EECS 470 Final Project Report

Harsha Valsaraju[#1], Kush Goliya[#2], Sean McLoughlin[#3], Subhankar Pal[#4]

CSE Department, University of Michigan, Ann Arbor
Team: Teamy_McTeamface

[1]harvv@umich.edu
[2]kgoliya@umich.edu
[3]smclough@umich.edu
[4]subh@umich.edu

***Abstract*— In this paper, we are presenting the MIPS R10000 2-way superscalar processor which our group has developed over the course of the semester. We began our design with creating the MIPS R10K modules discussed in lecture. At this point, one of our group members dropped out of the class, which added an extra challenge to our project. We then designed a complete stage, and hooked up the instruction fetch, decode, and execute stages from project 3 to our design, which lead to us being able to run assembly code on our processor. Next, we created a BTB and a bimodal branch predictor, which allowed us to pass mult_no_lsq. The next few weeks were spent designing our memory system with a LSQ and a non-blocking 4-way set associative write-back data cache. The last two weeks were spent debugging our processor to ensure its correctness, designing prefetching the BHT, and the victim cache, and doing performance analyses on different parts of the processor. This part was challenging, as there was a lot of bugs in our code as the deadline approached. In this paper, we will elaborate on our testing procedures, project design, and optional features to improve our processor's performance.**

***Keywords*— R10000, Superscalar, Caches, Prefetching, Testing**

I. Introduction

At the beginning of the semester, our group was given the task of creating an out-of-order processor in SystemVerilog with a multitude of advanced features. When given this assignment, our group decided setting several goals for the project:

1. 100% correctness with our processor
2. Form an in-depth understanding of out-of-order processors
3. Have a good IPC and clock period

In order to complete the given task and fulfill our group's goals, we decided to implement a MIPS R10000 scheme processor. Our processor features a 2-way superscalar width, a non-blocking 4-way set associative multi-ported write-back data cache, variable stride prefetching, a victim cache, and a branch history table with bimodal branch predictors.

We also created a multitude of scripts to help with our debugging and testing. These scripts include an automatic minimum clock period finder, a testing script which runs a set of programs on our processor and the processor from project 3 and checks for any differences, and an elaborate random assembly code test generator. We also have a debug mode for our processor, which will be explained later in this report. Using all of these tools, along with having a strong focus on unit-level testing, we were able to catch many different corner cases on our processor and reduce our testing and debugging time to only a few days.

In this paper, we will describe the stages of our processor in detail. We will then analyze specific features of our processor and how they improve performance. Finally, we will describe the dynamics of our group and explain conclusions about our project.

II. Design

The following is an in-depth explanation of each stage of our processor.

A. Instruction Fetch Stage

Our instruction fetch stage contains accesses to the instruction cache, along with the instantiation of our branch target predictor. All fetched instructions go to a buffer between the fetch and issue stages.

1) Instruction Cache Prefetching: Our instruction cache supports stride prefetching up to four lines. When a PC is sent to the icache controller, if it misses in the cache, it is sent to a set of miss status handling registers (MSHRs) which store the next n*8 instructions to request memory, where n is the number of lines to prefetch.

2) Branch History Table and Branch Target Buffer: Our branch history table records up to 2 bits of branch history and maps the history to a bimodal branch predictor. Because we have two bits of history, there

are 8 entries within the BHT. The branch target buffer is a lookup table with 64 entries which stores branch targets when a branch has been executed.

3) *Instruction Fetch Buffer (IFB):* So we don't have to stall fetching instructions even if the processor cannot dispatch instructions, we decided to add an Instruction Fetch Buffer so that the processor can continue to fetch instructions when the processor cannot dispatch.

B. Decode and MIPS R10K Stage

Our decode and MIPS R10K stage contains the instruction decoder, along with all modules that are used in the MIPS R10K design. When an instruction issues, it goes to a pipeline register between this stage and the execute stage.

1) *Instruction Decode:* As we have a two-wide machine, we have instantiated two decoders so that we can decode two instructions at a time and then send them to the proper modules in the MIPS R10K stage.

