ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMENTATION OF VECTOR EXTENSION ON RISC-V CORE USING SPINALHDL

SENIOR DESIGN PROJECT

Cemalettin Cem BELENTEPE
Yunus Emre ÇAKIROĞLU
Erinç Utku ÖZTÜRK

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JUNE 2023

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMENTATION OF VECTOR EXTENSION ON RISC-V CORE USING SPINALHDL

SENIOR DESIGN PROJECT

Cemalettin Cem BELENTEPE
040180255

Yunus Emre ÇAKIROĞLU
040190019

Erinç Utku Öztürk
04019102

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna Örs YALÇIN

Uygundur
Berna Örs Yalçın

[illegible]

JUNE 2023

We are submitting the Senior Design Project entitled as “Design And Implementation Of Vector Extension On VexRiscv Using SpinalHDL”. The Senior Design Project has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project Report by ourselves, and we have abided by the ethical rules with respect to academic and professional integrity .

Cemalettin Cem BELENTEPE
040180255

CBelentepe

Yunus Emre ÇAKIROĞLU
040190019

[illegible]

Erinç Utku ÖZTÜRK
040190102

[illegible]

FOREWORD

We are deeply grateful to Prof. Dr. Sıddıka Berna Örs Yalçın, our esteemed academic advisor, who has provided insightful guidance and unwavering mentorship throughout this project. Her sagacity and encouragement have opened us to new perspectives and learning opportunities, playing an instrumental role in shaping the direction and quality of our work.

Our heartfelt appreciation extends to Mr. Emre Göncü, our advisor from the industry. His inputs has considerably enriched our project, adding a layer of practical understanding to our theoretical knowledge.

We also wish to express our thanks to Mr. Yasin Yılmaz, whose assistance and commitment played a pivotal role in our project's success.

Lastly, we wish to express our sincere gratitude to Dolu1990, the creator of SpinalHDL. Without his innovative creation, this project would not have been feasible. His groundbreaking work has served as an inspiration and motivation for us, illuminating the path we tread in our exploration.

June 2023

Cemalettin Cem BELENTEPE
Yunus Emre ÇAKIROĞLU
Erinç Utku ÖZTÜRK

TABLE OF CONTENTS

ABBREVIATIONS

RISC	: Reduced Instruction Set Computer
CISC	: Complex Instruction Set Computer
HDL	: Hardware Description Language
ISA	: Instruction Set Architecture
VLEN	: Vector Length
CSR	: Control Status Register
LMUL	: Length Multiplier
EMUL	: Effective Length Multiplier
SEW	: Selected Element Width
EEW	: Effective Element Width

LIST OF TABLES

Page

LIST OF FIGURES

DESIGN AND IMPLEMENTATION OF VECTOR EXTENSION ON RISC-V CORE USING SPINALHDL

SUMMARY

The objective of this project is to develop and implement a RISC-V Vector Extension compliant core. For this purpose, SpinalHDL has been chosen as the design language. The selection of SpinalHDL serves two purposes: firstly, to evaluate the language's performance in handling a complex design, and secondly, to assess the advantages and benefits it offers in the design process.

Initially, all instructions outlined in the official set are categorized based on their intended purposes. Subsequently, in order to reduce the area occupied by the design, the utilization of multiple microinstructions is implemented. To achieve this goal, a dedicated decoder unit consisting of two stages is proposed. The first stage is responsible for converting instructions into microinstructions, while the second stage focuses on decoding these microinstructions.

The execution data path is designed with the maximum element width of 64-bit as the primary consideration. The proposed design ensures scalability in terms of this maximum element width, allowing for parallel processing to reduce the number of cycles required to complete an instruction. However, this scalability comes at the cost of increased area utilization.

Unit cells in the execute path are modules that have 64-bit data input, which is the maximum element width. This data can be utilized as an array of various data sizes, including 8-bit, 16-bit, 32-bit, or 64-bit. The cells in the design are compact and optimized to exploit the properties of the instruction, enabling their efficient implementation for this particular data type.

By employing this design strategy, the vector instruction set instructions are implemented and designed to interface with an RV32I or RV64I core. In order to test the design, it is presently directly integrated into a small RV32IM design with a modified MIPS architecture.

DESIGN AND IMPLEMENTATION OF VECTOR EXTENSION ON RISC-V CORE USING SPINALHDL

ÖZET

Bu proje, RISC-V Vektör Uzantısı standartlarına uygun bir çekirdek geliştirmeyi ve uygulamayı amaçlamaktadır. Bu amaçla, tasarım dili olarak SpinalHDL seçilmiştir. SpinalHDL'nin seçimi iki amaç için yapılmıştır: birincisi, dilin karmaşık bir tasarımla başa çıkma performansını değerlendirmek; ikincisi ise tasarım sürecinde sunduğu avantajları ve faydaları değerlendirmektir.

İlk olarak, resmi sette belirtilen tüm buyruklar amaçlarına göre kategorilere ayrılmıştır. Ardından, tasarımda kapladığı alanı azaltmak için birden fazla mikro-buyruk kullanımı uygulanmıştır. Bu hedefe ulaşmak için, iki aşamadan oluşan bir özel bir kod çözücü birimi önerilmiştir. İlk aşama, talimatları mikro-buyruklara dönüştürmekten sorumlu iken, ikinci aşama bu mikro-buyrukları çözümlemeye odaklanmaktadır.

Yürütme veri yolu, maksimum 64 bit veri genişliği temel alınarak tasarlanmıştır. Önerilen tasarım, bu maksimum veri genişliği açısından ölçeklenebilirlik sağlar ve talimatı tamamlamak için gereken döngü sayısını azaltmak için paralel işleme imkanı sunar. Ancak, bu ölçeklenebilirlik, artan alan kullanımı maliyetiyle birlikte gelir.

Çalışma yolundaki birim hücreleri, maksimum 64 bit veri girişine sahip modüllerdir. Bu veri, 8 bit, 16 bit, 32 bit veya 64 bit dahil olmak üzere çeşitli veri boyutları dizisi olarak kullanılabilir. Tasarımdaki hücreler, talimatın özelliklerinden yararlanmak için optimize edilmiş ve kompakt bir yapıya sahiptir, böylece bu belirli veri türü için verimli bir şekilde uygulanabilir.

Bu tasarım stratejisi kullanılarak, vektör talimat kümesi talimatları bir RV32I veya RV64I çekirdekle etkileşimli olarak uygulanmış ve tasarlanmıştır. Tasarımı test etmek için, şu anda değiştirilmiş bir MIPS mimarisiyle küçük bir RV32IM tasarımına doğrudan entegre edilmektedir.

1. INTRODUCTION

1.1 Purpose of Project

The purpose of the project is designing and implementing a circuit to accelerate cryptology and artificial intelligence applications. Our objective is implementing integer and fixed-point vector commands while complying to the Vector Extension Spec 1.0 document [1]. The currently available solutions make customizing circuits a lot harder. The HDL language of our choice which is SpinalHDL [2] will allow us to implement our circuit in a more modular manner. This way whenever someone wants to contribute to the circuit, they will not have to change all the circuit.

The reason we have chosen this problem is our interest in computer architecture and the fact that RISC-V Vector Extension Spec 1.0 was not implemented before.

We gathered our background information from our Computer Organization, Microprocessor Systems, Introduction to Embedded Systems, Computer Architecture, Logic Lab, Microprocessor Lab courses, previous internship experiences and our personal works.

1.2 Literature Review

Our project started with the idea of implementing RISC-V Vector Instruction Set Architecture (ISA) [1] based on newly proposed release candidate specification. To implement ISA, we needed a baseline open-source RISC-V implementation. After several researches we concluded that an open-source RISC-V implementation written in SpinalHDL [2] suits best for our implementation. SpinalHDL is an open-source high-level hardware description language (HDL) based on Scala [3]. It can be used as an alternative to VHDL or Verilog and has several advantages over them:

- It focuses on efficient hardware description instead of being event-driven.
- It is embedded into a general-purpose programming language, enabling powerful hardware generation.

In this manner we have chosen RV32IM implementation based on SpinalHDL called VexRiscv [5]. VexRiscv takes an object-oriented approach in implementation. Modular “plugin” system eases things like adding new instructions to decode stage. This allows us to focus on Vector Instruction Set Architecture implementation.

2. BACKGROUND INFORMATION

2.1 What is RISC-V?

RISC-V is an instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computer (RISC) architecture. It follows an open standard, allowing anyone to use, modify, and distribute it freely without royalty payments. The ISA encompasses various aspects of programming in a computer architecture, including data types, instructions, registers, addressing modes, memory organization, interrupt and exception handling, and external input/output operations.

What sets RISC-V apart from most other ISAs is its open-source nature. This means that it is available under open-source licenses, promoting accessibility and encouraging collaboration. RISC-V is designed with modularity and extensibility in mind. It provides a foundation of base instructions while allowing for additional instruction set extensions to cater to specialized workloads.

The base instruction set in RISC-V consists of fixed-length 32-bit instructions. This simplicity and consistency enable compatibility across different implementations. RISC-V is versatile and adaptable, capable of supporting a wide range of devices and applications, from small embedded systems to personal computers and large supercomputers.

One of the advantages of the RISC-V architecture is its support for different address spaces. It can accommodate 32-bit, 64-bit, and potentially even 128-bit address spaces. This capability enables efficient handling of large amounts of data and memory.

As a RISC architecture, RISC-V adheres to the philosophy of simplicity. This approach contributes to lower costs, higher processing speeds, and improved power efficiency. It follows a load-store architecture, which separates data processing operations from load and store operations, enhancing the overall performance and efficiency of the system.

2.1.1 RISC-V Instruction Sets

In computer architecture, an instruction set, also known as instruction set architecture (ISA), is the part of the computer's brain or processor that is responsible for programming. It is a protocol that the hardware understands and follows to execute commands coming from the software.

