CS535 Project Report

RISC V[ECTOR]

Siddarth Suresh, Benjamin Dunaisky

April 30th, 2025

General Architecture Overview:

RISC V[ECTOR] is a *general purpose* RISC CPU architecture. It is a clean-sheet design, but takes some inspiration from the RISC-V specification in instruction representation and design. The distinguishing feature of the **RISC V[ECTOR]** architecture is that it *provides basic support for vector types*.

The *word size* is 32 bits (4 bytes). As typical for RISC architectures, **RISC V[ECTOR]** uses a *fixed* instruction encoding and a *load/store architecture* for categorizing instructions.

RISC V[ECTOR] relies on the following submodules and projects:

RAM ("**R**AM **A**cts **M**agically") is a memory subsystem. It provides a library of classes which can be assembled together with an arbitrary number of levels, cache sizes, and ways.

rva (the **R**ISC **V**[ECTOR] **A**ssembler) is an assembler which provides a programming interface and syntax similar to MIPS. It outputs a binary file in big endian byte-ordering consisting of assembled instructions followed by user-defined data.

General Engineering Methods

The **RISC V[ECTOR]** simulator is split into three code repositories, each utilizing a unique engineering approach. The code is licensed under the copyleft *GPLv3 license* and is found publicly here:

- https://github.com/bdunahu/rva.git
- https://github.com/bdunahu/ram.git
- https://github.com/bdunahu/RISC-V-ECTOR-.git

rva

The **rva** assembler is written in Common Lisp, using the SBCL implementation. This software setup was chosen due to Lisp's excellence in quickly transforming lists of tokens. *A test-driven engineering approach was used for the lexing and parsing phase, using the fiveam framework*. Further phases do not have a test suite, but are trivial in design.

Currently, valid programs include both a `.data` and `.text` section, in that order. *Named variables in the `.data` section can be used in place of a displacement field in the instructions in the `.text` section*—for example, `load $6 rabbits($0)` stores the value of 'rabbits' into register 6. `addi $6 $0 rabbits` on the other hand, effectively stores a pointer to rabbits in register 6, which is useful in general and required for vectors.

This is possible because **rva** always places user-defined variables directly after the code section in the resulting binary, and always stores the code at memory address zero. For convenience, **rva** always places a HALT instruction (division by 0) at the end of each `.text` section.

To assemble a program, simply run `` `./bin/rva /path/to/program.asm' ``. This will output a binary (`.rv`) in the directory of the input program which can then be directly uploaded to the RISC V[ECTOR] GUI. If the program is malformed, ***rva** will instead output the column and line number, code context, and a hint on the expected syntax*. Note that **rva** will not attempt to assemble programs which do not end with the extension `` `.asm' ``.

rva also can be called with multiple flags. `` `--parse' `` will output an AST of the input program. This *AST is an s-expression, and itself represents a valid Lisp program*! This keeps the implementation simple—after the AST is parsed, it is simply evaluated in an environment which contains function definitions for all of the node labels.

The result of executing the AST program is a single list of raw instructions. `` `--emit' `` will cause the assembler to output these raw instructions in binary rather than write them to a file. This simple engineering approach allotted by Common Lisp is both unique and efficient.

RAM

RAM is written in C++ due to its speed, closeness to hardware, and support for object-oriented programming. *A test-driven engineering approach was used for testing the full DRAM class, written using the Catch2 framework*. *Most of the Cache, including basic multi-level functionality, are also tested*. **RAM** previously contained a CLI script to interact with. However this script was unmaintained and removed from the codebase, so the user's interaction with **RAM** is now entirely through the **RISC V[ECTOR]** pipeline simulation and GUI.

The recommended object-oriented approach was followed—both the DRAM and Cache objects inherit from a single `Storage' object which contains common attributes, such as a unique identifier to the class currently being serviced. Additionally, the `Storage' constructor accepts another storage device as an argument, which provides a simple interface for 'building' a storage subsystem with *any number of cache levels, each of which operate using an independent size and number of ways*.

The cache was not originally designed to operate with a variable number of ways and utilized a single 1-dimensional array to represent cache lines. Ways were implemented by including a new class method to translate an index into a set of ways into the true index that set of ways is located in the 1-dimensional array of cache lines, *abstracting away the fact that ways are present from the rest of the cache logic*. This was kept generic to allow setting a configurable number of ways.

