Design and Analysis of an R10K Superscalar Processor (Apr. 2018)

Alec Beljanski, *Student*, Carson Boden, *Student*, Alex Kisil, *Student*,
Kyle May, *Student*, Alisha Patel, *Student*

***Abstract*—This report details the design and implementation of an out-of-order microarchitecture that supports a subset of the Alpha64 instruction set. At a high level, the design included an R10K variant out-of-order core, 4-way superscalar execution, advanced branch prediction, and memory hierarchy enhancements including instruction prefetching, non-blocking caches, victim caches, and configurable memory priority. A visual debugger provided invaluable assistance in debugging the design, while the team remained organized and accountable via Trello. Final performance in the processor was significantly set back due to synthesis errors in the victim cache and return address stack, which were ultimately removed before submission. The average cycles per instruction (CPI) on performance oriented benchmarks averages at 2.92. The design is pipelined well, able to execute instructions with a clock period of 9.1ns.**

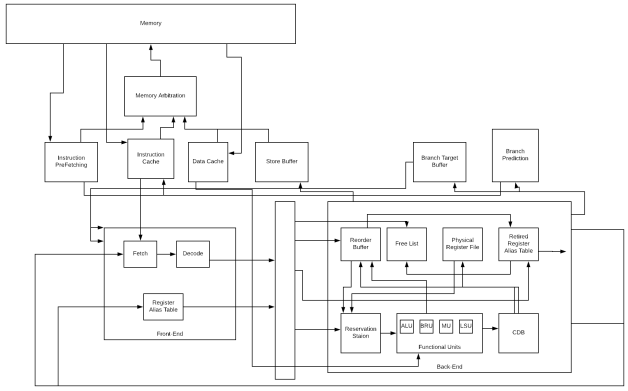

Fig. 1. The eight-stage final processor design has a complex layout to implement the R10K out-of-order algorithm.

I. Project Goals

THIS semester, five team members set out to design, build, and test an out-of-order processor that supports the Alpha64 instruction set. The four base requirements must be met:

1) out-of-order processor design
2) instruction and data caches
3) functional units
4) branch and address prediction

The eight stage pipeline in the final project incorporates these features. In addition, some advanced features must be implemented. The advanced features present in our pipeline are:

1) four-way superscalar
2) hardware prefetching
3) non-blocking data cache
4) advanced branch predictors

In addition to pipeline features, a GUI debugger has been built and extensively used. SystemVerilog and Synopsys tools were used for the processor design, simulation, and synthesis. Our team used other applications like Vim, Atom, Git, shell scripting, Trello, Google Calendar, and Vizhubs to help stay organized and communicate effectively.

II. Basic Modules

Many modules were given in past projects, meaning they required few, if any, changes to function correctly within the final pipeline. These modules and their alterations are listed below.

A. Arithmetic Logic Unit

The arithmetic logic unit (ALU) is an execution unit in the processor for simple mathematical operations, such as addition, subtraction, bitwise operations, and bit-shifts. Four ALUs were used in our design to accommodate the four-way superscalar nature of the processor.

B. Multiplying Unit

The multiplying unit, or MU, is a staple in any modern processor as it enables multiplication instructions. The multiplying unit remains unchanged from Project 2 in the processor.

C. Fetch Stage

The fetch stage is responsible for fetching the program counter (PC) for the current instruction and incrementing the PC appropriately. This module had to be parameterized from Project 3 to work with four-way superscalar, but it is otherwise unchanged.

D. Decode Stage

After instructions are fetched, they are broken up into several parts by the decode stage. This module was altered from Project 3 to work with four-way superscalar.

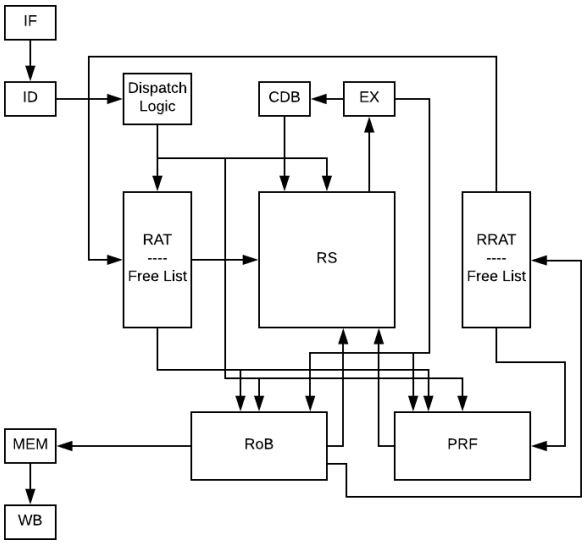

Fig. 2. The execution stage uses the majority of the R10K modules.

