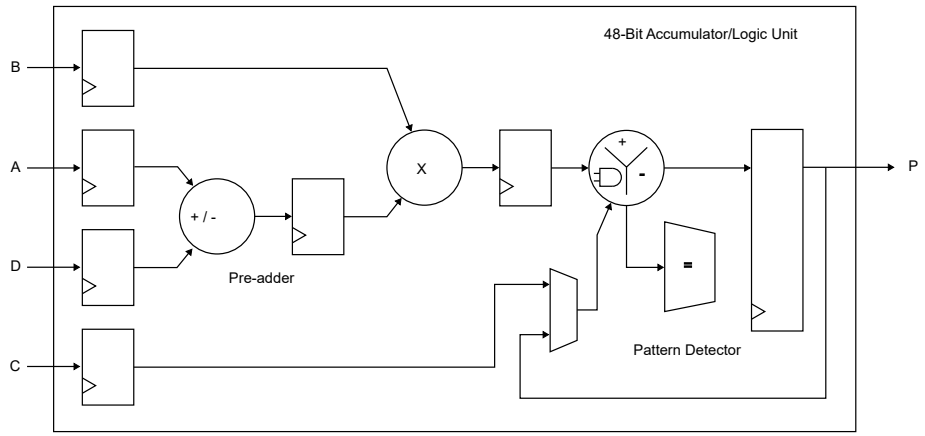


Figure 2.2: DSP48 block, as shown in [18].

2.5 Multiplication in Hardware

To multiply two binary numbers in hardware one could naively add a number to itself a 'multiplicand' number of times, but a number of more efficient algorithms exist such as Wallace trees [19], and Dadda multipliers [20]. In this section we will discuss one of these algorithms (Wallace tree), but more importantly we will also discuss how smaller multipliers can be grouped together to handle larger operand sizes.

2.5.1 Wallace Tree

In this method, two numbers are multiplied by first multiplying each bit of the multiplier with each bits of the multiplicand resulting in n^2 1-bit partial products. Note that a 1 bit multiplier can be built using just one AND gate. Each of the produced partial products has a weight associated with it depending on the weight of its factors, so the weight of a partial product resulting from multiplying a_m and b_n is:

$$a_m \times b_n = 2^{(n+m)} \tag{2.1}$$

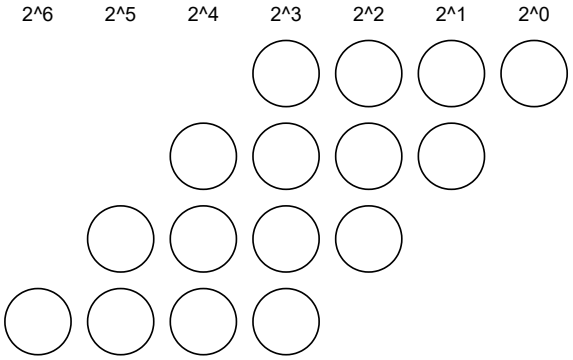


Figure 2.3: A Wallace tree showing the partial products of a 4-bit multiplication with their respective weights indicated at the top.

Now we start adding all the partial products with the same weight together (a reduction step), either using full adders or half adders depending on the number of partial products available in every weight, with the goal of adding as many partial products within the same weight at the same time. Shown in Figure 2.4a is the first reduction in 4-bit multiplication, where we try to cover as many partial products (white dots) as we can with adders (highlighted in red). In the second reduction (Figure 2.4b) we can see the sums of additions from the first reduction (colored yellow) as well as carry-outs from lower weights (colored green).

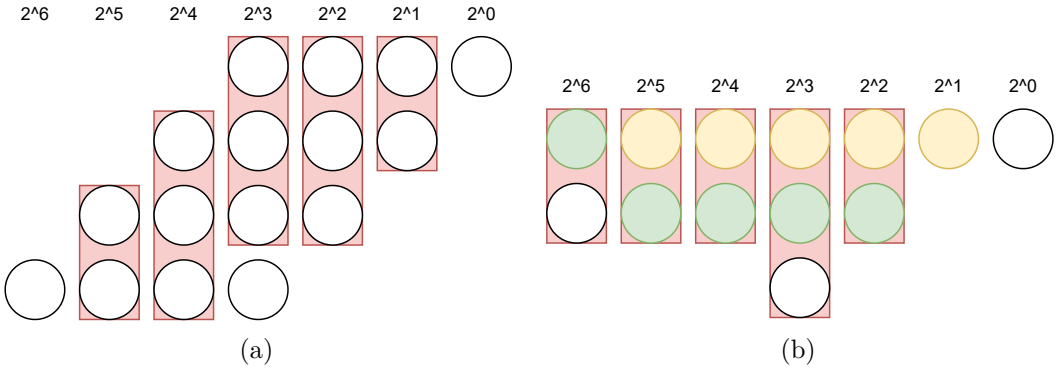


Figure 2.4: Reduction steps of 4-bit multiplication using a Wallace tree: (a) shows the first reduction step. As many partial products (white dots) are covered as possible with full- and half-adders (highlighted in red). (b) shows the second reduction step. The yellow dots are sums while the green dots are carry-outs of full-adders from the first reduction step.

We repeat this reduction step as long as any of the weights has 3 or more partial products, so at the end we have just two partial products in every weight, which is effectively just two numbers we will add together with a conventional adder (empty bits can be filled in with zero). The result of this adder will be the product of the multiplication.

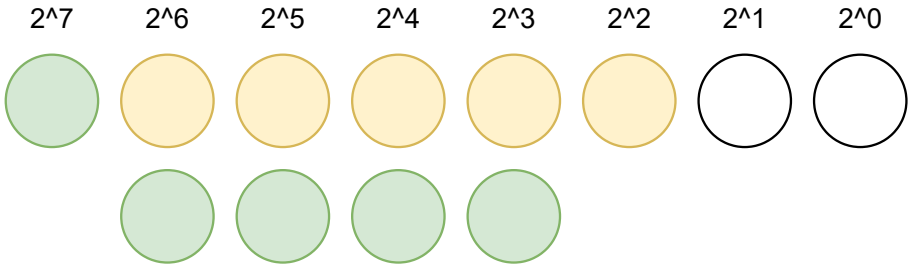


Figure 2.5: The result of the final reduction for a 4-bit multiplication using a Wallace tree. There is just two numbers left (the top and bottom row of bits).

Considering the time needed to execute this algorithm we note that all the additions in a reduction step can happen in parallel (constant time), and we only need $\log_2(n)$ reduction steps and one final addition to get our final result. This is a great improvement over the naive approach as it reduces the number of additions but more importantly it has a constant time on each reduction step.

2.5.2 Multiplier Grouping

In the work demonstrated in [21] the authors demonstrate a twin-precision multiplier that can perform a n-bit multiplication or $n/2$-bit multiplication, drawing from the fact that partial products of a n-bit multiplier contain $n/2$-bit multiplication products. A mathematical breakdown of a n-bit multiplication into smaller multiplication operations is very briefly shown in [22]. Drawing from that, we provide a generalized relation in Equation (2.2) that shows the decomposition of a n-bit multiplication into smaller m-bit operations where $n > m$:

$$\begin{aligned} A \times B &= (A_{n:m} \times B_{n:m}) \times 2^{2m} + (A_{n:m} \times B_{m-1:0}) \\ &\quad \times 2^{m} + (A_{m-1:0} \times B_{n:m}) \times 2^{m} + (A_{m-1:0} \times B_{m-1:0}) \\ &= (A_{n:m} \times B_{n:m}) \times 2^{2m} + (A_{n:m} \times B_{m-1:0} + A_{m-1:0} \times B_{n:m}) \\ &\quad \times 2^{m} + (A_{m-1:0} \times B_{m-1:0}) \end{aligned} \tag{2.2}$$

The Wallace Tree takes a similar concept to the extreme with $m = 1$. This is a useful observation that can assist in accommodating varying operand sizes during multiplication.

2.6 Pipeline Hazards

In computer architecture, hazards are conflicts within a pipeline forcing the pipeline to be stalled to avoid potentially erroneous instruction results [13]. There are three different types of hazards:

- **Structural hazards**: The underlying hardware does not support executing a set of instructions in the same clock cycle
- **Data hazards**: The result of an operation is not ready when a subsequent instruction needs that result. Three types of data hazards exist, which are read after write (RAW), write after read (WAR) and write after write (WAW) hazards. The latter two only occur when out-of-order execution is allowed
- **Control hazards**: The program needs to make a decision dependent upon the result of a previous instruction while other instructions are executing

2.6.1 Forwarding

Forwarding is a way to avoid being forced to stall the pipeline by making data from stages further down the pipeline available earlier [13]. This implies that certain data hazards can now be resolved, no longer demanding the pipeline to be stalled. It does, however, also imply that extra hardware must be added to the pipeline, which may change the critical path (thus decreasing the achievable clock frequency). Figure 2.6 shows the different flow through a normal five-stage pipeline with and without forwarding.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
vadd v0, v5, v9	OF	EX	MEM	WB							
vadd v2, v6, v7		OF	EX	MEM	WB						
vadd v5, v0, v2			OF	Stall	Stall	EX	MEM	WB			
vor v1, v5, v7				OF	Stall	Stall	Stall	Stall	EX	MEM	WB

(a)

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
vadd v0, v5, v9	OF	EX	MEM	WB							
vadd v2, v6, v7		OF	EX	MEM	WB						
vadd v5, v0, v2			OF	EX	MEM	WB					
vor v1, v5, v7				OF	EX	MEM	WB				

(b)

Figure 2.6: The normal flow through a pipeline where data hazards exist, (a) without forwarding and (b) with forwarding.

2.7 Processor Performance Metrics

In order to compare the performance of different processors, certain metrics are commonly used. In this project, we will determine the performance from three metrics:

1. **Cycles per instruction (CPI)**: Determines how many cycles are required to execute a single instruction, and is thus a latency (L). This will differ between applications
2. **Speedup**: Measurement of the relative performance between different processors. In this project, we will calculate this by comparing how many cycles the benchmarking programs require to finish executing:

$$S = \frac{L_{\text{old}}}{L_{\text{new}}} \tag{2.3}$$

3. **Hardware utilization**: The resource usage, power consumption and achieved clock frequency for different implementations

3

Methods

In this chapter the process of the thesis project will be discussed. First, we provide an overview of the general workflow in Section 3.1, then a list of materials used during this project in Section 3.2, the benchmarks created to evaluate our prototype in Section 3.3 as well as how the testbench was created to verify our prototype in Section 3.4.

3.1 Overview

Before implementing our design, we first started by defining the minimum requirements needed to run the programs described in Section 3.3. In particular we looked for the minimal set of instructions needed to run these programs. The instructions in this minimal set were given priority in our development.

After defining the minimal set of instructions, we created an abstract overview of the micro-architecture that demonstrated roughly how the layout of the vector pipeline would look like, along with any major internal components. After that, we followed an iterative process of designing, implementing and then testing components of the design (ALU, memory, etc.) independently, before grouping them together into a vector pipeline in a later stage.

