EECS 151 Final Project Report

FPGA Lab

Adyant Khanna and Russel Sofia

December 12, 2022

RISCV151

Table of Contents

1 Project Description and Design Requirements

This project requires us to use Verilog to a fully functional CPU by creating module's inside it from scratch and writing our own tests to ensure that everything works properly. On top of that, we have requirements that we have to pass and so we have to ensure that our implementation follows all the guidelines and passes all the requirements by the end of the project.

1.1 Project Functional Description

We designed a pipelined RISC-V CPU which is capable of the entire RISC-V instruction set with memory mapped I/O and branch prediction modules. We have to determine where to place pipeline registers and how to fix hazards that come with pipelining. On top of that, we create different modules separately and create testbenches specifically to test each module's functionalities. Initially, our goal was to make sure that our CPI (cycles per instruction) is below 1.2, and at the end we minimize time per program by the use of iron law. Since we have to improve on a program called mmult.c, we have a fixed number of instructions per program and all we can improve on is decreasing the cycles per instruction (CPI) and time per cycle (period, i.e. 1/frequency).

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

1.2 Designing a 3-stage Pipeline

Our design needs to minimize the difficulty of handling Data and Control Hazards while also maximizing performance. Since we know that BIOS, IMEM, DMEM, UART, Instruction Counter and Cycle Counter are all sequential modules, we place our pipeline registers in parallel with these modules to ensure that we can reduce the number of cycles per instruction (CPI). These pipelined registers also need to be placed such that performance would be maximized. Since frequency is inversely proportional to the maximum critical path, before forwarding we have:

$$clock\,rate \text{ pipeline } (\mathrm{f}_{\text{pipeline}}) = \frac{1}{\max(cp_{IF}, cp_{ID}, cp_{EX}, cp_{MEM}, cp_{WB})}$$

This means that we want the critical paths of our CPU to be as low as possible for maximum performance/clock frequency. Since all RAMs for BIOS, IMEM and DMEM are synchronous read and synchronous write this means that it essentially can act like a register and so we can place pipeline registers in parallel with the RAMs (BIOS RAM, IMEM RAM, DMEM RAM). We can't however have a pipeline register be in parallel with the RegFile and this is because the

RegFile has an asynchronous read and synchronous write. Our final design can be seen on section 2.1 CPU Block Diagram.
Our three stages are WF/D/X since memory is essentially a sequential element that is not counted as its own stage. Essentially, we treat BIOS, IMEM, DMEM and UART modules as registers in our pipeline because they synchronously read and synchronously write. Thus, we place pipeline registers in parallel with RAM modules which means that we eliminate one of the stages, namely the MEM stage.

# of stages = (# of positions of pipeline regs + 1) - # of eliminated stages
# of stages = (3 + 1) - 1 = 3 stages
Hence, our 3-stage pipeline is WF/D/X

Why is WF together? Remember that RISCV151 implementation has a RegFile that is asynchronous read & synchronous write. This means that we can write to RegFile while fetching instructions IF simultaneously, thus WF! This means that we are writing back and fetching instructions at the same clock edge!

If it is confusing, just think that RegFile acts like a register when we are writing data to it (synchronous write) and it acts like a combinational logic when reading data from it (asynchronous read). Furthermore, we can treat RegFile similar to IMEM where we divide the RegFile into two parts for clarity (they are still an instance of the same module).