E. Branch Resolution Unit

The branch resolution unit, or BRU determines the branch direction and branch target address. The branch direction and address were calculated using the modules *brcond* and *alu* from Project 3.

III. Standard Modules

Many out-of-order processors use similar algorithms and modules in the R10K scheme. Our processor is no exception. As a result, some modules in the final project are no different than any other modern processor. A brief overview of their functionalities are listed here.

A. Reorder Buffer

The reorder buffer, or ROB, maintains the original program order to commit the instructions in that order. The main requirements were to parameterize the number of committing and dispatching instructions as well as how many execution units completed each cycle.

B. Load-Store Unit

The load-store unit (LSU) prepares load and store instructions for memory operations. The unit consists of two main entities: the load buffer and the store queue. The load buffer requests a load from memory once all instructions the load depends on have executed. The store queue sends stores off to memory as soon as their addresses have been calculated.

C. Free List

The free list keeps track of what physical registers are in use. In addition, the free list sends out the next four available physical registers for instructions to use during execution.

D. Physical Register File

The physical register file (PRF) handles all read and write operations to the register file. In dispatch, register reads occur. Specifically, dispatching instructions request reads for their source registers. After execution, a write to the destination register in PRF occurs while in commit, registers that are no long used are freed.

E. Register Alias Table

The register alias table (RAT) holds translations between the actual and local destination registers of operations. When an instruction reaches the RAT, it translates the source operands to local aliases and updates the RAT to hold the new translation for its destination register.

F. Retirement Register Alias Table

The retirement register alias table (RRAT) is structurally similar to the RAT. However, the RRAT only updates when instructions commit, meaning it will contain the architected state of the register file at any given time. Thus, the RRAT acts as a snapshot of the program. On a branch mispredict, where the RAT is no longer valid, the RRAT can overwrite the RAT, and restore the architected state.

IV. Unique Modules

While some modules are standard parts of every pipeline, others require specific interfacing or functionality to work within a specific processor. The following modules are required for the final processor, however they have been heavily modified for a specific use case.

A. Common Data Bus Arbiter

The common data bus (CDB) Arbiter chooses which completed instructions can broadcast to the reservation station. The CDB broadcasts the physical register number and the data for that physical register so the reservation stations know what data is available and if they are ready to go to the reorder buffer (ROB). The processor uses a system of cascading priority selectors to determine what instructions to broadcast.

Originally, execution units would request the CDB for the same cycle that it would want to broadcast its result causing execution units to schedule less effectively. To optimize this behavior, a change to the CDB triggers requests a cycle early, determining if the execution unit would be available next cycle. This change is discussed later in the Analysis section.

B. Reservation Station

Instructions wait in the reservation station, or RS, for execution. Each instruction has operands and a functional unit that it requests once the operands are ready. The RS creates an interface for instructions to wait for their data then wake up, or request an execution unit. When a unit is available, the instruction is purged from the reservation station and sent off to the corresponding execution unit where it will eventually be broadcast to all other waiting instructions.

The RS designed for our processor is unique in that it does not hold all types of instructions waiting to be executed. Instead, it only holds instructions that require the multiplying unit (MU), branch resolution unit (BRU), or the arithmetic logic unit (ALU). Loads and stores, which require the load-store unit (LSU) are sent to it directly and instructions that do not require any function unit are never placed into the reservation station.

Furthermore, the reservation station uses logic to wake up when the corresponding operands are being broadcast, rather than after they have been written to the RS entry. This increases performance, since the RS entry can issue a full cycle earlier, provided there is a functional unit available.

V. Memory Hierarchy

A. Cache

Because retrieving data from memory is a time-demanding process, a cache is used as a smaller, faster, memory bank that holds frequently referenced data. Project 3 provided a direct-mapped, write-through, allocate-on-write cache.

Some changes were made to support a superscalar processor. Mainly, the read and write ports had to be parameterized such that multiple instructions and memory can read and write to the cache simultaneously. The processor uses two 32-block caches with 8-byte blocks: one for the data cache, and one for the instruction cache.