Currently, **RAM** supports more configurations than the **RISC V[ECTOR]** user interface allows (including fully associative). This is simply because allowing the user to assemble any size, ways, and numbers for their cache objects would require more input validation than we wanted to focus on.

RISC-V[ECTOR]

RISC V[ECTOR] is also written in C++, and the QT application framework. While some unit tests were created for individual pipeline stages, most of the testing came from running test programs through the GUI itself (integrated testing), which was not automated. Luckily, running the same significantly complex programs after each code change was enough to verify no regressions occurred.

RISC V[ECTOR] can be further divided into two subprojects, *the GUI and the pipeline, each of which is run on a different thread*.

To start the **RISC V[ECTOR]** program, simply follow the instructions in the README and execute the compiled binary: `./build/risc_vector'. A QT window will appear.

Buttons, checkboxes, sliders, and number selectors on the right facilitate configuration of cache levels, the pipeline (on or off), uploading files, and stepping through the program. Above the `Step' button is a slider which can be used to advance the program more quickly. Note that only `.rv` files, which are produced by **rva,** may be uploaded.

The user may interact with these buttons in any order, but if no program is uploaded and the `Initialize!' button is pressed, a small talking robot with a rather disgruntled face will appear on the bottom right and inform the user. The `Initialize!' button can be pressed at any time, and the simulator will be reset and reconfigured with the new user options.

The left side of the screen will initially be blank, but during simulation, the top left will populate a list of tabs displaying the register contents, DRAM and cache levels. These lists cannot be searched, but can be navigated with `PageUp`, `PageDn`, or an interactive scrollbar. The bottom right displays what instruction each of the five pipeline stages are currently processing. The mnemonic and squashed status are displayed, as well as three `slot' number fields, which contain various data (usually the contents of active registers). Please see the README of the **RISC V[ECTOR]** project for a screenshot of the GUI.

The pipeline itself is developed using a similar object-oriented approach as the storage devices in **RAM**. *All five stages inherit from a virtual class called* `Stage', which defines static attributes for data items which need to be accessed by more than one stage: the storage devices, the registers, and if the pipeline is empty (for if the pipeline is turned off).

A `Controller' class also inherits from the `Stage' class. The controller keeps track of the clock, and facilitates requests from the GUI by running for only a set number of cycles.

To ensure data is passed sequentially without collisions, `Stage' defines a default ‘advance‘ method. The ‘advance‘ *method is what each stage uses to interact with each other*, and accepts a single parameter representing the caller's current status: “Busy” or “Ready”. If the callee stage is also “Ready”, then a DTO object for the instruction, represented as a struct with some nested union fields, is passed to the caller.

The union allows for each DTO to neatly pack both vector and integer data types without unnecessary fields. The ‘slots’ provided in each union are utilized carefully by Execute, Memory, and Writeback—wherever possible, ALU operation results are placed into the first ‘slot’ as a convention. *RAW hazards are accounted for using a queue object*—registers being written to are first placed into the back of the queue by Decode, and Writeback pops each out as they are written back. Decode generates a stall if an instruction requests to read from a register already in the queue.

Also in the ‘`advance`‘ method is a call to ‘`advance_helper`’, *which is the main business-logic function each pipeline stage inheriting from* `Stage` *defines*. The controller facilitates movement through the pipe by always calling Writeback with “READY”.

Though the full implementation details are fairly complex and have a lot of edge cases, special consideration was given to ensuring the code base was clean—all three code bases had significant rewrites and iterations as progress was made towards new goals: adding multi-level cache, PUSH/POP instructions, and vector support to name a few.

Types and type operations

All integer types are *signed* and *32 bits*. Signed bits are represented using *two’s complement.*

The *arithmetic operations which are supported on the integer type* are: **add, subtract, multiply, divide, modulo, arithmetic right shift, left shift**. The *logical operations which are supported are:* **bitwise and, or, not, xor**.

The *operations which are supported on the vector type* are: **vector load, vector store, element-wise add, element-wise subtract, element-wise multiply, and element-wise divide**.

Registers

To support the integer and vector data types, the **RISC V[ECTOR]** architecture includes 24 general-purpose registers: **16** integer registers (x0 to x15) and **8** vector registers (v0 to v7). Vector registers are 256 bits (8 words).

Register	Description

x0	hardwired 0
x1	link register
x2	stack pointer
x3	condition code register
x4	vector length register
x5…x15	integer general purpose
v0…v7	vector general purpose

A special register which holds the program counter is also provided, but is only addressable to the programmer through branch instructions.