The vector pipeline was then tested as a whole with randomized test vectors (described in Section 3.4) before integrating the pipeline with a RISC-V scalar processor.

Agile methods were used to keep track of what assignments remained in a specific sprint, with scrum meetings held once a week on Mondays. On top of that, we had a weekly meeting on Wednesdays with our advisors to track our progress, brainstorm solutions, and guide us through technical challenges we faced.

The project was maintained on a local server using Git to keep track of development, and GitLab milestones were used to add tasks to a sprint, where each task is assigned to a person. Every task has subtasks that include writing documentation, creating a block diagram, writing a testbench and verifying functionality with the testbench.

3.2 Materials

The materials used in this project are listed in Table 3.1. The reasoning behind choosing these materials are mainly that they were available to us, as well as that they were used in the RISC-V processor we wanted to integrate with.

Table 3.1: List of materials used in this project.

Material	**Version**	**Description**
Vivado	2022.1	Used to synthesize the design and get relevant results
Questasim	2022.3	Used for simulating our testbenches
Python	3.11.2	Used to generate test vectors to use with the testbench
GCC	13.1	Used to create the testbench programs used to evaluate performance
VHDL	IEEE 1076-1993	Tool used to design and describe the prototype
Xilinx Kintex UltraScale	XCKU040[i]	FPGA development board

[i] Available **here**.

3.3 Benchmarks

As mentioned earlier, we aim to assess the use of RVV for machine learning applications. As described in Section 2.3, there are a few key operations that are commonly used in such applications. Therefore, we decided to create benchmarks to test these particular operations.

Table 3.2: The benchmarking programs that were created to assert prototype completion.

Program	Description
`SAXPY`	A common math operation of $\mathbf{Z} = a\mathbf{X} + \mathbf{Y}$
`ReLU`	An implementation of the `ReLU` function
`MATMUL`	Matrix multiplication of two matrices
Convolution	2D matrix convolution (without zero padding)

Each of the programs was written in C and with 8-bit data. The GCC compiler was used to compile the code, however since the compiler does not support auto-vectorization yet, some functions had to be manually vectorized (can be seen in Appendix A.1).

3.4 Testbench

In order to test the pipeline, a testbench and corresponding test vectors were created. Utilizing a Python script to generate test vectors, we created several testbenches for different parts of our prototype. The flowchart for the first two testbenches used (to verify the functionality of the ALU and the pipeline) is shown in Figure 3.1.

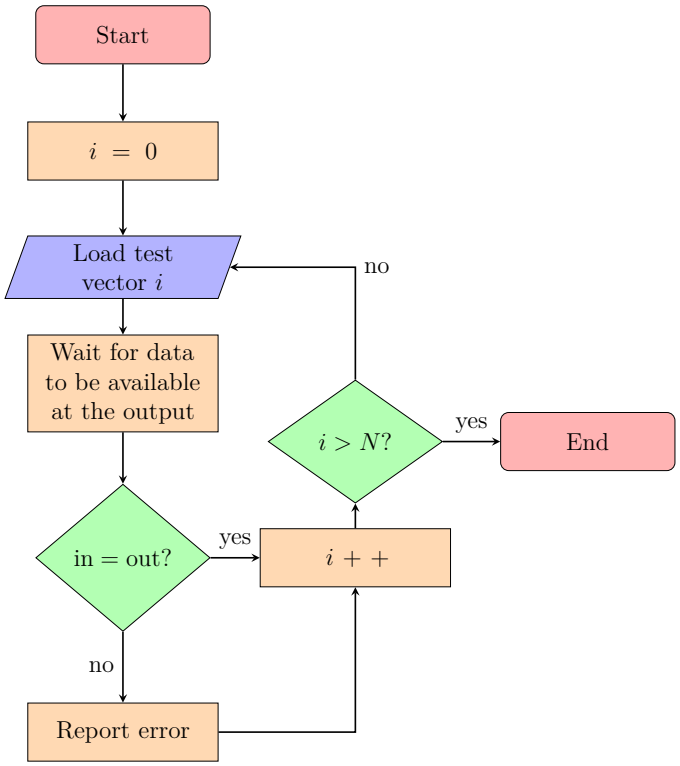


Figure 3.1: Flowchart of our testbenches used to test the ALU and pipeline.

It is worth noting that the pipeline was tested as a whole, including the hazard detection, forwarding functionality, and internal functional blocks (ALU, multipliers, etc.). However the test vectors did not include instructions that required or interfaced with the scalar core, namely memory instructions.

4

Prototype

In this chapter, the prototype we developed will be described in detail. First, an overview is given in Section 4.1, followed by an explanation of the vector pipeline and its stages in Section 4.2. We then discuss the ALU in Section 4.4, memory management in Section 4.5 and lanes in Section 4.6. In Section 4.7, we describe how hazard detection is implemented and masking is described in Section 4.8.

4.1 Overview

The prototype is mainly a vector pipeline integrated to an existing RISC-V scalar processor. This allows us to focus solely on designing the vector extension rather than the processor as a whole. This does mean however, that the prototype's design has to take into account the constraints, limitations and existing features of the scalar core.

While designing the prototype a few notable design decisions were taken that impacted the design of the underlying blocks. One such design decision was that all vector registers are, at least, 128 bits wide. This value was chosen since we would be able to handle more data than in the scalar pipeline and with different element widths, meaning that we would see the impact of vectorization on performance. At the same time, we did not want to make the design too complex, hence why we did not choose a minimum value larger than 128. If we did, we would have been required to use significantly more hardware to support the same instructions currently supported, impacting, for instance, the clock frequency.

Another design decisions was to split the computation into two different lanes, each handling 64 bits of data. This was done since version 1.0 of the RVV specification currently only supports element widths up to 64 bits [23]. We would thus be able to support all instructions without any large-scale design modifications.

Finally, memory accesses are handled through the scalar pipeline to utilize existing hardware and reduce the complexity of the prototype. See Section 4.5 for more details about how memory accesses are managed.

An overview of the entire system, showing our addition to it, is shown in Figure 4.1. The original diagram which Figure 4.1 is based on, is for the Microblaze [24] architecture which is similar *but not the same* to the internal RISC-V scalar processor with which we are integrating.

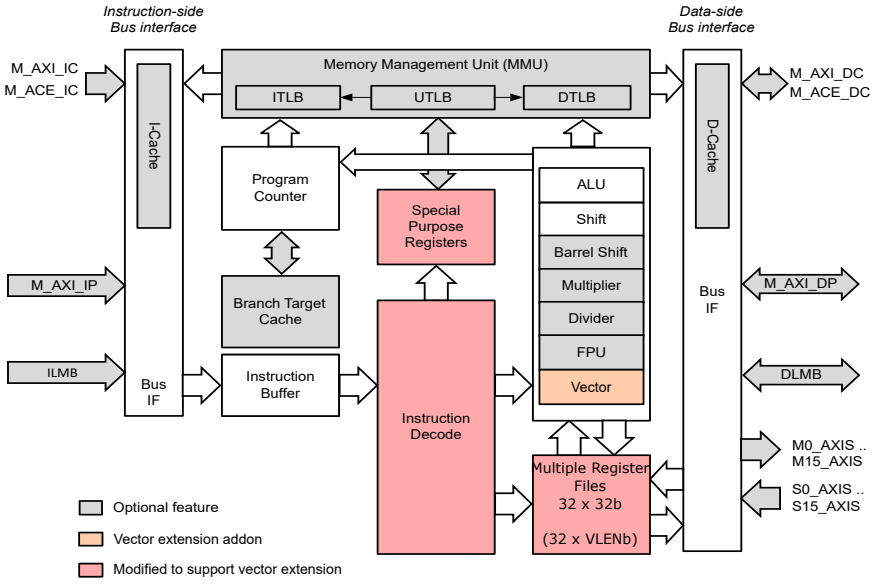


Figure 4.1: Overview of the complete system system containing RVV. The original block diagram can be seen in [24]. The *Instruction Decode*, *Special Purpose Registers* and *Register File* blocks also required modifications to support RVV. In particular, the register file now contains 32 additional registers for vector instructions, of which each register is vector register length (VLEN) bits wide.

4.2 Pipeline

The pipeline for the RISC-V vector extension consists of four stages:

- Operand fetch (OF): is partly shared with the scalar pipeline. All control signals are generated at this stage, and all vector operands are fetched
- Execute (EX): contains the ALU and is where the majority of instructions are completed
- Memory access (MEM): is used to assemble loaded data and to offer a space for multi-cycle operations (for instance, multiplication) to iterate
- Write-back (WB): updates the vector register file (VRF)

The block diagram of this pipeline is shown in Figure 4.2 and the different stages are described in the subsequent sections.

4.2.1 OF Stage

The first stage in the pipeline is the OF stage. In this stage the instruction is decoded to generate all the control signals for the following stages; this is also where all the data for the instruction are gathered from the VRF or requested from the scalar register file (XRF). All data must be ready and up-to-date at the end of this stage else this stage is stalled as described in Section 4.7.

Some notable signals that are passed from this stage are:

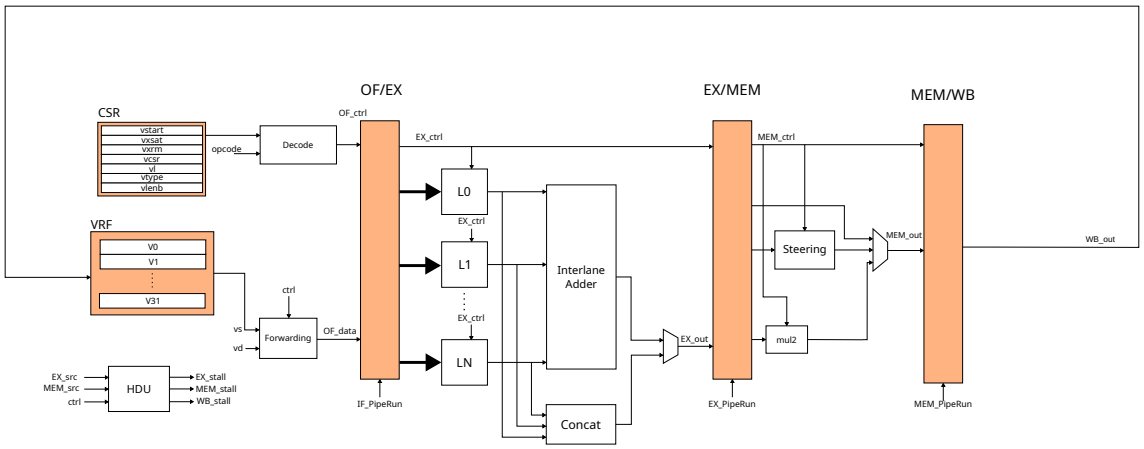


Figure 4.2: Block diagram of the RVV prototype.