In the case of RISC-V, the instruction sets are based on the Reduced Instruction Set Computer (RISC) principles, which emphasize efficiency in cycles per instruction (CPI). RISC design philosophy aims to minimize the complexity and number of instruction sets to improve performance, speed, and efficiency.

RISC-V architecture consist of multiple instruction set architectures. Most notable ones are:

- **Base Integer ISA**: This is the foundational part of the RISC-V ISA, which comes in three variants: RV32I, RV64I, and RV128I, designed for 32-bit, 64-bit, and 128-bit (theoretical) architectures respectively. Each variant defines a set of registers, the basic RISC-V instructions, and the control flow instructions.
- **Standard Extensions**: To increase functionality while keeping the base ISA stable and compact, RISC-V includes several standard extensions. The ratified standard extensions included "M" for integer multiplication and division, "A" for atomic instructions to support multithreading, "F" and "D" for single and double-precision floating-point instructions respectively, and "C" for compressed instructions that provide more compact code size. Additional extensions are under development.
- **User-level ISA**: This defines the instructions available to user-level software. It includes the base instruction set and the standard extensions.
- **Privileged ISA**: This defines additional instructions and functionality for privileged software, such as operating system kernels, hypervisors, and firmware.
- **Variable Length Extensions**: The base instruction set has a fixed length of 32-bit instructions, and the ISA supports variable-length extensions where each instruction can be of any number of 16-bit parcels in length.

2.1.2 RISC-V Standard Vector Extension

The RISC-V Standard Vector Extension, often referred to as the "Vector Extension" or "V Extension," is an addition to the core RISC-V instruction set that facilitates vector operations. These operations allow a single instruction to be performed on multiple data points simultaneously, which can significantly enhance performance for certain types of tasks, including digital signal processing, multimedia tasks, and machine learning.

What sets the RISC-V Vector Extension apart from vector instructions in many other architectures is its flexibility and scalability. Rather than having a fixed vector length embedded into the architecture, RISC-V allows the vector length to be configured at runtime. This means that the hardware can set the most efficient vector length based on its capabilities, and software can be written to maximize the available vector length.

Another notable feature of the RISC-V Vector Extension is full support for vector predication. This allows any vector operation to be selectively applied to individual elements in the vector, based on a separate mask vector. As a result, it can eliminate the need for many conditional branches in the code, enhancing code efficiency.

The Vector Extension is also equipped with operations that mix a vector with a single scalar value, in addition to operations on whole vectors. These vector-scalar operations are often useful in practical code. Moreover, it supports flexible memory access patterns, including strided and indexed loads and stores, greatly improving memory efficiency for a variety of workloads.

2.2 What is SpinalHDL?

SpinalHDL, a high-level hardware description language (HDL), presents several advantages over traditional HDLs such as VHDL or Verilog, making it an attractive choice for digital design.

Firstly, SpinalHDL employs a high-level abstraction, allowing for a more abstract and concise description of logic circuits. This feature enhances the readability and maintainability of the code, thereby increasing the efficiency of the design process.

Secondly, SpinalHDL leverages the features of the Scala programming language, including its support for object-oriented and functional programming. This leads to more modular and reusable code, further enhancing the efficiency of the design process.

Thirdly, SpinalHDL supports generics and parameterization, which can be used to create more flexible and reusable components. This feature enhances the versatility of the language, allowing for a wider range of design possibilities.

Furthermore, SpinalHDL has a strong typing system which can catch errors at compile-time rather than at runtime. This feature enhances the robustness of the design process, ensuring the reliability of the final product.

Finally, since SpinalHDL is built on Scala, it can leverage the Scala ecosystem, including its build tools, testing frameworks, and libraries. This feature enhances the adaptability of the language, allowing it to be used in a wider range of contexts.

In conclusion, SpinalHDL offers a compelling alternative to traditional HDLs, providing capabilities that VHDL and Verilog lack, while meeting or exceeding their performance and efficiency. By providing a high level of abstraction and leveraging the features of the Scala programming language, SpinalHDL enhances the efficiency, versatility, robustness, and adaptability of the digital design process.

2.3 Design and Test Environment

The design process involves three levels and utilizes three distinct tools, with testing conducted using the same set of tools. The tools are IntelliJ IDE with SpinalHDL, Xilinx Vivado with Verilog, and Visual Studio Code with OpenLane.

SpinalHDL is employed for the design itself, as well as for unit testing and integration testing. The rationale behind this choice is elaborated in other sections. The IntelliJ IDE is utilized as the development and testing environment due to its support for SpinalHDL.

To examine the compiled Verilog code generated by SpinalHDL, the RTL Schematic view of Xilinx Vivado is employed. This tool is also used for synthesis, implementation, and synthesis and implementation simulations. The preference for Vivado stems from the fact that the available FPGA kits for our project are Xilinx boards.

Furthermore, following FPGA implementations, OpenLane is utilized to verify the ASIC implementation of the design.

3. VECTOR EXTENSION ON RISC-V CORE USING SPINALHDL

SpinalHDL is a domain-specific language (DSL) built on Scala programming language. Scala programming language is a scalable general-purpose language with strong and static typing. This language is designed to support mainly object-oriented design and functional design paradigms [3]. SpinalHDL itself is a hardware description language (HDL). HDLs are languages designed for modeling hardware on dataflow, behavioral, or structural levels. Dataflow modeling allows the writer to describe their circuit on the gate level, using given operators such as 'and', 'or', 'assign', '+', and so on. On the other hand, behavioral modeling allows the writer to describe the behavior of the circuit with 'if - else', 'case', 'for', 'always@', and such keywords so the design process can be faster, and easier to both design and understand. Lastly, the structural design method allows the user to design modules by combining other modules that are written using any of the mentioned design methods. Examples of conventional HDLs are Verilog, VHDL, and System Verilog languages.

SpinalHDL also supports these modeling methods, but it differs by enhancing the support on structural design and changing the paradigm from event-driven to Scala's OOP and FP paradigms. SpinalHDL also allows its user to use all the features of Scala, so the circuit is much more modular and easier to combine than conventional HDLs. This results in faster and easier development with a paradigm more suitable for digital hardware description [4]. After the hardware is described using SpinalHDL, it is then compiled into Verilog or VHDL. Therefore, the rest of the development process can continue as it was before.

For this purpose, we also designed a simple RISC-V core called FlexiRISC. It is written and vitrificated on SpinalHDL. It is capable of RV32IM instruction sets. We have chosen to write a RISC-V core from scratch since we wanted a core that leverages SpinalHDL's features.

We also utilized SpinalHDL on our Vector extension for given benefits.

3.1 Proposed Vector Extension Architecture

The modified architecture being proposed is an adaptation of the MIPS architecture that incorporates the vector extension. When considering the vector extension in RISC-

V architecture, a notable deviation from the instructions found in the base integer set and other extensions is observed. Contrary to the instruction in the base integer set and the other RISC-V extensions, the vector extension works with wider data. However, this wide execute datapath would result in a larger overall size. To address this issue, a multicycle decoder is introduced to optimize the area occupied by the datapath. Block diagram of this architecture is shown in Figure **Error! No text of specified style in document.**.1.

The execute datapath is designed to be configurable, allowing the width of the datapath to be adjusted as a multiple of 64 bits which is the maximum ELEN for a vector instruction. The decoder is tasked with issuing the correct microinstructions for a vector instruction to be executed.

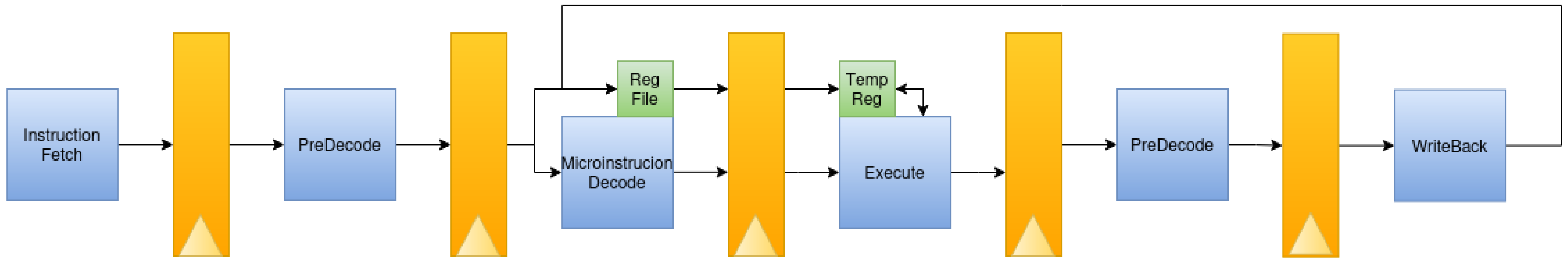


Figure Error! No text of specified style in document.**.1** Proposed Pipeline Architecture

3.2 Vector Register File and Control Status Registers

Vector Extension Spec [1] describes two sets of registers namely vector register file and control status registers. This section describes the design and the implementation of those register sets.

3.2.1 Control Status Registers

Control Status Registers (CSRs) are used for containing information about the state of the extension module [1, p. 9]. There are 7, XLEN-bit CSRs named ‘vstart’, ‘vxsat’, ‘vxrm’, ‘vcsr’, ‘vl’, ‘vtype’, and ‘vlenb’.

The CSR ‘vstart’ holds the information about the position of the element that will be processed next in the next instruction. This could be thought of as analogous to the starting value of the iterator in a ‘for’ loop.

The CSRs ‘vxsat’ and ‘vxrm’ holds the information for the saturation mode and rounding mode of the fixed value instructions accordingly.

The CSR ‘vcsr’ contains the ‘vxsat’ and ‘vxrm’ registers.

The CSR ‘vl’ shows how many elements are affected by a single vector instruction. The value of the register will only change after a ‘vset{i}vl{i}’ instruction.

