Marina: A Processor

Todd Charlton, Branden Ghena, Austin Maliszewski, Ben Mason, Paul Murley

December 11, 2013

Introduction

The performance of any microprocessor system pipeline depends on its ability to remain as close to fully utilized as possible. However, many operations that microprocessors perform have variable and unpredictable execution times, preventing fixed-stage pipelined systems from performing well.

In this paper, we present our upgrades to the VeriSimple Alpha, an implementation of a subset of the Alpha ISA written in synthesizable Verilog. Our upgrades include adding out-of-order execution capacity, simultaneous multithreading (SMT), and two-way superscalar execution, all of which are intended to let the processor do more at once and to minimize the time spent sitting idle.

1 System Design

We present Marina, a two-way superscalar simultaneous multithreading out-of-order microprocessor, written in synthesizable SystemVerilog. Marinaconsists of 20 discrete submodules and a top-level module that connects them together. In this section, we provide an overview of these modules.

Our processor provides a full implementation of the EECS 470 Alpha ISA subset, as required for this class. Additionally, it provides an implementation of several new instructions, to be detailed later.

1.1 Processor Stages

Our processor has a pipeline that somewhat resembles the standard RISC machine, however, instructions do not need to flow through all parts of the pipeline in order. Each stage is implemented by one or more System Verilog modules, and ends in flip-flop registers internal to the module implementing the stage. This allows us to have significantly simpler top-level logic.

1.1.1 Fetch Stage

Marina's fetch stage queries the branch predictor, branch target buffer, and instruction cache to select up to two instructions to send to the decode stage for decoding and register renaming. The fetch stage prefers to send one instruction from each thread if two threads are running and the current instruction from each thread is available in the instruction cache. If instructions are only available from one thread, the fetch stage will send both the current instruction and the next instruction in that thread, if available. This allows us to sustain superscalar execution abilities in single-threaded mode.

The fetch stage also checks the reservation stations and the reorder buffer to ensure that there will be adequate space for new instructions. This is done so as to make fetch the only source of pipeline stalling, at the cost of not being able to always dispatch instructions when the RS or ROB are almost full. We decided to accept this slight performance hit so as to greatly simplify the logic of stalling, which was significantly complicated by having multiple threads.

1.1.2 Decode Stage

Marina's decode stage decodes two instructions in parallel, and completes register renaming for the appropriate thread. If the registers are currently available in the physical register file, their values will be read out then. The decode stage writes the decoded values into pipeline registers for dispatch into the reservation stations and entry into the reorder buffer (ROB) in the next cycle.

1.1.3 Dispatch Stage

On the cycle after decode, instructions are dispatched into the reservation stations and are given an entry in the reorder buffer. Again, we are capable of dispatching two instructions simultaneously, regardless of thread. The dispatch stage also checks the common data bus (CDB) to see if any of the dispatching instruction's operands have been made available. In fact, Marina gets its name from this process.

1.1.4 Issue and Execution Stage

Marina has a combined issue and execution stage. Instructions that have acquired values for their operands are issued to execution units, which can then finish and commit back their values in the same

cycle. This is useful, since most of our execution units can finish in one cycle. However, multiplication is a multi-cycle operation and does not reap benefits here.

When functional units complete, they request permission to write to the CDB. The CDB arbitrator manages the CDB and stalls functional units as appropriate. The arbitration process gives priority to the commit unit (which returns the values from memory operations and described in more detail below), then to the multipliers, then to the address generation units, then to the arithmetic units.

1.1.5 Commit Stage

The commit stage takes instructions from the ROB and retires them. It is capable of retiring two instructions simultaneously, and prefers one from each thread, but is also capable of retiring two instructions from a single thread. During the commit stage, branch prediction correctness is reported back to the branch predictors (described below) and branch misprediction squashes are triggered throughout the system (when necessary).

The commit stage also handles processing for several instructions and interacts with both the data cache and the CDB. We decided early on that commit would be the only module to interact with the data memory. This decision was made because we recognized that we only had a limited amount of complexity to spend and we would be spending plenty on superscalar execution and SMT. We decided to make as many other simplifying choices as possible.

As we mentioned previously, one of these choices was to handle memory interactions once they have reached the head of the ROB. The data is sent to the data cache and that slot of the commit stage will wait until it has been serviced. Since we are not allowed to retire instructions out of order, while servicing a load or store no other instruction may be retired from the same thread. Results for a load are written back on the CDB, for which commit has the highest priority.

Finally, the commit stage also handles irregular instructions that could not be handled elsewhere in the system. This includes: `fork`, `cpuid`, `ldq_l`, `stq_c`, `halt`, and `illegal` instructions. Illegal instructions retiring in the commit stage cause system failure. The implementation of fork, cpuid, ldq_l, and stq_c are described below in section 1.4.1.

1.2 Advanced Caches

Our processor uses two caches: an instruction cache and a data cache. Each consists of 32 64-bit lines and each is four way set associative. Both caches are non blocking, meaning that memory requests can be processed in parallel. We implement this in the following way. When we send a request to memory for data, we add the address to a buffer that maps memory tags to addresses. We use this to determine whether or not a given request needs to be sent off to memory. We also use this so that we can send loads off to memory and, assuming we get the right acknowledgment from memory, we can in many ways forget about them.

