CSEE W4824 Final Project:
P6-Style Out-of-Order RISC-V Processor

Group 04, Cache Course:
Diwa Bhusal, Mateo Nery, William Wang, Kuan Zhang
Github: CS4824-Final-Project-Cache-Course

---

1. Introduction

This is the project report for a P6-style Out of Order Processor that handles RISC-V Instructions, designed by Diwa Bhusal, Mateo Nery, William Wang, and Kuan Zhang as part of the CSEE W4824: Computer Architecture course at Columbia University. The report that follows will describe the implementation, features, and performance of our Processor, as well as attempted features, challenges, and future improvements for our design.

2. Design Details

The processor follows a single-issue, out-of-order P6 architecture with a top module that wires together all six stages of the pipeline: Fetch (IF), Dispatch (ID), Issue (IS), Execute (EX), Complete (CP), and Retire (RT). Below is a diagram detailing the structure of our processor pipeline:

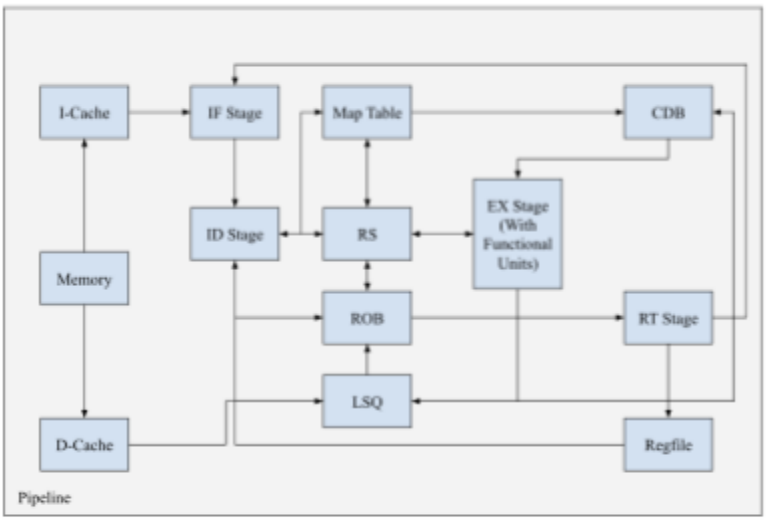

Figure 1. Architecture Block Diagram

2.1 Fetch Stage

The fetch stage (IF) fetches one instruction from memory in order, via the Instruction Cache. In the case that the I-Cache is unable to return valid data to the fetch stage, a stall signal is asserted to pause the pipeline downstream. The stage outputs a packet containing the fetched 32-bit instruction, the current and next PC values, and whether the packet contains valid information. This packet is then shared down the pipeline to the dispatch stage.

2.2 Dispatch Stage

The dispatch stage (ID) is responsible for decoding the instruction packet sent from the fetch stage and assigning correct inputs to the Map Table, LSQ, Re-order Buffer (ROB), and Reservation Station (RS). This stage then identifies any structural hazards detected in the ROB, RS, or LSQ, and forwards those signals downstream appropriately. When entering this stage, an instruction is assigned a unique tag, typically a ROB tag, that acts as a physical register identifier. The tag replaces the destination architectural register and allows out-of-order execution by eliminating false dependencies (WAR and WAW hazards). Each source operand is resolved instantaneously from available values or preceded by dependency tags until the value is broadcast using the common data bus (CDB). Operand readiness is dynamically monitored—when the CDB broadcasts a result, all RS entries waiting for that tag latch the corresponding value and set their readiness flags. Instructions in the RS are always filtered through each cycle. If both operands are available and the required functional unit is free, an instruction is marked for issue. At the same time, each instruction is loaded into the reorder buffer in program order. Upon an instruction completing execution and its result being placed onto the CDB, the ROB marks the corresponding entry as complete. Instructions retire in a first-in-first-out fashion: the head pointer of the ROB advances when the oldest instruction is both valid and complete. In the event of a branch misprediction, ROB has been written to trigger a pipeline flush and recover register mapping state to a checkpointed state stored at dispatch time. Below is a description of the functions of each unit within the dispatch stage:

