EECS 470: Computer Architecture
Tomasulo's Scholars Project Report

GROUP #10:

Zach Milan (zmilan@umich.edu)
Harshit Nanda (hnanda@umich.edu)
Enrique Orozco Jr. (orozcoen@umich.edu)
Maatla Sefawe (maatla@umich.edu)
Ayush Vashi (aavashi@umich.edu)

April 23rd, 2024

Contents

1 Introduction

This report describes the Out-of-Order processor designed by Tomasulo's Scholars based on the RISC-V Instruction Set Architecture for the EECS 470 (Computer Architecture) project. This processor is a 3-way superscalar, R10K based out-of-order processor. This report and processor is the culmination of the collective expertise and collaborative effort of Tomasulo's Scholars. In this report, we will provide a processor overview and elaborate on critical details such as design decisions, processor performance, team methodology, and areas of work.

1.1 Project Background

University of Michigan's EECS 470 (Computer Architecture) course gives students a detailed understanding of how computers are designed and implemented, culminating in a semester-long project where students team up and design an out-of-order processor. An out-of-order processor is a processor that implements out-of-order scheduling. Out-of-order scheduling is a technique that allows for computer instructions to be executed out of their original program order but committed in-order.

There are numerous performance benefits to out-of-order processors, and most modern processors utilize an out-of-order design. Therefore, engaging in a project of this magnitude and nature has allowed us to gain an understanding of what designing a modern processor is like and the challenges it presents. Alongside the real-world experience, this project has allowed us to build on our understanding of the material taught in lectures as we've had to implement it and overcome various challenges throughout the collaborative design process.

1.2 Base Requirements

For this project and our processor, our base requirements were to implement an out-of-order processor following either the P6 or R10K scheme. Additionally, we needed to have multiple functional units with varying latencies, implement an instruction cache and a data cache each limited to 256 bytes, and at minimum, implement a branch target buffer and a bimodal branch predictor.

1.3 Advanced Features

Besides meeting the basic project and processor requirements, we were tasked with selecting and implementing advanced features. Considering the performance trade-offs and time investment involved, we carefully deliberated and settled on the following features:

- 3-way Superscalar execution, Load-Store Queue with store-to-load forwarding, Speculative and retired Gshare branch prediction, GUI debugger, Instruction prefetching, 2-way set-associative non-blocking data cache, and a 2-bank non-blocking instruction cache with prefetching.