The CSR ‘vtype’ is set by the configuration instructions and used for holding the information about the register grouping (LMUL) and size of an element (EEW) along with some implementation values.

Lastly, the ‘vlenb’ register is a constant value, showing the length of a physical register in bytes.

3.2.2 Vector Registers

Vector register file of the architecture has 32 of the VLEN width registers. These registers are also addressable by the datapath length. Figure **Error! No text of specified style in document.**.2 illustrates an example that showcases a VLEN of 256 and a datapath width of 64.

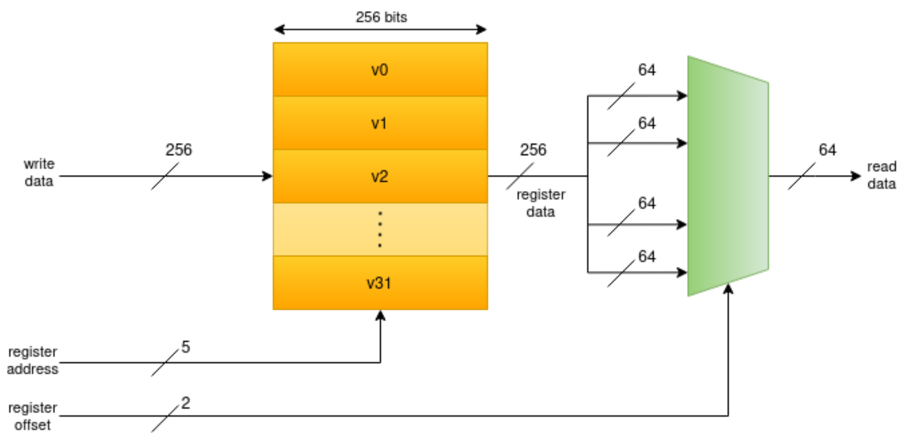


Figure Error! No text of specified style in document.**.2** Conceptual design of the vector register file

While the conceptual design of the vector register file resembles Figure **Error! No text of specified style in document.**.2, the actual implementation follows the approach depicted in Figure **Error! No text of specified style in document.**.3. This approach offers a more compact and simpler design, as it utilizes a single block RAM with a narrower width instead of employing both a block RAM and multiplexers. For the sake of simplicity, the write mask and the second read port are not depicted in the figure.

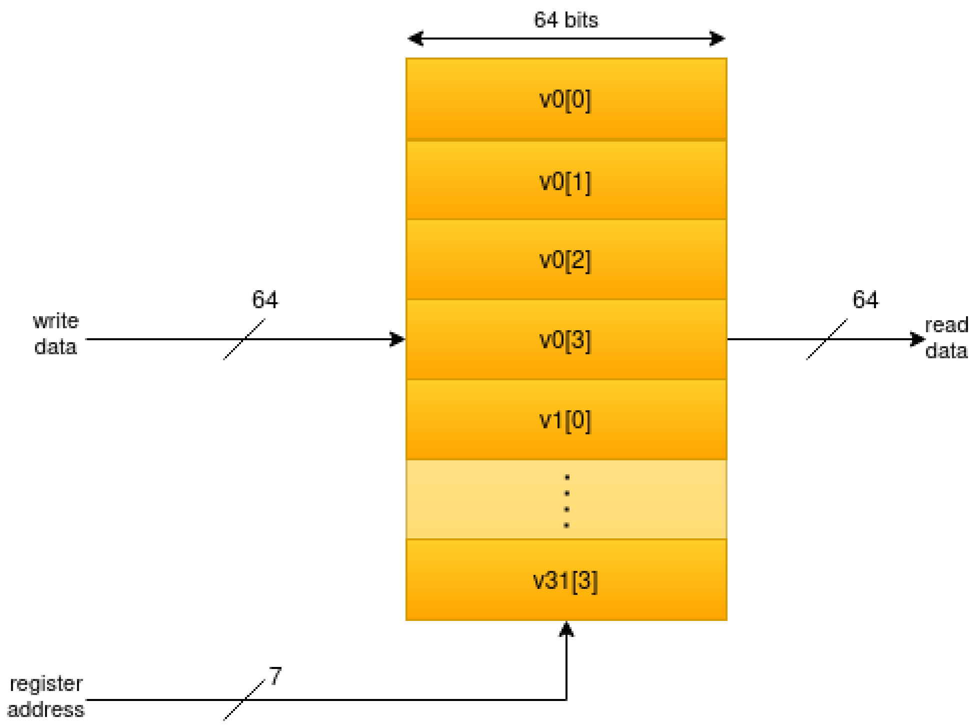


Figure Error! No text of specified style in document.**.3** Proposed vector register file

3.3 Configuration Setting Instructions

Vector Instruction Set consists of highly configurable instructions. These configurations are determined by Control Status Registers (CSR) which are configured by Configuration Setting Instructions. Most of the instructions make use of “Stripmining”

"Stripmining" is a technique for efficiently processing large numbers of elements by dividing the work into multiple iterations. In the context of the RISC-V Vector Specification, the application specifies the total number of elements to be processed (the "application vector length" or AVL) and the hardware responds with the number of elements it can handle in each iteration, stored in the "vl" register. This value is based on the microarchitectural implementation and the vtype setting. A loop structure can be used to keep track of the remaining number of elements and the number of elements handled by the hardware in each iteration.

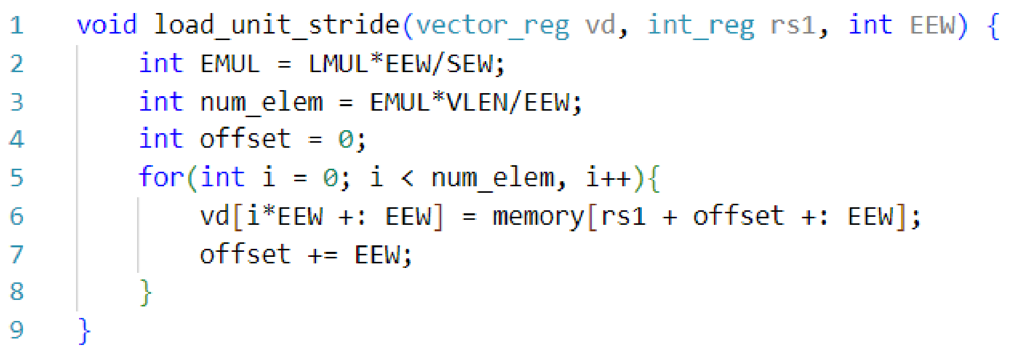
The RISC-V Vector Specification includes a set of instructions that allow the values in the "vl" and "vtype" control status registers to be quickly configured to match the needs of the application. These instructions, called "vset{i}vl{i}", set the "vtype" and "vl" control and status registers based on their arguments and write the new value of "vl" into the "rd" register. These instructions can be used to quickly adjust the configuration of the vector unit to suit the needs of different applications or parts of the same application.

These instructions configure Selected element width (SEW) setting and Vector register group multiplier (LMUL) setting field in vtype register which used in stripmining.

We have implemented Configuration Setting Instructions as proposed on RISC-V Vector Specification. Implementing these instructions was crucial since most of instructions are configured via related CSR such as Vector type register, vtype and Vector length register, vl.

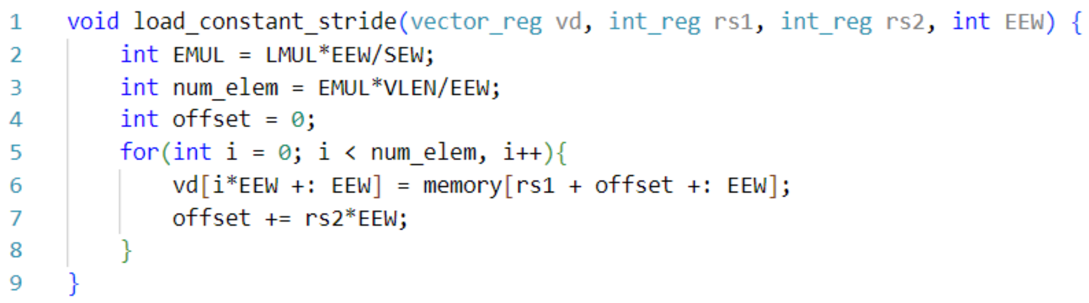
3.4 Load and Store Instructions

Vector Extension Spec [1] describes load and store instructions. These instructions are used to transfer data from or to the vector registers. According to the spec, the instructions have three main versions, unit stride, constant stride, and indexed. For the algorithms of each version, pseudocode is written to explain. Pseudocodes for load unit stride, constant stride, and indexed version can be seen in the following figures, Figure **Error! No text of specified style in document.**.4, Figure **Error! No text of specified style in document.**.5, Figure **Error! No text of specified style in document.**.6.



