veriNOTsimple™ Design Report

Group 4

Angel Cheng (angelhyc), Brandon Zhang (bdwzhang), Gary Huang (ioi),
Lindsey Forche (lforche), Zachary De Rosia (zderosia)

Table of Contents

Table of Contents 2
Introduction 3
Design Overview 4
Basic Features 4
Advanced Features 7
Future Roadmap 11
Testing 12
Broader Societal Impacts 13
Evaluation & Analysis 13

Introduction

The veriNOTsimple™ is a Two-Way Superscalar Out-of-Order processor based on the MIPS R10K architecture implementing the RISC-V ISA. The processor was designed in six weeks, implementing a correlated branch predictor, a non-blocking two-way associative instruction cache with prefetching, an *N*-ported non-blocking Fully Associative Data Cache, a Load-Store Queue with forwarding and speculating on load dependencies, and an *N*-way Retire. It pursues a low clock period, achieving an average CPI of 2.1 and clock period of 7.5ns.

Advanced Features	**Status**
Two-Way Superscalar	Integrated
Correlated Branch Predictor	Integrated
Non-Blocking Two-Way Associative Instruction Cache with Prefetcher	Integrated
Load-Store Queue	Integrated
Data Forwarding from Loads to Stores	Integrated
Speculating on Load Dependencies	Integrated
Bad Pairs Table	Integrated
N-Ported Non-Blocking Fully Associative Data Cache	Integrated
Early Tag Broadcast	Fully implemented, not integrated - very small performance gains and insufficient testing
Victim Cache	Partially implemented, not integrated - functional for stores and typically loads, but may have issues with memory tags
Early Flush Broadcast	Fully implemented, not integrated - insufficient testing
HyperScalar™	Fully implemented, not integrated - insufficient testing

Table 1. Advanced Features

Design Overview

Figure 1 shows major components of the pipeline and their connections. The entire pipeline consists of six major stages containing more than 100 different modules.

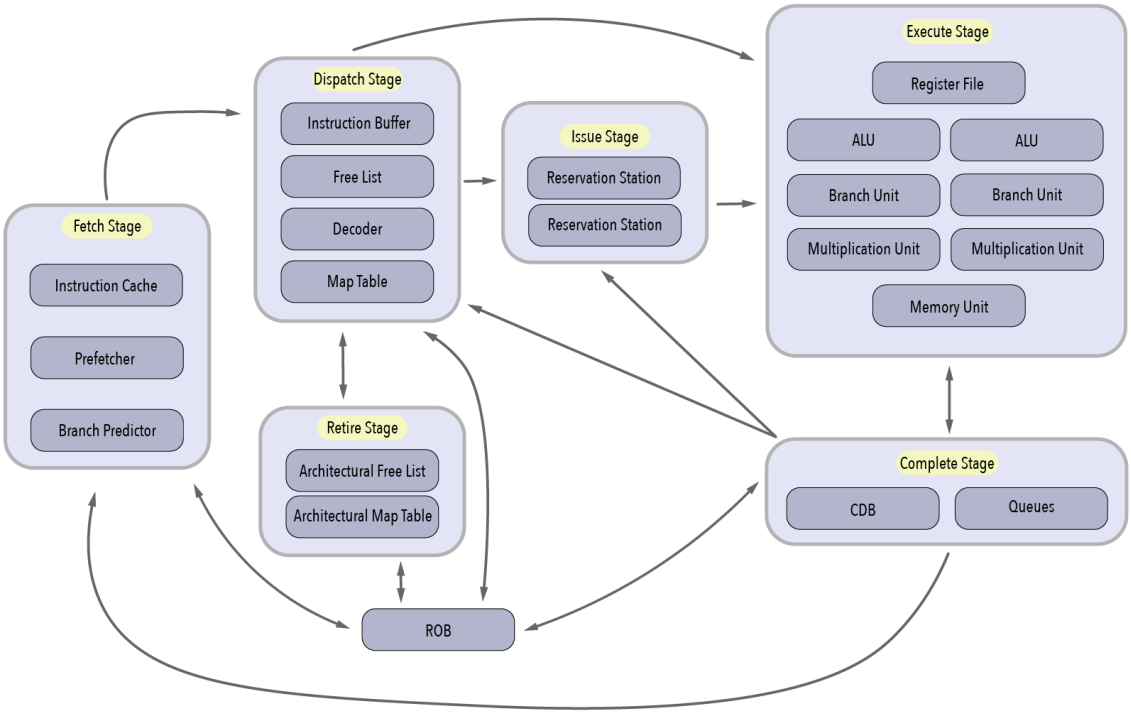

Fig 1. Processor Overview

Basic Features

3.1 Fetch

The Fetch stage contains an Instruction Cache and Branch Predictors to provide the processor with instructions. Every cycle, a maximum of two instructions can be retrieved from the Instruction Cache and sent to Dispatch. If the Branch Predictor predicts a taken branch or a flush occurs, Fetch discards extraneous instructions that are not part of the execution path.

3.1.1 Instruction Cache

Before building a custom Instruction Cache (I$), we used the provided I$ to get instructions from memory. Despite being fully functional, our strategy of targeting a low clock period meant high memory latency, and thus we designed a more sophisticated I$, as described in section 4.3.1.

3.1.2 Branch Predictor

The basic version of our Branch Predictor is implemented as a two-bit saturating counter. It is a four-state finite state machine, with each state corresponding to Strongly Taken, Taken, Not Taken and Strongly Not Taken, respectively. The former two output predictions for the incoming PC to be taken, while the latter two predict the opposite. After Complete sends back information on whether the branch is actually taken, the Branch Predictor modifies its state accordingly.

Included in the basic Branch Predictor is a basic Branch Target Buffer which contains 64 entries and is fully associative. It stores the known branches and their corresponding target PCs. After the branching

logic has been thoroughly tested, we designed an advanced Branch Predictor, which is described in more detail in section 4.2.

3.2 Dispatch

The Dispatch stage uses the Instruction Buffer, Decoder, Free List, and Map Table to decide what and where to send instructions: to the Reservation Stations, Reorder Buffer, or Load-Store Queue. If the Reorder Buffer, Load-Store Queue, and at least one of the Reservation Stations are not full and there are instructions ready to be sent, Dispatch will send them to the appropriate modules.

3.2.1 Instruction Buffer

The Instruction Buffer stores instructions that have been fetched but cannot be dispatched due to the Reservation Stations, Reorder Buffer, Load Queue or Store Queue being full. It is implemented as a shift register queue which can accept and output two instructions at a time to allow Two-Way Superscalar.

