EECS470 Final Project Report

Dhanvi Bharadwaj
dhanvib@umich.edu

Eric D'Urso
edurso@umich.edu

Luke Nelson
lukenels@umich.edu

Chase Ruskin
cruskin@umich.edu

Anthony Varkey
avarkey@umich.edu

Kyle Wang
kylewang@umich.edu

***Abstract*—This report presents the design, implementation, and analysis of an N-way superscalar out-of-order processor based on the RISC-V instruction set architecture. The processor incorporates advanced features such as a G-Share branch predictor, return address stack, instruction prefetching, variably-associative data cache, non-blocking instruction and data caches, and a victim cache. The design follows the MIPS R10K architecture, utilizing key components like the reorder buffer, reservation station, and physical register file to enable instruction-level parallelism. We developed comprehensive testing suites, including a GUI debugger and coverage-driven unit tests, to ensure functional correctness. Performance analysis led to optimizations in branch misprediction handling and instruction fetching throughput. The final configuration achieved an average CPI of 2.46 for C programs with a clock period of 16.0 ns, resulting in approximately 25.4 MIPS. Our project demonstrates the complexities of modern processor design and highlights the importance of effective project management and communication in large-scale hardware development efforts.**

I. Introduction

EECS470 uses a subset of the RISC-V instruction set architecture (ISA) to design microprocessors. To support this ISA, we designed an N-way superscalar out-of-order processor, implemented it using SystemVerilog, and synthesized it using Synopsys Design Compiler. We present our design, analysis, and testing of this major design experience.

A. Summary of Advanced Features

In addition to basic functionality of the ISA and the introduction of instruction level parallelism, the processor also implements several advanced features to enhance performance and the development process. These advanced features include the following:

- N-way superscalarness
- A functional GUI Debugger compatible with the entire processor, the processors submodules, and the Project 3 processor
- Advanced Branch Predictor (G-Share)
- A Return Address Stack
- Instruction Prefetching
- A Variably-Associative Data Cache
- Non-Blocking Instruction and Data Caches
- A Victim Cache
- Coverage-Driven Unit Tests

Details as to the design and implementation of these advanced features will be described in the subsequent subsections.

II. Design and Implementation

This section will examine the design and implementation of our processor. The core components are based on the MIPS R10K architecture. The development of this processor was split into four main components: (1) the development of the memory interface, instruction, and data caches, (2) the development of the in-order fetch and dispatch stage, (3) the development of the out-of-order core, and (4) the development of the retire stage. All of these components are discussed in depth in this section.

A. Memory

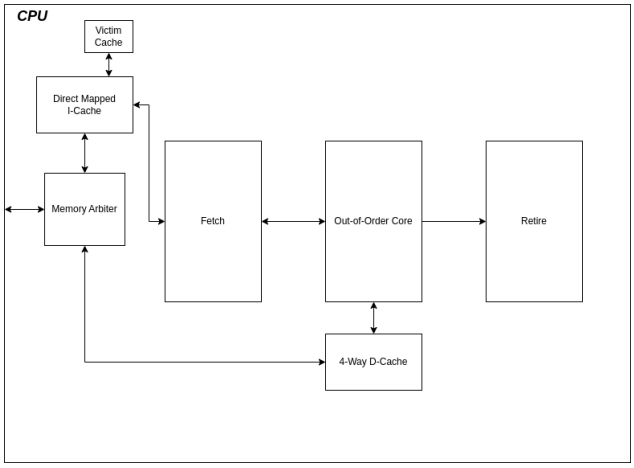

Fig. 1: The top-level structure for how the CPU handles memory transactions. We implement a 32 byte direct-mapped instruction cache with a 4 byte victim cache and a 32 byte 8-way associative data cache. The processor's fetch stage is responsible for gathering instructions to process, the out-of-order stages are responsible for processing instructions, and the retire stage is responsible for maintaining precise in-order state of processed instructions.

The constraints for the provided memory module were that: (1) all memory operations would have a uniform, 100ns latency and (2) all memory operations would execute in the order that they were dispatched. With this in mind, we designed our memory arbiter, data cache, instruction cache, and victim cache under the following specifications.

a) *Memory Arbiter:*

The memory arbiter connects the instruction and data caches with the memory interface. The caches are designed to assert their outputs until they are accepted by memory, and the memory arbiter controls this by always prioritizing data cache requests over instruction cache requests. This policy was

implemented under the assumption that instruction prefetching allows the instruction buffer to run-ahead, thus its current requests are not as urgent as those from the data cache. The memory arbiter does not track any information about what request or transaction tag came from which cache. Since memory transaction tags are unique, transactions returned from memory broadcast to both the data cache and instruction cache. Caches will ignore any data and tags that did not originate from them.

b) *Data Cache:*

The data cache is a non-blocking multi-ported write-back cache that leverages a configurable number of Missed Status Handling Registers (MSHRs) to interface with memory for cache misses. The data cache can have a configurable number of load ports and one write port. The data cache is variably associative, and uses a Not Most Recently Used (NMRU) eviction policy for non-direct-mapped associativity configurations. The data cache is designed to return hits to a load/store unit on the same cycle they are requested, and misses as soon as they become available in the cache. The MSHRs and the store queue share the write port to the data cache, and the store queue has priority over the MSHRs in case the data is in the cache and can hit in the same cycle. Once the store queue is not requesting the write port, a MSHR that has its data back from memory can write its data into the cache and broadcast the returned data back to the load/store unit. We chose to implement the write-back policy for writing store requests back to memory instead of the write-through policy in order to reduce contention of the memory bus during program execution. Once a halt instruction (`wfi`) is retired and all currently allocated MSHRs have completed, the data cache enters a memory synchronization state which iterates through each line in the cache and sends a memory store command for each data block that is both valid and dirty. Once all blocks have been checked, the data cache emits the true stop signal for the processor, indicating the cache has been flushed.