- ALU control signals that specify its operation mode
- EX stage output select signal that determines which component's output will carry the output of the EX stage
- `funct` signal that encodes the instruction function
- Vector mask (fixed at `v0`) and a mask enable signal
- Destination address
- Vector operands `vs1`, `vs2` and `vs3`, as well as the 5-bit immediate value
- `Instr_valid` to indicate if the instruction is not recognized or is invalidated externally (i.e. from the scalar core)

4.2.2 EX Stage

The EX stage is where most instructions produce their final result. The main blocks in this stage are:

- An extender block that determines how long and with what value (zero or one) to extend immediate values.
- The extender is followed by an input expander (described in Section 4.3) which 'splats' a scalar value into a vector suited for a vector lane.
- Lanes
- Inter-lane adder
- Multipliers
- Control and status registers (CSRs)
- Memory address generator
- Vector unpacker

The first two blocks, extender and expander, prepare and reshape immediate values to be used as vectors in other blocks. Lanes, inter-lane adder, and multipliers are all functional blocks that take in vector inputs and produce an output that will be passed on to the next stage of the pipeline. The multiplier block in this stage consists of 16-bit multiplier array in addition to the adder circuit to generate 32-bit multiplication results.

Lastly, the memory address generator block generates offsets that are added to a base memory address in the scalar pipeline, and the vector unpacker block splits a vector into chunks that are stored to memory. Both blocks direct their results to the scalar pipeline.

At the end of the stage we have a multiplexer (mux) to select which functional block to forward the result from to the next stage, in addition to the possibility of forwarding a vector operand without modification (needed for when a vector register should be moved to a scalar register). The select signal of this mux is determined in the OF stage and depends solely on the `funct` signal.

It is worth mentioning that the CSRs are updated in the EX stage as they are frequently read on the following instruction, and coincidentally the EX stage is where most instructions first use the CSRs.

4.2.3 MEM Stage

The MEM stage is mainly used for operations that require multiple cycles to complete. In our instance, the operations that require multiple cycles are memory load operations and operations that include multiplication. Looking at the MEM stage, we see two functional blocks to accommodate the multi-cycle operations. The memory load block, which steers loaded data to the right position in a vector, is described in Section 4.5 and the multiplier block is described in Section 4.4.3.

Similar to the execution stage, at the end of the MEM stage we have a mux to select which functional block to choose the result from and propagate to the next stage. The select signal of this mux is determined in the MEM stage and depends solely on `funct`.

At this stage, the vector mask is applied, if it is enabled, to the output of the stage (discussed further in Section 2.2.3).

4.2.4 WB Stage

Lastly, the WB stage is where we write the result to the destination register. Before writing to any register file, we check to see if the instruction is valid or not, and whether the result is bound for the VRF or XRF. Based on these conditions we decide if the result should be a committed to a register file or not, and to which register file in particular.

4.2.5 Synchronization

Since we are developing a vector extension, and only support in-order execution, it is crucial that our pipeline stages remain synchronized with the scalar pipeline. Therefore all stages, just like the scalar core, are one clock cycle long (for example, no `EX_1`, `EX_2`, ... stages). For operations that require multiple clock cycles to complete, we iterate them in the MEM stage, while stalling the previous parts of the pipeline. The reason we consolidate all the iterations into MEM stage, is to prevent generating a stall signal from the EX stage, as that is a restriction imposed by the scalar pipeline.

All stages carry a `stage_valid` signal which indicates if the current stage is carrying a meaningful instruction or not. This signal is raised in the OF stage if a vector instruction is received and is propagated down the pipeline. If a stage is invalidated for whatever reason (for example, stall), the instruction in that invalidated stage will continue to be invalid down the pipeline. Invalidating a stage with this signal is akin to inserting a bubble into the pipeline. Hence the stage's state will not be considered in hazard detection or forwarding, nor will it update the VRF.

Additionally, all pipeline stage registers are partly controlled by `PipeRun` signals originate from the scalar core. In the case a `PipeRun` of a stage is set to low, the pipeline register of that stage is not updated on the next clock edge except for the `stage_valid` signal which will be set to low. This effectively stalls the stage and inserts a 'bubble' into the following stage. Furthermore, the `PipeRun` signals are chained together so that if the latter stages of the pipeline have a low `PipeRun` signal, so will the earlier pipeline stages.

Although the `PipeRun` signals originate from the scalar core, they are used by all the pipelines (vector and scalar) to determine what stage should progress and to stay in sync. Hence the `PipeRun` signals are set to low if *any* pipeline needs to stall at a given stage.

Figure 4.3 illustrates the dependency between the different stage `PipeRun` signals, and how the stall signals affect them.

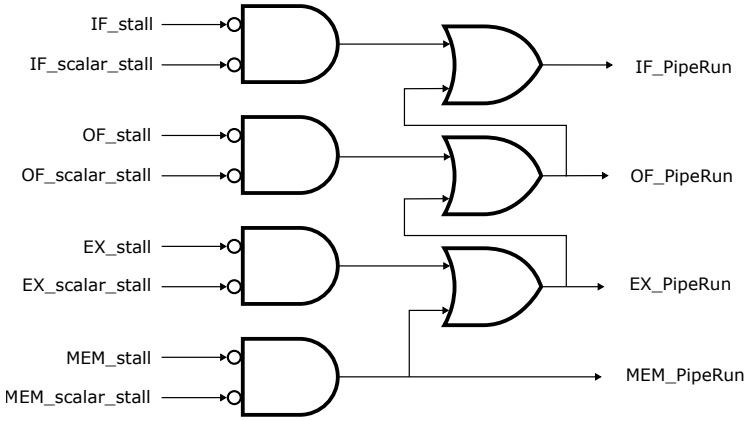


Figure 4.3: How the `PipeRun` signals are generated.

In the following sections we will look closer into the components that make the

different stages we briefly discussed above.

4.3 Scalar Input Expander

This block expands a scalar value into a vector format (see Figure 4.4). This is necessary as the arithmetic blocks expect vectors of data, and expanding the input in this manner allows the following blocks to operate without any consideration if the original input was a vector or a scalar value.

To achieve this expansion, the scalar value will be replicated across a vector depending on the target element width. The scalar value should already be sign-extended to 64 bits prior to entering this block.

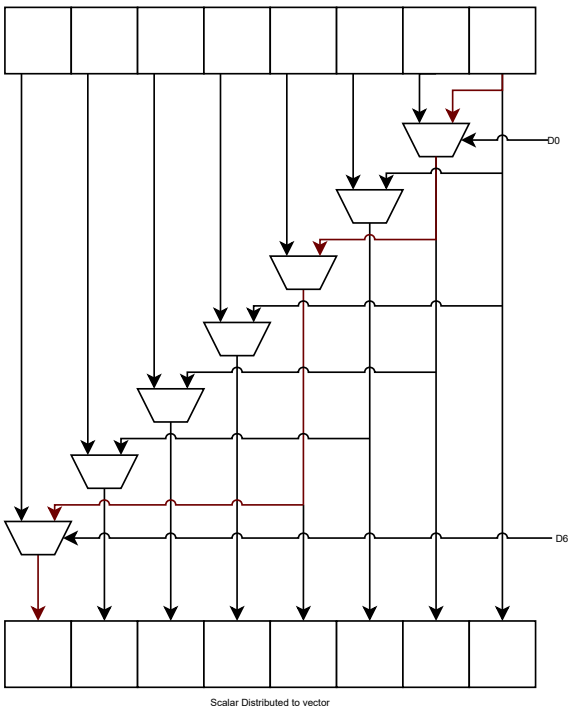


Figure 4.4: Block diagram of the scalar input expander.

The principle behind this design is that every output block (shown as a square in the lower row of squares in Figure 4.4) can either produce a duplicate of a previous group of data (each group being set element width (SEW) bits wide), or just a copy of the corresponding input group. In other words, the muxs decides whether to pass a copy of a previous group or to pass the input group as is. By using this block and expanding the scalar or immediate value we are able to use the same design for all ALU operations, even if one of the operands is a scalar/immediate rather than a vector.

For example, consider the scalar input (sign extended) was `0xFFFF FFFF ABCD 1234`. The output of the expander would then be as shown in Table 4.1. This shows the SEW least significant bit (LSB)s, duplicated $\frac{64}{\text{SEW}}$ times.

Table 4.1: Example of how the scalar input expander affects the scalar input.

In	SEW	Out
0xFFFF FFFF ABCD 1234	8	0x3434 3434 3434 3434
0xFFFF FFFF ABCD 1234	16	0x1234 1234 1234 1234
0xFFFF FFFF ABCD 1234	32	0xABCD 1234 ABCD 1234
0xFFFF FFFF ABCD 1234	64	0XFFFF FFFF ABCD 1234

4.4 Arithmetical and Logic Unit (ALU)

In the sections that follow, the ALU of our prototype will be described in detail.

4.4.1 Zero/Sign Extender

The purpose of this block is to either zero- or sign-extend a vector register. In RVV version 1.0, this is implemented by taking a slice of the lower bits of each vector element and then either zero- or sign-extending the rest [6]. The slices can either be 1/2, 1/4 or 1/8 of the SEW.

4.4.2 Adder/Subtractor and Comparator

This block performs addition and subtraction of two vector registers, and accommodates different element widths. This is achieved by having a 2:1 mux between every 8-bit adder, where we can choose what value the carry-in of the adder should have depending on the operation. For instance, if we are to perform the operation $f(A, B) = A - B$ on 8-bit element width, we have to set the carry-in high for each 8-bit adder and also negate the B operand (to perform 2's complement). Depending on SEW, the adders are grouped to behave as eight 8-bit adders, four 16-bit adders, two 32-bit adders, or one 64-bit adder. The `Sel_N` signal (shown in Figure 4.6) determines if the mux should pass the carry out of the previous block, or a bit that indicates if a subtraction operation should occur.

Since a subtraction operation can be used to perform a comparison, this block also supports such operations. This is to reduce the overhead of creating a separate block only for comparisons and instead using the already existing ALU to lower hardware resource usage. Table 4.2 shows the implication of different results of subtracting B from A. In order to support all of these instructions, a Karnaugh diagram was created to minimize the logical expression that determines if a subtraction operation should be carried out (see Figure 4.5).

Table 4.2: Implications of different results depending on the result of a subtraction.

Result	Signed?	Implication
Zero	N/A	$A = B$
MSB $= 1$ and V $= 1$	Yes	$A < B$
$\mathtt{C}_{\mathtt{out}} = 0$	No	$A < B$

def \ abc	000	001	011	010	110	111	101	100
000	0	–	1	1	–	–	–	–
001	0	0	1	1	–	–	–	–
011	1	1	1	1	1	1	1	1
010	–	–	–	–	–	–	–	–
110	0	0	–	–	1	1	0	0
111	–	–	–	–	–	–	–	–
101	–	0	–	–	–	–	–	–
100	0	0	1	1	–	–	–	–

Figure 4.5: Karnaugh diagram of the subtraction, the result being $b + \bar{d}e$.

