

RISCV 151 Project Report

Ayah Ahmad, James Deloye

March 6, 2024

1 Project Functional Description and Design Requirements

In this project, we designed and built a three-stage, pipelined RISC-V CPU with a UART and branch predictor, in Verilog, using a PYNQ-Z1 development board with a Zynq 7000-series FPGA. In our design, we sought to maximize the Iron Law of processor performance and minimize FPGA resource utilization, in order to optimize our CPU.

Iron Law

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

1.1 Pipeline Structure

We built a three-stage pipeline, where we predominantly considered the critical path and how to best minimize it. Our initial considerations depended on the synchronous elements (DMEM, IMEM, and BIOS Memory RAMs) of the datapath, in order to determine register placement, and our later considerations took into account the asynchronous combinational logic blocks where we sought to minimize the critical path of our design. This resulted in numerous iterations and refinements of our design, throughout the engineering process, while we tried to find a design with optimal performance. We also had to consider different ways of forwarding data and how to best deal with data and control hazards.

1.2 Memory Hierarchy

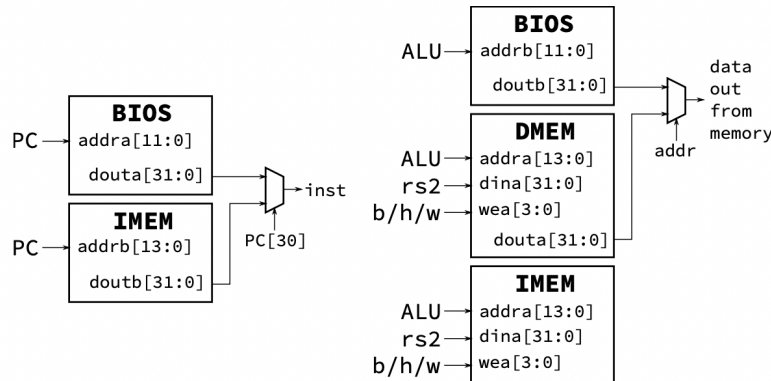


Figure 1: This diagram depicts the general memory hierarchy for the project. On the left, we have the instruction fetch logic and on the right we have the memory load and store logic.

Source: FPGA FA22 Project Specifications

1.2.1 BIOS Memory

This is where the processor begins its execution, as it gets all of its instructions from here. The BIOS program is able to read from BIOS Memory, DMEM, and IMEM, and write to both DMEM and IMEM, using the UART.

1.2.2 Instruction Memory (IMEM)

This contains the current instruction that the CPU is executing.

1.2.3 Data Memory (DMEM)

This contains the data from the execution of an instruction.

1.2.4 Memory-Mapped I/O (MMIO)

This allows us to use registers of I/O devices as though they were memory, by assigning them memory addresses. This was used for communications between the CPU and the UART.

2 High-Level Organization

2.1 Overview

We broke our datapath into three main parts, a Read/Decode stage (RD), an Execute stage (X), and a Memory Write-Back stage (MW). Our control signals were also split into these three main stages, in order to potentially reduce the critical path.