The instruction cache is quad ported, meaning that the core can read four instructions from it at a time. In general, this is to allow the fetch stage to read the current and next instructions from each thread. The instruction cache uses a pseudo-LRU eviction policy. Each set maintains and updates an eviction tree that points to the next line to evict in that set. Since the only time we are writing to the instruction cache is when data comes back from memory, and only one line can come back from memory at a time, we do not have to worry about evicting more than one cache line per cycle.

The data cache is only dual ported but offers many of the same capabilities of the instruction cache. It is non-blocking as well. Stores are fired off to memory and then forgotten. Load information is stored in a buffer to allow parallel load processing and to ensure the correct data makes its way to the correct requester. One major difference between the instruction cache and the data cache is that the data cache uses a pseudo random eviction policy within sets by use of counters. This is because the data cache may have up to three lines evicted in the same cycle. If a load comes back from memory and we get stores on each of the two lines from the core, we must in some cases evict three things, in rare cases all from the same set. Maintaining pseudo LRU trees to account for this possibility would create a huge amount of overhead, so we opted for pseudo random eviction.

1.3 Branch Prediction

The branch predictor used in our system was a local history predictor with a two-bit history and a standard two-bit saturating counter as the predictor. The branch history table is accessed by the lower five bits of the PC (after word access bits are removed). It stores the most recent two cycles of history for the associated branch or jump instruction. This history data is used to access the second-level prediction table, consisting of mappings from history to a two-bit predictor. These choices were determined through testing as shown in section 3.

In order to support a prediction of Taken, a Branch Target Buffer was also created. It stores the target address for branches and jumps as determined by the commit stage. It is a fully-associative structure with a random eviction policy. Each entry is tagged with the instruction's PC, allowing us to ensure that it will never alias with a non-branch instruction except in the case of self-modifying code. The BTB is 32 entries in size, allowing it to comfortably hold branch target data without evictions on most test programs.

1.4 Simultaneous Multithreading

As mentioned before, processors achieve their best performance when they are being completely utilized. However, because of the variable latency of some instructions, particularly the retrieval of a word of data from memory, in an in order machine, the processor is forced to insert many cycles of no-operation instructions, while waiting on the memory system. Out-of-order techniques help solve this problem because they allow the processor to replace those no-operation instructions with instructions that do useful work, that come later in program order, but are not dependent on the load causing the backup.

Out-of-order execution alone provides a significant advantage over in-order execution, however, when combined with simultaneous multithreading, the advantage is greater still, because the processor can now consider a second program when looking for ready instructions. This means there will be a much greater number of ready instructions because it is impossible for these programs to have register level data dependencies on each other. This section describes the modifications we've made to the standard out-of-order machine to support SMT.

1.4.1 New Instructions

In addition to the instructions defined by the EECS 470 teaching staff, we also implemented additional instructions to support our simultaneous multithreading features. These additional instructions are a load-lock instruction, a store-conditional instruction, and two `call_pal` support instructions (`fork` and `cpuid`).

Load-lock & Store Conditional: These two instructions serve a very important purpose in a multiprocessor or SMT environment, because they provide a framework for data synchronization. When data is loaded into a register with a load-lock instruction, the processor listens for writes to that same memory location from all threads, and causes a later store-conditional to fail if any thread completed a write in between the load-locking and store-conditional. This allows the processor to guarantee atomicity of the load-update-store procedure, a critical feature for writing an operating system or multithreaded programs.

These instructions primarily operate in the commit stage of our processor. Our commit module handles all of the interactions with data memory, so it seemed to be a logical choice for the hardware associated with these instructions. They flow through the rest of our out-of-order pipeline the same way regular load and store instructions do, but go through commit completely differently.

The Alpha architectural specification has clearly defined these instructions, so there was not much room for creative design choices. We implement the specification completely. As required, when a `ldq_l` (load lock) instruction completes, our core sets the `lock_flag` register for the requesting thread, and stores the requested memory address in the corresponding `lock_address` register. Like normal `ldq` instructions, it also returns the data from memory into the physical register that was associated with the architected register of the instruction at dispatch time.

Then, when a `stq_c` (store conditional) instruction is executed, the `lock_address` register for the requesting thread is checked for a match. If the addresses match, the `lock_flag` itself is checked. If the lock flag is set, the store is allowed to commit. If the `lock_flag` is not set, or if the `lock_address` does not match the target address of the store, the store is not allowed allowed to commit. The `lock_flag` is returned to the physical register that was associated with the architected register of the instruction at dispatch time. This allows the programmer to determine if the `ldq_l`/`stq_c` pair was successful. Also, the `lock_flag` for the thread is cleared. The `lock_flag` for the other thread is also cleared if the `lock_address` is the same as the target address of this instruction.

Finally, another component was added to the handling of regular `stq` (store quadword) instructions. When a store instruction is committing to memory, it checks the `lock_address` register for both threads. If there is a match, the corresponding `lock_flag` is cleared.

Together, this allows the processor to guarantee atomicity of the load-update-store procedure, a critical feature for writing an operating system or multithreaded programs.

```
/* FORK and save result in r30 */
call_pal  (0x100 + 30)
/* Get CPUID and save it in r29 */
call_pal  (0x200 + 29)
/* Go to thread b if it started */
bne $r29, thread_b_label
/* Go to logic if fork failed */
bne $r30, failed_to_fork_label
/* Go to thread a */
br        thread_a_label
```

Figure 1: Sample code for spawning a new thread.