3.2.2 Decoder

The Decoder is built on the decoder from the previous project, with additional features to support our processor's new execute stage. The execute stage is broken up into different units, ALU, Multiplication, Branch, and Memory, therefore the decoder is responsible for signaling the Functional Unit type to allow the instruction to be sent to the correct unit. Because of this requirement from the execute stage, the decoder now not only decodes the instruction and the registers/offsets numbers, it also decodes which functional unit this instruction is going to be sent to.

3.2.3 Free List

The Free List is implemented as a stack and is initialized to all of the registers not initially in the Map Table. It is updated on Dispatch and Retire, and on a flush, the Architectural Free List will be copied over. With the Two-Way Superscalar feature, up to two registers are able to be placed and removed from the Free List using combinational logic. There is also forwarding within the Free List. If there is a read and write request to the Free List at the same time, the register that was supposed to be written to the Free List will be outputted as the register that is read from the Free List. Beyond these few details, the Free List is implemented in a similar way as what was discussed in lecture.

3.2.4 Map Table

The Map Table stores the physical registers corresponding with each architectural register and whether the values in those physical registers are valid. Other than allowing Two-Way Superscalar execution by taking in and outputting twice as many inputs and outputs, as well as forwarding tags from the first instruction to the source tags of the second instruction, the Map Table behaves almost exactly as was taught in the R10K lectures.

3.3 Issue

Since all of the preparation work is performed in Dispatch, the Issue stage simply houses two Reservation Stations (RS) and communicates with the rest of the pipeline.

3.3.1 Reservation Station

Instead of dual porting our RS to support Superscalar, we created a fast design that only allowed one instruction insertion and removal per cycle, and simply duplicated a copy for each Superscalar. Unlike traditional designs, the RS selects the instruction to issue based on their type, instead of their age in the Reorder Buffer. The RS prioritizes high latency and high dependency instructions such as loads and

multiplications, which yields decent results after testing. These design decisions match our main strategy of focusing on achieving a low clock period while maintaining sufficient efficiency.

Experimental attempts using other removal heuristics such as the tree-PLRU mentioned in section 4.3.3 showed no significant CPI improvements. Synthesis tools reported that internal Common Data Bus forwarding in the RS was a longest path bottleneck, so it was temporarily disabled. However, the forwarding was later re-enabled after we optimized Complete.

3.4 Execute

The Execute stage is the core of the processor. It contains the Physical Register File and four different types of Functional Units. Once an instruction arrives from Issue, Execute retrieves the register values from the Physical Register File and sends them off to the corresponding Functional Units.

3.4.1 Physical Register File

With four read ports and two write ports, the Physical Register File follows the industry standard for a Two-Way Superscalar processor. Although internal forwarding was added in the original design, it was quickly removed as it caused a long combinational path throughout the entire processor.

3.4.2 Functional Units

Following the strategy of the RS, we duplicated the same set of Functional Units (FU) for both Superscalar. Except the Memory Unit, all FUs have a throughput of one; thus, they will never stall Issue. Moreover, all FUs have a unified interface, which reduces the complexity of Execute.

The Arithmetic Logic Unit and Branch Unit both process instructions using only combinational logic. Their implementation strictly follows the RISC-V ISA and covers all non-memory instructions in the RV32I Base Integer Instruction Set. They are divided into two units for better testing.

The Multiplication Unit uses a four-staged version of the provided pipelined multiplier along with a buffer queue to achieve a throughput of one and latency of five.

Finally, the Memory Unit contains the Load-Store Queue and the Data Cache, later described with more details in section 4.3. The module itself is merely an enclosure for the two most complicated modules in the processor.

3.5 Complete

Along with Issue, the Complete stage serves as the second set of pipeline registers to Execute. After an instruction proceeds through the combinational logic in Execute, it is stored into one of the five queues in Complete based on the type of the instruction (loads and stores are stored into separate queues). Then, on each cycle, two instructions are selected based on their type and are dequeued and completed. These queues effectively handle the bottleneck of the two Common Data Buses by delaying the completion of generally less critical instructions.

Exceptions are made to the Branch Unit. Since there are only two Branch Units, their results are announced immediately to the Branch Predictor. We decided against updating the Branch Predictor in Retire since it required more routing and also had a longer latency. We speculated that the only benefit of an in-order update order would be negligible since it is improbable for two executions of the same branch to be out of program order.

3.6 Retire

The Retire stage was simplified such that it is simply a container for the Architectural Map Table and the Architectural Free List; however, Retire works very closely with the ROB, and for this reason, the ROB is included as part of Retire in this document. The basic flow of instructions in Retire is as follows: the ROB recognizes a completed instruction, and sends the necessary data to the Retire module, which then updates the Architectural Map Table and the Architectural Free List.

3.6.1 Reorder Buffer

The Reorder Buffer (ROB) is treated as the entry to the Retire stage, though is not directly located inside of the Retire module. The ROB can hold 32 entries, each of which is made up a variety of information on each dispatched instruction; notably, the Program Counter, Next Program Counter (if the instruction flushes), register tags, a complete signal, and miscellaneous information regarding if the instruction is a flush, store, load, illegal, or halt which may impact how the instruction is retired.

The ROB itself is implemented similarly to that described in lecture. To track instructions, the ROB relies on a head and tail pointer. If the head pointer is pointing to a completed instruction, it will send the appropriate data of the instruction to Retire. For non-branch and non-store instructions, the ROB can retire *N* instructions, though it was set to two in the final submission due to it limiting the clock period. In the case of a store instruction, the ROB checks if there are enough available ports on the Data Cache to retire that store. Lastly, due to the Architectural Free List, flushing was trivial: set the head and the tail to the same value and send out a flush signal to the other modules.

3.6.2 Architectural Map Table

The Architectural Map Table stores the physical registers corresponding with each architectural register from instructions that have retired. Other than allowing *N*-Way Superscalar execution by accepting *N* retiring instructions, the Architectural Map Table behaves almost exactly as was taught in the R10K lectures.

3.6.3 Architectural Free List

The Architectural Free List was designed to enable a quick transfer of the previous state of the Free List prior to the flush. Upon Retire, *N* physical registers are added to the Architectural Free List. It is a stack and is initialized to all of the registers not initially in the Map Table, which is the same as the Free List. Beyond these few details, the Architectural Free List is a copy of the Free List.

Advanced Features

4.1 Superscalar Execution

The processor primarily uses Two-Way Superscalar execution except for Retire which uses *N*-Way Superscalar execution. For most modules in the processor, Two-Way Superscalar required doubling the inputs, looping over the logic a second time, and/or duplicating modules. In Fetch, it required adding logic to decide which instructions to send to Dispatch and which to discard based on various factors such as the current and previous Program Counters, the results of the Branch Predictor, and whether Fetch is stalled. Making Retire *N*-Way required changing the Reorder Buffer, Architectural Map Table, and Architectural Free list to loop over their logic *N* times rather than two times.