2) *Reservation Stations:* Our processor has one Reservation Station for all Functional Units. We currently have 8 reservation station entries in our processor. We chose this number due to the fact that we have 4 functional units. We have noted that 8 reservation station entries increases performance over 4 reservation station entries.

3) *Reorder Buffer (ROB):* Our Reorder Buffer consists of 32 entries with each entry containing the tag from the free list, the old tag for the Architectural Map Table, and various signals to state whether or not the instruction was a branch, if the instruction was a mispredict or not, and the correct target pc if it was a branch mispredict.

4) *Free List:* The free list is a circular buffer which supports pushing physical registers in from the bottom when they retire, and popping physical registers out from the top when an instruction dispatches.

5) *Map Table:* The map table is a CAM with the addresses being each of the architectural registers. The data that is looked up is what physical register is the architectural register mapped to at the time.

6) *Architectural Map Table:* The architectural map table is very similar to the map table in how it is structured, but the main difference is that it contains the actual retired physical registers that represent the architectural registers at any point in time whereas the map table contains the in-use physical registers. The output of it will go to the map table when the pipeline flushes due to any incorrect instruction sequence, such as a branch mispredict.

7) *Physical Register File:* The Physical Register File (PRF) simply stores the values that physical registers write when the instructions that use them retire. This module is used to create our writeback.out and program.out files.

C. Execution Stage

After an instruction has issued, it is sent to one of the functional units within the execute stage. Our design consists of 4 functional units: 2 general purpose functional units, and 2 pipelined multiplier functional units, each being 2-wide. The reasoning behind so few pipeline stages was that it only raised our clock period by 1ns, but it substantially improved the IPC of our processor on almost every test case. The multiplier functional units have been modified so that they can now handle several multiply instructions passing through it at once. The general purpose functional units handle all other instructions. When the execution is complete, the instruction goes to an instruction buffer between the execution and complete stages.

D. Complete Stage

The complete stage consists of the execute complete buffer, which holds all instructions that finish execute and cannot complete. We then have two common data busses to send the complete signals back to the decode and MIPS R10K stage.

1) *Execute Complete Buffer (EXCB):* In order to handle more than two functional units in our processor since it is 2-way superscalar, we needed a way to store instructions that have finished execute, but have to wait on the common data bus as it can only complete two instructions at a time. Because of this, we were required to make an Execute Complete Buffer to store all executed instructions in between execute and complete. If the buffer becomes full, it will send a signal to the Reservation Station preventing any instructions from issuing for a cycle.

2) *Common Data Bus (CDB):* As we have a two-wide machine, we only have two Common Data Bus wires. These wires send a tag broadcast and valid bit to the MIPS R10K Stage for the Reservation Station and the Map Table. It also sends the branch address to the ROB in order for the ROB to determine if the branch is a mispredict or not when the instruction retires.

E. Memory System

The memory system of our processor consists of a data cache controller, a data cache, a load-store queue, and a victim cache. Loads and stores go into the load-store queue and they issue into the data cache when they have calculated their addresses.

1) *Data Cache:* The data cache in our processor s a non-blocking 4-way set associative multi-ported write-back data cache. It is non-blocking so that we are able to send multiple requests to memory at the same time. It is also multi-ported so that we can have multiple accesses to the cache at the same time. We determined through our data cache analysis that a 2-way set associative cache was the most optimal number of sets. Finally, the write-back property of our data cache allows us to not have a large memory latency every time there is a store instruction, as stores will only make cache blocks dirty as opposed to writing through the cache.

2) *Load-Store Queue (LSQ):* The Load-Store Queue we have follows a size heuristic of 20% of the size of our ROB. The LSQ has load to store forwarding, which allows loads to execute faster when they are dependent on a pending store.

III. Testing and Verification

The following explains how we verified that our processor was working as intended.

A. Random Test Generator

Our random test generator (random_test_gen.py) helped us catch many bugs, especially with the branch target buffer. The generator creates a random test case from the instruction set that we are using for EECS 470, but without loads, stores, or subroutines. The reasoning behind not including these different instructions is that it is very difficult to get loads and stores to execute without memory errors when creating a random set of code, and subroutines would be exceptionally difficult to randomly generate. Because of these restrictions, the random test generator was mainly used for debugging our code before milestone 2.

