3 Idiots Processor Design Report

By Arpan Swaroop, Om Padmani, Yugal Kithany

1. Introduction

This project centers on designing and implementing an Out-of-Order (OoO) microprocessor tailored for the RISC-V 32IM ISA. OoO microarchitectures have become essential for modern processor design, addressing growing commercial demands for computational efficiency and versatility. Our implementation employs Explicit Register Renaming (ERR), a strategic refinement over Tomasulo's algorithm, effectively resolving potential issues such as excessive inputs to the Common Data Bus (CDB) and conflicts arising from simultaneous register writes. This report explains our design choices to enhance the performance and adaptability of our RISC-V processor by maximizing the instruction-level parallelism (ILP) and boosting instructions per cycle (IPC). Our design choices enhanced the utility of our base RISC-V implementation. Completing this design has been an essential step in learning about computer architecture due to the hands-on nature of the project. An in-depth understanding of practical computer architecture is complex and difficult to develop without dealing with implementation challenges and weighing feature tradeoffs. In doing so, we deepened our understanding of advanced microarchitectural concepts and equipped ourselves with valuable expertise for future careers within the semiconductor industry.

2. Project Overview

The first goal of this project was to address the baseline processor requirements such as OoO execution, memory instruction support, control instruction support, and general signed and unsigned multiplication, division, and arithmetic instructions. Past these initial goals, we hoped to have multiple advanced features that would allow us to increase our performance. We had aimed to implement a split load store queue, a branch predictor, and a prefetcher for both instruction and data accesses. These would be some enjoyable and attainable features to target that could significantly improve our performance. We were interested in one of the above features, leading to those specific ones being our choices.

Regarding work sharing throughout this project, we aimed to split tasks amongst our members equally and to aid each other in case someone finished early. Regarding organization, we decided to work on separate branches and then come together to integrate features efficiently. If project complications arose, we agreed to discuss them with each other and to defer to our project mentor if necessary. Below, our initial block diagram shows how we planned our processor out.

