CS535 FINAL REPORT: Team I Love You R3000

Srinanda Yallapragada, Ronan Salz, Jiangyi Qiu

Architecture Description

Our team was inspired by the high performance R3000 MIPS processor that was used in the Playstation 1 as well as the New Horizons space probe. We chose to implement a flavor of the MIPS-1 ISA, which uses a 32-bit word and a fixed-length 32-bit instruction, meaning we have a single instruction per word. This allowed us to support standard arithmetic instructions, such as addition, subtraction, multiplication, division, shifting, as well as logical operations, jumps, and branching. We designed the architecture to support 32 registers, each with a 32-bit width. While we captured many nuances of both the R3000 and the MIPS-1 ISA, there were two specific behaviors we aimed for in our project that we were not able to capture.

In terms of memory, we decided to support up to 1024 memory addresses or 2^10 addresses. With some changes, we could support up to 2^26, but for our purposes, we found 2^10 to be sufficient. To navigate this address range, we implemented several addressing modes, including immediate, PC+immediate offset, register direct, register indirect, and base+index+offset addressing modes.

Our memory system consists of two caches and one shared main memory. We split the memory into a data cache and an instruction cache, both of which access the same main memory. We made the decision to poll the data cache in the memory stage of the pipeline, while the instruction cache is called in the fetch stage. This decision was made because we anticipated running benchmarks on the simulator, which would lead to infrequent eviction of data instructions. Therefore, we opted to have a separate data cache that would evict data while keeping instructions in the instruction cache for longer periods.

We designed our caches to include a dirty bit and a valid bit, and we chose to use the write-back policy. When cache evicts, we write to memory. For cache write misses, we use write around policy.

Supported Instructions List

Our simulator supports almost all of the MIPS-1 architecture. An exhaustive list of supported instructions follows immediately below.

Name (opcode)	Example	Description + implementation notes
Add (0b000000)	add r0 r1 r2	r0 ← r1 + r2
Subtract (0b000001)	sub r0 r1 r2	r0 ← r1 - r2
Multiply (0b000010)	mul r0 r1 r2	r0 ← r1 * r2 If results are larger than 32 bits, we ignore the largest order bits and keep the lower order 32 bits in r0.
Divide (0b000011)	div r0 r1 r2	r0 ← r1 / r2
Mod (0b000100)	mod r0 r1 r2	r0 ← r1 % r2
Logical And (0b001010)	and r0 r1 r2	r0 ← r1 & r2
Logical Or (0b001011)	or r0 r1 r2	r0 ← r1 | r2
Logical Xor (0b001100)	xor r0 r1 r2	r0 ← r1 ^ r2
Logical Nor (0b001111)	nor r0 r1 r2	r0 ← $\overline{r1 \mid r2}$
Jump from Reg (0b001110)	jr r0	ProgramCounter ← r0
Jump from Reg and Link (0b001111)	jral r0	r31 ← ProgramCounter; PC ← r0
Move (0b010000)	move r0 r1	r0 ← r1
Set if Equal (0b01001)	seq r0 r1 r2	if r1 == r2, then r0 ← 1 else r0 ← 0
Set if Greater Than or Equal (0b010011)	sge r0 r1 r2	if r1 >= r2, then r0 ← 1 else r0 ← 0
Set if Greater Than (0b010010)	sgt r0 r1 r2	if r1 > r2, then r0 ← 1 else r0 ← 0
Set if Less Than or Equal (0b010100)	sle r0 r1 r2	if r1 =< r2, then r0 ← 1 else r0 ← 0