Branches are guaranteed to not create an infinite loop due to some restrictions on how branches are placed: conditional branch instructions are the only instructions that use register 29, and the condition on these instructions must eventually not be met, so the branch instruction will not execute. Therefore, depending on the type of condition, an add or subtract instruction on register 29 is included in front of the branch instruction so that register 29 will eventually not meet the condition of the branch. If a branch is taken, it will go to the instruction after the previous branch. There are no forward branches, and there will never be nested loops with branches. Because of these restrictions, creating a random test generator with branches was not terribly time consuming, but it saved us a lot of time for finding corner case bugs.

B. Testing and Output Comparison Script

The Testing and Output Comparison Script (test.sh) lets us run any test name that we have created within our test_progs directory and compare it against the stock project 3 output. If both of the writeback.out and program.out files are the same, then the script will print that the test passed. If the files are different, then the script will print failed, and it will open Vim to show the differences between the two files. This has been useful for confirmation that a test is working, along with a good starting point for debugging any potential problems with a test case.

C. Debug Mode

The DEBUG parameter which we have defined in our sys_defs.vh file will print out timestamps of each MIPS R10K module, along with buffers such as the IFB and the EXCB. This mode was used after running a test in our testing script to figure out specifically where a problem occurred. The debug mode printed program counter numbers for instruction retirement, and using that information, along with our Vimdiff from the testing script, we were able to find out exactly where in our pipeline a problem arose. From there, we were able to understand the exact problem and how to fix it relatively quickly.

We were considering creating a visual debugger using these outputs, but we decided against that because it was much easier for us to search through our file that we printed to as opposed to going through each cycle of a test's execution.

D. Testing Procedures

We used the given tests from project 3 to confirm that we were able to pass the tests after we turned our project in. This was difficult at first, as our memory system had a lot of bugs, but after passing the first few test cases, many others began to work too. The two tests that revealed the most bugs for us were objsort and fib_rec.

Once we were able to pass all of the given tests, we started to use the Decaf470 compiler and the tests that came with it to catch any final bugs in our processor. After running all the given test cases, we initially passed nearly all of the given decaf test cases, so this method only helped us find a few more bugs.

When we found that our processor worked on all of the above test cases, we decided to begin performance analysis of our processor by varying different structure sizes and turning certain features off. We believed that this method would shuffle the program execution order to expose more bugs in our processor. While this didn't reveal as many bugs as we thought it would, it did reveal some, which we fixed and then continued our analysis.

Finally, we decided to vary memory latency on our processor to reveal any more bugs that we had not caught. However, we did not catch a single bug using variable memory latency. Because we did not catch any bugs doing this, we were confident that our processor was functionally correct.