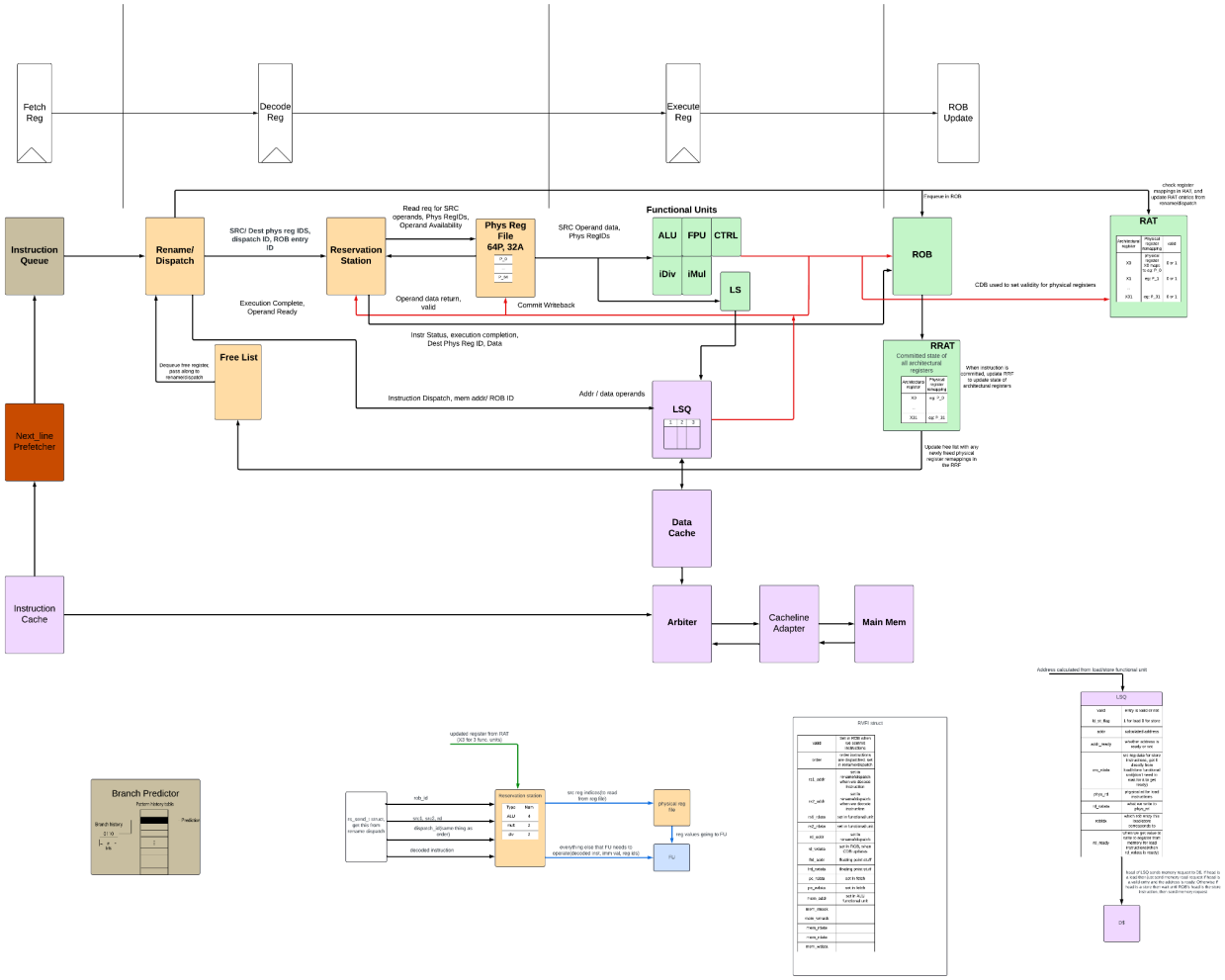


Figure 1: ERR Block Diagram

3. Design Description

3.1 Overview

This project was divided into three key checkpoints to promote consistent processor development. Each checkpoint significantly advanced the processor's functionality and compatibility with the RISC-V ISA. We incorporated the instruction cache early in the first checkpoint to reduce the design integration overhead. This decision improved our experience implementing the memory instructions, enabling us to focus on writing clear RTL for the other deliverables. The cache line adapter was also written modularly to ensure ease of integrating prefetchers. We assessed the requirements of each checkpoint and planned out the specific aspect of the processor that each would focus on. This preparation allowed for a fair workload distribution and made us think about integration early on. Finally, the most notable cause for our success was creating a well-defined block diagram and datapath for the processor and specifying the significance of each module and its place in the puzzle.

3.2 Milestones

3.2.I Checkpoint 1

The objectives of this checkpoint were to create a high-level microarchitecture diagram of the processor, shown in Figure 1, implement a parameterizable queue for internal instruction buffering, develop a cache-line adapter to handle DRAM bursts and integrate the fetch stage with a DRAM model.

Specifically, our queue was circular with head and tail pointers that had one extra bit to signify what lap each pointer was on. Our cache-line adapter had stages in which we handled different aspects of the burst memory, such as initiating the memory request, each stage of the burst, and when the request was finished. Our high-level block diagram described our overall design of the ERR architecture and how each of the components would interact with each other.

To test the developed RTL components, we utilized DUT testing on both the individual modules and the entire system once integrated. For instance, we tested the FIFO edge cases of enqueuing when full or dequeuing when empty before integrating the queue with the fetch stage. We also ensured the expected behavior was maintained when the queue wrapped around. Then, once integrated, we included more than 16 instructions(our instruction queue size is 16) to test the functionality thoroughly. To test the cache line adapter's compatibility with the DRAM burst memory, we opened the simulated waveform and verified that each burst was handled correctly within the Mealy machine. Once this was done, we tested it on an assembly file and ensured all the correct instructions were formatted into the instruction queue. The outcome of this checkpoint was establishing a functional fetch stage and integrating it with DRAM memory and an instruction cache.

3.2.II Checkpoint 2

For checkpoint 2, we had to implement all RV32I immediately and register instructions, excluding any control or memory instructions. We also had to integrate the multiplier and divider Synopsys IPs to implement the RISC-V M extension. This was the checkpoint where we had to design and implement most components in an explicit register renaming architecture. Checkpoint 1 was where we made the fetch stage; here, we made the components to implement the decode, execute, and write-back stages.