2 Design and Implementation

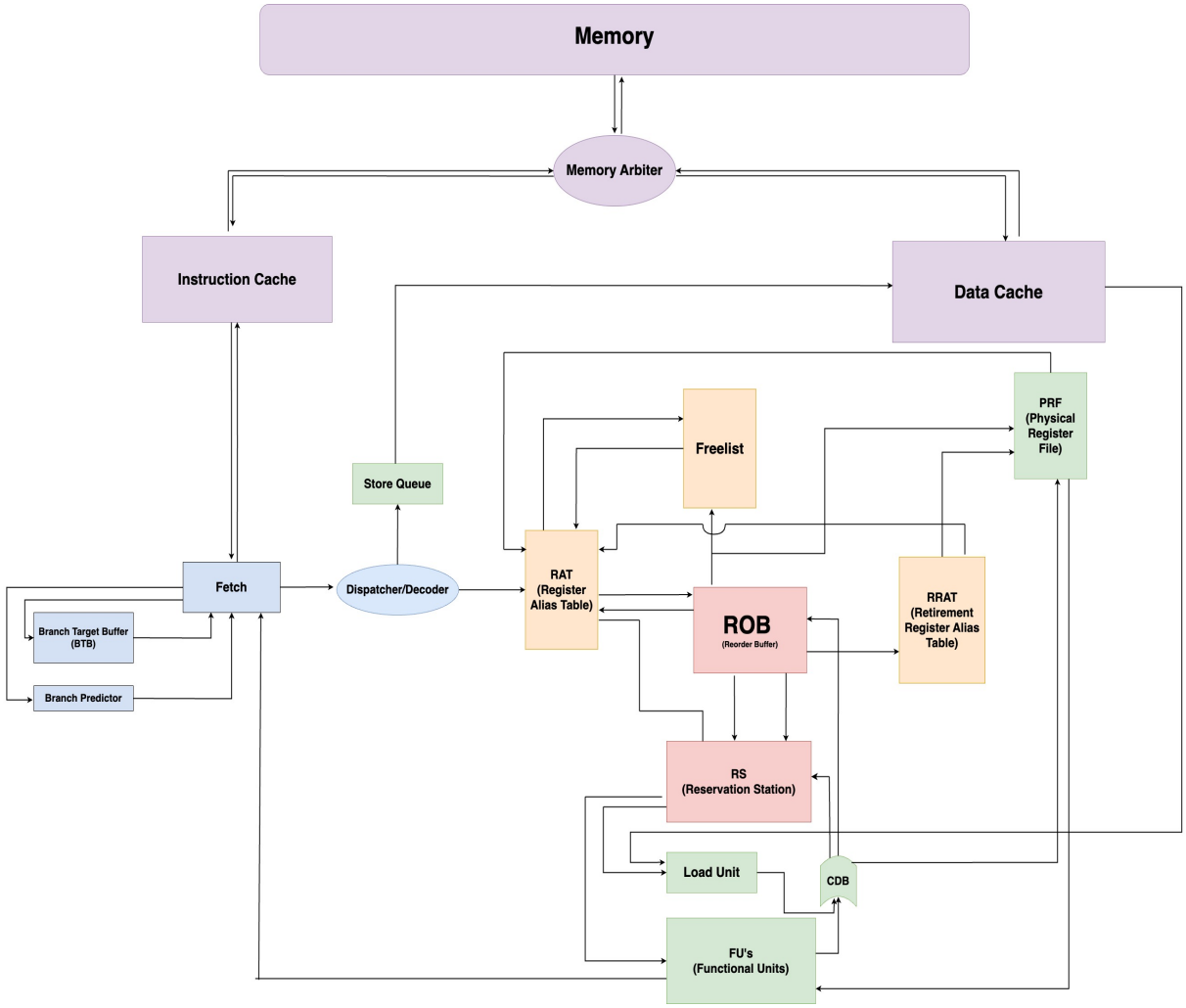

Figure 1: High-Level Overview of Our Processor

2.1 Functional Units

Our design included 3 ALUs, 1 mult unit (4 stages), 1 branch unit, 1 load address calculator, and 1 store address calculator. We only had one mult unit because including more could've put the mult units on the critical path.

2.2 Reservation Station

To determine the allocation of instructions to specific entries within the RS, we utilized a provided priority selector, which effectively assigns incoming instructions to available RS Entries.

The RS actively listens to the CDB to monitor the readiness status of instruction source registers.Specifically, for load operations, the RS only sets a load as ready when the associated stores, dispatched prior and currently pending the store queue, have calculated their addresses. This ensures that loads are only issued when the necessary data is available, mitigating potential data hazards and ensuring program correctness.

The issue logic within the RS leverages both the information from the CDB and the FUs busy bits to make informed decisions regarding instruction issuance. By considering the latency times of each functional unit, the RS predicts the availability of FUs for accepting new instructions. This proactive approach optimizes instruction issuance, reducing latency in the issue stage.

We carefully selected an RS size of 16 entries. Increasing the RS size beyond this threshold would have introduced critical path issues, particularly concerning the multiplier FU.

2.3 Reorder Buffer

The reorder buffer (ROB) serves to track the instructions in program order and can retire up to three instructions per cycle. The ROB Size was finalized to be 32 entries. Although our ROB occupancy was around 18 entries on average, we chose 32 to minimize structural hazards for programs we were able to get a high branch prediction rate on programs like *insertionsort.c*.

Within our processor architecture, the Reorder Buffer (ROB) assumes a pivotal role in maintaining program order and facilitating efficient instruction retirement. It serves as a crucial component in the out-of-order execution paradigm, allowing instructions to be executed out of order while ensuring they are retired in their original program order. If the instruction retired is a branch instruction, and was a mispredict, it is sent to the BTB and the direction predictor to update with the correct address/taken/not taken

2.4 Map Table, Freelist and RRAT

The freelist, a FIFO buffer, features three pointers: Head, commit head, and tail. The commit head moves when old PRNs are given from the ROB. The head pointer movement depends on the number of free PRNs accepted by the RAT. The tail moves depending on the speculative ARN mapping to PRNs. RRAT updates its ARN to PRN mapping when the ROB retires 3 instructions at the head. Instructions without destination registers are assigned ARN 0 in the RAT.

Module	Size
Arithmetic Logic Units (ALUs)	3
Multiplier	1
Branch Unit	1
Load Unit	1
Store Calculator	1
Reorder Buffer (ROB) Entries	32
Reservation Station (RS) Entries	16
Instruction Buffer Entries	9
Branch Target Buffer (BTB)	64
Pattern History Table (PHT) Entries	32
Global History Register (GHR)	5
Store Queue Entries	5

Table 1: Final Module Sizes

The RAT is responsible for register renaming in order to remove false dependencies. It pulls PRNs off the freelist and maps them to destination ARNs. For source registers, the PRN is pulled directly from the map table. Since we are three-way superscalar, we need to forward PRNs from older to newer instructions within the same dispatch packet in the case that there is a true dependency. Once the register renaming occurs in the RAT, the source and destination PRNs are passed to the ROB, RS, and Store Queue combinationally.

3 Advanced Features

3.1 3-Way Superscalar Design

Our primary advanced feature, which influenced numerous subsequent design decisions, is our processors 3-way superscalar-ness. This design allows for simultaneous execution of up to three instructions, significantly enhancing performance and throughput compared to a non-superscalar design. By leveraging instruction-level parallelism (ILP), the processor dispatches instructions to multiple execution units for parallel execution. This not only maximizes hardware resource utilization but in general can ultimately reduce program execution time in comparison to a non-superscalar designs.

