ELISA: EducationaL Instruction Set Architecture

Josh Sennett & Yash Adhikari

The ELISA Architecture

Overall Design

ELISA (EducationaL Instruction Set Architecture) is a general purpose instruction set architecture. ELISA was designed in tandem with an instruction set simulator as an educational tool for instruction set design. The distinguishing feature of ELISA is a customizable cache, allowing users to see how different cache configurations affect performance and data flow. The choices for customization include the following:

- Enabling up to three levels of cache
- Selecting the associativity
    - Choice of Direct Mapped, 2-Way, 4-Way, 8-Way, 16-Way, 32-Way, N-Way
- Choosing the number of lines in each level of cache and in DRAM (multiple of 2)
- Selecting the number of words per line for each each level of cache
    - Choice of 1, 2, 4, 8, 16, or 32 words per line
- Choosing the delay incurred at each level of cache and with access to DRAM

ELISA is a variant of the MIPS architecture with a word size of 32 bits and single instruction per word. Instructions can contain a maximum of three addresses per instruction. The instruction set supports integer and floating point data types; Table 3 describes each of the instructions that will be supported.

ELISA uses two sets of 32-bit registers for integer and floating-point values, each containing 32 registers. Table 1 describes the name and purpose of each register.

Table 1. List of registers with their assigned purpose

Reg. Name	Description
$R0	The constant 0
$R1	Used for pseudo-instructions
$R2-$R4	Values from functions
$R5-$R9	Function arguments
$R10-$R29	Temporary data
$R29	Stack pointer
$R30	Frame pointer
$R31	Return address

$F0-$F31	Temporary data (floating-point)

Memory and Cache

ELISA uses the Princeton architecture for memory organization. Memory is byte-addressable; with 32-bit registers, we have 2^{32} addressable bytes (4GB) of memory.

We have developed a write-through no-allocate memory and cache system. The memory system can be configured to contain any number of levels of caching, and each level can be custom configured with respect to:

- Size (number of lines)
- Width (line length)
- Time to access (in cycles)
- Associativity (from direct-mapped to fully associative), with a random eviction policy

However, for simplicity we allow the user to only select up to three levels of cache in memory. In case a user requires testing performance with a more levels of cache, the underlying code structure is already built for expansion of cache.

We expect that our simulator can be a valuable tool to help future computer architects understand how caching works, and how cache configurations affect performance. To do this, we provide tools to measure performance in terms of the total number of cycles to execute benchmarks specified by the user.

Addressing Modes

ELISA supports the following addressing modes:

- Register-direct
- Register-indirect
- Register-indirect with base, index and/or offset
- Immediate
- PC Relative

Load-Store Instructions:

ELISA is a load-store architecture; the instruction set includes load and store instructions for loading and storing bytes and words (see Table 3).

Control flow:

The instruction set includes conditional branch and unconditional jump instructions for control flow (see Table 3). The JR (jump register) instruction makes it possible to jump anywhere in the 32-bit address space with a single instruction.

Subroutine jump handling:

Subroutine jumps are handled using the JAL (jump and link) and JR (jump to register) instructions. First, arguments are passed onto the argument registers $R5-$R9. The JAL instruction will store the value of the program counter in the return address register ($31), and will change the program counter to jump to a subroutine.

Since our architecture uses only a single return address register, nested subroutines require storing a subroutine's return address and arguments on the stack (using a SW instruction with destinations starting at $R30) before it is replaced. The frame and stack pointers ($R29, $R30) are set to the first and last words of the frame respectively. As subroutines return, data is popped off of the stack. Values are returned onto registers $R2-$R4.

Integer ALU instructions:

The instruction set supports the following arithmetic and logic operations: add, add immediate, subtract, multiply, divide, shifts, and, and immediate, or, or immediate, nor, xor, and xor immediate. For further information please refer to Table 3.

Other Instructions:

There were several instructions including shifts such as shift right logical and shift left logical that were also implemented. Furthermore, we had to include instructions such as move from lo and move from hi to capture results from integer multiplication and division operations. We included other instructions such as J (jump) and JALR (jump and link register) to provide users with more options specific to their needs. These instructions can be referred to in Table 3. All arithmetic, load/store, and comparison operations for floating point numbers will have their own instruction set.

Instruction type field:

Following the MIPS architecture, our instruction set has three instruction types that provide different fields for register, immediate, and jump instructions. The formats for these three types is shown in Table 2. Rather than having a separate instruction type field, the instruction type is uniquely identified by the 6-bit opcode. For I- and J-type instructions, the opcode alone uniquely identifies the instruction; for R-type instructions, the opcode and a 6-bit function codes uniquely identify the instruction. Remaining, undefined opcodes will be handled by a NOOP instruction that will not perform a meaningful action and move on to the next instruction. Note that the $ symbol indicates a register.
R-type instructions include integer and float arithmetic operations that use a destination register $d and two source registers $s, $t. I-type instructions include $s and $t registers; $s is a source register, and $t is either a source register (for branch instructions) or a destination register (for other instructions). The remaining 16 bits are used to specify an immediate, such as an offset. Immediates are 16 bits in two's complement and are therefore always sign-extended. J-type instructions are used for unconditional jumps, and include an opcode and a 26 bit address.

The floating point instructions in general will follow a very similar format to R-type instructions as the one displayed below in Table 2. Most of the floating point instructions with the exception of 4 instructions follow a variant of the R-type instruction illustrated as R*. The most significant difference between the two types of instruction is that bits to represent registers $s, $t, and $d are shifted to the right by 5 bits. The space left behind is often filled with information identifying the type of instruction that is presented. The four instructions that are not covered by R-type variant are l.s, s.s, bc1t and bc1f which are all I-type instructions.

Table 2. Instruction Formats

Type	**Fields (32 bits in total)**					
R	Opcode (6)	$s (5)	$t (5)	$d (5)	Shift (5)	Function Code (6)
R*	Opcode (6)	- (5)	$s (5)	$t (5)	$d (5)	Function Code (6)
I	Opcode (6)	$s (5)	$t (5)	Immediate (16)		
J	Opcode (6)	Address (26)				

Example Instructions:

Here are three examples of R, I, and J instructions:

R-type:

`ADD $R3, $R1, $R2`

Instruction: `0000 0000 0010 0010 0001 1000 0010 0000`

Opcode (6)	$s (5)	$t (5)	$d (5)	Shift (5)	Function Code (6)
0000 00	00001	00010	00011	00000	100000

I-type:

`LW $R3, 6($R1)`

Instruction: `1000 1100 0010 0011 0000 0000 0000 0110`

Opcode (6)	$s (5)	$t (5)	Immediate (16)
1000 11	00001	00011	0000 0000 0000 0110

J-type:

`JAL 0xabcd`

1000 1100 0000 0000 1010 1011 1100 1101

Opcode (6)	Address (26)

`1000 11`	`00 0000 0000 1010 1011 1100 1101`

Table 3. Supported Instructions

Type	Syntax	Operation	Description	Operation*
		Arithmetic		
R	`ADD $d, $s, $t`	Add	Adds values in 2 different registers and stores the result in another register	$d = $s + $t
I	`ADDI $t, $s, i`	Add immediate	Adds value in a register to a sign-extended immediate value and stores the result in another register.	$t = $s + immediate
R	`SUB $d, $s, $t`	Subtract	Subtracts values in 2 different registers and stores the result in another register	$d = $s - $t
I	`SLL $d, $t, h`	Shift left logical	Shifts a register value to the left by a particular shift amount and places the value in another register	$d = $t << h
I	`SRL $d, $t, h`	Shift right logical	Shifts a register value to the right by a particular shift amount and places the value in another register. Zeros are shifted in.	$d = $t >> h
I	`SRA $d, $t, h`	Shift right arithmetic	Shifts a register value right by a particular shift amount and places the value in the destination register. The sign bit is shifted in.	$d = $t >> h
R	`MULT $d, $s, $t`	Multiply	Multiplies values in two different registers and stores the result in another register	$d = $s * $t
R	`DIV $d, $s, $t`	Divide	Divides values in two different registers and stores the result in another register	$d = $s / $t
R	`MFHI $d`	Move from HI	Moves the value stored in register $HI to register $d	$d = $HI