WB + IF	D
RegFile (WB)	-
PC Reg (IF)	PC Reg (D)
BIOS or IMEM (IF)	RegFile (D)

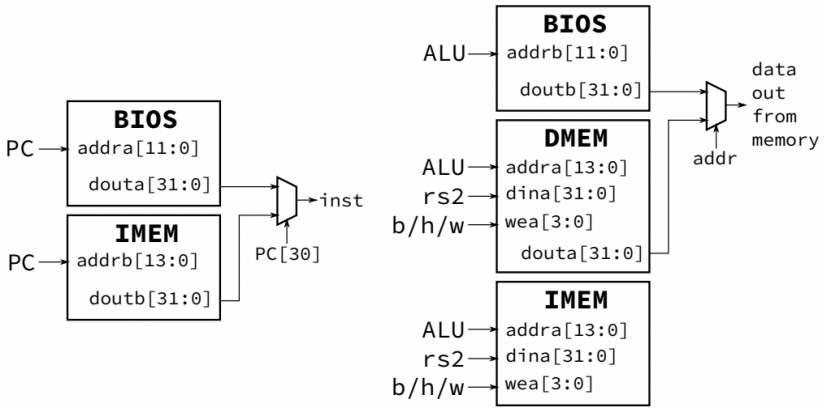
1.3 Memory Hierarchy

The RISCV151 Memory Architecture includes three distinct memories:

1. IMEM (Instruction Memory)
   IMEM is responsible for storing an instance of a running program where each 14-bit address stores a 32-bit instruction of the RISC-V ISA.
2. DMEM (Data Memory)
   DMEM is responsible for storing values into the address in memory.

3. BIOS (Basic Input/Output System)
   The BIOS consists of a BIOS ROM that is responsible for storing the BIOS Program and this serves as the base program that is run by the FPGA at the startup stage. On top of that, zero out other memories during startup.

   The BIOS Program then receives user programs that come from the UART module and load them into the instruction Memory. An important note is that IMEM and DMEM are part of the same instance of memory. This means that even though they are located in different part of the datapath, they still serve the same purpose and are combined in one module definitionFor the purposes of clarity, the I/O ports of IMEM and DMEM are separated as follows:



1.4 Memory Mapped I/O (UART)

The UART is the driving force within the CPU that allows for outside instructions to be stored into memory in our CPU. It works with the BIOS as the BIOS program receives the user programs over the UART and the UART is able to store these instructions into instruction memory. Thus our CPU interacts with the UART by being able to send and receive bytes which will be in the form of instructions and data outputs. The following I/O memory map from the spec explains how the program should interact with the UART based on the PC addresses.

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)

In our program, we should check when there is a store or load instruction as store will be for the transmitter data address and load will be for the receiver data and receiver control address. Based on this functionality and the fact that our CPU has a pipeline register during the MEM stage which includes the UART, we must check the instruction before and after the pipeline to know the transmitter data address and receiver data and control addresses respectively.

2 High Level Organization

The first thing we did was designing our RISC-V datapath and ensuring that we can modularize them into different modules to help us implement and debug each module separately. We split each mux into their own module. Individual CPU elements (n-input MUX, RegFile, Branch Comparator, Imm Gen, ALU, Adder, LDX, etc) and control logic for each stage (WF, D, X) are modularized so that we only have to worry about each stage separately and decode each instruction from the instruction pipeline registers. For each feature we add, we would update it to our datapath while ensuring that all the other features that were implemented before will still work. Once we were happy with our design, we started implementing each module in Verilog.

