University of Michigan

EECS 470: Computer Architecture

Final Project Report

2-Way Superscalar R10K Out-of-Order Processor

Group 2

Yichen Yang

Yuhan Chen

Zixuan Li

Haiyang Jiang

Tian Pang

December 7, 2018

Contents

1 Introduction **1**

1.1 Project Overview 1

2 Design and Components **2**

2.1 R10K Style Design 2

2.1.1 Out-of-Order Execution Algorithm 2

2.1.2 Register Renaming 2

2.1.3 Branch Miss Prediction Squash Technique 2

2.2 Number of Superscalar Ways 3

2.3 Re-order Buffer 3

2.4 Reservation Station 3

2.5 Functional Units 3

2.6 Data Memory Interface 3

2.6.1 Load Store Queue 3

2.6.2 Non-Blocking D-cache 5

2.7 Common Data Bus & Common Data Bus Buffer 5

2.8 Fetch Technique 5

2.8.1 Tournament Branch Predictor & Branch Target Buffer 5

2.8.2 Return Address Stack 6

2.8.3 Instruction Buffer 6

2.9 Instruction Prefetch 6

2.10 Victim Cache 6

2.11 GUI Debugger 7

3 Analysis **7**

3.1 Number of Superscalar Ways 7

3.2 Reorder Buffer & Reservation Station 8

3.3 Functional Units 9

3.4 Data Memory Interface 9

3.4.1 Load & Store Queue 9

3.4.2 Non-Blocking D-cache 11

3.5 Common Data Bus & Common Data Bus Buffer 12

3.6 Fetch Technique 12

3.6.1 Tournament Branch Predictor & Branch Target Buffer 12

3.6.2 Return Address Stack 14

3.6.3 Instruction Buffer 15

3.7 Instruction Prefetch 15
3.8 Victim Cache 17
3.9 Critical Path Reduction 18

4 Results **19**

5 Group Contribution **19**

6 Conclusion **19**

7 Future Improvements **20**

Abstract

An in-order processor executes instructions to guarantee the correctness of execution. However, it can hurt the performance of processor due to unnecessary stalls. The idea of out-of-order processor solves this problem by allowing younger instructions to bypass older instructions as long as there are no dependencies. Out-of-order processors can better exploit the hardware by occupying hardware resources whenever possible.

1 Introduction

1.1 Project Overview

The term project is to build on the Alpha64 pipeline to create a more advanced pipeline with a few of the features we are studying in *EECS470: Computer Architecture*. The top level design of our project is a 2-way Superscalar R10K out-of-order design.

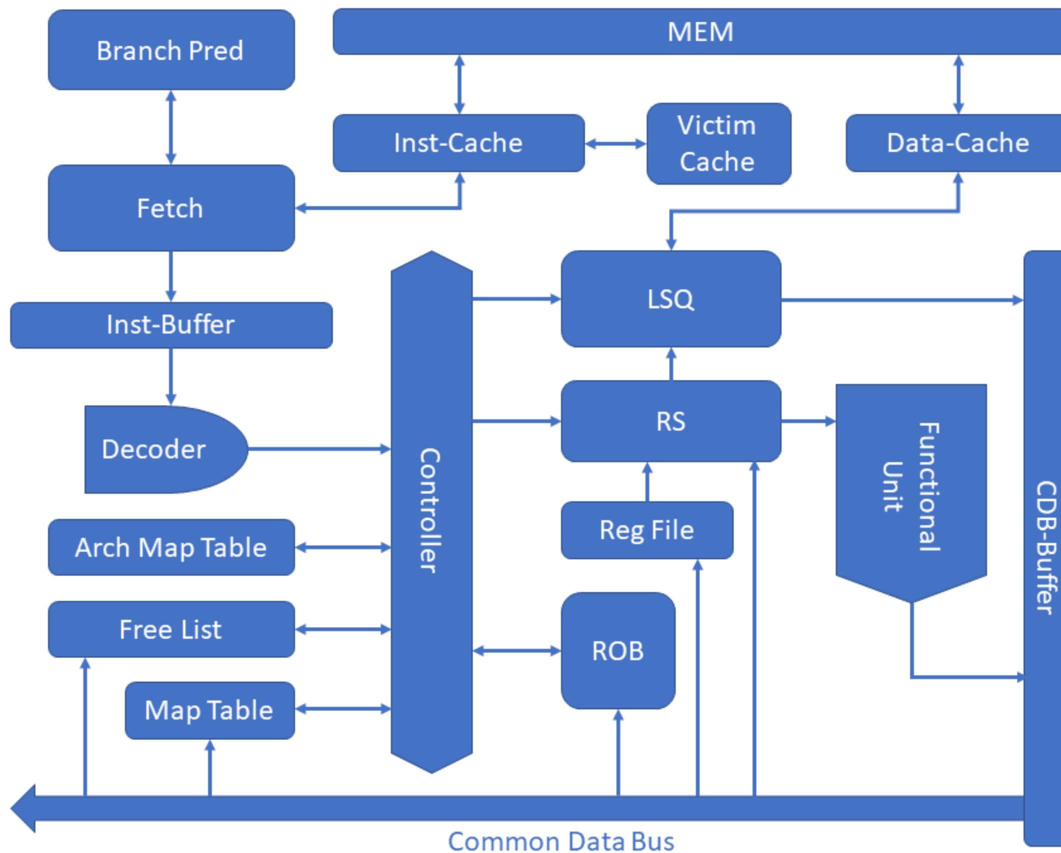

Figure 1: Project Overview

Figure 1 is the top level overview of our out-of-order processor design (note that the victim cache is not included in the final submission). The arrows indicate there are signal exchanges between two modules. Not all signals are wired in the figure, and many details are not revealed in the figure.

