
NORTHWESTERN
UNIVERSITY

Computer Science Department

**Technical Report**
**Number: NU-CS-2025-12**

May, 2025

# VIR-V: A RISC-V RoCC Accelerator for VCODE Computing

**Qinze Jiang**

## Abstract

The growing demand for high-performance computing has spurred the development of specialized accelerators to overcome the limitations of general-purpose processors. This thesis presents VIR-V, a RISC-V Rocket Custom Coprocessor (RoCC) accelerator designed for executing VCODE, which is an intermediate vector dataflow representation derived from the high-level parallel language NESL. NESL expresses nested data parallelism, which is flattened into segmented vector operations in VCODE. These are further optimized and structured into VIR, a dataflow form suitable for hardware execution. VIR-V maps VIR operations to hardware, enabling efficient and highly parallel execution of data-parallel operations.

Implemented in Chisel and tightly integrated into the Rocket-Core pipeline, VIR-V supports essential VCODE operations, including element-wise arithmetic, scans, reductions, and permutations. Verified and benchmarked using FireSim, an FPGA-accelerated cycle-exact simulation platform, VIR-V achieves an average speedup of 1.89× compared to baseline CPU execution across a range of microbenchmarks. These results validate VIR-V's effectiveness in accelerating high-level data-parallel workloads while maintaining full compatibility with the RISC-V ecosystem. This work lays the groundwork for future enhancements such as floating-point support and deeper pipelining to further improve performance.

## Keywords

RISC-V, RoCC Accelerator, VCODE, Vector Dataflow, High-Level Parallelism, Chisel HDL, Hardware Acceleration, FireSim Benchmarking

NORTHWESTERN UNIVERSITY

# VIR-V: A RISC-V RoCC Accelerator for VCODE Computing

MS THESIS

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Computer Engineering

By

Qinze Jiang

EVANSTON, ILLINOIS

May 2025

# ABSTRACT

The growing demand for high-performance computing has spurred the development of specialized accelerators to overcome the limitations of general-purpose processors. This thesis presents VIR-V, a RISC-V Rocket Custom Coprocessor (RoCC) accelerator designed for executing VCODE, which is an intermediate vector dataflow representation derived from the high-level parallel language NESL. NESL expresses nested data-parallelism, which is flattened into segmented vector operations in VCODE. These are further optimized and structured into VIR, a dataflow form suitable for hardware execution. VIR-V maps VIR operations to hardware, enabling efficient and highly parallel execution of data-parallel operations.

Implemented in Chisel and tightly integrated into the Rocket-Core pipeline, VIR-V supports essential VCODE operations, including element-wise arithmetic, scans, reductions, and permutations. Verified and benchmarked using FireSim, an FPGA-accelerated cycle-exact simulation platform, VIR-V achieves an average speedup of 1.89× compared to baseline CPU execution across a range of microbenchmarks. These results validate VIR-V's effectiveness in accelerating high-level data-parallel workloads while maintaining full compatibility with the RISC-V ecosystem. This work lays the groundwork for future enhancements such as floating-point support and deeper pipelining to further improve performance.

## ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my advisor, Professor Peter Dinda, for giving me the opportunity to do research and the suggestions for my research plan. I also sincerely appreciate the valuable instruction and advice provided by Karl Hallsby, who not only explained concepts and design strategies patiently, but also helped me set up environments and solve problems in coding throughout my research. And special thanks to my colleagues in the Prescience Lab for their support, discussions, and collaboration that have enriched this research.

This thesis is not just the result of my efforts but also a testament to the support and love of my family. My heartfelt thanks go to my parents and my grandmother, whose unconditional love and encouragement have given me the strength to persevere through challenges. Finally, I would like to dedicate this thesis to my grandfather, who, though no longer with us, has always been my source of inspiration. His unwavering love and belief in my abilities have motivated me throughout my journey. His values and teachings have profoundly shaped who I am today.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1
# INTRODUCTION AND BACKGROUND

## 1.1 Introduction

The rapid evolution of high-performance computing (HPC) has driven the need for specialized hardware accelerators to bridge the performance gap left by the stagnation of traditional Moore's law and Dennard scaling. In heterogeneous computing systems, accelerators play an essential role in executing computationally intensive tasks, thereby enhancing overall efficiency and throughput. In this context, we introduce VIR-V, a custom accelerator implemented as a Rocket Custom Coprocessor (RoCC) extension for the RISC-V architecture. VIR-V maps VIR operations, a high-level vector dataflow intermediate representation derived from VCODE, into hardware, which enables deeply pipelined, highly parallel execution of vector operations and reducing software overhead.

VIR-V is part of the Village project[1], which is a new implementation of the classic NESL nested data parallel high-level parallel language (HLPL). The Village project aims to bridge the longstanding gap between two competing paradigms in parallel programming. Low-level parallel languages (LLPLs), such as C/C++ with CUDA, OpenMP, and MPI, provide extensive opportunities for microarchitectural optimization, but sacrifice portability and readability. In contrast, HLPLs, such as NESL, abstract away hardware

[1]Contributors include Alex Butler, David Krasowska, Griffin Dube, Karl Hallsby, Kirill Nagaitsev, Liam Strand, Lucas Myers, Peizhi Liu, Peter Dinda, Ruiqi Xu and Qinze Jiang.

details, enabling clear algorithmic expression at the expense of performance. Village integrates these two paradigms, leveraging VIR, to enable aggressive compiler optimizations including operator fusion, in-place updates, and targeted vectorization, thus significantly improving performance while maintaining high-level clarity.

VIR-V serves as a hardware implementation of these optimizations, offloading critical vector computations from the CPU to specialized accelerators that operate seamlessly within the RISC-V processor pipeline. By combining the expressive advantages of HLPLs and the performance benefits typically associated with LLPLs, the Village project, including VIR-V, is attributed to a comprehensive and efficient framework for parallel computing in heterogeneous systems.

The remainder of this thesis introduces the background, and details the design and evaluation of VIR-V within the Village compilation framework, describing how high-level algorithmic specifications are effectively transformed into efficient hardware execution, and demonstrating performance improvements for HPC workloads.

## 1.2 Background

### 1.2.1 NESL and VCODE

NESL [1][2] is a classic high-level parallel language (HLPL) known for introducing nested data parallelism. Its syntax resembles functional languages like Standard ML, enabling concise and expressive descriptions of parallel algorithms. NESL simplifies the development of parallel software by clearly exposing algorithmic parallelism

without explicitly coupling code to hardware specifics. In practice, NESL augments basic functional language constructs with three core features: collection types, parallel comprehension (filter/map operations on collections), and parallel recursive function calls.