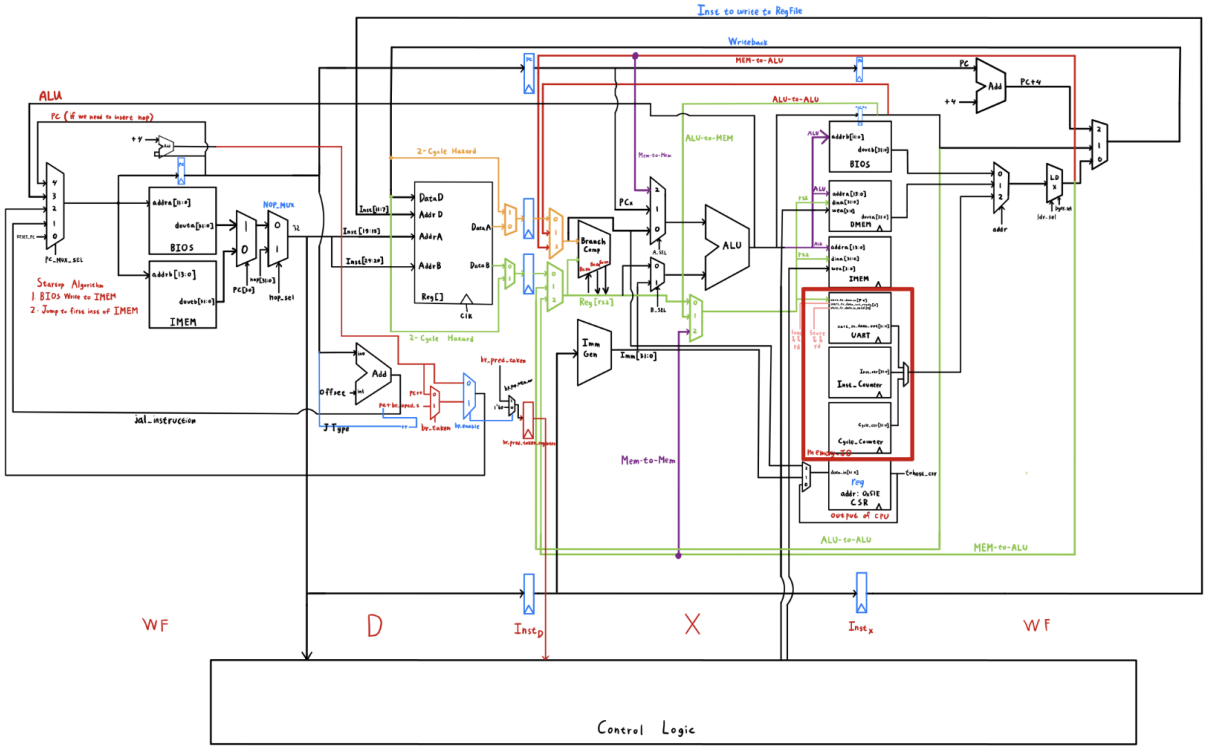
2.1 CPU Block Diagram

As mentioned in 1.2 Designing a 3-stage Pipeline, our write-back (WB) and instruction fetch (IF) runs in parallel and so we call it (WF) stage. Again, this is due to the fact that BIOS, IMEM, DMEM, UART, Instruction Counter and Cycle counter are sequential elements which means that they can be treated similar to pipeline registers. Placing pipeline registers in parallel with these modules is best. The only memory module we can't do this with is the RegFile.

After deciding the placement of the pipeline registers, we fix Data Hazards and Control Hazards. We resolve all data hazards using forwarding where we have different hazards that need to be resolved using forwarding. Control hazard is something that we initially just solve by NOP where we just inject a NOP instruction (addi x0, x0, 0) whenever we predict the prediction wrong. Since we alway produce the branch to not be taken, this means that we have

to inject a NOP instruction every time a branch is taken. In the later stages we implemented a branch predictor to further reduce cycles per instruction (CPI), see section 3.2 Branching.

A final thing we consider is that when we introduce forwarding, our critical path changes from just a combinational delay between a stage into a combinational delay between multiple stages. This means that we have to ensure that our forwarding will not influence our critical path drastically and upon implementation we changed some of the implementation of our modules to further reduce critical path and increase frequency, see section 4.1 Hardware Optimizations.



2.2 Submodules

We created sub mobiles that are located in different parts of our Verilog files. This allows us to write testbenches for each sub-block in a very controlled way while also ensuring that we can use abstraction when implementing the full CPU using these submodules. Additionally, we can hierarchically divide everything within each file to ensure that we can incrementally add features while easily writing testbenches for each level.