R	`MFLO $d`	Move from LO	Moves the value stored in register $LO to register $d	$d = $LO
Logic				
R	`AND $d, $s, $t`	And	Bitwise ands two registers and stores the result in another register	$d = $s & $t
I	`ANDI $t, $s, i`	And immediate	Bitwise ands a register and an immediate value and stores the result in another register	$t = $s & immediate
R	`OR $d, $s, $t`	Or	Bitwise ors two registers and stores the result in another register	$d = $s | $t
I	`ORI $t, $s, i`	Or immediate	Bitwise ors a register and an immediate value and stores the result in another register	$d = $s | immediate
R	`XOR $d, $s, $t`	Exclusive or	Bitwise exclusive ors two registers and stores the result in another register	$d = $s ^ $t
I	`XORI $t, $s, i`	Exclusive or immediate	Bitwise exclusive ors a register and an immediate value and stores the result in another register	$d = $s ^ immediate
R	`NOR $d, $s, $t`	Nor	Bitwise nors two registers and stores the result in another register	$d = !($s) & !($t)
Control Flow**				
I	`BEQ $s, $t, offset`	Branch on equal	Branch is taken if the two registers are equal in value	If ($s == $t), nPC += (offset << 2)
I	`BNE $s, $t, offset`	Branch on not equal	Branch is taken if the two registers are not equal in value	If ($s != $t), nPC += (offset << 2)
I	`BGEZ $s, offset`	Branch on greater than or equal to zero	Branch is taken if the register is greater than or equal to 0	If ($s >= 0), nPC += (offset << 2)

I	`BLEZ $s, offset`	Branch on less than or equal to zero	Branch is taken if the register is less than or equal to 0	If ($s <= 0), nPC += (offset << 2)
I	`BGTZ $s, offset`	Branch on greater than zero	Branch is taken if the register is greater than 0	If ($s > 0), nPC += (offset << 2)
I	`BLTZ $s, offset`	Branch on less than zero	Branch is taken if the register is less than 0	If ($s < 0), nPC += (offset << 2)
J	`J target`	Jump	Jump to the calculated address	PC = nPC; nPC = (PC & 0xf0000000) | (target << 2)
J	`JAL target`	Jump and link	Jumps to the calculated address and stores the return address in $31	$31 = PC + 8; PC = nPC; nPC = (PC & 0xf0000000) | (target << 2)
R	`JR $s`	Jump Register	Jumps to the address stored in $s	PC = nPC; nPC = $s;
J	`JALR $s`	Jump and link register	Jumps to the address stored in $s and stores the return address in $31	$31 = PC + 8; PC = nPC; nPC = nPC; nPC = $s;
R	`SLT $d, $s, $t`	Set if less than	If $s is less than $t, another register $d is set to 1	If $s < $t, $d = 1
I	`SLTI $t, $s, i`	Set if less than immediate	If $s is less than immediate, another register $d is set to 1	If $s < immediate, $t = 1
Memory Access				
I	`LB $t, offset($s)`	Load byte	A byte is stored into a register from a given address	$t = MEM[$s + offset]
I	`LW $t, offset($s)`	Load word	Load word into a register from a given address	$t = MEM[$s + offset]
I	`SB $t, offset($s)`	Store byte	The least significant byte of a register is stored into a given address	MEM[$s + offset] = (0xff & $t)
I	`SW $t,`	Store	Store a word into the given	MEM[$s + offset] =

	`offset($s)`	word	address from a register	$t
Arithmetic (Floating Point)				
R*	`ADD.S $fd, $fs, $ft`	Add	Adds floating point values in 2 different registers and stores the result in another register	$fd = $fs + $ft
R*	`SUB.S $fd, $fs, $ft`	Subtract	Subtracts floating point values in 2 different registers and stores the result in another register	$fd = $fs - $ft
R*	`MUL.S $fd, $fs, $ft`	Multiply	Multiplies floating point values in two different registers and stores the result in another register	$fd = $fs * $ft
R*	`DIV.S $fd, $fs, $ft`	Divide	Divides floating point values in two different registers and stores the result in another register	$fd = $fs / $ft
R*	`CVT.S.W $fd, $fs`	Convert Integer to Floating Point	Convert the integer in the given register to a floating point number	$fd = (FP) $s
R*	`CVT.W.S $fd, $fs`	Convert Floating Point to Integer	Convert the floating point number in the given register to an integer	$fd = (INT) $fs
Logic (Floating Point)				
R*	`C.EQ.S $fs, $ft`	Compare Equal	Compare the floating point values in the two registers and set the floating point condition flag true if they are equal.	$fs == $ft: True
R*	`C.LE.S $fs, $ft`	Compare Less Than or Equal	Compare the floating point values in the two registers and set the floating point condition flag true if the value in $fs is less than or	$fs <= $ft: True

			equal to the value in $ft.	
R*	`C.LT.S $fs, $ft`	Compare Less Than	Compare the floating point values in the two registers and set the floating point condition flag true if value in $fs is less than the value in $ft.	$fs < $ft: True
Control Flow (Floating Point)				
I	`BC1T offset`	Branch coprocessor true	Branch is taken if the coprocessor condition flag is set to true	If (cc == true), nPC += (offset << 2)
I	`BC1F offset`	Branch coprocessor false	Branch is taken if the coprocessor condition flag is set to false	If (cc == false), nPC += (offset << 2)
Memory Access (Floating Point)				
I	`L.S $ft, offset($s)`	Load Floating Point	Load floating point value into a register from the given address	$ft = MEM[$s + offset]
I	`S.S $ft, offset($s)`	Store Floating Point	Store floating point value in the given address from a register	MEM[$s + offset] = ($ft)

* The program counter (PC) is set to nPC and nPC is incremented by 4 for each instruction that is executed unless a branch or jump occurs.
** Offset refers to a value added to an address, while target refers to a address replacing another

The ELISA Simulator

General Overview

The ELISA simulator allows users to see a program executed step-by-step, providing insight into how assembly is converted to machine code, how instructions pass through the data pipeline, and how registers, cache, and memory change over time. In addition, the simulator allows users to configure cache, memory, and pipeline settings, allowing users to visualize how these configurations affect performance.

The full ELISA codebase has five primary modules which interact to provide the full functionality of the simulator.

Table 4: Components of the ELISA Codebase

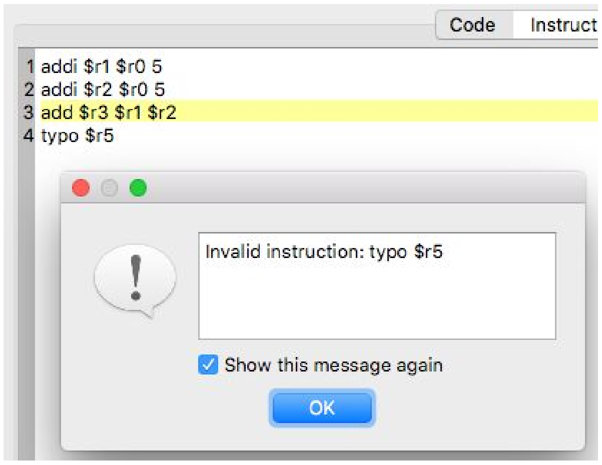
Component	Filename	Functionality
Assembler	`assembler.py`	Assembles programs, including utilities such as calculating the two's complement of a number and converting between floating point and binary (using IEEE 754 standard)
Memory	`memory.py`	Simulates cache and memory, interfaced using `read()` and `write()` functions
Simulator	`simulator.py`	Simulates pipeline stages, interfaced using `step()` method to step a single cycle, and a `set_instructions()` method which loads machine code instructions and data into memory
GUI View	`gui/mainwindow.py`	Provides a graphical display for the simulator
GUI Controller	`gui/gui.py`	Adds functionality to the GUI view, integrating the assembler, memory, and the simulator components into a single application

Assembler

The ELISA assembler converts a file into a list of numerical instructions; the full instruction set and syntax is specified in table 3. The assembler uses two passes; the first to parse labels, and the second to convert lines to machine code. The assembler supports the following features:

