EECS151 FPGA Lab

Final Project Report

Aaron Zheng Isaac Tsang

University of California, Berkeley

aaron.zheng@berkeley.edu isaacmiltontsang@berkeley.edu

November 18, 2025

Contents

1 Introduction

We present a 4-stage RISC-V processor core that integrates UART and broad memory-mapped IO functionality, following the RV32I instruction set divided into Instruction Fetch, Instruction Decode, Execute, and Writeback Stage. Our task was to optimize our processor on the "mmult.c" matrix multiplication task. Our initial processor performed at 65Hz and a final FoM of 30. Through pipelining, we were able to speed our processor up to 105 Hz and achieve a final FoM of 61.

2 Architecture Overview

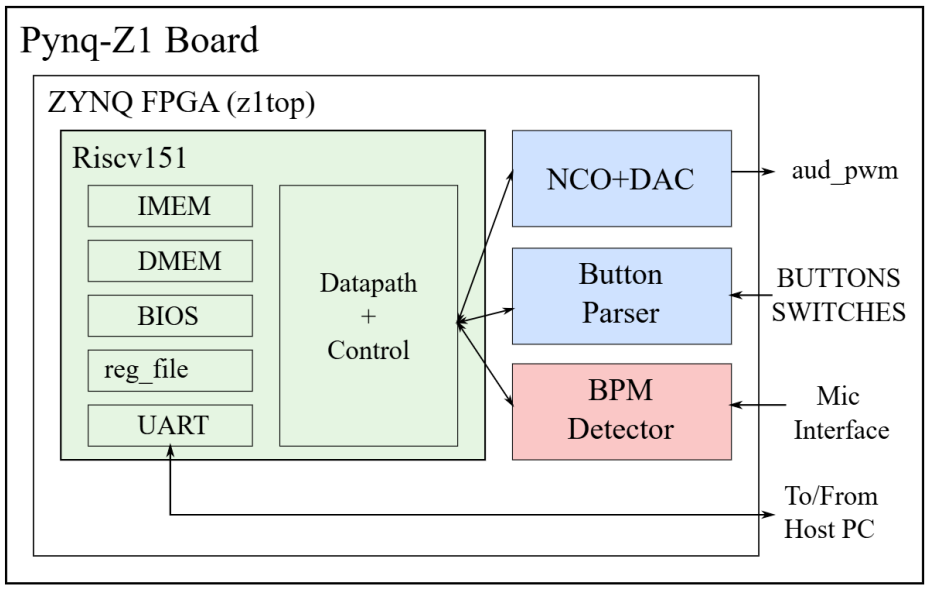

Figure 1: Project Overview: Block Level Diagram

The objective of our project is to guide the development of a custom pipelined RISC-V CPU. Our implementation focuses on the Green Components (labelled Riscv151) as per Figure 1, and our block interacts with the host PC through UART with memory-mapped I/O. We synthesize our CPU Processor on the Pynq-Z1 Board. Our processor uses three different memories, BIOS, instruction memory (IMEM) and data memory (DMEM). On boot, the system executes a small BIOS program which is read from BIOS memory. It tells the machine to initialize itself and get the instruction data from the UART into IMEM. Our CPU logic is able to write and read to DMEM through a user interface, and perform complex calculations efficiently. Our CPU does not interact with the audio, the buttons, and the mic interface of the Pynq Z1, as that is not the focus of the project. The optimization goal of the project is to **minimize execution time** of the mmult program, following the Iron Law of Processor Performance.

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

Figure 2: Iron Law of Processor Performance

As per Figure 2, the Instructions / Program is fixed, so we seek to minimize Cycles / Instruction (CPI) and Time / Cycle (Clock period) to achieve the minimization of execution time. Our project also uses Figure of Merit(FOM), a different metric more aligned with the metrics used in industry.

$$FOM = \frac{F_{max}}{CPI \cdot \sqrt{cost}}$$

Figure 3: Figure of Merit

2.1 Pipeline Stages

A key design decision was where to place our pipeline registers and how many pipelined stages we should have. Initially, we decided on the recommended 3 stage pipeline architecture with the Instruction Fetch, Execute, and Writeback stage, combining the Register File (RegFile) and the ALU into one Execute stage. However, we quickly found out that the four stage pipelined architecture offered significant speedup. Hence, we now have a 4-stage pipelined architecture, and they are Instruction Fetch, Instruction decode (getting instruction and reading from RegFile), Execute (reading from ALU) and Writeback. Our (unoptimized) architecture is shown in Figure 4.