2.2.1 Basic Modules

1. MUXes
   Defined generic modules for 2, 4, and 8-input muxes so that we can just set the inputs accordingly and don't have to define it on cpu.v everytime we need a mux. Instead, we will just call the mux module we need and set the inputs accordingly (say we want to make a 3-input mux, just set in0, in1, in2 to the corresponding wires and set in3 to 0).
2. Adder
   Defined a 2-input adder that sums the two different inputs and drives the output of the module to that sum.
3. ALU
   Defined an ALU using a case statement where each option implements the basic arithmetic operations and well as bitwise operations.
4. Branch Comparator
   Defined a comparator that compares the two registers during a branch instruction and outputs 1-bit br_eq and br_lt signals. Additionally, it can handle both signed and unsigned comparisons.
5. Imm Gen
   This module parses the instruction and generates the immediate of the given instruction based on the Opcode of each instruction.
6. LDX
   This module accepts an ldx_sel input which determines which type of load we are using (LW, LH, LB) while also differentiating signed and unsigned comparisons. The lower 2 bits of the output of the ALU determine the offset of the load instruction with respect to the data being read.

2.2.2 Control Logic & Forwarding

We modularize control logic so that we can focus only on each stage. We created three separate modules for each WF/D/X stage so that we can only worry about the control logic of individual stages. This means that our implementation abstracts each stage and therefore each stage will assume that the other stages have done all they have to process and forward any data required by the other stages. This means that our control logic is very readable and when adding a new feature to a particular stage, we just have to modify the control logic for that stage only. For forwarding, we added inputs to the muxes at the beginning of the stages to handle different data hazards.

2.2.3 Wires, Pipeline Registers and MMIO

All the modules are connected together in cpu.v, connecting each individual module from basic module, MMIO, and pipeline registers as necessary. This allows us to set the inputs to each of the Control Logic modules in order for the control logic to send back the output to drive each of the muxes in our datapath. Our interaction with MMIO also works similarly to control logic in a sense that we just have to call our memory modules in cpu.v and set the control signals (by calling the control logic modules) accordingly which will enable us to fully utilize all parts of memory in a singular file.

2.2.4 Branch Prediction Cache and Saturating Counter

We implemented a direct mapped cache and a saturating counter which is used to store addresses of branch instruction so that we can use it for future prediction. The saturating counter acts as a double measure where a single wrong guess will not change the stage, only two wrong guesses in a row for a particular branch instruction will change the prediction scheme. We use a 2-bit saturating counter and the MSB of the counter to indicate if the branch is predicted to be not taken (0) or if the branch is predicted to be taken (1), see section 3.2 Branching.

3 Detailed Description of Sub-pieces

As we implemented our CPU from our design and added improvements such as branch predictor, these additions added more pieces into our overall implementation. Section 2 gives a high level overview of the functionality of our overall CPU such as the stages, how our logic works, and more. This section will dive deeper into certain modules we added that were design decisions as well as other important functionalities to our CPU.

3.1 Control Logic

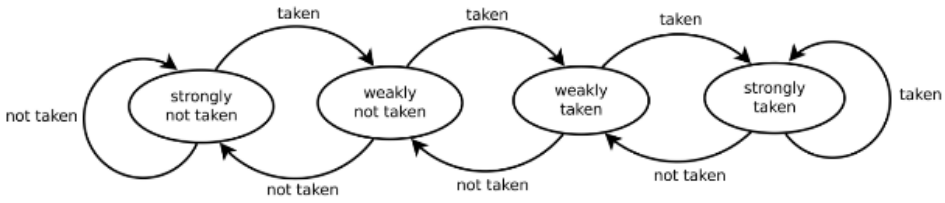
Control logic provided the backbone for our 3-stage pipeline as it handled the selects of all of our MUXs, ALU, forwarding logic, and more. The basic functionality of the control logic provides data for the select values of MUXs, ALU, as well as enabling write enable for the reg file. These are all determined based on the instruction that is being parsed based on the specific stage it is in. Each stage (WF/D/X) had its own module for control logic. However, when adding forwarding to our CPU, this added another degree of complexity and functionality to our control logic. Now we had to determine different hazards we would run into, including ALU-to-MEM, MEM-to-ALU, MEM-to-MEM, and 2-cycle hazards. To handle these, our control logic modules would now need to parse more instructions and have additional logic. For