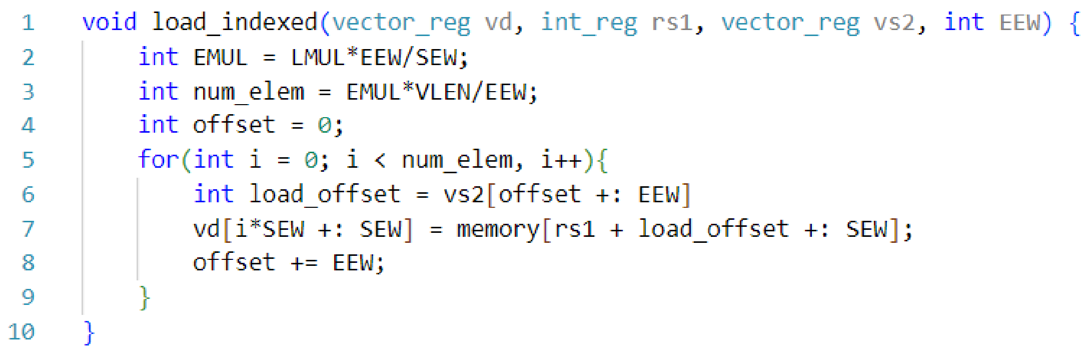
```
void load_unit_stride(vector_reg vd, int_reg rs1, int EEW) {
    int EMUL = LMUL*EEW/SEW;
    int num_elem = EMUL*VLEN/EEW;
    int offset = 0;
    for(int i = 0; i < num_elem, i++){
        vd[i*EEW +: EEW] = memory[rs1 + offset +: EEW];
        offset += EEW;
    }
}
```

Figure Error! No text of specified style in document.**.4** Pseudocode for the unit stride load instruction



```
void load_constant_stride(vector_reg vd, int_reg rs1, int_reg rs2, int EEW) {
    int EMUL = LMUL*EEW/SEW;
    int num_elem = EMUL*VLEN/EEW;
    int offset = 0;
    for(int i = 0; i < num_elem, i++){
        vd[i*EEW +: EEW] = memory[rs1 + offset +: EEW];
        offset += rs2*EEW;
    }
}
```

Figure Error! No text of specified style in document.**.5** Pseudocode for the constant stride load instruction



```
void load_indexed(vector_reg vd, int_reg rs1, vector_reg vs2, int EEW) {
    int EMUL = LMUL*EEW/SEW;
    int num_elem = EMUL*VLEN/EEW;
    int offset = 0;
    for(int i = 0; i < num_elem, i++){
        int load_offset = vs2[offset +: EEW]
        vd[i*SEW +: SEW] = memory[rs1 + load_offset +: SEW];
        offset += EEW;
    }
}
```

Figure Error! No text of specified style in document.**.6** Pseudocode for the indexed load instruction

After the above-mentioned pseudocodes are generated for the load instructions, by switching the data flow from memory to vector register to vector register to memory, store instructions are also designed. In the pseudocode, EEW denotes the effective element width of a vector, which is the SEW at the time of load/store. EMUL decodes the LMUL at the time of load/store. Those are connected to the register file with the select-by-element mode so that the correct element of the correct register can be selected for load/store operations. Since both load and store instruction has the same addressing modes, an address generator module is created and used for both of the instructions. This module takes the addressing mode along with the rs2 and vs2 registers and returns the correct address to work with. The pseudocode of the address generator algorithm is shown in Figure **Error! No text of specified style in document.**.7.

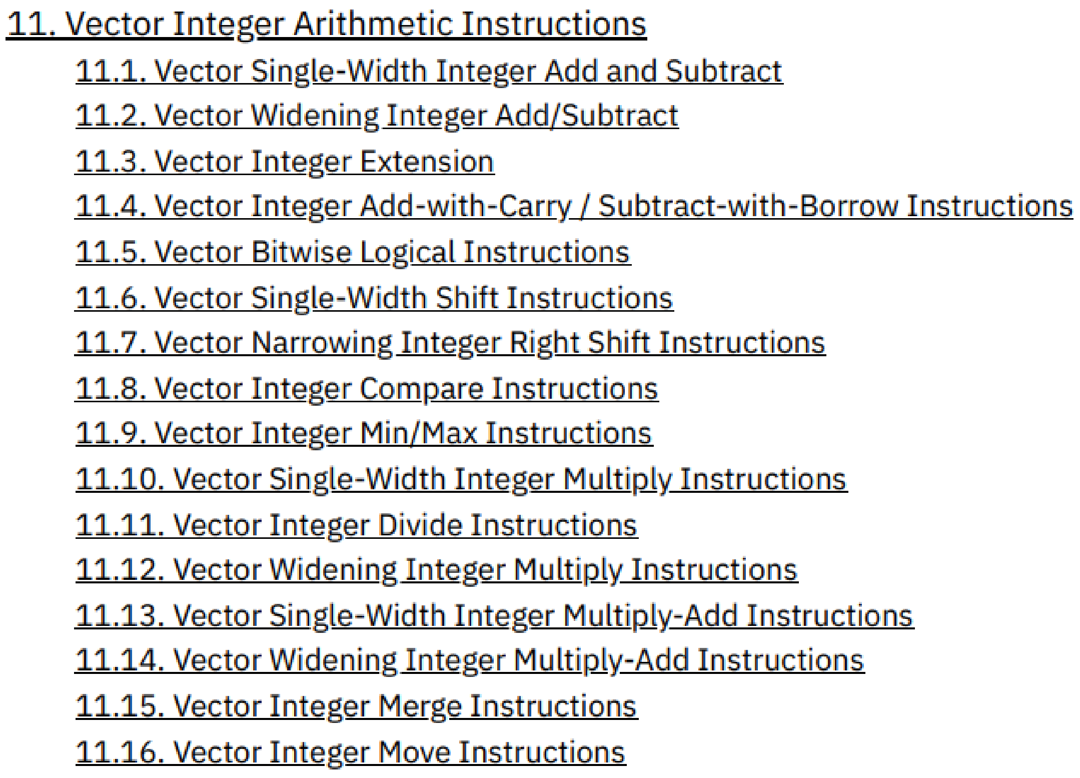
```
uint32_t address_generator(
    int addressing_mode,
    int_reg rs1,            // Base address register
    int_reg rs2,            // Constant stride register
    vector_reg vs2,         // Index register
    int EEW                 // Effective element width
) {
    static int offset = 0; // offset will be 0 at the start
    uint32_t address;

    switch(addressing_mode) {
        // Unit stride
        case 0:
            address = rs1 + offset;
            offset += EEW;
            break;

        // Constant stride
        case 2:
            address = rs1 + offset;
            offset += rs2*EEW;
            break;

        // Indexed
        case 1:
        case 3:
            address = rs1 + vs2[offset +: SEW];
            offset += EEW;
            break;
    }

    return address;
}
```

Figure Error! No text of specified style in document.**.7** Pseudocode for address generator unit

The microinstructions are generated based on the address generator algorithm and the load-store instruction outlined in the vector extension specification [1]. These microinstructions share the same opcodes as the base integer set, but the register fields differ depending on whether they belong to the vector set or the integer set, depending on the specific instruction being executed.

3.5 Arithmetic Instruction

As shown below in the Figure 2.5.1, There are 16 arithmetic instructions that are specified in the RISC-V Vector Extension ISA specification. [1]



11. Vector Integer Arithmetic Instructions

11.1. Vector Single-Width Integer Add and Subtract
11.2. Vector Widening Integer Add/Subtract
11.3. Vector Integer Extension
11.4. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions
11.5. Vector Bitwise Logical Instructions
11.6. Vector Single-Width Shift Instructions
11.7. Vector Narrowing Integer Right Shift Instructions
11.8. Vector Integer Compare Instructions
11.9. Vector Integer Min/Max Instructions
11.10. Vector Single-Width Integer Multiply Instructions
11.11. Vector Integer Divide Instructions
11.12. Vector Widening Integer Multiply Instructions
11.13. Vector Single-Width Integer Multiply-Add Instructions
11.14. Vector Widening Integer Multiply-Add Instructions
11.15. Vector Integer Merge Instructions
11.16. Vector Integer Move Instructions

Figure Error! No text of specified style in document.**.8** List of instructions specified in the RISC-V Vector Extension ISA specification

So far, a circuit for addition of vectors of different length has been designed and implemented. This circuit is the basis for the implementation of the following:

- Vector Single-Width Integer Add and Subtract
- Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions

A new adder has been designed for the cumulative addition operation. Those designs will be described further in the following subsections. So far, the circuitry that will be the basis for the instructions are being designed. In the following weeks those building blocks will be combined to implement arithmetic instructions.

3.5.1 Vector Adder Unit

The specs of the RISC-V require an addition unit that can support different EEW operations. Addition operation with different EEW's is show in the Table **Error! No text of specified**

style in document..1. To ease of readability, Table **Error! No text of specified style in document.**.2 is given to see mapping of elements. Tables are written for vlen of 64 bits. To achieve the target vlen VLEN/64 of those modules are put in parallel configuration.

Table Error! No text of specified style in document.**.1** Addition configurations with different EEW's.

EEW	configuration							
8	a7+b7	a6+b6	a5+b5	a4+b4	a3+b3	a2+b2	a1+b1	a0+b0
16	a3+b3		a2+b2		a1+b1		a0+b0	
32	a1+b1				a0+b0			
64	a0+b0							

Table Error! No text of specified style in document.**.2** Bit spans of the elements in the table above

EEW	Representation of bytes with units
8	ai=a[8*(i+1)-1,8*i], bi=b[8*(i+1)-1,8*i]
16	ai=a[16*(i+1)-1,16*i], bi=b[16*(i+1)-1,16*i]
32	ai=a[32*(i+1)-1,32*i], bi=b[32*(i+1)-1,32*i]
64	ai=a[64*(i+1)-1,64*i], bi=b[64*(i+1)-1,64*i]

A top module that can perform addition, subtraction, and comparison instructions is designed. Two's complement is taken by inversing B input and adding 1 to carry-in ports of adders. When performing subtraction carry-in inputs of designated adders are set to one. More information will be given about the logic to set carry-in inputs to one will be given in proceeding pages.

To meet the requirements stated above, a carry select adder with block size of 8 is generated. By following our design policy first, a 64 bits wide adder is generated. Then VLEN/64 of the adders are generated in parallel configuration. Choice of carry select adder is done with the consideration of speeds and ease of integration. In addition to this, the base 64-bit adder must be able to generate N, V, C, Z flags from 7, 15 ,23 ,31 ,39 ,47 ,55 and 63rd bits. The reasoning behind the generation of these flags will be explained shortly after. With addition of an and gate between the 8-bit adder blocks, carry propagation can be blocked and desired addition configurations are achieved. Carry in set signal that is required to perform subtraction operations is or-ed in every stage. A top-level view of 64-bit adder unit can be seen in Figure **Error! No text of specified style in document.**.9.