2.2.1 Re-order Buffer

The Re-order Buffer (ROB) contains all incoming dispatch information, represented as a 32-entry circular queue. It is responsible for maintaining in-order instruction retirement using the head pointer to indicate the next instruction to retire, and the tail pointer indicates the next entry. Upon receiving valid data from the dispatch stage, the ROB inserts the new instruction into the tail and tracks the destination register, opcode, and computed result when available. Upon

instruction completion, the ROB receives the result broadcast from the Common Data Bus (CDB), and the entry is marked as ready, to then retire the instruction once it reaches the head of the ROB. The ROB is further integrated with the LSQ to track store instructions, marking whether stores are ready to be committed. Using an extra bit to track ROB full logic, the ROB can send out a stall signal to the pipeline to halt incoming dispatches when the buffer is at capacity.

2.2.2 Map Table

The Map Table contains 32 entries, enabling register renaming by containing ROB tags to enable out-of-order execution. Upon dispatch, the map table assigns a new ROB tag to the received data, marks it as not ready, and indicates the register is now mapped to an in-flight instruction. The map table further listens out for a CDB broadcast of a completed instruction, to then mark the instruction as ready through updating the ready_in_rob bit and passing this information to the ROB to indicate instruction readiness. Upon retirement, the map table clears the entry associated with the retired instruction.

2.2.3 Reservation Station

The Reservation Station (RS) holds decoded instructions awaiting their operands, and enables dynamic scheduling and register renaming. Upon dispatch, an instruction is loaded into the RS if an entry is available. The RS stores the instruction with the associated ROB tag if the operands are not yet available. The RS listens for the CDB to broadcast the operands for an entry, and allows for the issue stage to determine if an instruction is ready to be passed to execution. Once an instruction has been issued and accepted by its corresponding Functional Unit (FU), the RS entry is cleared. Our initial working pipeline utilized a single-entry RS due to integration issues, but was later updated to a multi-entry version with entries reserved for specific functional units. We later describe this situation in the Challenges section of our report.

2.3 Issue Stage

The issue stage (ID) is responsible for selecting a ready instruction from the Reservation Station and issuing it to an available FU. The stage receives valid instructions from the RS, determines if a FU is available, and stalls in the case that no FU is available. The issue stage also informs the reservation station which entry has been issued. This element is further utilized in our multi-entry version of the pipeline, ensuring that multiple entries can store instructions to be executed.

2.4 Execute Stage

The execute stage (EX) is responsible for all logical and arithmetic operations via the FUs. Our initial design utilized a single ALU and the Conditional Branch module for all computations, but was later expanded to include an additional ALU and an 8-stage Multiplier. In this expanded version of the execute stage, both ALUs were responsible for branches and arithmetic, except for multiplication. The Multiplier is capable of handling both signed and unsigned multiplication, and its pipelined nature allows a new input every cycle. When a unit finishes execution, the computed result is stored in a packet that is then sent to the complete stage and is eventually broadcast from the CDB. If the instruction is a memory operation, it is instead routed to the LSQ for handling. Our initial pipeline used a single signal to track whether the ALU was busy and required stalling, while the updated pipeline was able to track the status of all three units and determine which instructions were ready to send to the complete stage, with arbitration logic being used to prioritize multiplication instructions over ALU results when forwarding. The differences in the execute stage will further be discussed in our Challenges section of the report.

2.5 Completion Stage

The completion stage (CP) receives execution results from both our ALU pipeline and LSQ, and arbitrates which result gets broadcast via the Common Data Bus. The Common Data Bus (CDB) takes results from the functional units and gives priority to the LSQ when broadcasting results.

2.5 Retire Stage

The retire stage (RT) is responsible for committing completed instructions in program order, and writing back to the register file, as well as handling branch mispredictions and exceptions. The retire stage receives input from the ROB on a completed instruction and determines if the instruction is a memory operation or a branch instruction. If the instruction is a memory operation, the retire stage validates the instruction and coordinates with LSQ for final memory commit. If the instruction is a branch instruction, the retire stage determines if the instruction was a mispredict and asserts signals to ensure the pipeline is cleared. If the instruction is neither a branch nor a memory instruction, the retire stage writes the result to the register file and ensures the ROB and map table are cleared accordingly.