Set if Less Than (0b010101)	slt r0 r1 r2	if r1 < r2, then r0 ← 1 else r0 ← 0
Set if Not Equal (0b010110)	sne r0 r1 r2	if r1 != r2, then r0 ← 1 else r0 ← 0
Add Immediate (0b010111)	addi r0 r1 x	r0 ← r1 + x
Subtract Immediate (0b011000)	subi r0 r1 x	r0 ← r1 - x
Multiply Immediate (0b011001)	muli r0 r1 x	r0 ← r1 * x
Divide Immediate (0b011010)	divi r0 r1 x	r0 ← r1 / x
And Immediate (0b011100)	andi r0 r1 x	r0 ← r1 & x
Or Immediate (0b011101)	ori r0 r1 x	r0 ← r1 | x
Xor Immediate (0b011110)	xori r0 r1 x	r0 ← r1 ^ x
Store Word (0b011111)	sw r0 r1 x	[[r1] + x] ← r0
Load Word (0b100000)	lw r0 r1 x	r0 ← [[r1] + x]
Load Immediate (0b100001)	li r0 x	r0 ← x
Branch (0b100010)	b 0x1234	ProgramCounter ← 0x1234
Branch if Equal (0b100011)	b r0 r1 0x1234	if r0 == r1, PC ← 0x1234
Branch if Not Equal (0b100100)	bne r0 r1 0x1234	if r0 != r1, PC ← 0x1234
Branch if Equal to Zero (0b100101)	beqz r0 0x1234	if r0 == 0, PC ← 0x1234
Branch if Not Equal to Zero (0b100110)	bnez r0 0x1234	if r0 != 0, PC ← 0x1234
Branch if Greater Than (0b100111)	bgt r0 r1 0x1234	if r0 > r1, PC ← 0x1234
Branch if Less Than (0b101000)	bgt r0 r1 0x1234	if r0 < r1, PC ← 0x1234
Branch if Greater Than Zero (0b101001)	bgtz r0 0x1234	if r0 > 0, PC ← 0x1234
Branch if Less Than Zero (0b101010)	bltz r0 0x1234	if r0 < 0, PC ← 0x1234

Branch if Greater Than or Equal To (0b101011)	bgte r0 r1 0x1234	if r0 >= r1, PC ← 0x1234
Branch if Less Than or Equal To (0b101100)	blte r0 r1 0x1234	if r0 <= r1, PC ← 0x1234
Branch if Greater Than or Equal To Zero (0b101101)	bgtez r0 0x1234	if r0 >= 0, PC ← 0x1234
Branch if Less Than or Equal To Zero (0b101110)	bltez r0 0x1234	if r0 <= 0, PC ← 0x1234
Jump (0b101111)	j 0x1234	PC ← 0x1234
Jump and Link (0b110000)	jal 0x1234	r31 ← PC+1; PC ← 0x1234
nop (0b110001)	nop	Do nothing
Halt and Catch Fire (0b111111)	hcf	End program execution