c) *Instruction Cache:*

Similar to the data cache, the instruction cache is a non-blocking multi-ported cache that implements MSHRs to resolve cache misses. We chose a non-blocking cache for the instruction cache to increase the throughput of instructions being returned to the instruction buffer in the fetch stage. The instruction cache is direct-mapped, and cache hits are returned to the instruction buffer on the same cycle they are requested. The first miss detected is sent to the victim cache and can still be acknowledged as a hit if found. Evicted blocks from the instruction cache are placed in the victim cache, and misses in both the victim and instruction cache trigger a memory request and allocates a MSHR to await the returned instructions from memory.

d) *Victim Cache:*

The victim cache is a one way, fully associative cache that can hold four blocks evicted from the instruction cache. In the event of a miss in the instruction cache, the victim cache is examined before allocating a MSHR. If the requested data is in the victim cache, it is swapped back into the instruction cache and a memory transaction is not issued. However, an MSHR will have priority over the write port into the instruction cache if it has its data back from memory, and the hit block in the victim cache will have to be dropped. This design choice was implemented due to its simplicity and the assumption that the MSHR data is more of a priority over the current hit data because it was requested many cycles ago due to previously being a miss.

B. Fetch

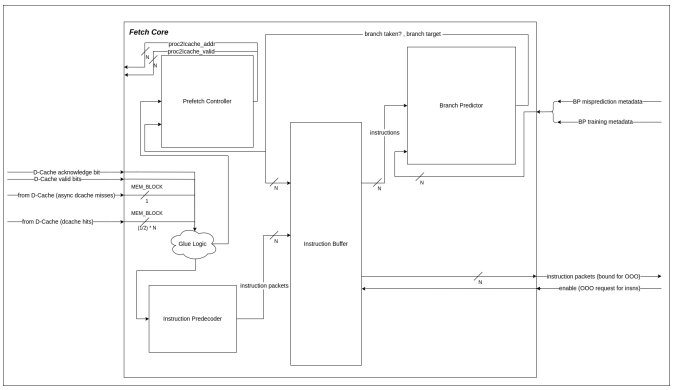

Fig. 2: Depicts the design of the fetch stage, and contains the interfaces and datapaths between submodules of the fetch stage: the prefetch controller, the instruction predecoder, the instruction buffer, and the branch predictor. The primary role of the fetch stage is to read instructions from memory and provide them in program order to the later defined out-of-order core.

The fetch unit was created to be responsible for providing instructions to the out-of-order core (Section II.C) in program order, with the caveat of branch speculation, at the highest rate possible. Since the out-of-order core had the ability to take in N instructions per cycle, the goal of the fetch unit was to provide N instructions per cycle. It is also necessary that they are provided in program order (outside of branches which have the ability to be rolled back if predicted incorrectly by the out-of-order core) for correct execution order of instructions.

The primary bottleneck of the datapath of the fetch unit was communicating with memory, as we were limited to a 2-word memory interface with significant memory latency. To better handle this constraining interface, the fetch stage pipelines memory accesses and caches the results before pulling them into an instruction buffer. This was done in the form of the above instruction cache (Section II.A.c). However, the fetch unit was able to provide even more utilities in order to limit the effects of this bottleneck.

a) *Instruction Buffer:*

The first improvement implemented was an instruction buffer. The goal of the instruction buffer was to provide a buffer where instructions could be queued in cases where the out-of-order core was unable to accept instructions, such as stalling due to a structural hazard in one of its units. The length of the instruction buffer was made customizable via a SystemVerilog parameter defining its depth.

The instruction buffer was implemented with a FIFO (first-in, first-out) buffer, which could be filled with N instructions requested from the instruction cache per cycle. The method of requesting is detailed in prefetching (Section II.B.b). The output of the buffer was also N instructions wide, as when the out-of-order core 'enabled' input from the fetch stage, the fetch stage would provide up to N 'ready-to-go' instructions to the out-of-order core for execution. In order to keep track of which instructions were 'ready-to-go', we used three pointers to define entries in the buffer: head, branch prediction head, and tail. Each entry in the buffer was also marked with two booleans of metadata, telling whether they contain a valid instruction (i.e. have they been returned by the I-Cache yet), and whether they have been run through the branch predictor, and contained the instruction, address, and place holders for branch prediction results. An instruction can only be given to the out-of-order core when all instructions prior to it are valid and have been 'eaten' by the branch predictor. The reason instructions are fed through the branch predictor is to make sure that if it is a branch, the instructions that follow it follow the predicted target of that branch. The purpose of the branch prediction head is to mark which instructions have been fed through the branch predictor and thus have been solidified as speculated program order and are ready to be handed off to the out-of-order core for computation.

b) *Prefetching:*

Prefetching is how we defined our logic for speculatively filling the instruction buffer as much as possible in order to feed the out-of-order core a steady diet of instructions. Our prefetcher had the ability to make N requests to the instruction cache at a time, and allocated a slot in the instruction buffer at the tail pointer (when space was available) on both a hit and a miss. If hit, the instruction was marked as valid, indicating it had been received from the instruction cache. If a miss, the fetch logic would watch the asynchronous memory returns from the instruction cache and mark it as valid when it returned.