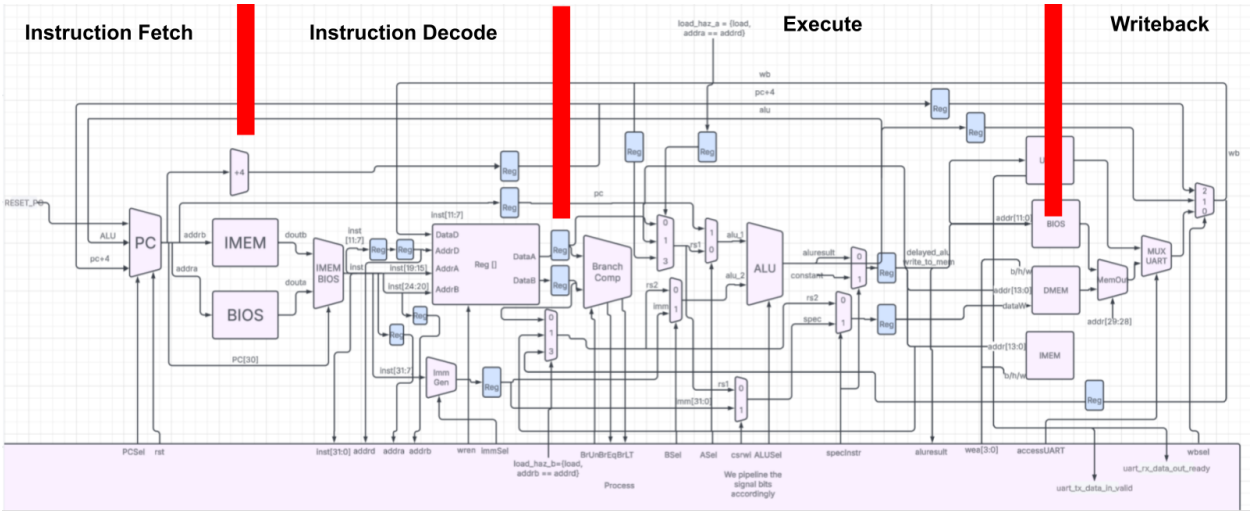

Figure 4: Initial CPU Pipelined Register

2.2 Memory Components

Our pipeline supports four distinct types of memory, each serving a specific role.

1. **BIOS Read-Only Memory**: Stores the initial BIOS program. It is read-only to prevent modification during execution. Typically, it initializes the CPU and loads the main program into memory.
2. **Data Memory (DMEM)**: Memory used for storing data during program execution. This includes stack variables, global variables, and dynamically allocated heap data. Supports both read and write operations.
3. **Instruction Memory (IMEM)**: Contains the program instructions to be fetched and executed by the CPU. While it is read-only during normal execution, it can be written during bootloading or program loading stages.
4. **Memory-Mapped I/O (MMIO)**: Facilitates interaction with different compoents by assigning specific memory addresses to I/O devices. Enables UART, allowing this datapath to communicate with host PC.

2.3 Synchronus Memory Components

In our implementation, we deal with synchronus memory components in our pipelines by keeping them in sync with our pipeline registers. These memory registers are synchronus write, asynchronus read. Our pipeline registers align perfectly with the nature of the memory components, essentially treating them as pipeline registers. In this way, we are careful not to overpipeline our circuit.

2.4 Critical Paths and Data Forwarding

Our program includes processing for two different hazard types: Data and Control Hazards. NOOP here refers to an empty instruction with value 0x13 (addi x0 x0 x0), which we pass in through the datapath when we want to null an instruction. On the other hand, PC refers to the Program Counter and defines the memory address from IMEM that we read from at any given time.

Data Hazards:

Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its execution. In a pipelined processor, this can lead to incorrect computation if the dependent instruction reads stale data. The most common form is the read-after-write (RAW) hazard, where, for example, instruction ADD x3, x1, x2 is immediately followed by SUB x4, x3, x5. In this case, the SUB instruction requires the result of the ADD, but the value of x3 is not yet written back to the register file.

To handle this, we initially implemented data forwarding (also known as bypassing), which allows intermediate results from the Writeback(WB) stage to be routed back to the Execute(EX) stage. We implement forwarding from WB to EX to make forwarding work for all data hazards, such as MEM-to-ALU (ex. lw x1 0(x3), add x3 x2 x1).