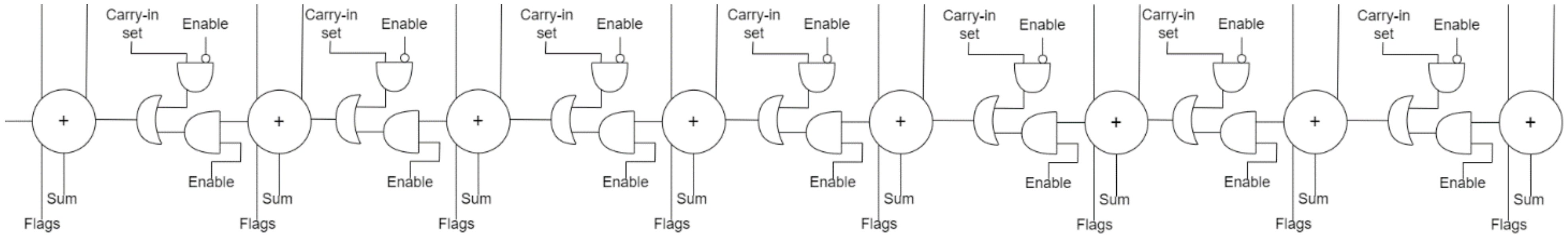


Figure Error! No text of specified style in document.**.9** Top level view of Adder

Every adder present in the figure is 8 bits wide. Every enable and carry in signals are generated according to the EEW. Propagation of carryout will be cut by setting enable to 0 when needed and carry in set will overwrite the carry out of previous stage if needed. A sample demonstrating the state of enable signal for rightmost adder is given in Table **Error! No text of specified style in document.**.3.

Table Error! No text of specified style in document.**.3** Enable signal table.

EEW	8	16	32	64
Enable	0	1	1	1
Carry-in set	1	0	0	0

To perform compare instructions the flag outputs of every adder block is used. When a compare instruction is set to perform; adder, performs subtraction operation. In this case flags will be generated on every 8*i^{th} bit. 8 sets of flags will be used to generate 8 comparison bits, and resulting comparison bits will be used in write-back stage to write the desired element. With this set-up comparison of every element is achieved without using the already existing adder.

3.5.2 Cumulative Operation Circuit

A cumulative adder circuit is designed for later usages in narrowing additions. This circuit will be a basis for dot product operation and other narrowing operations. The schematic of the circuit is given in Figure 2.5.2.1. Note that every circle in the diagram represents a small ALU like unit. The width of the unit is given in the number indicated inside the circle. Green line represents the copy of the circuit starting from the 128-bit adder. The second branch of the circuit is not drawn to save space and to avoid cumbersome representation.

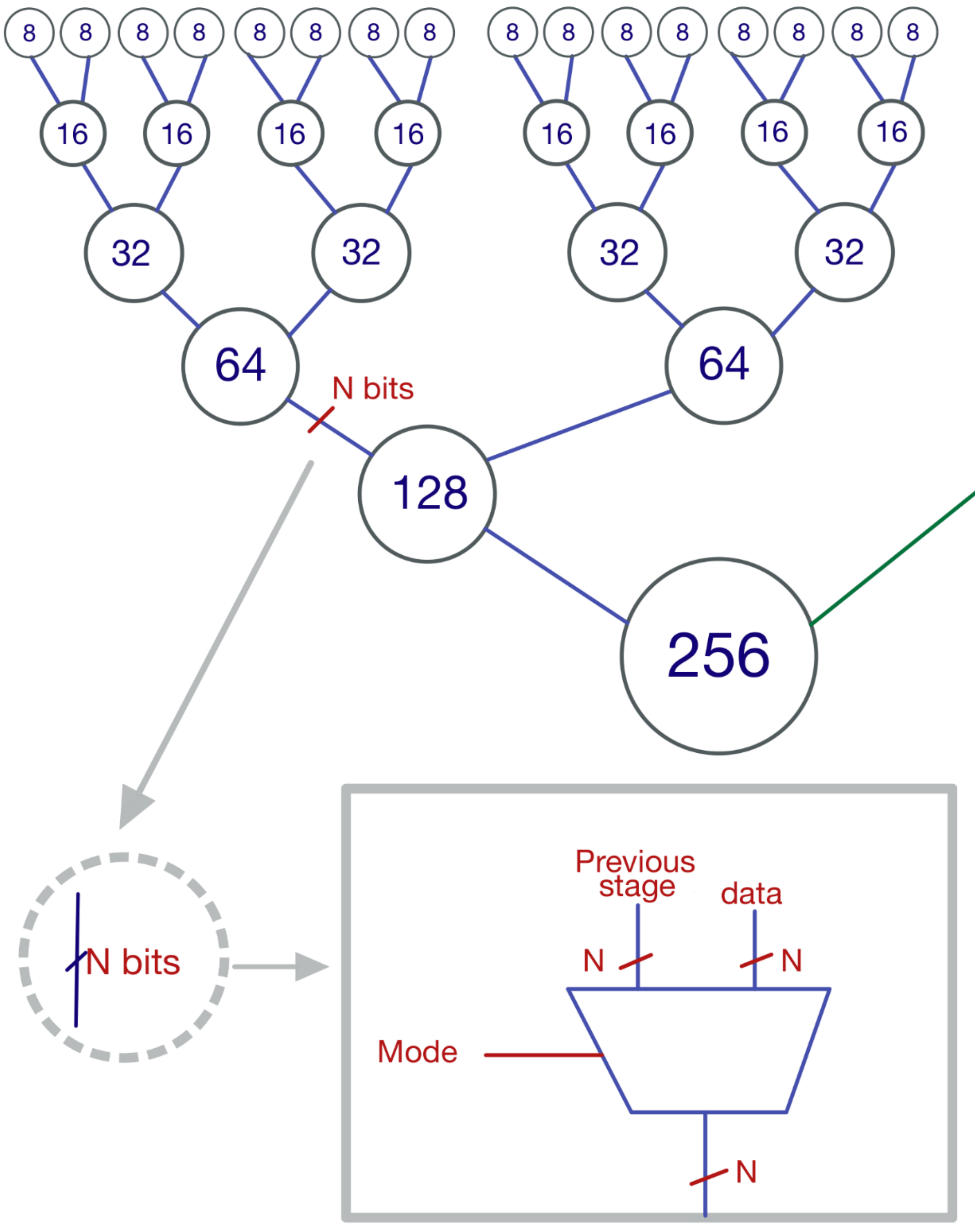


Figure Error! No text of specified style in document.**.10** Schematic of the cumulative adder

Blue lines in the circuit schematic represent a multiplexer. An instance of blue line is taken in Figure **Error! No text of specified style in document.**.10 in the circle on bottom left. More insight of this line is given in the rectangle on bottom right. Depending on the EEW, either data or the sum from the previous stage is passed to the next stage. 8-bit units on the top always receive 8 bits of data without a multiplexer. With the multiplexers operation can start directly from the corresponding element width.

For instance, if we are to make an addition by a vector with element width of 32 bits, the muxes inside the blue lines indicated in Figure 2.5.2.2 will be selecting data. Because we have fed data into those muxes the results of 8- and 16-bit adders will not be used. Addition of the element will start from the 32-bit adders and their results will be propagates until the 256-bit wide adder.

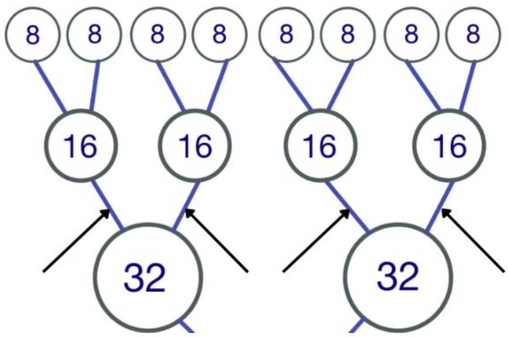


Figure Error! No text of specified style in document.**.11** An instance for addition

The schematic of the ALU like units is presented in Figure **Error! No text of specified style in document.**.11.

3.5.3 Shifter Circuit for Vector Single-Width Shift Instructions

Vector shifter needs to be able to shift every vector element by any arbitrary value. To achieve this, a set-up of 8-bit barrel shifters is implemented. To achieve the desired behavior, a barrel shifter like block is implemented. The mentioned block will be called pseudo-shifter.

To demonstrate the behavior of the design right shift operation will be mentioned, but to perform left shift a mirror-copy of the circuit is implemented. Pseudo-shifter takes shift amount and operand as inputs and will put the least significant “shift amount” of bits onto the most significant “shift amount” output bits. The remaining least significant bits will be 0. Truth table for 4-bit pseudo-shifter is given in Table **Error! No text of specified style in document.**.4. n-bit pseudo-shifter gives the same behavior in n bits.

Table Error! No text of specified style in document.**.4** Truth table for 4-bit pseudo-shifter

Shift Amount	y_3	y_2	y_1	y_0
0	0	0	0	0
1	x_0	0	0	0
2	x_1	x_0	0	0
3	x_2	x_1	x_0	0
4	x_3	x_2	x_1	x_0

The shifter selects between two sets of outputs. One set of output is generated by 16, 16-bit wide parallel shifters. These 16-bit shifters can work as one 16-bit shifter or 2 individual 8-bit shifters. Work dynamics will be mentioned soon. Similarly, the other set of output is generated by 4, 32-bit wide shifters. Similarly, this shifter can work as a 64-bit shifter or two 32-bit wide individual shifters. Set of 4 64-bit shifters and set of 16 16-bit shifters are illustrated in Figure **Error! No text of specified style in document.**.12 and Figure **Error! No text of specified style in document.**.13.