- Comments and empty lines are stripped
- Data specified in a `.data` section is loaded into memory
- Data types are inferred automatically (negative int -> two's complement; float -> IEEE 754)
- Labels are converted to calculated targets or offsets based on instruction type
- Integer registers can be specified using register number or MIPS nickname (e.g. `$zero` or `$r0`, `$t0 = $r8`, `$ra = $r31`)
- Assembling a file with syntax errors will raise an error pointing to the error-raising line (Figure 1)

We developed a comprehensive test suite (`test/test_assembler.py`) to ensure that these features worked correctly, and that each instruction is correctly assembled.



```
addi $r1 $r0 5
addi $r2 $r0 5
add $r3 $r1 $r2
typo $r5
```

Invalid instruction: typo $r5

Figure 1: A dialog message indicating invalid instruction syntax

Memory

The memory module is designed to simulate memory and cache independently from the ELISA simulator, which makes it easy to integrate into other ISA simulators or usable on its own. The Memory class supports the following configuration options:

- lines (int): lines of data (each line being 4 bytes)
- delay (int): number of cycles it takes for a memory access (read or write)
- noisy (boolean, default=False): If noisy, the Memory class will pause after each cycle
- name (str): a handle for the object

Cache

The Cache class is designed as a write-through no-allocate cache with random eviction. It supports the same configuration options as the Memory class, in addition to:

- words_per_line (int): words per line
- associativity (int): associativity level (e.g. 1 for direct-mapped, n for n-way set associative)
- next_level (pointer): pointer to the next level of memory (where reads will be requested and writes will be forwarded)

To invoke a `read()` or `write()` to memory or cache, the caller indicates the number of words requested and whether you are reading or writing a word or a byte. For example, for Memory instance DRAM, DRAM.read(0x5, words_requested=4) will return a block of four words (from 0x0 to 0x1c); DRAM.read(0x5, words_requested=1) will return a single word (from 0x4); and DRAM.read(0x5, only_byte=True) will return the byte (from 0x5). A cache will request the number of words from the next level of memory depending on its line width; the top level of cache (or memory, if caching is disabled) will return a single word or byte. A store-byte will cause each level of memory to specify `only_byte=True`, so that only a single byte in memory will be modified.

Simulator

The Simulator class simulates the full ISA, and is interfaced with a simple `step()` function. A Simulator's attributes include a memory heirarchy (an array of Memory/Cache objects), as well as integer and floating point registers, a pipeline buffer array, a destination table for data dependencies, a pipeline-enabled attribute, and a few attributes to track the progress of multi-cycle ALU operations.

Pipeline Stages

The ELISA architecture follows the MIPS five-stage pipeline design. The pipeline is implemented by "pulling" data from stage to stage in reverse order; a `step()` invocation calls the write-back stage, which calls the memory stage, and so on. The pipeline buffer array contains four elements, each containing a tuple of integers output from the previous stage and input into the next stage. For example, the memory stage acts based on the contents of `buffer[2]` (from the execute stage) and outputs its results to `buffer[3]` (for the write-back stage. The exceptions, of course, are that the instruction-fetch stage receives input by requesting data from the top level of the memory heirarchy, while the write-back stage writes to registers rather than a buffer.

If a pipeline stage needs to stall the pipeline (for example, if the instruction-decode stage detects a data dependency or if the memory-access stage is waiting for results), the stage writes a no-op into its output buffer and does not invoke the previous stage, simulating a structural hazard. To simulate disabling the pipeline, a simulator will wait until the instruction-fetch stage has been invoked five times before it is actually allowed to fetch from memory.

Testing

Due to the complexity of the pipeline stages, we found it incredibly useful to develop a comprehensive test suite for the simulator. The `test/test_instructions.py` and `test/test_simulator.py` test each instruction at least once, which helped us quickly identify bugs in our simulator's logic.

Programmable Interface

The Simulator class can be used programmatically, allowing users the most flexibility when designing a custom simulation. For example, we used our Simulator class to create a subclass for unit testing instructions. The following code defines a SingleInstructionSimulator class as a Simulator with no cache, a single instruction, and registers preset to values 0-31.

```python
class SingleInstructionSimulator(Simulator):
    """A basic simulator used for testing single instructions"""
    def __init__(self, instruction):
        super().__init__()
        DRAM = Memory(lines=2**8, delay=0)
        self.memory_heirarchy = [DRAM]
        instructions, data = assemble_to_numerical(instruction)
        self.set_instructions(instructions)
        self.R = list(range(32))
```

Figure 2: Using the Simulator class programmatically

Graphical User Interface Overview

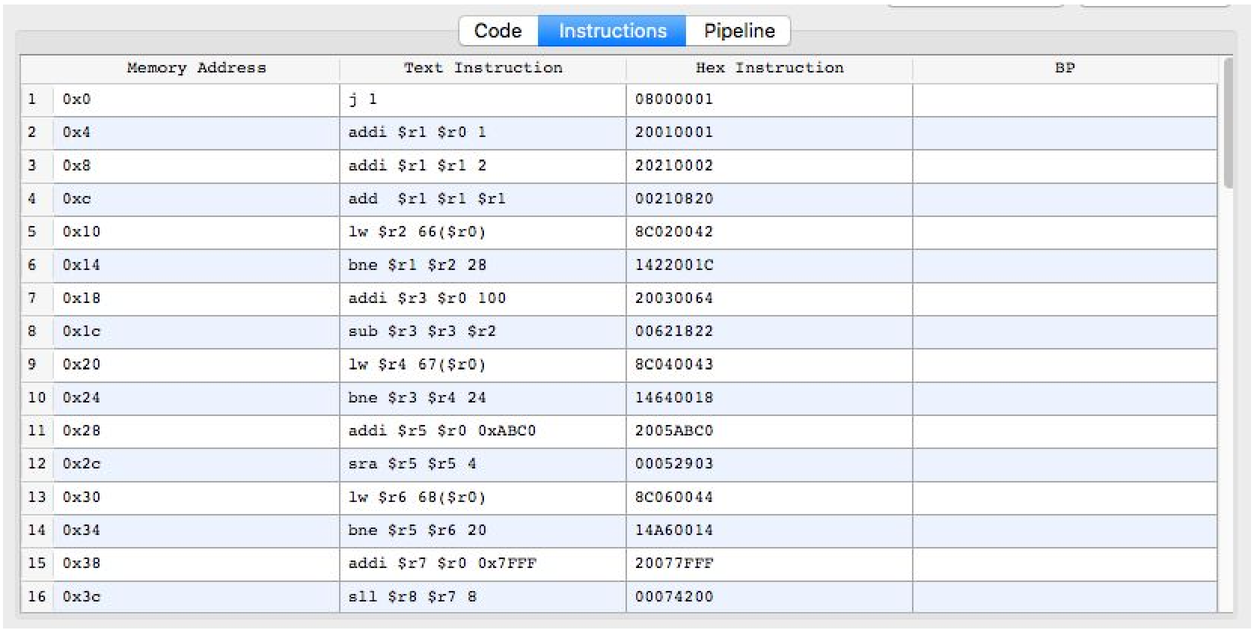
We also provide a graphical interface to the ELISA simulator. The simulator's graphical user interface (GUI) contains three sections (see Figure 5).

Data section:
The data section displays register, memory, and cache contents in three tabs. In the register table, each cell refers to a single register's contents denoted by the row and column headers. In the memory table, each row is a word in memory (containing a single word). In the cache table, each row is a cache line; columns denote which level of cache the line refers to, the line index, and the line contents (separated by tag, valid bit, and words). The cache and memory tables will update based on the current configuration; for example, additional columns are added to the cache table if the user increases the number of words per line, and the number of rows for memory and cache tables adjusts to match the size of memory and cache set by the user. Data can be viewed in binary, hexadecimal, or decimal formats; in addition, a “human-readable” format will view integer registers as negative or positive integers (rather than using two’s complement format), and floating point registers as floats (rather than in in IEEE 754 format).

Instruction section:
The instruction section contains three tabs for a code editor, an instruction table, and a pipeline table.

In the code editor tab, a user can import a program into the editor, modify the code, and export the modified code. When the program is ready to be loaded into the assembler, the user must press the "Load Instructions" button, which will assemble the current contents of the code editor into machine code. If an instruction is incorrectly specified and raises an error, a dialog box will pop-up to display where the error occured. If all instructions are correctly specified, the assembled instructions will be loaded into the instructions table.