To facilitate execution, NESL programs are first transformed into an intermediate representation known as VCODE [3]. VCODE operates as an abstract stack machine, processing instructions that manipulate arbitrary-length segmented vectors. Each VCODE instruction retrieves vector operands from the stack, executes the specified operation, and subsequently stores the resulting vectors back onto the stack. Among these operands, a vector of unsigned integers (e.g., `uint64_t`) may act as a segment descriptor, and is also retrieved from the stack. Each element of the descriptor specifies how the elements of another data vector are divided into smaller units, called segments. Specifically, the descriptor's entries indicate the length of each segment in the associated data vector. Critically, segmentation in VCODE is limited to a single level. More importantly, VCODE instructions inherently understand vector segmentation. Segments allow nested parallel operations from NESL to be represented as operations on flat, segmented vectors, simplifying the mapping of parallel constructs onto hardware.

For example, the NESL expression `(map (fn subvec => reduce (+) subvec) [[1,2,3], [4,5], [6]])` is a nested parallel operation where each reduction acts on a subvector of different length. After flattening, it becomes a single segmented reduction operation in VCODE that works on a flat data vector `[1,2,3,4,5,6]` and

a corresponding segment descriptor `[3,2,1]`. This tells the hardware to reduce the first 3 elements, then the next 2, and finally the last 1, thus preserving the semantics of the original nested structure, which will get `[6,9,6]` as the result. This representation simplifies hardware implementation significantly. Instead of managing multiple threads or recursive control paths, hardware can employ a linear, pipelined unit that processes the flat vector sequentially. The segment descriptor acts like a set of boundary markers, guiding the hardware on when to finish processing one group of data and begin the next. This eliminates the need for dynamic thread management or complex control logic, enabling efficient and scalable execution of nested parallel programs [4] in vector-style hardware.

However, this flattening transformation — collapsing nested parallel operations into single-level segmented vector instructions — can sometimes negatively impact performance. Deep nesting, once flattened, can lead to suboptimal memory behavior by repeatedly handling large segmented vectors rather than exploiting potential cache locality through controlled sequential execution. While incremental flattening [5], which is a solution to this problem, is not implemented for NESL in Village, we do implement other aggressive optimization techniques, including inlining, fusion, and in-place operations, to alleviate these performance limitations.

VCODE's instruction set (Table 1.1) encompasses a range of operations including element-wise arithmetic, scans, reductions, permutations, indexing, and packing. Each instruction clearly specifies work and depth complexity characteristics, making

performance modeling straightforward. The representation is highly amenable to vectorization, fusion optimizations, and targeted hardware implementations, because of its structured execution model and explicit dataflow semantics.

| Count | Instruction Type | Example | Work | Depth |
|---|---|---|---|---|
| 38 | Element-wise | `* INT` | $O(n)$ | $O(1)$ |
| 14 | Scans/Reductions | `+_SCAN` | $O(n)$ | $O(\lg n)$ |
| 6 | Permutations | `DPERMUTE FLOAT` | $O(n)$ | $O(1)$ to $O(\lg n)$ |
| 2 | Extract/Replace | `EXTRACT INT` | $O(n)$ | $O(1)$ to $O(\lg n)$ |
| 1 | Pack | `PACK INT` | $O(n)$ | $O(1)$ to $O(\lg n)$ |
| 2 | Ranking | `RANK_UP` | $O(n \lg n)$ | $O(\lg n)$ |
| 2 | Distribute/Index | `INDEX` | $O(n)$ | $O(1)$ to $O(\lg n)$ |
| 3 | Segment Descriptor | `LENGTHS` | $O(n)$ | $O(1)$ |
| 7 | Basic I/O | `READ` | $O(n)$ | $O(n)$ |

Table 1.1: Classifications and typical asymptotic complexities for classic VCODE instructions.

In traditional NESL implementations, VCODE instructions are executed by an interpreter. The Village project, however, leverages VCODE as a foundational intermediate representation, compiling it further into hardware-oriented dataflow graphs known as VIR. Unlike VCODE, VIR is not stack-based; it adopts a static single-assignment (SSA) form, where values are defined once and propagated through a directed acyclic graph of operations. This structure closely resembles a pure dataflow model, making it highly suitable for optimization and direct hardware mapping. By eliminating the interpretation layer and exposing data dependencies explicitly, the compilation flow from VCODE to VIR to accelerators like VIR-V enables significantly more efficient execution and hardware synthesis.

### 1.2.2 Chisel HDL

Chisel (Constructing Hardware In a Scala Embedded Language) [6] is a hardware design/generator language developed at UC Berkeley that leverages the Scala programming language [7] to offer enhanced productivity and flexibility compared to traditional hardware description languages (HDLs) like Verilog and VHDL. Originally, HDLs were created primarily for simulation purposes, and only later adapted for hardware synthesis. This historical development led to restrictions — synthesizable designs must be expressed using only limited subsets of the language, complicating design workflows and tool support. Furthermore, these conventional HDLs lack many of the powerful abstraction capabilities common in modern software programming languages, restricting opportunities for design reuse and efficient exploration of architectural variations.

Chisel addresses these limitations by embedding hardware construction concepts within Scala. Scala provides several advantageous programming paradigms including object-oriented and functional programming techniques, parameterized types, generics and automated type inference. Utilizing these features, Chisel significantly improves a designer's ability to build reusable hardware modules and perform thorough design-space exploration through highly parameterized hardware generators.

Unlike traditional HDLs, Chisel generates both synthesizable low-level Verilog, which is compatible with standard FPGA and ASIC synthesis tools. For simulation, Chisel designs are typically passed through external tools such as Verilator to produce high-performance, cycle-accurate behavioral C++ simulators. This separation of con-

cerns streamlines verification and accelerates design iterations. Consequently, Chisel is particularly well-suited for developing specialized accelerators with complex parameterization and optimization needs, such as the VIR-V accelerator described in this thesis.

### 1.2.3 RISC-V and RoCC Accelerators

RISC-V is an open-source instruction set architecture (ISA) designed explicitly with customization and extensibility [8]. This flexibility allows researchers to implement new hardware capabilities without redefining the entire ISA or affecting the compatibility of existing software.

Rocket-Chip is a fully-featured system-on-chip (SoC) implementation based on the RISC-V ISA [9]. A Rocket-Core is a single-issue, in-order processor with a classic five-stage pipeline. Rocket-Core supports the complete RISC-V privileged specification and can boot a full Linux operating system.