example, for ALU-to-ALU, our execute module would not only take in the instruction in the execute stage but also fetch the instruction from the WF stage (as it is stored in an instruction register). We would then determine whether the rd (not x0) of the wf instruction is equivalent to either rs1 or rs2 of the execute instruction and also check what kind of instruction we have in the execute stage. The type of instruction would allow us to determine what kind of hazard we have. Similarly, to handle other hazards, our control logic had to have expanded functionality to handle these cases. The execute control logic had the most additions as it has to handle the most forwarding cases. Additionally, with the addition of forwarding, the necessary MUXs, registers, and wires were added to handle each case.

3.2 Branching

Our branching implementation comes from the checkpoint 3 stage. There are two main components: the saturating counter and the branch prediction table implemented as a cache.

The saturating counter is a state machine that changes based on whether we take a branch or we do not take a branch. If a branch is taken, the saturating counter will increase and in the case a branch is not taken, the saturating counter will decrease. This saturating counter contributes to our prediction scheme as we utilize the most significant bit of the saturating counter to determine if we take a branch or not.



For our cache, we handle two stages: the decode stage which entails the “guessing” stage and in the execute stage which is the “checking” stage. In the “guessing” stage, the cache is passed in an address and it goes into the cache and checks the valid and tag bits of the address passed in. If the entry is valid and the tag matches with the address then we have a cache hit. With the cache hit, the top bit of the entry is used as the branch prediction.

On the other hand, the “checking” stage has two purposes. It first fetches the same branch instruction address that was in the decode stage and checks that entry into the cache (checking the valid and tag bits). If there is a hit then the “checking” stage ensures the saturating counter is updated based on whether the branch is actually taken or not and stored at that address in the

cache. Then this counter value is written back to the cache to ensure the cache is updated with the most recent prediction. On the other hand, if we had a cache miss, the "checking" stage then goes into the cache and creates a new entry in the tag, valid, and data arrays we have created in our cache implementation. The values stored here are based on the updated saturating counter value based on whether the branch is taken.

3.3 NOPs

For our 3-stage pipeline, there were 2 cases in which we had to flush our pipeline. The first case was with JALR. Since we evaluated the result of our JALR instruction (i.e, where we jump to) in our execute stage, we knew that we would have to flush whatever instruction was currently in our decode stage as there would be a low likelihood that the JALR instruction was jumping to PC+4. We added more logic to our CPU here by determining if a JALR instruction was in the execute stage, and if it was a JALR instruction, we would trigger our NOP MUX to 1 to pass in a add x0, x0, x0 instruction into our decode stage which would not do anything to our cpu, and our program could successfully continue. Additionally, we would inform our WF stage, which controls the PC select MUX, to jump to the address from the output of the ALU.

Similarly, branching would also require us to NOP the decode stage in any case where we predict the branch instruction wrong. With the addition of the checkpoint 3 branch predictor, we added logic to determine when we predicted correctly and when we did not predict correctly. Thus we would set the NOP MUX to 1 if the prediction was wrong and inform our WF stage to jump to the PC outputted by our ALU.

3.4 JAL and JALR

Our Jal and JALR instructions had slightly different implementations. With JAL, we added hardware in the decode stage which was an adder taking in the PC and the label and summing those two values. The sum would then be passed into our PC select MUX. Our WF control logic always checks if there is a JAL instruction in the decode stage and if there is, the logic will set our PC select MUX to ensure it outputs the value outputted by the JAL adder. This logic avoids us flushing a stage and thus improves our CPI. On the other hand, our JALR implementation was slightly different as it has to sum the value in rs1 and the immediate. Since our immediate generator was in the execute stage and our ALU had the functionality for summing rs1 and an immediate, we handled JALR in our execute stage. As a result, there was no additional hardware for JALR but because it is evaluated in the execute stage, we have to flush the decode stage since there will be an instruction in the decode stage that we will most likely not be jumping to.