4.2 Branch Predictor

The Branch Predictor is a correlated predictor (two-level predictor), with a combination of a two-bit saturating counter implementation for the pattern history table. The Branch Target Buffer is a 2-way set associative cache-like data structure. With the 2-way set associative implementation, look up time for the cache becomes O(1) time, rather than O(n) time (n = branch target buffer size). Another change to the branch target buffer is to add another attached column of branch history to the table, so that the new table would contain the branch PC, the target PC for this branch, and the branching history for this branch. The branch history is initialized to all 1's for branches that are taken the first time it is added to the branch target buffer, and all 0's if otherwise. In addition, each entry of the pattern history table is attached to a four state finite state machine, all initialized to weakly taken. This is to avoid the pattern history table changing its prediction for a pattern too fast.

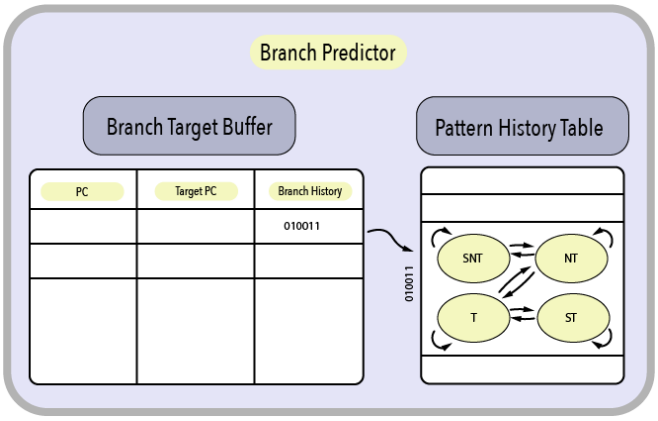

Fig 2. Diagram of Branch Predictor Implementation

Since our processor is Two-Way Superscalar, we have two branch predictors, one for each way. All the odd PCs will only go into one of the branch predictors, and the even PCs will go into the other. This design required us to add additional MUX logic in Fetch to ensure that the information from Complete goes into the correct Branch Predictor.

4.3 Memory Interface

4.3.1 Non-Blocking Two-Way Associative Instruction Cache with Prefetcher

The Instruction Cache (I$) is a Two-Way Associative cache that improves the latency of Fetch. Each cache block stores two instructions, which conveniently aligns with our Two-Way Superscalar pipeline. A Prefetcher is bundled within the I$ that exploits the sequential nature of instructions in memory. Once an address has been requested by Fetch, the Prefetcher begins marching through the subsequent address space, eight bytes at a time. A command is only sent to memory if an instruction does not already exist in the I$, which reduces the bus bandwidth consumed. If the march is too far ahead, it is stalled until requests from Fetch resumes. Similarly, if the requested PC is unexpected to the Prefetcher, it immediately restarts the march to avoid pollution of the I$. Introduction of the Prefetcher significantly increased the throughput of Fetch and aggressively pushed new instructions onto Dispatch almost every cycle.

4.3.2 Load-Store Queue with Forwarding and Speculating on Load Dependencies

The Load-Store Queue is implemented as two shift-register queues which manage load and store instructions to execute them out of order. It forwards store values to dependent loads which are smaller

or the same size and issues loads to the Data Cache speculatively, with additional logic to recover from incorrect speculation and prevent future incorrect speculation. It includes a Bad Pairs Table to limit flushes and uses priority selectors instead of for loops in multiple places to allow the Load-Store Queue to work in O(log(n)) time with n being the sizes of the Load and Store Queues. It also executes loads and stores over multiple cycles to allow a low clock period.

Upon dispatching load and store instructions, they are placed into the Load and Store Queues. When those instructions are issued to the Memory Unit, they enter the Bad Pairs Table which checks the loads against a table storing which load and store pairs led to flushing. If a load leads to a flush, it is marked as not ready until the corresponding store passes through the Bad Pairs Table or there are no stores older than the load in the Store Queue. If the load did not lead to a flush, then the load is marked as ready immediately. Each cycle, the Bad Pairs Table uses two priority selectors to select which ready loads to be executed, similar to the Reservation Stations.

In execution, stores place their values and addresses into their entries in the Store Queue and check if they alias with any younger loads. If they do, and the load has requested data from the data cache or received data from another store, the store is marked to flush upon retirement. Next, loads store their addresses in the Load Queue and search for any aliases with older stores. If there are any, and the store is at least as large as the load, then a priority selector is used to select the youngest store to receive data from and the data is forwarded to the load. If the store is not at least as large as the load, the load is marked as needing to be flushed to avoid needing logic to select which loads to request from the cache. If there are no aliases, the load is immediately requested from the cache. When loads receive data, they are marked as ready to be sent to Complete and, each cycle, the youngest and oldest loads are sent to Complete. This guarantees that the selection logic will select different loads without increasing the amount of logic required, and will usually be the same as selecting the oldest two loads since there are rarely more than two loads ready to complete in one cycle.

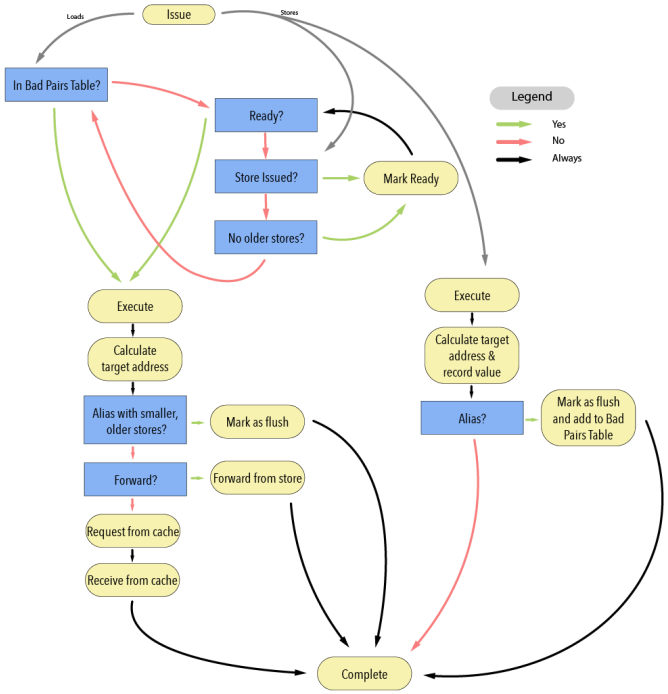

Fig 3. Load-Store Queue Control Flow

Figure 3 shows a flow chart of loads and stores through the Load-Store Queue. Upon retirement, loads and stores are removed from their respective queues and the pipeline is flushed if necessary, with stores being sent to the Data Cache.