To further improve the performance, we have designed and implemented separate instruction and data caches, multiple functional units with different latency, tournament branch predictor with branch target buffer, return address stack, instruction prefetching, victim cache and a load-store queue with data forwarding.

With these advanced features, we have achieved 7.7*ns* clock period with the average CPI 1.62 (shown in Table 1). Note that the average CPI ignores *btest1*, *btest2* and *halt*.

Table 1: Final Result

Avg. CPI	1.62
Clock Period	7.7

2 Design and Components

This section briefly introduces the design of the components in our project. Detailed analysis about the size and effect of each component is covered in the **Analysis** section.

2.1 R10K Style Design

In this project, we chose R10K structure as the basic design. We also added additional features to further improve the performance.

2.1.1 Out-of-Order Execution Algorithm

The high level out-of-order execution algorithm is to allow younger instruction to go before the older instruction. False data dependencies are eliminated by Register Renaming, instructions with true data dependencies are set aside to wait for the needed data in Reservation Station, so that the younger instructions will not be blocked. Although the executions can be out-of-order, the dispatch and retirement of all instructions must be in order to ensure the execution results are the same as in-order processor. All instructions are considered finished execution only when retired, otherwise they can be squashed as if never executed. More detailed implementation of each components is covered in the following context.

2.1.2 Register Renaming

Write-after-Read (WAR) and Write-after-Write (WAW) data dependencies are considered false data dependencies because they are avoidable, and Register Renaming is the technique used to eliminate the false data dependencies. False dependencies can go wrong because they read from or write to the same register, so if they are executed out of order, they may get the wrong data. By using Register Renaming, the architectural register used in assembly code will be mapped to a physical register, and the mapping information is stored in Map Table. The destination register of each instruction is mapped to a physical register, which ensures to eliminate any false data dependencies.

2.1.3 Branch Miss Prediction Squash Technique

Branch miss prediction squash signal is sent by ROB. Once one of the instructions, which is going to retire, is a branch instruction and the prediction is incorrect, the squash signal is set to high and is sent to each component in the processor. Each component clears all or part of its contents when they receive the squash signal.

2.2 Number of Superscalar Ways

The more superscalar ways we have, the more instructions we are able to run each cycle, meaning that we can have smaller CPI. However, the gain from adding more ways decreases dramatically as the number of ways increases, which means the gain of increment from 2 to 3 way is much smaller than that of increment from 1 to 2 ways. This is because the more ways we have, the more likely that instructions have dependencies. Also, the increment of ways makes the logic more complicated, and may increase the cycle time. As the execution time is the CPI multiplies the cycle time, we may not get much gain from making the number of ways too large. We decided to implement a 2-way superscalar out-of-order processor because it makes a great performance boost comparing to single way, yet the complexity of implementing it is relatively low.

In order to increase the performance, our design can dispatch 2 instructions, issue 6 instructions (including 3 ALU, 1 MULT, 1 LD and 1 ST instructions), execute 3 ALU instructions and 2 load instructions, complete 2 instructions and retire 3 instructions simultaneously.

2.3 Re-order Buffer

ROB is one of the key components to make sure the program is executed correctly. Instructions are sent to ROB and stored in order when dispatched. ROB is responsible for retiring the instruction in order to make sure the program is correctly executed. When mis-branched, ROB is responsible for squashing the instruction before it gets retired. Inside the ROB, an tail pointer is used to keep track of where to put the next instruction, and a head pointer is responsible for making sure the retirement is in order. ROB takes in the signal from CDB to determine which instructions are ready to retire; ROB also gets the correct branch target from CDB and compares it with the branch target stored in ROB certain entries, and if the branch target does not match, meaning the branch is mis-predicted, ROB clears all the entries and sends out squash signal to other components to eliminate the effect of mis-branch.

2.4 Reservation Station

Reservation Station is used to manage the issue stage. Instructions are stored in Reservation Station after dispatched. Reservation Station is responsible for determining whether an instruction is ready to issue. An instruction is issued when it has all the source data available, given that the associated function unit is available and the instruction is selected by the priority selector.

The Reservation Station has 16 entries. It takes in at most two instructions per cycle, and is able to issue at most 6 instructions per cycle, which consists of 3 ALU instructions, 1 multiplication instruction, 1 load instruction and 1 store instruction, and this corresponds to the number of Functional Units we have.