2.2 Datapath

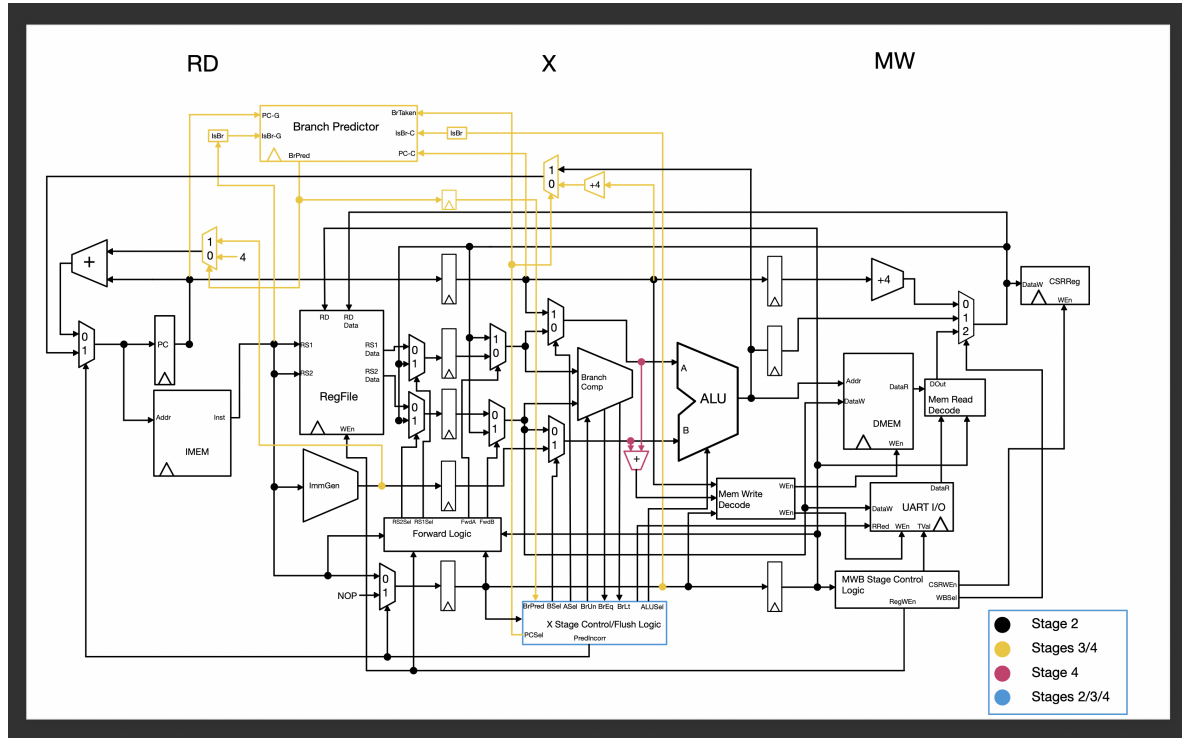


Figure 2: This diagram depicts the high-level organization of our CPU.

Pictured above is the final design of our CPU Datapath. As we progressed through the stages of the project, we iterated through different designs, optimizing along the way. In the second stage of the project, we landed on the design in black. Here we had our base CPU implemented, with forwarding logic to resolve control and Read-After-Write (RAW) data hazards. We implemented the majority of our control logic in this stage, adding more logic as we progressed through the following two stages. In the third stage, we built the two-bit saturating counter branch predictor and decided on how we would implement our `jal` forwarding logic. In the final stage, we changed our branch predictor twice, finally landing on this design of an agree predictor. We also implemented our `jal` forwarding logic and placed an adder in parallel with the ALU to reduce the critical path. We generally broke up our design into three parts, as described below.

2.2.1 Read/Decode

In this stage, an address is sent to the IMEM, where the instruction at that address is read, and generates the immediate, based off of that instruction. This immediate is routed to numerous places; primarily to a pipeline register, but also back to the program counter selection module, where, depending on the instruction, it could be broken down and the address (eg, in the case of branches) used as an input to the PC. The registers are also read from the RegFile and stored into pipeline registers. This stage can also receive forwarded data from the MWB stage to address dependent instructions that are one instruction apart.

2.2.2 Execute

In the execute stage, the control signals are generated from the instruction, branch comparisons are performed, and ALU does its computations. Data is forwarded to this stage from the MWB stage to deal with hazards if necessary. In addition, the necessary write-enable signals are calculated for the memory and UART.

2.3 Memory Write-Back

In this stage, the DMEM is read from or written to, and the RegFile is written back to. The CSR register can also be written to from here, and the UART interface for serial I/O is also located here.

3 Detailed Description of Sub-Pieces

3.1 Forwarding: Read/Decode

3.1.1 Program Counter Selection

The program counter starts at the base of the BIOS memory. Control of where the PC goes next is then mediated by the branch predictor, which will make a jump if the current instruction output of the IMEM is a `jal` or a predicted branch jump. Otherwise, the predictor always outputs 0, which results in the PC continuing to the next instruction at PC+4. The selection of the branch predictor can be overrode after by the instruction in the X stage if there is a branch misprediction or there is a `jalr` instruction, in which the current RD instruction is flushed and the next PC is set to the corresponding ALU output.