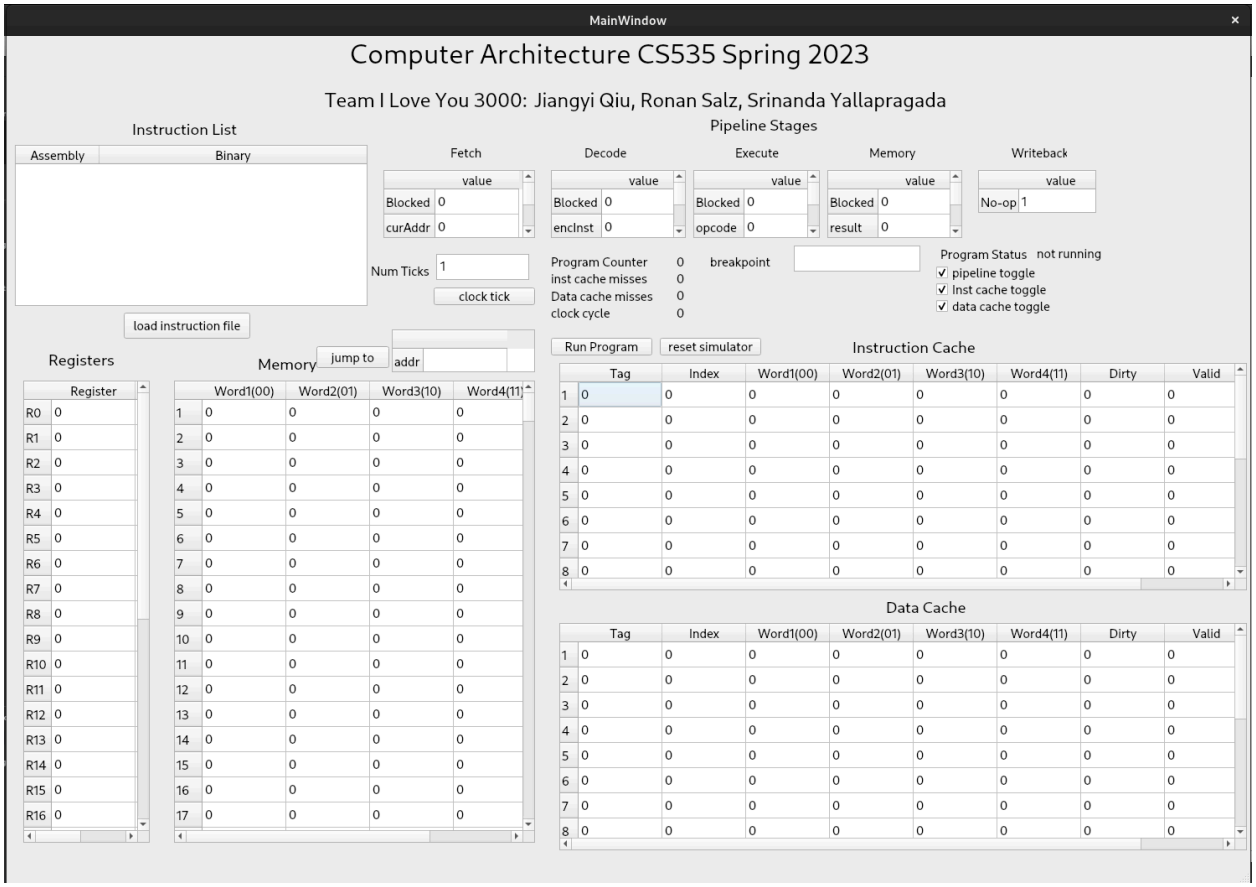
Simulator Description

Our simulator GUI was designed using the Qt framework. The GUI provides a user-friendly interface for interacting with the simulator, allowing users to easily visualize the state of the processor and the program being executed. We chose to use the Qt framework due to its cross-platform compatibility, which allows our simulator to be run on Windows, Linux, and macOS as well as its ease of use due to the Qt creator development environment.

The main window of the simulator GUI consists of several views, including registers, memory, caches, and pipeline. The register view displays the contents of each of the 32 registers, while the memory view allows users to view the contents of individual memory locations. The cache view displays the contents of the data and instruction caches, along with their respective valid and dirty bits. The pipeline view shows the current state of each stage of the pipeline, including any stalls that may be occurring.

In addition to these views, the GUI provides buttons for stepping through the program one instruction at a time or running the program at full speed. The program counter and other important program status information is also displayed, allowing users to easily monitor the progress of their program.

The GUI also includes support for loading and running user-written assembly programs using .ily file extensions. Users can simply write their program in our assembly language and load it into the simulator. We also provide multiple toggles to allow for performance monitoring in different states of the pipeline, and our split caches.



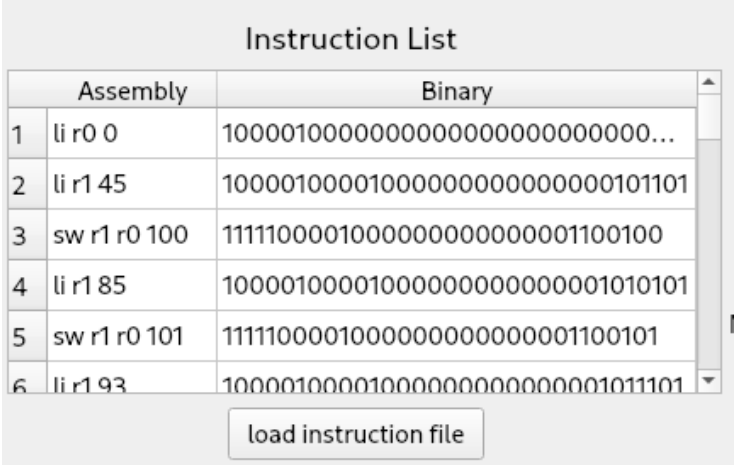
Steps to run a benchmark

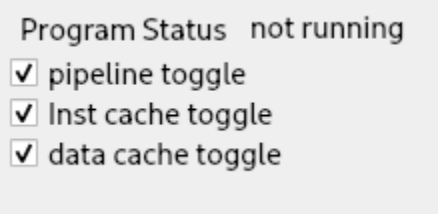
1) Run the simulator by executing the "./simulator" command in the "src/simulator" directory, after building it with Qt. Note that the setup for Qt installation is required.