A significant feature of Rocket-Chip's design is its support for customization via the Rocket Custom Coprocessor (RoCC) interface. As shown in Figure 1.1, RoCC accelerators are tightly integrated with the Rocket-Core and share system resources, including the L1 and L2 caches, floating-point units (FPU), and page table walker (PTW). The RoCC interface leverages RISC-V's custom instruction encoding space to transparently integrate custom hardware accelerators into the processor's pipeline. RoCC instructions flow through the main core's pipeline until they reach the execution stage, where they are dispatched directly to the accelerator hardware. Depending on the

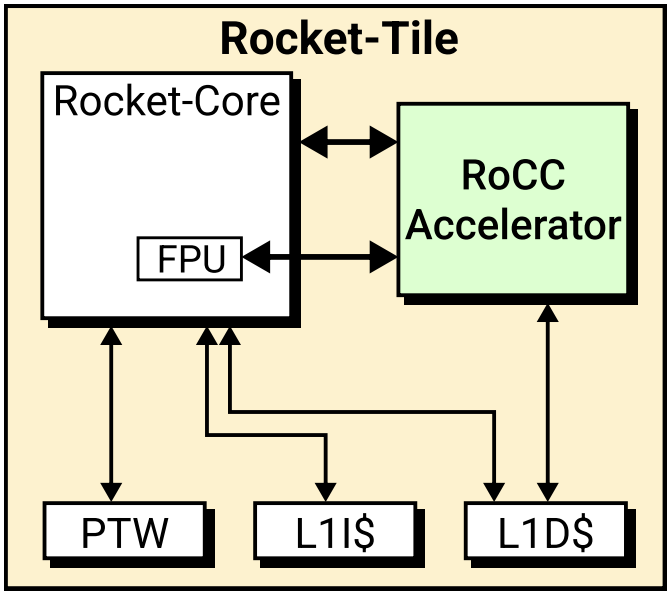


Figure 1.1: RoCC Accelerator Connection to RISC-V

instruction's nature and implementation, the main CPU core might stall briefly while awaiting completion, or in certain cases, run in parallel with the accelerator.

In this thesis, we introduce the VIR-V accelerator, implemented specifically using the RoCC interface of the RISC-V architecture. While our current integration targets the Rocket core, the design is also compatible with other RoCC-enabled cores such as BOOM. VIR-V instructions follow the same segmented vector memory model as VCODE, where vectors are contiguous in memory. During execution, the accelerator accesses memory directly through the tile's L1 data cache. In the event of an invalid instruction, the accelerator signals an interrupt, allowing the Rocket-Core to handle it as a standard RISC-V interrupt. Overall, this accelerator totals just 1200 lines of Chisel code.

VIR-V realizes a subset of the VCODE instruction set, directly mapping vector op-

erations previously executed in software into specialized hardware units. Structurally, VIR-V is a vector processor capable of simultaneously processing multiple 64-bit data elements simultaneously. VCODE's explicit vector abstraction and straightforward memory model greatly facilitate direct hardware implementation. Additionally, the inherent design semantics of VCODE guarantee that input vectors do not alias, enabling simpler hardware logic and efficient, conflict-free memory access during execution. While the RISC-V Vector (V) extension offers general-purpose vector processing capabilities, it does not natively support the segmented operations that arise from VCODE's flattened nested parallelism. Emulating such operations with standard vector instructions would introduce significant software overhead and control complexity. In contrast, VIR-V is specifically optimized for these patterns, allowing more efficient hardware execution and tighter integration with the Village compiler's output.

# CHAPTER 2

# HARDWARE IMPLEMENTATION

This chapter introduces the implementation of VIR-V's hardware components and functional verification for each operator. The operators in VIR-V are listed in A.1. Figure 2.1 illustrates the internal structure of VIR-V, and the functionalities of its individual components are discussed in detail in the remainder of this chapter.

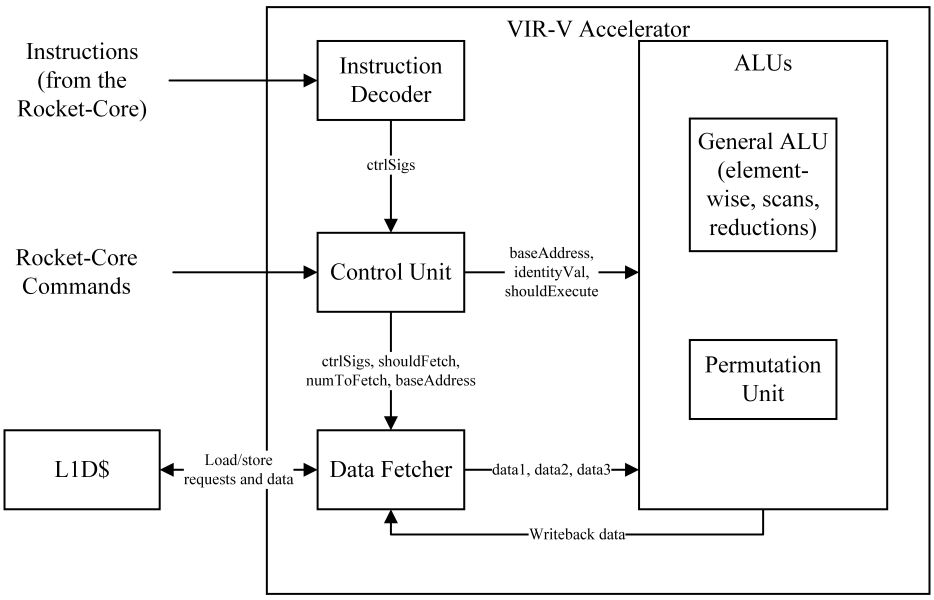


Figure 2.1: Internal structure of VIR-V

## 2.1 Instruction Decoder

The instruction decoder converts the received instructions generated by the Village compiler from VCODE to control signals. There are five control signals: `legal`,

numMemFetches, aluFn, identityVal, isMemOp. legal is to determine whether the input instruction is legal or not. numMemFetches defines how many operands must be fetched from the data memory. aluFn is sent to the ALUs to help recognize which operation will be executed on the operands. identityVal provides an identity value for all operations. isMemOp stands for whether this operator needs to read data from the memory.

We created six decoding tables for instruction decoding, five of which are to decode compute instructions and the other one is to decode control instructions, which provide configurations in every program.

## 2.2 Data Fetcher

The data fetcher handles the actual details of fetching and organizing the submitted memory requests. With this module, VIR-V can access memory through the Rocket Core's L1 data cache, offering a more straightforward interface for accelerator design. However, this approach typically provides lower throughput compared to using a direct TileLink connection [9].

We define a DataIO bundle as a fundamental data structure used throughout the accelerator design. It encapsulates both the memory address and the associated data value, each of xLen bits, as Figure 2.2 shows. This structure serves as the unified interface for operand representation, result delivery, and memory access coordination across all accelerator's components.

```
class DataIO(xLen: Int) extends Bundle {
  val addr = Bits(xLen.W)
  val data = Bits(xLen.W)
}
```

Figure 2.2: Chisel implementation of `DataIO` bundle.

The data fetcher receives `ctrlSigs` generated by the instruction decoder, the base address and the number of operands from the control unit, memory operation, and ALUs' results as the data to store into the memory. It outputs `fetchedData` as the data loaded from memory. The base address used by the data fetcher is assumed to be a physical address, as VIR-V operates outside the virtual memory system and does not perform its own address translation. This module supports two memory operations: store operation (`MemoryOperation.write`) and load operation (`MemoryOperation.read`). When a store operation is requested, the requested address and data follow the address and data of the module's input. In a load request, the address is the base address with an 8-byte offset, and the data output is unused because the cache does not use it for loads.