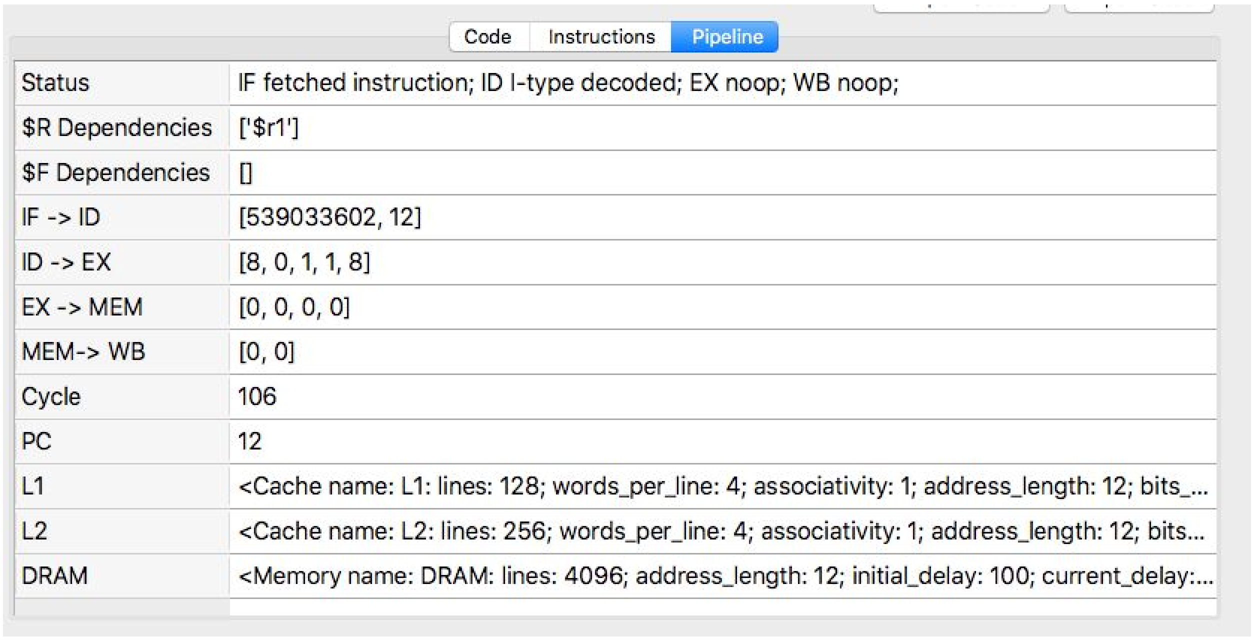
The instructions table (Figure 3) contains the memory address, text instruction, hex instruction, and a breakpoint indicator for each instruction. Breakpoints can be specified using the "Add Breakpoint" and "Remove Breakpoint" buttons, where breakpoints refer to the row number in the instruction table. Users can run instructions by pressing one of the "Step" buttons to either step one cycle, step N cycles, step until next breakpoint, or step until program completion. An breakpoint is reached when the simulator's program counter matches the instruction's memory address (i.e. when the instruction is fetched). If no breakpoint is specified or if the program never reaches the breakpoint, the Step until Breakpoint button will step until program completion.



	Memory Address	Text Instruction	Hex Instruction	BP
1	0x0	j 1	08000001	
2	0x4	addi $r1 $r0 1	20010001	
3	0x8	addi $r1 $r1 2	20210002	
4	0xc	add $r1 $r1 $r1	00210820	
5	0x10	lw $r2 66($r0)	8C020042	
6	0x14	bne $r1 $r2 28	1422001C	
7	0x18	addi $r3 $r0 100	20030064	
8	0x1c	sub $r3 $r3 $r2	00621822	
9	0x20	lw $r4 67($r0)	8C040043	
10	0x24	bne $r3 $r4 24	14640018	
11	0x28	addi $r5 $r0 0xABC0	2005ABC0	
12	0x2c	sra $r5 $r5 4	00052903	
13	0x30	lw $r6 68($r0)	8C060044	
14	0x34	bne $r5 $r6 20	14A60014	
15	0x38	addi $r7 $r0 0x7FFF	20077FFF	
16	0x3c	sll $r8 $r7 8	00074200	

Figure 3: The instructions table

The pipeline table (Figure 4) contains the current contents of the simulator buffer, along with useful information about the simulator such as the current memory and cache configuration, register dependences, and a status message containing information from each pipeline stage.



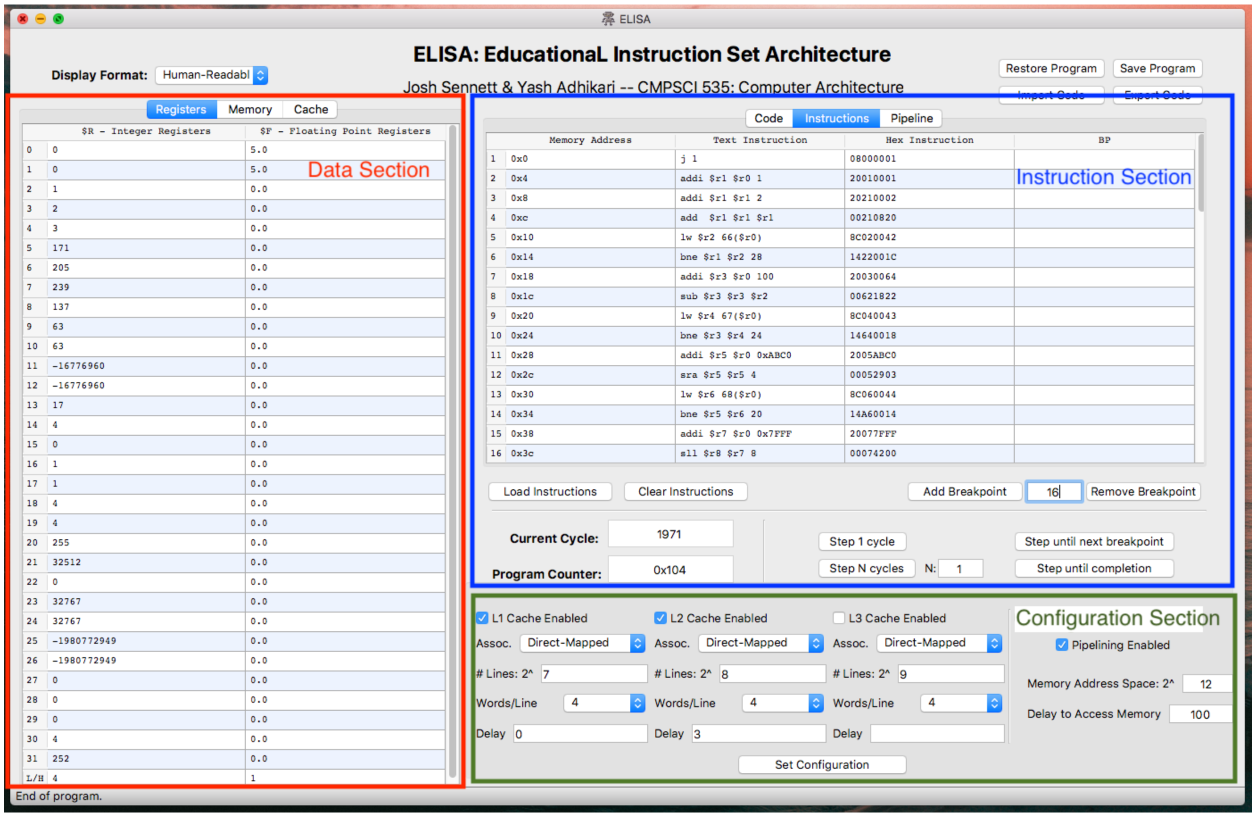
Status	IF fetched instruction; ID I-type decoded; EX noop; WB noop;
$R Dependencies	['$r1']
$F Dependencies	[]
IF -> ID	[539033602, 12]
ID -> EX	[8, 0, 1, 1, 8]
EX -> MEM	[0, 0, 0, 0]
MEM-> WB	[0, 0]
Cycle	106
PC	12
L1	<Cache name: L1: lines: 128; words_per_line: 4; associativity: 1; address_length: 12; bits_...
L2	<Cache name: L2: lines: 256; words_per_line: 4; associativity: 1; address_length: 12; bits...
DRAM	<Memory name: DRAM: lines: 4096; address_length: 12; initial_delay: 100; current_delay:...