In the decode stage, the first component is renamed dispatch. This is where we dispatch the instruction to the appropriate reservation station and the ROB queue. Additionally, rename dispatch is also where we dequeue a free register(for the destination register) from the free list and then update the register alias table(RAT) accordingly. Sitting in between decode and execution lies our reservation station. This is where we hold instructions until all the operands

are ready for execution. For every functional unit, we chose to have a separate set of reservation stations; this just made it more streamlined for us to handle issuing instructions to the functional unit for execution.

For our execute stage, we have our functional units; for this checkpoint, we included ALU/CMP, multiplication, and division functional units. We integrated given Synopsys multiplication and division IPs for our mul/div functional units. We chose sequential IPs to minimize our critical path and optimize our frequency. After executing the functional units, we have to broadcast the result on the central data bus(CDB) to the reservation stations, physical register file, return order buffer(ROB), and RAT. An important design choice we made was to have a separate CDB for each functional unit; this ensured that we could immediately broadcast the result as soon as it was ready, effectively minimizing wakeup latency. This was a tradeoff for a higher area, as each CDB is expensive area-wise because of the large amount of read/write ports that need to be synthesized.

Finally, we reach the last part of the pipeline, the write-back stage; in this stage, we have the ROB and the retirement register alias table(RRAT). Our ROB is a first-in-first-out queue, which ensures that we commit instructions in the correct order; this is integral since we execute instructions out of order. The RRAT stores the last non-speculative architectural state of the processor and is updated whenever the ROB commits an instruction. Every instruction processed in our processor is speculative until the ROB commits it. So, it is important to store the last non-speculative state(this will be important when we implement control instructions). Moreover, once the RRAT receives an update, we enqueue the old register that was mapped back into the free list.

For testing, we utilized given assembly test cases, wrote our own test cases, and had a random testbench, which tested all the instructions that our processor was required to process. Our random testbench excluded all control and memory instructions. Using random testing, we were able to find and fix many bugs in how we processed certain instructions. For example, we had to deal with the divide by zero exception in our division IP.

3.2.III Checkpoint 3

For checkpoint 3 we had to implement all of the control and memory instructions. This meant adding the load store queue module, the control functional unit, and the load store functional unit. We also had to make additions to the modules previously implemented such as adding control and memory stations to the reservation station and implementing flushing logic for mispredicted control instructions.

Specifically for the load store queue, we made a FIFO where we enqueued into it from rename dispatch if the specific instruction was either a load or a store. The same instruction would then go through the reservation stations and load store functional unit to compute the address to load from or store to. This address would then be sent to the load store queue to

update the index of the queue with the specific instruction's memory address. If the head of the queue was a store, the address was ready, and it was at the top of the ROB, then we could proceed and execute it. If the head of the queue was a load and the address was ready we could proceed and execute it.

The load store functional unit calculated the address from which to load from or store to using basic arithmetic operations. The control functional unit similarly calculated the address to branch/jump to as well as whether the branch should take place. This data was stored into the ROB and once the instruction became the head of the ROB, the branch/jump was taken if valid and the processor would flush all speculative values since we were statically predicting it to not be taken.

to test for this checkpoint we used both the given test cases like coremark_im as well as our own targeted C and assembly files as test cases for individual components. We also modified our random testbench to include the new control and memory instructions. With the random tb along with the provided test cases passing, we were confident in our processors correctness.

4. Advanced Design Features

4.1 Next-Line Prefetcher (Instruction Cache)

- Design: The Next-Line Prefetcher sits between the fetch module and instruction cache. If the instruction queue is not full, the fetch module will request the next PC address. The prefetcher will intercept this request, and calculate a prefetch address (current PC + 32'h20) if it deems that a prefetch is necessary. It will then send the original request to the instruction cache. When it gets a response, it will send a request for the prefetched address, and hold the mem_r_data until the prefetch responds. Once this happens, the NL will send the mem_r_data from the original response to the fetch module.