2) Load your instruction file by selecting the "Load Instruction File" button in the middle-left of the simulator screen.

3)Select your ".ily" file that you want to load into the simulator.

4) Upon successful loading, the instruction list will be populated.

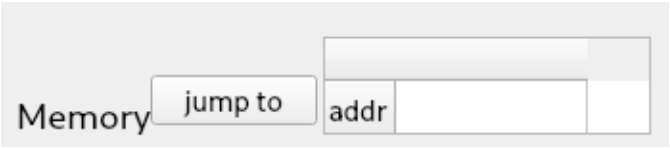




5)

a) Select your desired running modes using the toggle pipeline, inst cache and data cache checkboxes
b) Set your breakpoints. Leaving the breakpoints box blank means that no breakpoint is set. To set a breakpoint, input a program counter value into the breakpoint.
c) NumTicks can be updated to any value, this will allow you to jump a certain number of clock cycles forward while using the clock tick button.

6) Select the "Run Program" button to execute your program.
7) To jump to a particular address in memory view, use the "Jump to Addr" button.



8) After the program has executed, you can reset the simulator by clicking the "Reset Simulator" button. However, do not click the reset button during program execution. Otherwise, unintended behavior will occur.

Development Path:

Our project began with the development of the assembler, which we successfully built within the first two weeks of starting the simulator. We then focused on developing the memory system, which we represented using a 2D array and implemented load and store functions. We added timing logic to ensure proper execution of these functions.

Building on top of the memory system, we developed the cache system, which initially lacked any dirty bit functionality, resulting in write around to the main memory. Once we were confident in the effectiveness of the memory and cache systems, we started working on the pipeline and simulator GUI. This was one of the more complex phases of the project, as it required us to design a data structure for the pipeline and develop functions for managing pipeline stages while considering dependencies between instructions.We found that the dependency list aspect of the project to be the most challenging, and difficult to implement retrospectively, that is, having already built out most of the pipeline.

After connecting all components and ensuring the correctness of our dependency list, we had a working demo of the GUI and some basic instructions of our ISA. We continued to work on the simulator by adding the dirty bit functionality to our cache system and splitting it into data and instruction caches. We accomplished this by pointing the fetch stage to an instruction cache and

the memory stage to a separate data cache. This allowed us to identify data and instructions in memory efficiently. Finally, we tested the entire simulator and wrote benchmarks to evaluate its performance.

Software engineering methods:

Our team used GitHub as our primary version control software throughout the development of the simulator (https://github.com/rsalz47/i-love-you-r3000). We created multiple branches to work on different parts of the project simultaneously, allowing us to collaborate and merge changes seamlessly. We also utilized GitHub's pull request feature, which allowed us to review and discuss code changes before merging them into the main branch.

Communication was essential to the success of our project, and we primarily used Discord as our communication tool. We held regular meetings, both in-person and virtual, to discuss design decisions, technical issues, and progress updates. Discord's screen-sharing feature proved to be particularly helpful in troubleshooting code and clearing any developmental blocks.

To ensure interoperability, we agreed on function signatures before writing the functions, which helped maintain consistency across the different parts of the simulator. This approach also allowed us to work on different parts of the project simultaneously without interfering with each other's work. Header files and class definitions were agreed upon together as groups, and then assigned individually for implementation purposes. We would often communicate in pairs as group member A interacted with a module developed by group member B, and would occasionally have pairwise meetings to handle issues that involved two intersecting modules.