3.2 Instruction Cache and Fetching

The project specification dictates that our caches, both instruction and data, are limited to 256 bytes. Given that memory is fixed at 8 bytes and each RISC-V instruction occupies 4 bytes, our instruction cache can hold a maximum of 64 instructions at a time. Given these constraints, when

selecting memory hierarchy improvements for the instruction cache, we settled on implementing a dual-banked non-blocking prefetching instruction cache.

Our choice of a dual-banked design for the instruction cache was primarily influenced by 'Data Caches for Superscalar Processors' by Toni Juan, Juan J. Navarro, and Olivier Temam. Within this paper, they discuss the various implementations of caches within superscalar processors, primarily multi-ported vs. multi-banked. The dual-banked nature enables simultaneous access to two blocks (four instructions), accommodating our processor's degree of superscalar-ness (3), while also avoiding any possible critical path penalty that multiporting may have incurred. This capability is crucial for maintaining efficient instruction fetching, allowing us to fetch up to three instructions at a time. Furthermore, by being able to simultaneously access two blocks within our cache, but only fetching three of them, we had a degree of prefetching that further benefited our performance. The lack of bank conflicts, as a result of only fetching three instructions, also influenced our decision as it streamlined the implementation.

Within the two banks of our instruction cache, we maintain a direct-mapped scheme. We chose this scheme because it streamlined the replacement process and eliminated the overhead needed to track PseudoLRU or LRU bits.

Our fetch stage sends two addresses to the instruction cache at a time: PC and PC+8. These two addresses will always index into different banks. The instruction cache sees these addresses and will send two blocks back to the fetch stage, each with an accompanying valid bit. If the valid bit is 0, a cache miss has occurred. The instruction MSHR sees these same valid bits and addresses. If there are misses, the instruction MSHR registers these misses and provides the data back to the instruction cache once the instruction MSHR receives the corresponding block from memory.

Up to 3 instructions from the instruction cache are placed into the instruction buffer at a time, and the timing of this placement depends on which block of memory comes back first. If PC is a hit in the instruction cache, we will place it (and PC+4, depending on alignment) immediately into the instruction buffer. However, if PC+8 is a hit and PC is a miss, PC+8 (and PC+4, depending on alignment) will wait for PCs block to be fetched from memory, after which point all three are then placed into the instruction buffer in order. This must be done to maintain program order in the instruction buffer, which the dispatcher reads from and places instructions in order into the ROB.

3.3 Data Cache

Our data cache(DCache) is a 2-way set associative cache to increase our data cache hit rate and decrease the frequency of accessing memory compared to a direct mapped cache. Although a four-way set associative cache would've likely had a better hit rate, we decided that the time required to implement pseudo-LRU was better spent elsewhere. Our DCache has three ports: a load port, store

port, and a port for data blocks from memory. It can send both a miss on a load and a miss on a store to the MSHRs in one cycle. Our DCache is a write-through allocate-on-write cache. It will write a block to memory as soon as a hit on a store occurs. We decided to do write-through instead of write-back due to the simplicity of keeping memory updated with current data with the write-through scheme.

3.4 Miss Status Handling Registers and Memory Arbiter

In order to handle multiple outstanding misses, we implemented multiple miss status handling registers (MSHR's) that allow our processor to queue miss requests to memory. This allows for data to return to the processor a lot faster, as the caches don't need to wait 100ns for data to return to accept another request for data.

We implemented separate MSHR controllers for the ICache and the DCache, since the implementations varied due to the data MSHR (DMSHR) controller needing to handle stores to memory. For each register, we implemented a state machine with four states to keep track of whether a miss is waiting for data from memory, and whether a block needs to be sent back to the cache or memory. For load misses in both the instruction and data cache, data is sent back to each cache once their respective MSHR's receive the block back from memory. For store misses in the data cache, the associated DMSHR first performs a load on the relevant block from memory, modifies the loaded block with the relevant store data, then stores the modified block back to memory and allocates a line in the data cache as well (enforcing the write-allocate policy in the data cache).

The top-level MSHR controller for both the IMSHR and DMSHR modules utilizes multiple priority selectors. The first priority selector controls which empty MSHR's receive a new miss. Since two misses can be sent to both MSHR controllers per cycle, this priority selector will select two MSHR's to accept new misses. The second priority selector controls which MSHR is currently allowed to send a request to the memory arbiter. Only one MSHR is allowed to send a request to the memory arbiter per cycle.

The instruction MSHR (IMSHR) controller, DMSHR controller, and DCache can all make separate requests to memory in the same cycle. The IMSHR controller can send one load request, the DMSHR controller can send one load or store request, and the DCache can send a store in the case of a store hit. Therefore, an arbiter is required to select which request to memory is granted. The store hit always has first priority, as stores need to be written back to memory as soon as possible to avoid stalling the store queue. The DMSHR controller has next priority, since the DMSHR controller can also send stores to memory in the case of a store miss. The IMSHR controller has the last priority, since the data memory system needs its data first to ensure that instructions are completing execution as soon as possible.