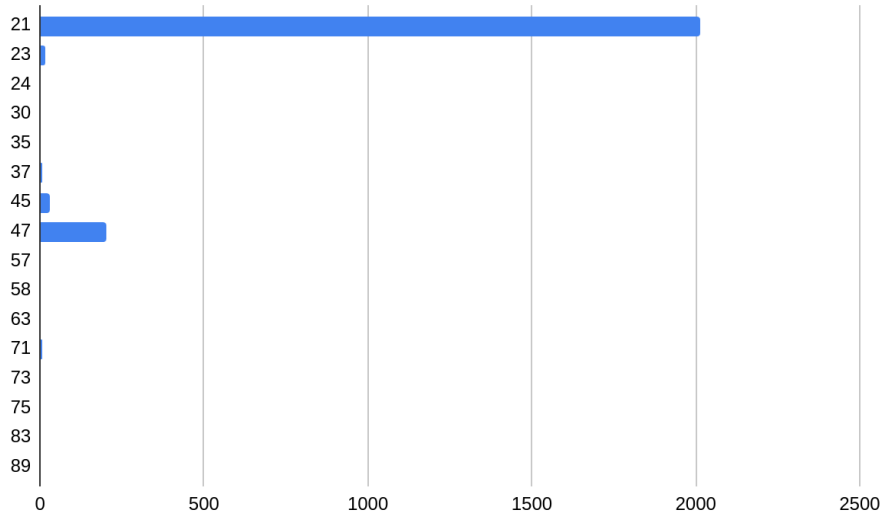
- Testing: This was tested as a DUT first to make sure that the expected behavior occurs, and the prefetch only occurs when intended. Then, I tested this on a simplified version of the provided mergesort. Finally, once removing any remaining bugs I tested it on the rest of the benchmark suite.

- Performance Analysis:

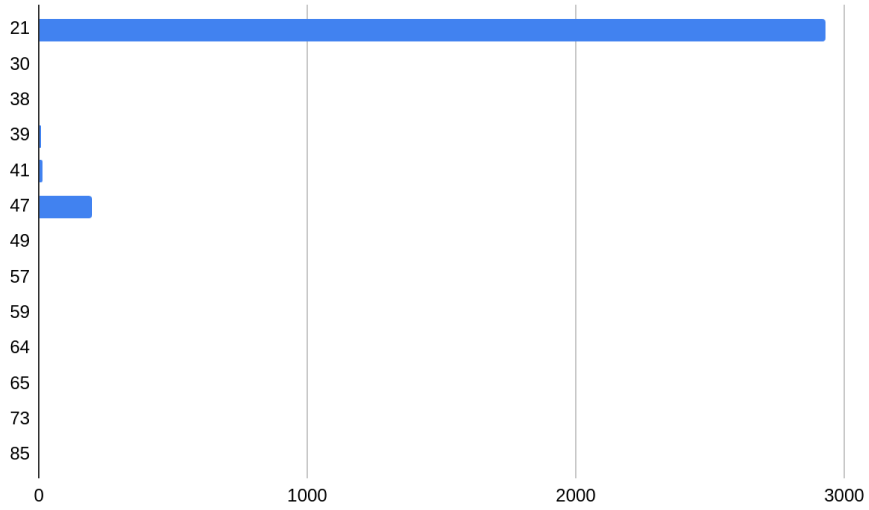
    - The Area increase with this advanced feature is 688.142003. The AMAT decreases from 31.086 to 24.53 on coremark_im after integrating this with the baseline. However, the delay increases and this is likely due to an accuracy of roughly 82%, but this could have been improved if it was better integrated with the BTB prediction. An effort was made to increase this accuracy by integrating the stride prefetcher directly into the cache_line adapter, but there was not enough time to finish this exploration. The implementation for NL PF within cache_line was done on the NL_redo branch, linked here

	coremark_im	FFT
NL AMAT	24.53229323	19.22280181
NL IPC	0.339568	0.360153
NL Delay	1382394300	1382394300
no NL AMAT	31.0864897	22.17652051
no NL IPC	0.355823	0.373018
no NL Delay	1319240440	2220563520
NL PF Accuracy	81.996	

Next, these histograms show the frequency of each wait for i_cache request in cycles. This was derived using a performance counter to and adding a wait_state_check, on the NL_performance branch here. Evidently there are significantly less (roughly 900) occurrences of the cpu having to wait 21 cycles, and this is because there was a prefetch hit.