The prefetcher worked by keeping track of a speculative PC and attempting to fetch as far ahead of the real PC in program order as possible in order to preemptively fill the instruction buffer. To do this, it would start at a baseline of the real PC: at reset this was given as 0x0, and on branch misprediction results from the out-of-order core would be the correct branch target of the misprediction. From here, it would move ahead (assuming branches not taken) and fill the instruction buffer until the instructions being fed through the branch predictor resulted in a predicted branch taken. When the branch was taken, it would change the speculative PC to the branch target, and also move the tail pointer in the instruction buffer to invalidate all instructions fetched that went down the wrong speculative path.

c) *Branch Prediction:*

In order to perform accurate branch prediction, we needed to utilize a robust branch predictor which could predict branches with a high rate of accuracy. We chose to implement a G-Share predictor. When making predictions, G-Share efficiently accounts for local branch history and global branch history by hashing together part of the current branch PC and a global history register via an XOR. Our implementation of G-Share allows for the global history register width to be customized via a parameter. We implemented both a version of G-Share that only updates the Global History Register (GHR) on retire, and one that speculatively updates the GHR on every cycle assuming the predicted direction of any current branch is correct. For the speculative version, we keep a counter of the number of not-resolved predictions. In this version, the true length of the GHR is determined by a width parameter supplied plus the number of total entries in the Reorder Buffer (ROB), which accounts for the maximum in-flight branches. For recovery from mispredictions, we simply shift the GHR right by the value of the counter to guarantee that we flush all the speculative updates following the misprediction.

d) *Branch Target Buffer:*

The branch target buffer (BTB) allows us to reliably predict the destination address of branches by storing a mapping of part of the PC of the branch instruction to its destination once discovered, which can then be reused in future encounters of that branch. Our branch target buffer is a direct-mapped cache indexed by a customizable number of lower bits from the branch PC. We chose 10 bits for our final configuration to provide a reasonable size for the cache while minimizing the chance of collisions between different branches due to the number of possible entries. Our BTB has one read port and one write port to support predicting entering new destination information for one retired branch per cycle.

e) *Return Address Stack:*

We implemented a return address stack (RAS) to improve performance on predicting function return destinations to go beyond the BTB's performance. Our return address stack has a customizable depth. We used predecoding to determine whether a given instruction was a function call (`jal`, `jalr` where destination register is not logical register zero) or a function return (`jalr` where destination register is logical register zero). All function calls are pushed onto the return address stack, and whenever a function return is detected the topmost entry of the RAS is popped from the stack and its address plus four is used as the predicted target address of the branch. We add four to proceed to the next instruction following the original function call.

All of these units together comprised the fetch unit, or fetch stage, of our processor. They aided the processor to avoid the memory bottleneck by loading an instruction buffer to avoid wasted time when the out-of-order core stalls, prefetching instructions into the instruction cache in order to save time waiting on cache misses, and have accurate branch prediction to eliminate unnecessary rollbacks on a branch misprediction.

C. Out-Of-Order Core

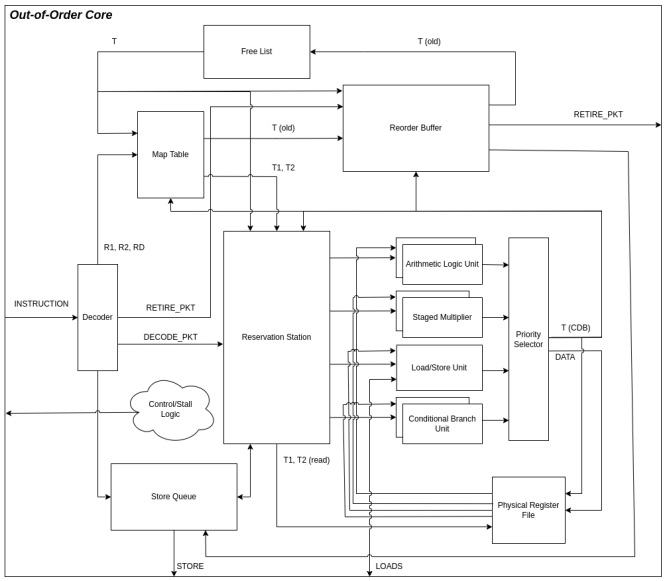

Fig. 3: The Out-Of-Order (OOO) core is responsible for issuing, executing, and completing instructions out of order. The inputs to the OOO core from the fetch stage are in-order, and the outputs from the OOO core to the retire stage are in-order. The OOO core involves data structures and techniques such as register renaming to eliminate write-after-write and write-after-read dependencies. A stall is sent back to the fetch stage if any data structure (ROB, RS, free list, map table) have a structural hazard.

The main priority of the out-of-order core is to avoid false hazards, commonly known as write-after-read (WAR) errors, write-after-write (WAW) errors, and structural hazards, which force the processor's pipeline to stall.

Our pipeline follows the R10K processor implementation popularized by the famous MIPS R10K processor designed in the 1990s. This design utilizes register renaming to rename logical registers (for example, the 32 provided by RISC V-32) into a customizable amount of physical registers.

This process happens by renaming logical registers to physical registers 'on the fly' in such a way to avoid some of the false hazards described above. For example, in a write-after-write hazard, if two instructions will write data into logical register r1, then we can rename the r1 in the first instruction to p1, and in the second instruction to p2, effectively allowing us to simultaneously calculate and hold the results to both instructions, even though the compiler or programmer named them the same logical register.

This approach, however, requires comprehensive tracking of state, which is defined by these modules below: the map table (Section II.C.b), the free list (Section II.C.c), and the physical register file (Section II.C.f).

Another main tenet is the ability to maintain 'precise state' so that if a program halts or gets interrupted, its state can be recovered precisely. This involves all instructions coming both into and out of the out-of-order core in order, as their state must be retired in order to maintain precise state. This is handled by the Reorder Buffer (Section II.C.a).

a) *Reorder Buffer:*

The Reorder Buffer (ROB) is responsible for tracking instructions throughout their lifetime in the out-of-order (OOO) core and retiring them after they have completed. The ROB organizes instructions in rows, where each row has at least 1 valid instruction and up to N valid instructions. If there is a cycle where no valid instructions are dispatched, a ROB row is not allocated. We greedily allocate a ROB row if at least 1 valid instruction is dispatched in order to push instructions through the backend as soon as possible. After being entered into the ROB, the ROB listens to the Common Data Bus (CDB) for any physical tags of in-flight instructions and marks them as complete as they come back on the CDB. The ROB retires a row when all instructions in the row are complete. Instructions that are invalid in a ROB row are marked as complete upon being entered into the ROB since they were not assigned a physical tag and will never have a response broadcasted on the CDB.