Figure 4: Pipeline table

Configuration section:
The third section provides the user with an interface to configure cache, memory, and pipeline settings. While the programmable simulator can technically support any number of levels of cache, the GUI allows users to enable up to three cache levels to keep the display simple and decluttered. For each level of cache, the user can customize the cache's associativity, size, words per line, and delay; as well as memory size and delay; and whether pipelining is enabled or disabled. The "Set Configuration" button will apply these settings to the simulator, update the memory and cache tables in the data section, and will reset the simulator's program counter and register, cache, and memory contents to initial values.

The GUI also provides the following functionality:

- Save and restore an in-progress program; saving fully serializes the simulator including all configuration settings as well as the register, cache, memory, and pipeline data needed to restore and continue running a program. Restoring a program will deserialize the simulator and update the GUI so the program can continue running.
- The GUI validates input, preventing you from even typing invalid characters in several fields. For example, cache size and delay do not permit typing non-digit characters.
- Invalid combinations of input will raise an error dialog box (for example, if the level of associativity is greater than the number of lines in cache)



Display Format: Human-Readabl

	$R - Integer Registers	$F - Floating Point Registers
0	0	5.0
1	0	5.0
2	1	0.0
3	2	0.0
4	3	0.0
5	171	0.0
6	205	0.0
7	239	0.0
8	137	0.0
9	63	0.0
10	63	0.0
11	-16776960	0.0
12	-16776960	0.0
13	17	0.0
14	4	0.0
15	0	0.0
16	1	0.0
17	1	0.0
18	4	0.0
19	4	0.0
20	255	0.0
21	32512	0.0
22	0	0.0
23	32767	0.0
24	32767	0.0
25	-1980772949	0.0
26	-1980772949	0.0
27	0	0.0
28	0	0.0
29	0	0.0
30	4	0.0
31	252	0.0
L/H	4	1

	Memory Address	Text Instruction	Hex Instruction	BP
1	0x0	j 1	08000001	
2	0x4	addi $r1 $r0 1	20010001	
3	0x8	addi $r1 $r1 2	20210002	
4	0xc	add $r1 $r1 $r1	00210820	
5	0x10	lw $r2 66($r0)	8C020042	
6	0x14	bne $r1 $r2 28	1422001C	
7	0x18	addi $r3 $r0 100	20030064	
8	0x1c	sub $r3 $r3 $r2	00621822	
9	0x20	lw $r4 67($r0)	8C040043	
10	0x24	bne $r3 $r4 24	14640018	
11	0x28	addi $r5 $r0 0xABC0	2005ABC0	
12	0x2c	sra $r5 $r5 4	00052903	
13	0x30	lw $r6 68($r0)	8C060044	
14	0x34	bne $r5 $r6 20	14A60014	
15	0x38	addi $r7 $r0 0x7FFF	20077FFF	
16	0x3c	sll $r8 $r7 8	00074200	

Breakpoint: 16

Current Cycle: 1971
Program Counter: 0x104
N: 1

	L1	L2	L3
Cache Enabled	✓	✓	
Assoc.	Direct-Mapped	Direct-Mapped	Direct-Mapped
# Lines: 2^	7	8	9
Words/Line	4	4	4
Delay	0	3	

✓ Pipelining Enabled
Memory Address Space: 2^ 12
Delay to Access Memory 100

Figure 5: Data, Instruction, and Configuration Sections of the GUI

Software Engineering Methods Used

We developed ELISA using Git for version control (publicly available at https://github.com/jsennett/ELISA). Our typical workflow was to create feature branches that, once tested, would be merged with a master branch. We approached coding using test-driven development; specifically, we used Pytest (https://doc.pytest.org/) to create unit test suites. We ended with 124 unit tests in total for our assembler, simulator, and memory modules. In addition, as demoed in class, we developed assembly scripts that tested correctness of each type of instruction as sort of bootstrapped unit tests.

We designed ELISA to be modular; the assembler and memory classes are independent subcomponents that integrate with the simulator, but work independently too and could be swapped with similar components. Our design fits the Model-View-Controller paradigm, making it easy to use the same simulator with different views (programmatically or through the GUI). Last, we designed ELISA to have a very simple programmable interface; you can assemble and run a program in just a few lines (see Figure 2 [the SingleInstructionSimulator subclass]).

Efforts by Teammate

Pair Programming:
- Overall design of the Simulator class
- Memory and Cache classes (Demo)
- Simulator pipeline design
- Support for cache associativity
- Assembly script to demo partial ISA (mid-march)
- Assembly script to demo full ISA

Josh independently:
- Assembler + assembler test suites
- Overall design of the Memory and Cache classes
- Graphical user interface
- Interactive script to demo memory
- Utils (float-binary converter, twos-complement converter)
- Exchange sort benchmark
- Implementation of half of the instructions
- Final report sections: Simulator Design, Software Engineering Methods, Conclusion

Yash independently:
- Initial design of pipeline (reworked together due to no “pulling” of data)
- Pipeline features (data dependencies, disabling of pipeline, multiple stage ALU operations)
- Implementation of half of the instructions
- Matrix multiplication benchmark

Final report sections: ELISA Architecture, Benchmark Performance Result
Final report sub-sections: Summarized learned lessons from benchmarks in conclusion

Benchmark Performance Results

There were two benchmarks - matrix multiplication and exchange sort that were tested with ELISA to assess performance. Since the goal of ELISA is to teach effects of altering computer architectures, we used clock cycle counts as a means of understanding how performance changes with different designs.

Matrix Multiplication:

We wanted to test a given computer architecture around the limits of its capacity. For the matrix multiplication benchmark, we started with the first matrix of size 50 by 50 and the second matrix of size 50 by 100 resulting in another matrix of size 50 by 100. The two matrices containing the data were initialized with random numbers between 1 and 9 inclusive. The total memory these matrices along with the instruction set requires is 12,560 words. We can initialize DRAM to a size significantly larger than the requirement. However, if we want to test around the limits of capacity, we have to alter the size of cache.

We know that storing words back into memory will not necessarily change with respect to cache size as our architecture is designed as a write-through, no allocate cache. Thus, the effective memory we are interested in changing will be 7,560 words which includes the two matrices with data and the instruction set. If we want to fit the entire data set into cache for easy access, we can set the cache to contain 2^{13} words.

In order to compare performance, we have to first set a baseline. Therefore, we started with no cache, pipeline disabled, memory of size 2^{16} words, and memory delay of 10 cycles. With all of the possible configurations, this architecture should yield the largest number of cycle counts. The next step was to enable pipelining of instruction set which will significantly improve performance. Usually, enabling of pipeline would increase the performance by a factor of the number of pipeline stages. However, in this benchmark, a number of delays are incurred due to memory access. Furthermore, we know that our configuration does not have cache yet which would help with memory access delays. Thus, such a large performance boost is not expected. Our final set of configurations involved enabling direct-mapped caches and testing a range of cache sizes by altering both the number of lines and the number of words. We expect that, when the cache contains more than 2^{13} words, there will not be a significant performance increase.