3.5 Branch Predictor

We implemented a sophisticated branch prediction algorithm that increases our prediction rate. When three instructions are fetched from the ICache, they are sent to three separate identical BTBs and three identical direction predictor modules for parallel access. Each BTB uses the 6 bits from the PC, ignoring the offset, to index into the BTB and then checks whether the tag (the entire PC) matches the PC of the instruction. If it does, and the direction predictor predicts taken, a prediction of the next program PC is made and forwarded to the fetch stage.

The direction predictor, which uses a GShare algorithm, combines a speculative global branch history register (GHR) with the branch instruction's PC to generate an index for the pattern history table (PHT). The PHT then uses this index to predict whether the branch will be taken. When a branch prediction is made, the outcome of that branch is not yet known, but the processor must continue executing instructions to maintain high performance. Therefore, the GHR is updated speculatively based on the predicted outcome of branches. If a prediction is later found to be incorrect (ROB), the speculative entries must be reverted to maintain the accuracy of the register. This process involves restoring the GHR to its state before the incorrect prediction, and this is done by having the retired GHR ,updated by the ROB, be copied to the speculative GHR.

For unconditional branches (jal or jalr), once we determine that they are unconditional, in the first iteration of misprediction, we predict that they will always be taken.

3.6 Load-Store Queue

For our load and store system, we implemented a store queue (SQ) and a load unit. The SQ receives stores upon dispatch and sends the store to the memory command once it is retired from the ROB. Loads that enter the RS must monitor the state of the store queue to ensure the load is only issued to the load unit once every store preceding it in program order has calculated its address.

In cases where a load is dependent on a store currently in the store queue, data forwarding occurs from the store to the dependent load. Stores currently in the store queue will forward their data to a dependent load, with the latest dependent data being sent. The load unit will handle selecting which bytes of the forwarded data to receive based on the load's size and block offset.

3.7 Graphical User Interface (GUI) Debugger

In order to avoid looking through large text dumps of our processor's state over time, we decided to implement a Graphical User Interface (GUI) Debugger to simplify the process of debugging our hardware. Our GUI Debugger displays the both latched and combinational state of important components of our Out-of-Order processor for any given clock cycle in a program we run, and it is easily interacted with through simple keystrokes.

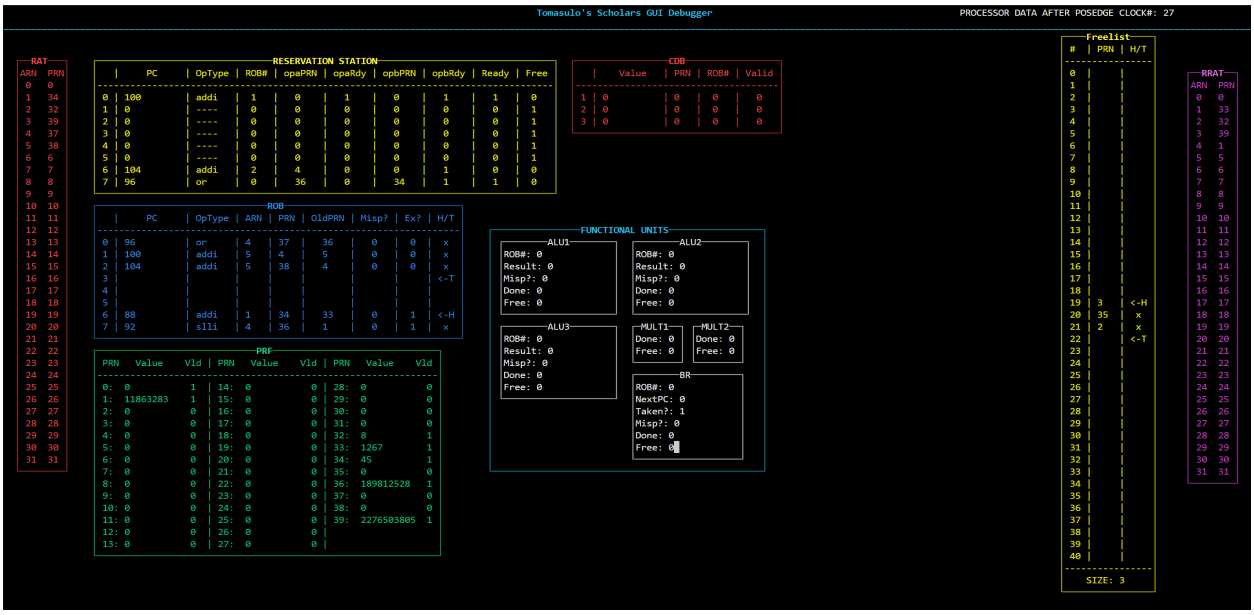

Figure 2: The GUI Debugger in action

The GUI Debugger was written using C++ and the NCurses library, and is approximately 1500 lines long. The following modules were chosen to be displayed based on their importance to the R10K Out-of-Order scheme: the Reorder Buffer (ROB), the Reservation Station (RS), the Register Allocation Table and Retirement Register Allocation Table (RAT & RRAT), the Common Data Bus (CDB), the Physical Register File (PRF), the Register Freelist, and various Functional Units (3 ALU's, 2 Multipliers, 1 Branch Unit). Each structure has a dedicated set of helper functions within the source code of the GUI Debugger. These functions handle tasks such as initializing the associated NCurses window, parsing input from the CPU's output file, clearing outdated data, and updating the window with new information. Figure 2 illustrates the GUI Debugger in operation, displaying different data within each module at clock cycle 27.