Since only one memory request can be handled in a clock cycle, there is a simple finite state machine to control the memory operations when there are multiple outstanding memory requests. When the `start` signal is asserted and the base address is valid, the state machine will move from the `idle` state into the `running` state to start a memory request. For the load operation, the data from cache will be stored into `vals` first in order to reorder the data, because the data returned to the data fetcher may not be in the order as requested due to data locality (e.g. the second requested data in L1 cache could

be returned before the first requested data in L2 cache). For the store operation, the module only increments the number of requests, and the data from cache is transferred to module's interface. The state machine remains in the `running` state if all required data has not been fetched, or it moves into the `idle` state.

## 2.3 ALUs

In VIR-V, we have implemented a subset of the VCODE instructions in a RISC-V processor design, which currently has 33 instructions, including all integer and boolean element-wise and scan/reduction operations, and one permutation operation. All integer and boolean element-wise and scan/reduction operations are integrated in a general ALU, and the permutation operation is in a separate ALU for the convenience of other permutations' development. Both of the general ALU and the permutation unit take three operands, identity value, base address and several control signals from the control unit as the inputs, and output the result vector. The data type of operands is a data type called `DataIO` (Figure 2.2), which is defined in the data fetcher and includes the data and its corresponding address.

### 2.3.1 Element-Wise Operations

Element-wise operations apply a straightforward operation to each element of a vector individually, producing a new vector with the results in the same order as the input, and all input vectors involved must have the same length. We created a Chisel hardware

generator function called `elementWiseMap` (Figure 2.3), which is not a software function but a high-level hardware description. This generator constructs a hardware module that performs a pairwise binary operation on two vectors of `DataIO`. This function takes two vectors and a binary operator as inputs. `zip()` generates a tuple for each pairwise element of `xs` and `ys`, and `zipWithIndex` adds an index to each pair. The variable `results` stores the calculation results and their corresponding address, where the addresses of the element-wise operations are contiguous and each offset is 8 bytes. Importantly, this function defines hardware behavior and structure; it does not perform any computation at runtime like a typical software function. Instead, it elaborates into RTL (Register Transfer Level) logic during synthesis.

```
def elementWiseMap(xs: Vec[DataIO], ys: Vec[DataIO],
                   op: (UInt, UInt) => UInt): Vec[DataIO
                     ] = {
    val indexedPairs = xs.zip(ys).zipWithIndex
    val results = WireInit((0.U).asTypeOf(Vec(batchSize,
      new DataIO(xLen))))
    results := indexedPairs.map{ case ((x, y), i) => {
        val result = Wire(new DataIO(xLen))
        result.addr := io.baseAddress + (i.U * 8.U)
        result.data := op(x.data, y.data)
        result
        }
    }
}
```

Figure 2.3: Chisel implementation of element-wise operation between two input vectors.

`SELECT INT` has a minor difference compared to the other element-wise operations,

since three operands involve in the calculation. The third operand, `in3`, is registered in `selectFlags`. And tuples of `in1, in2, in3` are created to map the operands to the 2-to-1 multiplexers, which make choices between `in1` and `in2` based on `in3`.

We also created a pipelined computing unit for `* INT` and `/ INT` operations. `muldivBank` includes `batchSize` instantiated `MulDiv` units and their configurations (the default value of `fn` is `FN_MUL` and it should be configured based on the executed operations). `MulDiv` is a generator module from Rocket-Chip, which provides templated creation of pipelined integer multipliers and dividers. The pipelining structure solves the timing violations of multiplication and division caused by long combinational paths exceeding the critical timing constraints in FPGA prototype.

### 2.3.2 Scan Operations

Scan operations compute cumulative prefix sums. Conceptually, this process computes prefix results in the following way: given an input vector `[a, b, c, d]`, the scan operation with addition produces `[a, a+b, a+b+c, a+b+c+d]`. The addresses of the scan operations' results are contiguous and the offset is 8 bytes. For `+_SCAN INT`, `AND_SCAN BOOL`, `OR_SCAN BOOL` and `XOR_SCAN BOOL` operations, the function `scan()` is to calculate cumulative sums with an identity value sent by the control unit provided by the control unit. This is not a software function but a parameterized function that constructs a functional unit for cumulative (prefix) operations. And as Figure 2.4, we use a similar pipelining structure to `* INT` and `/ INT` operations to realize `*_SCAN INT` to meet the timing requirements. The difference from pipelined element-wise operations

is that `muldivBank` takes the result of last calculation as input to generate a pipelined calculation. The initial value of `in1` is an identity value and `in1` takes the result from last `muldivBank` during the rest of the scan operation. And `in2` is the next element in the batch. When the computation in `muldivBank(batchsize-1)` is finished, the result will become a new identity for the scan of the next batch.

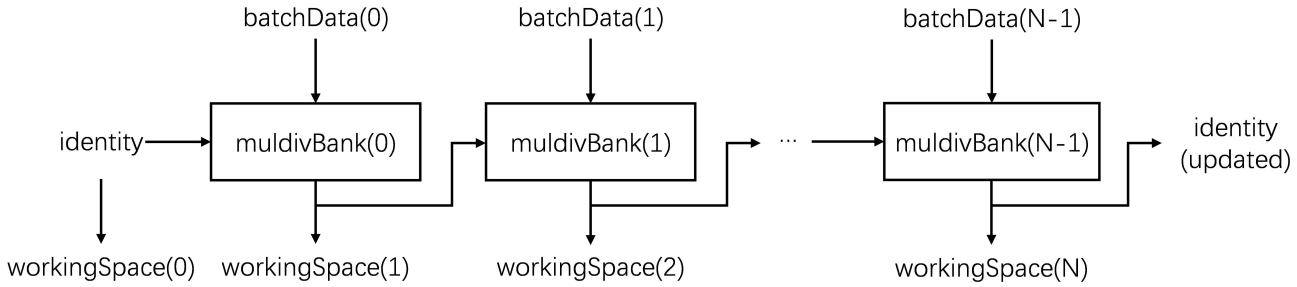


Figure 2.4: Pipeline structure of `*_SCAN INT`

In the implementations of `MAX_SCAN INT` and `MIN_SCAN INT` operations, we created a pipeline consisting of several instantiated comparators (Figure 2.5). The comparator module produces its result one clock cycle delayed, and the prefix sums from each stage of the comparator pipeline will be captured as the final results. However, if the clock frequency is high or many comparisons are strung together in a chain, this pipeline may violate timing requirements because the comparator module processes a single `xLen` comparison relatively slowly.

```
final class Comparator(val xLen: Int) extends Module {
  val io = IO(new Bundle {
    val req = Input(Valid(new ComparatorReq(xLen)))
    val resp = Output(Valid(new ComparatorResp(xLen)))
  })
  val x = io.req.bits.in1
  val y = io.req.bits.in2

  val result = Wire(new ComparatorResp(xLen))
  result.data := DontCare
  switch (io.req.bits.fn) {
    is (ComparatorOp.min) {
      result.data := x.min(y)
    }
    is (ComparatorOp.max) {
      result.data := x.max(y)
    }
  }

  io.resp.valid := RegNext(io.req.valid)
  io.resp.bits := RegNext(result)
}
```

Figure 2.5: Chisel implementation of a comparator.