The condition code register

The condition code register maintains a set of four bits that the arithmetic and compare operations set:

Bit 0: *greater than*
Bit 1: *equal to*
Bit 2: *underflow*
Bit 3: *overflow*
Bits 4-31: empty

CMP and CEV are the only instructions that set the greater than and equal to bits. Nearly all arithmetic operations set the underflow/overflow bits. There is a separate conditional branch instruction for each bit. This is documented below.

Instructions & addressing modes

Memory model and addressing modes

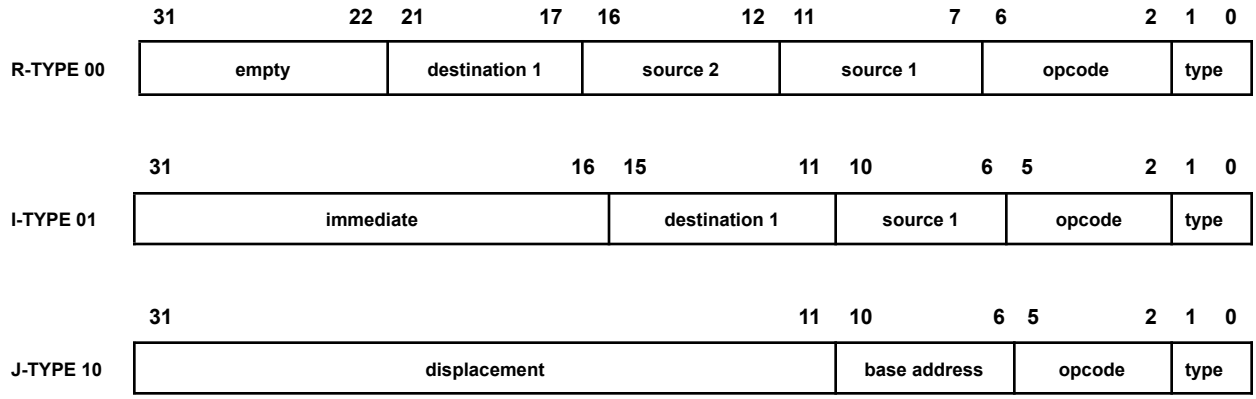
The maximum number of operands per instruction is three, fetching a *single instruction per word*. It uses a Princeton *memory organization* and its *address unit* is a full word. The address range is 64 kilobytes. (16384 words)

Similar to RISC-V, our architecture supports *three addressing modes: register, immediate, and displacement*. The size of the displacement field varies per instruction type, but is proportional to the entire 64 kilobytes address range.

Register indirect addressing is accomplished by placing a value of 0 in the 16-bit displacement field, and *absolute addressing* is accomplished by using the constant 0 register as the base register. The addressing modes used for each instruction are encoded into the opcode.

Instruction Types

Being a *load/store architecture*, the supported instructions for ALU operations, loads, stores, and jumps can be grouped into four types: register to register (R-TYPE), ALU immediates and loads/stores (I-TYPE), and jumps/subroutine management (J-TYPE).



Immediate fields and accessing invalid memory addresses

All immediate fields are treated as a signed integer. In the case an instruction (jump, load, etc.) attempts to access an address larger or smaller than the address space, **RAM** ensures the memory address that is *actually* accessed is:

```
((a < 0) ? ((a % MEM_WORDS) + MEM_WORDS) % MEM_WORDS
         : a % MEM_WORDS)
```

Where `a` is the requested address and `MEM_WORDS` is the total address space. This ensures out of bound memory addresses cleanly wrap around.

Instructions supported

A comprehensive list of instructions supported by **RISC V[ECTOR]** and the **rva** are listed below. *All undocumented invalid instructions* (including non-existent opcodes or other nonsensical instructions) *will be treated as a no-op*.

Mnemonic	Opcode	Format	Operation
ADD	00001	R	Adds the value in source register 1 to source register 2. Sets the result to the destination register. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
SUB	00010	R	Subtracts the value of source register 2 from source register 1. Sets the result to the destination register. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
MUL	00011	R	Multiplies the value in source register 1 to source register 2. Sets the result to the destination register. The destination register holds the least significant 32 bits in case of overflow. The upper 32 bits are discarded. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
QUOT	00100	R	Divides the value in source register 1 with the value in source register 2. Sets the destination register with the *quotient* of the result. Division by 0 sets all the bits in the destination register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0. *Division by the zero register results in a halt.*
REM	00101	R	Divides the value in source register 1 with the value in source register 2. Sets the destination register with the *remainder* of the result. Division by 0 sets all the bits in the destination register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0. *Division by the zero register results in a halt.*
SFTR	00110	R	Arithmetic right shifts the value in source register 1