4.3.3 N-Ported Non-Blocking Fully Associative Data Cache

The Data Cache (D$) supports a parameterizable number of load and store requests from the Load-Store Queue, all of which are governed under a set of guarantees. It assures that all store requests are final, but load requests are discarded on a flush, and operations on the same cache line will appear in-order. To simplify the testing of memory instructions, early versions of the D$ merely served as an interface with Memory. Using this fixed guarantee, we were able to easily upgrade the D$ without affecting the rest of the memory system.

Since there is a limit of 256 bytes on its size, we designed the final version of our D$ to be fully associative. Old cache blocks are evicted using the tree-PLRU policy. We selected tree-PLRU for its good balance between speed, simplicity, and performance, and implemented it based on *Wikipedia*. Regarding stores, we used write-back and allocate-on-write policies for better temporal locality and low memory bandwidth usage. To achieve maximum performance, all requests are processed out of order (while maintaining the guarantee) using five different queues demonstrated in Figure 4.

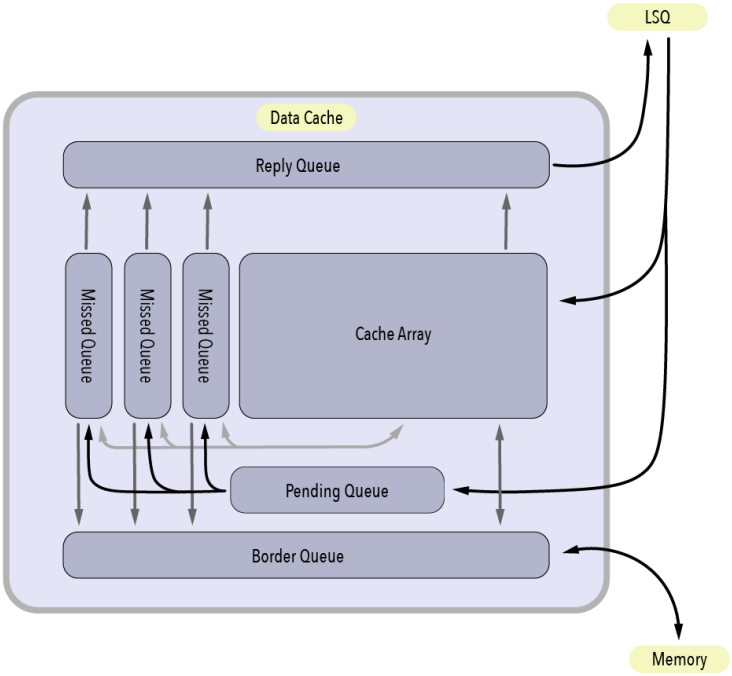

Fig 4. Data Cache Diagram

All incoming requests are first checked with the Cache Array, which contains all cache blocks, for any tag match. If there is a hit and the Pending Queue is empty, the result is immediately sent to the Reply Queue to be sent directly back as a load response. This process all fits within one clock cycle, ensuring a fast common case. Otherwise, if the Pending Queue is not empty, the request is enqueued into the Pending Queue and delayed for processing. However, if the Pending Queue is empty, the request can be selected to be placed into one of the many parameterizable Missed Queues. Once allocated, each Missed Queue

can only contain requests regarding the same cache block. This restriction delivers the order guarantee promised to the Load-Store Queue, while achieving high performance on independent operations.

Allocating a Missed Queue from empty to occupied adds a new load command into the Border Queue, which bears the sole responsibility of communication with Memory. Each new cache block returned from memory evicts an older block using tree-PLRU, and the awaiting Missed Queue will begin emptying its content to the Reply Queue. Store requests are handled similarly, except they ignore the flush signal and only modify the according cache block instead of continuing onto the Reply Queue. Overall, our non-blocking D$ is able to efficiently process large amounts of simultaneous requests even under high memory utilization.

4.3.4 Memory Controller

Since the memory bus only accepts one command per cycle, we built a special controller to manage the traffic. As both the Instruction and Data Cache obey the common memory interface using tags, the controller is simply an arbiter. Originally we planned to incorporate a victim cache inside the controller, following the same interface. However, we ran out of time and believed that adding it would not be as beneficial as the other features we planned.

Future Roadmap

5.1 Early Tag Broadcast

Early Tag Broadcast with arithmetic, logical, and branch operations is implemented and works in the "early_tag_broadcast" git branch, but was not included in the final design. With our implementation of Early Tag Broadcast, the Common Data Bus does not broadcast completed tags, the Issue stage marks instructions working with Early Tag Broadcast as ready in the Reservation Stations since their values are written to the Physical Register File by the time the next instructions are issued. However, it was not included in the final design due to showing little improvement in performance and insufficient testing. Early Tag Broadcast could be included in future designs after more testing.

5.2 Victim Cache

Our Victim Cache (V$) was designed to be entirely invisible to the rest of the processor, such that it could be removed or added without needing to alter any modules, and would sit within the Memory Controller and communicate with the Data Cache. This means that it would intercept memory requests made by the Data Cache and decide whether to send them to the Memory regardless, or send a response of its own. Unfortunately, due to difficulties with tag priority between the Memory and the V$ with load requests as well as time constraints, this was not included in the final submission.

5.3 Early Flush Broadcast

To reduce the penalty of pipeline flushes, we implemented Early Flush Broadcast from Execute to Fetch. With slight modifications to Fetch and Execute, our implementation of Early Flush Broadcast showed a two-cycle decrease in flush penalty on average. However, the feature was not included in the final version of the processor due to insufficient testing.

5.4 HyperScalar™

HyperScalar™ allowed us to execute and complete two additional instructions each cycle on top of Two-Way Superscalar execution. With HyperScalar™, Dispatch dispatched instructions that did not write to the Physical Register File to Reservation Stations separate from the standard ones. These Reservation

Stations also had separate functional units, except for the Memory Unit which the HyperScalar™ Reservations Stations issued store instructions to. After HyperScalar™ execution, the instructions were immediately marked as complete in the Reorder Buffer without writing to the Physical Register File. HyperScalar™ was not included in the final version of the processor due to insufficient testing. We believe HyperScalar™ to be a clever design which should be included in future designs since it would boost performance without significant complexity or structural changes.

Testing

6.1 Module Level Testing

The first level of testing was module level testing. Upon the creation of a module, a testbench was made. These were commonly used and modified when the modules were changed or when a problem arose at a higher level of the pipeline. They were also useful when verifying if all of the components synthesized correctly. We targeted full coverage for all of the testbenches, but these proved to be largely obsolete towards the end of the design process.

6.2 Stage Level Testing

The next level of testing was stage level testing. Once the individual modules and the stage modules were created, a stage level testbench was created. Unlike the module level testbenches, the primary use of these testbenches was to verify that the modules compiled and synthesized without error. These also were used less towards the end of the project when we transitioned to pipeline level testing.