The block diagram of this block is shown in Figure 4.6.

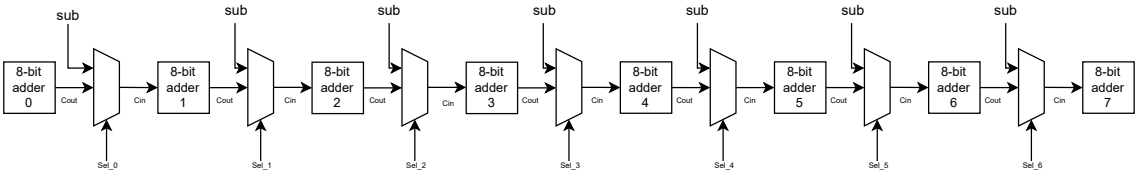


Figure 4.6: Block diagram of the adder/subtractor & comparator block.

4.4.3 Multiplier

Multiplication, like most blocks described here, is dependent on the value of SEW. Creating a multiplier for every possible SEW is unreasonably expensive so instead we create larger multipliers using a cascade of smaller multiplier blocks, the smallest being a 16-bit one.

There are two reasons we decided on 16-bit multiplication being our base multiplication unit, the first is that it maximizes hardware utilization when accommodating the different SEW sizes. Second, a 16-bit multiplier can likely be implemented using one DSP48 block which is available on most of the FPGAs we are considering to target.

In order to accommodate all SEW we need 16 16-bit multipliers per lane. Multiplication is implemented by first calculating all partial products, which are all combinations of 16-bit chunks of the 64-bit input data, that are then added together to produce the final result. The block diagram of the different cases are shown in Figure 4.7.

For SEW = 32 we need to group some of the multipliers similar to what was described in Section 2.5.2. Since the partial products in Equation (2.2) are powers of two, we

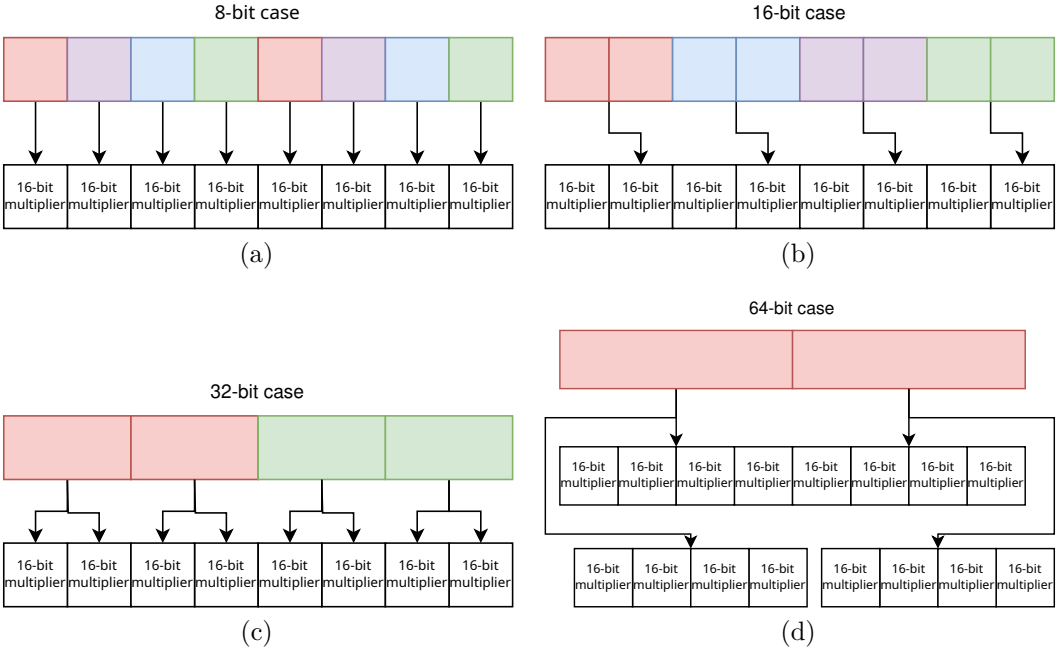


Figure 4.7: Block diagram of the multiplier block configured for (a) 8-bit, (b) 16-bit, (c), 32-bit and (d) 64-bit.

can implement them in hardware using left shifts. The grouping in the 32-bit and 64-bit cases are shown in Figure 4.8. We see that we need four 16-bit multipliers to achieve 32-bit multiplication in addition to some circuitry to add the partial products, while for SEW = 64 we need to group four 32-bit multipliers.

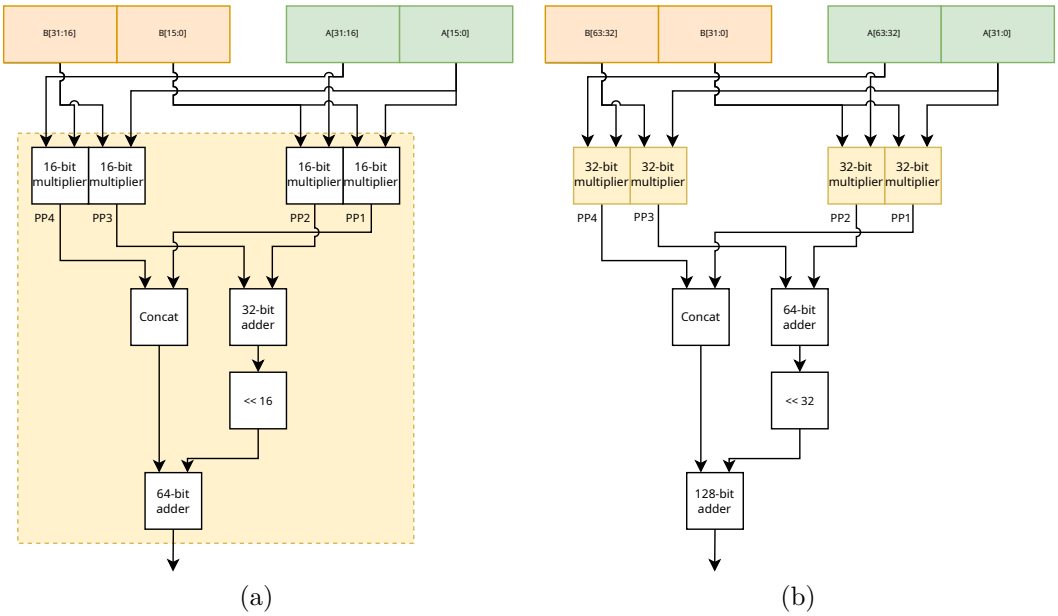


Figure 4.8: Internal block diagram of the (a) 32-bit and (b) 64-bit multiplier blocks.

All the configurations above work as intended for unsigned integers, since we operate

on parts of operands (i.e. $A_{m-1:0}$) and do not currently retain the sign of a particular part of the operand.

It should also be noted that multipliers described in this section produce the full product size, though that might not be necessary if only narrowing multiplication is needed (output and input have the same bit width) and, therefore, the amount of multipliers needed when grouping can be reduced.

4.4.4 Shifter

The shifter block is implemented by determining how large each element is, by reading the current SEW configuration, and from that shifting the vector register by the appropriate amount. The vector register is shifted by $\log_2(\text{SEW})$ bits since shifting by anything more than that will yield the same result as shifting by $\log_2(\text{SEW})$ (see Figure 4.9).

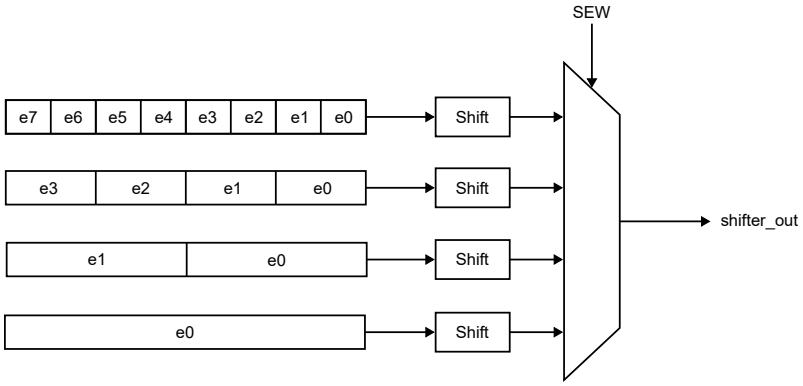


Figure 4.9: The block diagram of the shifter. The current SEW configuration changes the amount to shift by and how many elements to shift.

4.4.5 Intra-lane Reduction

The intra-lane reduction block is used to calculate the reduction sum. Depending on SEW, the result will be available at different stages (as seen in Figure 4.10).The orange mux will be active when SEW = 16, blue mux when SEW = 32 and none of them are active when SEW = 8. For SEW = 64 no reduction is performed inside of a lane.

The intra-lane reduction adder is supplemented with inter-lane adders that scale in a similar manner depending on the number of lanes.

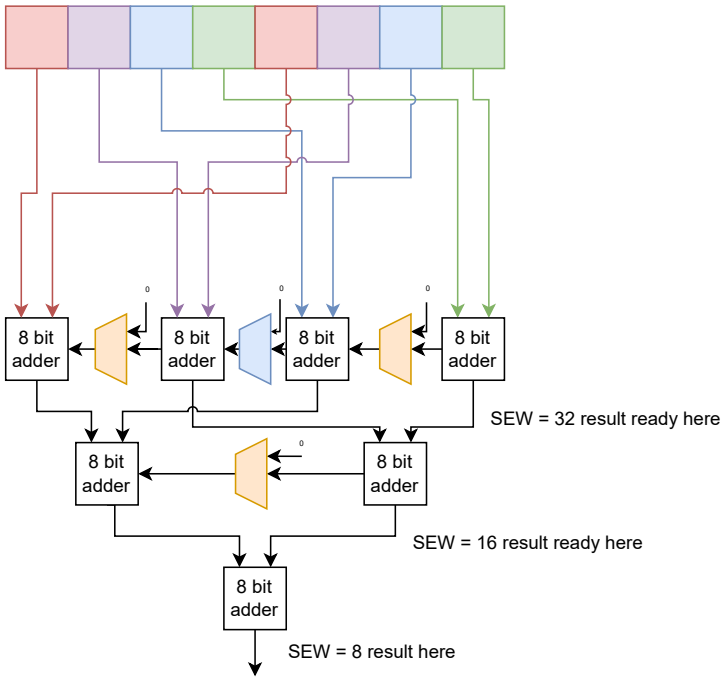


Figure 4.10: The intra-lane reduction block diagram.

4.5 Memory Management

Since our vector extension in RISC-V uses the same main memory as the scalar core, the only thing concerning memory management we had to consider was address generation and steering data into the correct chunk of a vector register.