3.1.2 Instruction Memory

The structure of the instruction is given in Section 1.2.2. Selection of which memory is used is controlled by the upper four bits of the incoming PC value, which will either point to the BIOS or instruction submemory. The corresponding instruction is then retrieved and output.

3.1.3 RegFile

The RegFile has two asynchronous read ports and a single synchronous write port. It outputs the corresponding data in the register if the register address is valid, otherwise it outputs 0 as a default value. The write data and address comes from the WB stage and writes the data to the appropriate register if the Write Enable signal is high.

3.1.4 Immediate Generation

The immediate generator simply takes in the current instruction, interprets its type via the opcode, and extracts the applicable immediate as specified for the instruction type in the RISC-V ISA.

3.2 Forwarding: Execute

3.2.1 ALU

The ALU uses the corresponding control signals from the X stage control logic to select which operation to perform on the A and B inputs.

3.2.2 Branch Comparator

The branch comparator takes in the relevant signals from the control stage and performs the comparisons on data to determine whether a branch is taken. This data is ultimately used to determine if the pipeline needs to be flushed and different instructions need to be loaded in.

3.2.3 Memory Write Decoding

The memory write decoder interprets which write enable bits to set on which memory submodules/UART controller depending on the input address and what type of write instruction it is. If it is not a `store` type instruction the controller does not enable writes for any of the memory.

3.3 Memory Write-Back

3.3.1 Data Memory

The DMEM block acts as a wrapper for the instruction and data submemories with synchronous reads and writes as detailed in section 1.2.3. The write enabling is controlled by the memory write decoder that comes before it in the datapath.

3.3.2 Memory Read Decoding

This decoder uses the instruction and upper bits of the memory address in order to determine which submemory/UART to read output from and in what format to extend the data bits depending on whether they are signed or unsigned.

3.3.3 UART Input/Output

This acts as a controller for the UART interface, accepting the appropriate control inputs from the control logics in the X and MW stage as well as maintaining the internal counters for the number of instructions executed and the number of branches and how many were correctly predicted.

3.3.4 CSR Register

This is a single register used as the output destination for CSR instructions in simulation.

3.4 Branch Prediction

The original branch predictor used is a cache-based two bit saturating counter that allows us to determine whether to take a branch or not by looking at the top bit of the counter for that given instruction in the cache. In the execute stage the correctness of the prediction is checked by using the branch comparator output to set the next value of the saturating counter. During the optimization stage of the project, we attempted to create a more advanced branch predictor, primarily using the methods detailed in [Agner Fog's paper on the microarchitecture of Intel and AMD CPUs](#). To try to improve prediction rates, we tried implementing a two-level branch predictor, which uses the behavior of the last two jumps made for each branch to further stratify the predictions for the next branch. Otherwise, the predictor behaves very similar to the two-bit saturating counter.

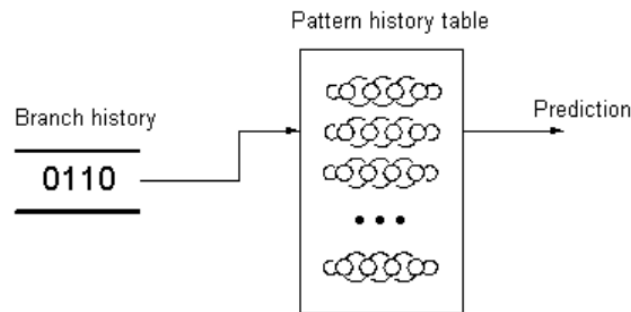


Figure 3: This diagram depicts the basic structure of a two-level predictor. *Source: Fog (1996)*

In addition, we also implemented the even more advanced agree predictor, which utilizes the last 8 global history jumps to do enhanced predictions. Corresponding to the global history is a two-bit saturating counter for each history that provides an indication of whether the global history agrees or not with the normal cache prediction for that history. If it doesn't agree, it flips the cache's predicted output. [The original paper on agree predictors by Sprangle Et al. was used to implement this.](#)