3.5 MUX Modules

Although our MUXs do not lead to our circuit operating differently, we implemented our MUXs differently by creating MUX modules. We created modules for a 2-input, 4-input and 8-input MUX. These modules act the same as if we just had combinational blocks in our normal code, but the purpose of these modules was to increase code organization and easily copy and reuse MUX implementations as we were coding.

4 Status and Results

At checkoff, we completed the requirements for the project alongside implementing and evaluating trade offs between different optimizations. Our final design was a 3-stage pipeline that used the branch prediction scheme from checkpoint 3, data forwarding, and small optimizations that reduced our critical path. The final performance metrics for our project include: our CPU runs at a frequency of 80 MHz with a CPI of 1.082. Regarding area utilization:

- Slices had a 8.33% Util% and 1108 were used.
- 2089 LUTs for logic were used with the majority (1977) “using 06 output only” and
- 100 LUTs for memory were used
- 1610 Slice registers were used with a 1.51 Util%
- 34 Block RAMs were used on the board.
- 0 DSPs used

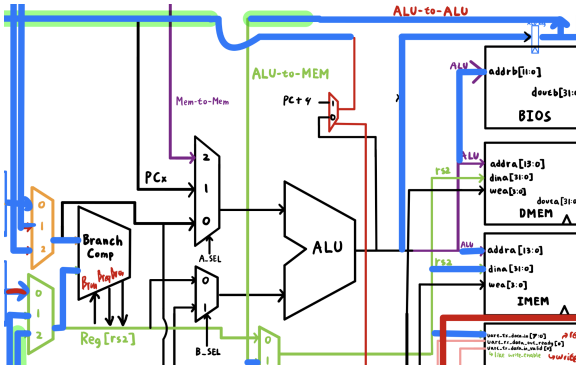
Below are optimization tradeoffs we implemented and considered. In the end our final design took in optimization 4.1 which was the hardware optimization we had with the branching logic and utilizing the ALU versus adding another MUX for the logic. We made our decision primarily based on the performance of the ALU utilizing the Iron Law equation with our CPI and frequency, while also evaluating the tradeoffs in metrics between other potential implementations. In the end, our final implementation achieved the highest frequency compared to other optimization implementations at 80 MHz and the lowest CPI at 1.082. The mmult.c program ran at 0.174s, using the Iron Law, which was the lowest out of all other optimizations.

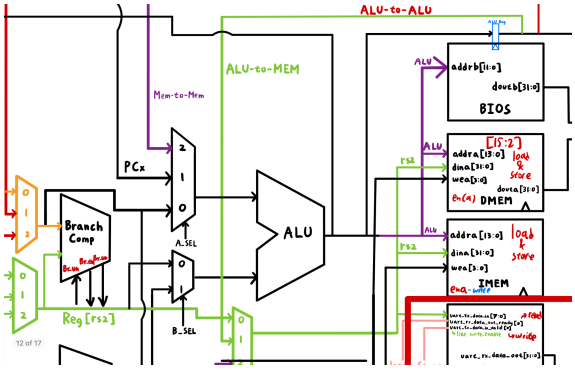
```
All user specified timing constraints are met.


_____________________________________________________________________________________________

| Clock Summary
| _____________

_____________________________________________________________________________________________

Clock                Waveform(ns)         Period(ns)      Frequency(MHz)
_____                ____________         __________      ______________
CLK_125MHZ_FPGA      {0.000 4.000}        8.000           125.000
  cpu_clk_int        {0.000 6.250}        12.500          80.000
  cpu_clk_pll_fb_out {0.000 20.000}       40.000          25.000
```

```
Unrecognized token:
151> jal 10000000
Result: 0001f800
Cycle Count: 00e502ed
Instruction Count: 00c4c2ed
Branch Instruction Count: 00382040
Correct Branch Prediction Count: 001be041

151> jal 10000000
Result: 0001f800
Cycle Count: 00d4daaf
Instruction Count: 00c4c2ed
Branch Instruction Count: 00382040
Correct Branch Prediction Count: 002c087f
```

