EECS151 Final Report

Leonard Wei, Ryan Ma

December 2022

Contents

1 Project Functional Description and Design Requirements 3
1.1 3-Stage Pipeline . . . 3
1.2 Memory Architecture . . . 3
1.2.1 BIOS . . . 4
1.2.2 Instruction/Data Memory . . . 4
1.2.3 Memory-Mapped IO . . . 4
1.3 Branch History Table . . . 5
1.3.1 BHT as a Direct Mapped Cache . . . 5
1.3.2 Saturating Counter . . . 6

2 High-level organization 7
2.1 Fetch-Decode Stage . . . 7
2.2 Execute Stage . . . 7
2.3 Memory-Writeback Stage . . . 8

3 Detailed Description of Sub-pieces 8
3.1 Control Logic . . . 8
3.2 Forwarding Logic . . . 8
3.3 Branch Comparator . . . 8
3.4 ALU . . . 9
3.5 Store Load Modules . . . 9
3.6 Immediate Generator . . . 9
3.7 Program Counter and Instruction Fetch . . . 9
3.8 Memory Partioning . . . 9

4 Status and Results 10
4.1 Optimizations . . . 10
4.1.1 Jal & Jalr Optimization . . . 10
4.1.2 Branch Predictor Optimization . . . 10
4.2 Critical Path . . . 10
4.3 Clock Speed . . . 11
4.4 CPI . . . 11
4.5 Area Utilization . . . 12

5 Conclusions 13

1 Project Functional Description and Design Requirements

The purpose of this project was to create and design the digital logic for a pipelined RISC-V CPU running the RV32I ISA. The design is then to be loaded onto the Xilinx PYNQ-Z1 FPGA. The processor is also equipped with a set of memory-mapped IO that includes three major different memories: BIOS Memory, Data Memory, and Instruction Memory. By interacting with these different memories, users should be able to upload instructions via the memory-mapped UART and execute instructions on the CPU.

The goal of the CPU design was to minimize the overall *Iron Law* of CPU performance by minimizing CPI and maximizing operating frequency.

$$\frac{\textbf{Time}}{\textbf{Program}} = \frac{\textbf{Instructions}}{\textbf{Program}} \cdot \frac{\textbf{Cycles}}{\textbf{Instruction}} \cdot \frac{\textbf{Time}}{\textbf{Cycle}}$$

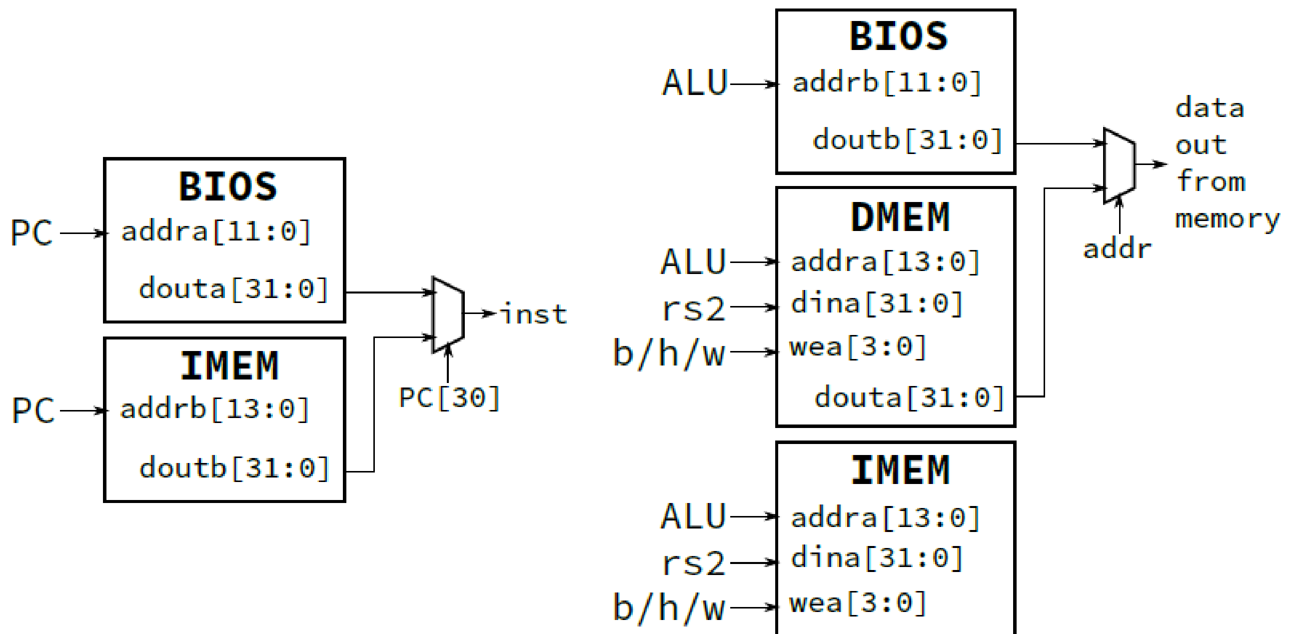
We can benchmark these metrics by running mmult.c, which performs matrix multiplications, to see our CPI. With tools like Vivado and these metrics, we set out to optimize our design with a target CPI of less than 1.2.