B. Data Cache Controller

The data cache controller works with the cache to signal when memory is requested for a load or a store instruction. While the cache holds the data to be retrieved, the data cache controller decides what is retrieved or stored and whether it is valid.

C. Instruction Cache Controller

Much like the data cache controller, the instruction cache controller interfaces with the cache to signal when data needs to be read. Unlike the data cache, however, the instruction cache does not allow the processor to write to memory. Since the processor does not support self-modifying code, the instruction cache does not allow writes to instructions within it.

D. Memory Arbiter

The modules that require interfacing with memory are the data cache, store buffer, instruction cache, and prefetcher. However, there is a hierarchy of importance in terms of which modules need to request memory. This hierarchy, as seen in Fig. 3, is designed to maximize performance and grants access to memory bus in the following order:

1) Data Cache
2) Instruction Cache
3) Store Buffer
4) Prefetcher

By prioritizing memory accesses in this order, the processor ensures that loads are sent to memory over reading instructions and that fetching instructions is prioritized over other memory operations. This optimization reduces the number of stalls due to memory operations, increasing performance.

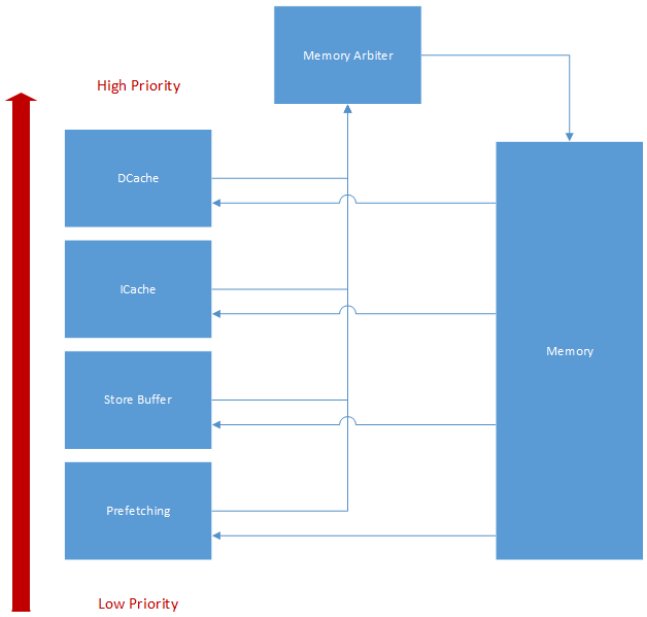

Fig. 3. The memory arbiter uses the priority list to determine which module to give memory bus access.

E. Miss Status Handling Register

The miss status handling register (MSHR) holds information for memory requests that miss in the cache. The MSHR allows the cache to be non-blocking, meaning it can miss and still service requests while waiting for the data to be returned from memory. The register assigns misses randomly into the register and requests the memory bus with the fixed priority selector. Ideally, the MSHR would request memory in the same order they come in, but our MSHR was not made with this optimization in mind.

F. Hardware Prefetcher

The prefetching module requests instructions from memory when the memory bus is idle. By prefetching instructions, the processor avoids incurring the latency of fetching an instruction normally. As a result, performance improves quite a bit in some cases. After modifying the size of the prefetch buffer and the number of block its fetching, a buffer size of four and block size of four were chosen.

G. Store Buffer

The store buffer holds onto store instructions and waits until memory is not required by higher priority modules before sending the data off to memory. The higher priority modules, the instruction cache and data cache, can stall the processor and reduce performance without access to memory. Thus, the store buffer provides a waiting area for non-essential memory writes, giving the more important modules free reign. When both the instruction cache and data cache are idle, the store

buffer can send its stores onto the memory bus, maximizing performance.

H. Victim Cache

As caches sizes are limited, it is natural for some commonly used data to be evicted from them due to certain access patterns leading to that data not being referenced recently. The problem is especially apparent in direct-mapped caches, much like the data and instruction caches we used for our processor.

To help alleviate this problem, one or more victim caches may be integrated into the memory hierarchy. These are generally a fraction of the size of the L1 caches and are fully associative. Memory that gets evicted from the L1 cache will then be placed in the victim cache, where an LRU (least-recently-used) or pseudo-LRU policy will be maintained for eviction. Therefore, if several frequently used blocks of data map to the same line in the direct-mapped cache, they will reside in the victim cache in case they are referenced again.