2.5 Functional Units

The Functional Unit consists of 3 ALU and 1 Mult. The ALUs are used to compute all the computations except for multiplication. The Mult unit uses an N-stage multiplier, and the number of stages is parameterized.

2.6 Data Memory Interface

2.6.1 Load Store Queue

Out of order memory operations are relatively harder to be implemented because large memory addresses make renaming mechanism not feasible. Also, memory addresses, unlike other instructions, may be ambiguous during execution.

We implemented load queue and store queue separately, with data forwarding from SQ to LQ. The sizes of LQ and SQ are both quarter of the ROB size.

Store queue (SQ) is a FIFO structure, just like ROB. A head and a tail are used to maintain the SQ. When a store instruction is dispatched, it is allocated in the SQ as the tail. After the address and value of this store instruction are available, they are written into the corresponding entry and labeled as resolved. Then the head entry is retired one by one according to the retire signals from ROB.

Load queue (LQ) is like RS. An instruction randomly allocates an empty entry in the LQ in the dispatch stage. The LQ records its ROB index and current SQ tail. After the instruction is issued from RS, the LQ stores the solved address. The LQ then forwards data from SQ if it depends on the SQ data, and it accesses the data from cache. When there is a cache miss, the LQ records the memory response and waits on the third port of cache, which is directly from the memory. The detailed state diagram is Figure2, and the interface between LQ and Dcache is in Figure3.

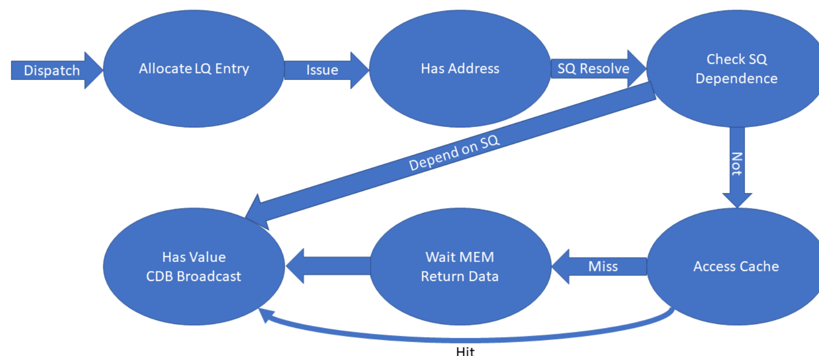

Figure 2: LQ State Diagram

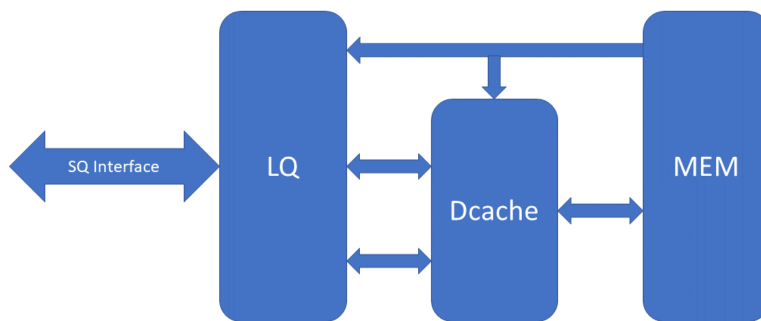

Figure 3: LQ & Dcache Interface

2.6.2 Non-Blocking D-cache

Data Cache(Dcache) is implemented parameterizedly. This allows quick analysis with respect to the number of ways. The block size of Dcache is fixed to 8 Bytes, because memory returns 8 bytes of data each cycle. The policy of our Dcache design is write-through and allocate-on-write.

Dcache allows two load instructions and one store instruction from load queue and store queue to access data each cycle. Dcache gives the highest priority to the store instruction, then the first load instruction and lastly the second load instruction, so Dcache always sends the store data and the memory address to memory given the store request exists. Dcache returns the data of both load request at the same cycle given that the data is available in Dcache. If any of the two load instructions gets miss on Dcache, Dcache sends the missed load address to memory given there's no store request. Dcache can send only one request to memory each cycle, no matter it's a store request or a load request; this is because memory can take in only one request each cycle. In the case of a load request, memory returns a response to Dcache the same cycle it gets a request, and Dcache returns the response to the load queue. Memory returns a data along with the tag that matches the response a few cycles ago; Dcache stores the data in its entry and sends the data and tag from memory to load queue.

The least-recent-used (LRU) is maintained by a four-bit counter for each entry in Dcache. When a specific set is accessed, the LRU counters for each way are incremented by 1, and the way that just get accessed resets the LRU counter to 0. When a set is filled, and a new coming data needs to be stored, the way with the largest LRU is replaced.

2.7 Common Data Bus & Common Data Bus Buffer

Common Data Bus(CDB) has two buses carrying the data that are to be broadcast to reservation station and re-order buffer, map table and register files. This number is constrained by the project rule.