Vector loads and stores depend both on the effective element width (EEW) which is encoded in the instruction, as well as the addressing mode chosen. The addressing modes are:

Indexed: Each access is given by a vector register holding an array of indices. Both ordered and unordered indexing is supported by the RVV specification [23], however this implementation only supports unordered indexing

Strided: Accesses elements starting from a base address and then in increments of the stride. For instance, if the EEW was 8, base address was $0x100$ and the stride was $0x4$ we would access addresses $0x100, 0x104, 0x108, \ldots$

Unit-strided: Continuous access starting from the base address

In our system, memory is word-aligned and byte-addressable, so each load/store address will read/write to an entire word (32-bits) with address of that word being a multiple of 4. In the case of misaligned memory access - memory element accessed is not naturally aligned to the size of the element - the RVV specification permits the designer to raise an exception and not support this type of access. We decided not to support this because we deemed it to be too complex and the benchmarking programs we had compiled never used this feature.

Since a vector element can be of size $[8, 16, 32, 64]$, we needed to calculate a way to determine the number of accesses that are required to fetch an element (see Table 4.3).

Table 4.3: The number of words needed to be accessed depending on the EEW.

EEW	Required accesses
8	1
16	1
32	1
64	2

For indexed memory accesses, we fetch a vector register from the VRF and, depending on the EEW, generate different amounts of offsets. This process is shown in the bottom half of Figure 4.11 and the FSM is illustrated in Figure 4.12. The `done` signal that returns the FSM to the idle stage is dependent upon the EEW, meaning that indexed load and store instructions will take different amounts of time to complete. The mux select signal depends on the EEW and a 4-bit counter signal that is continuously updated when in the `running` state (see Table 4.4). Since the EEW indicates how many elements we need to access, the `done` signal is set at different times. For example, if EEW = 8 we need 16 clock cycles to access all elements, while when EEW = 16 we only require eight clock cycles.

Strided memory addresses are formed by adding an offset to the base address in a cumulative fashion. Since, as mentioned previously, strided memory accesses always access the memory at the base address first, we need an FSM to control the select signals of the muxs in Figure 4.11 to determine what and when to add an offset to the base address. This FSM is shown in Figure 4.13, where the mux select signals are updated according to Table 4.5.

Table 4.4: Mux select depending on EEW and counter for indexed memory accesses. "&" indicates concatenating the signal with another signal.

`EEW`	`select[1:0]`
8	`counter[3:2]`
16	`counter[2:1]`
32	`counter[1:0]`
64	`counter[0]` & `0`

Now that we are able to generate the correct addresses, we now look into how we unpack a vector register for storing and how we pack it when loading.

For store operations, the same structure used in generating the indexed memory offsets (shaded purple area in Figure 4.11) is used, with the input to the block being the vector register we want to store. With this setup, we have the data element we want to store ready at the same time as the memory address in the EX stage. The width of the element we want to store is determined by either EEW or SEW depending on the store instruction. The outgoing data to memory is accompanied

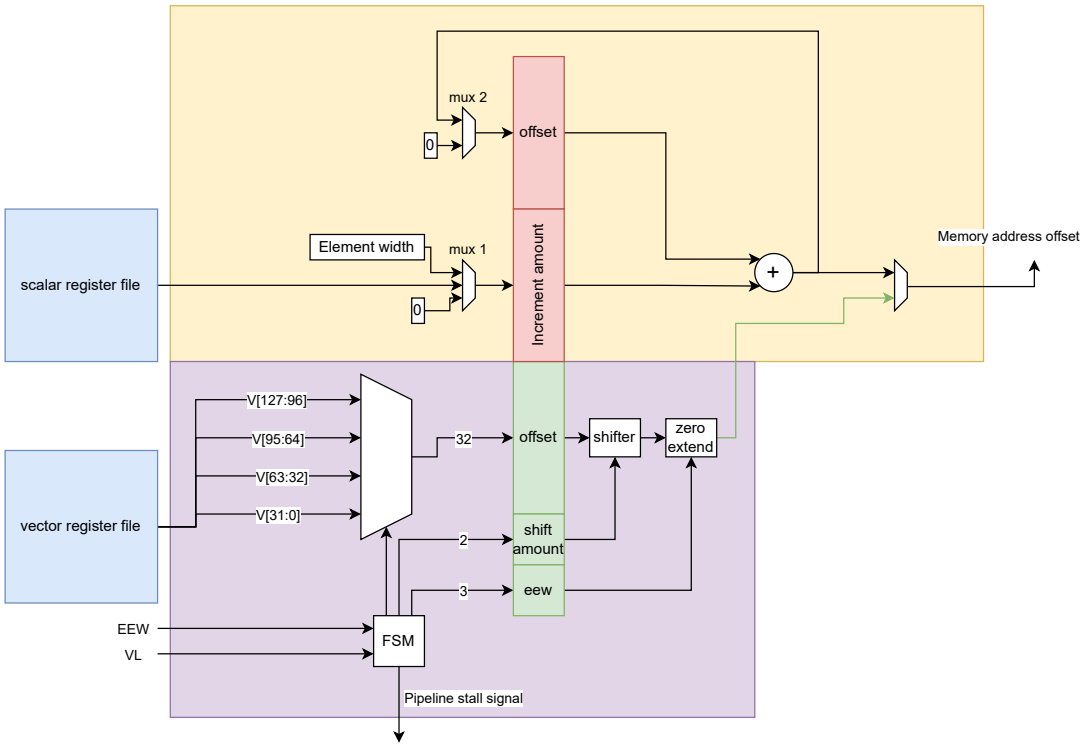


Figure 4.11: Block diagram of memory accesses. The top half (orange) shows the memory address offset generation for strided memory access, and the bottom half (purple) shows it for indexed memory access.

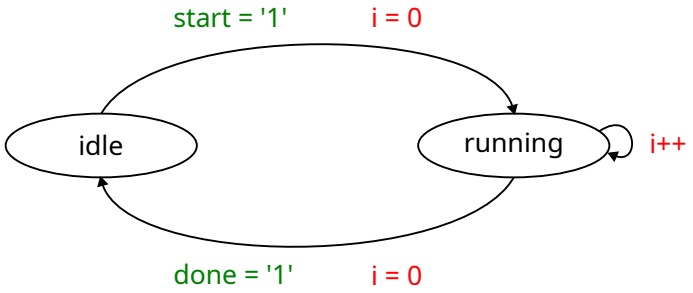


Figure 4.12: FSM of indexed memory access.

by a `byte_enable` signal that determines which bytes will be committed to memory. This is necessary for writing elements to memory that are smaller than a word and with a memory address not strictly word-aligned.

For load operations, we need to pack the bytes we receive into one vector register. Note that we receive an entire word from memory that we then select bytes from and add to our vector register. The number of bytes we select depend on EEW or SEW depending on the addressing mode (strided vs indexed). The memory address' lower two bits determine which bytes to choose from in cases the loaded data is smaller than a word.

The packing mechanism, depicted in Figure 4.14, works by inserting the bytes we select from the loaded word into the top of a register. The register with loaded

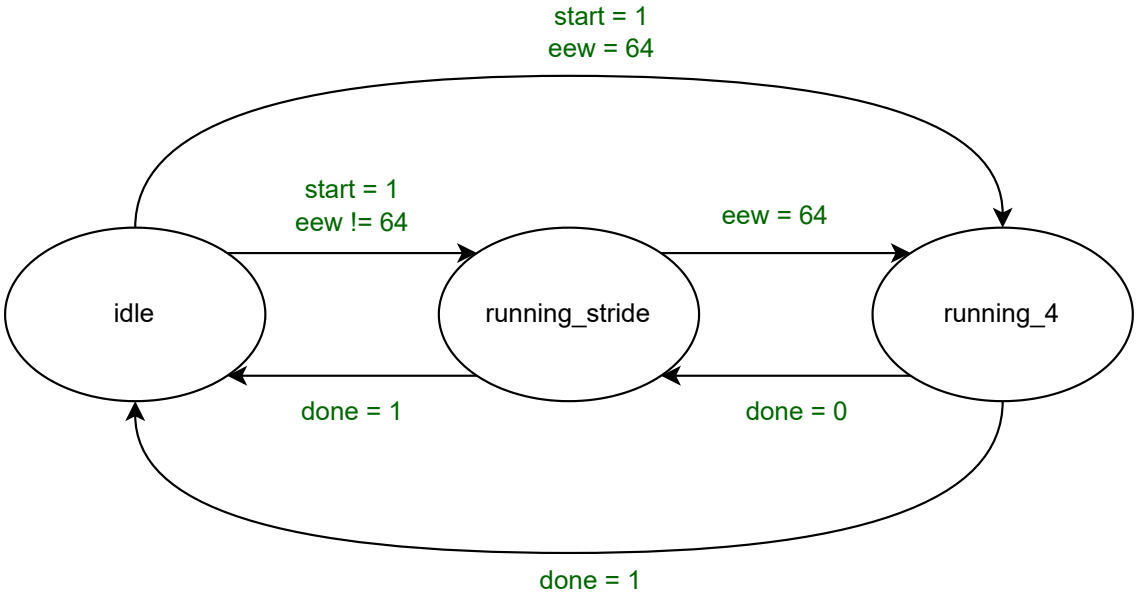


Figure 4.13: FSM of strided memory access.

Table 4.5: mux select depending on type of operation and FSM state. `running_stride` adds the stride from the VRF and stores it in the internal register for the base address. `running_4` is used for $EEW = 64$, where we need two memory accesses for every element.

Mux	State	Unit-strided	Strided
`mux 1`	`idle`	10	10
`mux 2`	`idle`	1	1
`mux 1`	`running_stride`	00	01
`mux 2`	`running_stride`	0	0
`mux 1`	`running_4`	00	00
`mux 2`	`running_4`	1	1

bytes is then right-shifted the appropriate amount (according to SEW) to make room for the next group of bytes we will load. Note that the top of the register varies depending on the number of elements in a vector, as set by vector length (VL), and their respective width. So in the case VL = 8 and VSEW = 8, the 'top' of the register would start at bit 63 and end at bit 56.

This mechanism is not controlled by a counter nor any explicit FSM, but instead continually inserts new elements to its output vector register. This has the side effect of continually updating the output with partial results until the final correct result is achieved at the last load.