1.1 3-Stage Pipeline

The original design requirements specified we needed to create a three-stage pipelined CPU, so our final design reflects that idea. By placing pipelined registers throughout our datapath, we can reduce the overall CPI of our processor by a significant amount. However, a pipelined CPU will run into data and control hazards, so a critical part of our design was to find where to place the pipelines and add forwarding logic/stalls as needed to avoid hazards. By reducing hazards, we can maximize the number of instructions running in the CPU and minimize the CPI.
We also needed to consider the critical path of the CPU. When adding registers and forwarding logic, we may need to increase the number of components that an instruction needs to run through, thus increasing the critical path. We also needed to find the most efficient ways to place these components so that the critical path is reduced and the operating frequency is increased.

1.2 Memory Architecture

The CPU has three major sections of memory: BIOS Memory, Instruction Memory, and Data Memory. The memories are technically all block RAMs located on the FPGA, so by using different address types and ports, we separate the address space of the memory into the different memory types. A rough port list is shown below.



1.2.1 BIOS

The BIOS is meant as a simple way for the user to interact with the CPU. User programs are sent through the UART and loaded into the BIOS memory. To start the program, the user will JAL to the base address of the loaded program in the BIOS memory. BIOS memory access from the CPU is read-only for PC and Data addresses where the top four bits equal 4'b0100.

1.2.2 Instruction/Data Memory

The instruction and data memory are technically one large chunk of memory. These will store the instructions and data during the execution of the CPU. PC addresses with the top four bits equaling 4'b0001 will have read-only access. Data addresses with the top 4 bits equaling 4'b001x (where PC[30] == 1'b1) will have write-only access.

1.2.3 Memory-Mapped IO

The memory-mapped IO is meant for various user interactions. Data addresses with the top four bits equaling 4'b1000 are for memory-mapped IO read and writes. Functions for UART signals including the ready-valid handshake and UART in/out run on the memory-mapped IO. Counters for performance evaluation (cycle counter, instruction counter, reset, branch counters) are also located in the memory-mapped IO. The addresses for memory-mapped IO are shown below.