6.3 Pipeline Level Testing

The final level of testing was pipeline level testing. At this level, we transitioned to using the assembly and C programs. When the pipeline was just fully initialized, smaller assembly programs like rv32_halt.s were used. Once all of the assembly programs were working correctly, the C programs were used. We added several custom C programs as well to add even more pressure to the memory specifically. These programs exposed more bugs due to the increased pressure that they put on the various components. It was common for a problem to come up when using these programs. There would occasionally be a combination of events that was hard to replicate within a testbench, so these programs were our main way to test in the final portion of the project.

The first external C programs attempted were from the SPEC CINT92 test suite. Unfortunately, since many of these programs are too large or require too much I/O, none were able to be run due to time constraints, though 026.compress was the most realistic attempt. The most interesting program that was successfully run on the processor was an interpreter for the esoteric programming language BrainF*ck. The interpreter uses a char* set at the beginning of main to represent the input code, and executes the code, printing the output into memory. While the print function does not entirely work as expected, any inputted BrainF*ck code is executed correctly. Other programs in our custom suite include a hashtable and manipulation for harshly testing memory operations, pseudo-random execution of bitwise operations, and a simple matrix Multiply-Accumulate program.

When we transitioned to this level of testing, we also transitioned to using hierarchical synthesis. With hierarchical synthesis, we could treat the various components as black boxes and focus on the correctness after synthesis without a long wait time. For the last week of testing and optimizing, we transitioned back to non-hierarchical synthesis to lower our clock period.

6.4 Testing Scripts

To thoroughly test our programs, we used a variety of scripts. We started with the scripts that were used to test Project 3. These scripts included check_ref.bash, which tested a single test program running on our processor against the same test program running on the base multi-cycle Project 3 processor, and check_all.bash, which ran through all of the test programs sequentially on both processors. Both scripts checked for correctness, which was very valuable in our initial testing of the programs.

While check_all.bash was useful, due to the single threaded execution of bash scripts, it was also rather slow. For this reason, a new script named fast_check_all.bash was created. This script attempted to simulate all test files simultaneously, but due to temporary files produced by VCS, this was not possible to do in a single repository. Therefore, the entire repository was cloned for every test file, then they were asynchronously executed and the output returned to the main folder. While this was only ever done for simulation and no synthesis files were to be copied, the CAEN lab computers were not very receptive as dozens of Gigabytes would end up being copied on every execution and this caused many problems.

Broader Societal Impacts

We achieved a low clock period and CPI, but it was a learning experience, so beyond focusing on the project tasks, we did not focus on any major ways to impact society. However, we did incorporate a Functional Unit specific to branches and an advanced Branch Predictor. Both of these features are beneficial to the processor and the overall society. The Branch Unit is combinational, so it can detect a branch as soon as it enters the Functional Unit, and the Branch Predictor had an average of 78.81% accuracy. Due to these two components, we are able to recognize a branch quickly and resolve any misprediction quickly. Overall, we may not have achieved an energy efficient machine; however, we did achieve a swift branch clean up.

Evaluation & Analysis

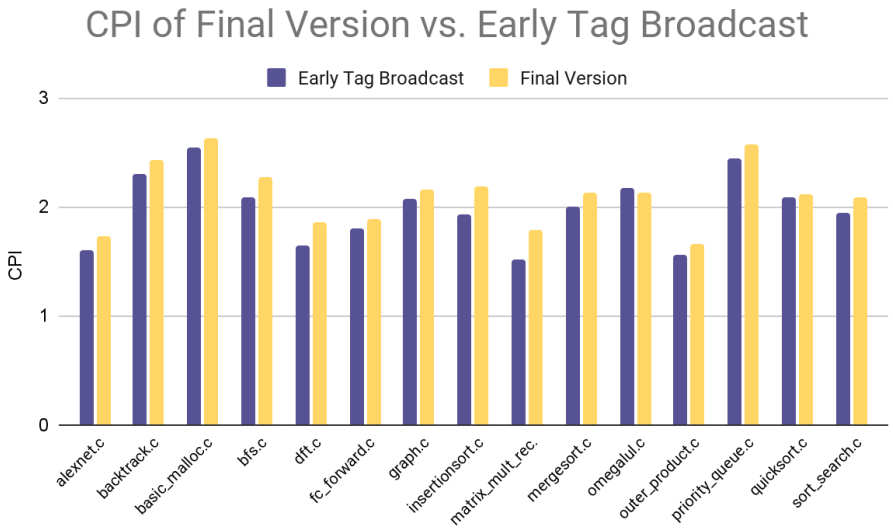

Fig 5. CPI Comparison of Final Version vs. Early Tag Broadcast

Figure 5 shows that adding Early Tag Broadcast for ALU and branch operations only gave a minor improvement to the processor's CPI compared to without it. This made it not worth the testing or potential decrease in clock speed it might have required, leading to us not including it in our final design.

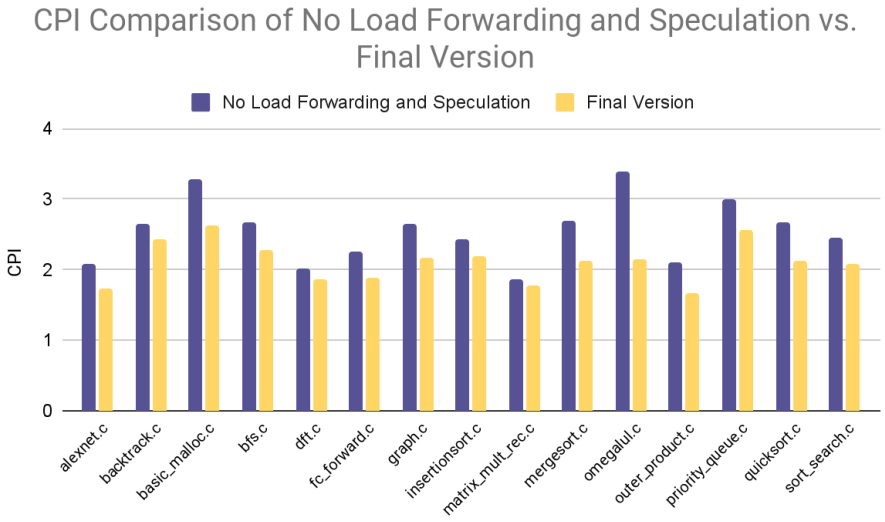

Fig 6. Without Load Forwarding and Speculation vs Final Version

As shown in Figure 6, forwarding stores to loads and speculating on load dependencies improved performance in every test case. The improvement across all of the .c files was 16.4%, but this improvement was most likely not worth the time spent adding this functionality to the processor and ensuring it worked.