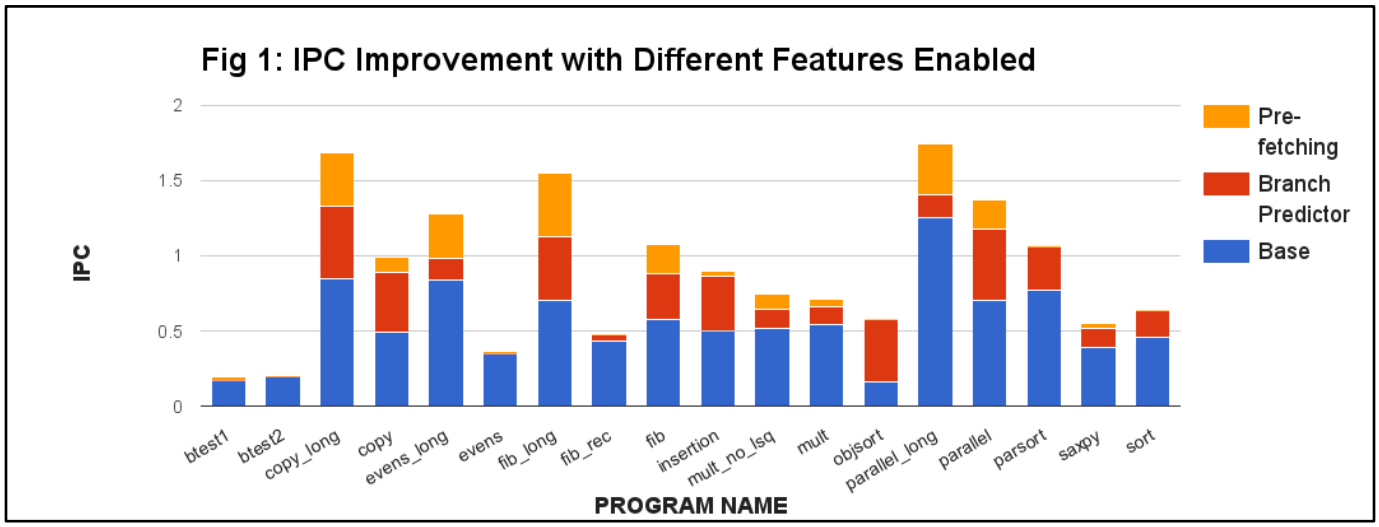
IV. Performance Analysis

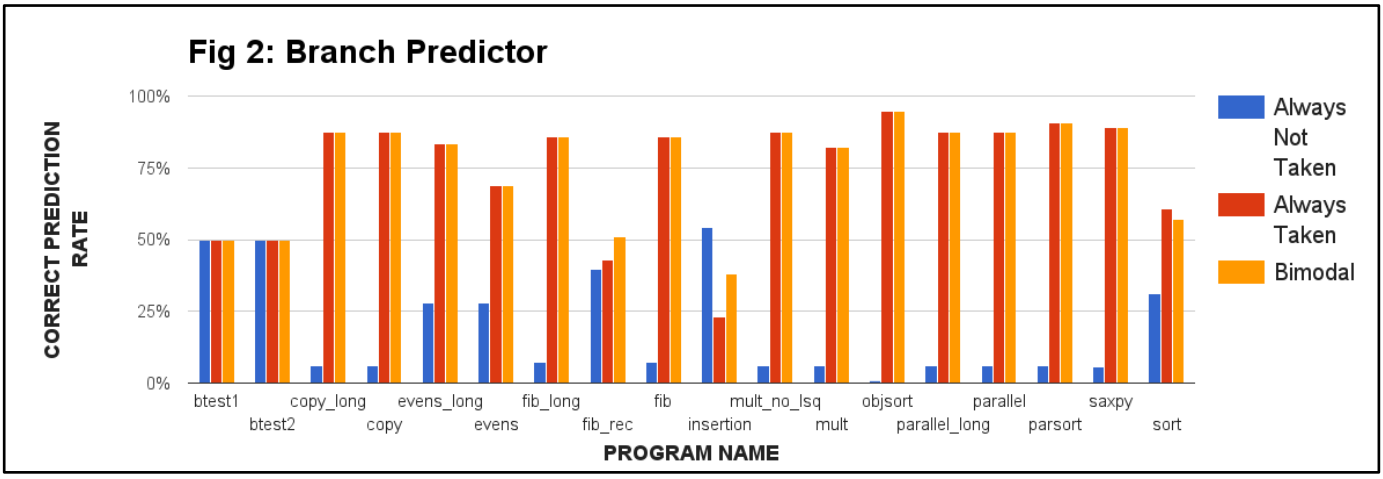
The following provides some analysis on the performance of our processor with and without certain features turned on across a wide variety of provided test cases. Our team provided analysis on overall IPC improvements, branch prediction correctness rate, instruction cache hit rate with variable stride prefetching, data cache hit rate with varying associativity, and an analysis of choosing the BHT over the bimodal branch predictor.

A. Overall IPC Improvements

Figure 1 contains an overview of the IPC improvements in our processor for each of the test cases in project 3 as we incrementally turned on our features. The base design is with the branch predictor predicting always not taken and no prefetching enabled. The branch predictor is our bimodal branch predictor, and our prefetching is 4-line stride prefetching.