The selection of signals for display within each module was based on their significance for debugging purposes. For instance, in the Reservation Station, key information for each entry is included, such as instruction type, destination register, physical register numbers of operands 1 and 2, and whether that entry is ready for issue to a Functional Unit.

We made several key design decisions to enhance the GUI Debugger's user experience. This included adding color differentiations for modules, converting RISC-V opcodes to strings for clearer instruction readability, and integrating dynamic head and tail pointers for FIFO-based structures like the Freelist and Reorder Buffer. Users can easily navigate to any clock cycle with a few keystrokes, and an informative message notifies if a cycle is unavailable. Running the GUI Debugger is also simplified through our makefile with the command "make gui" after regular program simulation finishes.

4 Performance Analysis

4.1 Overall Performance

Our processor achieved 100% correctness upon executing all assembly and C programs provided for testing, attaining a clock period of 12ns. During optimization, we identified multiple wires originating from the Reservation Station as part of our critical path. By reducing the RS size to 8 and the ROB size to 16, we were able to synthesize with a clock period of 11.5ns. However, in our final submission, we opted to stick with larger ROB/RS sizes (Table 1) to prevent them from filling up too quickly, especially when put under pressure by more rigorous test cases. Table 2 below displays the execution times for the provided test programs.

Programs	Cycles per Instruction (CPI)	Instructions per cycle (IPC)	Execution Time (ms)
alexnet	3.14107	0.31836	7.88041
backtrack	3.16371	0.31609	0.27342
basic_malloc	4.34746	0.23002	0.04925
bfs	3.01062	0.33216	0.12587
dft	2.61915	0.38180	1.82340
fc_forward	3.09910	0.32267	0.02477
graph	3.53146	0.28317	0.47149
insertionsort	2.15510	0.46402	1.92953
matrix_mult_rec	1.51622	0.65954	0.39440
mergesort	3.24106	0.30854	0.36851
omegalul	3.97297	0.25170	0.00353
outer_product	1.65440	0.60445	7.77752
priority_queue	4.38694	0.22795	0.07660
quicksort	2.59430	0.38546	0.84164
sort_search	2.51115	0.39822	3.42193
copy_long	0.71791	1.39294	0.00510
fib_long	0.95141	1.05107	0.00728
evens_long	0.99702	1.00298	0.00402
hw4_1	0.36453	2.74324	0.48120
hw4_2	0.40193	2.48803	0.01303
fib	1.86000	0.53763	0.00335
parallel	1.35500	0.73801	0.00325

Table 2: Performance Statistics

4.2 Performance Bottlenecks

Our analysis identified the ICache as the primary bottleneck in our processor. Although we lacked data on ICache hit rates due to our specific implementation, the observed miss rate appeared notably high. This could stem from frequent evictions or high branch misprediction rates, requiring fetching of new instructions from memory.

Notably, the instruction buffer frequently held fewer than three instructions on average across many C programs, as depicted in Figure 3. This suggests a dispatch rate of fewer than three instructions per cycle in the out-of-order system, undermining the efficiency of a three-way superscalar processor. Conversely, programs capable of saturating the instruction buffer exhibited improved IPC. Aggressive prefetching might have bolstered performance by maintaining a fuller instruction buffer.

Furthermore, the single-memory-port restriction emerged as another bottleneck. Consequently, we introduced a memory arbiter to handle multiple requsts to memory. As mentioned earlier, we prioritized DCache requests over ICache requests, thereby impeding the rate of fetching new instructions from memory. Prioritizing ICache requests could have enhanced instruction buffer occupancy and overall system throughput.