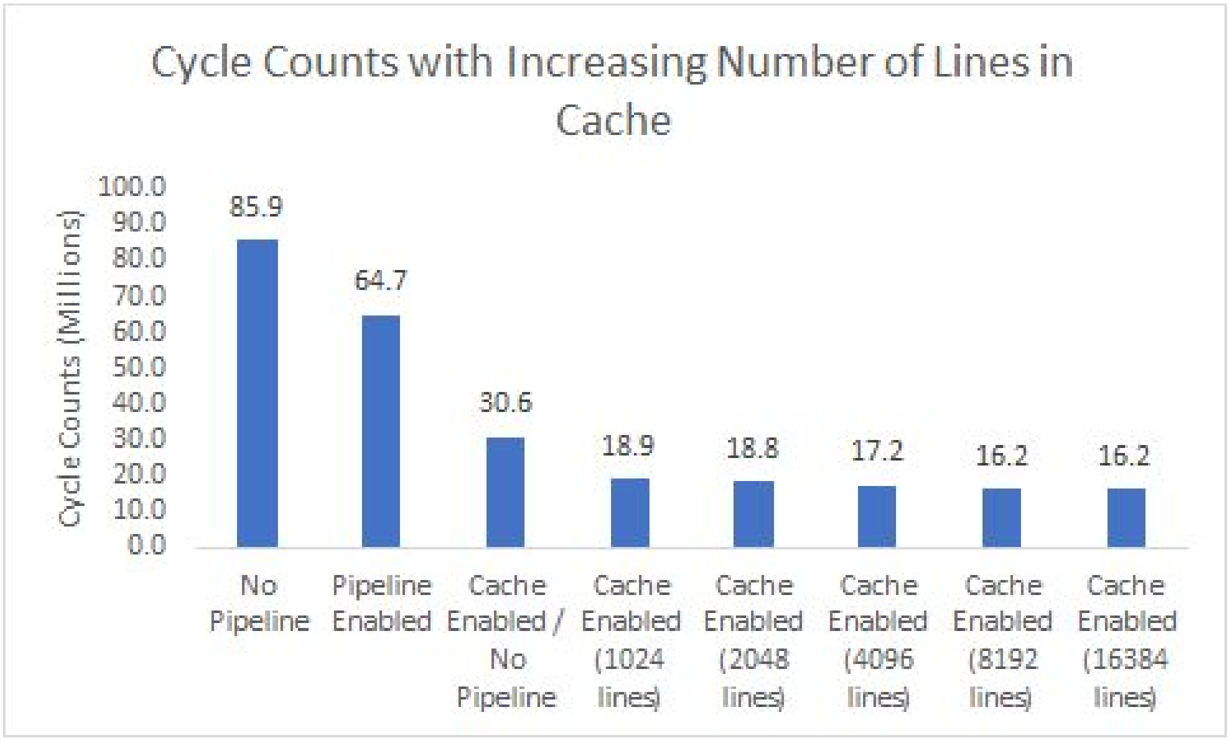


Figure 6: Performance measured in cycle counts with and without the pipeline feature enabled as baseline and increasing cache lines with 1 word each.

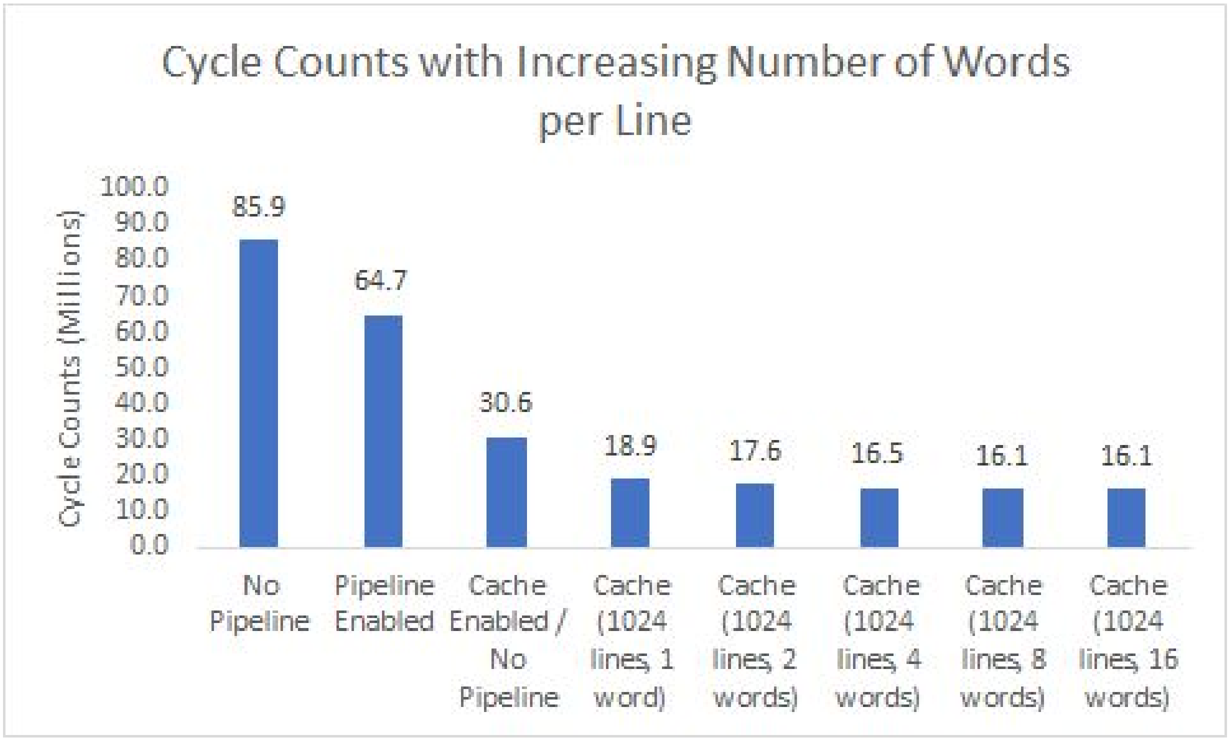


Figure 7: Performance measured in cycle counts with and without the pipeline feature enabled as baseline and increasing number of words with 2^{10} (1024) lines each.

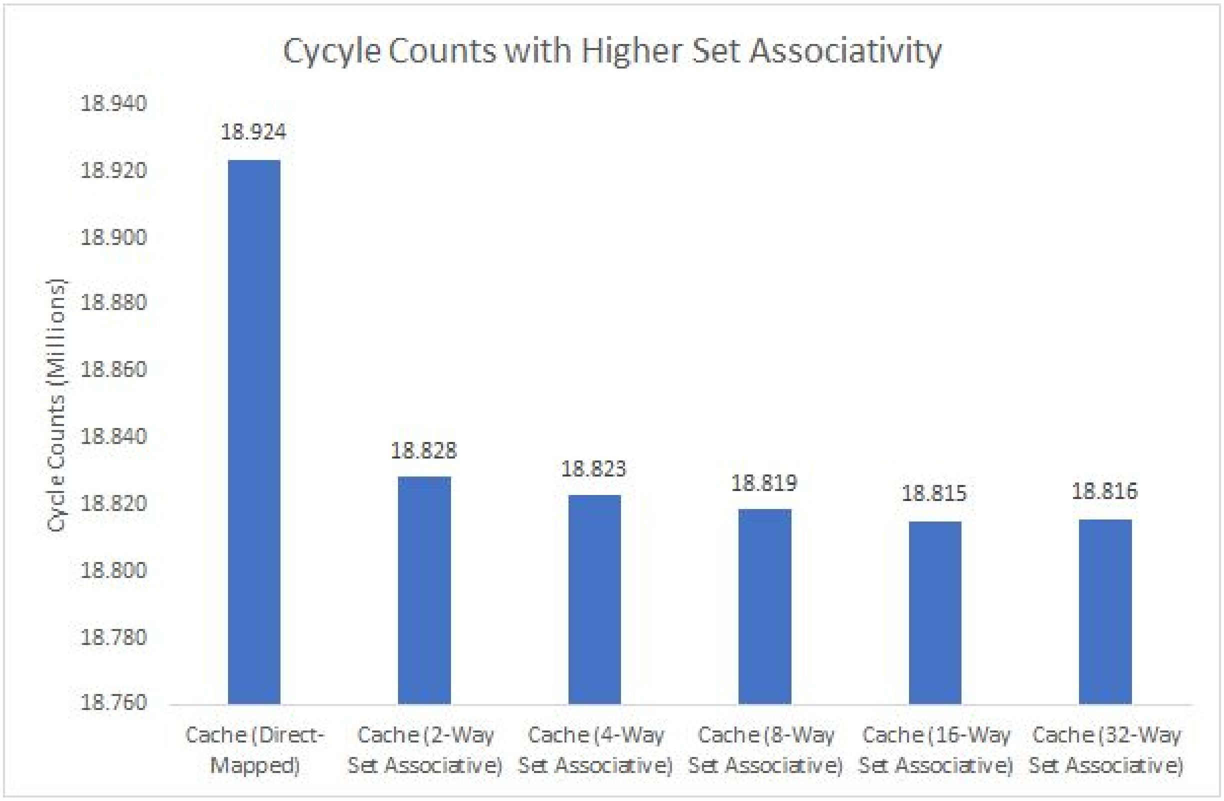


Figure 8: Performance measured in cycle counts with increasing number of set associativity. Each configuration only contains 1 word per line and 2^{10} number of lines.