Coremark_im with NL PF



Coremark_im without NL PF

4.2 Stride Prefetcher (Data Cache)

- Design: The Stride Prefetcher sits between the Split Load/Store Queue and data cache. When requesting a data address, the prefetcher will intercept this and calculate a prefetch address with an updating stride. It will then send the original request to the data cache. When it gets a response, it will send a request for the prefetched data, and hold the mem_r_data until the prefetch responds. Once this happens, the Stride PF will send the mem_r_data from the original response to the Split Load/Store Queue. This is expected to thoroughly improve data access times for merge sort and other benchmarks that continuously access the consistently increasing areas.

- Testing: This was tested as a DUT first to make sure that the expected behavior occurs, and the stride increments correctly. Then, I tested this on coremark_im, but the stride value always increased to very large values, making it not beneficial to integrate into the final submission

- Performance Analysis:

    - The Area increase with this advanced feature is 670.8184. The AMAT increases significantly from 31.086 to 38.58 on coremark_im after integrating this with the baseline, causing the IPC to decrease quite considerably from 0.3558 to 0.2927. This is likely due to a poor accuracy of only 34%, but we tried using different base stride values than 20 but it did not seem to help. It would likely have performed better if it was directly integrated into the cacheline adapter, but even

then the accuracy would have been off, so perhaps the method of updating the stride did not work. The performance counters were done in the same way as the NL prefetcher. It was not integrated into main because of the very poor performance, and so it is on the 3_yugal5 branch, linked here

Area Increase:	670.8184
	coremark_im
Stride AMAT	38.5817820
Stride IPC	0.292752
no Stride AMAT	31.0864897
no Stride IPC	0.355823
Stride PF Accuracy:	34.331%

4.3 G-share Branch Predictor with BTB

- Overall design: Our branch prediction scheme involved a G-share branch predictor along with a BTB. We implemented both the branch predictor and the BTB in the fetch stage where we pass in the pc, and get the speculative branch prediction and branch target in the next clock cycle. We only speculatively take a branch if we both get a branch taken prediction from G-share, and a BTB hit. This is to ensure that whenever we speculatively branch we have a branch target to branch to

- BTB design: Our BTB is implemented using SRAM, where we had a 16 entry, direct mapped BTB. Each entry is indexed using the bottom 4 bits of the pc, and each entry is tagged with the top 28 bits of the pc of a valid control instruction. Additionally, each entry also stores a 32 bit branch target address. Whenever we get a BTB hit, we can be sure that the pc we used to index the BTB corresponds to a control instruction. We update the BTB whenever we mispredicted the branch target, or whenever we were supposed to non-speculatively branch and didn't speculatively branch. The BTB works well with code that has branches that we take repeatedly like for loops. Tradeoffs with this design primarily include more area(primarily SRAM), and added critical path, because we input every PC we get from fetch into the BTB.

- G-share design: Our G-share uses a pattern history table implemented using SRAM, where we had 256 entries, each entry storing a 2-bit saturated counter. to index into this table we XOR the bottom 8 bits of the pc with the global history register(GHR). The GHR stores the outcomes of the last 8 non-speculative branches(1 for branch taken, 0 for branch not taken). Using the GHR we can take into account the global branch history of the program to index into the pattern history table. We update the pattern history table whenever we commit a control instruction, since two bit saturated counters need to be

updated based on the outcome of the branch. to do this we store the pattern history table index in the rob queue entries for control instructions, so that we can update the pattern history table efficiently. The G-share branch predictor can be very accurate if the recent history of branch predictions determines the current branch prediction(such as nested if statements). Tradeoffs with this design include more area(primarily SRAM for pattern history table), and added critical path because we input every PC we get from fetch into the branch predictor.

- Testing: to test the BTB, I wrote a C file with nested for loops to test lots of repeated branches. to test G-share I wrote a simple Design Under Test test case, and tested it against the same nested for loop C file. I also made an assembly file to test all sorts of branch instructions, and to check various branching conditions.