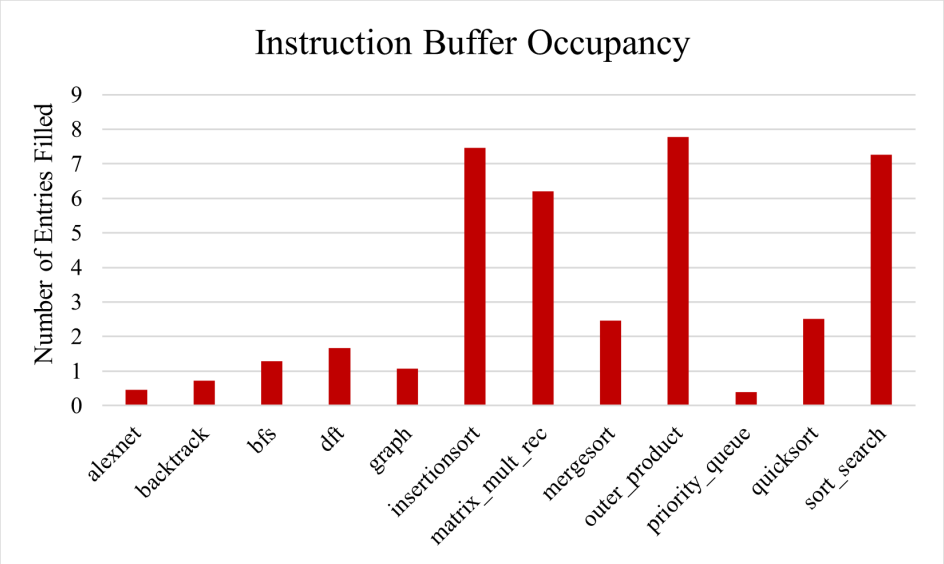

Figure 3: Instruction Buffer Occupancy

4.3 Branch Prediction

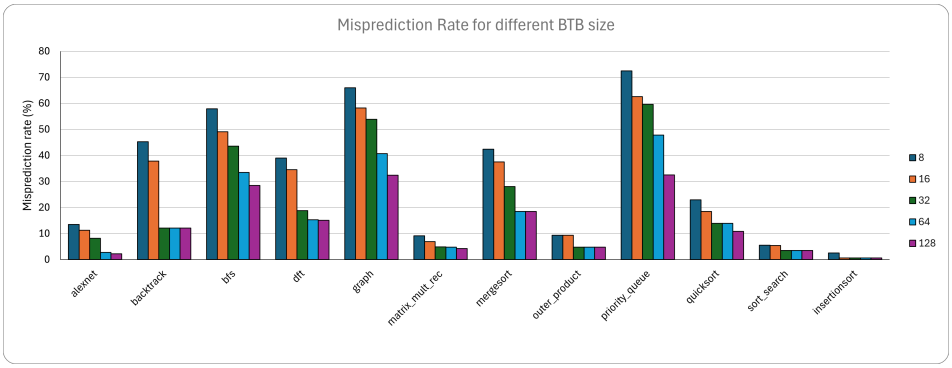

Figure 4: Misprediction Rate with varying BTB Size

Fig. 4 presents the misprediction rate for the given public test cases when tested across different BTB sizes ranging from 8 to 128 entries. The X-axis categorizes the data by different C programs, while the Y-axis represents the misprediction rate as a percentage.

After analyzing the misprediction rates for all the C programs with different BTB sizes, we opted for a BTB with 64 entries. This size provides an optimal balance between hardware resource utilization and prediction accuracy. The chart demonstrates that for most programs, the misprediction rate drops significantly when utilizing a 64-entry BTB, especially in comparison to smaller sizes such as 8, 16, and 32 entries.

Moreover, we found that scaling the BTB size up to 128 entries does not yield a proportional decrease in misprediction rates in most test cases, as observed with benchmarks such as dft.c, mergesort.c, and backtrack.c. The marginal improvements beyond a 64-entry BTB suggest that a 64-entry BTB suffices to capture the majority of the locality and branching patterns, negating the need for the added hardware expense and increased complexity of a larger BTB.

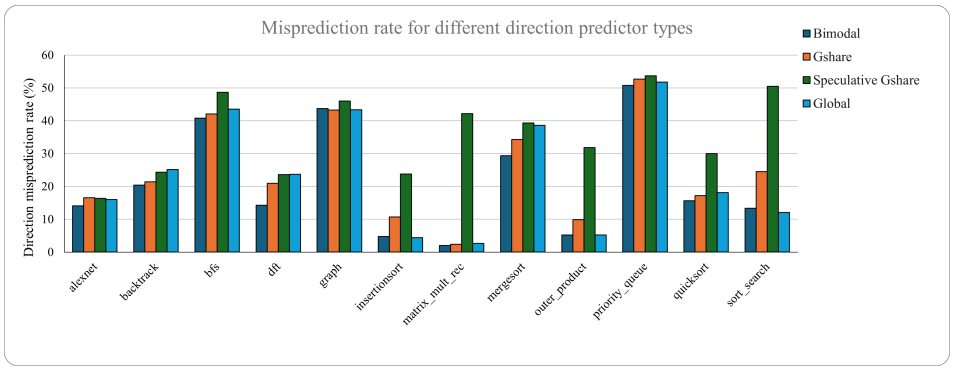

Figure 5: Misprediction Rate for different direction predictor types

Fig. 5 illustrates a detailed analysis of branch misprediction rates utilizing a variety of prediction algorithms with a constant BTB (branch target buffer) size of 64 and PHT (pattern history table) size of 32. All predictors below ignore the offset bits of the PC.

1. Bimodal: Utilizes 5 bits after the PC offset for indexing.
2. Gshare: Retired Global history register (confirmed taken or not taken from the ROB) XOR with PC (5 bits).
3. Speculative Gshare: Speculative Global history register (based on predicted T/NT of previous branches) XOR with PC (5 bits).
4. Global: Global history register (GHR) is indexed directly into the PHT table.

From Figure 5, it is evident that the bimodal predictor outperforms other prediction models in the majority of C programs tested, with the exception of graph.c where its performance is surpassed by the Gshare predictor. The Gshare with speculative global history register doesn't have good prediction rates and it may be because it relies on the patterns of the branch behavior to make accurate predictions, but if the program has unpredictable branching patterns, speculative Gshare may then struggle to make accurate predictions.

