American University of Beirut

EECE 499

A RISC-V Vector Processor

A RISC-V Vector Unit Implementation

Authors:
Imad Al Assir
Mohammad El Iskandarani

Supervisor:
Dr. Mazen Saghir

May 23, 2020

Contents

1 Introduction

Field Programmable Gate Arrays (FPGA) have been in the hardware scene for quite some time now. Their programmable fabric allows the creation of highly parallelizable and modular applications with respectable efficiency. This trait made it the perfect platform for our RISC-V Vector Processor Unit design.

Vector processors are processing units that operate on one dimensional arrays called vectors. Instead of an instruction targeting a single scalar register, a vector instruction targets a vector register composed of several elements. Another key feature are the lanes that operate in parallel, thus boosting throughput and making them optimal for number crunching applications, such as machine learning algorithms.

In this report, we will go through our implementation of a RISC-V Vector Processor Unit. Each component will be identified separately, and design decisions will be justified and explained.

2 Design

In this project, our role was to design the foundations of a RISC-V vector processor, i.e. a decoder, controller, register file, Arithmetic-Logic Unit (ALU), and memory. Our implementation was based on the RISC-V Vector Extension Spec v0.8. The processor consists of 3 pipeline stages (for simplicity): Decode (Control signals generation and Register File access), Execute (ALU and memory) and Write-Back. The processor supports chaining between ALU lanes themselves, and between memory and ALU. This technique aims to reduce computational speed by taking the result directly from the generating unit, thus skipping additional register file accesses and avoiding Read-After-Write (RAW) hazards. To be able to implement chaining, we chose to have 2 execution lanes (for both ALU and memory). This also implies having 2 register banks to be able to service the lanes efficiently.

3 Controller

3.1 Decoder

3.1.1 Description

The decoder is a purely combinational circuit: it takes an instruction as input and divides it into fields needed by the datapath. Refer to Appendix A for a detailed description of each field.

3.1.2 Diagram

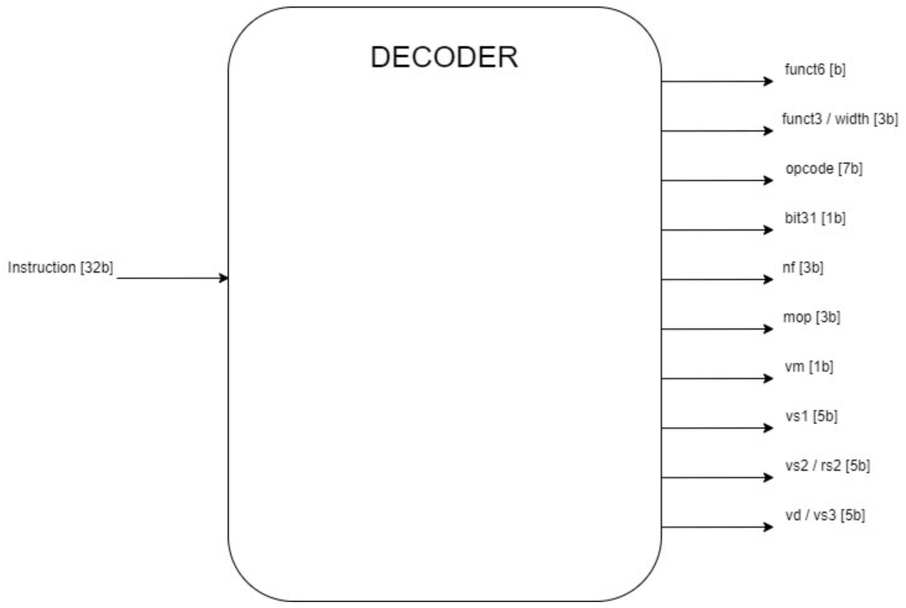

Figure 1: Decoder Diagram

3.2 Control Unit

3.2.1 Description

The controller takes the instruction fields from the decoder and outputs control signals according to the instruction type. It also contains Control and Status Registers (CSRs). Refer to Appendix A for a detailed description of each field.