The ROB handles branch mispredictions by entering a rollback state upon detection. If a row that is ready to retire has a branch misprediction, as heard from the CDB, it will output a rollback detected signal to the other modules of the OOO core and begin rollback. First, the ROB retires the instructions in the row of the branch misprediction up to the mispredicted branch instruction, and then enters rollback. During rollback, it reverses the read address of the ROB to use the tail, and outputs the instructions in reverse order of their allocation. This reversal essentially "unwinds" the processor's state cycle by cycle until the ROB is empty. After the rollback sequence, the map table and free list are restored to the architectural precise state of the retired mispredicted branch instruction. The ROB lowers its stall signal once rollback is complete and the OOO core can begin to accept more instructions from the fetch stage. We chose to implement the rollback strategy due to its simplicity in comparison to other branch recovery strategies such as checkpointing.

b) *Map Table:*

The Map Table is responsible for dynamic register renaming and mapping logical registers to physical registers. It processes up to N instructions per cycle sequentially, ensuring instruction order is preserved even as multiple mapping are updated in the same cycle. Internally it maintains two memories: a tag table and a ready bit table.

For each incoming instruction, the tag table provides the current mapping of the logical register to a physical register, while the ready bit table determines if the corresponding physical register has been marked ready. Writes to the tag table update the physical register mapping for a logical register, and the ready bit is cleared to signal that the new register is not yet ready, unless a rollback state overrides this behavior. Additionally logical register zero is hardwired to physical register zero and is always marked ready.

To handle dependencies within a cycle, instructions are processed sequentially, ensuring that updates made by earlier instructions within the same cycle are immediately visible to later instructions. This approach avoids hazards caused by

inner-cycle dependencies, as later instructions always see the most up-to-date state of the tag and ready bit tables. During updates, the CDB (common data bus) is monitored to set ready bits for completed physical registers. If the logical register tag has been overwritten by a newer instruction before the CDB broadcast, the ready bit remains unchanged, preventing stale updates.

To support branch misprediction recovery, the Map Table can rollback its state. On rollback the OOO core sends the previous physical register tags to overwrite current entries. The Map Table assumes that the incoming tags to write will be in reverse order on cycles of rollback to ensure that any inner-cycle dependencies are undone correctly. This rollback mechanism simplifies state recovery while maintaining correctness of logical-to-physical register mappings.

c) *Free List:*

The Free List is responsible for managing the availability of physical registers in the processor. It operates in a circular FIFO queue that racks free registers, allowing up to N registers to be freed or allocated in a single cycle. Its basic logic is the same as ROB with a head and a tail pointer pointing to the first and the last available physical register. The FIFO queue also employs internal forwarding, allowing registers freed and requested in the same cycle to be immediately visible for allocation. Additionally, physical register 0 is statically reserved and excluded from the allocation process.

In the dispatch phase the Free List provides up to N oldest available physical register to support register renaming for new instructions. These registers are selected in order of availability to maintain efficiency. If fewer than the requested number of registers are available, a structural hazard is signaled, stalling the pipeline. During the retire phase, physical registers that are no longer needed are returned to the Free List. These registers correspond to those previously mapped by instructions now retiring from the Reorder Buffer (ROB). Once freed, they are marked as available and added back to the pool of free registers.

d) *Function Units:*

The function units (FUs) are a set of modules that perform operations on the data read from the physical register file. With more FUs than lines on the CDB, a priority selector selects among FUs that have valid data to broadcast their tag on the CDB and write results into the physical register file. Each FU has the same set of control signals to tell the Reservation Station (RS) when it can accept more data and tell the CDB when it has data ready to complete.

Our FUs consist of integer arithmetic logic units (ALUs), conditional branch units (CBU), a load/store unit (LSU), and a pipelined integer multiplier unit (MULT). Our MULT implementation requires 8 cycles to execute multiply instructions, and the LSU implementation requires a variable number of cycles for loads dependent on if the requested data is a hit or miss in the data cache.

The ALU computes arithmetic and logical operations. The CBU handles unconditional and conditional branch results and target addresses. The MULT handles multiplication operations. The LSU handles computing the load and store addresses as well as handles load requests to the data cache. Computed store addresses take a single cycle and are sent to the store queue to update the state of the store instruction's entry in the queue.

e) *Reservation Station:*

The reservation station (RS) is the key component which allows us to exploit instruction-level parallelism.

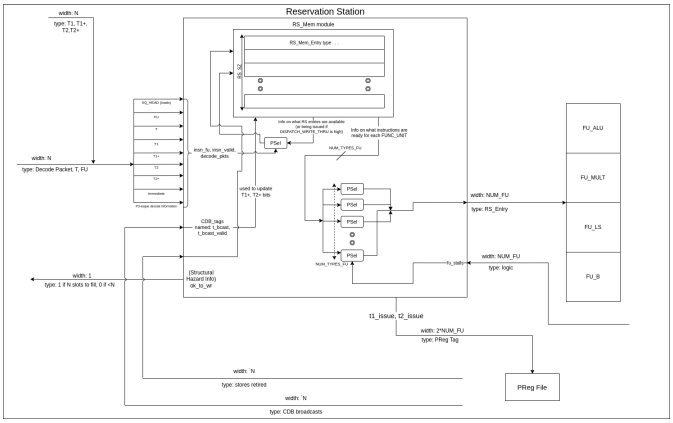

Fig. 4: The high-level implementation of the Reservation Station. The inputs to the Reservation station include instructions and required metadata for issuance to execution units, as well as updates from the common data bus and other locations about the readiness of data operands.It outputs instructions after data dependencies have resolved to execution units using a priority selection scheme.

The reservation station takes in renamed instruction packets: these packets contain instruction metadata (provided by the decoder) as well as other information such as physical register tags for both source and destination registers, the SQ head position (for load dependencies), and branch predictions. The RS has the ability to take in up to N instructions per cycle, and has a parameterizable size. The memory of the RS is implemented using a priority selector to find the first N empty locations in the buffer, and slot in the respective incoming instruction packet.