3. Advanced Features

Beyond the baseline out-of-order pipeline, our design implemented a set of advanced architectural features intended to improve the performance and correctness of the processor. Below, we detail those features:

3.1 Load-Store Queue

Our Load-Store Queue (LSQ) was implemented to handle in-flight memory operations in the out-of-order pipeline. Our LSQ is unified and contains 16 entries, each storing either a load or a store. Furthermore, the LSQ was built with both blocking and non-blocking configurations, though issues with the non-blocking configuration led our team to complete testing with the blocking implementation. Using a circular queue structure similar to the ROB, each entry in the LSQ tracks the validity of its store data, the address, and retirement status for stores. Memory operations at the head of the queue are only retired once marked as ready. The LSQ regulates interaction with the D-Cache using a memory arbitration scheme, and our LSQ allows for speculative loads to post results to dependent instructions through the CDB. This speculation allows the execute stage to avoid stalling due to unresolved memory instructions, to improve CPI.

3.2 D-Cache Improvements

Our directly mapped, non-blocking D-Cache implementation was updated with a Write-back policy with dirty tracking to more effectively handle memory instructions with the LSQ. Incoming memory addresses are split into their tag, index, and offset fields, and the D-Cache can evict dirty lines upon mispredicts. Missed requests are tracked using the tag-to-address mapping table to avoid redundant loads and overwrites. If a cache line is marked as dirty and a new line maps to the same index, the dirty line is written back to memory before replacing it, and the D-Cache can use write-back upon halts and misses to ensure dirty data is flushed.

3.3 Speculative Execution and Branch Misprediction Recovery

Although our branch prediction implementation was disabled due to challenges upon program testing, the pipeline supports speculative execution with a misprediction recovery mechanism in the backend. Mispredicted branches are trapped by the retire stage, which checks predicted resolutions against expected ones. A Flush signal is sent to the pipeline upon misprediction, allowing the pipeline to recover and resume correct execution.

4. Attempted Features

Due to time constraints, some advanced features were attempted but were unable to be fully integrated into our pipeline. However, we believe that the effort put into these features makes them worth mentioning, and we describe those features and future opportunities for development below.

4.1 Fully Featured Branch Prediction Stack

Beyond the initial branch prediction requirements, a more advanced branch prediction was attempted using a Branch Target Buffer (BTB), 2-bit saturating counters, and a Return Address Stack (RAS). This work can be found in our repository under the branch_predict branch. Due to the complexity of recovery logic and control path complexity with these advanced features lying outside our time constraints, this attempt was not integrated into the final pipeline. Future work could involve fully integrating the improved BTB and RAS, with more sophisticated branch prediction schemes and control flow handling.

4.2 Z-Cache

Beyond the I-Cache and D-Cache, our team had planned and built a Z-Cache module to improve the memory hierarchy of our model, via implementing a more associative, flexible, and cache-efficient data cache based on the Z-Cache architecture described in *"The ZCache: Decoupling Ways and Associativity"* by Sanchez and Kozyrakis (MICRO 2010). Although this module was not fully integrated due to time constraints and cache challenges during testing, it demonstrates our forward thinking and understanding of advanced architectures, and would be implemented in future updates to the design. The Cache would be designed with 8-way associativity, non-blocking, least recently used eviction policy, and a 3-way H3 universal hash family for decoupled set indexing, implemented in the h3hash module in our repository.

5. Challenges and Bugs

During the final development and testing phase of the processor, many bugs and challenges were encountered. We discuss some of the challenges, bugs, and potential solutions and improvements below.