### 2.3.3 Reduction Operations

A reduction operation is a computation that combines a set of values into a single result using a binary operation (e.g., addition, multiplication, maximum). The result of a reduction operation is stored at the base address. For `+_REDUCE INT`, `AND_REDUCE BOOL`, `OR_REDUCE BOOL` and `XOR_REDUCE BOOL` operations, the function `reduction` takes the value of `in1` and an operator to complete the calculation. Similar to `*_SCAN INT`, a

pipeline consisting of `MulDiv` modules is used in the implementation of `*_REDUCE INT` to create a pipelining structure but only the output of the last pipeline stage is taken as the result. Pipelining comparators are also used in the `MAX_REDUCE INT` and `MIN_REDUCE INT` operations and the result is taken from the last stage of the pipeline.

### 2.3.4 Permutation Unit

We designed a dedicated ALU for permutation operations that implements the basic permutation operator. A scatter-style permutation permutation instruction, denoted as `permute(data, index)`, rearranges the elements in the data vector such that the output at position `i` receives the value from `data[index[i]]`. That is, `out[i] = data[index[i]]`. This definition matches the NESL semantics, where the `index` vector specifies the destination for each corresponding element. Because the VIR intermediate representation guarantees that permutation indices are unique and non-overlapping, the hardware can safely execute each permutation assignment in parallel.

From Figure 2.6, we can see that the loop generates several identical combinational logics to assign the data and addresses. Each permutation works in parallel with others and the time complexity of this hardware implementation depends on the critical path of a single permutation logic. Therefore, a permutation in hardware is essentially an $O(1)$ operation. In software implementation, `permute(a, i)` permutes elements in `a` to indices in `i` [1]. But apart from shuffling addresses, the load/store operations in memory and the calculation of addresses need extra computation and bring more cost of resources. In hardware, however, the index vector directly determines routing at compile

time, and permutation is performed purely through combinational wiring without extra memory instructions or control logic.

```
for (i <- 0 until batchSize) {
    workingSpace(i).data := io.data(i).data
    workingSpace(i).addr := io.baseAddress + (io.index(i)
      .data * 8.U)
}
io.out.valid := true.B
```

Figure 2.6: Chisel implementation of permutation operation.

## 2.4 Control Unit

The control unit is to generate and send various control signals to other components based on Rocket-Core commands (`roccCmd`) and the output of the instruction decoder (`ctrlSigs`). The generated control signals are listed in Table 2.1.

The control unit includes a finite state machine describing how a single calculation of VIR-V operates. The state machine diagram is presented as Figure 2.7. Here are the details of the states and transitions:

**`idle`** This is the default state of VIR-V's operation. When the RoCC command is valid and legal, and the coprocessor reads data from memory, the state machine moves into the `fetch1` state.

**`fetch1`** This state is used to fetch the first operand. If this data fetching is completed and the number of operands is 2 or 3, the state machine moves into the

`fetch2` state.

**`fetch2`** This state is used to fetch the second operand. When this data fetching is completed, if the number of operands is 3, the state machine moves into the `fetch3` state; if the number of operands is 2, it moves into the `exe` state.

**`fetch3`** This state is used to fetch the third operand. The state machine moves into the `exe` state if the data fetching is completed.

**`exe`** This state is for the VCode operator executions. There are three situations in state transitions when the execution completes:

- If the operation is a reduction and there are still remaining operands, the state machine moves into the `fetch1` state to fetch the operands. And the addresses of the first and second operands will increase by `(batchSize*8)`.
- If the operation is a reduction and there is no operand, the state machine moves into the `write` state.
- If the operation is not a reduction, the state machine moves into the `write` state.

**`write`** In this state, VIR-V writes the results into data memory. If the operation is `SELECT INT`, the address of the third operand will increase by 8 when VIR-V has processed 64 pairs of vectors. If there are still remaining operands,

the state machine moves into the `fetch1` state and the addresses of the first and second operands will increase by `(batchSize*8)`; otherwise, it moves into the `respond` state. For all operations except `PERMUTE INT`, the base address should move forward by `(batchSize*8)`. `PERMUTE INT` keeps the base address the same throughout the entire execution.

**`respond`** This state is to respond that the writeback is finished. When the response is completed, the state machine moves into the `idle` state.

Table 2.1: List of control signals.

| Name | Type | Function | Target(s) |
|---|---|---|---|
| `busy` | Boolean | Indicates if the accelerator is operating. | General ALU, permute unit |
| `accelReady` | Boolean | Indicates if the accelerator is ready for new tasks. | RoCC interface |
| `shouldFetch` | Boolean | Indicates if the accelerator needs to fetch data. | Data fetcher |
| `rs1Fetch` | Boolean | First operand read signal. | Data fetcher |
| `rs2Fetch` | Boolean | Second operand read signal. | Data fetcher |
| `rs3Fetch` | Boolean | Third operand read signal. | Data fetcher |
| `baseAddress` | Unsigned Integer | Initial address of store operation. | General ALU, permute unit, data fetcher |
| `numToFetch` | Unsigned Integer | Number of fetched data. | Data fetcher |
| `shouldExecute` | Boolean | Indicates if the operation should be executed. | General ALU, permute unit |
| `writebackReady` | Boolean | Indicates if the accelerator is ready for writeback. | Data fetcher |
| `responseReady` | Boolean | Indicates if the response to main processor is ready. | VIR-V |

## 2.5 Full-Program Simulation

We developed bare-metal programs in C to calculate the results and compare them with the results from VIR-V to verify the VIR-V's functions. The basic format of these test cases is shown in Figure 2.8. Arrays `a[N]` and `b[N]` are two input vectors, and array `c[N]` is to store the results returned by the C program. `ROCC_INSTRUCTION_S` is an instruction to send single operand to RoCC. For example, in Figure 2.8, `ROCC_INSTRUCTION_S` sends

