**ECE575 Computer Architecture**
**Project: RISCV Vector Processor**
**May 9, 2023**

**Students:** *Anna Eaton*
*Mark Castellano*
*Mihir Kavishwar*

## 1 Abstract

This project aims to implement a RISC-V vector processor, which is a type of processor designed to handle vector-based computations efficiently. The project focuses on implementing the RISC-V Vector Extension (RVV), which is a standard extension to the RISC-V instruction set architecture that enables vector operations, to our 7-stage RISC-V scalar pipeline. The RVV extension includes a set of vector registers, vector memory access instructions, and vector arithmetic instructions. The hardware design involves developing a custom processor core that supports the RVV extension, including the vector register file and associated hardware logic to enable vector operations. The software design includes developing an assembler and a compiler that can generate code that uses the RVV extension [1]. The project will involve simulating and testing the processor design using a variety of benchmarks and test programs. The goal is to demonstrate that the RISC-V vector processor can perform vector operations more efficiently than a traditional scalar processor.

## 2 Design

Our goal was to implement the RISCV Vector ISA. Our vector register file stores 32 vectors, each 64, 32-bit elements long. We also have a register to store the intermediate value from a multiply-accumulate instruction and a register to store the length of the vector. In RISCV parlance, our $\mathrm{ELEN} = 32$ bits, our $\mathrm{VLEN} = 64 \times 32 = 2048$ bits, we only allow $\mathtt{SEW} = 32$ bits, $\mathtt{LMUL} = 1$, and we used the undisturbed mask/tail setting, making all other settings illegal. We get performance from increasing the number of lanes in our pipelined processor such that we can operate on 4 vector elements each clock cycle. This requires our vector register file to have $4\times$ the number of ports compared to the scalar register file. We also have more ports on our data memory so that we can load and store 4 elements of data each clock cycle.

We only implemented a subset of the instructions from the RISCV ISA. We implemented three categories of instructions: configuration-setting, strided load and store, and integer arithmetic. The configuration-setting instructions are `vsetvli`, `vsetivli`, and `vsetvl`. The strided load and store instructions are `vle32.v`, `vse32.v`, `vlse32.v`, and `vsse32.v`. The vector integer arithmetic instructions are `vadd.vv`, `vsub.vv`, `vmul.vv`, `vmulh.vv`, `vmulhu.vv`, `vmulhsu.vv`, `vmacc.vv`, `vnmsac.vv`, `vmadd.vv`, and `vnmsub.vv`. More information can be found in Table 1. We chose to implement these instructions after we compiled two of the benchmark files from previous assignments and saw which instructions we needed to implement in order to run those benchmarks. We did this with the help of Jinzheng, who generously helped us make a vector compiler that we used to get the instructions we needed.

While the datapath only contains 4 lanes, the vector register file can store vectors as long as 64 elements. To compute a vector operation with a vector longer than the number of lanes,

[1]The compiler and assembler were developed by Jinzheng Tu, a senior graduate student in Prof. Wentzlaff's group

Table 1: RISC-V Vector Instructions

| **Instruction** | **Description** |
|---|---|
| vsetvl | set VLR from register |
| vsetvli | set VLR with 12-bit immediate |
| vsetivli | set VLR with 5-bit immediate |
| vle32.v | 32-bit unit-stride load |
| vse32.v | 32-bit unit-stride store |
| vlse32.v | 32-bit stride load |
| vsse.32.v | 32-bit stride store |
| vle32.v | 32-bit unit-stride load |
| vse32.v | 32-bit unit-stride store |
| vlse32.v | 32-bit strided load |
| vsse32.v | 32-bit strided store |
| vadd.vv | vector-vector integer add |
| vsub.vv | vector-vector integer subtract |
| vmul.vv | vector-vector signed multiply (low bits) |
| vmulh.vv | vector-vector signed multiply (high bits) |
| vmulhu.vv | vector-vector unsigned multiply (high bits) |
| vmulhsu.vv | vector-vector signed/unsigned multiply (high bits) |
| vmacc.vv | integer multiply-add, overwrite addend |
| vnmsac.vv | integer multiply-sub (overwrite minuend) |
| vmadd.vv | integer multiply-add (overwrite multiplicand) |
| vnmsub.vv | integer multiply-sub (overwrite multiplicand) |

we stall the Fetch stage once a vector instruction has entered the Decode stage. Then, we send 4 elements from the vector to the Execute stage every clock cycle, and stop stalling the Fetch stage once we have sent the entirety of the vector down the pipe. If the vector length is not a multiple of 4, we only enable the $vectorlength\%4$ lanes to write to memory when the last group of elements is sent down the pipe.

One challenging instruction we wanted to implement was multiply-accumulate. This instruction multiplies two vectors and then adds the destination vector to the product. To execute this instruction, we essentially treated it as two separate instructions. First, we performed vector multiplication and stored the result in a 33rd vector register called the Intermediate Register. Then, we added the vector in the Intermediate Register to vector destination register and stored the result in the vector destination register.