Work Done by Person:

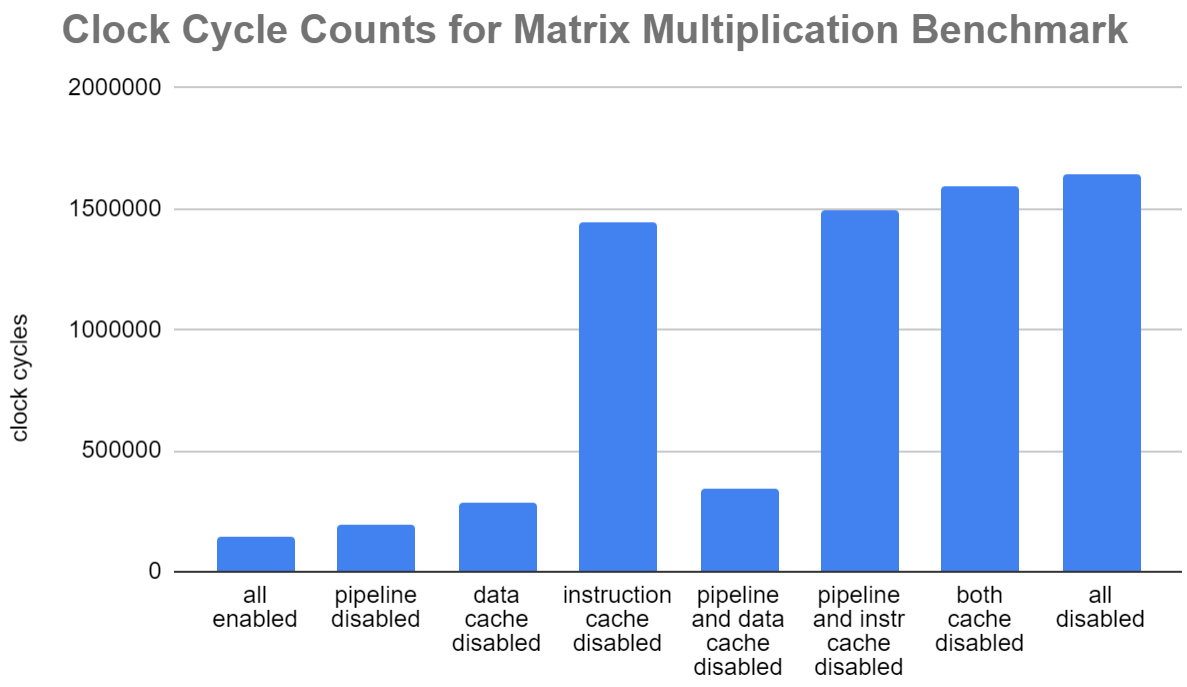
Srinanda Yallapragada	Ronan Salz	Jiangyi Qiu
Simulator GUI	Assembler	Pipeline
Memory system	Base Cache model	Instruction set population
Splitting L1 cache into Inst and Data caches	Pipeline	Exchange Sort Benchmark
Dirty Bit Implementation	Instruction set population	Memory Demo cmd
Fibonacci Benchmark	Matrix Multiply Benchmark	Dirty Bit Implementation
Timing Logic for Memory System	Dependency List	Dependency List
Final Report Draft 1 Main Writer	Final Report Draft 1 Editor	Final Report Draft 1 Editor

Performance Report:

Benchmark	Configuration Details	Results
Matrix multiply (multiply a 10x10 matrix with another 10x10 matrix) Mem delay = 100 Data cache = 2 Inst cache = 2	Pipeline ✔ Data cache ✔ Instruction cache ✔	Clock cycles = 142338 Data Cache misses = 526 Inst Cache misses = 158
	Pipeline × Data cache ✔ Instruction cache ✔	Clock cycles = 197073 Data Cache misses = 526 Inst Cache misses = 158
	Pipeline ✔ Data cache × Instruction cache ✔	Clock cycles = 285189 Data Cache misses = 0 Inst Cache misses = 158
	Pipeline ✔ Data cache ✔ Instruction cache×	Clock cycles = 1443939 Data Cache misses = 526 Inst Cache misses = 0
	Pipeline × Data cache × Instruction cache ✔	Clock cycles = 339873 Data Cache misses = 0 Inst Cache misses = 158
	Pipeline × Data cache ✔ Instruction cache ×	Clock cycles = 1498295 Data Cache misses = 526 Inst Cache misses = 0
	Pipeline ✔ Data cache × Instruction cache ×	Clock cycles = 1591339 Data Cache misses = 0 Inst Cache misses = 0
	Pipeline × Data cache × Instruction cache ×	Clock cycles = 1641095 Data Cache misses = 0 Inst Cache misses = 0
Exchange sort (sort a random order of ints from 1-100 in ascending order) Mem delay 100 Data cache 2 Inst cache 2	Pipeline ✔ Data cache ✔ Instruction cache ✔	Clock cycles = 220919 Data Cache misses = 601 Inst Cache misses = 55
	Pipeline × Data cache ✔ Instruction cache ✔	Clock cycles = 358063 Data Cache misses = 601 Inst Cache misses = 55
	Pipeline ✔ Data cache × Instruction cache ✔	Clock cycles = 1620099 Data Cache misses = 0 Inst Cache misses = 55