For our project, we designed a fully-associative victim cache containing four 8-byte blocks to be instantiated twice: once for the data cache and once for the instruction cache. While the design greatly improved performance in simulation, it was ultimately omitted from the final project due to causing problems with incorrect functionality in synthesis.

VI. Branch Prediction

Branch predictors are an essential component to any modern processor. Branch predictors allow a processor to speculatively execute more instructions which improves performance if those speculations are correct. The following sections highlight the different types of branch predictors tested in the processor.

A. Bimodal Predictor

The one-bit bimodal predictor is a simple predictor that assumes a branch will jump in the same direction it did when it was last seen. As such, if a branch is taken, the bimodal predictor will assume it will be taken the next time it is seen.

The two-bit bimodal predictor is similar to the one-bit predictor, but it has four distinct states instead of two. The four states allow for branches to predicted strongly and weakly taken and not taken, or that branches can be confirmed not-taken once and still predicted taken.

B. Branch Target Buffer

The branch target buffer (BTB) is a cache for branch prediction targets. Whenever the processor encounters a branch that is predicted taken, the BTB determines which address the branch might go to. Our processor uses a directly mapped BTB using the last five bits of the branch instruction as an index into the BTB.

C. Local History Predictor

The local history predictor uses the history of a branch in order to predict whether the next branch should be taken. It uses the results of whether or not that specific branch was taken to update the local history for that specific branch. In order to predict if a branch should be taken, it first uses the branch address to see what its local history is, then uses the local history to index into a table predicting whether the branch should be taken.

D. Return Address Stack

The return address stack, or RAS, uses the return instruction to predict the branch target location. Since functions can be called from many locations, using only the BTB for function calls will have poor results.

The RAS stores the address of a function call in order to predict the return location when the return instruction is eventually called. The addresses are stored in a stack, meaning the last address stored will be the first address returned, since functions are called in the same way. Due to the nature of functions, the RAS has 100% correctness, since every call has a matching return.

However, the RAS is limited by its size. If too many functions have been called before returning, the relevant addresses will have shifted out of the RAS, meaning there will not be a prediction once the return is called.

E. Taken Predictor

A taken predictor is the simplest branch predictor tested. When predicting a branch, it assumes that branch is taken. It can only do so if the branch has already been taken, otherwise the address will not be in the branch target buffer. In the case the branch is being seen for the first time, the branch is predicted not-taken.

VII. Miscellaneous Components

Alongside the modules used in the processor, some code was written to test functionality and performance. These components and their uses are listed in this section.

A. Correctness Hooks

The integration of correctness hooks was vital to ensuring the pipeline components functioned as expected. The primary goal of the hooks is to output information about each stage in the pipeline after each cycle. The stages focused in the testing hooks are the request and response from the instruction cache, the instructions being dispatched into the backend, and the state of the register file on a branch mispredict.

The hooks output this information to a file that could be viewed, even if the program in the processor did not complete. The output was vastly helpful in determining functionality, and ultimately helped get the processor to pass even the simplest test cases. Afterwards, the visual debugger was the primary source for debugging, but longer programs were still easier to analyze quickly due to the correctness hooks.