There are a few inferences that can be made about this chart:





- Tests that benefit the most from the branch predictor likely have simple loops that are always taken. btest1 and btest2 both do not receive benefits from the branch predictor because no branch is executed twice in those tests.
- Tests that benefit the most from next line prefetching likely have long sequences of code before a branch instruction. Tests such as objsort do not get as much benefit from prefetching likely because of many branches/subroutines, along with a large amount of instructions causing the instruction cache to fill up.

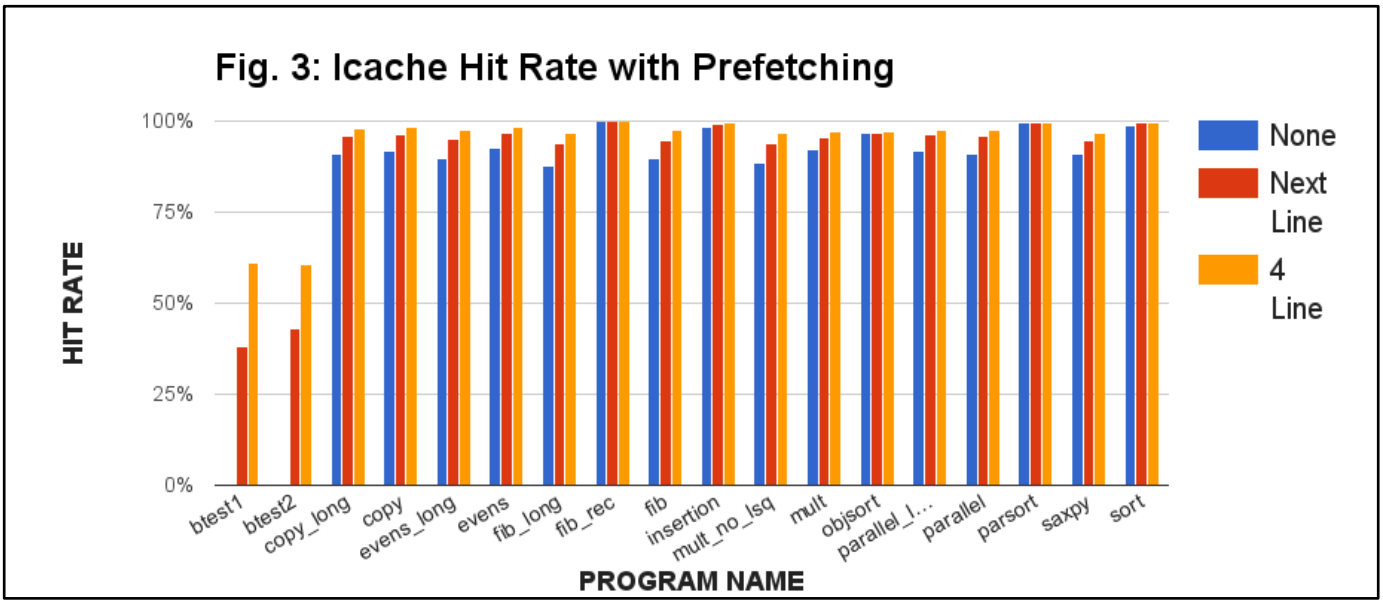
There is also an abnormality that we observed from looking at this chart. Even though mult_no_lsq only has one branch, next line prefetching did not help the program much. We assume this might be the case for a few reasons:

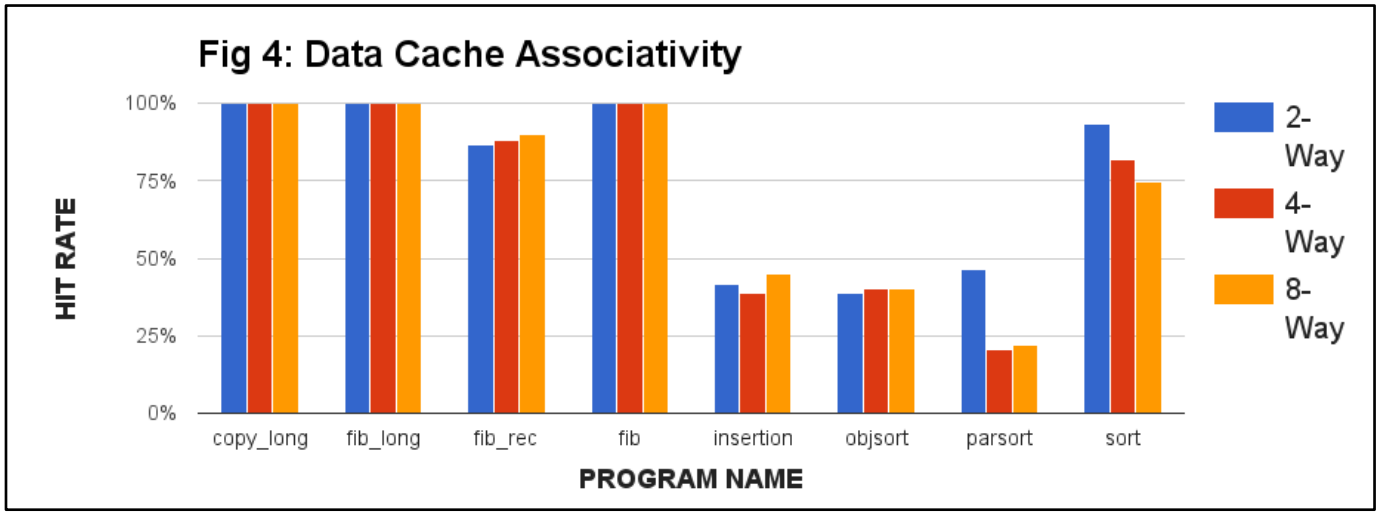
- There are not as many instructions as tests with nops such as copy_long.
- The branch instruction is not taken as much as in other tests, so there are not as many instructions to execute.

B. Branch Predictor Analysis

Figure 2 shows the analysis that we performed on our bimodal branch predictor. We measured the correct prediction rate with the static predictions of always taken and always not taken, and then with the dynamic prediction of our bimodal branch predictor.

The branch predictor analysis was somewhat disappointing. Even predicting that the branch was





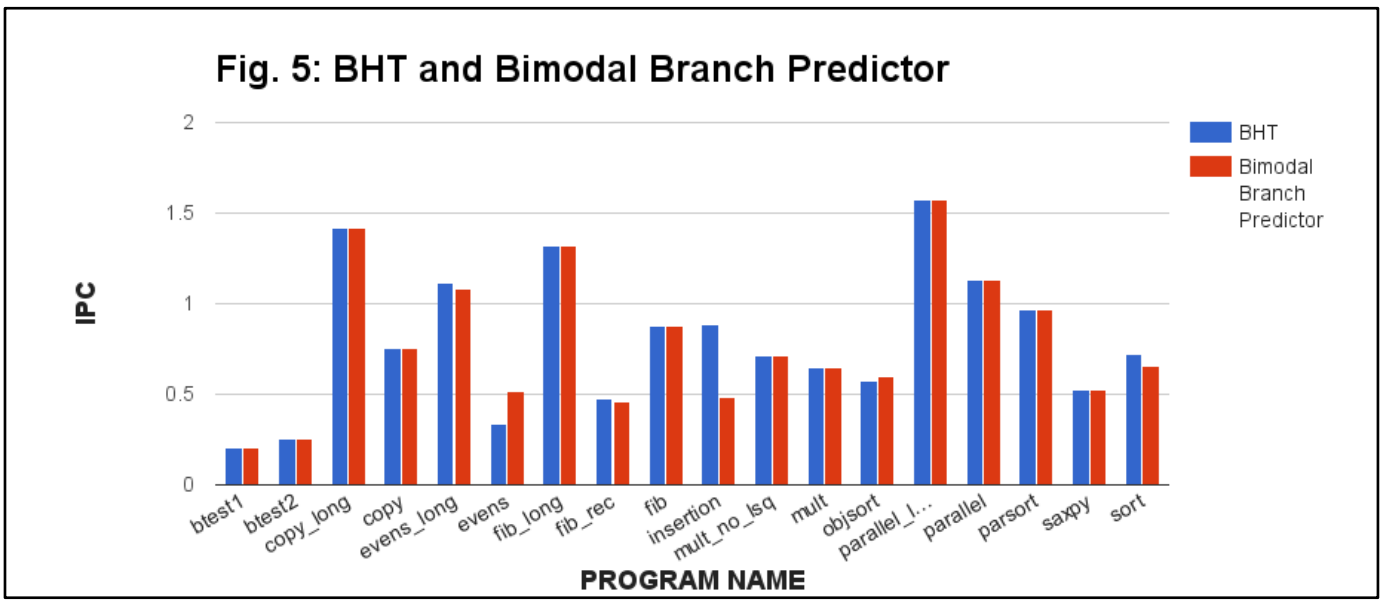
always taken provided nearly identical correct prediction rates to bimodal prediction, and in the case of sort, it was actually a better prediction rate. The important thing that we noticed from this analysis was that both btest1 and btest2 have no differences in how to predict the branches taken. This is because no branch is ever executed twice in these tests, so it is very difficult to increase IPC on these tests based on the branch predictor implementation.