4.4 Data Cache Performance

Our DCache performed well for most programs. The DCache hit rate was between 90% and 98% for most C programs, as seen in Fig 6. This high hit rate was great for our processor, as we were able to avoid the 100ns memory latency for a lot of loads. The one exception was matrix_mult_rec.c, which saw a hit rate of 20.81%. We were not sure as to why this hit rate was so low.

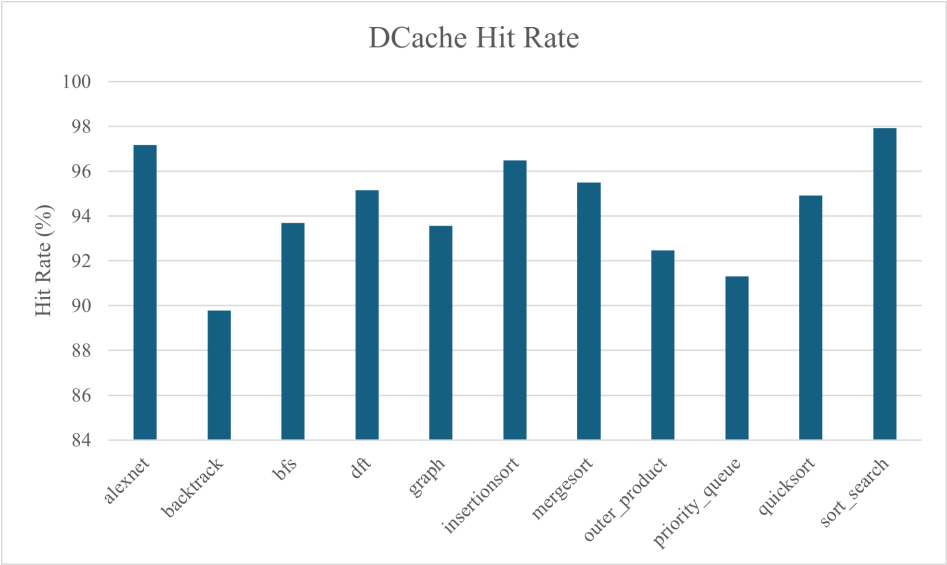

Figure 6: DCache hit rate

4.5 Store-to-Load Forwarding

Fig. 7 illustrates the percentage of loads receiving all necessary data directly from forwarded stores, thereby eliminating the need to request data from the DCache. This visualization underscores the significant advantage of implementing data forwarding within our memory architecture. Without data forwarding, each load requiring data from the DCache could incur a latency penalty of 100ns in the event of a DCache miss, highlighting the potential performance impact of memory latency on load operations.

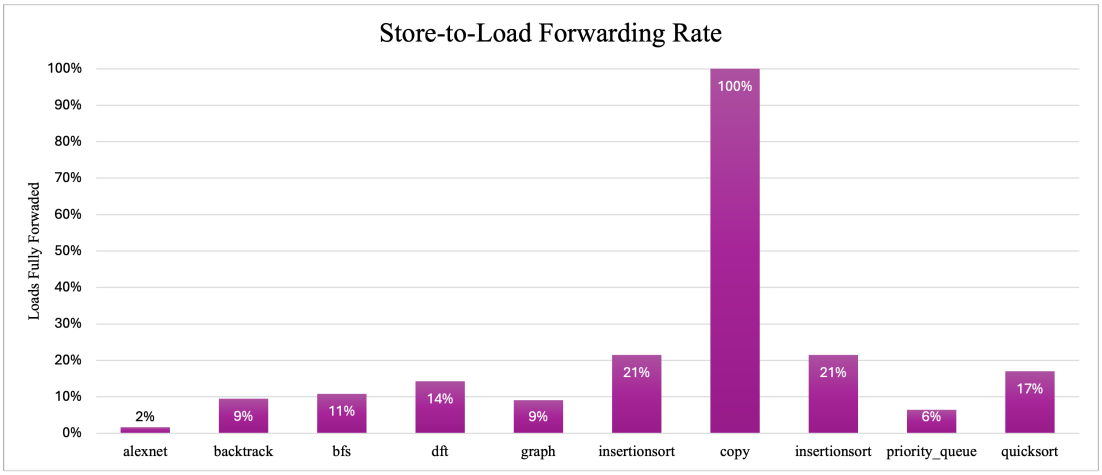

Figure 7: Forwarding Rate

4.6 ROB

For most programs, our ROB maintained a high number of free entries. This likely stems from the fact that our instruction buffer does not dispatch as many instructions as we would like, as its average occupancy for most programs is less than our degree of superscalarness (3), as seen in Fig. 3. For the three programs highlighted in yellow in Fig. 8, the ROB nears full occupancy, allowing us to do more analysis to determine the optimal ROB size. We see that for ROB sizes 8 and 16, the ROB nears full occupancy, while for ROB size 32, there are more free entries on average. During synthesis, we found that a ROB of size 32 was not on our critical path. Therefore, we set our ROB size to be 32 in our final submission to avoid possible structural hazards.