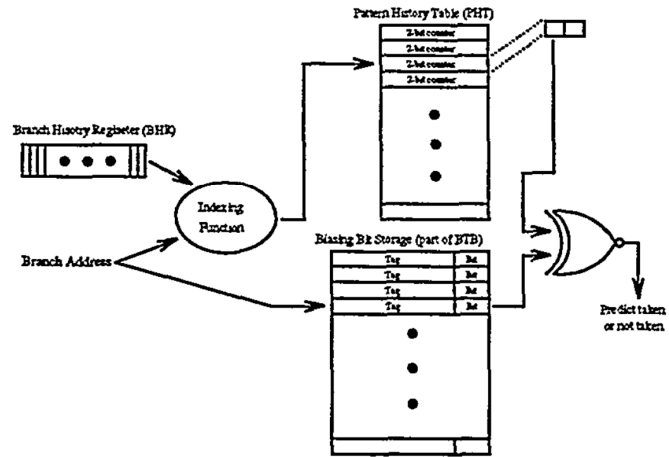


Figure 4: This diagram depicts the basic structure of an agree predictor.
 Source: Sprangle Et al. (1997)

4 Status and Results

4.1 Overview

Our final CPU can run at approximately 65 MHz, but this frequency was only attained after a number of optimizations and considerations. In our pre-optimized datapath, our CPU was only able to attain a frequency of approximately 50MHz; the optimization that most increased frequency was our consideration of the critical path. Our other optimizations (branch prediction and `jal` forwarding) were able to play a significant role in reducing the CPI, but generally did not impact the frequency of the CPU.

In our considerations for optimization, we wanted to stick closely to our original design, as we had iterated through it numerous times, and felt confident in it. While removing our forwarding paths may have allowed us to attain a higher frequency, we didn't want to get rid of them, due to their high utilization. We also discussed adding more stages to the datapath but agreed that the complications associated with the increased pipelining would not be worth it, both in terms of the additional hardware and maintaining overall simplicity of design. These considerations limited our ability to increase frequency, so we instead decided to focus our optimizations on increasing CPI.

4.2 `jal` Forwarding

In order to reduce the number of `nops` we'd have to insert into our pipeline, we decided to treat `jal` instructions as special-case instructions, where we would detect if an instruction was a `jal` instruction, as detailed in Section 3.1.1. Previously, whenever we had a `jal` instruction, we would have to insert a `nop`, in order to stall for a cycle until the `jal` address was computed. When doing the `jal` forwarding, we initially started by creating an entire module dedicated to it, but later realized that we would be able to reduce our area utilization by re-using control signals and combining it with the pre-existing hardware for branches. In this final design, we were able to reduce the stalling we had to do and were thus able to reduce our CPI on the `mmult.c` program, by approximately 0.02. We went from a CPI of 1.184 with the original design, to 1.164 after `jal` forwarding.

Since `jal` constitutes a minor subset of the entire RISC-V Instruction Set, we knew that the gains that we achieved with `jal` would be highly dependant on the use of `jal` within a given program. Thus, we sought to further reduce our CPI using branch prediction.

4.3 Branch Prediction

We iterated through three different designs for our branch predictor (outlined in section 3.4). The first was the naive implementation of the Saturating Counter Cached Branch Predictor, which was implemented in Checkpoint 3. Our initial Checkpoint 2 CPI was 1.184, whereas the new CPI with the branch predictor was 1.102. Without the branch predictor, our accuracy on the provided `mmult.c` program was only 0.497, but with the branch predictor implemented, we got an improvement of 158%, with a new accuracy of 0.785.

We then implemented a Two-Level Branch Predictor (TLBP), which further improved our CPI by decreasing it from 1.102 to 1.079. From the naive implementation, without the branch predictor, to this implementation, the accuracy improved 174%, with a new accuracy of 0.864. This however, led to higher area utilization than the Saturating Counter Cached Branch Predictor, and only yielded a 110% improvement.

We further implemented the Agree Branch Predictor (ABP), which was our final branch predictor design. Not only did this branch predictor have less area utilization than the TLBP, the newer ABP also outperformed the TLBP. The ABP reduced our CPI to 1.064 and **improved our original accuracy by 188%**, giving us a **final branch prediction accuracy of 0.934**.