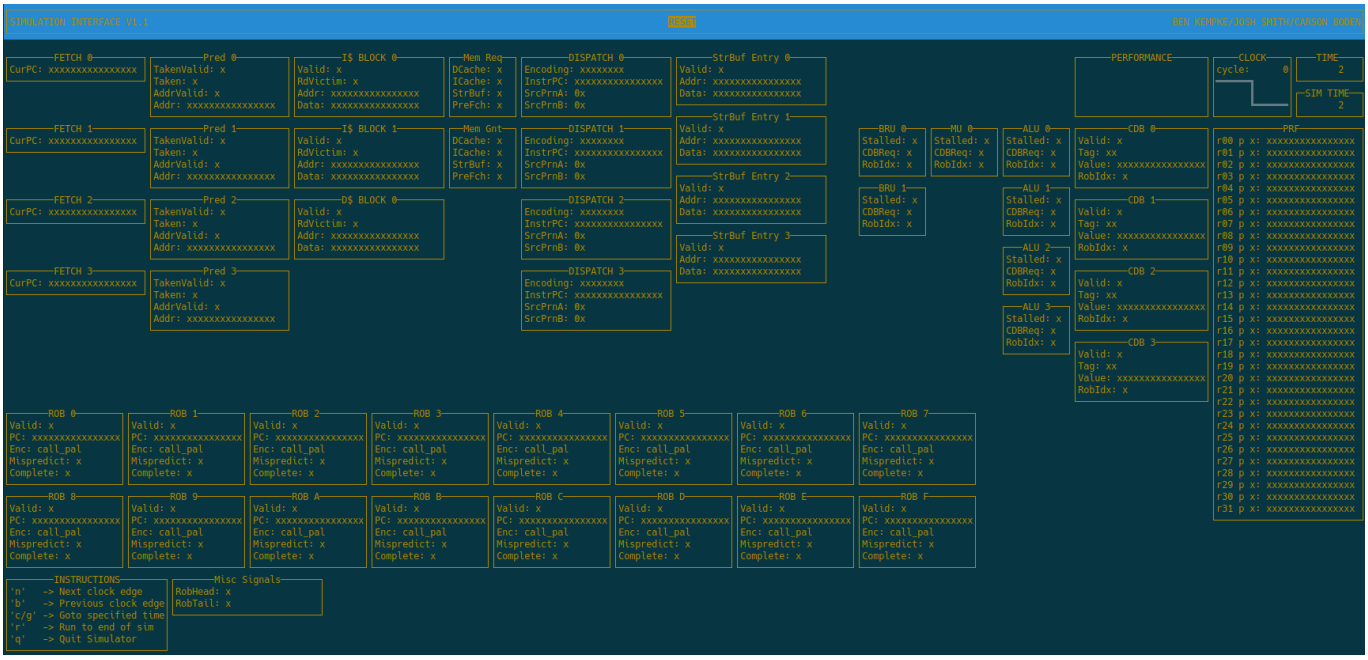

Fig. 4. The visual debugger makes it easy to determine functionality within the pipeline at a glance. The debugger has also been parameterized to aid testing different size modules.

B. Performance Hooks

To analyze performance, strategic hooks were added into our code. Three aspects of the processor provided feedback: branch prediction, cache performance, and structural hazard stalls. We observed how many branches were committing and how many were predicted correctly to see how branch predictors were performing. By observing how many branches we were mispredicting, it was easier to compare the performance of different branch predictors.

To determine memory hierarchy performance, hooks in the data and instruction cache measure the hit and miss rates. Structural hazards were measured by how often the processor stalled and contributing factors to those stalls. The main structural hazards were observed in the RS, LSU, ROB, and the PRF.

C. Scripts

A variety of scripts were designed and used by the team to maximize efficiency. Scripts were used to create test cases, run regression tests, assemble test files, and compare performance. While the scripts were not part of the project assignment, they were integral to quickly verify correctness or create a test when needed. Additionally, some scripts used for the Git repo ensured that team members would push modules with a corresponding passing testbench.

D. Visual Debugger

While the original correctness hooks were important when first wiring up the pipeline, the visual debugger is important for determining niche integration bugs. The debugger provides a clearer layout of nuanced information and changes as the pipeline executes a program.

The visual debugger, as seen in Fig. 4, is designed using C and ncurses, a programming library allowing for user interfaces in the terminal. Once the debugger has started, the user can press keys to navigate through the program as it is run on the processor. Our visual debugger shows information in the caches, ROB, memory arbitration, PRF, and more. Furthermore, the debugger highlights information as it changes to make it easy to determine the control flow of the program.

The debugger was developed with parameterization in mind as well. Each of the displays is fully configurable to show any number of functional units, ROB entries, or instructions. The result is a powerful, dynamic tool that can be used to find bugs much more efficiently.

VIII. Final Parameters and Setup

After the processor was fully functional, it was tested with a variety of parameters to optimize for performance. First, we started by measuring performance changes when altering large structure sizes, like the PRF, the ROB, and the RS.

When determining the ROB size, we measured the average time to execute a single instruction and compared it to other configurations. We ended up choosing a 16-entry ROB because it gave better overall performance, taking into account CPI and clock period changes. Then, the RS was set to sixteen entries and the PRF to sixty-four entries because neither of those contributed to the critical path but would guarantee that there would be no structural hazards which cause stalls, reducing performance.

Then, we started playing with more auxiliary structures, like the number of functional units. Four ALUs, two BRUs, and a single MU allows the processor to service a reasonable amount and variety of instructions. The ALUs were particularly important to make sure that we could actually execute

four instructions each cycle. An average number of branches would be one in seven instructions, so we knew that two BRUs would be able to handle more than average number of branches, while not contributing to the critical path. We played with the number of MUs, but making it higher did not yield higher performance.