5.1 Multi-Entry Reservation Station and Functional Unit Challenges

During our final integration for testing, our team encountered challenges with issuing instructions due to signal mixups in our multi-entry reservation station and multiple functional units. Given the limited time for testing and integration challenges, our team decided to prioritize successful program completion over the inclusion of these features. However, shortly after submitting our final processor design, our team was successful in uncovering the source of the errors within our multi-entry model, namely a signal mixup within the stalling logic of the pipeline and a clock desync within the multiplication module. We were able to fix these errors and managed to complete some testing and analysis with this updated pipeline, which will be discussed later in the performance analysis section of the report.

5.2 D-Cache Arbitration

While integrating our I-Cache and D-Cache with the memory module, one challenge we encountered was the timing desync between modules. Namely, the memory module request and response were not available to be seen by both caches, resulting in challenges with arbitration and timing. The timing aspect was particularly challenging for our team, as while most module features within the caches were based on the positive edge, the arbitration required checking values on the negative clock edge, creating issues with several C programs due to this timing desync. In future implementations, we hope to resolve these issues to avoid errors in the memory due to simultaneous cache requests based on desync.

5.3 D-Cache Write-back Halt Timing

While our D-Cache structure had write-back implemented, halting occasionally caused issues within C programs due to write-back on dirty cache lines, causing errors. This resulted in some C programs with mismatched memory outputs, which is a bug we plan to fix in future implementations.

Overall, we believe that while our pipeline faced several problems, the work that we were able to complete indicates our ability to resolve these challenges in the future.

6. Synthesis

Our team used Synopsys to check that the processor was synthesizing properly while minimizing time spent on issues in synthesis. We implemented the following synthesis test strategy: Individual modules were tested for synthesis initially, with a test_synthesis.sh script created to test the modules. The script first checked whether the module passed its testbench and created its .out file. Then, the module was synthesized to compare the synthesized output against the testbench, and Make Slack was called to check timing violations.

7. Performance Analysis

For our testing strategy, our team began with the smaller module strategy described above, and several modules were grouped for testing. This included the ROB and RS, as well as the dispatch stage, ROB, and map table, using combinational test benches to observe their combined integration. With numerous debug wires and display statements used to dump outputs as well as data between units, our team was able to improve our ability to identify input mismatches or internal unit errors during integration testing. For smaller module tests, the VCS waveform compiler was used, but due to the large number of signals within the pipeline, we relied mostly on internal debug statements for larger integration tests. After our initial testing was successful, we adjusted the cycle timing to the required values to ensure proper functionality for programs. Below is an analysis of the initial version of our pipeline, as well as the testing for our updated version and a comparison of performance between the two versions.

7.1 Initial Pipeline CPI and D-Cache Utilization

Below is the analysis of the initial processor metrics, covering its CPI, D-Cache hit and miss rates, LSQ utilization, and ROB utilization. Notably, our CPI average was 10.3, with the best CPI being 3.8 on mult_no_lsq.s and its worst being 30 on halt.s. Notably, this average CPI was likely caused due to the large spikes in CPI for halt.s and matrix_mult.s. Figure 2. Covers the CPI results for every test program.