Since the processor is designed to have 5 functional units(3 ALUs, 1 MUL, 1 LD) that need to use CDB, only 2 buses might hurt the performance because they could cause the functional units to stall. In order to reduce the number of stalls, CDB Buffer is added to the processor. CDB Buffer acts like a queue which saves the results of the functional units that exceed the limited number of buses the CDB has. Since CDB Buffer is designed to be a queue which is first-in-first-out(FIFO), the older results from the functional units are guaranteed to be broadcast earlier than younger results. The size of CDB Buffer is equal to that of ROB. The reason for that is discussed in section **Analysis**.

2.8 Fetch Technique

2.8.1 Tournament Branch Predictor & Branch Target Buffer

The branch predictor consists of two parts: branch direction predictor and branch target address predictor. Tournament branch predictor works for direction prediction and BTB works for target address prediction.

Direction Prediction

Tournament branch predictor consists of two global direction predictors, Gshare and simple BHT. Simple BHT is a 2-bit counter, where 0 represents strongly not taken, 1 represents not taken, 2 represents taken and 3 represents strongly taken(this is default set for all 2-bit counters in fetch). The default value of each counter in tournament predictor is 1, which is weakly not taken. Every time branch result comes from retire stage, BHT updates the value based on its previous value. 8 bits of PC are used to index BHT to get its direction prediction. Gshare is more complex than BHT, which is made up of BHR and PHT mentioned in lecture. BHR stores the last 8 branch results. The index used for visiting PHT equals to the result of 8 bits of PC exclusive or BHR. Every index of PHT is also a 2-bit counter and the value in the visited index represents the predicted direction of Gshare. A chooser is used to select one of these two predictors' results as the final prediction. It's essentially also a 2-bit counter and works similarly as BHT and PHT. When the value of counter is 0 or 1, simple BHT is chosen; otherwise Gshare is chosen. Every time the prediction of Gshare is correct and BHT is wrong, the value of chooser increases by 1.

If Gshare miss-predicts and BHT predicts correctly, the value of chooser decreases by 1. Otherwise the value of chooser doesn't change. It deserves mention that all the results used by tournament branch predictor comes from retire state.

Target Prediction

BTB looks like a cache, which stores instruction PC tag, valid bit and target address. When an instruction takes branch for the first time, its PC tag and target address are stored in BTB and valid bit becomes 1, which means this target address can work as a valid target prediction for fetch stage. Once it doesn't take branch, valid bit is modified to 0. For the first time it takes branch again, valid bit is written back to 1. Every time another branch instruction with same index and different tag is to be written into BTB, the currently-stored instruction is removed. Every cycle fetch stage sends two PCs to BTB and BTB compares index, tag and valid bit to determine whether it provides valid target address prediction for them. Fetch stage calculates next-PC based on direction and target address prediction. The size of our BTB is 32, which is determined by number of bits of index. The reason for that is discussed in section **Analysis**. Our BTB is speculate, meaning that all the results used to update BTB comes directly from execute stage.

2.8.2 Return Address Stack

Return address stack provides branch target address for return instructions specially. Every time a call instruction is detected in fetch stage, PC+4 is stored in a FILO(first in last out) stack. When a return instruction is detected in fetch stage, the top address in the stack is taken and given to fetch stage as branch target address. This really improves the performance when the processor is running test cases with many return instructions, e.x. *objsort*.

In our design, the size of RAS is 64, which can hold enough target addresses for return instructions. The reason for that is provided in section **Analysis**.

2.8.3 Instruction Buffer

Instructions from instruction cache are stored in the Instruction buffer(Ibuffer), and the decoder requests directly from the Ibuffer for the next instruction to decode. The purpose of having an Ibuffer is to prevent unnecessary stall in the fetch stage. When RS or ROB is filled and can no longer take in instructions, the Ibuffer can still have the space to store incoming instruction. The size of our Ibuffer is twice the size of ROB. In the case of that Ibuffer is filled, the fetch stage will be stalled.

2.9 Instruction Prefetch

Since memory delay is significant in the design, it is important to make use of the cycles during memory delay to improve the performance. Instruction prefetching is added to Icache controller to improve the performance of instruction fetching.

Every time Icache controller receives fetch request of a certain address from processor, 8 "latter" addresses starting from "Next-PC" calculated in fetch stage are written to prefetch buffer, the size of which is 32, if they aren't in Icache or in the buffer. As long as there are requests not sent to memory, Icache controller would send them to memory in order when there is no new fetch request from processor. Data is written to Icache when memory tag matches with response stored in prefetch buffer.

Instruction prefetching improves the performance of our design to a great extent. Detailed description and data is in section **Analysis**.

2.10 Victim Cache

Based on former EECS 470 students' experience, we decided to only write victim cache for Icache because victim cache works little on performance of Dcache.

There are 4 entries in the victim cache, which stores the data “kicked out” by Icache. The block that accepts the kicked-out data is determined by LRU. It’s guaranteed that no entry is in both Icache and victim cache simultaneously. Once an entry in victim cache is called by processor again, it is written back to Icache and data in that index is kicked out to victim cache. If memory and victim cache want to write to the same index in Icache, data from memory has the priority.