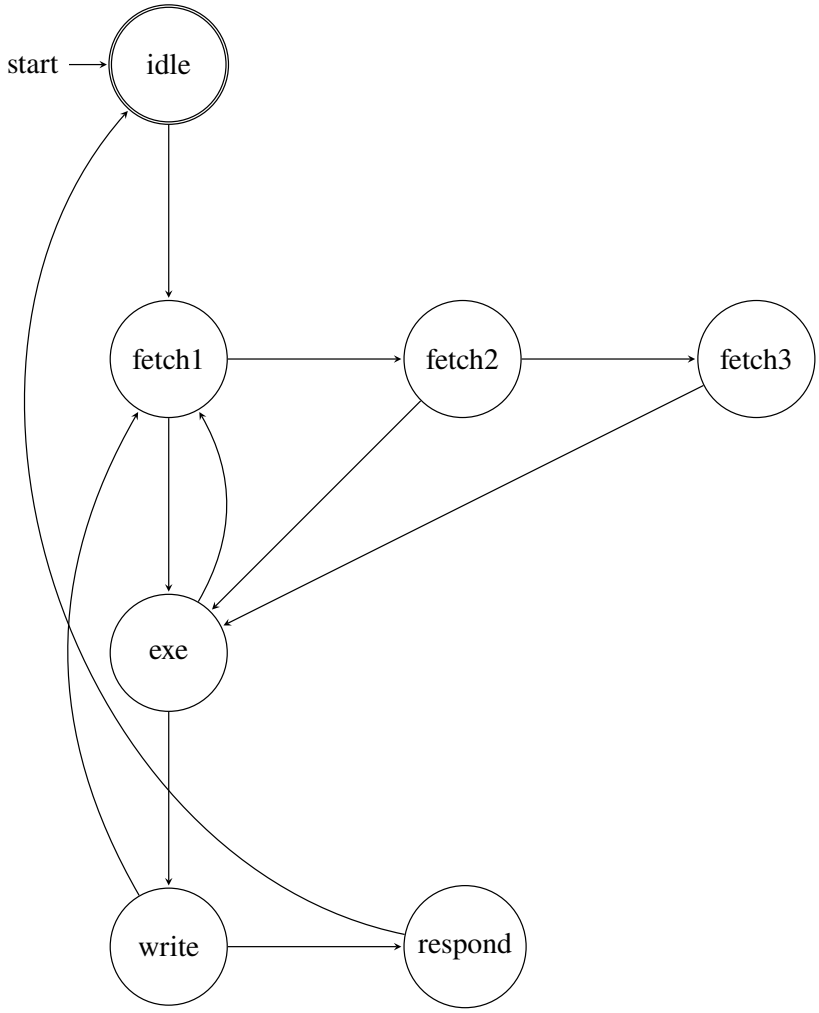


Figure 2.7: State machine of VIR-V's operation (transition conditions are in the text of Section 2.4).

`SET_NUM_OPERANDS` (`0x40`) first to define the length of input vectors, and then sends `SET_DEST_ADDR` (`0x41`) to set the destination vector base address for the next vector operation. `SET_THIRD_OPERAND` is only used in the text of `SELECT INT` operation. `ROCC_INSTRUCTION_DSS` sends the addresses of two input vectors and the type of operation to the RoCC to execute the calculation and return `status`, which represents a successful calculation if it equals to 0.

The C program will calculate by traversing and operating each pair of input vectors and write the results in `expected [N]`. If the calculation in the coprocessor is successful, the comparison between the results from the VIR-V and C program will begin. The comparison will be interrupted if two results are different and it will return the index of results that do not match. This bare-metal program returns 0 if the operation is finished by VIR-V and VIR-V's results equal to the expected values. In the Verilator simulations, we configured the lane width (denoted as `batchSize` in the Chisel code) as 2 and 8 respectively for each test case.

```c
#include <rocc.h>
#include <stdint.h>
#define NUM_ELEMENTS N

int main() {
    int64_t c[N],status;
    int64_t a[N] = {X, Y, Z, ...};
    int64_t b[N] = {A, B, C, ...};
    // status = 0 means success, non-zero indicates error
    ROCC_INSTRUCTION_S(0, N, 0x40); // Send size of vector
    ROCC_INSTRUCTION_S(0, &c, 0x41); // Send destination address
    ROCC_INSTRUCTION_DSS(0, status, &a, &b, FN); // Perform operation defined by FN

    /* Compute expected result in software for comparison */
    int64_t expected[N];
    for(int i = 0; i < N; i++) {
        // expected[i] = result of op(a[i], b[i]) as defined by FN
        // e.g., if FN = PLUS_INT, then expected[i] = a[i] + b[i]
    }

    /* Compare hardware result (c) with expected result */
    int arrays_equal = 1;
    if (status == 0) {
        for(int i = 0; i < N; i++) {
            if(c[i] != expected[i]) {
                arrays_equal = 0;
                return i+1; // return mismatch position
            }
        }
    }
    else {
        return 10; // hardware returned error status
    }
    return ((status == 0) && arrays_equal) ? 0 : 4;
```

Figure 2.8: Basic template of test cases.

# CHAPTER 3
# BENCHMARKING

## 3.1 FireSim

FireSim is an FPGA-accelerated, cycle-exact simulation platform developed at UC Berkeley that enables accurate and scalable simulation of large computing clusters directly on public cloud infrastructure [10]. Unlike traditional hardware simulators, which often involve prohibitive costs, complex maintenance, and limited scalability, FireSim leverages Amazon EC2 F1 cloud instances, which provide remote access to real, physical FPGAs hosted in AWS data centers, integrating large-scale FPGA-based simulation with software-driven network modeling. This approach significantly reduces costs and improves accessibility, allowing users to rapidly deploy large-scale hardware experiments without upfront capital investment.

A key characteristic of FireSim is its ability to simulate detailed RTL-level designs at scale. Individual compute nodes within FireSim simulations derive directly from synthesizable RTL, enabling realistic and cycle-exact hardware modeling. Nodes in the simulated cluster run full software stacks including standard Linux distributions, and interact through detailed, user-configurable, high-bandwidth network models. FireSim's flexible design means that changes in network topology, latency, and bandwidth can be adjusted at runtime without requiring FPGA resynthesis.

Moreover, FireSim's unique cloud-based deployment model makes simulations of thousands of nodes practical, providing performance on the order of millions of instructions per second, which is orders of magnitude faster than traditional software-based cycle-exact simulators. Users interact with simulated nodes seamlessly over standard network protocols such as SSH, thus treating simulated clusters similarly to physical deployments. This combination of realism, scalability, and ease of use positions FireSim as a powerful tool for hardware-software co-design research, particularly suited for evaluating and optimizing complex accelerators such as the VIR-V RoCC accelerator discussed in this thesis.