3.2.2 Implementation

Our controller code is split into 3 processes:

1. The first one implements reading and writing from/to the CSRs.
2. The second is purely combinational and implements the control signals generation logic.
3. The third implements writing to vl (vsetvl and vsetvli instructions). Although vl is a CSR, we chose to implement the writing in a separate process since it is read-only and can only be modified through the vsetvl and vsetvli instructions.

Here are the control signals used:

- WriteEn: Enables writing to register file
- SrcB: Selects vector, scalar, or immediate as second operand
- MemWrite: Enables writing to memory
- MemRead: Enables reading from memory
- WBSrc: Selects if write back source is from ALU or MEM
- mv: Indicates if instruction is a move/merge instruction or not
- extension: Specifies if extension from memory is signed or unsigned
- addrmode: Specifies addressing mode (unit stride, strided, indexed)
- memwidth: Specifies number of bits per memory transfer

3.2.3 Diagram

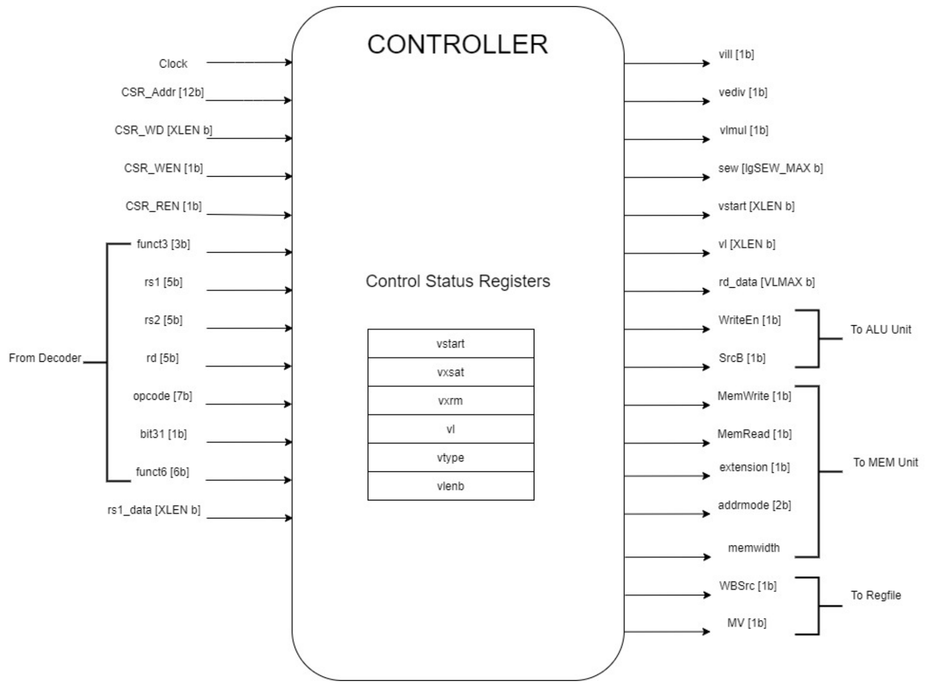

Figure 2: Controller Diagram

4 Register File

4.1 Description

The register file stores the vector registers in banks. Each bank is composed of 16 vector registers. The current design contains 2 banks, since our ALU unit has 2 lanes. More about that later.

4.2 Implementation

Here are some details about our register file implementation:

- Register mapping: 1 bank contains the upper 16 registers, and the other contains the lower 16 registers.
- Implicit dispatcher: We assigned each bank to a respective memory or ALU lane, meaning both operands should belong to the same lane (i.e. it is the compiler's job to ensure that). However, writing can be to any bank, not necessarily the same as the corresponding lane.