4.1 Hardware Optimizations

After completing checkpoint 3 and beginning the optimization phase, we first found our critical path in hardware/build/post_route_timing_summary.rpt. We evaluated a few elements within the critical path that we could slightly change in our design to hopefully reduce the critical path and increase the frequency of the CPU. One of these changes was that initially we had a MUX that had two elements, the PC + Label and PC + 4. This MUX would be for the case when we predict wrong in our decode stage and as a result we have to change our PC value in the next cycle.





We removed the MUX for our optimization and put this logic into the ALU by also passing in the PC value and logic that we implemented to determine whether our prediction was correct or not. We found that removing this hardware and adding this logic within the ALU reduced our initial critical path after checkpoint 3 and our frequency went from 75 MHz to 80 MHz. We still maintained the same CPI with this design since we did not introduce new stalling logic or anything else that would change the number of cycles run. Thus for this optimization our CPI was 1.082 and this was our final design.

4.2 Stalling

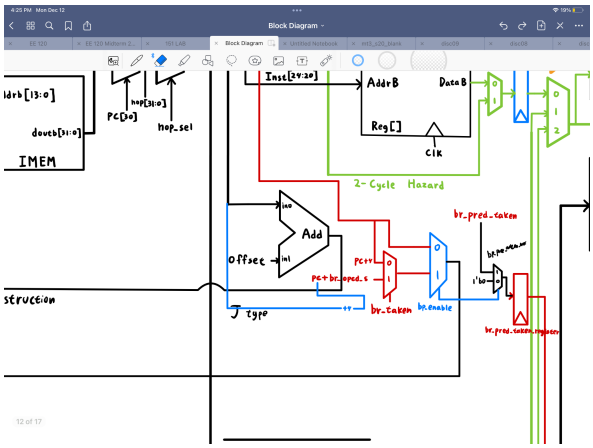
Another optimization we evaluated was removing forwarding and stalling for data hazards. The reasoning behind this was that we considered whether forwarding logic would potentially increase our critical path. We were aware of the tradeoff here as without forwarding we would expect a higher CPI. After implementing forwarding for all data hazards from our original implementation after checkpoint 3, we got a CPI of 1.409 without branching and 1.326 with branching and it could only run at a frequency of 75 MHz. We expected the result of the CPI as we completely removed data forwarding. However, the frequency result was a bit surprising at first as we expected the removal of forwarding to reduce frequency as it allowed us to reduce some hardware in the critical path, including reducing the size of MUXs or even removing MUXs such as the MUX for the 2-cycle hazard or teh Mem-to-Mem hazards for the rs2 register for store instructions. In the end, this result was not feasible as it did not meet the minimum of 1.2 CPI and the frequency did not increase so the overall performance significantly dropped.

4.3 Branch Prediction Cache

Another optimization we implemented was improving the branch prediction scheme by utilizing a new branch cache. We implemented a 2-way associative cache, different from the direct cache in checkpoint 3. After passing the local branch_prediction tests and our own cache testbench to ensure the functionality, we tested it on our FPGA. Our hypothesis was that this would impact the CPI, as there would either be more hits or misses, and we wanted to see if this cache would improve our performance by reducing our CPI. After running mmult, we found that our CPI had in fact increased to 1.163, which was almost the same as our CPI prior to implementing the branch prediction scheme from checkpoint 3. Thus we determined that a 2-way associative cache was not beneficial for our implementation as it did not get as many hits as the direct cache from checkpoint 3. Additionally, the frequency was not altered as expected. Thus we made the tradeoff between a direct mapped cache and 2-way associative cache and found the direct mapped cache to improve performance significantly, especially for the mmult program on the FPGA.