			by the value stored in source register 2. Sets the result to the destination register. Shifts in the right direction shifts all of the bits to the right and fills the upper bits with a copy of the previous most significant bit. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the register to the smallest positive number (0, in two's complement representation) or -1 in case source register 1 was negative. Does not modify the condition code register.
SFTL	00111	R	Arithmetic left shifts the value in source register 1 by the value stored in source register 2. Sets the result to the destination register. Shifts in the left direction shifts all of the bits to the left and fills the lower bits with a zero. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the destination register. Does not modify the condition code register.
AND	01000	R	Performs a bitwise AND between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions to 0.
OR	01001	R	Performs a bitwise OR between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions to 0.
NOT	01010	R	Performs a bitwise NOT for the value in source register 1. Sets the result in the destination register. Values in source register 2 are ignored. Always sets the overflow and underflow conditions to 0.
XOR	01011	R	Performs a bitwise XOR between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions

			to 0.
ADDV	01100	R	Performs element-wise addition between the vector stored in source vector register 1 and the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
SUBV	01101	R	Performs element-wise subtraction of the vector stored in source vector register 1 from the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
MULV	01110	R	Performs element-wise multiplication between the vector stored in source vector register 1 and the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. *Each resulting word stored in the destination register is limited to its least significant 32 bits in the event of an overflow.* Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
DIVV	01111	R	Performs element-wise division between the vector stored in source vector register 1 and

			vector stored in source vector register 2. Sets the *quotient* of the division (element-wise) in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. *Division by 0 within a vector sets all the bits in the corresponding word in the destination vector register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0.*
CMP	10000	R	Sets the greater than condition bit in condition register if value in source register 1 is greater than value in source register 2. Sets the equality condition bit in condition register if value in source register 1 is equal to value in source register 2.
CEV	10001	R	Sets the equality condition bit in the condition register if the value in source vector register 1 is equal to value in source vector register 2. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the instruction is treated as a no-op. If the value in the vector length register is greater than 8, it is capped at 8.

I Type

Mnemonic	Opcode	Format	Operation
LOAD	0001	I	Loads the value present at the memory address calculated by the sum of the value in the source register and the immediate value into the destination register.
LOADV	0010	I	Loads the value present at the memory address calculated by the sum of the value in the source register and the immediate into the destination vector register. The exact number of words copied is determined

			by the vector length register. If the value in the vector length register is greater than 8, it is capped at 8.
ADDI	0011	I	Adds value in the sign-extended immediate to the value in source register and stores the result of the sum in the destination register. Sets the overflow/underflow condition codes to either zero or one depending on if one occurs.
SUBI	0100	I	Subtracts value in the sign-extended immediate from the value in source register and stores the result of the difference in the destination register. Sets the overflow/underflow condition codes to either zero or one depending on if one occurs.
SFTRI	0100	I	Arithmetic right shifts the value in source register 1 by the sign-extended immediate. Sets the result to the destination register. Shifts in the right direction shifts all of the bits to the right and fills the upper bits with a copy of the previous most significant bit. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the register to the smallest positive number (0, in two's complement representation) or -1 in case source register 1 was negative. Does not modify the condition code register.
SFTLI	0101	I	Arithmetic left shifts the value in source register 1 by the sign-extended immediate. Sets the result to the destination register. Shifts in the left direction shifts all of the bits to the left and fills the lower bits with a zero. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the destination register. Does not modify the condition code register.
ANDI	0110	I	Performs a bitwise AND between the value in source register and sign-extended immediate and stores the result in the destination register.
ORI	1000	I	Performs a bitwise OR between the value in source register and sign-extended immediate and

			stores the result in the destination register.
XORI	1001	I	Performs a bitwise XOR between the value in source register and sign-extended immediate and stores the result in the destination register.
STORE	1010	I	Stores the value present in source register in memory at the address calculated by sum of value in base address register and value in displacement field in memory.
STOREV	1011	I	Stores the value present in the source vector in memory starting at the address calculated by sum of value in base address register and value in displacement field. The exact number of words copied is determined by the vector length register. If the value in the vector length register is greater than 8, it is capped at 8.