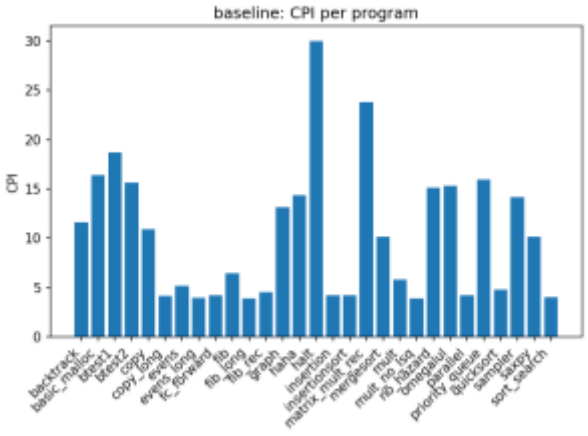

Figure 2. Baseline CPI per Program

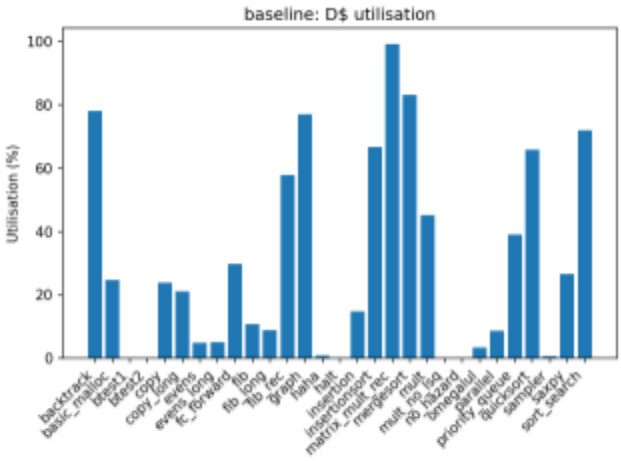

Figure 3: Baseline D-Cache Utilization per Program

Additionally, we considered the correlation between CPI and D-Cache utilization as a likely source of outliers, and Figure 3 contains our findings on the D-Cache utilization per program. Notably, several of the higher spikes in CPI were correlated with D-Cache utilization, and we investigated this correlation by observing D-Cache hit and miss rates as shown in Figures 4 and 5, respectively.

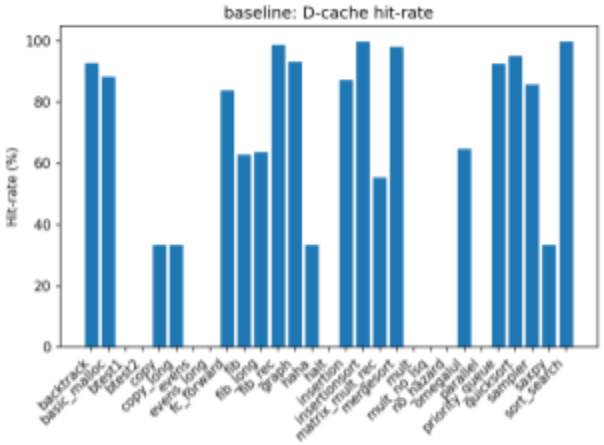

Figure 4. Baseline D-Cache Hit-Rate per Program

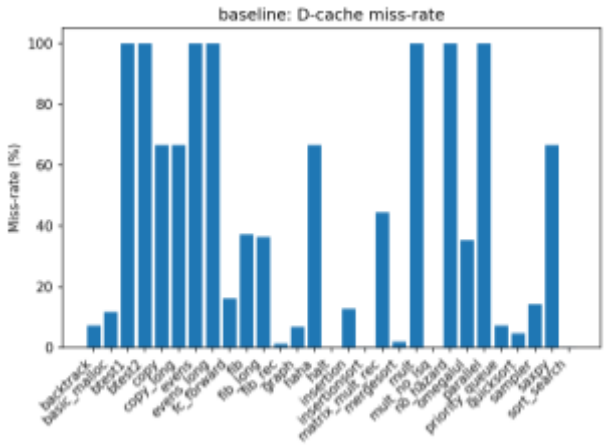

Figure 5. Baseline D-Cache Miss-Rate per Program

From figures 4 and 5, the mean was 51.5% hit rate, with the best being 99.8% in search_and_sort.s. We measured the D-Cache utilization based on the percentage of space in the D-Cache that was occupied by valid cache lines. A similar approach was used for measuring LSQ and ROB utilization, shown in Figures 6 and 7, respectively.