`call_pal` **support instructions:** The Alpha instruction set defines a special instruction called Call Privileged Architecture Library or `call_pal` for short. This instruction takes as its only argument, the immediate number of the PAL entry to run. In the original VeriSimple processor, there is only one supported PAL entry: `halt`. We added two new families of PAL entries, fork and cpuid. We say "families" because we slightly abuse the `call_pal` instruction and the constant-folding feature of the assembler to handle these commands. This is perhaps best shown by an example, which we'll present after explaining what the instructions do.

Fork is used to spawn a new thread. In our implementation it behaves somewhat similarly to the Unix `fork()` system call, in that the new thread begins executing at the address of the instruction immediately after the fork instruction. Fork returns a 0, 1 or -1 to the parent thread, indicating that it was thread 0, thread 1, or that fork failed, respectively. This allows a program to try forking, and to detect if forking failed. We also provide a `cpuid` instruction which allows a programmer to query the processor to find out the ID of the thread currently executing. This allows programmers to `fork`, execute `cpuid` and branch based on these results. Spawning a new thread looks something like like Figure 1.

We also modified the behavior of the halt `call_pal` instruction to halt only the currently running thread.

2 Tools & Testing

This section of our report serves to discuss the tools we created to aid in testing of Marina, and to describe our testing procedures.

2.1 Tools

In order to help facilitate our development of Marina, we developed a number of tools to make programming and testing easier. Some of these tools were incredibly useful, and we would highly recommend some of these techniques to other groups; others we see as "candy" that provided no practical benefit.

2.1.1 Autogeneration of `top.v`

One of the tools we found the most useful in developing the project was verilog-mode for Emacs. Verilog-mode has a number of features designed to reduce the redundancy of verilog and to make coding easier. We used verilog-mode's AUTOs extensively, to complete virtually all of our top-level module automagically. For the most part, the AUTOs worked fantastically, and dramatically reduced our workload while also removing all concerns about `top.v`'s correctness.

Upon telling Emacs to update the AUTOs, verilog-mode searches for all instances of the comment `/*AUTOINST*/` in module instantiations and automatically fills out all the connections. It then searches for the `/*AUTOWIRE*/` comment and creates `wire`s for all of the automagically connected wire names.

Because we anticipated using verilog-mode to connect our modules together (and because it's good style), we made sure to name the inputs of each module with the same name as the output of a module. This allowed the automagic connections to work perfectly, with minimal effort. The only exception to this was the wiring for our functional units. Since we wanted to be able to manipulate the numbers and types of our functional units by changing 'defines, we could not use automagic connections for them, since verilog-mode is not SystemVerilog compatible and does not yet understand generate commands.

We created an additional file `top.v-template`, which was used to maintain a clean version of the top.v module without any of the autogenerated content. In this file, we created dummy instances of all of our modules with the `/*AUTOINST*/` comment and inserted an `/*AUTOWIRE*/` comment before them. We also added the buses needed for our functional units and instantiated them in a generate block.

We also created an Emacs-Lisp script to help complete the AUTOs. The reason for this was two fold. The auto completion in verilog-mode doesn't quite support SystemVerilog. Since SystemVerilog is mostly a superset of Verilog, it mostly works, but it didn't work correctly with enums, so additional processing was necessary to clean up after the AUTOs were completed. It seemed tedious to have to do this every time we wanted to rebuild the top-level module, so we decided to automate it. The secondary reason is that Austin was the only Emacs user on our team,

so we needed a way for everyone else to rebuild top.

We want to quantitatively demonstrate how amazingly helpful this was. All together, our final version of `top.v-template` was only 209 lines. Our final version of `top.v`, after running the AUTOs and cleanup script, was 1032 lines. Emacs automagically wrote 823 lines of code for us; 823 lines that were guaranteed to be correct.

2.1.2 Testing Scripts

We quickly saw the importance of having an easy way to verify our processor's functionality on both the published test cases and on our group's test cases. We also wanted a way to quickly identify regressions in functionality. Therefore, we wrote `bash` scripts that compile our processor, run it against all the test cases, comparing the output with the expected output. If the test case passes, the word `PASSED` is printed in green; if the test case fails, the word `FAILED` is printed in red, along with a helpful message telling us where the test case failed, making it very obvious to see when a change has caused a regression. Also, the script will print `WARNING` in yellow, if a test case has passed with assertion failures.

2.1.3 ncurses Visual Debugger

We saw some value in having a way to visually watch instructions flow through the pipeline, so we decided to extend the ncurses-based debugger that was provided with Project 3. Our debugger was capable of showing all instructions currently in flight in the ROBs of both threads, as well as all instructions currently waiting in the unified reservation stations. The debugger also showed us the processor's interactions with the memory system, which proved very useful in debugging the caches.

2.2 Testing Methodology

The following two sections serve to document the way we tested individual modules and the system as a whole.

2.2.1 Module-level testing

After meticulously writing each module for each stage of our processor, it turns out that every module had at least one mistake. Welcome to the world of computer science. Writing flawless code is almost unheard of for programs longer than 50 lines. Understanding this pleasant fact helped us refine each module. For every module that we wrote, there was an accompanying testbench (excluding the program counter).