Inside the LSU, we tried making the entire structure larger; it did not increase performance, so we left it at the original parameterization, with four entries for loads and four entries for stores. On average, about 25% of instructions are memory operations, so, with a ROB size of sixteen, we would on average be dealing with four memory operations.

The store buffer holds stores that are in the process of going to the memory system; our system could service stores relatively quickly, so we did not make this structure any larger.

To determine the optimal number of blocks to prefetch on a miss, we adjusted the number of prefetched blocks on a miss and determined that the configuration that created the best result was one block.

IX. Group Contributions

The group used an anonymous survey to determine everyone's opinion about group contributions. The weighted average of each person's contributions can be seen in Table I.

The group met almost every day in the Shapiro Undergraduate Library to work on the design and coordinate workloads. Planning and working in the vicinity of one another maximized efficiency and made it easy to run ideas by other members.

TABLE I
Group Contributions by Percentage

Team Member	Contribution
Alec Beljanski	19%
Carson Boden	23%
Alex Kisil	14%
Kyle May	25%
Alisha Patel	19%

X. Analysis

The main measurement of performance was the average cycles per instruction for given benchmarks. The average CPI over more performance-driven benchmarks, shown in Fig. 5, is 2.92. This value is higher than anticipated, and the remainder of the section will highlight how the processor design contributed to the performance.

Another measure of performance was our clock period of 9.1ns. Considering a four-wide superscalar design, the clock period is satisfyingly short.

A. Load-Store Unit Dispatch Performance

Internally, the Load-Store Unit is a separate store queue and load buffer. When dispatching instructions, the processor

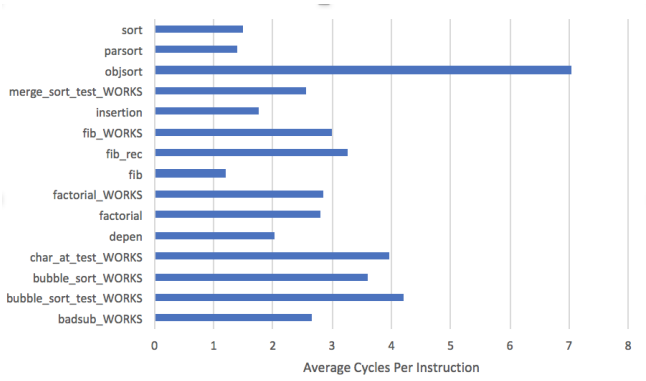

Fig. 5. The CPI of the processor is relatively high for most testbenches, considering the superscalar nature of the processor.

stalls on any loads following a store in that cycle's set of instructions. Since setting up the dependencies for the load buffer are complicated, the stalls allow stores to dispatch before them.

When undergoing performance tests, the processor stalled significantly due to LSU structural hazards. The first approach to solve this problem was to increase the load buffer and the store queue size to alleviate the pressure on the structures. This idea yielded minimal decrease to the overall number of stalls, and therefore the stalls were most likely due to the bottleneck that we introduced with the dispatching logic in the LSU.

Upon further analysis, relatively small amounts of stalls were due to the dispatching logic in the LSU, as seen in Fig. 6. The benchmark measured our standard pipeline configuration (16 entry ROB, 16 entry RS, 4 entry store queue, 4 entry load buffer) and determined how many stalls were due to the LSU. Altering the number of the load buffer and store queue entries to be larger than the ROB size guaranteed that the only types of stalls from the LSU were due to the dispatching logic. That ratio of the new number of stalls to the original number stalls would be the ratio of stalls forced by the dispatch logic.

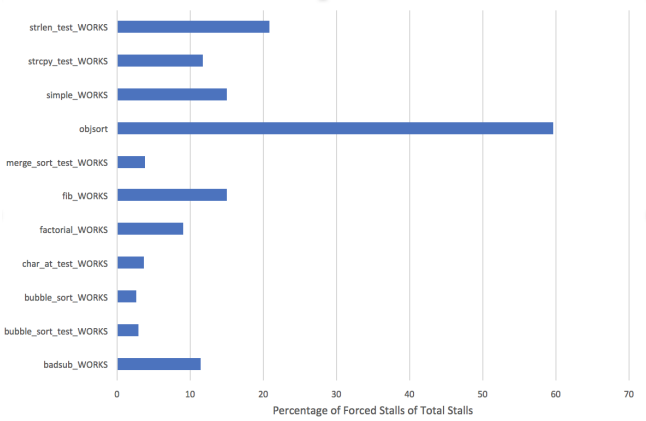

Fig. 6. Testbenches like *objsort* force a large percentage of stalls.