Number of Flushes from Incorrect Load Speculation			
Program	Without Bad Pairs Table	Final Version	X Improvement
alexnet.c	5413	77	70.3
backtrack.c	238	13	18.3
basic_malloc.c	22	13	1.7
bfs.c	113	30	3.8
dft.c	2605	651	4.0
fc_forward.c	6	3	2.0
graph.c	318	52	6.1
insertionsort.c	5048	1573	3.2
matrix_mult_rec.c	10	1	10.0
mergesort.c	176	21	8.4
omegalul.c	3	3	1.0
outer_product.c	6717	30	223.9
priority_queue.c	32	14	2.3
quicksort.c	1083	213	5.1
sort_search.c	5175	7	739.3

Table 2. Flushes from Load Mis-speculation

Table 2 shows that the Bad Pairs Table significantly reduced the number of flushes from mis-speculated loads by storing loads until the stores that aliased with them were issued. In total, the Bad Pairs Table reduced the number of flushes from mis-speculated loads by about 10x..

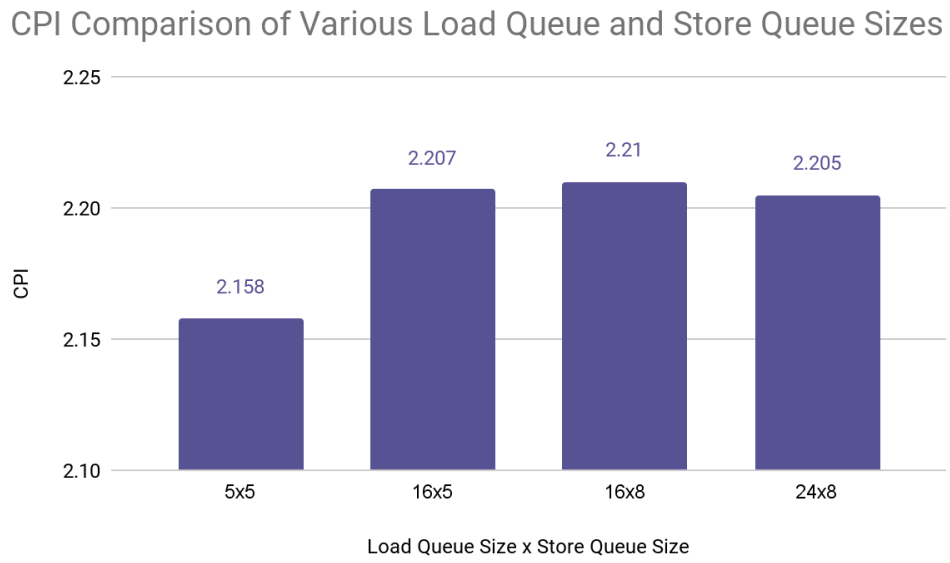

Fig 7. Without Load Forwarding and Speculation vs. Final Version

When deciding on the size for the Load Queue and Store Queue, we tested with 4 combinations, primarily to do with changes in the Load Queue size. We left the Store Queue small in an attempt to reduce the chance of a store alias. Interestingly, we found that the smallest size that we tested for both modules proved to be the best in performance. Unfortunately, we were unable to test lower sizes, as our modules would not compile with smaller queues.

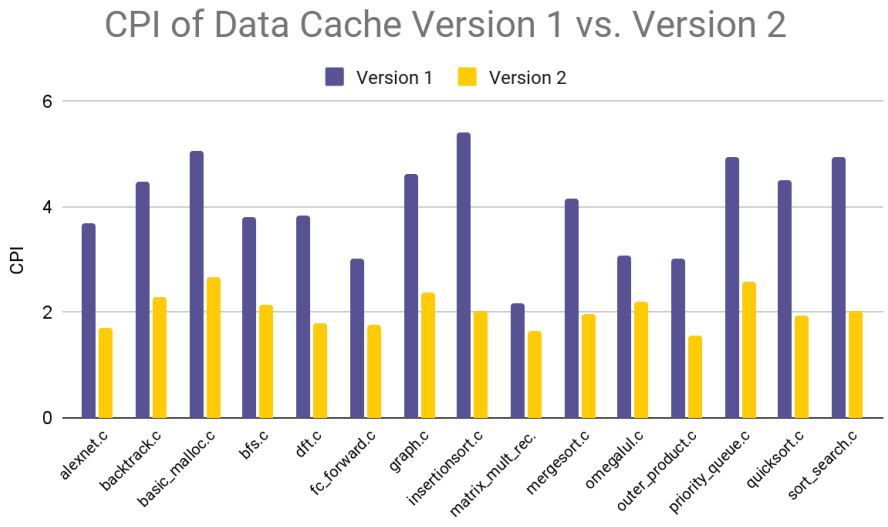

Fig 8. Without Load Forwarding and Speculation vs Final Version

Throughout development, we had created two versions of the data cache. The first version was created as a simple interface for the Load-Store Queue; it merely satisfied the requirements and forwarded all requests to Memory. The simplicity of this initial design provided us with a great platform for testing the rest of the memory system, since we are almost certain that any operations committed to the Data Cache

are completed without errors. After upgrading to the final version of our Data Cache, we observed a significant improvement in the overall CPI metrics of the processor. From Figure 8, we can see that for almost all benchmarks, Data Cache Version 2 reduced the CPI by approximately 2x.

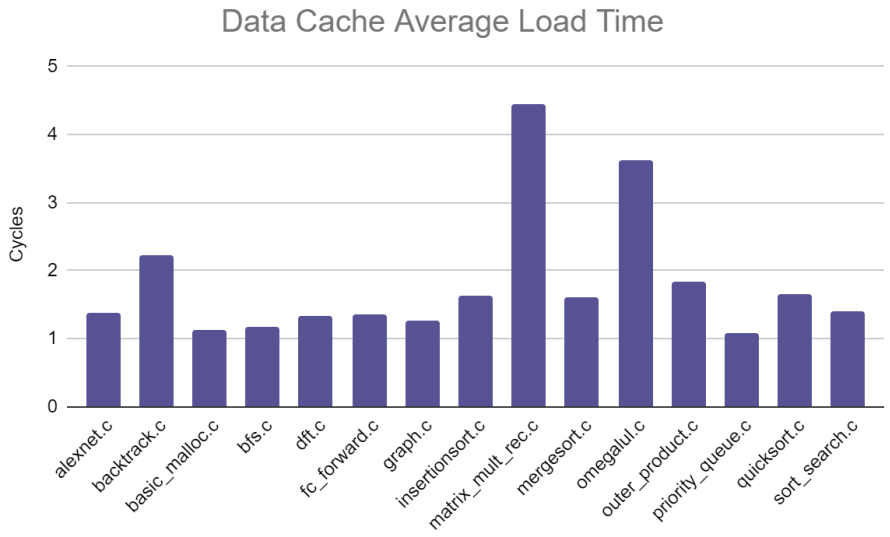

Fig 9. Data Cache Average Load Time

While optimizing the CPI of our processor, we added many metrics which are recorded during the execution of test programs. For example, we track the movement of each instruction as they flow through the pipeline, noting down how long they spend in each stage. Using this data, we calculated the average number of cycles it takes the Data Cache to respond to a load request, including both cache hits and misses, presented in Figure 9. The chart shows that the Data Cache has an overall excellent latency for a variety of applications.