## 3.2 VIR-V Benchmarking

To quantitatively assess the performance benefits of the VIR-V accelerator, we selected the Xilinx VCU118 XDMA [11] as the FPGA platform in the simulations on FireSim. And we conducted a series of microbenchmark experiments comparing the accelerator's execution efficiency against a baseline CPU implementation without the accelerator. The benchmarks encompass a representative set of vector operations, including `+ INT`, `* INT`, `+_REDUCE INT`, `+_SCAN INT`, and `PERMUTE INT`. Each of these corresponds to a specific hardware operator within VIR-V, such as element-wise arithmetic, reductions, prefix scans, and permutations, evaluating its correctness and performance under realistic workloads. Each microbenchmark takes 100-million-element long input vectors for each input and is generated entirely using the Village project's toolchain, demonstrating that

our functional units can be implemented directly in hardware, and can be lowered to via our toolchain. Both the baseline CPU implementation and the VIR-V-accelerated version are compiled from the same NESL source programs using an identical compilation flow, ensuring a fair comparison by eliminating frontend variability. It is important to note that these benchmark results are preliminary and conducted against the in-order Rocket core as the baseline. Comparisons with more aggressive out-of-order cores such as BOOM are left for future work, and may yield different performance dynamics. In addition, the design was synthesized with the lane width configured as 8. The operating frequency of the VCU118 FPGA during simulations was set to 90 MHz.

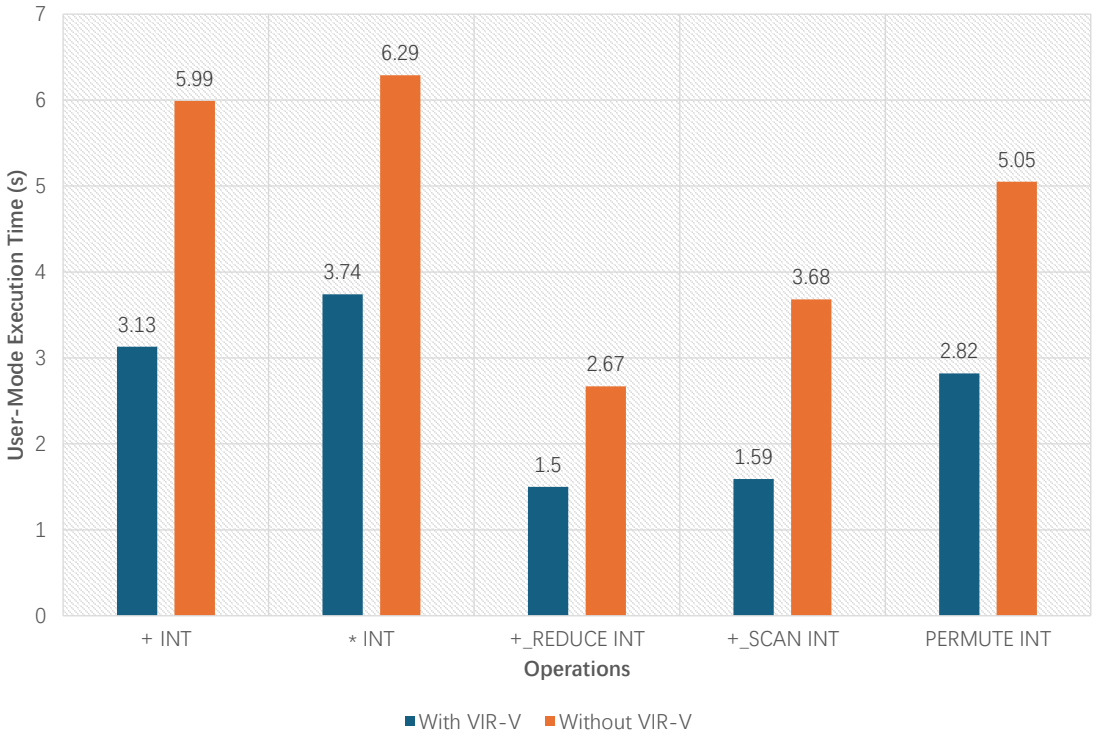


Figure 3.1: User-mode execution time with VIR-V accelerator compared to CPU execution without VIR-V (lower is better).

As illustrated in Figure 3.1, the user-mode execution times are significantly reduced across all tested operations when they are offloaded to VIR-V. For instance, the `+ INT` operation exhibits a substantial reduction in execution time, from 5.99 seconds without VIR-V to 3.13 seconds with the accelerator enabled. Similar improvements are observed for other operations, such as `* INT` (6.29s to 3.74s), `+_REDUCE INT` (2.67s to 1.50s), `+_SCAN INT` (3.68s to 1.59s), and `PERMUTE INT` (5.05s to 2.82s). Figure 3.2 presents the corresponding speedup values, normalized against the non-accelerated execution baseline (i.e., 100% indicates equal performance). The results reveal consistent speedups across all benchmarks, with `+_SCAN INT` achieving the highest acceleration at 231%, followed by `+ INT` (191%), `PERMUTE INT` (179%), `+_REDUCE INT` (178%), and `* INT` (168%).

These results clearly demonstrate the effectiveness of VIR-V in enhancing the computational throughput of vectorized operations. The observed speedups, ranging from 68% to over 130% improvement, validate the design objectives of VIR-V in accelerating VCODE computing. Structurally, these gains can be attributed to several factors. First, VIR-V offloads key data-parallel operations — such as addition, multiplication, segmented reduction, scan, and permutation — into dedicated hardware pipelines, eliminating the control and memory overhead associated with general-purpose CPU execution. Second, by operating on flattened segmented vectors derived from VCODE, VIR-V simplifies control logic and avoids dynamic thread management, allowing for highly efficient pipelined execution. Furthermore, the use of the RoCC interface enables