Table 3: I/O Memory Map

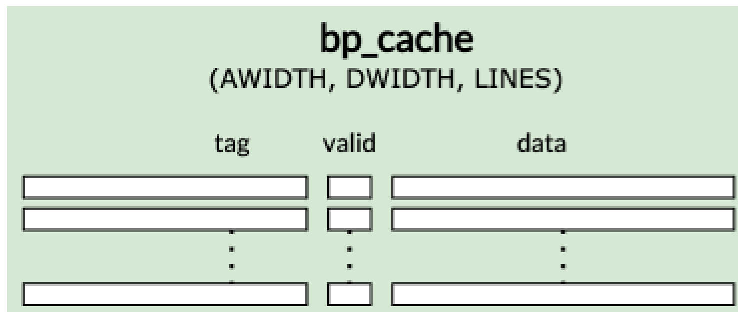
Address	Function	Access	Data Encoding
32'h80000000	UART control	Read	{30'b0, uart_rx_data_out_valid, uart_tx_data_in_ready}
32'h80000004	UART receiver data	Read	{24'b0, uart_rx_data_out}
32'h80000008	UART transmitter data	Write	{24'b0, uart_tx_data_in}
32'h80000010	Cycle counter	Read	Clock cycles elapsed
32'h80000014	Instruction counter	Read	Number of instructions executed
32'h80000018	Reset counters to 0	Write	N/A
32'h8000001c	Total branch instruction counter	Read	Number of branch instructions encounted (Checkpoint 3)
32'h80000020	Correct branch prediction counter	Read	Number of branches successfully predicted (Checkpoint 3)

1.3 Branch History Table

One important part of our processor that will help speed up branching instructions is a branch predictor. Our implementation of the branch predictor involves the use of a branch history table (BHT) that records if a branch to a particular address has been taken in the past. As more of the same branch is taken, there is more confidence in predicting the branch should be taken. However, if the branch is not taken multiple times, there is more confidence that the branch should not be taken. The BHT's job is to hold the branching history to determine how we should predict the next branch. After the branch has been evaluated, the counter value is updated to taken/not taken depending on if the branch was *actually* taken or not so that future predictions will follow a general trend.

1.3.1 BHT as a Direct Mapped Cache

As per the design specifications, we created our BHT as a direct mapped cache. The benefits of this cache is the better hit time and least difficult tag-checking complexity, trading off for worse hit rate (given a larger cache) and more difficult replacement/overwrites. Each "data" segment of each cache entry held the value from a saturating counter that would count how many of that particular branch has been taken. Our cache structure is displayed below:

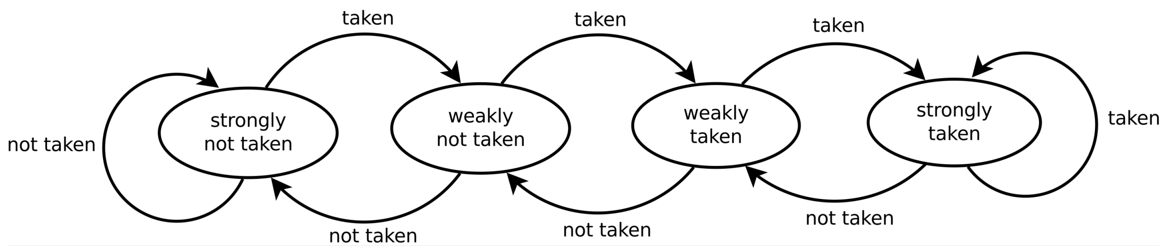


One distinction we made was instead of creating a 2D reg that holds all the data, we created separate tag, data, and valid regs so that the process of invalidating the cache and evictions had less logic to process.