Average Free ROB Entries per Cycle:				
ROB Size:	8	16	32	64
alexnet	5.69	13.31	29.04	61.03
backtrack	4.94	11.9	27.54	59.51
bfs	5.2	11.7	26.54	58.23
dft	5.04	11.59	26.07	58.06
graph	5.58	12.7	27.95	59.57
insertionsort	0.84	1.33	8.75	40.04
matrix_mult_rec	2.8	8.97	23.04	53.8
mergesort	4.33	9.93	24.35	56.23
outer_product	1.03	1.96	7.19	34.52
priority_queue	6.17	13.8	29.49	61.43
quicksort	3.21	7.97	20.46	49.61
sort_search	0.92	1.81	9.38	36.85

Figure 8: Average number of free ROB entries per cycle

5 Testing

5.1 Unit Testing

To ensure the correctness of each individual module within our processor design, we conducted comprehensive unit testing. This involved the creation of tailored test cases for every module, to cover a wide range of scenarios and edge cases. These test cases were designed to simulate various inputs and conditions that the module might encounter during actual operation.

In the unit testing phase, we generated data packets to feed into each module. By scrutinizing the behavior of the module against these test cases, we could verify its functionality and identify any potential issues or bugs. This approach allowed us to rectify any discrepancies or inconsistencies in the module's behavior, ensuring its correctness before integration into the larger processor framework.

5.2 Integration Testing

Once individual modules passed the unit testing phase, we proceeded to integrate them into larger sections of our processor design incrementally. Initially, we combined critical components such as the RS, FUs, and CDB, testing their interoperability and correctness as a cohesive unit.

Gradually, additional modules were integrated, and testing was performed at each stage to assess the overall functionality and performance of the integrated sections. This incremental integration approach enabled us to detect and address any compatibility issues or dependencies between modules, ensuring seamless integration into the complete processor architecture.

5.3 Validation Testing

Upon the integration of all modules, we conducted validation testing to ascertain the correctness and performance of the entire processor design. To validate the processor's functionality, we compared the output results, including writeback and memory files, against ground truth files generated by the Lab 4 reference implementation.

Furthermore, to ensure robustness and resilience across diverse workloads, we conducted extensive testing using a variety of test cases. These test cases were generated by recompiling multiple C programs with different optimization flags, thereby creating novel scenarios designed to stress different aspects of the processor's operation. By subjecting the processor to a slew of diverse test cases, we could verify its stability, performance, and adherence to design specifications across varying conditions and workloads.

6 Contribution

Zach Milan (18.5%): Zach worked on the RS, RAT, Instruction Cache, integrating the Fetch stage, their associated testbenches, debugging for each, and designed and wrote the Graphical User Interface (GUI) Debugger. In addition to this, he assisted in writing the bash script to simulate our processor at various structure sizes and helped write testbench tasks for other various modules.

Harshit Nanda (21%): Harshit worked on the RS, ROB, Freelist, RRAT, Branch unit, Fetch, Mult unit, PRF, CDB, Branch prediction, BTB, Bash scripting, DCache. He worked on designing, making and testing all these modules. He also made a bash script to do regression testing. He also did the testing and debugging of the MSHR, store queue, load unit. He worked on the integration and debugging of the memory system, the out-of-order system and the fetch-decode-dispatch stage.

Enrique Orozco Jr. (18.5%): Enrique worked on the RAT, Fetch, ALU, Mult, BR, and the instruction cache. Alongside these modules, Enrique worked heavily on the test benches for these modules and helped create tasks for various other modules, such as the rs_cdb_fu, and rs. Additionally, Enrique worked on bash scripts, such as a performance script that simulated the processor at various sizes for different structures, outputting the data into an output file used for analysis.

Maatla Sefawe (21%): Maatla worked on the RS, ROB, RRAT, PRF, Decoder, Dispatcher, Load Unit, Store Queue, Load Calculator, DCache, MSHR, Mult Unit, CDB, Fetch and Memory Arbiter. He worked heavily on designing, developing and testing these modules. He developed the analysis code to get detailed insight into the performance of our system. He worked on the integration and debugging of the memory system, the out-of-order system and the fetch-decode-dispatch stage.

Ayush Vashi (21%): Ayush worked on the RAT, ALU, Branch Unit, Load Calculator, Store Calculator, Mult Unit, CDB, RS, DCache, MSHR, and Memory Arbiter. He helped develop, test, and debug all of these modules. He worked on the integration and debugging of the memory system, the out-of-order system and the fetch-decode-dispatch stage He also gathered data and analyzed the performance of the processor.

7 Conclusion

Overall, we achieved 100% correctness on every public test case while employing all unique optimization compiler flags. Additionally, we attained a clock period of 12ns. Although we had hoped for better CPI/IPC performance, we are nonetheless satisfied with the overall performance of our processor.

We extend our sincere gratitude to Dr. Brehob, Ian, Mustafa, and Bradley for their invaluable help and guidance throughout the semester.