Although victim cache increases Icache hit rate greatly, we decided not to add it to our design because it hurts BTB prediction correctness drastically. This is explained in detail in section **Analysis**.

2.11 GUI Debugger

In such a large project, both coding and debugging can be challenging. Considering the potential heavy debugging work of this project, we implemented a visual debugger. It has the same structure as the one in Project 3, which is based on C and snooping outputs from Verilog testbenches. It can display values and changes of all the main tables and queues in our processor as well as the important data and control signals cycle by cycle. Also, it can be self-adapting to all module sizes.

It turns out that the visual debugger was really helpful in our debugging phase. We can observe the data moving clearly and spot the behavioural exception rapidly and then find the bugs in the internal logic with DVE.

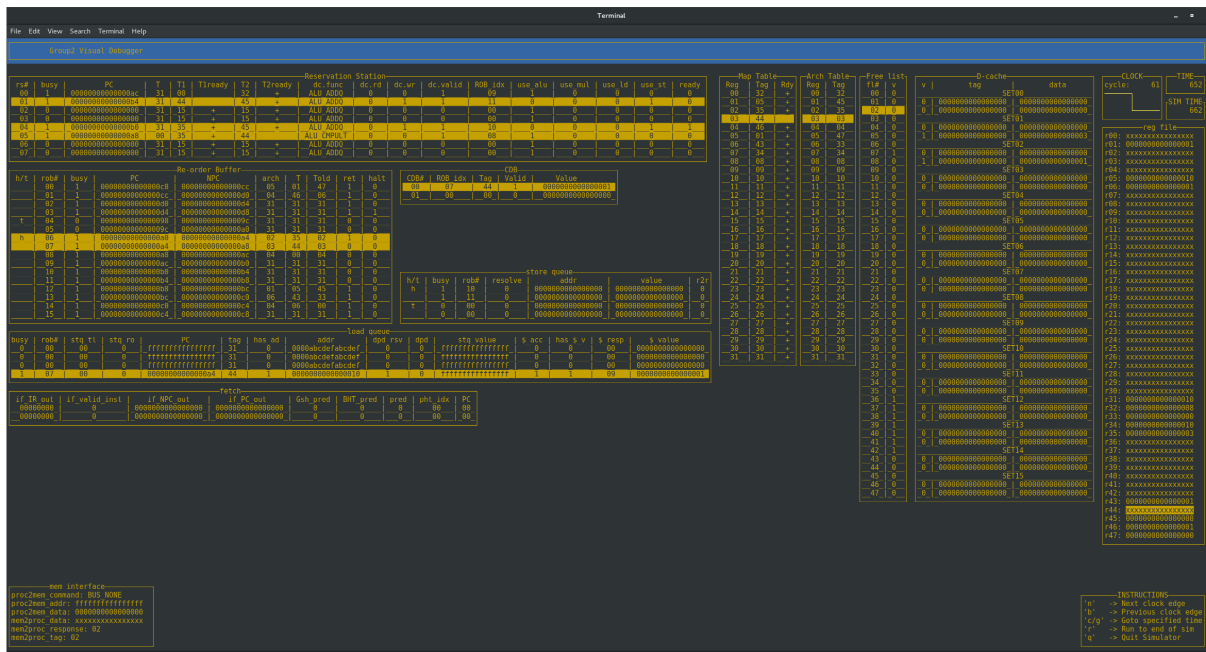

Figure 4: Visual Debugger

3 Analysis

3.1 Number of Superscalar Ways

As our design is 2-way superscalar, we should have some test cases have 1 <IPC< 2. Based on the result in Table1, there are a lot of test cases, whose CPI is smaller than 1 and larger than 0.5. *copy_long* has 0.58 CPI, which means its IPC is 1.72.

3.2 Reorder Buffer & Reservation Station

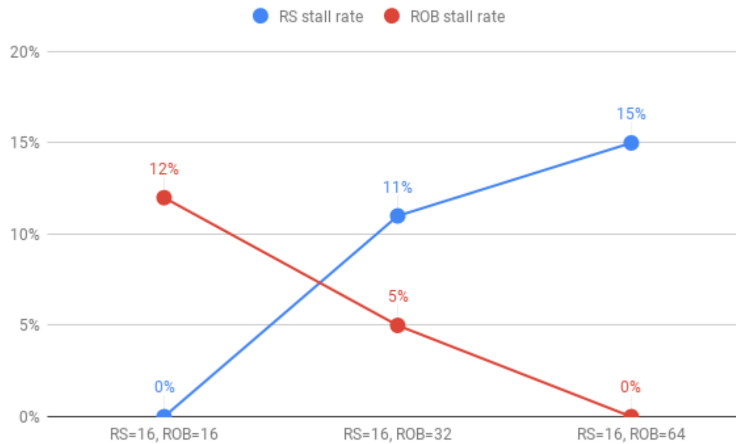

Figure 5: ROB/RS Size Ratio vs. RS and ROB Stall Rate

We first tried to discover the optimal ratio of RS and ROB size, so we fixed the RS size to be 16, and tested with ROB size being 16, 32, and 64 respectively. According to Figure 5, when ROB size is 16, the RS never stalls because ROB is filled before RS needs to stall. When ROB size is 32, ROB stall rate decreases to 5% and the RS stall rate increases to 11%, meaning that both RS and ROB can be filled. When ROB size is 64, ROB never stalls because RS is filled first. Because stalling either ROB or RS stalls the fetch stage, making the ratio of ROB/RS too large or too small is meaningless, we decided to keep ROB twice as large as RS.