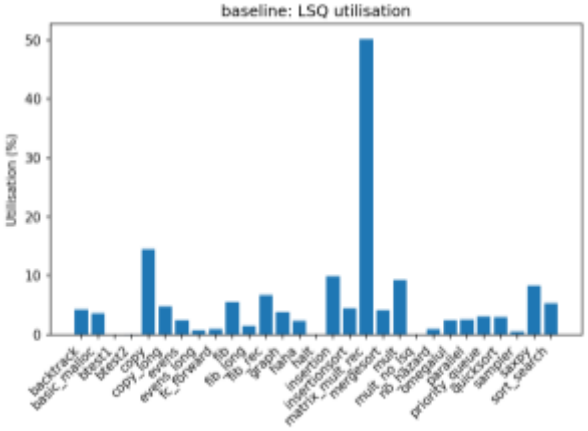

Figure 6. Baseline LSQ Utilization per Program

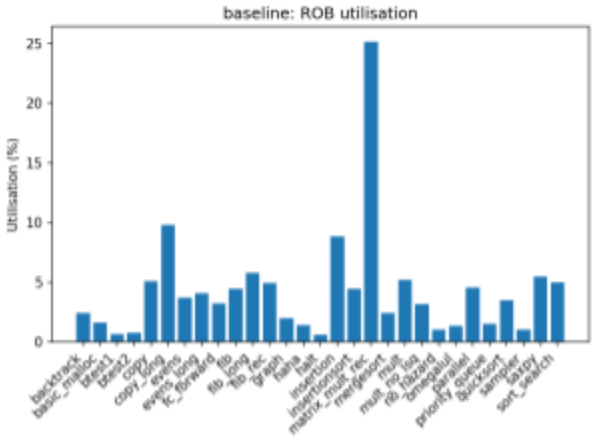

Figure 7. Baseline ROB Utilization per Program

From both Figures 6 and 7, mergsort.s had a notably high cache usage, which correlated with a higher CPI. This correlation indicates that the high usage of caches may have meant a large number of stalls due to both LSQ and ROB being full. In the future, we would further like to explore how adjusting the sizes of both LSQ and ROB will affect program results.

7.2 Initial Pipeline CPI and D-Cache Utilization

Additionally, we were able to obtain test data for several programs with the updated pipeline that utilized multiple functional units and multiple RS entries, and compared several metrics between the baseline pipeline and the updated pipeline. In Figure 8, we compared the CPI of both pipelines, as displayed below. For the updated pipeline, the average CPI was 10.6, with the best being 3.8 for evens_long.s and worst being 30 for halt.s. Notably, the height of halt.s' CPI indicates that other spaces in the pipeline would need optimization to reduce CPI, outside of the functional units. However, we did notice several programs observed lower CPI, and several programs increased their CPI. This may be because the pipelined multiplier increases the number of cycles per multiplication. However, the pipelined multiplier's benefit comes from its ability to reduce requirements for cycle time, and future tests will focus on observing the new optimal clock cycle time.

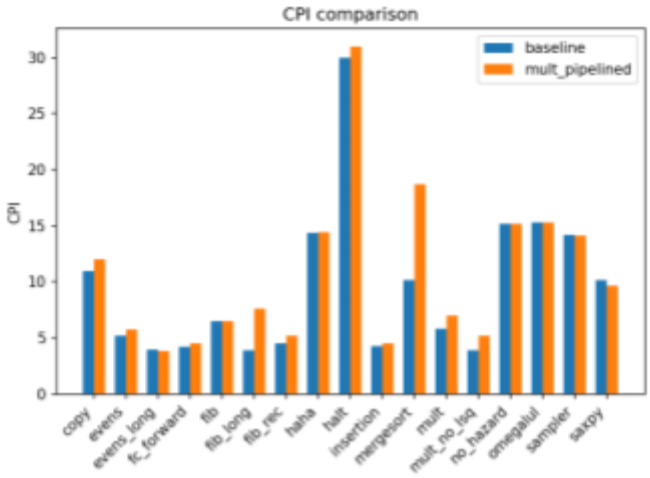

Figure 8. CPI Comparison between Baseline and Multi-FU Pipeline

We further tested the LSQ and ROB usage for the updated pipeline and compared it against our baseline. The results of that testing are shown in Figures 9 and 10 below. In both cases, we observed higher utilization of the ROB and LSQ, confirming our hypothesis that the longer number of cycles used for the pipelined multiplier resulted in higher occupation of the LSQ and ROB, resulting in longer wait times for instructions and likely increasing CPI.