The RS is also in control of what instructions are issued to the function units. This is also implemented with priority selectors. Each function unit type has its own priority selector which looks at the memory buffer and selects up to the number of that type of function unit of instructions to issue. This priority selector also takes into account a stall signal returned from each function unit, giving information about whether it is available to receive a new instruction on the next cycle. The RS can only issue instructions for which all source operands are available (and ready). To do this, the RS listens to the CDB (and the Store Queue for loads) and marks metadata bits in its instruction packets to store whether source operands are available or not, and then, on egress from the RS, the instruction packet essentially 'visits' the physical register file on the way and receives the data based on the physical register addresses it requires.

f) *Physical Register File:*

The Physical Register File (PRF) stores the actual data for instructions that write data to registers using the physical register tag as the index into its memory. Since our design is based on the MIPS R10K architecture, the PRF is the single source of truth for a program's register data. It has twice the number of read ports for the FUs to read their source operand data and N write ports for N completed instructions to write their results back into the PRF.

g) *Store Queue:*

The store queue maintains precise program state by handling memory store operations while interacting with key components of the out-of-order core. Our store queue design is implemented as a circular FIFO structure, supporting multiple store dispatches and retirements per cycle. During dispatch the store queue allocates an entry for each incoming store instruction, where it awaits address and data resolution. The Load/Store Unit (LSU) updated the store queue entry once the address and data had been computed, marking the store as complete. Stores remain in the store queue until they are ready for retirement to ensure the speculative state does not affect program correctness.

Our store queue design does not include store-to-load forwarding. Instead, the store queue enforces strict ordering by blocking all subsequent loads if there are unresolved older stores. Specifically, the Reservation Station uses the head and tail pointers of the store queue to assess potential hazards. Loads are issued only when older stores advance to the head pointer, signaling readiness for retirement and guaranteeing memory consistency.

The Reorder Buffer coordinates store retirement by signaling the store queue when a store is ready to commit. Upon receiving the signal, the store queue issues the store to the Data Cache in program order. Once the data cache acknowledges the operation, the SQ clears the corresponding entry and advances its head pointer. This interaction ensures stores are committed in order and preserves precise program execution.

D. Retire

The retire stage finalizes instruction execution by committing the results to architectural state. It consists of an Architectural Map Table, which updates the mapping of logical registers to physical registers as instructions retire in program order.

a) *Architectural Map Table:*

The architectural map table is an instantiation of the same map table module used in the out-of-order core (Section II.C). This instantiation, however, only connects inputs to this module and serves to maintain the precise state of the program at a given time. The architectural map table only connects write inputs and does not interact with ready bits or rollback mechanisms. Its sole purpose is to update the architectural state as instructions complete and retire, ensuring the committed logical-to-physical register mapping is correct. In the case of an interrupt, this would be used to recover the state of the processor prior. By maintaining this committed state independently of speculative execution, the retire stage guarantees correct program behavior and state recovery.

III. Analysis

After integrating all of our units and verifying functional correctness across all stages of the processor, we conducted a detailed analysis of our pipeline performance and behavior. We explored key metrics such as instruction throughput, resource utilization, and stall behavior under various workloads. By evaluating each unit's stalls and performance within the entire system, we worked to target the critical areas of our design to gain the most improvement within the time remaining.

A. Minimizing Branch Misprediction Penalty

One bottleneck we identified in our processor was the stalling of the OOO core due to branch misprediction. Since we used rollback as our branch recovery strategy, the number of stall cycles due to a branch misprediction is the number of ROB rows to unwind. If the ROB allocates rows at a rate much faster than instructions retire, then the processor will pay a large penalty in stalling many cycles. If the ROB retires instructions at a rate much faster than allocating rows, then the processor will spend very few cycles in a given rollback sequence. More mispredictions will also increase the number of stalls, so it is also a priority to have an accurate branch predictor.

We worked to minimize the branch misprediction penalty of the ROB by detecting a misprediction from the CBUs as soon as possible and triggering the OOO to immediately stop accepting more instructions until the rollback was resolved. By refusing more instructions into the OOO when a misprediction is found (sooner than retired), we can spend fewer cycles in rollback due to not allocating ROB rows between the time the misprediction was computed and the time the misprediction retired. Recall that the ROB retires instructions one row at a time, so this architectural decision has increased benefits for programs that have frequent dependencies between near instructions and a processor that has large N superscalar factor. Increasing N independently for a processor with high utilization (many valid instructions in each row) can also decrease the stall penalty for rollback because more instructions are unwound each cycle, however, this comes with the associated costs of a wide N, such as longer dependent paths.

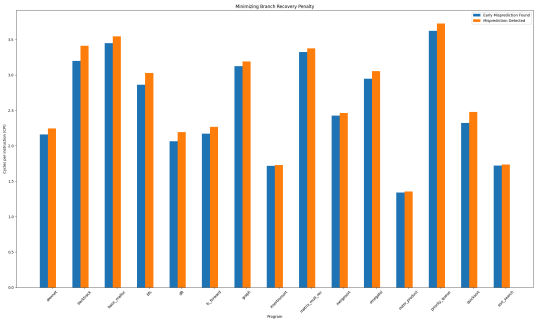

Fig. 5: The average CPI of the C programs for the branch misprediction found early implementation compared to normal branch misprediction detected. A lower CPI results from the early stall logic sent back to the frontend to stop sending instructions to the ROB until the misprediction is recovered.

B. Increasing Utilization and Fetching Throughput

Another bottleneck we identified in our processor stemmed from not being able to send instructions fast enough to the OOO core to process. This resulted in low utilization despite increasing N. As the ROB greedily allocates a row in a single cycle if at least one instruction is valid, it is important to have at least N instructions ready at every cycle to keep utilization high inside the OOO core.