J Type:

Mnemonic	Opcode	Format	Operation
JMP	0001	J	Performs unconditional jump to the address in memory calculated by sum of the value in base address register and value in the immediate field.
JRL	0010	J	Performs unconditional jump to the address in memory calculated by sum of the current PC and value in the immediate field. The value in the base register is ignored.
JAL	0011	J	Sets the return address register to the memory address of the next instruction and performs an unconditional jump to the address in memory calculated by sum of the value in base address register and value in the immediate field.
BEQ	0100	J	Checks the EQ (equal) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
BGT	0101	J	Checks the GT (greater than) condition in the condition code register and branches to the

			memory address calculated by sum of value in PC and the value in the displacement field.
BUF	0110	J	Checks the UF (underflow) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
BOF	0111	J	Checks the OF (overflow) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
PUSH	1000	J	Increments the stack pointer by four bytes, writes the value in the base address register to the memory address pointed to by the stack pointer.
POP	1001	J	Loads the memory address pointed to by the stack pointer into the base address register, then decrements the stack pointer by four bytes.
RET	1010	J	Performs an unconditional jump to the address stored in the link register.

Memory subsystem, stack management & subroutine support

As mentioned, **RAM** uses a *configurable level cache*, the size of which is generated depending on the number of cache levels used, using this formula:

```
lines = 2 ^ (5 + (level / total levels) * (12 - 5))
```

This ensures even cache spacing between 2^5 and 2^12, but complicates benchmarking. Note that the size of DRAM is always 2^14 lines.

Additionally, cache is 2^k-way *set associative, write back, and write allocate on a write miss*. The replacement policy is least recently used. The size of the cache line is 4 words.

RISC V[ECTOR] provides lots of subroutine support. A special stack pointer (x1) register to support subroutine calls is provided in conjunction with the push and pop instructions. *A single dedicated return register is provided* to support subroutine calls and returns with the JAL and RET instructions. It is up to the programmer or compiler to manage pushing/popping the return register on the stack for deeper subroutine calls.

The programmer also will likely want a frame pointer for this, but the convention used is up to them. See the example programs in the **rva** repository for more information.

Management plan

We will use a version control system and github for storage of project code and documentation files. The team will maintain a list of active issues to work on, and utilize unit testing using the Catch2 library. Pull requests will always be used to ensure merged code is always functional and peer reviewed. The team will communicate via email, text, and phone, and meet in person Tuesday and Saturday at 11am to review our status and plan work for the next week, or Zoom if that is not possible.

Initially, we will work together to create a set of github issues related to initial APIs for interfacing between the memory and pipeline in C++. We will also create issues relating to the API between the simulator and Python process responsible for the GUI, using the Python.h library to convert the minimal number of data structures necessary for displaying the memory layout. We plan to each claim issues on both sides, so that we each maintain a full understanding of how the simulator works together, and each part is built up simultaneously in pieces.

Finally, as Siddarth starts developing the assembler, Benjamin will continue building out the instruction set. Then we will divide the benchmark work between us. To show the effect of our vector extension, we will compare the benchmarking performed for matrix multiplication using the vector extension and non-vector baseline.

At the end, Siddarth will be responsible for the portion of the report that describes the simulator and using it through the UI while Benjamin focuses on reporting the effects of the benchmarking under the different modes. We will each write a section about what we learned, and who ended up doing what on the project.

Efforts by Person

Project Phase 1:

During project phase 1, Benjamin implemented the initial constructors and method declarations for the storage, DRAM, and cache classes with some simple existence tests. Soon after, Siddarth worked on fleshing out the program entry point, arg parsing, and command line interface for the first demo, while Benjamin added helper methods to parse memory addresses along with a method to allow DRAM to write an address.

Later, Siddarth implemented DRAM methods to read a single address and a full line and a method to properly wrap memory accesses, while Benjamin worked on a method to write a line to Cache and a method to fetch a line from memory on a Cache miss using a write allocate policy.

Next, Siddarth implemented methods to read both an address and a full line from cache, as well as write a line to DRAM, while Benjamin added the ability to write a line to Cache, and finished the CLI script for the first demo.

Project Phase 2:

During project phase 2, Benjamin first cleaned up the storage device implementation (with demo fixes), and started work on the generic stage and controller objects for the pipeline. Meanwhile, Siddarth began learning the QT library and similarly implemented the initial GUI.

Benjamin then templeted out the five pipeline stages, and filled in the initial fetch, decode, and execute stages. During this time, Siddarth continued iterating on the GUI layout, and connected it to the previous storage devices, as well as the controller object.