- Mask bit: If the instruction is masked, the first bit of an element from vector v0 is read to see if this element is masked or not. This signal is driven to the other bank as well as to the output of the register file.
- Counters: We used the following counters to keep track of the elements read/written in the vector register: number of elements read, number of elements written, number of bits read, number of bits written.
- newInst: Is a control signal that signals the entry of a new instruction. It is used to reset the counters.
- Read on rising edge, write on falling edge: We found it to be more sensible than reading on the falling edge and writing on the rising edge, since the pipeline registers are clocked on the rising edge.
- SEW-size transfer: On every clock cycle, SEW bits are read and, if the result is ready, SEW bits are written. The elements read are sign extended.
- Masked operations: When the result is ready, we look at the mask bit: if it is 0, neglect the result and don't write, else write. Note that this is a naive implementation of masking; a more efficient one is to be implemented soon.

4.3 Diagram

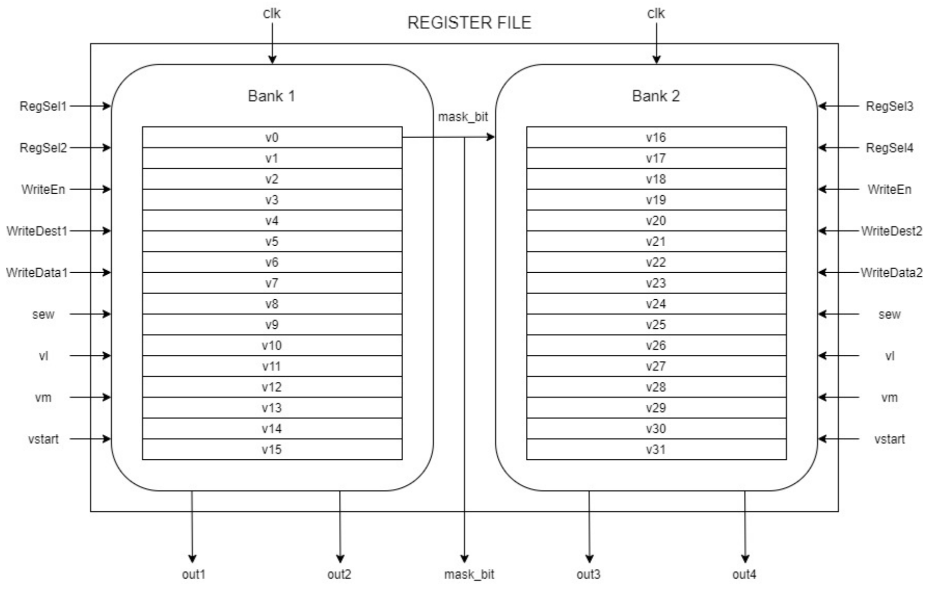

Figure 3: Register File Diagram

5 Arithmetic-Logic Unit

5.1 ALU Unit

5.1.1 Description

The ALU is the unit that performs all arithmetic and logical operations. It consists of lanes, where each lane is assigned a bank in the register file (e.g Bank 1 maps to Lane 1, Bank 2 to Lane 2). Refer to Appendix B for the list of supported ALU instructions.

5.1.2 Implementation

The ALU lane in itself is purely combinational; it performs the desired operation based on funct6 and funct3, where the operands are SEW_MAX bits wide.

5.2 Mv Block

5.2.1 Description

The MV block handles merge/move instructions. We chose to make a dedicated block for these 2 instructions because they require knowledge of the mask bits, unlike the regular ALU operations.

5.2.2 Implementation

The circuit is purely combinational, just like the ALU. The mask bit is examined and the operation is executed. The outputs of the ALU and MV blocks are multiplexed and selected by the controller, based on the funct6 field (which acts like an opcode for ALU instructions).