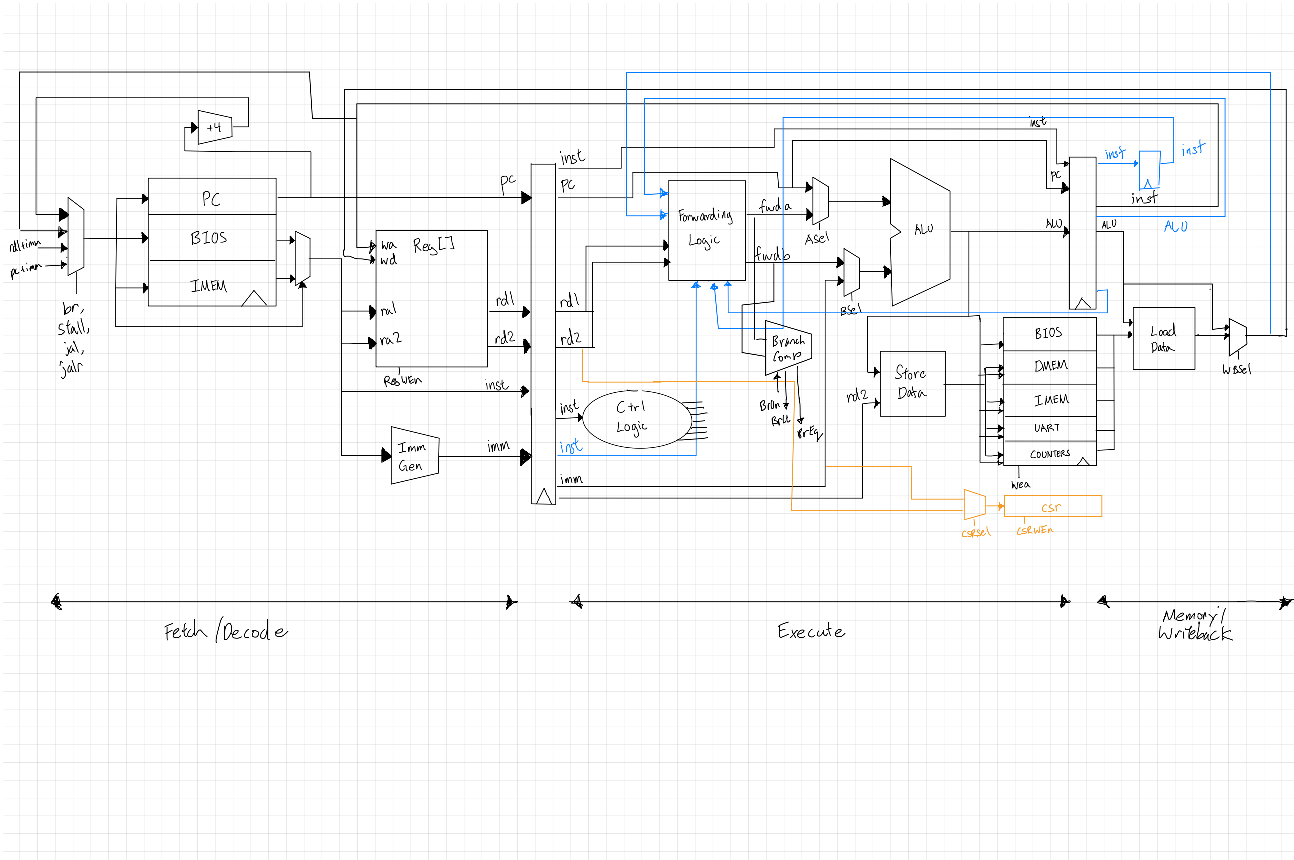
```
reg [TWIDTH-1:0] tag [LINES-1:0];
reg [DWIDTH-1:0] data [LINES-1:0];
reg [LINES-1:0] is_valid;
```

1.3.2 Saturating Counter

The saturating counter was helpful here as it helped determine a relative confidence of our branch prediction. Using a two-bit wide counter, we would increment the counter if the branch was taken and decrement the counter if the branch was not taken. By only selecting the MSB of the data as our prediction, we can follow a system of relative confidence, as displayed below:



2 High-level organization



Our CPU is a 3-stage pipelined processor where it broken up into Fetch-Decode, Execute, and Memory-Writeback stages. We chose a FD/EX/MW pipeline because the timing would fit well with the synchronous read-write memories as well as shorten the critical path.