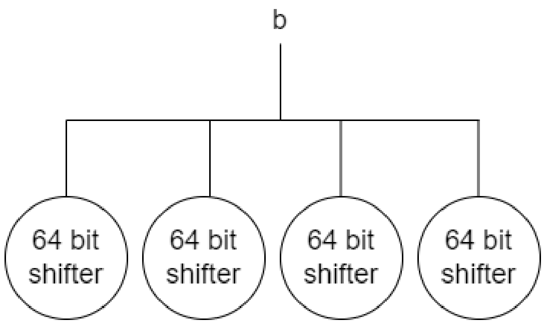


Figure Error! No text of specified style in document.**.12** Set of 64 bit shifters

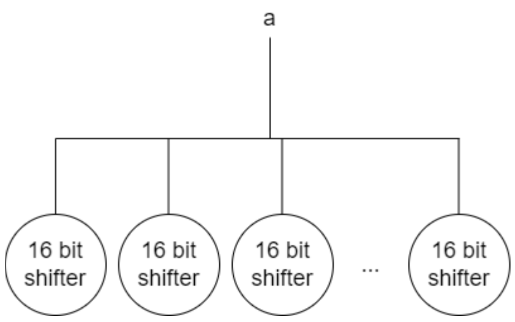


Figure Error! No text of specified style in document.**.13** Set of 16 bit shifters

A mux chooses between the outputs a and b depending on EEW[1].

3.5.3.1 16-bit Right shifter

Top-down design of the 16-bit right shifter is presented in Figure 2.5.3.1.

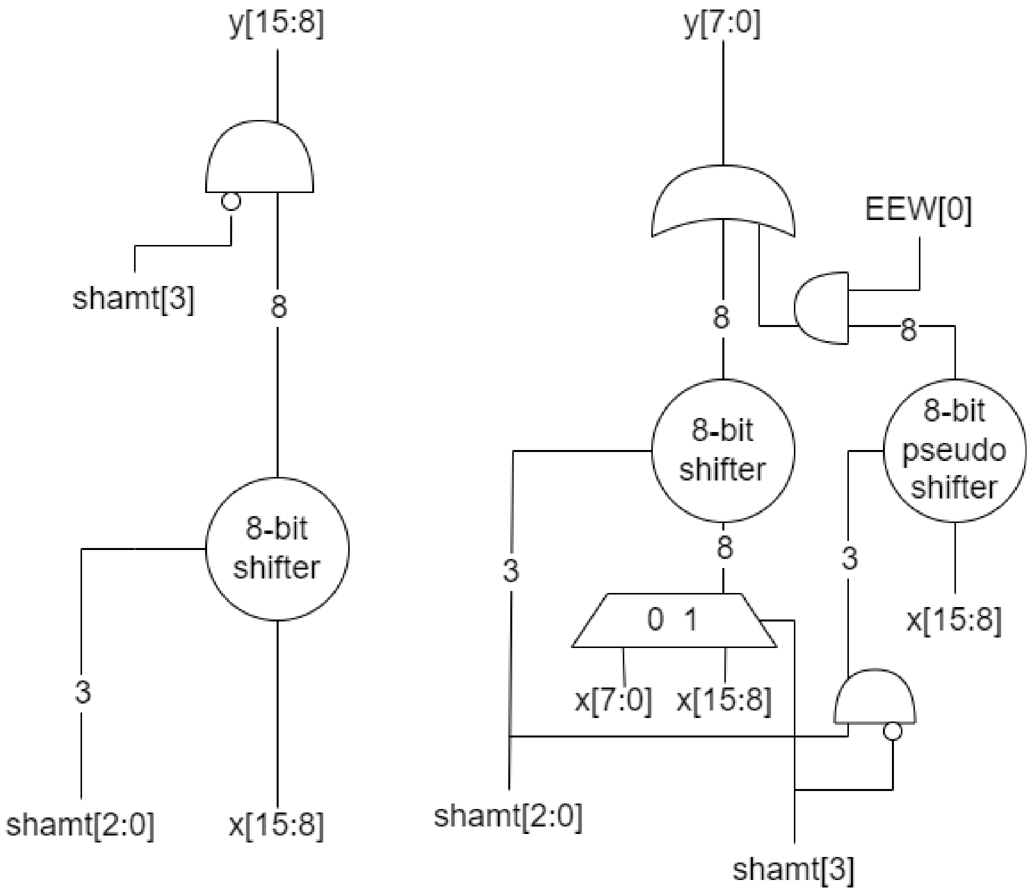


Figure Error! No text of specified style in document.**.14** Top-down view of 16-bit shifter

Here in the circuit, operands are x and shamt. X represents the input to be shifted and shamt is the shift amount to be performed, y is the output from the circuit. Operation of the circuit will be explained in two cases.

- Case 1: when the shifter is working as a 16-bit single shifter.
- Case 2: when the shifter is working as two separate 8-bit shifters.

Case 1: When the shifter is working as a single 16-bit shifter.

When the shifter is working as a single 16-bit shifter EEW[0] will be 1. This will let the output of the pseudo shifter to appear in the input of the or gate. When the shift amount is less than 8, shifter on left will shift the 8 most significant bits of the x input. Likewise, shifter on right will shift the least significant 8 bits by shift amount. The resulting zeros from the shift operation will be filled by output of pseudo shifter.

When the shift amount is greater or equal to 8, shamt[3] will equal to 1. Because of the resulting 1 from shamt[3] and gate at the output of the left shifter will produce zeros. At the same time, when the shift amount is greater or equal to 8, the and gate will force the shamt input of the pseudo shifter to 0. This will make all the outputs of the pseudo shifter zeroes. The mux on the input of the right shifter will shift switch the inputs from x[7:0] to x[15:8]. This way the shifter will start shifting the most significant bits.

All of these operations combined will simulate a single 16-bit shifter from 2 separate 8-bit shifters. An example will be shown to solidify the understanding.

Let’s consider an example when x bit series is 1010_1010_0101_0101, shamt is 2 and 10, and EEW[0] is 1.

When shamt=2 left shifter will shift the most significant 8 bits of x by two bits.

0	0	1	0	1	0	1	0

On the other hand, shifter on right will generate the following bit series.

0	0	0	1	0	1	0	1

And finally, pseudo shifter will generate the following bit series.

1	0	0	0	0	0	0	0

Anding the outputs of right shifter and pseudo shifter will give the following series.

1	0	0	1	0	1	0	1

Notice by appending the and operation result of pseudo shifter and right shifter to left shifter output we arrive at the following series 0010_1010_1001_0101. Indeed, this is the result of the binary sequence 1010_1010_0101_0101, shifted right by two bits.

When shamt is 10 resulting most significant 8 bits will be zeroes thanks to the and operator at the output of the shifter. Shamt input to the pseudo shifter will be forced to zero. This will result in zeroes on the output of the pseudo shifter. Therefore, output of the least significant 8-bits will be determined by the output of the right shifter. The resulting output for least significant 8-bits will be as follows.

0	0	1	0	1	0	1	0

By appending this result to 0000_000 we arrive at 0000_0000_0010_1010.

Case 2: when the shifter is working as two separate 8-bit shifters.

When the shifter is working as two separate 8-bit shifters EEW[0] will be 0. In this setup output of pseudo shifter will be orred with zeroes. So, two shifters will be able to work individually. This set up will work as long as the shift amount is less than 7 which is the expected input range.

Generating this circuit in parallel 16 times will be able to shift 16-16 bits wide elements or 32 8-bits wide elements.

3.5.3.2 64-bit Right shifter

64-bit right shifter is given in the Figure **Error! No text of specified style in document.**.15.

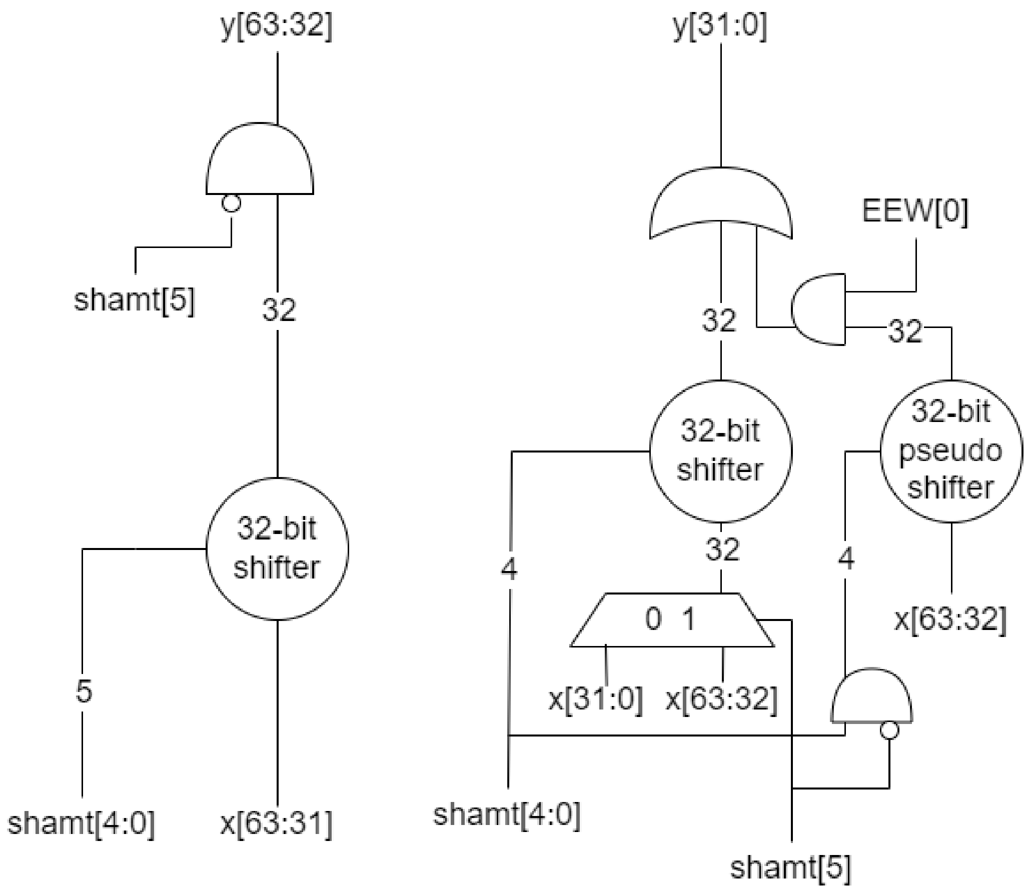


Figure Error! No text of specified style in document.**.15** 64-bit shifter top-down view

The circuit behaves the same as the 16-bit shifter circuit does. The only differences are the buses being 32 bits wide instead of 8 bits and shamt[5] is used in the control circuit instead of shamt[3]. The circuit again consists of two length(y)/2 bits wide shifters and pseudo shifter.