We realized that this was because we spent an inordinate amount of time (in cycles) waiting for instructions to load or be given to the fetch stage from the instruction cache (Section II.A.c). This was evidenced by determining the throughput of our ROB and Instruction Buffer. We found that across all C and Assembly program workloads, our average ROB utilization (tracked by which entries in the ROB of size 16, N=2 were in flight) was 59.39%. Our instruction buffer (size 16, N=2) was only utilized at an average rate of 23.64% (tracked by how many valid instructions were waiting in the instruction buffer and were predicted to be on the correct control path). This bottleneck was caused by the fetch stage, as we were unable to provide a steady diet of N instructions per cycle to the fetch stage.

To increase instruction fetching throughput, we designed our instruction cache to be non-blocking, and allowed the fetch stage to aggressively prefetch in an attempt to fill the instruction buffer as fast as possible. By keeping the instruction buffer full, the OOO core can ideally accept N valid instructions every cycle. On branch recovery, the instruction buffer is cleared and the prefetcher begins prefetching from the mispredicted branch's corrected PC. By overlapping prefetching with rollback, we can hide the cost of the rollback when the next set of instructions are missed in the instruction cache. We redesigned the fetch stage to handle the instruction cache's non-blocking nature and prefetch ahead even if previous instructions were missed. The instruction buffer has a head pointer that dispatches instructions when they are valid and stored in the buffer, which tries to dispatch up to N instructions that are valid and contiguous in relation to program order.

These two improvements (non-blocking I-cache and instruction prefetching) allowed us to improve on both metrics (ROB utilization and Instruction Buffer utilization). These metrics prove that the Instruction Buffer both had enough instructions to keep the Reorder Buffer and OOO core occupied. Here is a bar chart showing the improvement of each of these utilizations before and after each of the improvements.

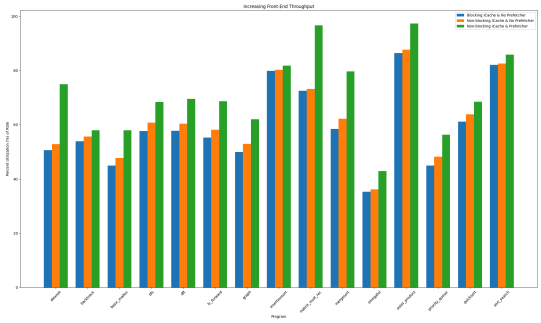

Fig. 6: The ROB size utilization was tracked by percentage of occupied entries in ROB for each C program. The non-blocking I-cache with the upgraded prefetcher provided the ROB with the highest utilization in all programs, eliminating unnecessary allocations of invalid instructions within a ROB row.

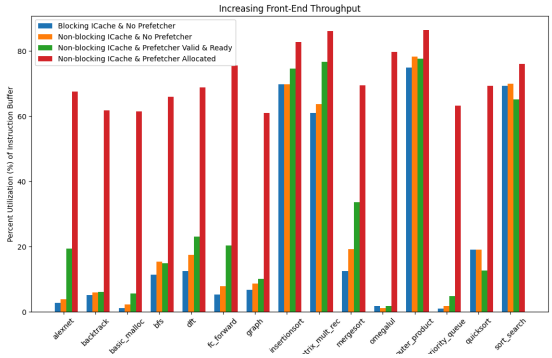

Fig. 7: The Instruction Buffer utilization tracked by % of occupied entries in IB by C program. The data shows that the utilization improved following changes to Instruction Cache and addition of a prefetcher, enabling higher throughput to the out-of-order core.

We can tell that we improved drastically on both utilizations, as well as CPI although undocumented due to lack of space. To decipher the legend on the Instruction Buffer graph, we show two utilization metrics on the Prefetching Instruction Buffer, one describes the total allocated entries, and one describes the amount of entries which were eventually handed off to the OOO core (as a percentage of how many could have been) (labeled as valid and ready). We still see that both of these values are an improvement over the non-blocking I-cache, showing that we improved the utilization of our out-of-order core, and enabled the out-of-order core to provide more instruction-level parallelism. However, the more accurate measurement of our improvement is the % of Valid and Ready, as this indicates how many instructions were available to be given out to the out-of-order core. This is also supported by the results of the Reorder Buffer utilization chart, as it shows that we increased the utilization of our ROB from an average of 59.39% to 67.69%. This allowed more instructions

to be in-flight at the same time, which is crucial to the out-of-order processor.

C. Branch Prediction

We also noticed that despite increasing ROB utilization, we were still severely limited by branch predictions, as significant amounts of cycles were spent rolling back speculated instructions, meaning that our efforts filling the ROB to reach ILP were getting mitigated.

As such, we designed multiple types of branch predictors to see which would provide us the highest accuracy, and thus allow our out-of-order core to work its magic most effectively. As a point of reference, we also tested a simple saturating 2-bit global counter predictor and predicting always taken and always-not-taken.

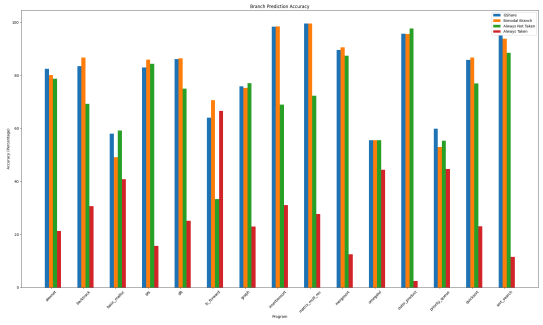

Fig. 8: The prediction accuracy for the 4 different branch prediction schemes: G-Share, bimodal, always not taken, and always taken. The average accuracy for G-Share was 0.8057, while the average accuracy for bimodal was 0.8356. The average accuracy for always not taken was 0.7199, and the average accuracy for always taken was 0.3159.