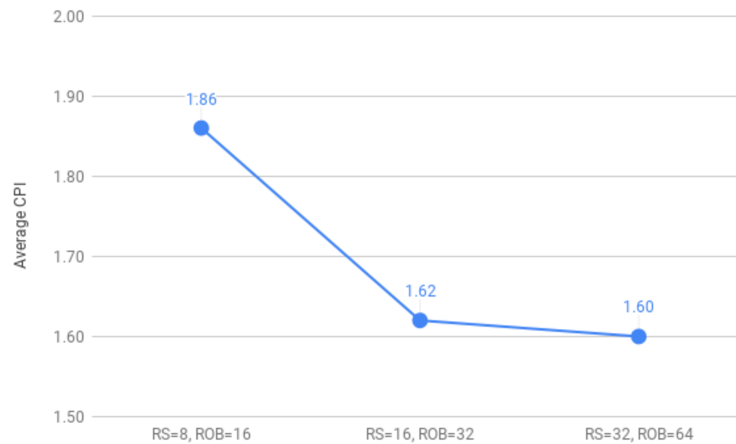

Figure 6: RS Size vs. CPI

We tested with RS size being 8, 16 and 32, and fixed ROB size twice the size of RS. According to Figure 6, the result shows increasing the size from 8 to 16 can decrease the

average CPI dramatically, but further increasing the RS size to 32 brings little improvement, because stall caused by RS and ROB is very small. Considering that having more entries in RS and ROB may increase the clock cycle, the little improvement does not worth doing it.

3.3 Functional Units

Our processor has three ALUs. While more than 3 might yield better performance, three ALUs are optimal because we focus on the 2-way superscalar, which limits the number of instructions to be fetched and broadcast at the same cycle. Three ALUs are not the performance bottleneck. Another functional unit is multiplier. Two choices are available - 4-stage or 8-stage multiplier. Figure 7 shows the CPI using the multiplier with different stages. 4-stage multiplier yields lower CPI especially in the test program "multi_no_mem". However, 4-stage multiplier has longer critical path. We chose to use 8-stage multiplier finally because it yields the overall shorter runtime.

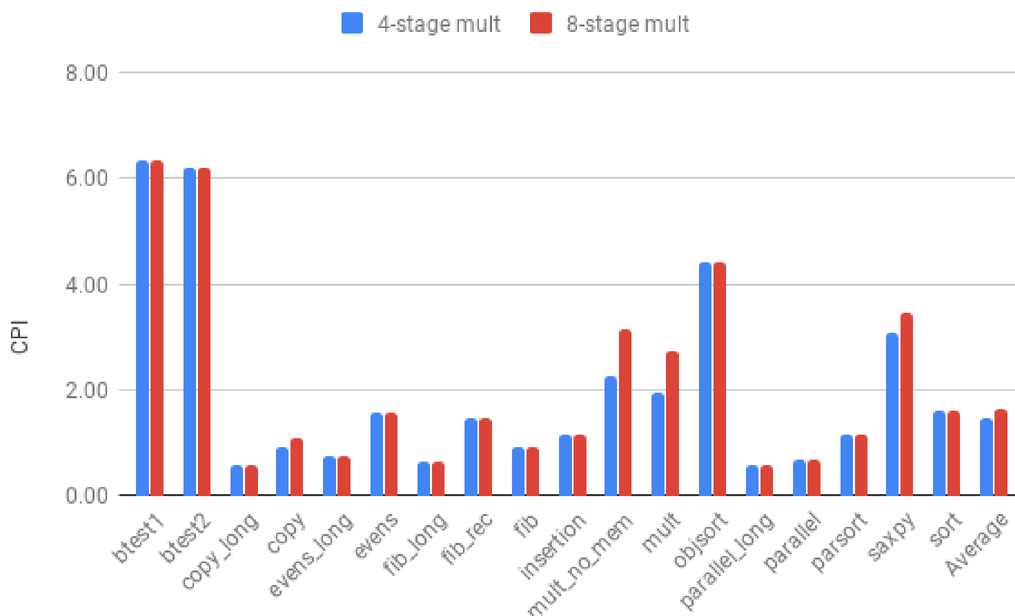

Figure 7: CPI vs. Different Multiplier Stages

3.4 Data Memory Interface

3.4.1 Load & Store Queue

We tested difference size of LSQ (1/8, 1/4 and 1/2 of ROB size) and collected the data of LSQ stall rate and CPI. We got the following result (Figure 8, 9).