5.3 Diagram

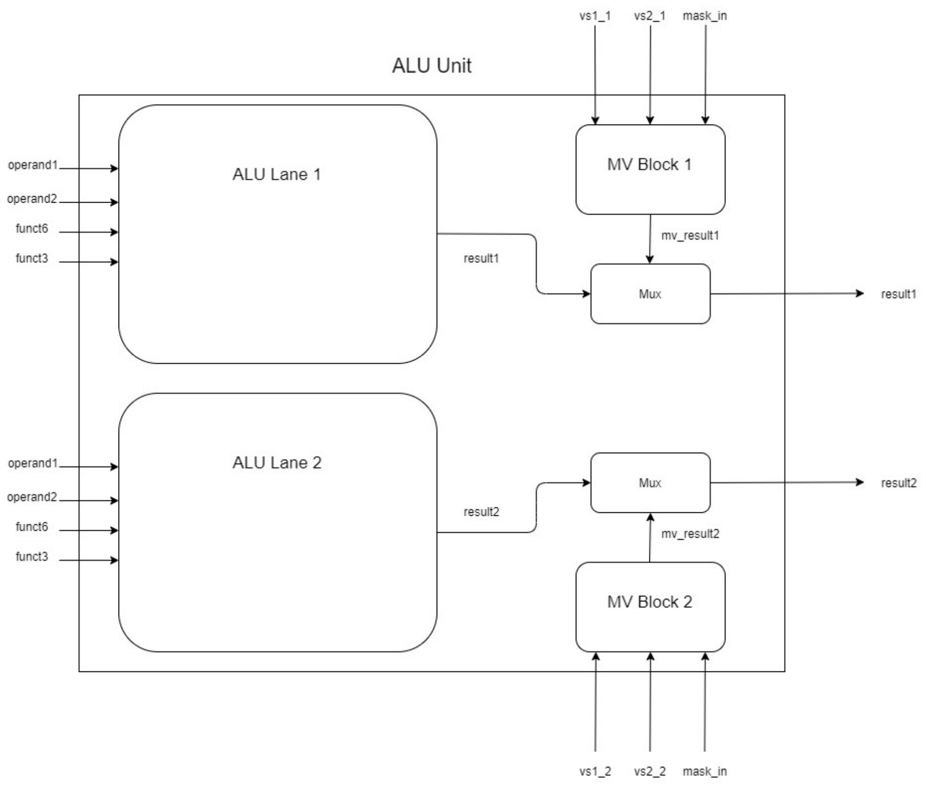

Figure 4: ALU Unit Diagram

6 Memory

6.1 Memory Unit: Address Computation

6.1.1 Description

The memory lane computes the 32-bit address based on the desired addressing mode, transfer size and some control signals. It is composed of lanes, similar to the ALU unit. Refer to Appendix B for the list of supported memory instructions.

6.1.2 Implementation

Contrary to the ALU unit, the memory unit is clocked since it needs to increment and feed a new address to memory every clock cycle.