However, this solution was functional but too slow, and led to an excessively long critical path that intertwined between the pipelined stages of WB and EX. To shorten the critical path and hence decrease minimum clock period, we decided on an approach that would calculate, at Decode stage, whether we would have a data hazard, conditioned on the current instruction at execute stage. If there is a Data Hazard, we would read from PC instead of PC+4 at IF and pass in NOOP to EX stage.

Control Hazards:

Control hazards arise from instructions that change the flow of execution, such as jumps (JAL, JALR) or conditional branches (BEQ, BNE, etc.). Since the outcome of these instructions is not determined until the Execute stage, subsequent instructions fetched speculatively may need to be flushed if the branch is taken. For example, if BEQ x1, x2, label is followed by instructions that would not execute if the branch is taken, those instructions must be discarded upon branch resolution.

To mitigate this, initially, our pipeline will add a NOOP as the next Instruction Fetch instruction if the previous instruction was a branch or a jump instruction. This will guarantee that there is no control hazard, as the next instruction after the branch / jump logic is executed 2, rather than 1 cycle after, allowing time for the ALU output from the jump / branch instruction to propagate back into PC.

However, while speeding up our CPU datapath, we realized that this process is unideal for BRANCH instructions, since we are always taking the case of the worst efficiency. This efficiency is that branch instructions always take two cycles to run, which we believe would unnecessarily increase our CPI. Instead, we assume branch not taken as our default value. On the off case where branch is taken in the Execute stage, we replace the instructions in the IF and ID stages with NOOP and change PC to the right value in the next stage. Assuming branch not taken is more common than branch taken, this approach is viable for speedup.

Summary:

Overall, our experiments found that both Data and Control Hazards can be more efficiently solved compared to our original implementations. Data Hazard's problem is the long critical path that it creates from its data forwarding paths, which can be solved by identifying the Hazard at Decode

stage and NOOP'ing the next instruction at Execute stage. Notably, we find this comes at a slight cost to CPI, but significantly increases the clock frequency, thus increasing overall FOM. Branch Control Hazards can be sped up by assuming Branch Not Taken by default, and NOOP'ing previous instructions if taken instead.

3 High Level Organization

To design a RISC-V CPU, it is very important to maintain verifiability, abstraction and structure. In order to achieve this, we began our design by designing a detailed block diagram, iterating several times before implementing it on Verilog RTL. This design is Figure 5.

Once we started Verilog, we inevitably had to change our design as we realized optimization gains and potential errors in our original block diagram. We tried to be consistent with changing the block diagram so it reflects the actual Verilog RTL. As a result, we also have a final design, which reflects in detail our actual Verilog implementation. Converting Block Diagrams to Verilog RTL and vice versa required high levels of abstraction and precision. The updated block diagram is shown in Figure 5.

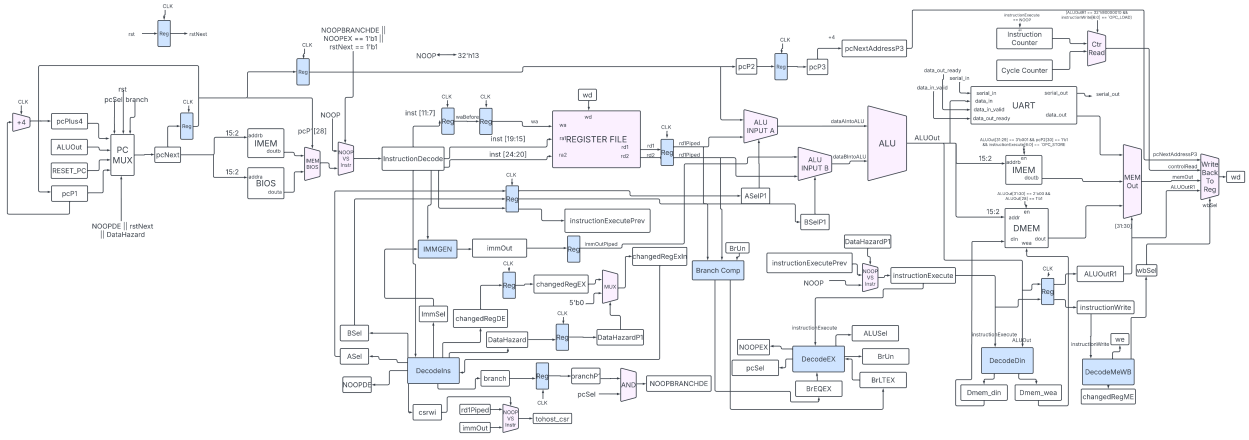

Figure 5: Updated Block Diagram

3.1 Pipeline Design

The four stages of our pipeline were carefully balanced to reduce timing violations. Our pipelined design includes Instruction Fetch, Instruction Decode, Execute, and Writeback. To minimise timing issues, we align the pipelined registers with the synchronus regfile and memory components.