2.1 Fetch-Decode Stage

The Fetch-Decode stage is comprised of the Program Counter (PC), BIOS memory, Instruction Memory, Register File, and the Immediate Generator. We decided to include the Immediate Generator in this stage because we could use it for our JAL optimization where the jump address could be immediately computed with the PC address and the immediate. We also chose to include the Regfile because because it was has asynchronous read but would greatly increase the critical path if we opted for a F/DX/MW pipeline.

2.2 Execute Stage

The Execute stage is comprised of the ALU, Branch Compactor, Forwarding Logic, Control Logic, and Store modules. The Control Logic module lives here because it's signal is used exclusively by

the execute and memory-writeback stage. It also have the standard components for the Execute stage and does not differ much from a 5-stage pipeline. We had execute be it's own stage because we expected it to have our longest critical path.

2.3 Memory-Writeback Stage

The Memory-Writeback stage is comprised of all the memory partition, UART, Counters, and Load Modules. We merged the Memory and Writeback stages of a 5-stage pipeline because memory was synchronous read and write. We could take advantage off the clocked signal and add supporting pipeline registers to form the Execute and Memory-Writeback partition.

3 Detailed Description of Sub-pieces

3.1 Control Logic

We opted to use a combinatorial control logic module instead of ROM style module similar to the one that CS61C's RISCV Logisim project would use. Using logic gates instead of ROM would decrease our overall size but could increase our critical path length. However, the increase to the path was minimal because each control signals was driven by a couple logic gates.

3.2 Forwarding Logic

The Forwarding Logic Modules takes in several input to produce a "rd1" and "rd2" for the Branch Comparator and ALU to use. The "rd1" and "rd2" could from the fetch-decode pipeline registers for rd1 and rd2, memory-writeback pipeline register containing the ALU output, or a past-previous register that saved the writeback data. We needed to added this "past-previous" register to address the two-cycle ALU-to-ALU hazard by more of the pipeline history.
We implemented the Forwarding Module by abstracting the inputs into current, previous, and past-previous pairs of instructions and data. We employed combinatorial logic to compare the opcodes and rd, rs1, and rs2 addresses to compute which data to route out out the module. We could have used a ROM to implement the Forwarding Module to reduce the critical path but development would but much more difficult.

3.3 Branch Comparator

The Branch Compactor is quite a standard implementation where if makes equal and less than comparisons and outputs it to a BrEq and BrLt. It also will used unsigned values if notified by the BrUn signal.

3.4 ALU

The ALU is also standard implementation where an ALUSel signal is used to pick the desired operation.

3.5 Store Load Modules

Since the store and load involved quite logic so we opted to modularize them. The modules aim to abstract the data formatting for the store and load instructions (i.e. perform the bit operation to derive a halfword from a word). Each module takes in an address and the data to be stored or loaded. The last 2 bits are extracted to identify the correct bit operations to derive the correct format for the store or load.

3.6 Immediate Generator

The immediate generator consists of a standard implementation where the immediate is arranged from the input instruction and sign extended to 32 bits. The only special feature this module has is that is package along with its control signal. This is to allow for this the easy integration of the module in its spot in the fetch-decode stage.

3.7 Program Counter and Instruction Fetch

Since design involves the PC and memory to be clocked at the same time, there was some difficulty implement this PC update with the instruction fetch. If the PC was feed into the memory, we would require 2 clocks to fetch our instruction—something we don't want. Instead, we would have to compute the instruction address before the clock. We achieved by created some combinatorial logic that would compute the next PC address to and feed them both to the PC and memory. This resolved our timing issues for the Instruction Fetch.

3.8 Memory Partioning

To implement memory partitioning, we used the 4 MSB bits of the memory address and several logical operations to create a selection signal for a mux attached to the output of the required memory modules. There was also a special case for IMEM where the second MSB bit PC address needed to be 1 in order to initiate a memory write.