6.1.3 Diagram

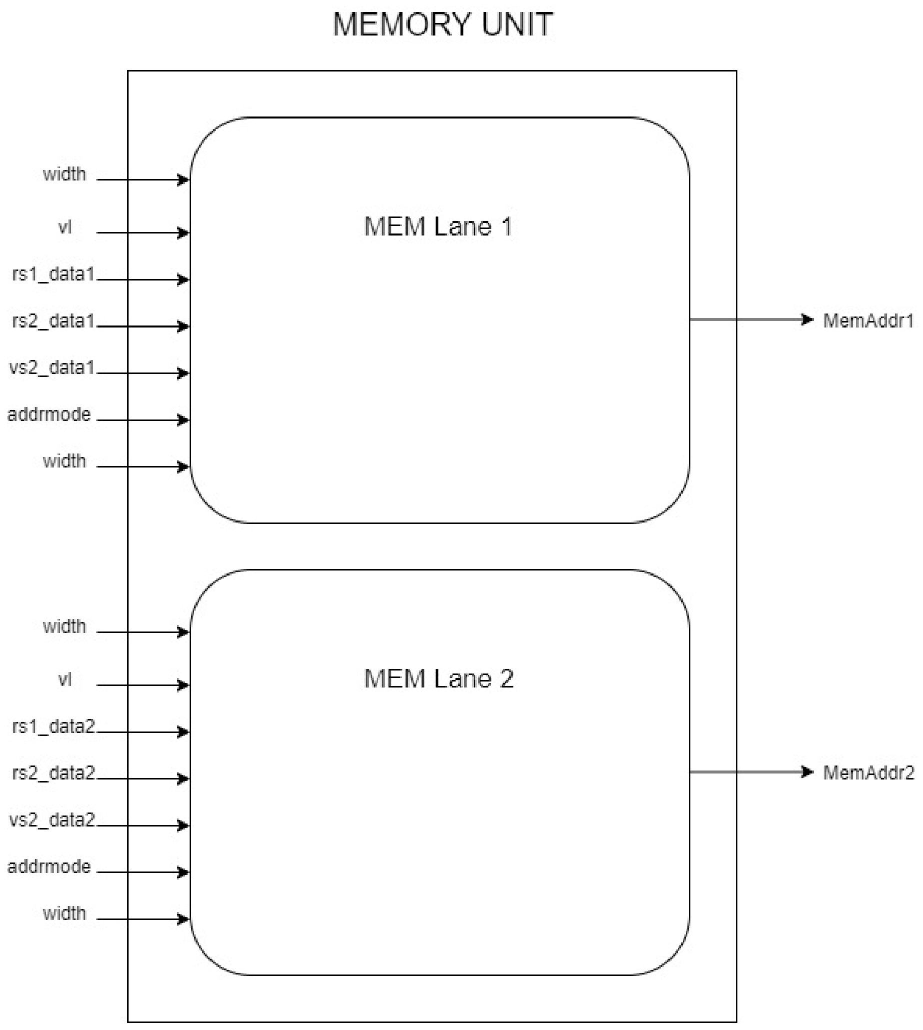

Figure 5: Memory Unit Diagram

6.2 Memory

6.2.1 Description

The memory is the address space where we store to/load from data. Similar to our register file, it is composed of banks. Each bank has 1 read and 1 write ports.

6.2.2 Implementation

- Size: We decided to make each bank fit into 1 BRAM chip of our Zynq FPGA.
- Transfer size: The width of the transfer is determined by the controller. It could be 8, 16, 32 or SEW bits.
- Addressing: Although the address coming from the MEM lane is 32 bits, the bits above the 10th are used to select the bank, since the address space of each bank is 10 bits wide.

6.2.3 Diagram

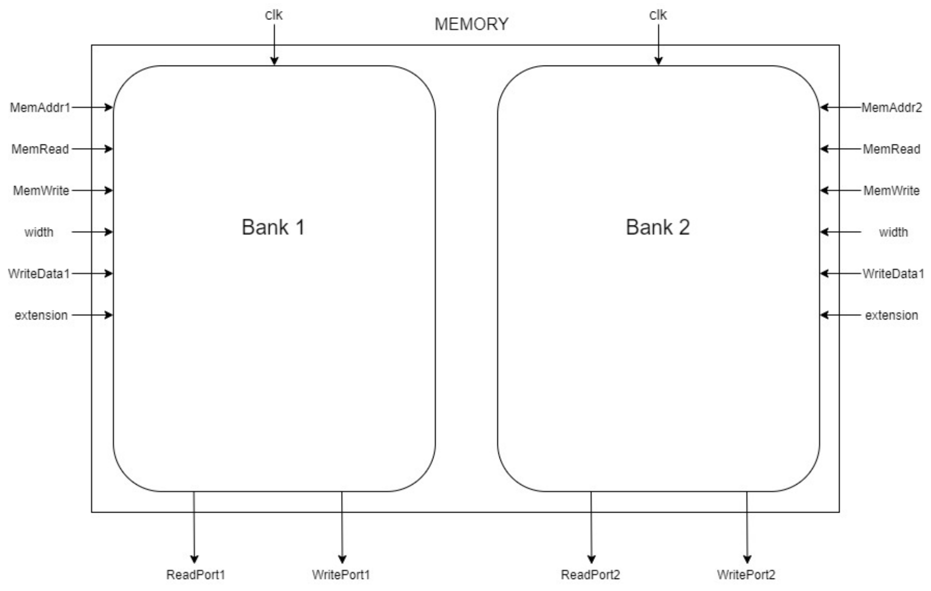

Figure 6: Memory Diagram

7 Chaining

7.1 Description

Chaining is a method used in vector processors to speedup computational speed. It is similar to data forwarding in a classic scalar processor, where the results generated by an execution (ALU or memory), can be passed on to another execution unit next cycle immediately, bypassing the register file writeback.