3.2 Control Logic

To organize our control logic, we divided our control logic within the 4 stages into 4 separate blocks. Each block takes as input the **current** 32-bit instruction. This instruction is obtained in

the instruction decode, and pipelined all the way from execute to writeback, block to block. In this way, we can clearly tell what instruction each stage is executing and debug control logic of each stage separately. Control logic of each stage is defined with combinational case statements that depend on the input, which is mainly the instruction at current stage. For a full picture, please see the updated CPU sketch.

3.3 RISC-V Modules

For our program, there were key building blocks that we had to implement, many of which were complex and dependent on multiple conditions. In addition to the control logic blocks explained above, we had to implement separate blocks for our Arithmetic Logic Unit(ALU), Branch Comparator, and ImmGen, among others. The ALU is by far the most complex, as this module required us to have to define outputs for all the different arithmetic operations that a RISC-V processor could perform, such as sign extension, bit-wise OR, and left/right shifts. For each of these blocks, we carefully designed them and created customized test benches to test their functionalities. In addition, we also used 2-to-1, 4-to-1 multiplexer logic throughout the design, for selecting PCNext, IMEM vs BIOS, writeback to regfile data, as well as for data forwarding in the initial design.

3.4 Address Space Partitioning

Our memory read/write is dealt with in the Instruction Fetch and Write Back stages, which use the PC and the ALU output as the address, respectively. To separate reads and writes from the different memory blocks, we partition the 32-bit memory address space by the top 4 bits. The target memory block we use depend on Table 1.

Address[31:28]	**Address Type**	**Device**	**Access**	**Notes**
4'b00x1	Data	Data Memory	Read/Write	
4'b0001	PC	Instruction Memory	Read-only	
4'b001x	Data	Instruction Memory	Write-only	Only if PC[30] == 1'b1
4'b0100	PC	BIOS Memory	Read-only	
4'b0100	Data	BIOS Memory	Read-only	
4'b1000	Data	I/O	Read/Write	

Table 1: Memory Mapping by Address Bits

To calculate the IMEM and DMEM address bits, we take the bottom 16 bits of our 32-bit address space. The top 14 bits of these 16 bits will be the word address, indexing into one specific 4 byte word in the RAM, and the bottom 2 bits will be the byte offset, denoting which specific byte to be addressed. To enable write of less than a full word, we add a byte-mask, a 4 bit input where each bit aligns with a byte in memory, and its value determines whether to write to that byte.

3.5 Memory Mapped I/O through UART

Our CPU architecture uses memory-mapped I/O to communicate with the UART, where registers of I/O devices are assigned memory addresses, so load and store instructions can interact with I/O as they do memory. Here is the memory map.

Address	Function	Access	Data Encoding
`32'h80000000`	UART control	Read	{30'b0, uart_rx_data_out_valid, uart_tx_data_in_ready}
`32'h80000004`	UART receiver data	Read	{24'b0, uart_rx_data_out}
`32'h80000008`	UART transmitter data	Write	{24'b0, uart_tx_data_in}
`32'h80000010`	Cycle counter	Read	Clock cycles elapsed
`32'h80000014`	Instruction counter	Read	Number of instructions executed
`32'h80000018`	Reset counters to 0	Write	N/A

Table 2: Memory-Mapped UART and Counter Interface

4 Datapath and Control

Our datapath is a 4-stage pipeline containing Instruction Fetch (IF), Instruction Decode (ID), Execute (E) and Writeback (WB) stages. In our RISC-V processor design, the datapath integrates several key components crucial for the execution of instructions: the program counter (PC), Instruction Memory (IMEM), Register File (RegFile), ALU, Data Memory and multiple pipeline registers (between IF/ID, ID/EX, EX/WB) to pipeline our datapath and reduce the maximum combinational path. The high level block diagram is shown in Figure 5.

To organize our code, we create blocks for the control logic at the Instruction Decode, Execute and Writeback stages, each one taking in an input that is the current instruction at that stage. This allows us to more easily control the control logic, and minimize human error. Each of our control blocks are **combinational logic blocks**, implemented via a Verilog *always* block with case statements, based on specific parts of the instruction and relevant inputs.