4.4 Critical Path Minimization

In the original design, the critical path was caused by forwarding, where the result from the BIOS Memory was being forwarded to the UART, bypassing registers. Here, we noticed that other than placing a register along this path (which would result in a reduced CPI), the only other possible way to minimize this path was by minimizing the ALU delay. Thus, we placed an adder in parallel with

	SC Cache BP	Two-Level BP	Agree BP
CPI	1.102	1.079	1.064
LUTs (/53200)	2125	3827	3215
Block RAM Tiles (/140)	34	34	34
FFs (/106400)	1912	2957	2447
DSP Blocks (/22)	0	0	0

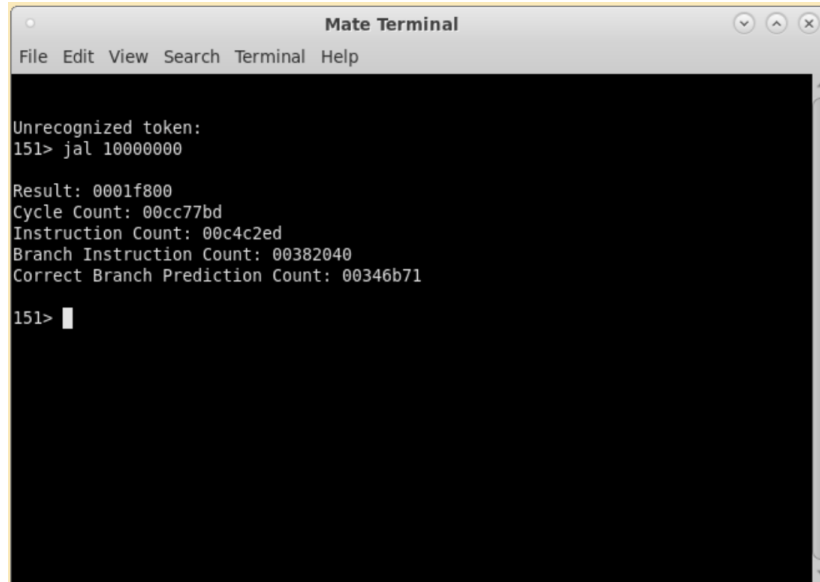
Table 1: Area utilization of different branch predictor implementations.

the ALU and used that as an input to the MemWriteDecoder. This resulted in the largest frequency gains; going from approximately 50 MHz to 60 MHz.

After minimizing this critical path, we found another critical path from DMEM to IMEM, again caused by forwarding and bypassing registers. Instead of inserting NOPs into our pipeline, we instead decided to do a similar optimization as before; we again used the adder in parallel with the ALU as input to our PC flush mux, which reduced the critical path further. Here, the frequency gains were not nearly as good as before, only allowing us to increase to approximately 65 MHz, as our final frequency. At this point, we had nearly halved the fanout of the path with the largest slack time, and our wire delay was already nearly as much as the logic delay, so we figured that any further optimizations would likely not yield impressive results.

4.5 Integration and Final Results

Since each of these pieces was created and tested independently, in order to gauge performance on the original design, integration of all of these pieces together yielded the best results. Our final **LUT count** was **3349 out of 53200 available**. Our final **Block RAM Tile count** was **34 out of 140 available**. Our final **Flip Flop count** was **2446 out of 106400 available**. Our final **DSP block count** was **0 out of 22 available**. Once we combined all of these submodules, we were able to achieve a **final frequency** of **65 MHz**, but since our focus was on optimizing the CPI, we felt no need to improve it drastically. Thus, we were able to achieve a **final CPI** of **1.039** and a branch prediction accuracy of **0.934**.



```

Mate Terminal
File Edit View Search Terminal Help

Unrecognized token:
151> jal 10000000

Result: 0001f800
Cycle Count: 00cc77bd
Instruction Count: 00c4c2ed
Branch Instruction Count: 00382040
Correct Branch Prediction Count: 00346b71

151>

```

Figure 5: Final CPI

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 7.692}	15.385	65.000
cpu_clk_pll_fb_out	{0.000 20.000}	40.000	25.000