The testbenches were written to emulate actual inputs the module would receive from other modules when integrated into the final processor. The reason behind writing so many testbenches was simple. If we can guarantee that each module provides correct output, then come time for integration, the processor should theoretically work. Also, in unit testing if an error occurs, it is easier to find. Since the testbench is only written for one module and therefore the bug lies within that module. In integration testing, if a bug occurs, it becomes hard to track down which module was the source of the bug.

This process proved its worth many times over. The test benches were able to catch many subtle bugs that we did not consider when writing the modules. Additionally, the test benches aided down the road when changes were made to the original modules in an attempt to increase performance. Whenever one of these changes was made, the module was run through the testbench again to ensure that the fix did not cause a correctness regression.

We think the extensive module level testing contributed significantly to our relatively painless integration process. Each module was tested by a different team member than the one who wrote it. We feel that this strategy worked well.

2.2.2 Test programs

The next step after unit testing was integration testing. This stage involved connecting all of the modules together and running actual programs through our processor. Initially, this process focused on correctness. We used the single cycle processor to generate the solutions we used to test against the output of our processor.

The second part of our integration testing was to evaluate the processors performance on specific test cases. The performance evaluation was two-fold. The first attempt was to decrease the CPI. The next attempt was to bring down the clock period. In this stage, special testbench names were created that were designed to test the performance of the processor on specific scenarios. Examples of these test programs include a program with independent instruction chains, a program with many memory accesses in a row, and a program with multiple conditional branch instructions. At this point, we incorporated the cache and branch prediction percentages into our program output. These reports allowed us to test different implementations in the cache and predictor and watch how they affected their own accuracy, and ultimately CPI, on those test programs.

In addition to the supplied test programs and the various specific testbenches written by our team, we also created some test cases strictly for SMT. The most basic test was to have both threads write to the same memory 50 times, with the end goal being the shared memory to finalize at a value of 100. The next step was to add nop instructions to one thread and continue to have both count to 50 again. Interestingly enough, this test case exposed a last minute bug. However, this bug was resulting from a superscalar register freeing issue in the RRAT. Lastly, we wrote some test programs which are combinations of two larger test cases like insertion and copy or insertion and fib_rec. In these programs, the final contents of memory were hard to validate, but the separate thread's writeback.out's should have been their respective test cases solution. These SMT programs also showcased our processor's ability to achieve low CPI's on simultaneous programs.

3 Analysis

This section serves to detail our analysis process and to explain our choices, and what we've learned about our processor.

3.1 Branch Predictor

We used a seeded local history predictor with a two-bit history and a two-bit predictor. In development, we had originally created a three-bit local history predictor using the normal two-bit saturating counter prediction scheme. In testing, we tried various other options to determine if they had any improvement on the prediction accuracy and thus the CPI of our processor. In all cases, using branch prediction is a significant on-average improvement over predict not-taken.

Among our tested improvements to the branch predictor are a standard two-bit history predictor, a two-bit history predictor that begins seeded with values, and a two-bit history predictor using a different two-bit saturating counter. The seeded branch predictor was initialized as Strongly Taken for branches taken within the last cycle, and Strongly Not Taken otherwise. The state machine for the different saturating counter used can be seen in Figure 1.

Results of the testing can be seen in Figure 3. Testing was performed using the `Copy, Fib_rec, Insertion, Objsort,` and `Parsort` assembly tests. Due to the small size of the programs run on our system, the three-bit history predictor took too long to warm up and was ineffective. This same reason led to the effectiveness of seeding the branch predictors.

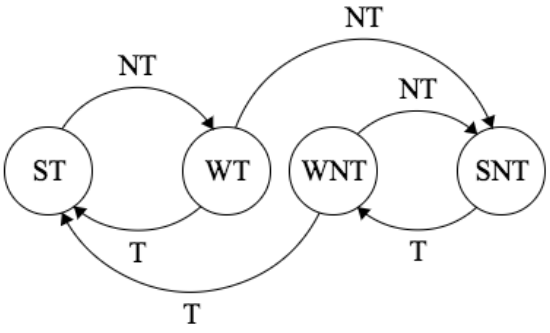

Figure 2: Alternate FSM for a two-bit predictor.

With longer tests, the improvements from a seeded predictor would be amortized to very small gains and three-level history predictors would be more effective. The attempt to utilize a different two-bit predictor state machine was experimental, but we had hoped it would lead to better results due to spending less time in the inner "weak states". In testing, however, the different counter design proved to be ineffective.

3.2 Out-Of-Order Parameters

There are a number of fundamental parameters for any out of order system. When optimizing our processor, we varied a number of these parameters.

Reorder Buffer Size: The size of the reorder buffer is essentially the maximal size of the instruction window that can be considered for out-of-order execution. Initially, we began with a 64 entry ROB for each thread. In our testing, we varied the size of the ROB and measured its effect on both achievable clock period and cycles per instruction on a subset of the published test cases. Unfortunately, we had a limited ability to vary these parameters, because our implementation depended on the ROB size being a power of two.

When we finally began working with synthesis, we realized that ROB size was very highly correlated with synthesis time, so we spent most of our time synthesizing with a ROB of 16 so as to avoid having to wait forever for synthesis.