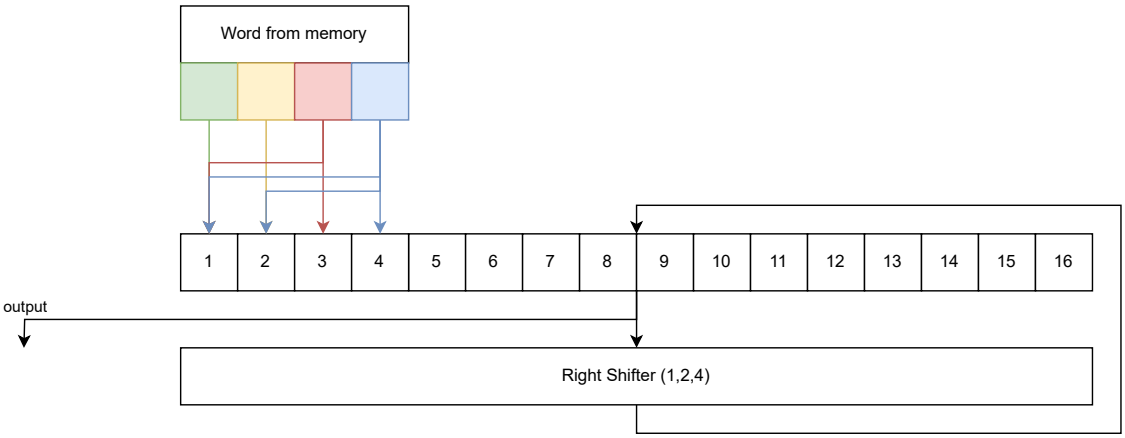


Figure 4.14: Packing mechanism of words loaded from memory

4.6 Lanes

A lane is used to reduce the sizes of certain components necessary for the vector extension instructions, such as ALU operations. By splitting up the 128-bit data into two 64-bit lanes we can perform all operations we allow inside of each lane and then combine the results before writing it back to the register file or main memory.

A lane currently contains a single ALU and a mux to select between an immediate value or a vector register for the ALU. We can modify our register-transfer level (RTL) code to accommodate a different VLEN), following the rule:

$$\text{VLEN} = \{64N \mid N \in [1, 2, 4, 8, \dots]\} \tag{4.1}$$

where N is the number of lanes. The reason behind the constant 64 is that each lane handles 64 bits of data, meaning that this is our minimum VLEN.

4.7 Hazard Detection

As mentioned in Section 2.6, RAW hazards occur when an instruction has not written back its result yet, while a subsequent instruction needs that result as an input. WAR and WAW hazards are not present in our pipeline as we perform strictly in-order execution. In the case of the vector pipeline, the source operands can either be an immediate value encoded in the instruction, a vector register, or a scalar register, and an instruction can of course have mixed source operand types. In case the source is an immediate, the value is readily available so no RAW hazard is possible, but in the two latter cases we need to check that no other instruction in the pipeline is going to write to these registers.

Since we are integrating with a scalar core, and the scalar registers might be written to in the scalar pipeline, we rely on the scalar hazard detector to detect hazards between the pipelines that occur when the vector pipeline is reading a scalar register that is being modified in the scalar pipeline. The remaining cases are detected and handled within the vector pipeline.

With the above in mind, in order to have a RAW hazard five conditions must be met between the OF stage and another stage in the vector pipeline and for the same operand register in the OF stage:

1. The operand register address in OF stage is equal to the destination address of the other pipeline stage
2. The operand register in OF stage and the destination register of the other pipeline stage refer to the same register file
3. The instruction in the other stage intends to write a value to a register file
4. The instruction in OF stage will use the value of the operand register; some instructions use an immediate value or read the destination register
5. The instruction in both stages is valid

When a RAW hazard has been detected, the OF stage will be stalled (using the piperun mechanism discussed in Section 4.2.5) until the result of the conflicting instruction further down the pipeline can be used. However these cases are minimized by the addition of a forwarding block (see Section 4.7.1) where a result can be forwarded as soon as it is ready in the pipeline.

Structural hazards in our design only occurs when a memory instruction is being executed. This is because we need to generate multiple pseudo-instructions in order to interface with the memory multiple times until all elements have been read or stored. The structural hazard detection block is shown in Figure 4.15.

Control hazards are not present in the vector pipeline but are instead handled in the scalar pipeline.

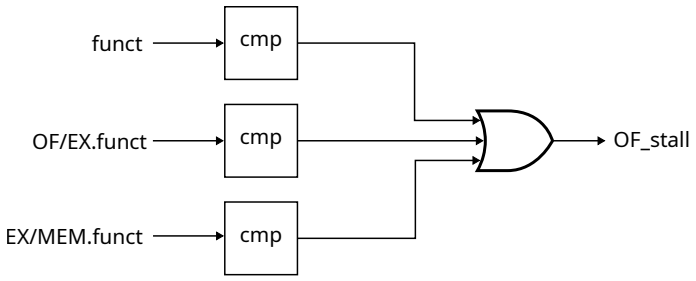


Figure 4.15: Structural hazard detection in the RVV pipeline.

4.7.1 Forwarding

When a RAW hazard is detected, a pipeline would typically have to stall until the data it needs is written back to the register file and can be read, however this can be mitigated if the result can be forwarded to the OF stage as soon as it is ready and when it is needed.

In our implementation we have limited forwarding to only vector register within the pipeline, meaning that if the vector pipeline is writing to a scalar register that is needed by the next vector instruction, we will still need to stall. Additionally there

is no forwarding between the scalar and vector pipeline, so a stall is issued in case one pipeline requires data from the other. This limits the forwarding conditions in the vector pipeline for each operand vector register in the OF stage to:

1. The source vector register address in the OF stage is equal to the destination address stored in the pipeline stage register
2. The destination register in the pipeline stage refers to the vector register file
3. The instruction in both stages is valid
4. The result in the pipeline stage is ready

If the above conditions are met, the result from the conflicting stage is forwarded, else we read the operand from the register file.

With the addition of the forwarding block, the hazard detection unit is updated to raise a stall signal only if forwarding conditions are not met, so the updated (simplified) logic expression is:

$$\texttt{OF\_stall} = (\text{hazard conditions}) \wedge \neg(\text{forwarding conditions}) \tag{4.2}$$

The forwarding block was implemented both with and without forwarding from the execute stage (by using an `if ... generate` block in VHDL). The reasoning behind this decision was that we could produce more results for comparison, meaning that we could see the impact of supporting/not supporting forwarding from the execute stage. Figure 4.16 shows how the data hazards are detected in our design with this arrangement.

Figure 4.16: Data hazard detection in the RVV pipeline. (a) without EX forwarding and (b) with EX forwarding. The red sections indicate the additional hardware necessary for EX forwarding.

4.8 Masking

Masking allows us to selectively apply a function to vector elements, with masked elements being handled per the mask policy while unmasked elements are updated.

Per the specification, RVV supports two mask policies. The first one, mask agnostic, permits updating elements that are not masked. The second policy, mask undisturbed, does not allow this. In order to simplify the design, we decided to always use mask undisturbed, even if mask agnostic was chosen. This meant that we only needed to support one of the two policies, while still complying with the specification.

When it comes to applying the mask, we decided to operate on all elements in the operand register regardless of the mask (with a few exceptions discussed below), and then apply the mask to the resulting register. This is done by selectively choosing elements between the outdated destination register and the result register, before updating the destination register.

Furthermore, RVV permits a user specified CSR VL, which specifies the number of elements in a vector. If VL is less than the number of elements in the vector, then the remaining elements, called tail elements, are treated as if they are masked. Tail elements can be specified to have similar policies as masked elements (agnostic and undisturbed) and we decided to always apply the undisturbed policy for the same reasons we always apply one masking policy.

Effectively, this means we now have two masks to apply to a result, and we combine them by performing a bit-wise AND between the two. This results in a new mask that specifies what elements to update, which is then used selectively choosing elements between the outdated destination register and the result register.

Applying the mask after a result is acquired works correctly for operations that operate on elements between two vectors but not for those that operate on elements within the same vector. In the latter cases (namely reduction operations), the mask must be applied during the operation and not after it, as the mask directly affects the result, and so these operations have the mask applied to the input registers and not the result register.

During the process of implementing masking several modifications of it were made. At the start, we assumed that that we would only need to apply the mask when writing back to the VRF in the WB stage. We later came to the realization that, since we support forwarding, we would also need to apply the mask on the forwarded data, and so the mask is now applied in earlier stages (MEM or EX depending on when the data can be forwarded).

5

Results

Here the results are shown and described according to the metrics described in Section 2.7. We will show the CPI and speedup results in Section 5.1, as well as hardware utilization in Section 5.2. The results were generated both by inference (for instance, calculating the number of clock cycles needed for an application) and by data provided by the synthesis tool Vivado, targeting a Xilinx Kintex Ultra-Scale FPGA. The data we retrieved from Vivado was resource usage, clock frequency, critical path and power consumption.

In order to produce comprehensive results, two parameters (forwarding from EX and VLEN) were modified according to Table 5.1. We then, for each run, used the optimal clock frequency that did not have timing violations. This was done by starting with a lenient timing constraint (50 MHz) and then increased or decreased it in order to find the highest clock frequency without timing violations (F_{max}). It was then assumed that the highest clock frequency without timing violations was the optimal one.

Table 5.1: The parameters used for different runs.

Parameter	**Values**
EX forwarding?	True, False
VLEN	128, 256, 512

In order to compare our results, results from a RISC-V processor core without vector extension were retrieved, utilizing an internal RISC-V processor core available from AMD.

5.1 CPI and Speedup

The CPI and speedup were calculated by running vectorized matrix multiplication code (Appendix A.1) on different design configurations in a simulation environment prepared for us by AMD. The design configurations were various lengths of the vector registers (VLEN), and were chosen to be 1/4, 1/2, 1 and 2 times the length of data in one column or row in a matrix. Since we are multiplying 64×64 matrices with 8-bit elements, the length of one vector is $8 \times 64 = 512$ bits, so the tested lengths are VLEN $\in [128, 256, 512, 1024]$ bits.

It should be noted that enabling or disabling forwarding from the EX stage had no effect on the performance as it happens that there is no scenario in the matrix multiplication program where a data hazard is resolved through EX forwarding (so it is not used).

The simulation environment made it possible to trace instruction execution on every cycle and mark the start and end of a function call, allowing us to measure the execution time, number of clock cycles, and also the number of instructions during the entire function call.

To calculate the CPI, we first mark the start time and end time of the function. We define these two points at the start of the clock cycle when the first and the last instruction in the function is indicated in the program counter (PC). After that we use the clock period to divide the duration of the execution time and attain the total number of clock cycles needed to run the entire function.

At the same time we count the number of instructions between the start and end time of the function by counting the number of times the PC transitions to different values.

With these two numbers attained we can calculate the CPI and the values are shown in Figure 5.1.