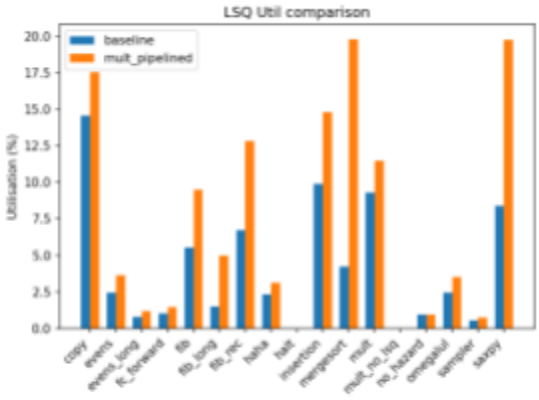

Figure 9. LSQ Comparison between Baseline and Multi-FU Pipeline

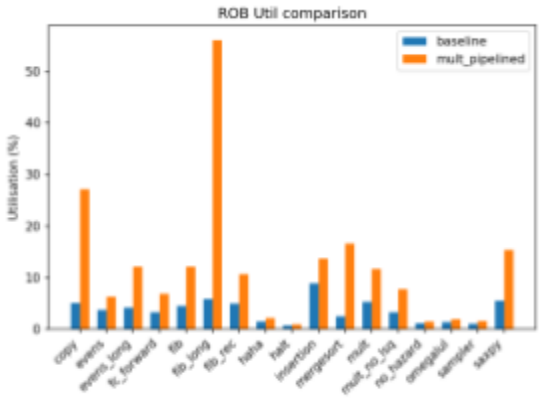

Figure 10. ROB Comparison between Baseline and Multi-FU Pipeline

8. Future Work and Involvement

8.1 Future Work

While our team was not able to complete all the features we had initially planned for our design, we look forward to working on the following features and improvements in the future:

- Fix .c program bugs: We hope to debug several issues occurring during the .c program testing, and aim to work on resolving some of the timing issues we identified that may be the source.

- Full integration of attempted features: Our team worked hard on several advanced features, such as improved caches and advanced branch prediction methods. The features we described in the Attempted Features section of this report would be some of the first steps to take in improving the performance of our processor

- Superscalar Implementation: At the beginning of this project, our team had hoped to complete a superscalar implementation of our design. We had built initial packets and stages with this feature in mind. However, the complexity of superscalar and the timing constraints we encountered required us to discard this feature. In the future, we would aim to implement a 2-way superscalar design into our processor and update our modules as needed.

8.2 Involvement

With a smaller team that had no prior SystemVerilog experience, we knew this project would be challenging. Yet during the timeline of our project, our team faced several unexpected challenges both inside and outside the scope of this course that further pushed our team to the limits of our capabilities. Nevertheless, our team persevered and built a design we are proud of, and truly put incredible effort into making this project happen. While it is difficult to describe the exact percentage of involvement on different features of the project due to several early modules being full group efforts, such as the ROB and RS, below is a brief detailing of the strengths each teammate contributed to:

- Diwa Bhusal contributed significantly to the testing, integration, and synthesis checks for our design and made significant contributions to the ROB, map table, and reservation station.

- Mateo Nery made the primary designs of the complete, retire, fetch, and issue stages, as well as the ROB, RS, and implementation of multiple FUs into the execute stage, as well as updating the multiplier for unsigned and signed instructions.
- William designed the LSQ and D-Cache and made major progress on the integration of the pipeline, especially with the multi-entry RS and multiple functional units. He contributed significantly to the debugging and testing of full pipeline programs.
- Kuan Zhang made significant contributions to the execute and dispatch stage, and assisted significantly in helping integrate the full pipeline.

9. Conclusions and Acknowledgements

We have implemented a P6-style Out of out-of-order processor that handles RISC-V Instructions. We were able to pass all assembly cases and several .c programs, indicating that most of our design features were functional, but with some bugs still to be found. Additionally, our team at the time of submission was still debugging the multi-entry RS and multiple functional unit design, but was able to complete it shortly after submission and presented initial data findings above. In the future, we hope to continue debugging the design, implementing the attempted features, and adding new advanced features.

We would like to express our gratitude to Professor Khan, Matthew Weingarten, and Michael Grieco for their guidance and support throughout the project.