- Performance Analysis: Below I have compiled some performance metrics for the provided test cases. I also provide an explanation for each metric:

    - G-share accuracy: (number of correct g-share predictions)/(number of control instructions committed). This metric represents how accurate the branch predictor's predictions were in isolation(disregarding BTB hits). I calculated this by storing the G-share prediction for each control instruction in the rob queue entries, and checked if the prediction was correct at time of committing.

    - Speculative branch prediction accuracy: (number of correctly speculated branches)/(number of control instructions committed). This is the most important statistic, where I measure how many correct speculative branches we take.

    - BTB hit rate: (number of BTB hits)/(number of control instructions committed). This is represents how often we get a BTB hit

	Coremark	Mergesort	FFT	AES SHA
G-share accuracy	87.82%	77.38%	99.04%	95.72%
Speculative branch prediction accuracy	77.55%	68.14%	64.76%	18.74%
BTB hit rate	49.93%	44.49%	78.26%	27.09%

Branch predictor metrics

    - Below I also show performance speed up from incorporating BTB + G-share(IPC speedup and program delay speedup)

	IPC	Delay(micro seconds)
Coremark	0.2528	1845.12
Mergesort	0.3269	2284.56

FFT	0.3173	2594.53
AES SHA	0.2618	3990.69

Base design metrics

	IPC	Delay(micro seconds)	IPC speedup	Delay decrease
Coremark	0.3601	1287.49	42.44% speedup	30.22% decrease
Mergesort	0.3905	1900.96	19.46% speedup	16.79% decrease
FFT	0.373	2192.96	17.55% speedup	15.48% decrease
AES SHA	0.273	3803.17	4.28% speedup	4.70% decrease

metrics with BTB + G-share integrated

- Conclusions: We realized after collecting these performance metrics that our G-share branch predictor works very well in isolation, but our BTB misses very often. Therefore we believe that the primary bottleneck in our Branch prediction performance was our BTB. to improve this we could have increased BTB size, made the BTB an associative cache, implemented a simple adder to precompute branch addresses, or integrated a return address stack. We realized late into our development that the BTB is really only useful for control instructions where we can't immediately compute the branch address, specifically JALR instructions. We also realized that JALR is mostly used in RET instructions, so if we had more time we would have integrated a return address stack, and experimented with precomputing branch targets.

4.4 Split load store Queue

- Design: Our split load store queue design was based upon the idea of a separate load queue and store queue. We then had a module combining the two that arbitrated who was in control and serving at any given moment in time so that both queues didn't try accessing memory at the same time and cause issues. Furthermore, we gave priority to stores if they were ready to execute. One thing we did to make later checks easier was we had a load store order where any memory instruction was given an unique order representing its original order in terms of being fetched. In terms of loads, we did them out of order in between stores. We accomplished this by going through each of the active entries in the load queue and checking if they were ready to execute and whether they were younger than the current store queue head through the use of the unique memory instruction order discussed above. If this was true, we loaded whatever index met the conditions first, starting from the head. This way, stores waiting to get ready and to become the head of the ROB didn't stall loads that we could execute anyways without issue. Some trade offs for this design are a larger critical path in the load store queue due

to the additional logic necessary to facilitate its operation. However we gain IPC in workloads that have a heavier percentage of memory instructions, especially loads. Workloads with a high percentage of stores and a low number of loads may suffer from this implementation.

- Testing: For the split load store queue we validated correctness by performing DUT testing with targeted load store programs. We made sure that there were no RVFI errors and that the spike logs were correct to confirm correctness. Moreover, we ran all of the given test cases in the same manner, ensuring that no RVFI errors were present and that the spike logs matched. By doing this thorough testing we were confident in our split load store queue's correctness.

- Performance Analysis:

    - On programs like Mergesort with heavier memory instruction composition, we had an improvement in IPC. However general programs like Coremark suffered slowdowns. Additionally, we incorporated performance counters that calculated the average time to complete a load, and we saw a speedup in both cases with both Coremark and Mergesort.

	Coremark_im	Mergesort
IPC improvement with split LSQ	-1.206%	3.2568%
Average Load Time	11.72%	20.91%

Split Load Store Queue metrics

4.5 Benchmark Analysis (Python)

- Design: Automated performance profiling and bottleneck identification using Python scripts. Given a DIS file, it determines the type of each instruction. Then using an ELF file, it determines the number of branches that actually happen, and if they are taken or not.