As you can see in Figure 4, varying the size of the ROB did not significantly, or predictably affect performance. We're not entirely sure why this happened, since we expected CPI to be improved by increasing the ROB size, since increasing the ROB size allows the processor to consider more instructions for speculative execution. We had a few ideas as to why the

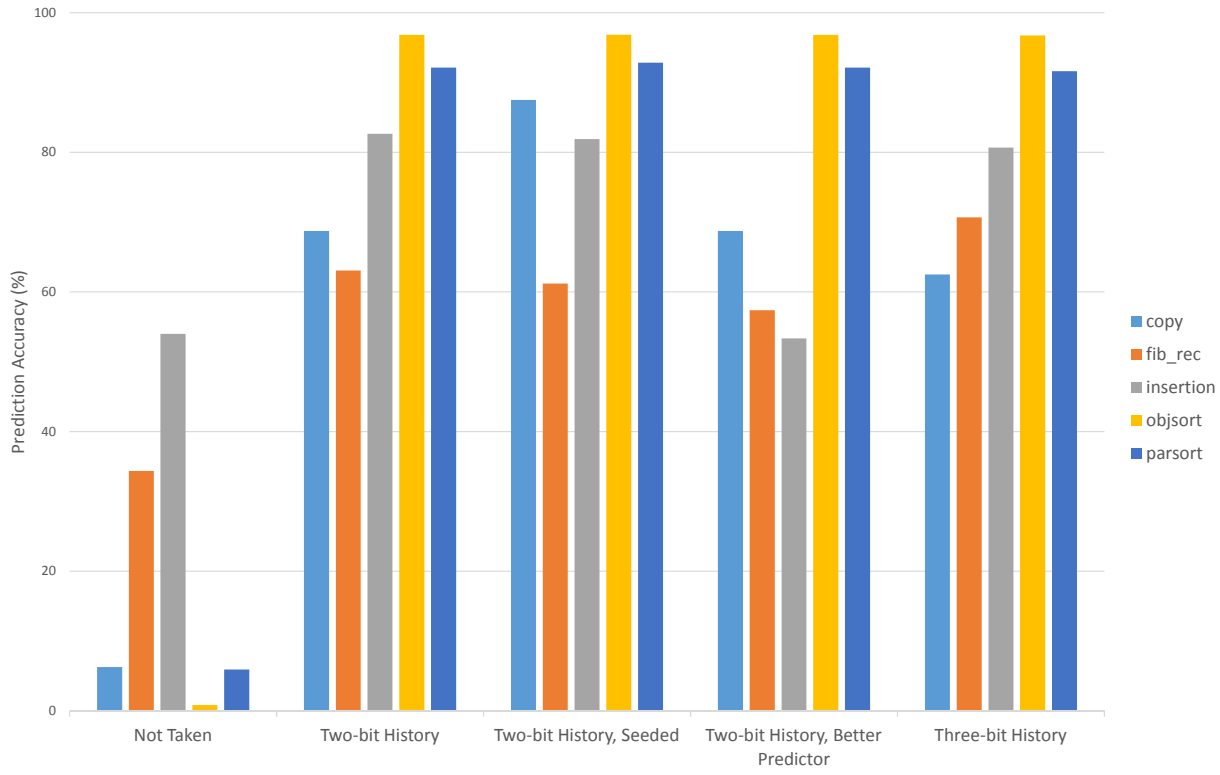

Figure 3: Branch predictor accuracy.

improvement was so minimal. First, we believe that we may have been somewhat limited by branch predictor accuracy, so as to never be able to take advantage of the additional ROB. Second, we believe that the increased space may have allowed more unneeded, and later squashed, instructions to be fetched, causing unnecessary instruction cache contention.

Therefore, other metrics motivated our decision for this parameter. Increasing the ROB size negatively affected our synthesis time, and the achievable clock period, therefore, we tried to keep it as small as possible while still maintaining acceptable performance. 16 entries per thread seemed to be the best spot.

Reservation Stations Size: The size of the reservation stations limits the number of instructions that can be presently waiting on arguments. By definition, the RS should be no larger than the ROB, since instructions must wait in both the ROB and the RS when awaiting arguments. Likewise, it makes sense to have the ROB be larger than the RS because most programs do not have a perfect dependency chain: that is, there will be some instructions in the middle that resolve their arguments and complete execution before the instructions before them. These instructions require ROB entries until they are allowed to commit, but may give up their reservation station. Finally, and perhaps most importantly, the reservation stations are generally implemented as a content addressable memory – increasing the size of a CAM makes it much slower.

As you can see in Figure 5, varying the size of the RS did not significantly affect CPI. However, it significantly affects clock period. We attempted to synthesize the processor with a doubled RS, which added 1.39ns to our clock period. This certainly seemed like a bad tradeoff.

Again, we're not sure why making the RS bigger didn't help much. Theoretically it should have had a large impact, since more instructions could be pending at any given time. We think that our scheduling logic was probably pretty terrible, and that this contributed a lot to the lackluster performance. Basically, we believe that instructions were getting picked out of the RS in a very suboptimal order, causing large dependency chains to build up, until eventually the system backed up and had to issue those instructions. That happens often with a smaller RS, so the effects were less pronounced.