Figure 6: Final Clock Frequency

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	3349	0	0	53200	6.30
LUT as Logic	3249	0	0	53200	6.11
LUT as Memory	100	0	0	17400	0.57
LUT as Distributed RAM	100	0			
LUT as Shift Register	0	0			
Slice Registers	2446	0	0	106400	2.30
Register as Flip Flop	2446	0	0	106400	2.30
Register as Latch	0	0	0	106400	0.00
F7 Muxes	475	0	0	26600	1.79
F8 Muxes	208	0	0	13300	1.56

Figure 7: Final Slice Logic

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	34	0	0	140	24.29
RAMB36/FIFO*	34	0	0	140	24.29
RAMB36E1 only	34				
RAMB18	0	0	0	280	0.00

Figure 8: Final Memory

5 Conclusion

From design, to implementation, to testing and debugging, to seeing all the optimizations work on the FPGA, the entire experience was definitely enlightening.

Had we had more time, however, we may have delved even further into our optimizations. While we were generally quite happy with our overall design and results, had there been more time to test on the FPGA, we may have tried increasing our frequency and pushing it past 100 MHz, perhaps by increasing the number stages of the datapath. Since one of our main priorities was to keep what was functional, functional, we erred on the side of safety, especially with respect to moving around modules in our datapath. Our optimizations on the frequency side were rather meager, compared to those on the CPI side, and seeing both of those fully realized and optimized would have been a rather exciting experience.

Additionally, next time we would likely try to finish the second checkpoint a little sooner. While the deadline was flexible, we fell a little behind, and ended up submitting it later than we would have liked. Thankfully, this didn't carry over to the third checkpoint, but had we completed it sooner, we may have been able to work on the optimizations sooner, and thus experiment more. For example, it would have also been interesting to do other extensions, perhaps trying to implement the RISC-V M Extension and allow our processor to multiply and divide.

Overall, we learned a lot; we were able to really delve into the process of building our very own RISC-V CPU.

6 Extra Credit

6.1 Branch Predictor

See section [3.4](#) Branch Prediction, for further elaboration on the design of the Agree Branch Predictor, and [4.3](#) for its performance.

6.2 Testing

In addition to the given tests, several test frameworks were written in order to minimize issues later down the line. Unit tests were written for individual modules like the ALU, immediate generator, and regfile. In addition, thorough assembly tests were written that caught many issues with memory behavior. Finally, more advanced tests were written for the branch predictor that would allow for a basic heuristic of how good the branch predictor is predicting in simulation before running mmul on the FPGA.

7 Division of Labor: Ayah Ahmad

As a team, we tried to split the work as evenly as possible; while there were times where one person would work on the project more, due to a busy week, perhaps, the other person would try to compensate for it in the following parts. For Checkpoint 1, we worked together on the design of the CPU. Both of us had similar ideas for how we wanted to divide our pipeline into the three stages, so after discussing and deciding on the final design, James drew it up and pushed it to GitHub. Since drawing it up seemed to take a while, I took ownership over the Checkoff 1 questions, and again, we both discussed each of the points together, but I ended up writing it up. It seemed like a pretty balanced split. For checkpoint 2, we both worked on building out different parts of the CPU. Again, each of us did around half of the smaller modules (I can't remember what the exact split was but it was 3 small modules for me to implement, and 3 for James). After that, while I worked on forwarding and control logic, James worked on Memory and UART integration, but it did feel like James did more work in this stage. For checkpoint 3, I think the split was a little less clear; the final design for the saturating counter and the cache were James', but I had started implementing them at the same time; he had finished his sooner though so we just proceeded with using his implementation. I ended up writing the testbench for the cache to verify its functionality. For Checkpoint 4, James predominantly worked on improving the CPI, by re-writing our branch predictor (this was all his idea and this work; I was really, really impressed by it). I worked on reducing frequency and fanout, and finished up JAL forwarding. Since I felt like James did a lot of work on the project in previous stages, I didn't want him to feel like he was getting the short end of the stick, so during this checkpoint I did a lot of testing and trying to really look at the tradeoffs between different designs. This is also why I asked to write the majority of the lab report. I didn't want the split to feel uneven. I think we did do, or try to do, around the same amount of work and split it evenly overall, even if each individual stage wasn't an exact 50/50 split.