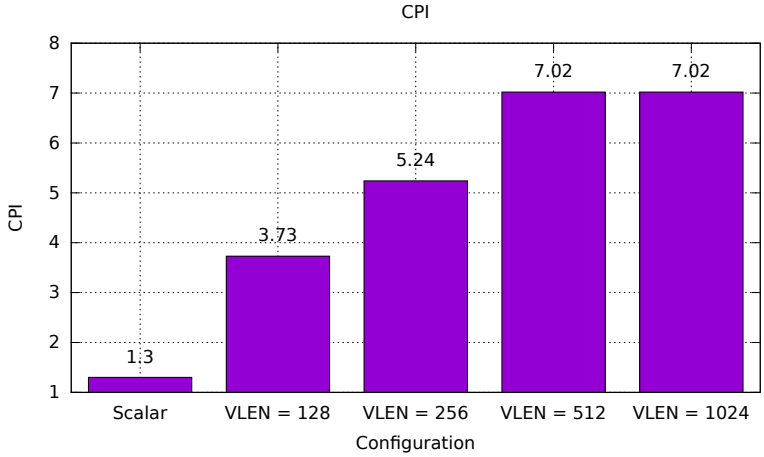


Figure 5.1: CPI of different implementations of the RISC-V pipeline.

As seen in Figure 5.1, the CPI increases with VLEN, however we can see in Figure 5.2 that the number of instructions decreases at a greater proportion than the increase in CPI. This hints to an overall reduction in the number of clock cycles for the entire functions as VLEN increases.

The CPI increase can be attributed to vector load instructions taking more clock cycles to load data, as a vector register can now contain more elements, and per our implementation, we require at least one clock cycle to load every element into a vector.

Similarly for the number of instructions, as we load and operate on more elements in

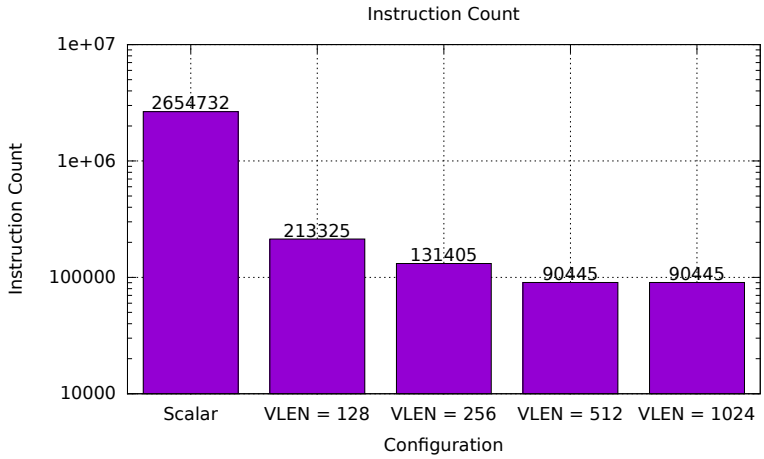


Figure 5.2: Instruction count of different design configurations.

vector register with increasing VLEN, the number of iterations to load and operate on data decreases, leading to fewer instructions overall.

Past VLEN = 512 the number of instructions and CPI remain the same, as the size of the register increases but the size and number of elements operated on remain the same. In this particular case 64 8-bit elements are operated on, which equates to 512 bits.

To contrast the vector extensions performance with the scalar core, we ran an equivalent matrix multiplication function that does not utilize any vector instructions, and measured the number of clock cycles taken. To calculate the speed up, we divide the number of clock cycles taken when running solely on the scalar core by the number of clock cycles taken when vector instruction are used. The relative speedups are shown in Figure 5.3.

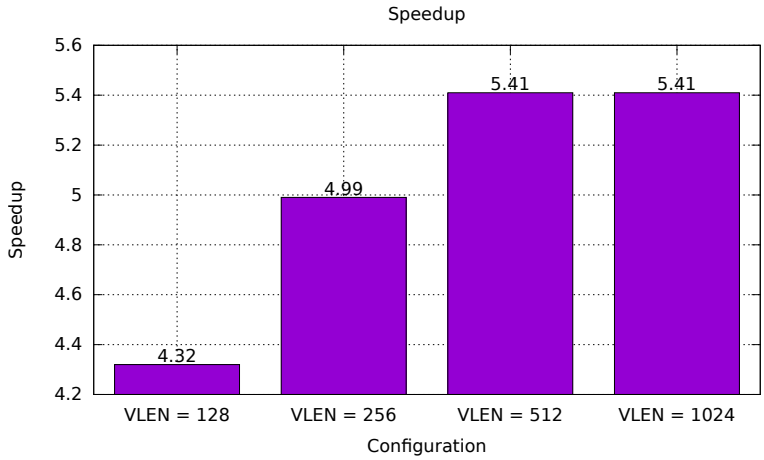


Figure 5.3: Speedup of different implementations of the RISC-V pipeline with respect to the scalar pipeline.

We can see VLEN = 512 bits gave us the greatest speed up of 5.41. This can

be explained by the fact that an entire row or column in the matrix is 512 bits (64 elements $\times$ 8 bits = 512 bits), and thus can be loaded and operated on within one iteration to produce one element of the output matrix. Larger VLEN configurations would provide virtually no speed up as they will not reduce the number of operations.

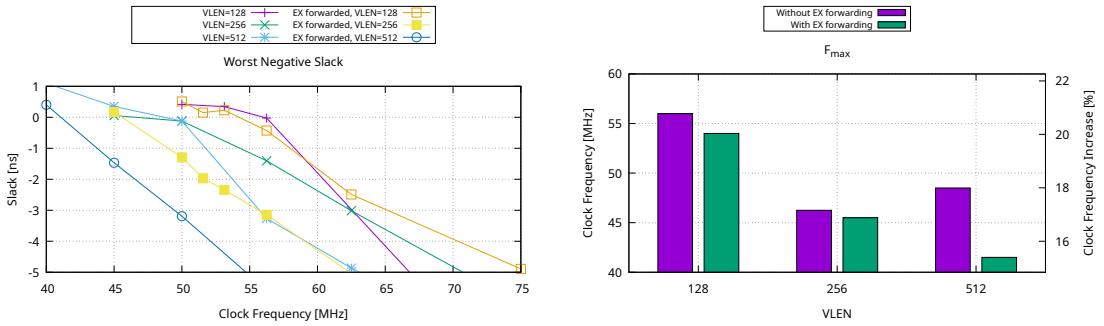
5.2 Hardware Utilization

The hardware utilization is split up into two different results:

1. **Absolute values**: The actual data retrieved when synthesizing and implementing the design
2. **Relative values**: The same data, now relative to the scalar values

Timing results are shown in Figure 5.4, resource usage in Figure 5.5 and power consumption in Figure 5.6. All figures show both the absolute and relative values.

We expect resource usage to double when doubling VLEN, but to remain almost unaffected with or without EX forwarding being supported. This is because the different VLEN configurations modify the size of registers and data paths throughout most of the design to match VLEN, whereas supporting EX forwarding only modifies the forwarding logic and hazard detector slightly. As we can see in Figure 5.5, the resource usage nearly doubles when doubling VLEN, which aligns with our expectation.



(a) (b)

Figure 5.4: Achieved (a) WNS and (b) $F_{\max}$ for different implementations of RVV. The absolute $F_{\max}$ is shown in the y-axis on the left-hand side, while the relative increase is shown on the right-hand side.

In terms of power consumption, we expected the static power to be fairly constant, since the tool used to obtain this data (Vivado) does not produce the most accurate results. For dynamic power, however, we expect to see an increase with higher clock frequencies and increased resource usage, since these are what largely determine the dynamic power:

$$P_{dyn} = \frac{1}{2} C V_{DD}^2 \times f_{clk} \tag{5.1}$$

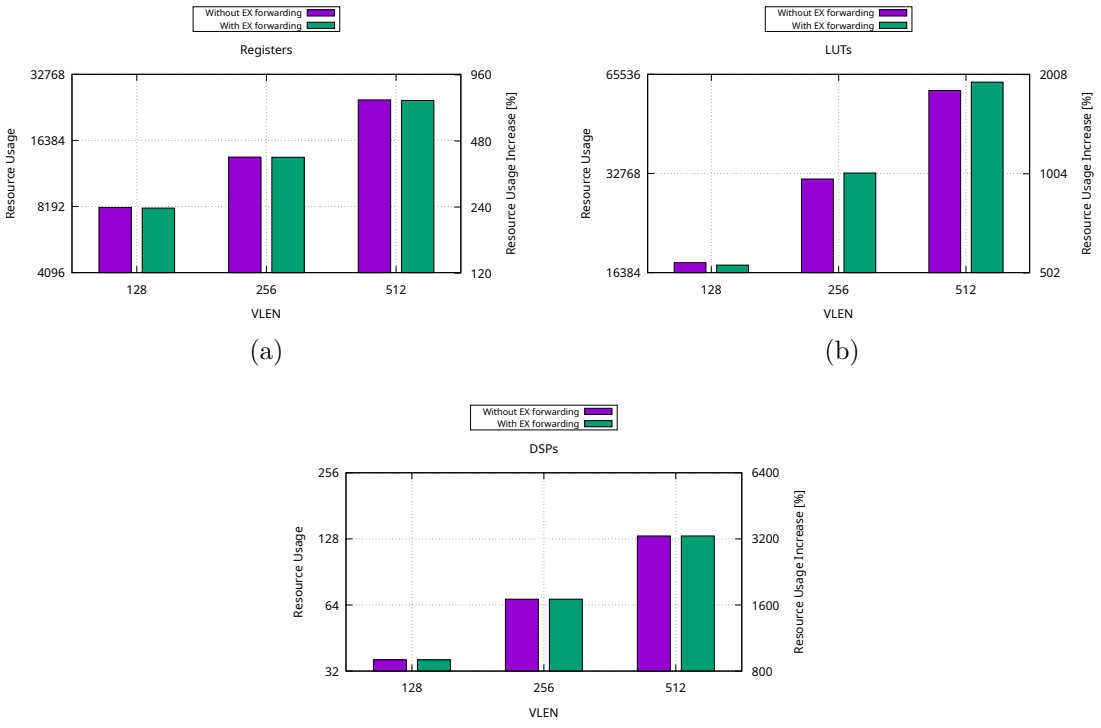


Figure 5.5: Increase in resource usage in terms of (a) registers, (b) LUTs and (c) DSPs of different implementations of RVV. The absolute values are shown in the y-axis on the left-hand side, while the relative increase in resource utilization is shown on the right-hand side.

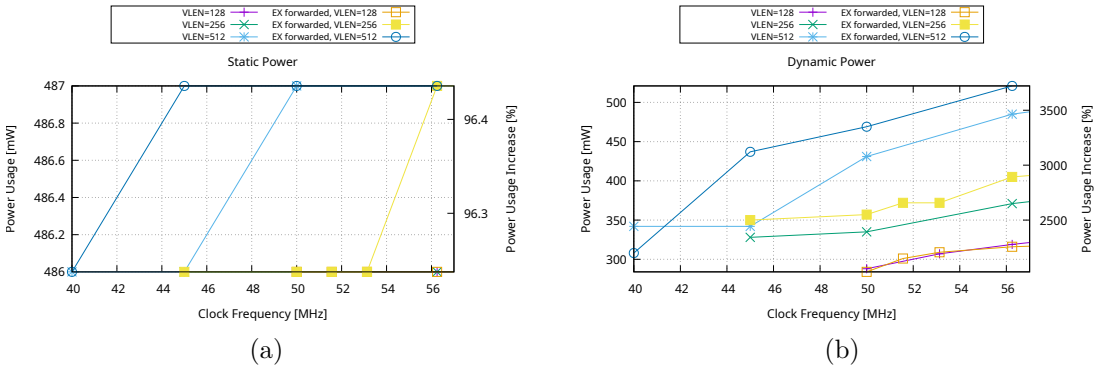


Figure 5.6: Increase in (a) static power and (b) dynamic power for different implementations of RVV. The absolute values are shown in the y-axis on the left-hand side, while the relative increase in power consumption is shown on the right-hand side.