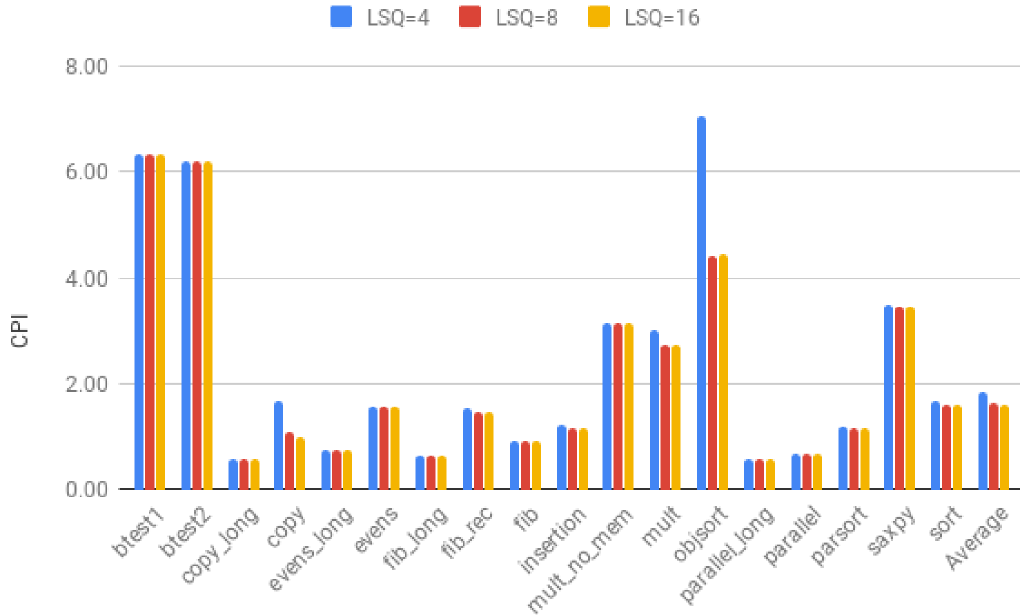

Figure 8: CPI vs. LSQ Size

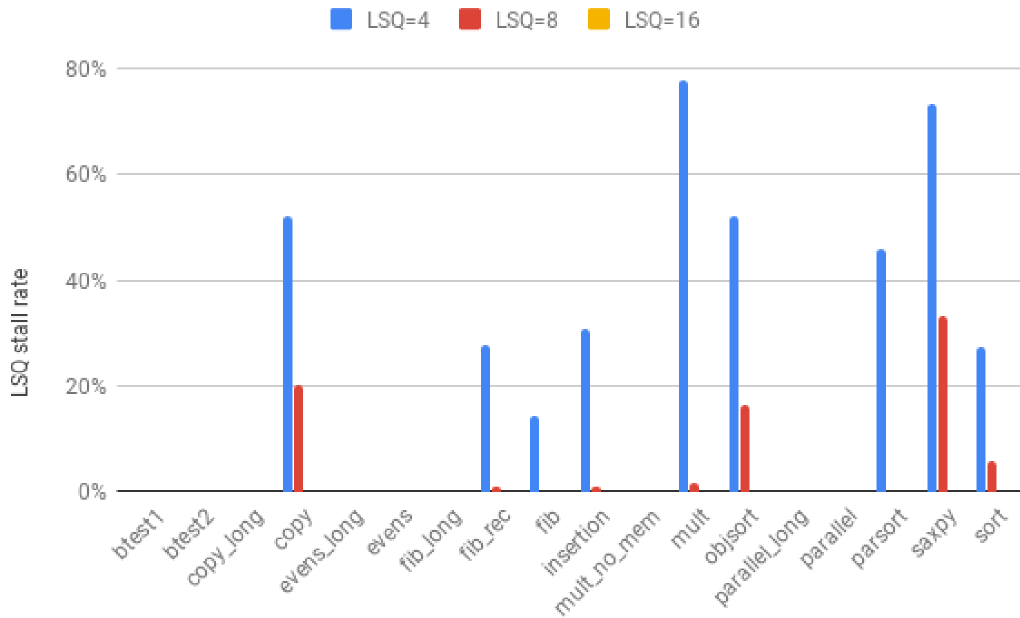

Figure 9: LSQ Stall Rate vs. LSQ Size

Based on the result, we found when LSQ size equals to 1/8 ROB size, there are a lot of stalls caused by LSQ. When LSQ size equals to 1/2 ROB size, most of the test cases

don't fully utilize the LSQ, and there is no boost to the CPI. So we decided to let LSQ size be 1/4 of the ROB size, which matches the result in the lecture slides.

3.4.2 Non-Blocking D-cache

The size of the Dcache is limited to 256 Bytes, with the block size being 8 Bytes. There are only 32 entries in total, so the number of ways is very restricted. We tested our implementation with the number of ways being 2, 4 and 8. The result is shown in Figure 10.