Our stalling logic is unique and is a refinement of our initial approach, a simple data forwarding technique that forwards data from the Writeback Stage into registers in the execution stage, conditioned on the Writeback address being equal to those register addresses. To reduce the critical path caused by this data forwarding, we took the approach of detecting data hazards at Decode stage, then NOOP'ing the next instruction at Execute stage and stalling PC. This logic is relatively simple to implement, thanks to the separated control logic block that we implement for each stage, allowing us to get control logic at each stage dependent on inputs from other stages.

A non-standard addition in our design is a branch prediction bypass path that assumes "branch not taken" and squashes instructions upon incorrect prediction—an approach that optimizes branch

instructions. The reasoning behind this design choose is because this offers us an increased of CPI (a boost from around 1.4 to 1.32), without much increase in cost. We have considered implementation a non-naive branch predictor. However, we did not think that the increase of cost in the program would offset the increase of area cost and potential Fmax loses. We also considered an implementation of branch take, but that, under our consideration, would decrease our FOM.

5 Verification

5.1 Testing and Bug Fixing

Before starting our CPU design, we wrote simple testbenches for the main modules: the ALU, Branch Comparator, Immediate Generator (ImmGen) and the RegFile. This allowed us to avoid errors down the line, ensuring that most errors later can be traced directly to cpu. Upon passing, we then built the main CPU architecture.

We mainly used the testbenches provided by staff, including `cpu_tb.v`, `asm_tb.v`, `isa_tb.v`, and all the others. We believe these benches to be rather sophisticated in their ability to test for problems, as they allowed us to fix multiple errors, like branch forwarding errors, wrong jump ASel error, wrong BSel for branch instructions, and many others. The `echo_tb.v` we found very helpful, as it helped us catch a major bug in our initial UART implementation, where we did not make asynchronous write possible due to our pipelining logic, and instead used the `data_out_valid` signal from the **current** cycle as opposed to the pipelined `data_out_valid`. We also wrote some testbenches ourselves to check errors when it aroused, such as writing a testbench checking the JAL is jumping from BIOS to IMEM correctly, as we had trouble debugging.

Our most significant bug stemmed from an issue with `IMEM`, which was undetected by all the provided testbenches but caused our `jal` instruction—responsible for beginning execution of the `mmult.c` program—to fail. Initially, we believed this to be a bug in the `jal` implementation, and we debugged waveforms instruction by instruction. After hours of struggling and addressing smaller issues, we observed that `bios_tb.v` exhibited inconsistent behavior, sometimes passing and sometimes failing. This prompted us to analyze its result logs more deeply. Although all tests reported as passed, we discovered that reads to `IMEM` were silently failing. Further waveform inspection and code review revealed that the write enable (`we`) signal for `IMEM` was misconfigured, preventing instruction memory from being written properly. In hindsight, writing a custom testbench to directly verify memory components earlier in the process would likely have reduced the time spent diagnosing this bug.

5.2 SystemVerilog Assertions

An assertion we wrote to check if Program Counter(PC) resets to the right address is as follows:

```
parameter logic [31:0] PC_RESET =32'h4000_0000;
property pc_resets_to_reset_addr;
  @(posedge clk)
  rst |=> pc == PC_RESET;
endproperty
assert property (pc_resets_to_reset_addr);
```

This assertion first defines the default value of `PC_RESET`, which is set equal to the previously defined `RESET_PC`. It then checks that, on the rising edge of the clock, if `rst` is asserted, the program counter (`pc`) will be set to `PC_RESET` on the following clock cycle.

We also wrote other various SVA assertions, which can be seen on GitHub.

6 Status and Results

6.1 Functionality Summary

- **Working:** All components are working as expected.
- **Not Working / Partially Working:** NONE

6.2 Checkpoint Metrics

- **CPI after Checkpoint 2/3:** 1.35

6.3 FPGA Resource Utilization (Checkpoint 4)

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	418	0	0	13300	3.14
SLICEL	276				
SLICEM	142				
LUT as Logic	1227	0	0	53200	2.31
using O5 output only	0				
using O6 output only	1086				
using O5 and O6	141				
LUT as Memory	45	0	0	17400	0.26
LUT as Distributed RAM	44	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	44				
LUT as Shift Register	1	0			
using O5 output only	1				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	448	0	0	106400	0.42
Register driven from within the Slice	291				
Register driven from outside the Slice	157				
LUT in front of the register is unused	53				
LUT in front of the register is used	104				
Unique Control Sets	18		0	13300	0.14