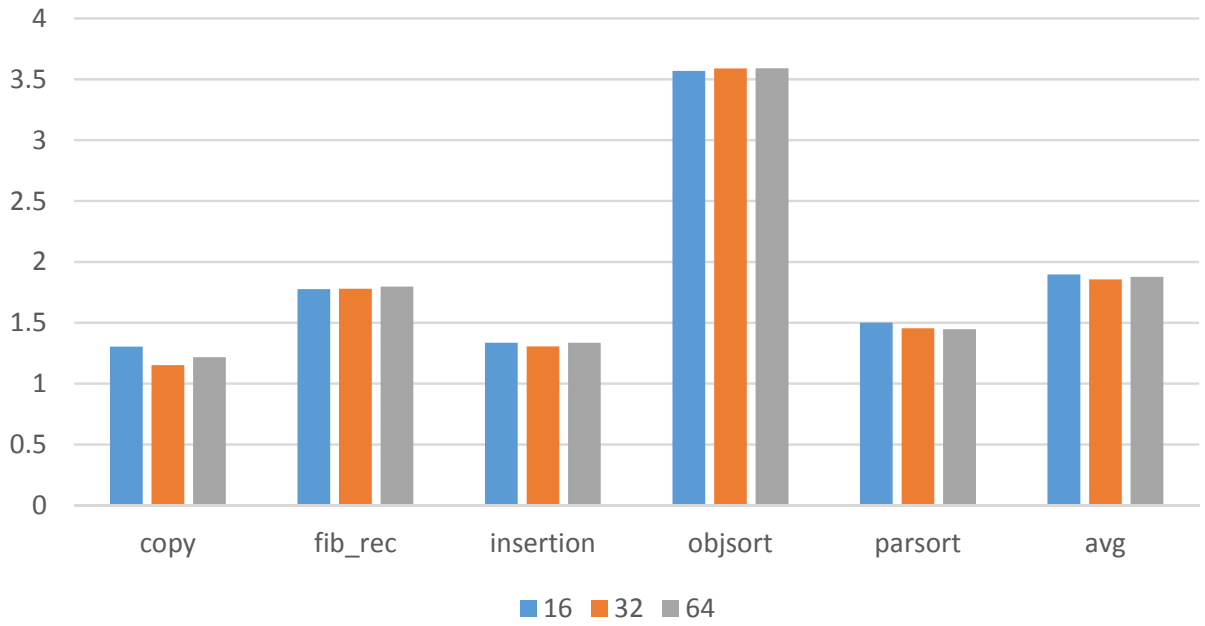

Figure 4: ROB size vs CPI for selected tests.

Number of Functional Units: We also parametrized the number of each type of functional unit to allow us to determine which selection gave the best performance. Initially, we started with four ALU's, two multipliers, and two memory/branch units. We tried to drop down to one multiplier because we figured two may be an unnecessary amount of hardware. However, one multiplier hurt our CPI on several test benches and we reasoned that having an extra multiplier has marginal impacts on the critical path. Multipliers themeselves impact our critical path, but adding an extra does not.

A change that we did accept was utilizing three memory/branch units instead of two. We discovered that there were more instructions that were serviced by this functional unit than we originally thought. Likewise, there were not as many instructions being issued to the ALU to warrant supplying four ALU's. When we went through with the change, our tests reported a non-trivial increase in CPI. This is not to be mistaken with a dramatic increase, but CPI jumped up by 0.3 or 0.4 on average, which we thought was worth keeping. In addition, this kept our total number of functional units to the same number we previously had.

Cycle Endpoints: The final parameter that we looked at was determining where cycles should begin and end. Certain decisions were obvious. It felt right that fetch and decode should be their own cycles. Likewise, we intially wanted issue and finish to occur in different cycles, and we implemented this at some point in our development, however, it ended up significantly worsening the CPI, and fixing that seemed hard. It did improve the clock period, but not by enough to make up for the CPI hit.

3.3 Clock Period vs CPI

Our final clock period was 12.51 nanoseconds. We brought the clock period down to under 12 nanoseconds, but could not get it low enough to have better overall performance than 12.51, due to memory latency. Table 1 shows CPI for five different benchmarks for different clock period ranges.

A graph of our processor's performance relative to clock period can be seen in Figure 6. Our goal was to be as low vertically as possible, so we stayed at 12.51 ns even though we could have shrunk the clock period a little more.