3.5.4 Divide by two unit

This circuit's sole purpose is shifting the input by 1 bit when enabled. This circuit is needed to perform averaging of operands without needing to add the shifter unit to the critical path of execute. By addition of this circuit, shifter circuits can be placed in parallel to the adder. Similarly, to shifter module, 16 16-bit halvers are generated in a parallel to generate *a* output of a mux; and 8-64 bit halvers are generated in parallel to generate *b* output of the mux. The mux chooses the right output by EEW[1]. Circuit diagram for 16 bit halver can be seen in Figure **Error! No text of specified style in document.**.16 and diagram for 64 bit halver can be seen in Figure **Error! No text of specified style in document.**.17.

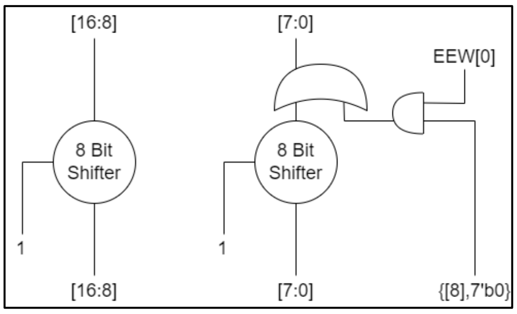


Figure Error! No text of specified style in document.**.16** 16 bit halver diagram

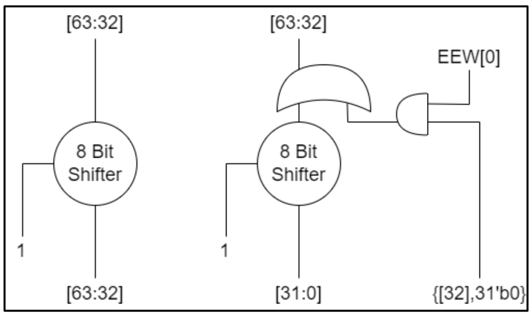


Figure Error! No text of specified style in document.**.17** 64 bit halver diagram

Shifter module hardwired is to 1 input is a mere cable, and the and result of EEW[0] with [8] only needs to be orred with [7]. So, with the critical path of two logic elements and a mux, division by 2 is implemented.

3.5.5 Vector Multiplier Unit

Similar to other parallel arithmetic operations, this unit is also designed using repetitive unit cells. These cells are specifically designed with a maximum element width of 64 bits, enabling them to perform parallel multiplication of elements in sizes of 8, 16, 32, and 64 bits. Essentially, this unit serves as the multiplication counterpart to the vector adder unit.

The design of the unit cell draws inspiration from the fundamentals of multiplication. When multiplying two numbers, X and Y, each consisting of 2*N bits, the multiplication process can be represented as follows: $X * Y = (X_h * 2^N + X_l) * (Y_h * 2^N + Y_l) = (X_h * Y_h) * 2^{2N} + (X_h * Y_l + X_l * Y_h) * 2^N + (X_l * Y_l)$. To facilitate a better understanding of this concept, Figure **Error! No text of specified style in document.**.18 provides a detailed illustration. Also from the figure, it can be seen that if a 2N-bit number is divided in two N-bits, and multiplication of numbers can also be expressed in terms of those numbers. For example, multiplication of two 32-bit numbers with other two 32-bit numbers are also calculated as intermediate results in such topology. Figure **Error! No text of specified style in document.**.19 illustrates this fact.

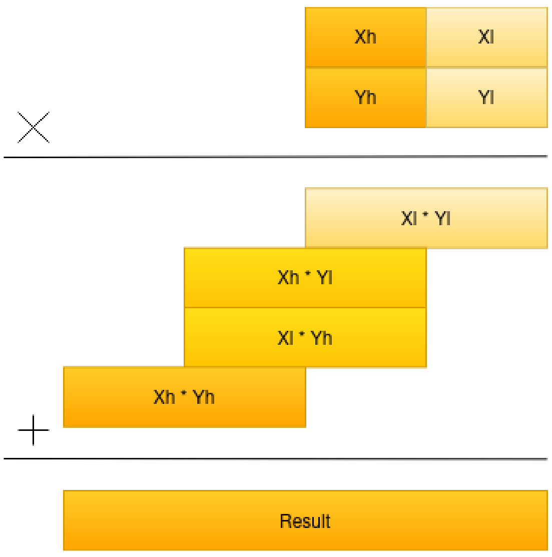


Figure Error! No text of specified style in document.**.18** Multiplication topology

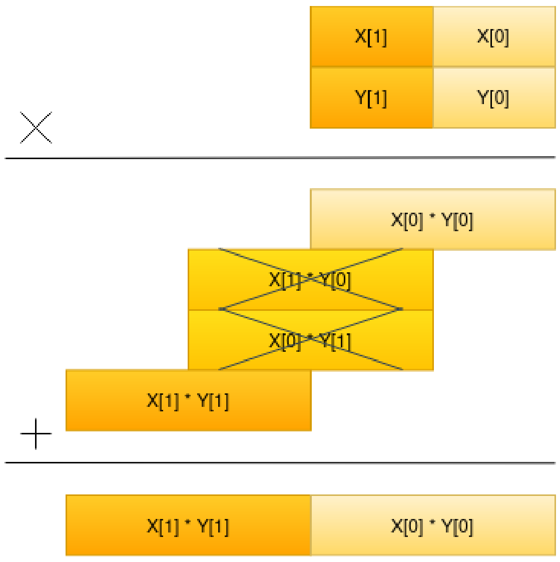
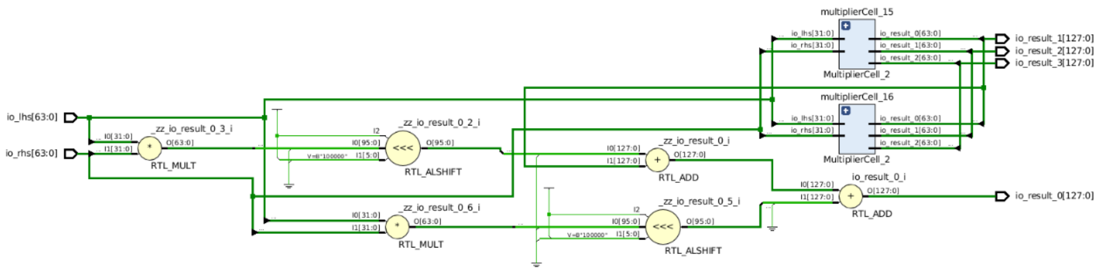
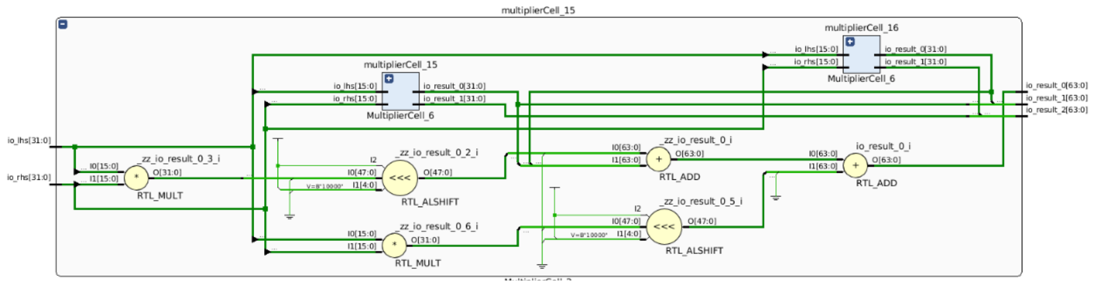


Figure Error! No text of specified style in document.**.19** Optimized multiplication topology

From these facts, a unit with two 64-bit data inputs, and four 128-bit data outputs for each mode(8, 16, 32 and 64) can be created. Where the inputs and the outputs can be an array of 64, 32, 16 and 8-bit and the output is the element wise multiplication of these elements. The unit can be designed recursively. Which means that while the H*L multiplications are done with N-bit multipliers, H*H and L*L multipliers are done with the N-bit version of the vector multiplication unit. This recursive structure is terminated at the 8-bit case. RTL schematic of the designed circuit is shown on Figure

Error! No text of specified style in document..20. RTL schematic also shows arithmetic shifts, but they are only used for the arrangement of the wires and are constant shifts.