4.4 Jal Instruction

Another change we had first considered in our design and then determined the tradeoffs for was the Jal implementation we had. This was our design:



We had an adder in the decode stage to add the PC plus the offset from the Jal as we knew that the jal instruction was guaranteed to jump to wherever the label indicated. This was designed so we would not have to add a stall as we did for the JALR implementation since it utilized the value of rs1 and immediate generator. We also evaluated the tradeoffs of not having this adder and the following are the results: without branching we had a CPI of 1.184 and with branching we had a CPI of 1.102. The highest frequency this implementation achieved was 80 MHz. When determining the tradeoffs between our implementation with the adder for Jal in the decode stage versus not having the implementation, we were able to achieve the same frequency of 80 MHz and also have a lower CPI. The one benefit that not having this Jal adder could provide is reducing hardware space utilization, but we did not see this having a significant effect on performance compared to the higher CPI with this implementation.

4.5 4-Stage Pipeline

Finally, we implemented a 4-stage pipeline. Coming into this, we had seen in our critical path that our LDX mux and a few steps beyond were a part of the critical path. Thus we have another set of pipeline registers after our WB Mux. Now we had a new writeback stage in our pipeline. Here we expected that our critical path would hopefully slightly decrease. When we ran our implementation, we saw that this in fact did not reduce our frequency as the critical path was more embedded within the memory stage when reading or writing to our different memories. As part of the implementation, we retained data forwarding for Mem-to-Mem, Mem-to-ALU, ALU-to-Mem and 2 cycle hazards, but stalled for 3-cycle hazards. The result of this was that without branching we had a CPI of 1.205 and without branching we had a CPI of 1.14, as expected due to more stages and stalling for 3 cycle hazards. The surprising part was that our frequency was only at 75 MHz, not an improvement from our previous implementation. Thus in the future, we would pipeline the MEM stage to reduce the critical path and ideally improve the frequency.

5 Conclusion

We both really enjoyed taking on this project. This was both of our first times taking on a hands-on electrical engineering project from start to finish. Reflecting back on our process from first being introduced to the project to eventually getting a checkoff in dead week, there are a few things that we thought we did well and a few things that we would do differently next time. We got started on the design after wrapping up Lab 5 and went thoroughly through the spec. The idea of first beginning with a 3-stage pipeline and then adding more stages if needed added a lot of clarity. Additionally, the checkpoint 1 questions, especially the questions about how we would handle forwarding logic and certain functionalities that we were initially confused about, added a lot of clarity. These initial steps allowed us to spend a lot of time on the design portion which we think allowed us to have a decent frequency result at checkpoint 2 (~70 MHz). Additionally, there were a few modules that we put into separate files, such as a 2-input MUX and 4-input MUX module, which allowed us to stay clean across our code and keep track of all our elements which helped during debugging.

From the project there were a few things we took out of it aside from better understanding the functionality of a CPU and the FPGA boards. The first takeaway was the importance of first understanding the objectives and creating a roadmap on how to tackle a big project like this. Understanding the basic functionality we needed for checkpoint 2 (i.e, 3-stage, handling hazards, CPI < 1.2, etc.) allowed us to create a design to meet these needs that we could then build up on for future checkpoints. Additionally, we learned the importance of documenting as we built our CPU. This was crucial for debugging whenever we ran into issues. Also, documentation by continuing to modify our datapath allowed both of us to evaluate different design tradeoffs and implementation logic.

The biggest challenge for us throughout the project was time as a limiting factor. We had a lot of fun during the optimization process but also spent hours debugging different optimization strategies due logic, test bench logic, and other factors. In the future, we would like to continue to improve the speed of our CPU, such as adding more pipeline stages. Additionally, in the future, we would like to try other branch prediction cache implementations, such as a fully associative cache, which we think would significantly improve our CPI and improve our speed.

In sum, we managed to complete all functionalities of this project and the feeling of completing the entire project is second to none. A lot of hard work has been put into this project and we also get to reap the benefits of implementing such a marvelous CPU in just two months.