7.2 Implementation

For chaining to be possible, 2 instructions should come directly after each other, and the destination register of the old should be a source register of the new instruction (e.g. vadd.vv v1,v2,v3 followed by vsub.vv v4,v5,v1). For our implementation, we used a variable 'age' for each of the 2 execution lanes, and checked the conditions (stated above) for chaining to be possible. This controller then outputs mux select signals which will tell each execution unit where to get its operands from: the other ALU lane, memory or register file.

Appendix A

Generics

- ELEN: maximum element width in bits.
- XLEN: scalar register length in bits.
- VLEN: number of bits in register.
- VLMAX: maximum number of elements in a vector register.
- SEW_MAX: standard element width in bits.
- lgSEW_MAX: log of maximum standard element width.

Instruction fields from Decoder

- funct6: used to differentiate between ALU instruction families (add, subtract, etc...).
- funct3: used to differentiate between ALU instruction families further.
- opcode: used to differentiate between instruction types (ALU, MEM, vsetvl).
- bit31: Most significant bit of the instruction; used for vsetvl and vsetvli instructions.
- nf: encodes the number of whole vector registers to transfer for the whole vector register load/store instructions.
- mop: memory addressing modes.
- vm: indicates if instruction is masked or not (1 if unmasked, 0 if masked).
- vs1: first vector operand.
- vs2/rs2: second vector operand/scalar register.
- vd/vs3: vector destination.

Control and Status Registers

- vstart: specifies the index of the first element to be executed by an instruction.
- vxsat: fixed-point accrued saturation flag.
- vxrm: fixed-point rounding mode.
- vl: specifies the number of elements to be read/written from/to a vector register.

- vtype: contains the following fields:
  - vill: used to encode that a previous vsetvli instruction attempted to write an unsupported value to vtype.
  - vediv: used by EDIV extension.
  - vlmul: vector register group multiplier (LMUL) setting.
  - vsew: standard element width encoding.

  vlenb: number of elements in bytes (VLEN/8).

Appendix B: Supported Instructions

Since the project is still in its early stages, only part of the RISC-V instruction set was implemented. These instructions are listed below, with a quick description. Please refer to the spec sheet for more details.
Note that .vv stands for vector-vector instructions, .vx vector-scalar instructions, and .vi vector-immediate instructions.

7.4 Vector Unit-Stride Instructions
vlw.v: vector load word signed
vlwu.v: vector load word unsigned
vle.v: vector load element
vsw.v: vector store word
vse.v: vector store element

7.5 Vector Strided Instructions
vlsw.v: vector load word strided signed
vlswu: vector load word strided unsigned
vlse.v: vector load element strided
vssw.v: vector store word strided
vsse.v: vector store element strided

7.6 Vector Indexed Instructions
vlxw.v: vector load word indexed signed
vlxwu.v: vector load word indexed unsigned
vlxe.v: vector load element indexed
vsxw.v: vector store word indexed
vsxe.v: vector store element indexed
vsuxw.v: vector store word unsigned indexed
vsuxe.v: vector store element unsigned indexed

12.1 Vector Single-Width Integer Add and Subtract
vadd.vv: element-wise vector addition
vadd.vx
vadd.vi
vsub.vv: element-wise vector subtraction (vs2[i]-vs1[i])
vsub.vx
vrsub.vx element-wise vector reverse subtraction (vs1[i]-vs2[i])
vrsub.vi