The data showcased above gave expected results. We see that without cache or pipeline enabled, the given instruction set for matrix multiplication takes 85.9 million cycles to execute. This number represents the worst possible configuration, and as expected, we see that all other configurations result in a better performance and a lower number of cycle count. As we enable pipelining of the instruction set, we see that the total number of cycle counts drops down to 64.7 million. We know that we are unlikely to see a 5 fold increase in performance as loading words into registers takes a 10 cycle delay which accounts for a significant portion of the total number of clock cycles. This theory is confirmed by the next two configurations where cache is enabled but pipeline feature is turned off and when pipeline feature is left turned on. We observe that cache with no pipelining only takes 30.6 million cycle counts. This number is further reduced down to 18.9 million cycle counts for cache with pipelining. At this point we also see almost a 5 fold increase in performance as accessing L1 cache may occur within one clock cycle.

In figure 6, we see a continued increase in performance as the number of lines in cache increases until it reaches 2^{13} lines where it plateaus. We expected to see this since we are working with 1 word per line and all relevant data from matrix 1 and 2 can be loaded directly into L1 cache. Thus, when the size of L1 cache is increased further to 2^{14} lines, there is no change in the number of clock cycles it takes to complete the instruction set. This same theory also applies to data presented in figure 7. We see that the number of cycles decreases with increasing number of words per line. Since we are increasing the number of words per line and keeping the number of lines constant, we are effectively increasing the size of the cache. Once we can hold more than 2^{13} words, we should not see a significant improvement in performance. As expected, after 8 words per line, there is no performance boost. We know that 8 is the correct number since we are working with 2^{10} number of lines and 2^3 words per line which gives a total size of 2^{13} words. It is interesting to note that for the same size of memory, increasing the number of words in cache results in better performance than increasing the number of lines in cache. Even though the difference is small, we consistently see this phenomenon with 2, 4, 8, and 16 words per line. This likely occurs due to the fact that loading a high number of words significantly decreases the number of cache misses.

Finally, we also tested the effect of increasing the set associativity of a given computer architecture. Since we started with a cache size of 2^{10} lines and 1 word per line, we expected to see an increase in performance with an increase in set associativity. However, as we increase the associativity and hold the same number of lines in cache, we effectively decrease the size of L1 cache. Our data expectedly shows a bounce back in the number of cycles with 32-way set associative cache taking a higher number of cycles to complete the instruction set than the 16-way set associative cache.

Exchange Sort:

We approached analysis of exchange sort in a different manner than the analysis of matrix multiplication. We varied the size of the input vector while holding the configurations of the architecture constant. There were 4 different settings tested for this mode which were as follows:

- DRAM only
- DRAM + L1 cache
- DRAM + L1 cache + L2 cache
- DRAM + L1 cache + L2 cache + L3 cache

The size of DRAM was 2^{20} lines, size of L1 cache was 2^{10} lines with 4 words per line, size of L2 cache was 2^{12} lines with 4 words per line, and size of L3 cache was 2^{14} lines with 4 words per line.These four settings were tested with pipelining feature turned off and turned on. Figure 9 displays the results achieved for these defined settings.

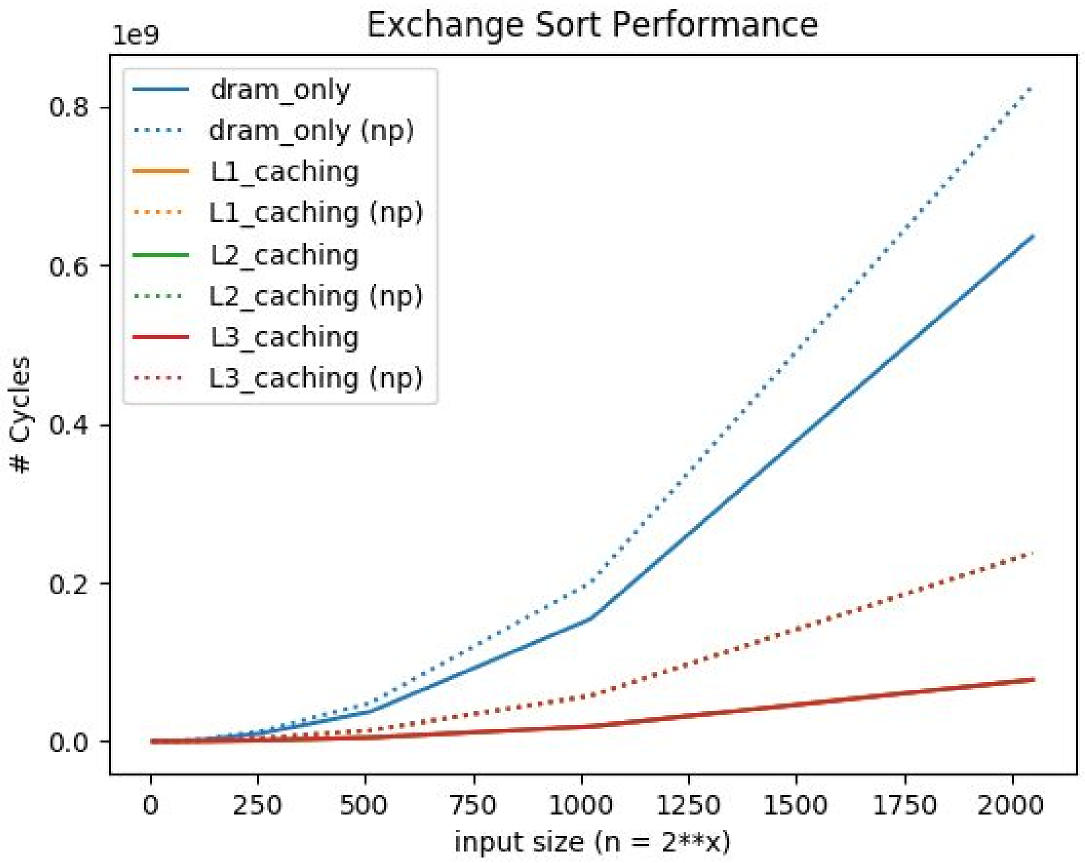


Figure 9: The plot above displays the performance while enabling 3 different levels of cache and toggling between pipelined and non-pipelined versions of architecture.

We see that the non-pipelined version of the architecture performs poorly compared to the pipelined versions. Furthermore, enabling of cache leads to a significant boost in performance as delays incurred from memory access are significantly lowered. We had also seen these two features before with matrix multiplication, and these results are well within our expectations. The more interesting result we achieved was that when we enabled only one level of cache, it gave similar performance to when we enabled all three levels of cache. Infact, if we look at the exact number of cycles for the same input size in L1, L2, and L3 cache we can see that L3 cache takes slightly longer to complete than L1 cache.