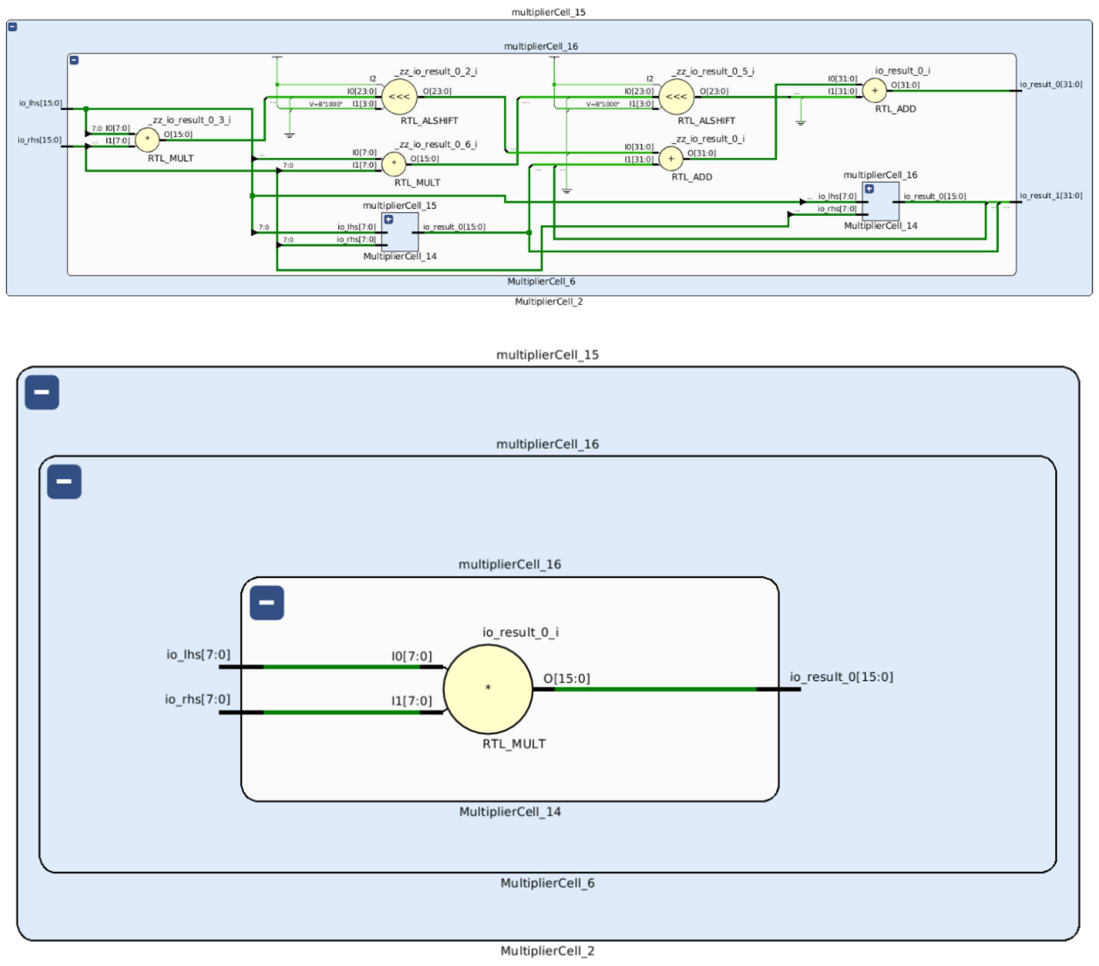


Figure Error! No text of specified style in document.**.20** RTL schematics for multiplier circuit

Furthermore, to adjust the length of the datapath, the unit cells can be combined in parallel. By arranging multiple unit cells next to each other, the datapath can be tailored to handle different data sizes. For example, if a 64-bit datapath is required, a single unit cell would be adequate. However, for a 256-bit datapath, four unit cells would be arranged in parallel without any interconnections between them. This parallel configuration offers scalability and flexibility, allowing the system to accommodate various data sizes without the need for additional inter-module connections. The block diagram illustrating this configuration can be seen in Figure X.X.

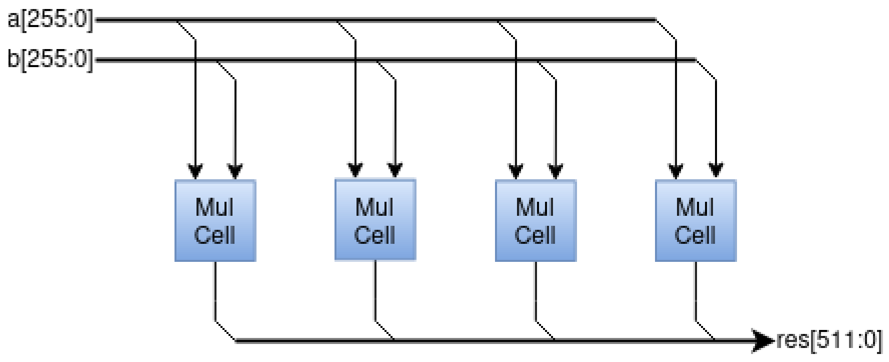


Figure Error! No text of specified style in document.**.21** Example of combined multiplication cells in datapath

3.5.6 Vector Division Unit

Unlike the other arithmetic operations, such as addition, subtraction, and multiplication, the division operation does not make use of a dedicated unit specifically designed for vector operations. This is because the division algorithm employed by the divider takes into account the length of the input and output values. The number of cycles required to perform the division is directly related to the number of bits involved in the division process.

Considering these factors, it appears that there is no immediate requirement for additional circuitry or specialized modules to adapt the division operation to a different and potentially more efficient computing module. The division algorithm implemented in the divider already accounts for the length calculations and cycle count, making it unnecessary to introduce further circuitry for this purpose.

3.6 Permutation Instructions

To implement permutation instructions. We utilized our CISC style decoder to handle most of permutation instruction. We chose this way because permutation instructions are too complex to implement on hardware. This approach allowed us to save space in FPGA.

3.6.1 Slider Unit

We also designed a slider unit similar to shifter to work with vector slide instructions. Unlike shift operations, a vector register group's elements are moved up and down by the slide instructions. To accomplish that we modified our shifter circuit slightly. If we shift our vector register by EEW amount multiplied with slide amount, we can achieve slide operation. Since our shift amounts are limited by EEW, we got rid of unused shifts in our slider circuit to save space.

4. REALISTIC CONSTRAINTS AND CONCLUSIONS

4.1 Realistic Constraints

The constraints of this project can be classified into three categories: economic impact, cost analysis, and standards. Each of these categories is explained in detail in the subsequent sections.

4.1.1 Social, environmental and economic impact

The primary objective of this project is to enhance processing power within a single unit and decrease the time required to run applications. While it has versatile applications, the ultimate aim is to optimize runtime efficiency.

The social impact of this project lies in its ability to leverage increased computational capabilities. Various domains, such as scientific research, cryptography, simulation, and others that can make use of this extension, stand to benefit significantly. By enabling more efficient and powerful computations, the project can contribute to advancements in these fields, fostering innovation, knowledge sharing, and potential breakthroughs in scientific understanding and technological applications.

Additionally, the reduction in computation time achieved through this project has a positive environmental impact by lowering power usage. By optimizing processing power and improving efficiency, the project offers a greener alternative for the industry. Decreased power consumption translates to reduced energy requirements, which can contribute to sustainability efforts and help mitigate the environmental impact of computing technologies. This greener alternative aligns with the growing emphasis on energy efficiency and eco-friendly practices in various industries.

4.1.2 Cost analysis

By enabling more computation within a single unit, this module eliminates the necessity for additional compute units in a system. This reduction in the number of units required leads to a decrease in manufacturing costs. With the module's enhanced processing power and decreased computation times, the overall cost of utilizing this module is also reduced. This cost reduction can be attributed to the optimized efficiency and improved performance, resulting in more efficient resource utilization and potentially lowering expenses associated with computational tasks. Overall, the module's ability to consolidate computation and reduce costs can have a positive impact on the economic viability and affordability of utilizing this technology.

4.1.3 Standards

This project strictly adheres to the RISC-V Vector Extension ISA v1.0, which means that any compiler capable of compiling code for a RISC-V machine can be used to leverage the capabilities of this project. The compatibility with existing RISC-V compilers ensures flexibility and ease of integration with different software development workflows. Developers can utilize their preferred compilers and programming languages to harness the benefits of the project while leveraging the extensive tooling and ecosystem already available for RISC-V architecture.

4.2 Conclusion

To conclude, this project has presented a comprehensive architectural proposal for the RISC-V Vector Extension. The implementation phase involved the development of a majority of the modules using SpinalHDL, a hardware description language. However, it is important to note that the integration process and subsequent integration tests were not conducted as part of this project and are recommended for future work.

One of the key findings of this project is the successful demonstration of a RISC-V Vector Extension module that strikes a balance between flexibility and compactness. This achievement highlights the potential of leveraging SpinalHDL, as its impact on the overall design and implementation is undeniably significant. The project's outcomes serve as a valuable foundation for further exploration and refinement in the field of RISC-V Vector Extension development.

In addition to the accomplishments mentioned earlier, an important discovery made during this project was the compact nature of the RISC-V Vector Extension instructions and the opportunity it provided to explore unorthodox module designs for enhanced performance. This aspect of the project allowed for greater creativity and fostered the improvement of our design skills. By leveraging the unique characteristics of the vector instructions, we were able to devise innovative solutions that contributed to the overall efficiency and effectiveness of the core. This finding highlights the flexibility and adaptability of the RISC-V architecture, as well as the potential for further exploration and innovation in future projects. The experience gained through this endeavor has undoubtedly enhanced our design capabilities and broadened our perspectives in the field of processor development.

5. REFERENCES

[1] K. Asanovic, "GitHub," 20 September 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0. [Accessed September 2022].

[2] C. Papon, "SpinalHDL," 2022. [Online]. Available: https://spinalhdl.github.io/SpinalDoc-RTD/master/artefacts/SpinalHDL_docs-master.pdf.

[3] [Online]. Available: https://www.scala-lang.org/.

[4] C. Papon, 2022. [Online]. Available: https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Foreword/index.html#why-moving-away-from-traditional-hdl.

[5] C. Papon, "GitHub - SpinalHDL/VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation," [Online]. Available: https://github.com/SpinalHDL/VexRiscv. [Accessed 2022].