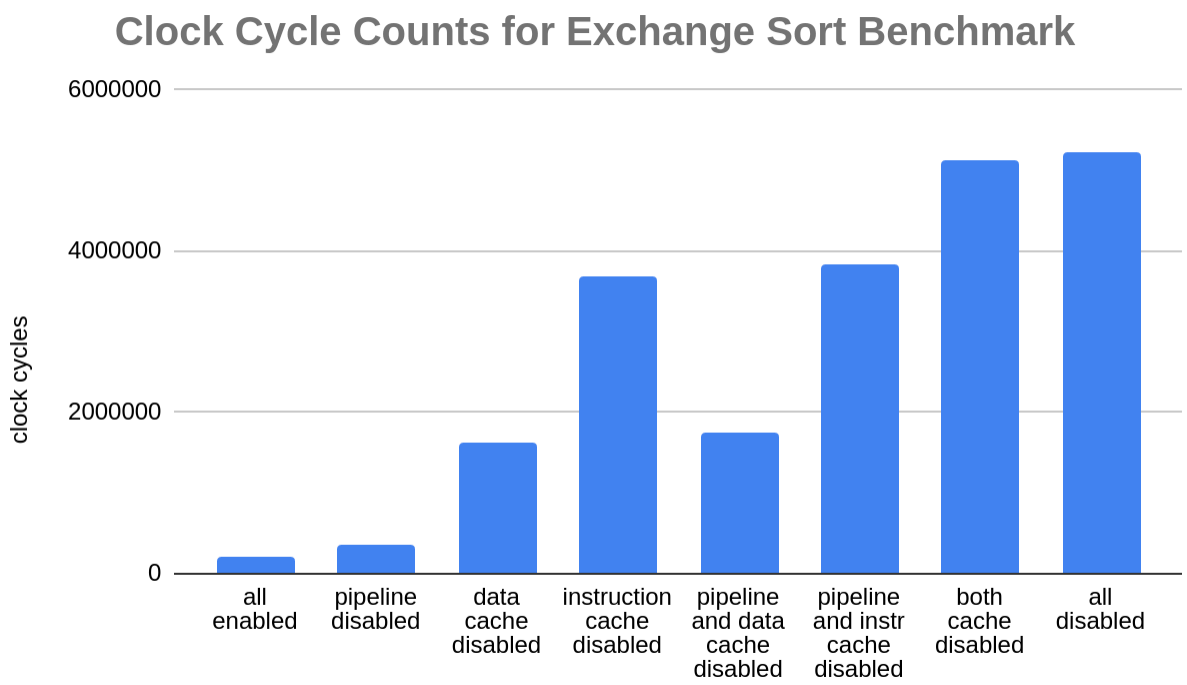
	Pipeline ✔ Data cache ✔ Instruction cache×	Clock cycles = 3688322 Data Cache misses = 601 Inst Cache misses = 0
	Pipeline × Data cache × Instruction cache ✔	Clock cycles = 1762295 Data Cache misses = 0 Inst Cache misses = 55
	Pipeline × Data cache ✔ Instruction cache ×	Clock cycles = 3830093 Data Cache misses = 601 Inst Cache misses = 0
	Pipeline ✔ Data cache × Instruction cache ×	Clock cycles = 5125025 Data Cache misses = 0 Inst Cache misses = 0
	Pipeline × Data cache × Instruction cache ×	Clock cycles = 5234325 Data Cache misses = 0 Inst Cache misses = 0