We believe that the main reason this phenomenon occurs is because a cache miss for a load word operation takes significantly longer to complete when all three caches are enabled. Furthermore, our data set length is at a maximum of 2048 words while our L1 cache has 1024 lines with 4 words each, effectively storing 4096 words. This renders the three different levels of cache to be just as effective as just one level of cache. Thus, in order to visualize a distinction between three levels of cache, we lower the cache sizes. Finally, unlike matrix multiplication, the actual data is constantly changing and will result in significantly more cache misses. During matrix multiplication, the original data from matrix 1 and matrix 2 is never altered. The result is simply stored in another matrix 3. However, in exchange sort, we write back to DRAM whenever two elements next to each other are not in the correct order. This write back operation will invalidate the lines in cache as the architecture is predefined to be a write-through, no allocate cache. Since this invalidation occurs every time the exchange sort loops through the entire set of data, it will also impact the results. Figure 10 displays the result and table 5 lists the details with smaller cache sizes for which settings are displayed below:

- DRAM: 2^{16} lines
- L1 Cache: 2^{6} lines, 1 word per line
- L2 Cache: 2^{8} lines, 1 word per line
- L3 Cache: 2^{10} lines, 1 word per line

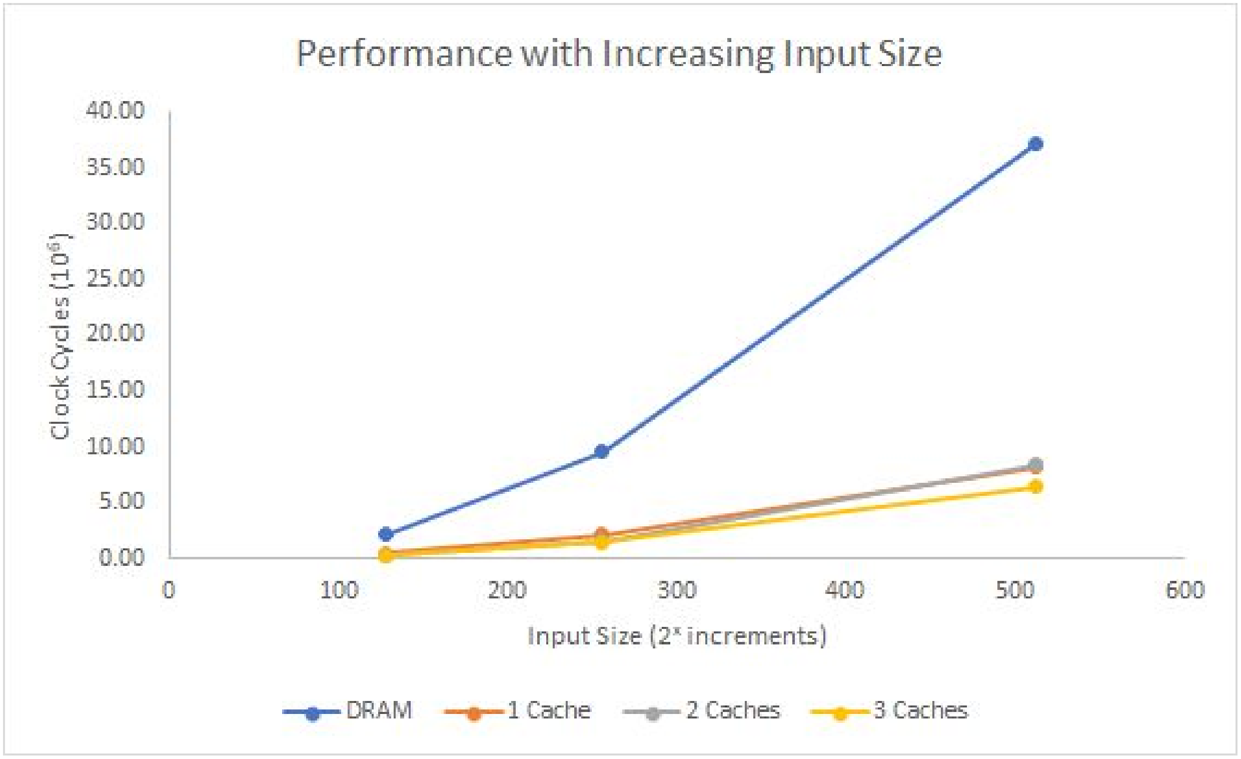
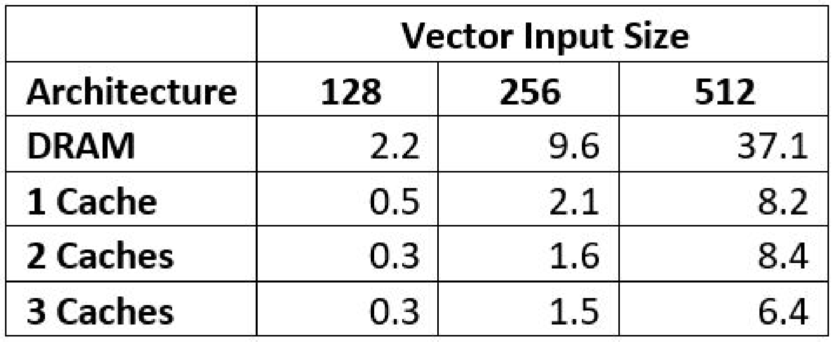


Figure 10: The plot above showcases the improvement by including multiple levels of cache. The yellow line which represents three levels of cache performs the best across all different input sizes.

Table 5: Cycle count in millions of cycles for varied input size



	Vector Input Size		
Architecture	128	256	512
DRAM	2.2	9.6	37.1
1 Cache	0.5	2.1	8.2
2 Caches	0.3	1.6	8.4
3 Caches	0.3	1.5	6.4

In general we see that when the cache size is smaller than the size of the data, architecture with three levels of cache requires a lower number of clock cycles and shows an increase in performance. The only exception seen in the table above was with the input size of 128 words. Both settings of 2 levels of cache and 3 levels of cache produce similar results. However, this scenario can be easily explained by the fact that the size of L2 cache is set to 2^8 which is 128 words (1 word per line). If all of the queried data can fit within L2 cache, existence of L3 cache will not assist in improving the performance.

Conclusion & Lessons Learned

This project was extremely valuable learning experience for us to learn the MIPS architecture and assembly language in detail. While it is one of the simpler architectures, it was really interesting to learn about all of the different components that make up an instruction set and an architecture.

There were two turning points in this project. In March, we implemented most of the functionality of the GUI and the simulator. At a certain point, we found certain parts of the simulator easier to understand and debug by using the GUI, since we display the contents of the pipeline buffer between each pipeline stage. Then, in early April, we finished encoding and implementing the full instruction set and the GUI. We had developed an assembly program and started re-running it with different configurations; for example, with and without pipelining, with varying levels of cache associativity, and with varying delay for memory access. It was very rewarding to find that ELISA was not only working, but that it was actually useful--in the first case as an assembly debugging tool, and in the second as a way to understand the impact of memory, cache, and pipelining configuration on program performance.

Writing the benchmarks gave us an appreciation of how significant memory access is in a typical program. For example, in the exchange sort program, every exchange of two unordered numbers required two loads, the comparison, and then two stores to swap their order places. As a result, caching plays a huge role in improving performance, reducing several hundred cycles of memory access to just a few once the data is cached.

Another observation we noticed was that for our simple memory model, a direct-mapped cache remained mostly empty. Since data and instructions were stored in consecutive lines in memory, they mapped to a relatively small portion of the cache, leading to an effective cache size much smaller than the actual cache size. While increasing associativity helps reduce cache evictions, the cache is better utilized when we use more memory.

From the benchmarks, we gained valuable insight into the advantages and disadvantages of different schemes of cache and memory. We used matrix multiplication to understand the impact of enabling and choosing the size of cache, choosing different numbers of words per line in cache, and selecting different levels of associativity. With exchange sort we visualized the impact of enabling three levels of cache with data that is smaller and larger than the size of cache itself.

Both these benchmarks confirmed our previously held belief that significant performance improvements occur by enabling pipeline and caches. What was interesting to understand through this project was that simply enabling the pipeline did not produce a 5 fold increase in performance. This occurs due to a significant number of memory access requests which increased the total number of clock cycles. Once at least a single cache level was enabled, performance increased by almost 5 folds.

In matrix multiplication benchmark, we varied the size of cache by either changing the number of lines or the number of words per line. After the size of the cache grew larger than the effective size of data requiring memory access, we did not see significant improvements in performance. The interesting feature that was relatively less obvious was that increasing cache size by increasing the number of words per line provided slightly better results than simply increasing the number of lines. We reasoned that this feature occured because one line containing multiple words will decrease the total number of cache misses. Similarly, when we increased the associativity, we saw an initial boost in performance. However, this turned around when associativity got too high since we were effectively lowering the total cache size.

In the exchange sort benchmark, we first saw that including 3 levels of cache did not produce better results. Infact, we saw a small increase in the total number of clock cycles for 3 levels of cache compared to only 1 level of cache. This occurred because our data was smaller than the cache size and a cache miss with more cache levels incurs more delays. After proceeding to a much smaller cache size, we started to see the benefits of multiple layers of cache.