Next, Benjamin added the rest of the execute instructions (with the exception of subroutine and vector instructions), while Siddarth pivoted to the pipeline, filling in the initial memory and write back stages.

Next, Siddarth modified the GUI and pipeline to work on separate threads, while Benjamin began integration-testing the pipeline and fixing all the bugs. The bugs prevailed, so Siddarth then joined to help verify the pipeline was working properly.

Project Phase 3:

During project phase 3, Benjamin began work on and completed the full assembler, while Siddarth implemented the feedback from the previous demo by adding a widget in the GUI to view instructions as they are processed in each stage of the pipeline.

Next, Benjamin began cleanup of the cache and pipeline, and separated the cache into its own repository (**RAM**). Additionally, he rewrote a few aspects of the cache to handle multiple levels. During this time, Siddarth implemented the GUI logic required to load a program dynamically into the simulation.

Finally, Benjamin implemented logic to support the specialized PUSH and POP instructions, and wrote a few test programs to test the pipeline and demonstrate the functioning of the majority of the instruction set, aided by Siddarth to ensure the pipeline met the requirements for the second demo.

Project Phase 4:

During project phase 4, Benjamin created a few new widgets for the GUI, including those for tab display, and to configure the cache for multiple levels and ways, while Siddarth refactored the pipeline with templates and added the full vector extension: ADDV, SUBV, MULV, DIVV, LOADV, and STOREV to the pipeline.

Next, as Benjamin added the JAL and RET instructions, added multiple ways to the cache, and added overflow and underflow condition codes, Siddarth fixed numerous small display issues in the new GUI and register display.

Finally, as Benjamin developed the benchmark programs and began timing, Siddarth finished the vector instructions, adding overflow and underflow checks, while we both collectively finished the timing results and began analyzing them.

Benchmarks and Performance

RISC V[ECTOR] supports a highly configurable cache. Users may select a number of cache levels up to 6 and configure the number of ways for each independently. A constant parameter across all our benchmarks is that the number of words per line is 4.

DRAM delay is 100 cycles, while level 1 cache is a single cycle—successive cache levels increase by a factor of `5: 5, 10, 15, …, 5(k-1)` for subsequent `k` caches.

DRAM is always `2^14` lines. With the benchmarks above, a single cache is `2^7` lines. With two cache levels, level one cache is `2^5` lines, where level two cache is `2^10` lines. A greater number of levels were not tested using these configurations, as we found none of the programs made use of enough data to spill from level 2 cache.

	L0	L1 W1	L1 2W	L1 4W	L1 8W	L1 16W	L2 1W,1W	L2 2W,1W	L2 2W,2W	B/W
exchange-sort	1303724858	63843396	62482219	62466160	62766160	63376776	53187322	52519828	52519828	54.70%
	1272247534	35754481	33817499	33801440	34093633	34988493	24600666	23855122	23855122	
hard-matrices	116288483	8311146	7521831	7852101	8682725	9619702	5708895	5708624	5707715	44.50%
	112915219	5205410	4433132	4763402	5594026	6570744	2613121	2619139	2618382	
primes-generator	1002794943	57100446	57087417	57087417			57084006	57082724	57082724	45.00%
	964786191	22287829	22273697	22273686			22270348	22269004	22269004	

Overall trends:

1. *Cache is more important than pipeline*
2. *Pipeline speeds up systems with cache*
3. *Higher levels of cache needn't always provide speedups*
4. *The benefit of ways depends on the program's temporal and spatial locality during memory accesses*
5. *The best configuration can provide massive speedups*

Benchmark 1: Exchange Sort

The first benchmark is the exchange sort program on an array containing 750 randomly initialized generated integers (see '`./input/hard-matrices.asm`' in the **rva** repository for actual code).

This benchmark keeps track of two pointers to data in the array and iterates over each, making use of a subroutine call; complete with saving off registers and pushing arguments to the stack, in order to compare elements and perform a swap.

Due to the amount of data and how many times it is iterated over, it is the slowest program.

Interesting Findings and their Analysis:

Clearly, *the more access to memory, the more having a cache matters*. *Adding a pipeline on top of cache nearly halved the amount of cycles the program took to complete, suggesting the throughput with a pipeline is doubled*. The same is not the case when no cache is used—but we note that a speedup does still occur when the pipeline is on—this implies that the throughput is still greater even though fetching an instruction still takes *100* cycles. Where does this come from?