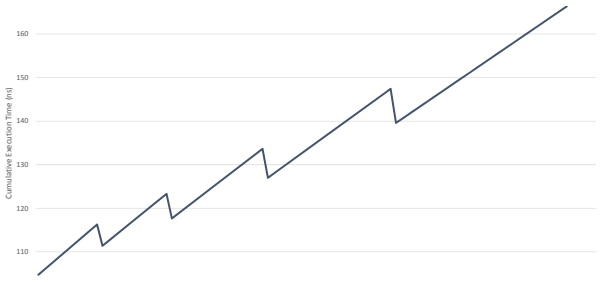

Figure 6: Clock period effect on performance

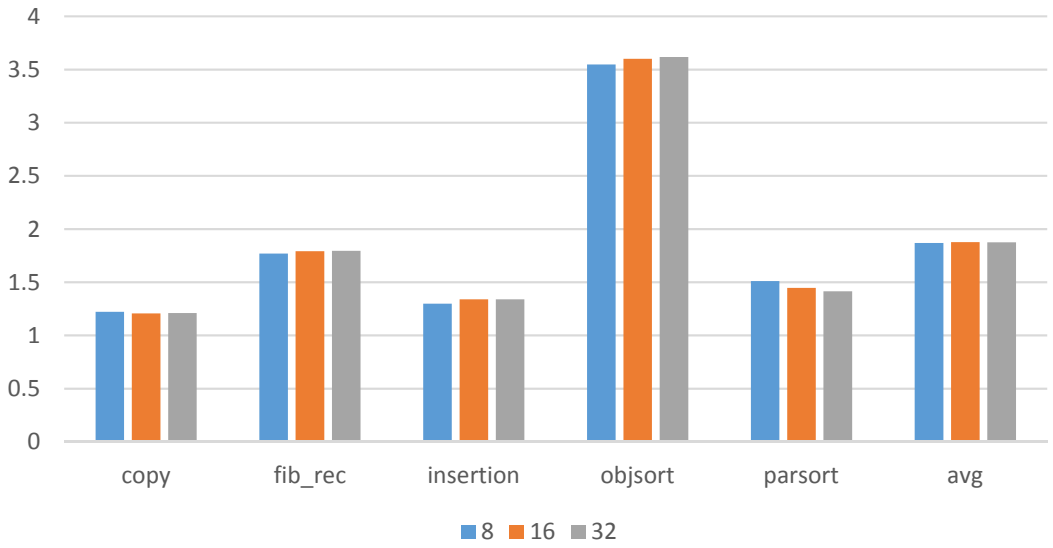

Figure 5: RS size vs CPI for selected tests.

	20-24.9 ns	16.7-19.9 ns	14.3-16.6 ns	12.5-14.2 ns	11.2-12.4 ns	10-11.1 ns
copy	1.215 ns	1.238 ns	1.262 ns	1.285 ns	1.308 ns	1.331 ns
objsort	2.624 ns	2.934 ns	3.245 ns	3.556 ns	3.868 ns	4.178 ns
fib_rec	1.628 ns	1.675 ns	1.722 ns	1.770 ns	1.817 ns	1.864 ns
insertion	1.169 ns	1.229 ns	1.264 ns	1.309 ns	1.355 ns	1.400 ns
parsort	1.178 ns	1.283 ns	1.388 ns	1.493 ns	1.597 sn	1.702 ns

Table 1: Cycles per instruction on different benchmarks

3.4 Simultaneous Multithreading

As part of the analysis of our processor, we analyzed the abilities of the multithreaded nature of Marina. We ran tests on their own in two-way superscalar mode to determine how long they take, what prediction rates they achieve, and what number of instruction cache and data cache evictions they suffer from. Next, multiple tests were joined into a single SMT test. This incurred a slight overhead in the form of the `fork` and `cpuid` followed by a branch to the appropriate code for the given thread. Measurements of the code were again taken to determine both the speedup of the code gained by performing the two tests in parallel rather than back-to-back as well as possible problems caused by running the two simultaneously. The speedup results of testing can be seen in Figure 7

The tested combinations showed a large amount of variation in the speedup gained. Conceptually, a desirable case would be one program which involvs many loads and stores which cause delays, and another program involving many quickly executing instructions such as ALU operations which can be used to fill those delays. This example is shown in the smtFibCopy test, which combines an ALU-intensive program (fib.s) with a memory-intensive program (copy.s).

In other cases, much lesser gains can be seen, or even losses in execution time. These can be due to several different reasons. Since branch predictors are shared between the two threads, it is possible that they will throw off the history results for each other. It is also possible that one thread will fill up the shared reservation stations, reducing the gains which could have been achieved with the second thread. Another reason for lesser gains would be a large difference in program lengths. If one thread finishes significantly before the other, the advantages of running them simultaneously will be lost. This can be observed a little in the smtInsertionCopy test, which combined copy.s, at 167 instructions, with insertion.s, at 785 instructions.

In both the modestly-gaining smtInsertionMult test and performance-decreased smtInsertionParallelLong