We were somewhat surprised by this result; it yields relatively good performance while simplifying the design significantly. However, the worst case does still occur, as seen in the *objsort* benchmark. The amount of stalls due to this logic

was a significant portion of the total stalls in general, around 60%.

B. Forward Wake-Up Performance

Wake-up, select, execution, and CDB broadcast are pipelined into three stages. In the first cycle, the RS selects any instructions that are awake and have a functional unit available to send to execution units. The second cycle (or multiple cycles for the multiplier) is completing the actual computation. Instructions in the third cycle request the CDB and broadcast the result. This change decreased the processor's ability to take advantage of instruction-level parallelism.

With the original scheme, independent instructions going to the same functional unit would cause a dead cycle. Thus, CPI suffered significantly. For example, our custom benchmark *gold* could only reach around 0.5 CPI, when 0.33 is the expected target. To deal with this problem, CDB requests became decoupled from CDB broadcasts. Now the CDB requests happen a cycle early, allowing the CDB arbiter to choose which execution units to schedule the next cycle and decreasing the CPI of *gold* to 0.33, which is an acceptable level.

Afterwards, dependent instructions would always have two dead cycles between them. The amount of dead cycles between depended instructions was reduced by allowing the RS entry to wake up the same cycle it received its information. This benefit was easily observed through the *depen* benchmark, which consists of a chain of dependent instructions in a loop, with dependencies between the loop iterations. Originally, the benchmark had a CPI of 2.9; after this optimization, it was reduced to 2.0.

Fig. 7. The different block sizes for prefetching produce various levels of performance.

C. Instruction Prefetching Performance

The prefetch module is part of the memory hierarchy we created to reduce stalls while giving certain priority to modules. Internally, the module contains a buffer for 4. This buffer is simply just an MSHR that was made for the instruction cache and data cache. In the processor, the data cache has the highest priority, then the store buffer, then the instruction cache, and finally the prefetcher. Since the prefetch is lowest priority when granting requests to the memory bus, we were initially unsure how much this module would improve performance. However, the performance improvement was clear between no prefetching and prefetching four blocks, which is what is in the final processor.

As shown in Fig. 7 prefetching was helpful in improving the CPI of the processor. However, the number of blocks prefetched drastically changed the CPI of the test cases. For example, prefetching four blocks instead of prefetching one block proved to be detrimental to performance in branching tests like *btest1*. In *btest1*, prefetching four blocks resulted in a CPI of 6.74 while the CPI when prefetching one block was only 2.99. Poorer performance was expected from prefetching more blocks as this results in fetching instructions that will not be needed while simultaneously polluting the instruction cache. We did not anticipate just how big of a difference the prefetching size would make. However, prefetching four blocks improved the performance the most in *catch_bubbles*, a program composed of mostly loads. In this program the CPI for prefetching four blocks is 2.68 while the CPI for prefetching one block is 2.75. Since various test cases performed best at different prefetch block sizes, we decided to prefetch four blocks for the final processor. After more analysis, we see that perhaps prefetching one block would have resulted in better performance.

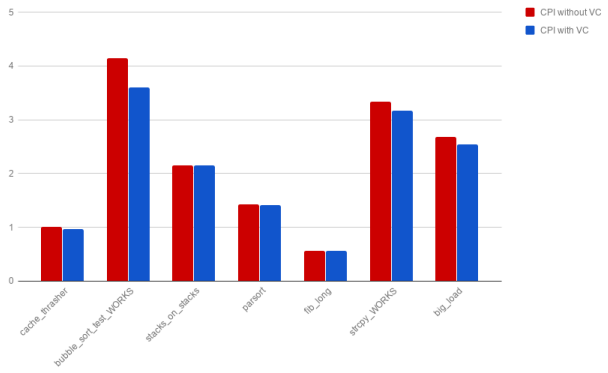

Fig. 8. The victim cache always provides equal or improved performance.

D. Victim Cache Performance

While the victim cache was ultimately excluded from the final design of the processor, we performed an analysis on the performance of the processor with and without the victim cache to quantize its benefit.