4 Status and Results

4.1 Optimizations

4.1.1 Jal & Jalr Optimization

One of the major optimizations that we attempted was the optimization for JAL. By cutting down the number of cycles that a JAL operation used to just one cycle (by feeding output directly from the Fetch/Decode stage, JAL instructions can be executed much earlier in the pipeline. However, the main issue we ran into was because JAL would execute so fast, it could potentially execute out of order, faster than instructions that are already in the pipeline. We would then have to invalidate those instructions, which is another trade-off in speed especially in situations where there are not as many JAL instructions. This problem was exacerbated by having consecutive JAL, JALR, or branch instructions in the pipeline as all of those operations would potentially be slowed or return the incorrect result. Solutions to this problem might have included reworking the datapath/pipeline or inserting bubbles around JAL to prevent this problem.

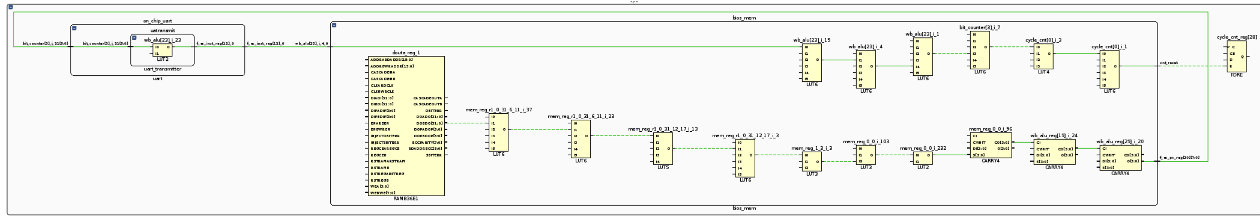
4.1.2 Branch Predictor Optimization

While the branch predictor provided by the spec is relatively complicated, it is not always the better solution. For larger caches (128 lines by default), the issue is that the cache is at first very cold with many compulsory misses, meaning that initial predicts were more likely to be wrong, and for a small number of branches, a correct prediction is even harder to get.

One optimization is possibly to turn the branch predictor into a dynamic predictor: if the branch moves forward, predict branch taken. If backwards, predict not taken. This would, in effect, be more helpful for simple looping instructions as it bases the prediction off the current program execution's direction. However, this optimization did not yield much of an improvement for us. In fact, we got slightly worse CPI (1.34 vs 1.32) with this optimization. The cause of it is the way we handle flushes when there are multiple branches together. Since we are flushing the other instructions, it seems that the improvements made by this optimization were minimal, if any at all.

4.2 Critical Path

When calculating the timing of the critical path, we found that at 50 MHz, we had a considerable amount of hold-slack on the Max-Delay path, running from DMEM → BIOS MEM → IMEM. After some consideration, this critical path was because of our forwarding logic that stemmed from forwarding logic from the writeback stage that fed into our cycle counter, as seen below.
Reworking this forwarding path would have taken more time than we had (as it would involve reworking the datapath for ALU forwarding), so we decided to leave it as it is. We could have also found a different, though possibly more complicated, method to count cycles/instructions that might have helped us reach 60 MHz on this design. At 50 MHz, our Max Delay and Min Delay timings are 1.085ns and 0.094ns respectively.



Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	1.085 ns	0.000 ns	0	3723
Hold	0.094 ns	0.000 ns	0	3723
Pulse Width	8.750 ns	0.000 ns	0	911

4.3 Clock Speed

Given the formula to calculate the clock speed:

$$\mathbf{CLK} = 125\text{MHz} \cdot \frac{\mathbf{CLKFBOUT_MULT}}{\mathbf{CLKOUT_DIVIDE} \cdot 5}$$

We first used the values 34/17 to test 50 MHz for our design. While this value passes, we noticed that the critical path was quite large and could probably be stressed a little more. We then tried 36/15 to get 60 MHz, which yielded a hold time slack of -0.012ns. While this is still very close, it is still a hold time violation, so we decided to back down to 50 MHz. The only way to reduce the critical path as mentioned in Section 4.2 was to redo our forwarding logic. We were limited on time, so while we knew what needed to be done, we did not have time to actively explore this option.