12.4 Vector Bitwise Logical Instructions
vand.vv: element-wise vector and
vand.vx
vand.vi
vor.vv: element-wise vector or
vor.vx

vor.vi
vxor.vv: element-wise vector xor
vxor.vx
vxor.vi

12.5 Vector Single-Width Bit Shift Instructions vsll.vv: shift left logical
vsll.vx
vsll.vi
vsrl.vv: shift right logical (zero-extended)
vsrl.vx
vsrl.vi
vsra.vv: shift right arithmetic (sign-extended)
vsra.vx
vsra.vi

12.7 Vector Integer Comparison Instructions
vmseq.vv: vd[i]= vs2[i]==vs1[i]
vmseq.vx:
vmseq.vi
vmsne.vv: vd[i]= vs2[i] != vs1[i]
vmsne.vx
vmsne.vi
vmsltu.vv: vd[i]= vs2[i] $<$ vs1[i] unsigned
vmsltu.vx
vmslt.vv: vd[i]= vs2[i] $<$ vs1[i] signed
vmslt.vx
vmsleu.vv: vd[i]= vs2[i] $<=$ vs1[i] unsigned
vmsleu.vx
vmsleu.vi
vmsle.vv: vd[i]= vs2[i] $<=$ vs1[i] signed
vmsle.vx
vmsle.vi
vmsgtu.vx: vd[i]= vs2[i] $>$ vs1[i] unsigned
vmsgtu.vi
vmsgt.vx: vd[i]= vs2[i] $>$ vs1[i] signed
vmsgt.vi

12.8 Vector Integer Min/Max Instructions
vminu.vv: element-wise minimum unsigned: vd[i]= min(vs2[i],vs1[i])
vminu.vx
vmin.vv: element-wise minimum signed: vd[i]= min(vs2[i],vs1[i])
vmin.vx
vmaxu.vv: element-wise maximum unsigned: vd[i]= max(vs2[i],vs1[i])
vmaxu.vx
vmax.vv: element-wise maximum signed: vd[i]= max(vs2[i],vs1[i])

vmax.vx

12.9 Vector Single-Width Integer Multiply Instructions
These instructions return the same number of bits as the operands (i.e. SEW)

vmul.vv: returns lower SEW bits of element-wise multiply signed
vmul.vx
vmulh.vv: returns higher SEW bits of element-wise multiply signed
vmulh.vx
vmulhu.vv: returns higher SEW bits of element-wise multiply unsigned
vmulhu.vx
vmulhsu.vv: returns higher SEW bits of element-wise multiply signed-unsigned
vmulhsu.vx

12.10 Vector Integer Divide Instructions
vdivu.vv: vd[i]= vs2[i]/vs1[i] unsigned
vdivu.vx
vdiv.vv: vd[i]= vs2[i]/vs1[i] signed
vdiv.vx
vremu.vv: vd[i]= vs2[i] rem vs1[i] unsigned
vremu.vx
vrem.vv: vd[i]= vs2[i] rem vs1[i] signed
vrem.vx

12.15 Vector Integer Merge Instructions
vmerge.vvm: vd[i] = v0[i].LSB ? vs1[i] : vs2[i]
vmerge.vxm
vmerge.vim

12.6 Vector Integer Move Instructions
vmv.v.v
vmv.v.x
vmv.v.i

Appendix C: Code Structure

In this appendix, we explain how our code is structured. All our code is present on Github.

- The decoder can be found here.
- The controller (called Control Unit in our code) can be found here.
- There is a top level, Controller, that groups the decoder and control unit. It can be found here.
- The register file consists of two banks, Bank1 which is the lower bank and contains v0, the mask register, and Bank. These banks are joined under instantiated in the Register File.
- The ALU consists of 2 lanes instantiated in 1 ALU unit. The ALU unit has pipeline registers before and after it, splitting the the read, execute and write-back stages. The structure with these pipeline registers can be found here.
- The Memory Unit consists of 2 Memory Lanes, similarly to the ALU. The unit's purpose is to compute the address to fetch from in memory. The memory itself is formed of banks, similarly to the register file.
- To test things out, we joined the register file and ALU here.