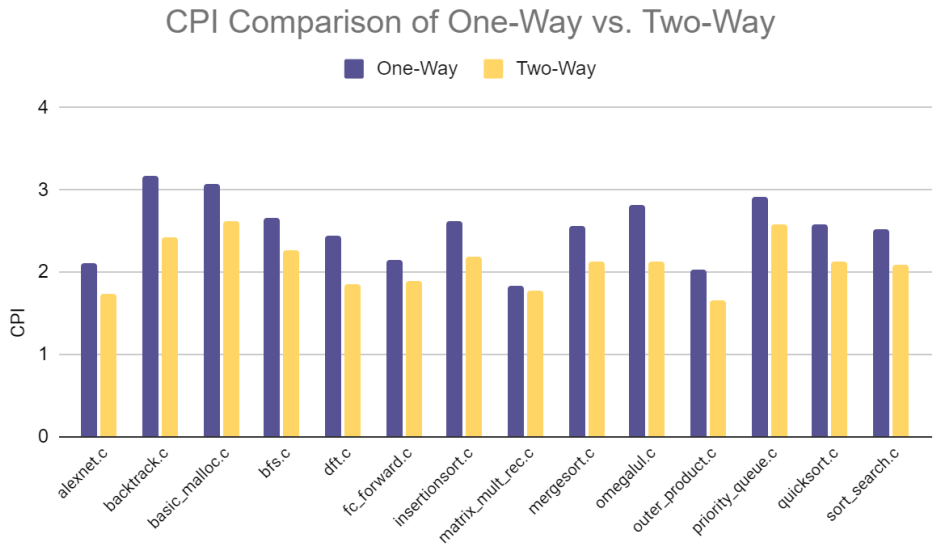

Fig 10. One-Way vs. Two-Way Superscalar

Two-Way Superscalar was implemented for this project. For the sample programs above, it does appear that there was a slight improvement in the CPI due to using Two-Way Superscalar over One-Way Superscalar. It does not have as large of an improvement as expected. It is likely that this improvement was not larger due to data hazards, since there are many loads, stores, and data dependencies in these files.

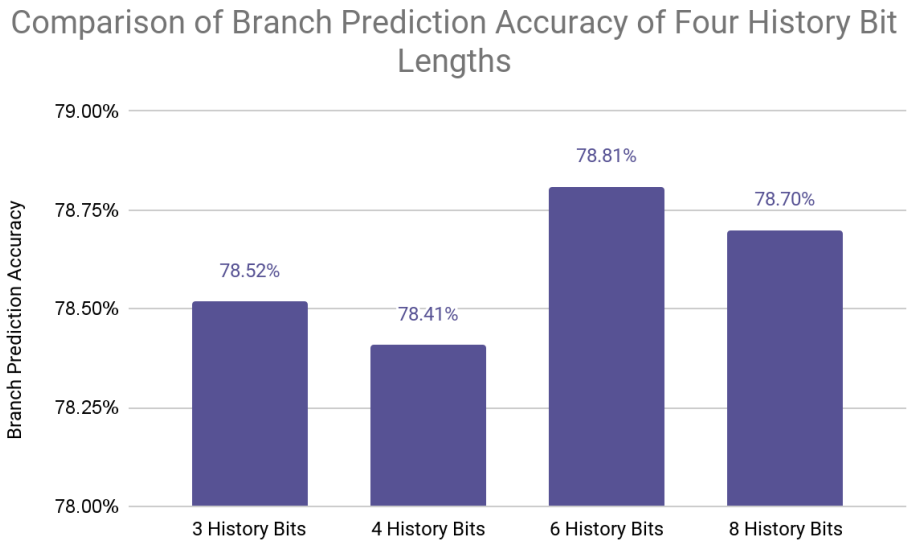

Fig 11. Bar chart of the average branch prediction accuracies for different branch history bits

We did experiments with varying amounts of history bits for the branch predictor history table, and recorded the accuracy for each program, and averaged the accuracy. Our accuracy is calculated by counting the number of correct branches over all branches that the complete stage receives – that is, after the complete stage checks if the npc is the same as calculated. We tested [3, 4, 6, 8] bits, and at the end, 6 history bits had the highest performance (78.81% accuracy) out of these parameters, therefore we decided to go with 6 history bits for our final branch predictor implementation.

It is important to note that some programs that are very short would have lower accuracies, since all branches would miss at least once due to cold start. By going through the assembly files and the output of our processor by hand, the short programs have achieved the highest accuracy possible, but some of them still have only 30 - 40 % accuracy. If we removed these files in the average accuracy, the 6 history bits predictor can achieve 85.27% (ignoring omegalul.c and fc_forward.c which are performing 33% and 52% since all misses are cold starts).

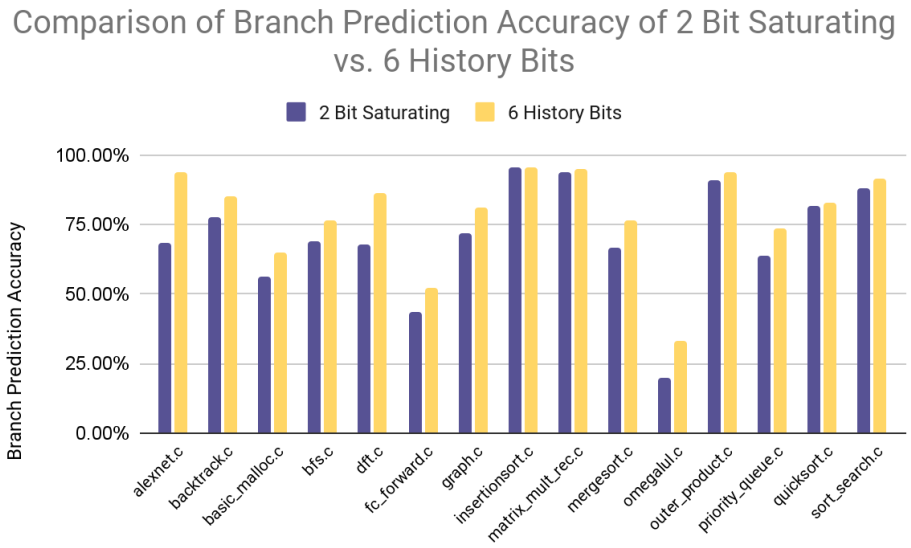

Fig 12. Branch prediction accuracies between a 2 bit saturating predictor and a pattern based predictor

The original branch predictor is a 2 bit saturating counter, and our final advanced branch predictor is the 6 bit pattern based predictor. As seen from the graph, using the 6 bit pattern based predictor increased the branch prediction accuracy significantly for every program.