The leading two predictors with the highest average accuracy for our current implementation were G-Share at 0.8057 and the global 2-bit bimodal branch predictor at 0.8356. Due to time constraints, the branch prediction analysis was short-lived, although future work can look to adjust the GHR width and BTB size to improve prediction accuracy. The refactoring of the fetch stage to improve instruction throughput also could have undermined previously established assumptions surrounding the branch predictor.

According to these analyses, our processor has a large design-space exploration. Fine-tuning the many configurations and parameters can either greatly improve our design or hinder it with unscalable and poorly synthesizable datapaths. By identifying and targeting the largest bottlenecks, we believe we limited the critical issues in our design to maximize performance gains and reduce pipeline hazards.

IV. Testing

To verify our processor, we incorporated varying degrees of testing: unit tests, integration tests, and system tests.

At the module level, we created unit tests with coverage-driven test generation to traverse the module's state space while minimizing the number of cycles required to reach specific states. Unit tests leveraged Verb, a lightweight and flexible UVM-inspired framework that allowed us to write our models in Python and continue to simulate in our traditional VCS testbench environment. The Python models are responsible for generating the set of input test vectors as well as the correct set of output test vectors. These test vectors are written to files and read during simulation for the Device Under Test (DUT) to drive its inputs and compare its outputs. This testing methodology allowed us to identify and fix bugs sooner in our development process, saving us time during integration. To debug issues, we used waveform viewers and print line debugging to gain more information about our designs and eventually identify the root cause of any issue.

At the integration level, we created basic tests to check dataflow and compatibility of interfaces across modules. Any discrepancies or issues at this testing level forced us back to the module level to update designs and unit tests to ensure correctness was maintained with every change.

At the system level, we used the top level testbench for the cpu to interface with memory in order to run assembly and C test programs. To compare correctness, we also simulated an already verified in-order processor and checked for differences between the memory files (.out) and register write-back files (.wb) of our processor. This form of testing introduced a couple edge cases and uncaught bugs that we were able to quickly resolve using previously stated debugging techniques.

A. Coverage Driven Unit Tests

When verifying hardware, not only is checking what the design generated as outputs important, but also checking how the design generated those outputs is just as important. This tracking of knowledge in how the design reached a set of outputs is known as coverage.

In order to maintain correctness on an individual module level, we used coverage driven unit tests to formally verify correctness on random inputs, and inputs characterized to reach certain edge and corner cases. To do this, we use the previously mentioned Verb testing framework. Verb consists of a SystemVerilog package with predefined macros, a Python package to implement common test functions, and a command-line utility to check events and assert matching outputs. This allowed us to write a correctness model in a more human-readable manner and compare our expected outputs to the received outputs of every DUT. This approach enabled us to be confident in our testing because our tests generated cases that covered the design's state space while automating correctness checking. While this methodology was overall effective in reducing the number of bugs found in the final system test, it was also more complex because we had to really understand the design to model it using Python. A discrepancy between the model and the DUT sometimes indicated our model was functionally incorrect in comparison to our hardware design. As an added benefit, this forced us to more thoroughly understand the ideal implementation of the tested module, in turn leading to a smoother integration process.

We also utilized system tests to test the accuracy of our program against the expected results of given C and Assembly programs, as well as custom-designed programs expected to

test corner cases in our design. These verified the accuracy of our processor as a whole, validating that we integrated the individual modules, pipelines, and stages of our processor accurately.

B. Tooling Suites

Throughout the course of the development of our processor, we leveraged several different tools for testing, debugging, and development to aid our design and verification process. Some of these tools were made by our team during the design process of our processor, and others were adapted and utilized to meet our needs.

a) *GUI Debugger:*

The Verilog Visual Debugger (VVDB) is abstractly designed to be able to display information in a number of different configurations. The visual debugger reads both a configuration file (in JSON format) and a value change dump (VCD) file. The configuration file describes signals in the VCD that need to be displayed by the debugger. The debugger then uses its parsing unit to parse each signal, or `struct` of signals into an entry. The entry objects hold the values of the signal at different timestamps, as well as the current timestamp. With this we are able to pause, step through cycle-by-cycle, and graphically view components of the machine. The display unit of the debugger then uses a sequence of these entries to display the values specified in the configuration file at a specific time. The display unit also contains the ability to search through individual signals, and module instantiations. Consequently, due to its modular nature, VVDB is capable not only of displaying the signals of the processor described in this report, but also of the EECS470 project 3 processor, as well as all of the aforementioned modules defined in this report. Our VVDB helped more than the provided debugger because we were able to specifically search for signals and view them in more detail. The VVDB GUI was also easier to read and interact with than that of the provided debugger's interface embedded in the terminal. We selected a frontend library that would make the implementation of a scalable user interface relatively straightforward. VVDB is capable of assisting in the development of any module with a clock signal for which a valid configuration file can be produced.

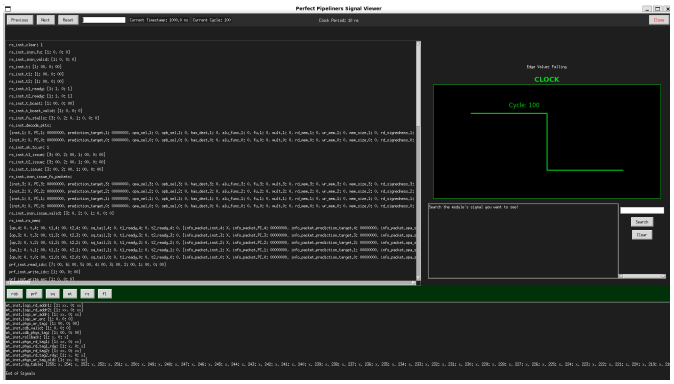

Fig. 9: Depicts the Verilog Visual Debugger on implemented pipeline. The main pane lists the signals tracked by the debugger, and the bottom pane allows the signals to be searched by the module they are defined in. The right pane displays the clock edge, and the search menu, and list of search parameters. The top bar displays various information about the clock period, cycle, and timestamp at the time the current signals occur.