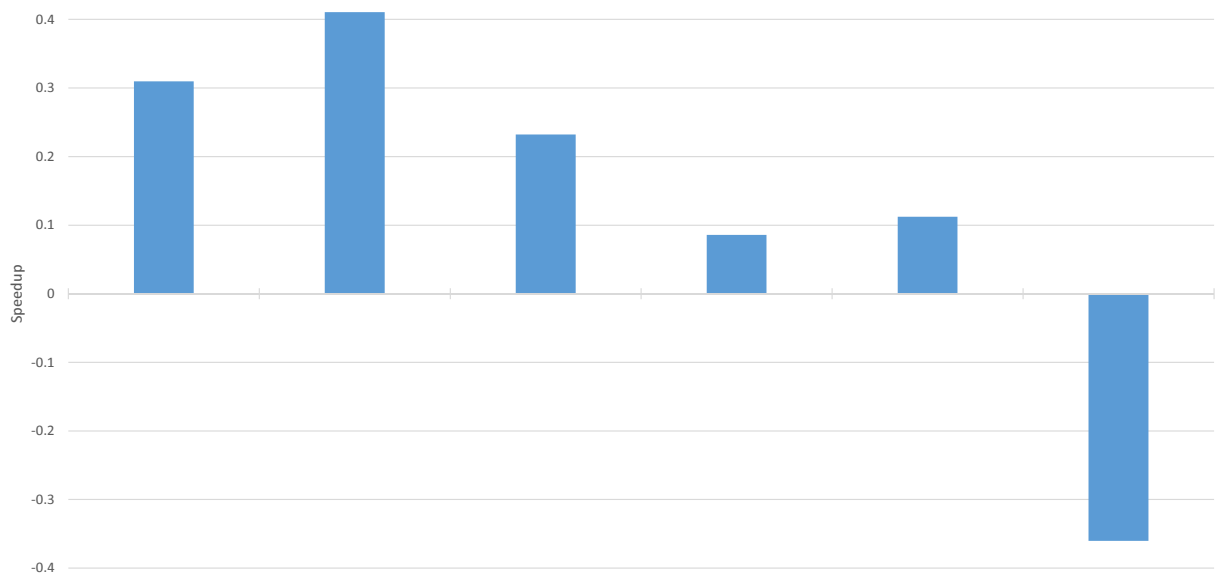

Figure 7: Speedup achieved by running two programs at once.

tests, however, losses were due to increased contention within the shared caches. When run separately, neither the insertion test nor the mult test have any instruction cache evictions. When run together, however, 26 evictions are seen. Even more telling, for the smtInsertionParallel_Long test, while parallel_long had 44 instruction cache evictions when run alone, when run simultaneously with Insertion, 492 evictions from the instruction cache were observed. This extremely high rate of cache contention explains the overall losses in performance when running the two programs in parallel.

3.5 Prefetching

Our processor uses a straightforward scheme to fetch instructions as far as two instructions in advance. We used the branch predictor to determine the next two predicted addresses and, given that they were not in the cache already and that nothing else is using the memory bus, we issue advance requests to memory for them. Given that we used a modest prefetching scheme and used a fairly small memory latency, we did not expect to see more than modest gains in performance from prefetching. This is, in fact, what we saw, but we remain pleased with our decision to add this feature given that it took so little effort to implement.

At our clock period of 12.51ns, prefetching decreased the runtime of our five test benchmark (copy, objsort, insertion, parsort, fib_rec) by 2.68%. On long programs with less instructions than instruction cache slots, it makes sense that prefetching does not help, and may even hurt. It may help slightly the first time through the loop, but after that, with all instructions already loaded into the cache, there is no benefit to prefetching and it may even hurt with putting unnecessary data into the cache on mispredicts. With a better branch predictor and/or better prediction logic or a larger instruction cache, prefetching might have been more effective. However, in our processor, prefetching is a cheap, modest improvement.

4 Group Dynamics

In regards to the division of work, each member performed the tasks set out for them by themselves and others in our team meetings.

Looking at the modules in our processor, the majority were written by Austin and Branden. Austin wrote the ROB, fetch, decode, and cdb_arbitrator modules. He also wrote the SMT changes to the reservation station and the commit modules. Branden wrote the RAT, RRAT, PRF, and commit modules. He also helped design the branch predictor and BTB along with Paul and Ben. Lastly, he performed updates on the three functional units.

The caches were handled exclusively by Paul. He wrote the instruction cache and data cache with non-blocking features, and added a prefetch to our instruction cache. He also wrote the mem_arbitrator

module.

Todd wrote the three functional units, the ALU, multiplier, and memory/branch unit.

All members helped write the base reservation station for Milestone 1.

The testing was done by Todd, who wrote testbenches for every module that Austin and Branden wrote, keeping consistent with our policy of testing modules that you did not write. He also wrote test benches for the three functional units.

The cache testing was done by Paul and the branch predictor was left without a testbench. The reasoning behind this was that the group believed the predictor could not hurt our correctness, only our performance.

Upon integration of our processor, all group members were present during the last two weeks to assist in debugging. Paul was the driving force behind synthesis and dealing with the many issues that arose in that department. Todd worked on any correctness issue that popped up for our base processor or because of SMT. Austin and Branden were both working on improving our performance and helping out with every issue we dealt with near the end. The importance of their discussions and input for any major concern on the processor cannot be overstated. Ben developed tests to identify bugs during integration.

Additional tasks that were completed by group members include Paul setting up the `ncurses` visual debugger, Austin auto-generating the top level module and keeping it updated, Branden working on our top_level testbench and managing the git housekeeping.

Ben was ineffective during the last part of the project due to him falling behind due to an unforeseen absence.

Percentages:

- Todd: 22%
- Branden: 22%
- Austin: 22%
- Ben: 12%
- Paul: 22%

5 Conclusion

Our final synthesizable processor achieved a final clock period of 12.5 nanoseconds and an average CPI of 1.88 for the larger test cases. Our processor arrived at the correct final state for every test program that was supplied and every test program we added for analysis. Our initial timeline was followed for the most part, and the project was all but submitted by the evening before the due date.

We feel that our processor's SMT capabilities make it very unique, since it's an extra feature that is not commonly attempted by 470 students because of the complexities of supporting two threads. Given this, we felt that SMT/superscalar was difficult enough to accomplish, so we devoted less attention to low-level optimizations and extra modules like a load store queue or an early branch resolution module. We think that our processor achieves a reasonable tradeoff between being able to run multiple programs at once and single program performance. We hit the minimum clock period that offered a reasonable memory latency for our situation. And n the end, we were more than happy to just complete a working processor with simultaneous multithreading; the performance numbers were simply a bonus.

Appendix A: Block Diagram of Processor