From Figure 5.6 we can see this trend, where the static power remains mostly the same for different timing constraints, but the dynamic power consumption steadily rises when increasing the clock frequency.

The optimal obtainable clock period was assumed to be fairly constant for different VLEN configurations, since resources on an FPGA board are located close to one another, therefore using more resources should not affect the clock period substantially. As seen in Figure 5.4, most of our data points follow this trend, however one interesting thing to point out is that we are able to find a larger $F_{\max}$ with EX forwarding and VLEN = 512 than VLEN = 256. We believe this to be caused by the synthesis tool running for a longer amount of time for this configuration, thus allowing it more time to attempt to find a route that works.

6

Conclusion

In this project, we aimed to investigate the capabilities of RVV in terms of its proclivity for machine learning. To achieve this we first developed an RVV prototype, and then integrated it to an existing RISC-V scalar processor. After that, we compared the performance and resource utilization of the scalar core with different configurations of the RVV prototype. The performance measurements were obtained by running a matrix multiplication operation in a simulation, as such an operation is common in machine learning algorithms.

Although the prototype was developed within a limited scope (as mentioned in Section 1.4), we believe it lays the groundwork for future development.

Per our results, adding RVV to an existing processor is an expensive process and consumes significant resources, with the minimum vector length of VLEN = 128 bits increasing resources considerably. Adding the extension also affects the clock frequency of the entire processor, although we believe the effect can be minimized with more time and effort.

This cost however can be justified if a significant number of operations that will be conducted can be vectorized, such as the case in machine learning, and benefit from added vector pipeline. In the case of matrix multiplication we recorded a 5.4 speedup at the appropriate VLEN, with a significant reduction in total instructions.

Therefore, it is important when deciding on the VLEN to consider the expected sizes of data and how they can be best vectorized. From our findings, the best performance in matrix multiplication is attained at a VLEN equal to or greater than the number of bits in an entire vector being operated on. However, increasing VLEN beyond the number of bits being operated on only wastes resources.

In the case of soft-core implementations, RVV offers the unique advantage of allowing the user to reconfigure the hardware, and in particular VLEN, to best suit the expected workload to reduce the time and power needed to run an application.

6.1 Challenges

Initially, we had planned to construct the ALU, then when that step was achieved we would begin work on the pipeline. Based on feedback on our plan from our advisors at AMD, we decided against this and instead began by designing the

pipeline. This caused the design phase (which was initially planned to occur at a later stage) to be extended by three weeks. Because of this, we quickly ended up behind schedule, however, we had better understanding of the project which helped during the development phase.

One other challenge was creating the top level of the vector extension pipeline. This was mainly caused by us not accounting for certain issues, such as forwarding and masking. The consequence of this was that the top-level design took longer to construct, as well as forced us to append new limitations to the project in order to finish on time.

Another set of challenges was faced when working on integrating our extension with the existing scalar core. Notably synchronizing the scalar and vector pipelines took a significant effort when designing the vector pipeline as we had to abide by existing constraints in the scalar pipeline. Furthermore, our inexperience and unfamiliarity with the scalar core made this process take more time than first anticipated, as we had to heavily rely on our advisors to understand what modifications where necessary and how to make them.

6.2 Future Work

The current state of the project is that only a small subset of instructions are supported and verified to work. In order to support the entirety of the RVV specification, one would need to add support for: Division instructions, fixed-point instructions and floating-point instructions to name a few. The RVV pipeline construction simplifies adding these instructions.

Another feature of RVV is register grouping that allows multiple registers to be interfaced as just one register. We expect this could provide a great performance improvement as a smaller VLEN can provide a similar performance improvement as a larger VLEN.

Furthermore there is plenty left to be optimized and made more efficient. For instance the pipeline stages are not well balanced, with the WB stage having almost no functionality at the moment besides writing to the VRF.

Bibliography

[1] *Search trend for Machine Learning and Aritifical Intelligence*, Dec. 2022. [Online]. Available: https://trends.google.com/trends/explore?date=all&q=%2Fm%2F01hyh_,%2Fm%2F0mkz.

[2] J. L. Hennessy and D. A. Patterson, "Introduction," in *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, pp. 2–5, ISBN: 012383872X.

[3] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.

[4] Xilinx, "Versal: The First Adaptive Compute Acceleration Platform (ACAP)," 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp505-versal-acap (visited on 02/09/2023).

[5] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, version 20191213, Dec. 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (visited on 02/21/2023).

[6] K. Asanović, A. Waterman, A. Chapyzhenka, *et al.*, *RISC-V "V" Vector Extension*, https://github.com/riscv/riscv-v-spec, version 1.0, Sep. 2021.

[7] M. Johns and T. J. Kazmierski, "A Minimal RISC-V Vector Processor for Embedded Systems," in *2020 Forum for Specification and Design Languages (FDL)*, 2020, pp. 1–4. DOI: 10.1109/FDL50818.2020.9232940.

[8] V. N. Chander and K. Varghese, "A Soft RISC-V Vector Processor for Edge-AI," in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, 2022, pp. 263–268. DOI: 10.1109/VLSID2022.2022.00058.

[9] N. Kovačević, Đ. Mišeljić, and A. Stojković, "RISC-V vector processor for acceleration of machine learning algorithms," in *2022 30th Telecommunications Forum (TELFOR)*, 2022, pp. 1–4. DOI: 10.1109/TELFOR56187.2022.9983779.

[10] M. Ali, M. von Ameln, and D. Goehringer, "Vector Processing Unit: A RISC-V based SIMD Co-processor for Embedded Processing," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 30–34. DOI: 10.1109/DSD53832.2021.00014.

[11] V. Maisto and A. Cilardo, "A Pluggable Vector Unit for RISC-V Vector Extension," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1143–1148. DOI: 10.23919/DATE54114.2022.9774501.

[12] Free Software Foundation, Inc., *GCC, the GNU Compiler Collection*, version 13.1, May 23, 2023. [Online]. Available: https://gcc.gnu.org/ (visited on 05/29/2023).

[13] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*, 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 0123747503.

[14] Xilinx (AMD), *Vivado*, version 2023.1, 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html (visited on 05/29/2023).

[15] S. Chetlur, C. Woolley, P. Vandermersch, *et al.*, *Cudnn: Efficient primitives for deep learning*, 2014. arXiv: 1410.0759 [cs.NE].

[16] W. Commons, *File:2d convolution animation.gif — wikimedia commons, the free media repository*, [Online; accessed 4-April-2023], 2023. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:2D_Convolution_Animation.gif&oldid=726542205.

[17] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 4. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ISBN: 978-3-642-45309-0. DOI: 10.1007/978-3-642-45309-0.

[18] Xilinx Inc., *DSP48 Block*, Aug. 2018. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html (visited on 05/29/2023).

[19] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964. DOI: 10.1109/PGEC.1964.263830.

[20] L. Dadda, "Some schemes for fast serial input multipliers," in *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*, IEEE, 1983, pp. 52–59.

[21] M. Sjalander and P. Larsson-Edefors, "Multiplication acceleration through twin precision," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1233–1246, 2009. DOI: 10.1109/TVLSI.2008.2002107.

[22] S. Perri, P. Corsonello, M. Iachino, M. Lanuzza, and G. Cocorullo, "Variable precision arithmetic circuits for fpga-based multimedia processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 995–999, 2004. DOI: 10.1109/TVLSI.2004.833400.

[23] A. Alon, K. Asanović, A. Baum, *et al.*, *RISC-V "V" Vector Extension*, Sep. 2021. DOI: 10.1109/TVLSI.2019.2950087.

[24] Xilinx, Inc., *MicroBlaze Processor Reference Guide*, version UG984 (v2018.2), Jun. 2018. [Online]. Available: https://docs.xilinx.com/v/u/2018.2-English/ug984-vivado-microblaze-ref (visited on 05/30/2023).

[25] H. Wang, Z. Chen, K. Cheng, *et al.*, *RISC-V Vector Extension Intrinsic Document*, https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/master/examples/rvv_matmul.c, 2023.

A

Testbench Programs

Here we list the C code used to create the testbenching programs.

The programs were compiled using GCC [12] and with the following compiler flags:

```
-march=rv32imv -mabi=ilp32 -O3
```

Listing A.1: Compiler flags used for the benchmark programs.

A.1 Matrix Multiplication

The functions used are adapted from [25], provided by the RISC-V organization, to accommodate 8-bit operations for a 64×64 matrix multiplication, and are shown in Listing A.2. Note that the matrix multiplication utilizing vector instructions directly calls GCC intrinsic to perform the vector operations.

```
// matrix multiplication utilizing vector instructions
// A[n][o], B[m][o] --> C[n][m];
void matmul(char a[64][64], char b[64][64], char c[64][64],
            int n, int m, int o) {
  size_t vlmax = __riscv_vsetvlmax_e8m1();
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      char *ptr_a = &a[i][0];
      char *ptr_b = &b[j][0];
      int k = o;
      vint8m1_t vec_s = __riscv_vmv_v_x_i8m1 (0, vlmax);
      vint8m1_t vec_zero = __riscv_vmv_v_x_i8m1 (0, vlmax);
      for (size_t vl; k > 0; k -= vl, ptr_a += vl, ptr_b += vl) {
        vl = __riscv_vsetvl_e8m1(k);

        vint8m1_t vec_a = __riscv_vle8_v_i8m1 (ptr_a, vl);
        vint8m1_t vec_b = __riscv_vle8_v_i8m1 (ptr_b, vl);

        vec_s = __riscv_vmacc_vv_i8m1(vec_s, vec_a, vec_b, vl);
      }

      vint8m1_t vec_sum;
      vec_sum = __riscv_vredsum_vs_i8m1_i8m1(vec_s, vec_zero,
                                             vlmax);
      char sum = __riscv_vmv_x_s_i8m1_i8(vec_sum);
      c[i][j] = sum;
    }
  }
}

// matrix multiplication utilizing scalar instructions solely
void matmul(char a[64][64], char b[64][64], char c[64][64],
            int n, int m, int o) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j) {
      c[i][j] = 0;
      for (int k = 0; k < o; ++k)
        c[i][j] += a[i][k] * b[j][k];
    }
}
```

Listing A.2: Matrix multiplication of a 64×64 matrix. Adapted from [25].