## 3 Testing Methodology

We wanted to reuse a lot of the testing suite from previous projects so we could directly compare this processor to previous projects. We generated `.vmh` files of `ubmark-cmplx-mult.C` and `ubmark-vvadd` with a RISCV Vector compiler and assembler. After verifying that the scalar operations still worked properly, we tested the vector instructions on these two benchmarks.

During this process we had to redesign the testing framework with our own six-ported memory unit, and also redo the base simulation so that we could access the amount of data parallelism we needed out of both RF and memory.

Listing 1: Vvadd Compiled with Vector Instructions

```
100b8: 37 15 01 00 lui a0, 17
100bc: 13 05 05 00 mv a0, a0
100c0: 57 70 04 c5 vsetivli zero, 8, e32, m1, ta, mu
100c4: 07 64 05 02 vle32.v v8, (a0)
100c8: b7 15 01 00 lui a1, 17
100cc: 93 85 05 1a addi a1, a1, 416
100d0: 87 e4 05 02 vle32.v v9, (a1)
100d4: 13 06 00 00 li a2, 0
100d8: 57 04 94 02 vadd.vv v8, v9, v8
100dc: 93 06 41 00 addi a3, sp, 4
100e0: 27 e4 06 02 vse32.v v8, (a3)
100e4: 93 06 41 02 addi a3, sp, 36
100e8: 13 07 05 02 addi a4, a0, 32
100ec: 07 64 07 02 vle32.v v8, (a4)
100f0: 13 87 05 02 addi a4, a1, 32
100f4: 87 64 07 02 vle32.v v9, (a4)
100f8: 13 07 05 04 addi a4, a0, 64
100fc: 07 65 07 02 vle32.v v10, (a4)
10100: 13 87 05 04 addi a4, a1, 64
10104: 87 65 07 02 vle32.v v11, (a4)
10108: 57 04 94 02 vadd.vv v8, v9, v8
1010c: 27 e4 06 02 vse32.v v8, (a3)
10110: 93 06 41 04 addi a3, sp, 68
```

## 4 Evaluation

The first goal of this project was to get some vector operations working without disrupting the functionality of the scalar architecture. We successfully implemented strided loads and vector-vector addition. However, they do not work with bypassing. These instructions alone are not enough to execute the benchmarks, so we were unable to make direct comparisons to a scalar processor.

## 5 Discussion

While we were unable to completely implement the vector co-processor, we reached a number of intermediate checkpoints that we were satisfied with. First, our scalar processor maintained complete functionality. Second, we successfully implemented the strided load instruction, which demonstrates that we are able to broadcast data across the lanes of the pipeline and make strided reads from memory. Third, we successfully implemented vector-vector integer add and subtract, which demonstrates that we are able to iterate over the elements of a vector in the register file and bring those elements into the lanes of a pipeline, and write back those elements into the vector register file. Moreover, in implementing some vector instructions, we demonstrated the concept that vector processors are more capable of exploiting innate parallelism in the data and this get improved performance over scalar processors when executing vectorized code.