C. Icache Prefetch Analysis

Figure 3 shows our analysis on our instruction cache hit rate with variable stride prefetch length. We varied the stride length of our prefetcher and measured the hit rate for each of the given test cases.

The instruction cache hit rate was already relatively high without any prefetching. However, the more that stride prefetching ran, the better the hit rate was, even if it was only marginally better. There are a few interesting things to discuss about this chart:

- Both btest1 and btest2 run substantially better with 4 line prefetching as opposed to next line prefetching, and no prefetching made the icache miss every time. We believe the reason for this happening is because the branches in both btests may take you only a few instructions ahead, and the prefetcher will have taken these instructions into the icache already. Because of that,



these instructions that were branched to would already be in the instruction cache.
- Tests such as fib_rec look like they have a 100% hit rate in the instruction cache. This is not the case. The hit rates just happen to be insanely high (around 99.99%).

D. Data Cache Associativity Analysis

Figure 4 shows our analysis on our data cache hit rate with variable amounts of associativity. We measured the data cache hit rate for various associativities on the given test cases and compared the results.

We were initially surprised with these results. We expected that the higher associativites would get a higher hit rate on all test cases. However, we realized that this was not the case because we have a pseudo-random removal policy for the data cache in order to lower our clock period, as an LRU policy would raise it substantially. Because of this, we decided to finalize our design with a 2-way implementation and attempt to reduce our clock period even more.

E. Branch History Table Analysis

Figure 5 shows our analysis on our branch history table. We measured the IPC of the given test cases against our bimodal branch predictor and compared the results.

We implemented the branch history table in order to improve on our bimodal branch predictor, which was causing test cases with lots of branches to underperform. The average IPC of using the BHT is better than that of the bimodal branch predictor. Because of this, we decided that the BHT was a better design than our bimodal branch predictor, and that was the predictor that we used in our final design.

V. GROUP DYNAMICS

Harsha (25%): Victim Cache, Data Cache, ROB, RS, Store Queue, Non-blocking multiplier, FUs, BHT

Kush (25%): IF stage, Prefetcher, Overall integration, FUs, BTB, Minimum clock period finder, Branch Predictor, MSHR, Store Queue, Data Cache Controller

Sean (25%): Map Table, Complete Stage, Stride Prefetcher, Load Queue, Random test generator, MSHR

Subhankar (25%): Data Cache, Data Cache Controller PRF, Load Queue Free list, BTB, ROB, Victim Cache, Testing scripts, BHT

VI. CONCLUSION

We have completed our semester project by creating a MIPS R10K 2-way superscalar processor. This project taught us a multitude of things, such as how to debug difficult errors, how out-of-order processors work on a very detailed level, and how to work as a team with a large code base. Due to a team member dropping our group, we decided to lower the amount of features that we would have and achieve a lower clock period. We were able to finish our design with a clock period of 13.14 and an average CPI of 1.61 among the given test cases. Due to our debugging methods, we believe that we have achieved our goal of 100% correctness, and we also have decent performance. We learned the most about out-of-order processing when we implemented the Load-Store Queue with store to load forwarding, stride prefetching, and our write-back data cache.