Table 3: FPGA Resource Utilization Report from post$_p lace_u tilization.rpt$

6.4 Critical Path

- **Critical Path Length:** 11.615ns

6.5 Best Design Point

- **CPI:** 1.32
- **Clock Frequency:** 105.0
- **Figure of Merit (FOM):** 61.36
- **Time per Program (mmult):** [Use Iron's Law: T = Instructions × CPI × Clock Period]

6.6 Critical Path Explanation

The current existing critical path involves MEM to MEM in the execution stage. It then passes through the InstructionExecuteForCL (The register storing the instruction in execution stage),

before handing back to the IMEM. We have optimized the critical path such that there is minimum LUTs, and reducing the routing delay (Our routing delay is ~ 6.5ns). However, we are unable to reduce the critical path further to push the FMAX. One failed attempt was to reduce bit length of the instruction going into the execution stage. However, during implementation, that increased the critical path's slack, with a decreased cost in area. Ultimately, we settled for the final design as shown in the report.

6.7 Optimizations Considered

- **Removal of Forwarding**: Removed forwarding of the CPU. Allowed a higher max frequency, from 60 MHz to 85 MHz.
- **Naive Branch Prediction**: Naively guess branch no take for any branch prediction. Improved the CPI from 1.4 to 1.32 for the MMULT program.
- **Optimization of Signal Bits**: Moved around the signal control bits. Since our critical path is in the execution stage, moving the signal bits allowed us to increase the FMax additionally by 20 MHz, to 105 MHz.

7 Results and Evaluation

Metric	Value	Target	Notes
Max Frequency	105 MHz	70 MHz	Great
CPI (best-case)	1.28	≤ 1.3	Good
Field of Merit	61.36	≥ 60	Good

Table 4: Performance summary after synthesis and timing closure.

8 Conclusion

8.1 Reflections

Overall, we were able to present a high-performing 4-stage RISC-V CPU that has optimal FOM and metrics. However, we acknowledge that we spent much more time debugging than others, leading to all-nighters at times and McDonalds consumption at 3am. We found Checkpoint 2 and 3 fairly difficult due to the lack of clarity (i.e. hints / tutorials etc) as compared to Checkpoint 1, which was comparatively straightforward to follow. We believe the sections for memory-mapped I/O, CSRW, and UART were really complicated to understand.

Our worst bug occured right around two weeks before the deadline of the project, taking us 3 full days to debug. We did not have a working script for Checkpoint 3 yet, and it made us doubt our

design and even consider rebuilding our entire CPU. Ultimately, this issue was resolved through meticulous line-by-line code review and waveform inspection.

To summarize, we believe that this final project was a very hard and rewarding project and more instructions and guidance on the (very clearly written) spec would have been helpful. Although there were times that we were struggling hard, looking back we were very happy with what we accomplished, and we wish more could have been added to the project, such as implementing the MIC interface or Button Parser to communicate with our Datapath.

8.2 What we would have done differently

If we were to approach this project again, we would first invest more time in thoroughly understanding the datapath before writing any Verilog RTL. Taking a "learn-as-you-go" approach led to significant debugging overhead that could have been avoided with a stronger upfront foundation. We also would have coordinated our work schedules more closely with the professor's and lab TAs' office hours, allowing us to resolve questions and roadblocks more efficiently as they arose.

We might have also consider doing a 5 stage pipeline, with branch prediction enabled. For a 4 stage CPU, we did not believe implementing a branch prediction was worth it. However, that would significantly decrease the stalling times in a 5 stage. (2 instead of 1). We believe we would be able to achieve a higher FMax (at least 115 MHz) with lower CPI (at most 1.25) with that design, albeit with potential area increases. However, we did not want to change the design late into the project. Instead, we optimized as much as possible to reduce the critical path delay.
Another improvement would be to start Checkpoints 2 and 3 significantly earlier—ideally more than a week before the deadline—to give ourselves adequate time to iterate and validate our design. We also would have maintained a continuously updated datapath block diagram. Relying solely on reading RTL source code for debugging proved inefficient; a clear, evolving visual reference would have streamlined our understanding and accelerated the bug-fixing process.

Most importantly, we would adopt a test-driven development approach. In hindsight, one of our most critical bugs could have been avoided entirely had we written custom testbenches for verifying IMEM read and write operations. As Verilog RTL designs quickly grow in complexity, incrementally developing and validating features with corresponding testbenches becomes essential. This strategy would have helped us localize bugs more effectively and ensure correctness throughout development.

8.3 Workload Distribution

The workload is distributed fairly. Both members contributed significant workload to the project.