## 6 Figures

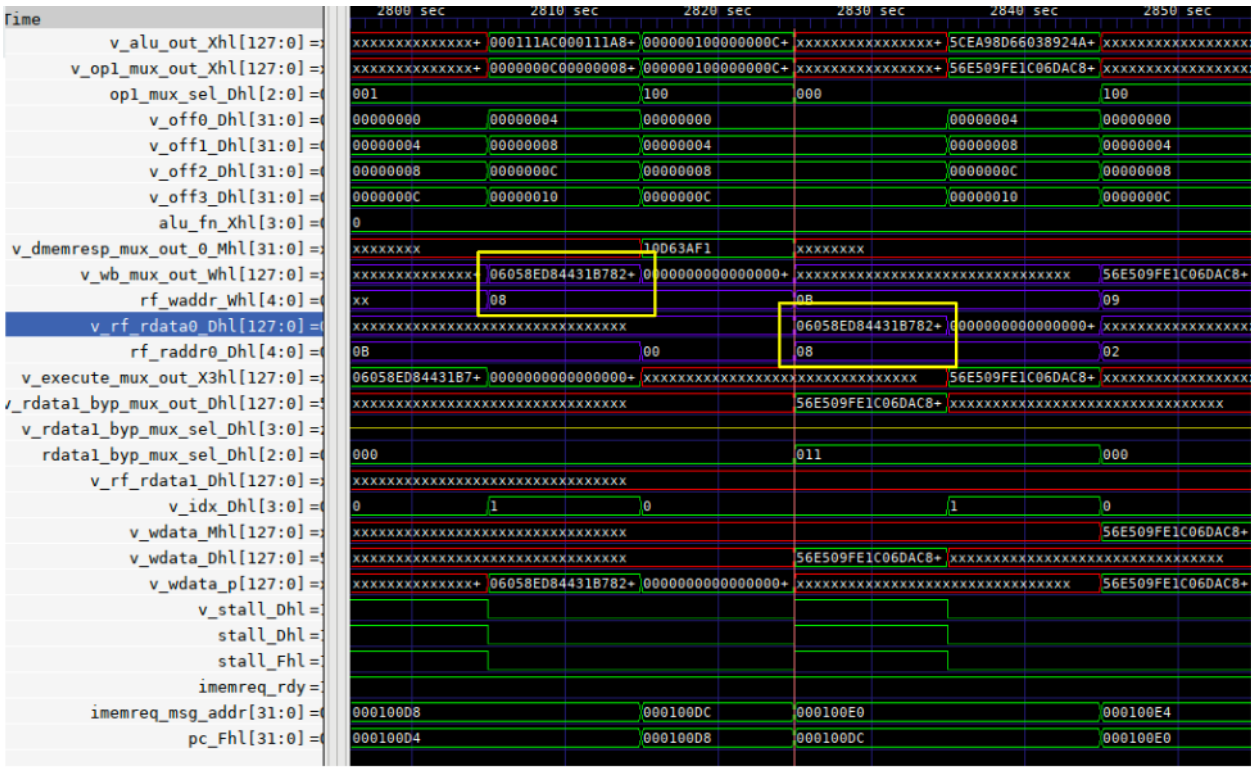

Figure 1: Waveform of vvadd benchmark showing v8 being written by load instruction and then later read by an add instruction

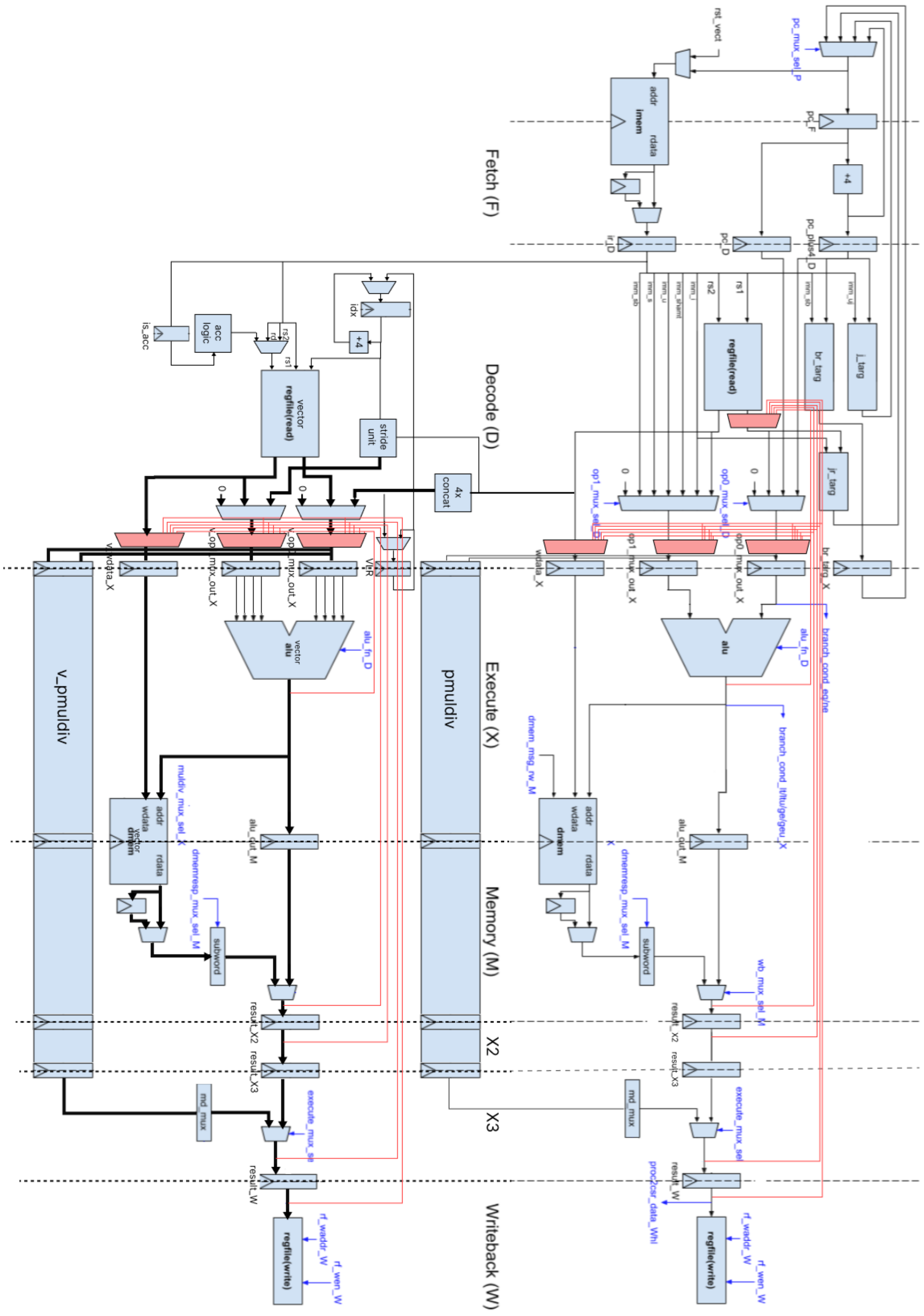

Figure 2: Datapath