The victim cache module was implemented as a fully-associative, 4-line cache with 8-byte blocks. As it is meant to alleviate the potential problem of more than one commonly referenced block mapping to the same line in the data or instruction cache, the largest impact was expected to be visible on test cases that were very memory intensive and had many cache misses.

As Fig. 8 shows, *bubble_sort_test_WORKS* had the largest improvement out of the analyzed test cases, dropping about .54

cycles per instruction, with *strcpy_WORKS* and *big_load* having moderate improvements, *cache_thrasher* having a small improvement, and *stacks_on_stacks*, *parsort*, and *fib_long* having negligible or no improvements on CPI. All test cases were run at a clock period of 10ns.

While all of these test cases make requests to memory, it is natural that we should only see improvement on some of them due to varying cache miss rates and what types of cache misses the test cases cause. For example, *stacks_on_stacks*'s lack of improvement in CPI is expected, since the data cache has a 100% hit rate and the instruction cache has a 98% hit rate. The high hit rate prevents the victim cache from eliminating misses.

Based on the performance boosts seen in the victim cache, the largest improvement expected will be for *parsort* as the data cache has a hit rate as low as 46%; yet, there is no substantial improvement. This issue could could simply be due to a large number of compulsory misses, since the victim cache will not populate until there have been loads in the program. However, *parsort* references the same data many times, which should dilute the amount of compulsory misses there are. The more likely reason for the lack of improvement is due to the small size of the victim cache. With only four lines, it is not possible for the victim cache to resolve misses on data whose references were relatively spaced out, even if it was referenced many times in the given program. It is possible that the nature of *parsort* sorting two halves of an array in *parallel* causes data in one half to be evicted from the victim cache before the program switches back from sorting the other half.

Overall, the victim cache shows a substantial CPI decrease in programs that are heavy in memory accesses and frequent misses on a few given blocks. Compulsory misses and spaced out cache misses for a given block can counteract the utility of the victim cache due to its initial emptiness and comparatively small size.

E. Branch Prediction Performance

During the development of the processor, a variety of branch predictors were tested. We started with a very simple always taken predictor that was not very successful. Moving on to more sophisticated predictors, the table predictor and the bimodal predictor performed similarly on the given benchmarks, with one not being clearly better than the other, and since the branch predictors were not on the critical path, we decided to use the table predictor in our final implementation. Fig. 9 shows the different predictors and their mispredict rates.

The return address stack was another consideration we hoped to implement into our pipeline. While it did not end up in the final design because it caused synthesis errors, it would have given the processor significant improvements on certain test benches, specifically *fib_rec* and *objsort*. Figure 10 shows the difference in misprediction rates with and without the RAS using the table predictor. Because improvements in misprediction rates were only for specific testbenches, it was not included in the final processor due to the synthesis issues.

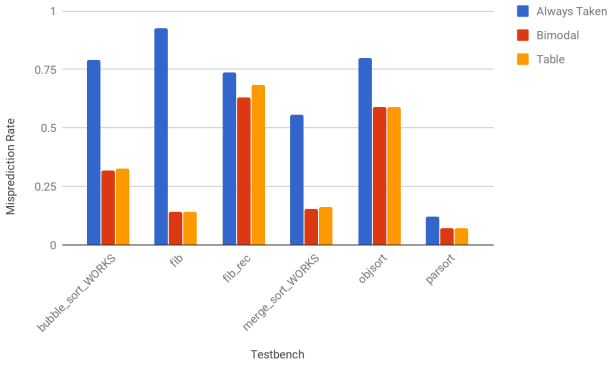

Fig. 9. The always taken predictor performs surprisingly well compared to the bimodal and table predictor.

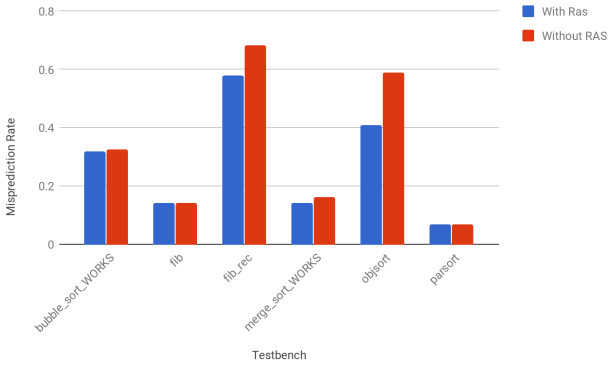

Fig. 10. The RAS provides marginally better performance for testbenches that use multiple function calls.