- Testing: Ran on .dis and .ELF files of shorter programs, and made sure that ILP, instruction, etc were accurate, and then ran on whole benchmarks

- Performance Analysis: Provided insights into performance tuning and feature impacts, allowed us to determine the number of reservation stations to use, and other parameters. Below is a sample output of the files:

RISC-V Disassembly Benchmark Analysis mergesort

Total Instructions: 1446

Instruction Mix:

Load Instructions: 30.36%

Branch Instructions: 3.04%

Arithmetic Instructions: 21.92%

Store Instructions: 36.93%

Other Instructions: 7.75%

Branch Statistics:

Branch Instructions: 3.04%

Conditional Branch Instructions: 1.38%

Memory Operations:

Load Instructions: 30.36%

Store Instructions: 36.93%

Dependency Analysis:

Average Dependency Distance: 4.01

RISC-V Disassembly Benchmark Analysis FFT

Total Instructions: 1446

Instruction Mix:

Load Instructions: 30.36%

Branch Instructions: 3.04%

Arithmetic Instructions: 21.92%

Store Instructions: 36.93%

Other Instructions: 7.75%

Branch Statistics:

Branch Instructions: 3.04%

Conditional Branch Instructions: 1.38%

Memory Operations:

Load Instructions: 30.36%

Store Instructions: 36.93%

Dependency Analysis:

Average Dependency Distance: 1.18

RISC-V Disassembly Benchmark Analysis Coremark_im

Total Instructions: 4842

Instruction Mix:

Arithmetic Instructions: 42.63%

Load Instructions: 13.92%

Store Instructions: 26.81%

Other Instructions: 9.21%

Branch Instructions: 7.43%

Branch Statistics:

Branch Instructions: 7.43%

Conditional Branch Instructions: 5.14%

Memory Operations:

Load Instructions: 13.92%

Store Instructions: 26.81%

Dependency Analysis:

Average Dependency Distance: 2.00

Spike Log Branch Analysis mergesort.elf

Total Branch Instructions: 4405

Taken Branches: 2091

Not-Taken Branches: 2314

Spike Log Branch Analysis FFT.elf

Total Branch Instructions: 13825

Taken Branches: 13633

Not-Taken Branches: 192

Spike Log Branch Analysis Coremark_im.elf

Total Branch Instructions: 55170

Taken Branches: 29516

Not-Taken Branches: 25654

4.6 Return Address Stack (unintegrated in CPU)

- Design: We designed the return address stack(RAS) initially when planning out advanced features, but never got to integrating it, since we switched focus to integrating the BTB. In our design we basically push the return address onto the stack if the processor commits a CALL instruction. We defined a CALL instruction to be a JALR instruction where the destination register is X1(register 1). When we looked at multiple disassemblies of C files with function calls, we found every CALL instruction to follow this pattern. Whenever the processor decodes a RET instruction we just pop off the top of the stack and forward that as the branch target. We defined RET instructions as JALR instructions with destination register X0, and with the branch target being held in register X1(the return address register according to the RISC-V application binary interface).
- Testing: I wrote a design under test test case, where I drove all the input signals and observed the output signals. I tested multiple CALL instructions and then multiple RET instructions to see if the RAS would return the correct return address.
- Performance Analysis: This advanced feature allows the processor to properly process JALR instructions, and follow C calling convention.

---

6. Conclusion

In this project we were tasked with creating an Out-of-Order processor tailored for the RISC-V 32IM ISA. We were assigned this to gain a deep understanding of advanced computer architecture concepts, and as a design challenge to design the most performant and efficient microprocessor. As discussed earlier, Out-of-Order microarchitectures are ubiquitous with modern day processors, and throughout this project we sought to emulate and improve on designs taught to us in class. I believe we were successful in this endeavor choosing to go with an ERR architecture, and the integration of several advanced features such as a nextline/stride prefetcher, G-share branch predictor, branch target buffer, and a split load store queue. Throughout this we also learned a lot to improve upon in future designs, such as incorporating performance testing earlier on to find bottlenecks earlier on and making each component more modular to make integration more streamlined. Overall this project will serve as valuable experience setting us up for success in our future careers in computer architecture.