If fetch is using the DRAM while mem is attempting to store, then after fetch finishes, there will be two instructions in the pipeline. MEM will begin storing, and during that time, the instruction just fetched can move through decode and execute. We believe this is the single reason why *a pipeline is better than no pipeline when cache is off*.

As for other results, exchange sort does make use of enough data to benefit from a second cache level (other *smaller programs were negatively impacted by another level*, implying they did not use the second level enough to make having one worthwhile).

T*he number of ways being increased did offer a slight improvement—suggesting that the addresses accessed fall evenly into the index 'bin's which represent the set of ways.* This makes sense, because *data accesses are sequential*. That means each bin will be filled evenly. Too many ways however, and it becomes clear some ways are used more frequently than others (possibly due to the stack being referenced the most).

Benchmark 2: Hard Matrices

The second benchmark is a matrix multiplication program on two randomly generated initialized matrices, one having a shape of 100 x 18 integers and the other having a shape of 18 x 36 integers.

This benchmark supports loops, conditional branches and really large data to ensure that the cache is stressed through a large number of read operations.

Interesting Findings and their Analysis:

Results were similar to the exchange sort—a cache matters first, then a pipeline (for the same reasons as above). The matrix multiply clearly also made use of a second level of cache, which is reasonable given it operates on a large amount of data.

Interestingly, *using multiple ways immediately yields worse results*. We believe the size of the matrix impacts how well the set of ways is utilized. Matrix multiply differs from exchange sort in that *data is accessed non-sequentially in the case of indexing the columns* (matrices are stored in row major order). In the case of our matrix, the locations of the data seem to have favored a certain set of ways.

Benchmark 3: Primes generator

This program uses trial division to generate primes up to 5000. This benchmark supports subroutine calls, loops and conditional branches.

It does not make use of large data structures. Rather, it accesses memory primarily through the stack and storage of found integers. While the general trends apply relating to the cache and pipeline, it gained an almost insignificant benefit from a second level of cache and number of ways!

What we learned over the past 3 months

1. **C++ for both the GUI and Simulator?:** Deciding which languages and libraries to use was not done on a whim and involved significant research and weighing of pros and cons. In the end, we decided to code our simulator in C++ because of its maturity as an OOP language, ability to perform at high speeds under stress, and closeness to hardware. For the GUI, we initially proposed to implement a scripting feature that would have made use of a live python interpreter. Python is (we think) easier and more responsive to program a GUI in. However, we soon decided that the scripting feature was not as useful as the ability to save and load the simulator state, especially if the program takes a long time to run. *Thus, we reasoned that there was no longer any reason to use Python or the added complexity of converting C objects to Python with the Python.h library* (though we were looking forward to trying this!). We instead went with Qt 6 for ease of integration with C++. *Neither of us had much C++ experience or any experience with QT*, but seeing the benefits we undertook the task anyways and accomplished much learning in this regard. The result is an application that cleanly uses multiple threads to handle both GUI and simulator work, and has a relatively good user experience.
2. **Common Lisp for the Assembler?:** Like with C++ and QT, we had little experience with Common Lisp. Lisp-like languages are unique in that they manipulate both data and code as lists, and thus have mechanisms to convert between the two. Previous experience with Scheme in compilers suggested that Common Lisp would present a concise way to iterate over a list of tokens and directly convert an AST to a stream of words (or another IR) by converting it to code using the `eval` function. While we had difficulty learning the language to construct a parser (see below) the binary emission after obtaining an AST was immediately successful, and we feel the result is exciting and innovative!
3. **One codebase or many? Learning about git submodules, git-filter-repo (tool), and the modular approach.**
   In the final product, **RAM** is a git submodule of **RISC V[ECTOR]**. It was not always so!---*the pipeline was for most of the project built directly on top of the storage devices*. Both codebases used the same enums and macros, but if they were not shared, then we found the objects could be significantly simplified (or in many cases, removed entirely). It turns out that the cache really shares no similarities to the pipeline. Separating them did not introduce duplication, but allowed each piece of code to be more closely tailored to the problem it was intended to solve. *By compartmentalizing, we simplified the development experience on both codebases, made our approach more modular, and ensured our code 'did one thing and did it well'!*
4. **A GUI is worth 1000 tests:** Having a good GUI that shows the proper state of the simulator is irreplaceable in verifying the individual pipeline stages are

functioning together with the storage. If the GUI provides insight into what objects each phase is currently working on, running a single benchmark can tell us information about how long each pipeline stage is taking, if it processes an instruction correctly independently, if it processes an instruction correctly with multiple objects in the pipe, if it processes hazards correctly, and more. If we include a button to run to completion, even small changes in the implementation output a drastically different result. As a result, unit tests were almost entirely dropped after the GUI was fully completed in phase 4 of the project, as t*est maintenance was also challenging*—small changes to the implementation often required many changes in the test files, especially as we learned from our incorrect initial attempts.