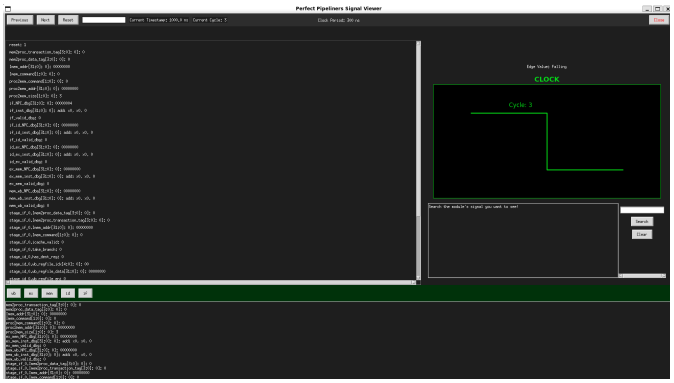

Fig. 10: Depicts the Verilog Visual Debugger on the EECS470 project 3 pipeline. The stages of the pipeline are listed on the bottom pane, and the remaining components of this configuration are identical to that of the implemented pipeline in Fig. 9.

b) *Orbit:*

For a faster development experience, we used Orbit, a SystemVerilog package manager. Orbit allowed us to script our workflows for testing and benchmarking without having to explicitly specify the list of source code dependencies for each SystemVerilog module. By using Orbit, we were also able to introduce new workflows outside of the course's provided VCS simulations, such as using ModelSim for simulation. Having a docker image of a version of ModelSim alongside Orbit allowed us to run regression tests on GitHub Actions for each change to the codebase.

c) *GitHub Actions:*

We implemented GitHub Actions Continuous Integration and Continuous Deployment (CI/CD) features in order to automate our testing. We designed a test script which would run both our coverage-driven unit tests described above for each module as well as a bank of system tests, where we would run C and assembly programs under different input parameters and verify the write-back and memory output files matched that of the already verified in-order processor. This strategy ensured that when features were merged into the main branch

of our codebase, we could ensure full compatibility with our current design as well as full accuracy and functionality of our new features. Unfortunately, later in the process of developing the processor, GitHub actions stopped support for some of the dependencies of the ModelSim version we had access to. Although the CI flow no longer works, it provided an invaluable source of verification of the source modules earlier in the development of the project.

V. Project Management

At the beginning of the project, we set forward-looking goals for each of the milestones provided to us. They were as follows:

- Milestone 1: Design all individual modules needed to define memory-less R10K-style Out-of-order Core (i.e. RS, ROB, Map Table(s), Free List, Function units)
- Milestone 2: Integrate all above units into an out-of-order core and implement a simple fetching unit (predict not taken, no prefetching) and retire unit (with only architectural map table). Be able to run simple programs such as `mult_no_lsq` and `no_hazard` (ignoring the `sw` instruction).
- Milestone 3: Optimize fetching unit for branch prediction, prefetching, instruction buffer, etc. Implement memory modules (store queue, data-cache, instruction-cache). Integrate all units back in and be able to run any program (especially the ones given to us).
- Deadline: Optimize unit. Perform analysis on workloads and processor modules to fine tune designs and parameters for sizes of items in modules in order to have the most advantageous processor.

We are happy to say that we were able to achieve all of our individual group's and required goals for these milestones with the small caveat of achieving the milestone 3 goals 1-2 days after milestone 3 passed.

Some strategies of project management that we employed were clear long-term and short-term goal setting, partner programming and brainstorming, frequent inter-group communication, and common work times/work sessions. For setting long-term goals, we first set long-term goals when we started the project, and then adapted them to fit our current state after each milestone meeting. Short-term goals were set and changed more frequently, and weren't always in an organized meeting but often over other routes of communication such as group chat or phone/video calls, but were designed to make sure that we all were utilizing our time efficiently and were not letting anyone fall behind. During the project, we heavily implemented partner programming and working in small groups over working individually. Anecdotally, comparing modules done by group members individually vs in small groups, we saw a much higher quality of work (in efficacy, documentation, simplicity) and lower time taken to code it up when working in a small group. We also believe that this helped each of us know more about the processor as a whole rather than our own individual parts. Lastly, we greatly valued our inter-group communication between small groups, as we often would be working on modules which would interface with each other, and were able to do well at making sure that we would line up both by brainstorming beforehand, and checking during the implementation process.

Overall, we think that we did a very solid job at all aspects of project management over this semester and are satisfied with our performance as a team.

VI. Lessons Learned

One lesson we learned was that although one can have confidence in modules individually using rigorous unit-testing with Verb, this does not save a team from having to rewrite interfaces after reconsidering functionality and resolve deeper edge cases that were not considered when developing the original testbenches. For example, when implementing our branch prediction system, even though we had completed the ROB significantly earlier, we still had to revise the interface to support passing back the real target address to update the BTB, which hadn't been considered earlier. Such modifications are inevitable, but hopefully as we become better designers and architects we will write better interfaces and testbenches.

Another lesson we learned was that unnecessary N-way implementations could significantly harm our clock period while not helping our CPI. For example, our G-Share was originally supposed to predict up to N branches in one cycle when we planned to have checkpoints that allowed for recovery from multi-branch-speculation. However, even after we abandoned this advanced feature we left this implementation in for a while and it significantly raised our minimum clock period due to the dependent for-loops involved.

A third lesson learned from this project is to emphasize communication and keep the team's goals consistently aligned. Trying to manage a group of engineers and keep everyone on the same page was found to be difficult due to varying schedules, responsibilities, and technical experience. It's important to consistently relay new information and updates throughout the design process and always ask questions when design decisions and their tradeoffs are not very clear. It's also important to keep the bigger picture in mind when introducing changes to module behavior, as one change can affect the whole system and require major refactoring in other parts of the system.