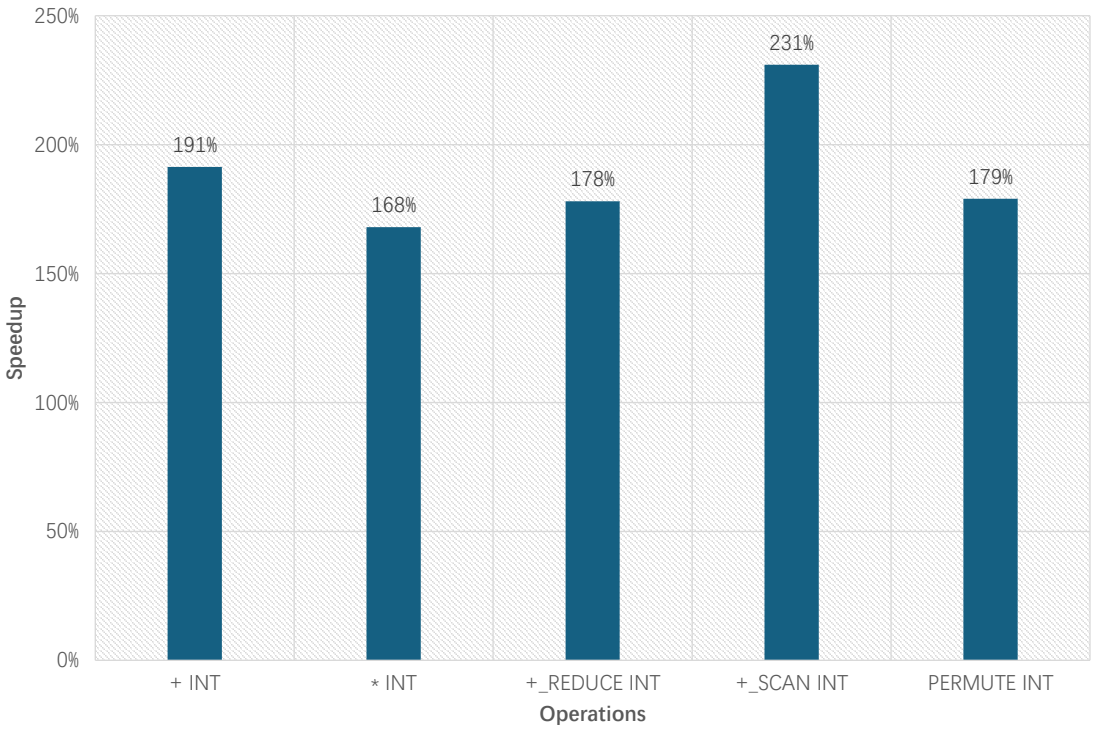


Figure 3.2: Speedup of microbenchmarks with VIR-V accelerator compared to CPU execution without VIR-V. 100% indicates equal runtimes.

low-latency communication between the processor and accelerator without incurring system-level software overhead. Significantly, the consistent performance gains across different vectorized operations indicate that VIR-V offers robust support for a wide variety of data-parallel workloads.

# CHAPTER 4
# CONCLUSION

## 4.1 Results

This thesis presents the design, implementation, and evaluation of VIR-V, a RISC-V RoCC accelerator for VCODE computing. We introduce a hardware architecture that directly supports a subset of VCODE operations, providing a lane-oriented SIMD design capable of efficient vectorized computation. The VIR-V accelerator is implemented in Chisel, integrated into the Rocket-Core via the RoCC interface, and verified through extensive simulations and functional tests.

The microbenchmarks demonstrate that VIR-V delivers substantial performance improvements across key vector operations. Specifically, the accelerator achieves an average speedup of 1.89× over CPU-only execution, confirming its ability to effectively offload and accelerate data-parallel workloads. These results validate the core design objectives of VIR-V and highlight its potential for enhancing the execution efficiency of high-level parallel programming models within the RISC-V ecosystem.

## 4.2 Future Work

We are currently extending VIR-V to support floating-point operations. The same frameworks that we have already developed will apply to these, and the primary con-

cern is which FPU implementation to use. There are two potential options to implement this feature. One is integrating the floating-point units (FPUs) in Chipyard with our accelerator. These FPUs, which are based on the HardFloat library and used in the Linux-capable Rocket cores supporting RV64GC, provide IEEE-754-compliant operations and are modularly compatible with Chisel-based designs. Leveraging HardFloat will enable our accelerator to support floating-point vector operations with minimal integration overhead [9]. The other is generating a customized FPU, which will be helpful to meet specific requirements of VCODE floating-point arithmetic. To customize an FPU, FloPoCo [12] is an available FPU generator that provides Verilog implementations of floating-point operations following the IEEE 754 standard. We are evaluating the feasibility and engineering effort of the options mentioned above.

Our next step is to transform VIR-V into a deeply pipelined streaming processor. This approach would enable us to fully exploit the inherent parallelism present in the relevant VCODE instructions, as highlighted by the work and depth complexity analyzes in Table 1.1.

We will also explore the memory connection strategy for the accelerator. Utilizing the L1 data cache could leverage data locality, particularly when the main processor accesses the vector immediately before or after the accelerator. However, this may lead to excessive L1 cache usage, potentially causing performance degradation due to the high volume of reads and writes. In such cases, a more effective design would involve connecting the accelerator to the TileLink-based L2-crossbar, which interfaces directly

with the system bus and bypasses the L1 cache. For operations involving very large vectors, the optimal approach will connect the accelerator directly to main memory if possible.

# REFERENCES

[1] G. E. Blelloch, *NESL: A nested data-parallel language (version 3.1)*. Citeseer, 1995.

[2] G. E. Blelloch and J. Greiner, "A provable time and space efficient implementation of nesl," *ACM SIGPLAN Notices*, vol. 31, no. 6, pp. 213–225, 1996.

[3] G. Belloch, S. Chaterjee, J. Sipelstein, and M. Zagha, *Vcode reference manual (version 2.0)*, 1994.

[4] D. Li, H. Wu, and M. Becchi, "Nested parallelism on gpu: Exploring parallelization templates for irregular loops and recursive computations," in *2015 44th International Conference on Parallel Processing*, IEEE, 2015, pp. 979–988.

[5] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 53–67.

[6] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225, ISBN: 9781450311991.

[7] M. Odersky, P. Altherr, V. Cremet, *et al.*, "An overview of the scala programming language," 2004.

[8] K. Asanović, R. Avizienis, J. Bachrach, *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[9] K. Asanović, R. Avizienis, J. Bachrach, *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[10] S. Karandikar, H. Mao, D. Kim, *et al.*, “Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[11] AMD Adaptive Computing, *Vcu118 evaluation board user guide*, v1.5, UG1224, Document No. UG1224 (v1.5), AMD, Mar. 2023.

[12] F. De Dinechin and B. Pasca, “Designing custom arithmetic data paths with flopoco,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

# APPENDIX A

# LIST OF OPERATORS REALIZED IN VIR-V

Table A.1: List of VCODE Instructions in VIR-V

| No. | Type | Name |
|---|---|---|
| 1 | Element-wise | + INT |
| 2 | Element-wise | - INT |
| 3 | Element-wise | * INT |
| 4 | Element-wise | / INT |
| 5 | Element-wise | % |
| 6 | Element-wise | < INT |
| 7 | Element-wise | <= INT |
| 8 | Element-wise | > INT |
| 9 | Element-wise | >= INT |
| 10 | Element-wise | = INT |
| 11 | Element-wise | != INT |
| 12 | Element-wise | LSHIFT |
| 13 | Element-wise | RSHIFT |
| 14 | Element-wise | NOT BOOL |
| 15 | Element-wise | AND BOOL |
| 16 | Element-wise | OR INT |
| 17 | Element-wise | XOR BOOL |
| 18 | Element-wise | SELECT INT |
| 19 | Scan | +_SCAN INT |
| 20 | Scan | *_SCAN INT |
| 21 | Scan | MAX_SCAN INT |
| 22 | Scan | MIN_SCAN INT |
| 23 | Scan | AND_SCAN BOOL |
| 24 | Scan | OR_SCAN BOOL |
| 25 | Scan | XOR_SCAN BOOL |
| 26 | Reduction | +_REDUCE INT |
| 27 | Reduction | *_REDUCE INT |
| 28 | Reduction | MAX_REDUCE INT |
| 29 | Reduction | MIN_REDUCE INT |
| 30 | Reduction | AND_REDUCE BOOL |
| 31 | Reduction | OR_REDUCE BOOL |
| 32 | Reduction | XOR_REDUCE BOOL |
| 33 | Permutation | PERMUTE INT |