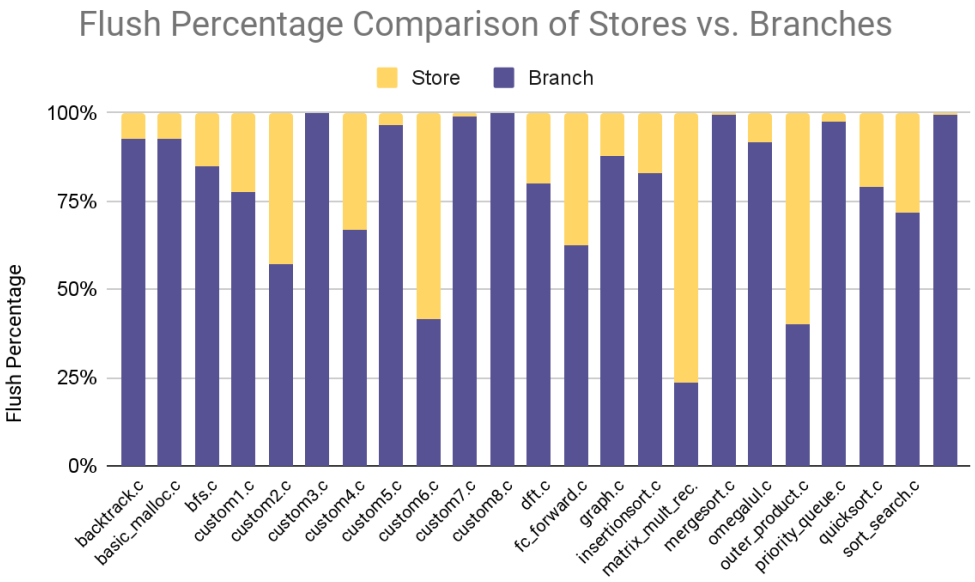

Fig 13. Flush Percentage graph

Based on Figure 13, the majority of flushes came from branches. While the Branch Predictor did decrease the number flushes, there are still flushes that occur. With an even better Branch Predictor, the overall percentage of flushes could decrease. There are also flushes due to stores. The number of flushes due to

stores would be higher, but with the inclusion of the Bad Pairs Table, the number of flushes due to stores is relatively small for all of the test programs.

During testing, we set the Reservation Station and Instruction Buffer sizes to 16 and the Reorder Buffer size to 32. It did appear from these sizes that only the Reorder Buffer had stalls. There would be stalls in the Reorder Buffer when there are long latency instructions in the Function Units. The Reorder Buffer was full most of the time, and when the size was adjusted to 64, it created stalls in the Reservation Station and increased the CPI. We decided to stay with a size of 32 for the Reorder Buffer due to the stalls that it passed to the Reservation Station. The sizes of the Reservation Station and Instruction Buffer could potentially be decreased. The Reservation Station generally would have stalls when there are data dependencies; however, with the size 16, we did not see any stalls occur on this structure. The Instruction Buffer generally was not filled up because the Load-Store Queue would be filled up first then cause the Instruction Buffer to start filling up, but this did not seem to cause stalls on the Instruction Buffer, so we did not change its size.

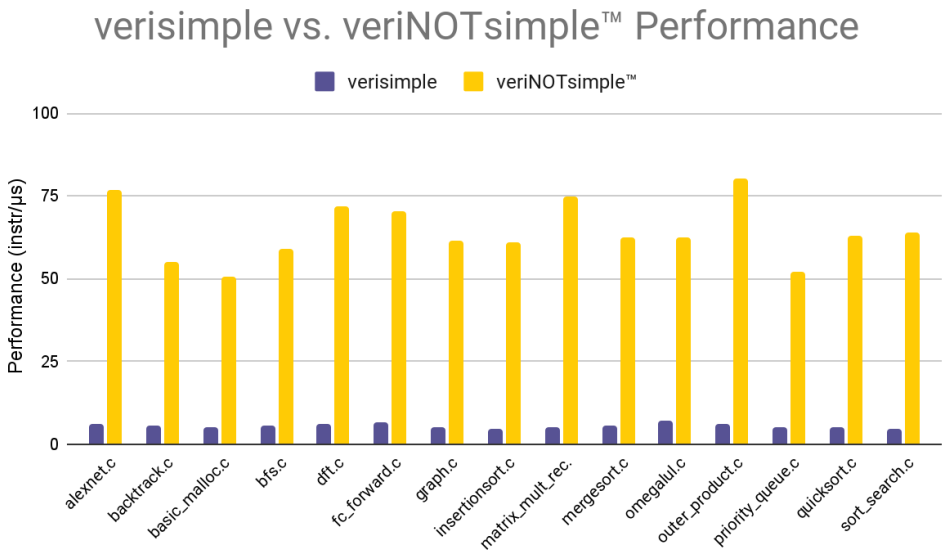

Fig 14. Performance Comparison between In Order Pipeline and Our Processor

Overall, using the inverse Iron Law, we observed a significant increase in performance, as measured by the average number of instructions processed per microsecond, compared against the in-order processor. Since the in-order processor must access memory in one cycle, we assumed its lowest clock period to be the latency of memory, which is 100ns. The juxtaposition in Figure 14 displays the massive advantage of our veriNOTsimple™ processor with its CPI of 2.1 and clock period of 7.5ns in a variety of applications. Further analysis shows that our processor excels in compute intensive benchmarks such as alexnet.c and outer_product.c, which is expected due to our low latency, low clock period functional units. Moreover, our robust memory system and out-of-order speculation heuristics effectively concealed the long latency of memory operations even on high memory pressure benchmarks such as backtrack.c and insertionsort.c.

From the very beginning, our strategy was to focus on constructing a fast processor with a low clock period, which has proven to be successful by our favorable performance results. Measuring and recording clock periods in synthesis tools early on in the project timeline gave us better understandings of the bottlenecks and hidden penalties of various SystemVerilog programming practices. We were fortunate to

have a functional processor before the final week, and were able to dedicate most of that time to testing, bug fixing, and optimizations. To reduce the clock period, we carefully examined all report files generated by synthesis tools to locate longest paths in the pipeline and eliminated them by either simplifying the logic, removing unexpected dependencies, or introducing pipeline registers. To reduce the CPI, we added many performance monitoring metrics and implemented experimental features such as Early Tag Broadcast and HyperScalar™, which were unfortunately left out of the final submission due to insufficient testing. Ultimately, this project gave us insight into the full development cycle of a modern processor and its most critical components, while challenging us with numerous design decisions which shaped the final destination of our processor, the veriNOTsimple™.