Analysis of Performance:



Examining the matrix multiplication benchmark data, we can see that the instruction cache is quite possibly the singular most important component of our entire CPU simulator. With just the instruction cache disabled, CPU performance drops to almost the minimum of all configurations tested, and the performance benefits provided by the instruction cache alone are greater than

the benefits provided by the pipeline and data cache combined. This appears to be a four times speedup. Additionally, we can note the pipeline alone does not provide a particularly large speed up on its own; the difference between everything disabled and only pipeline enabled (the last two columns) is negligible.



In the exchange sort benchmark, we see once again that the instruction cache is the single most important factor for performance. However, unlike the matrix multiplication benchmark, we see that there is a greater performance loss due to the data cache being disabled as the exchange sort has much more data access compared to matrix multiplication. This explains the larger gap between pipeline disabled and data cache disabled as well as pipeline and instr cache disabled and both cache disabled. We also see that the exchange sort is a much longer benchmark needing many more cycles to complete compared to the matrix multiplication. Once again, the change with or without the pipeline is negligible.

Roadblocks and Design Decisions:

In building our MIPS architecture simulator, we gained valuable insights into the challenges of designing and implementing a complex system with many small but interdependent parts. We learned that developing these parts independently could lead to errors and difficulties when bringing them together, requiring careful testing and debugging.

One of the main design decisions we made was limiting our memory to 2^10 words of memory. While this limited our memory size significantly, it allowed us to develop our programs more efficiently as we were able to wrap our heads around the smaller memory addresses much

easier. We also found that the limited memory was still enough to fit our benchmark tests and so in this instance we did not find our limited memory to be a bottleneck.

Another location where we had to make changes was that our initial proposed instruction set had many instructions that we would not end up using. As we started writing our benchmarks, we found that we did not use a few of the instructions that we were intending to implement so we chose to cut down on the number of instructions we would support and this is reflected in the difference between our proposal ISA and the instructions we actually support.

One challenge we faced was working with signed ints. We chose to use uint32_t type ints for all of our pipeline and memory stage processing assuming that whatever interprets the numbers will use two's complement to convert it into negative ints. (typecasting). This has resulted in unexpected behaviors while trying to utilize unsigned ints. Running into these issues also prevented us from implementing floating point instructions which was one of our initial goals

We found ourselves limited by the 32bit uint limit as well as when we tried to compute fibonacci numbers, we found that we would wrap around and get unexpected behavior trying to compute anything beyond 40 numbers in the series.

For non technical team management issues, Srinanda fell ill and was unable to maintain his productivity towards the end of the semester, so we shifted the workloads in a way that allowed us to complete the tasks on time. Srinanda picked less technical tasks such as writing up the report while Ronan and Jiangyi picked up writing the benchmarks.

Finally, we encountered challenges related to the long-term maintenance of our code. In hindsight, we realized that having more macro declarations would have facilitated our ability to play around with different values for our memory systems. Specifically, we found that the lack of a macro for cache lines hindered our ability to test different cache sizes, as we had only created one for memory.

Learning Summary:

This project taught us the values of many traditional programming paradigms. We developed best practices for code maintenance as we advanced further in the project. As found ourselves bumping into other team members' code, we had to develop consistent APIs that team members would expect to interact with, so project work could proceed at an acceptable pace. It also taught us the importance of writing code you can look back on: as we hooked up the memory system to the pipeline, we learned that these individual modules are much less "fire-and-forget" than we had hoped. The emergent behavior of two pieces of code working in conjunction can be wildly more complex than either independent constituent part, and so we learned to future proof our code so that bugfixing and modifying prior work was approachable.

We also got intimately familiar with how the pipeline and memory interact. Before working on the pipeline, we did not expect so many edge cases and difficulties to pop up. By the end of the project, though, we found that the pipeline messes up a lot of prior modules, such as the memory subsystem. The data and control hazards that can occur in the pipeline make the entire operation much more complicated than we initially expected.