It is also worth noting that for Checkpoint 2, we were able to pass timing for up to 65 MHz. It seems like that the branch predictor adds a considerable level of complexity to the system (due to the extra wiring required to use the cache, and then go back and check if the guess is correct). With the branch predictor in, we were not able to pass even 60 MHz.

4.4 CPI

Our CPI for the implementation with the branch predictor is as follows:

$$\mathbf{CPI} = \frac{17693358}{12894957} = 1.372$$

Our CPI for the implementation with the branch predictor is as follows:

$$\mathbf{CPI} = \frac{17075697}{12894957} = 1.324$$

```
Mate Terminal
File Edit View Search Terminal Help

151> jal 10000000
Result: 0001f800
Cycle Count: 01048df1
Instruction Count: 00c4c2ed
Branch Instruction Count: 005011bf
Correct Branch Prediction Count: 00179b7d

151> jal 10000000
Result: 00001f800
Cycle Count: 010d42ee
Instruction Count: 00c4c2ed
Branch Instruction Count: 0005c5040
Correct Branch Prediction Count: 000000000

151>
```

While the improvement seen here is small, there is still a noticeable improvement overall. We can see that with a more flushed out implementation (fixing the flaws in our forwarding logic and improve jump/branch instruction handling) could yield even greater of an improvement.

4.5 Area Utilization

The resource utilization for our CPU is as follows:

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	Bonded IOB (125)	ILOGIC (125)	OLOGIC (125)	BUFGCTRL (32)	PLLE2_ADV (4)
z1top	1748	667	82	16	552	1604	144	34	13	1	1	2	1
bp (button_parser)	15	29	0	0	12	15	0	0	0	0	0	0	0
clk_gen (clocks)	0	0	0	0	0	0	0	0	0	0	0	2	1
cpu (cpu)	1733	638	82	16	540	1589	144	34	0	0	0	0	0

One interesting (but trivial) observation from this is that with the branch predictor implemented, there were considerably more LUTS and MUXes in use versus without the branch predictor implementation. It makes sense considering the extra logic required for prediction.

5 Conclusions

Even though our processor did not have the target CPI of 1, we believe it was still overall successful, and more importantly, we got to experience the process of digital logic design from the ground up. From the intial design diagram planning stage to the development stage to the final polishing steps, each part brought us different challenges and taught us what to do (and not to do).

For a more complicated chip design, our overall workflow was acceptable, but it could have been much better. We spend adaquete time at the beginning planning our block diagram. However, we were a little too confident in our design which led to future pitfalls in trying to fix and optimize certain instructions (like JAL). We did a good job of parallelizing our work: one person made the modules while the other person wrote testbenches for them. However, when it came to testing and debugging issues, we had more trouble working together as one person's bug would often bottleneck the other person's work. Working on optimizing while another person debugged (or even tackling a different area of the project) might have been a more efficient workflow, especially as the project neared completion.

Another way we could have dealt with our initial design flaws was to write more detailed *integration* testbenches. While we had thorough unit tests for each module, we mostly relied on the provided integration tests to test our implementation. If we had spent more time writing integration tests that spelled out potential timing/pipeline flaws in our design, we could have tackled those issues much earlier on in the design process.

Working with synchronous and combinational logic was also a tough part of the project. Because there were so many moving parts, understanding the timing requirements of different pieces of logic was critical. When do you want a signal to arrive before the clock? When do you want a signal to arrive after the clock? When is it ok to run logic combinationally? Wiring said logic proved to be more of an issue than previously mentioned. However, this was something that testbenches exposed on a modular level. Debugging on the integration-scale with custom testbenches can still be improved.

Overall, even if we did not reach our target CPI, we still were able to explore many facets of optimizations for the critical path, branch predictor, and datapath/pipeline that taught us many things about digital logic design. Working on an extensive, complex project was a thorough learning experience for both of us and a huge leap into the world of digital logic design.