5. **…But, Automation is still Nice:** Though use of the GUI was more time-effective than unit tests, it was a chore to manually verify each change did not result in a regression. If we were to do the project again, *we would develop a system to automatically run the simulator on a set of test programs, dump the contents of the registers and memory, and diff them against a known good result*. We would use little or no unit tests!
6. **Parsing in Common Lisp—esrap versus trivia**
   'trivia' is a common lisp pattern-matcher, while 'esrap' is closer to a full parser-generator. I (Benjamin) am a scheme programmer, which provides a `match' expression very useful for parsing grammars, because you can match non-sequential tokens and easily collect the rest into a single list to pass to a different rule-parsing function. The trivia library appears similar at first glance, but lacks this feature. I tried to make this library work for a couple days, and even entertained extending its features by practicing my ability to write complex macros. I soon instead found esrap, which offered the type of string matching perfect for generating a full AST in a single pass and was not too difficult to learn! This was the largest hurdle for the assembler, but the end result is very compact!
7. **Print Statements and Logger—Input Output Kills Performance**
   Our final project phase included both bug-hunting and benchmarking. While implementing a fix, we accidentally left two debug print statements in the execute phase. Given we were running benchmarks which took up to a minute to finish, we immediately noticed that the print statements resulted in these benchmarks not completing. While the large amounts of text being output to the console was a giveaway to the cause, we witnessed firsthand how slow I/O and system calls to the kernel are when they are in code meant to run, in our case, billions of times in one minute.
8. **Templates are powerful:** As we have started adding vector instructions, we are faced with an issue with restructuring our pipeline to support integer registers and vector registers. Should we have separate pipelines? Should we have separate

classes? This is where we have decided to use templates to make our pipeline more generic and easily extensible in the future. The templates have helped us to reduce redundancy and have ensured that we do not have to change the code in too many places in case something fails.

9. **Separation of concerns is important for effective collaboration:** Throughout the development phase, we have been in constant communication in deciding how to break the project down into loosely coupled independent modules that can be integrated easily while at the same time be worked upon independently (we avoided both working on the same codebase—storage, GUI, pipeline, at once!) This has helped us maintain strong velocity in our project. Not sticking to just one part of the simulator has helped both of us understand how to build it as a whole and yet *we have rarely had to waste our time on resolving merge conflicts*.
10. **Subroutines without RET? And Subroutine Overhead**
    Our original proposal included a JAL instruction without a RET instruction. This is likely our biggest initial oversight. We have since been careful to ensure our JAL, RET, PUSH, and POP instructions made saving off registers and jumping easy, and think the result is intuitive and realistic! Because we utilized a stack in many of our benchmarks (including the heavy-weight primes-generator, which near-exclusively interacts with the storage through the stack), we directly observed why subroutines incur such a heavy overhead.
11. **PC-relative versus Absolute Addressing—Address Space Too Small**
    In our initial proposal, we were sure of our need for absolute versus PC-relative addressing. When it came time to distinguish the two, we found the difference would be mostly insignificant given that both instructions have the same size displacement field, and the displacement field is large enough to cover the entire address space. We find that in our final implementation, there is not enough to distinguish between the **JMP** and **JRL** instructions, but left both in, in the event that we could eventually distinguish their usefulness.
12. **Other small things—incorrect initial assumptions**
    There are many other less notable things we originally got wrong—figuring out the specifics of how to correctly bubble instructions up through the pipe was challenging, as it involved communication in both directions between the caller and callee pipe stages. Initially, the parent did not signal to the callee if it was ready. When we realized this was needed, we passed another function argument down. But the function signature of the `advance' became complicated—we had to return both the callee's status and an instruction DTO. Additionally, in the same method, we originally always bubbled down the status of our parent to our children—for example, a stall in MEM resulted in FETCH receiving a stall from DECODE even if DECODE was not currently processing an instruction. This

incorrect assumption became obvious when we increased the delay of the storage devices and finished the GUI.