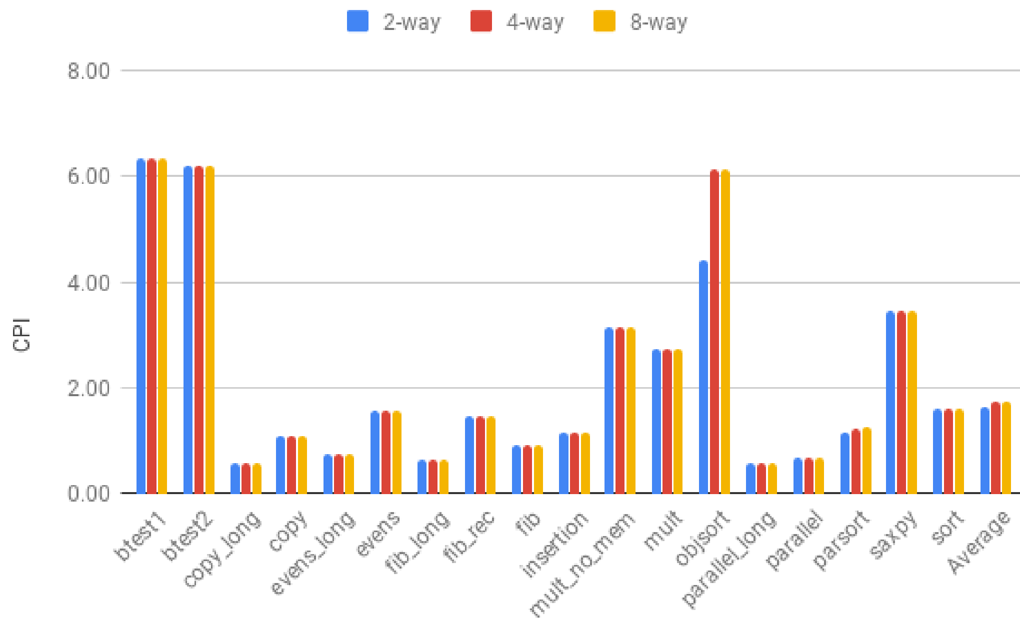

Figure 10: CPI vs. Different Number of Ways for Dcache

The figure shows that for most of the test cases, the number of ways does not affect the CPI a lot, but for some cases like *objsort* and parsort, the CPI increment from 2-way to 4-way can be significant. 8-way implementation shows very similar result with 4-way. Another disadvantage of having more ways is that we need to check more ways every time we try to find a piece of data, which may possibly increase the cycle time. The 2-way Dcache has the best overall performance, so we stuck with 2-way.

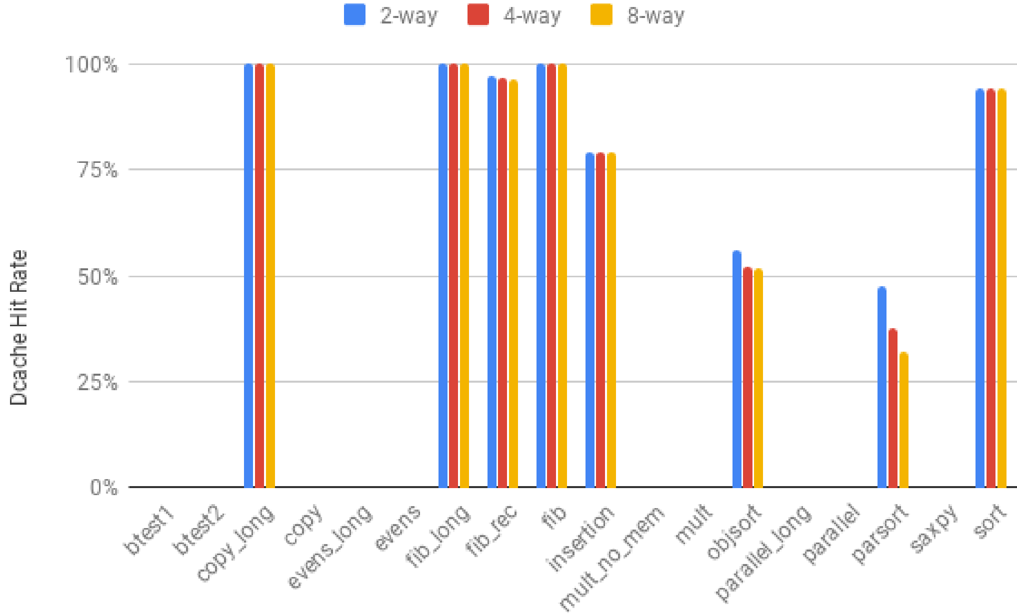

Figure 11: Dcache Hit Rate vs. Different Number of Ways

Figure 11 shows the Dcache hit rate with respect to different number of ways. It matches with the CPI result, because the higher the hit rate, the lower the CPI will be. This figure is another evidence that the 2-way design has the best performance.

3.5 Common Data Bus & Common Data Bus Buffer

The size of CDB Buffer is equal to that of ROB. Since every instruction in-flight is saved in ROB, CDB Buffer is rarely full. In the case that CDB Buffer is full, functional unit has to stall.

3.6 Fetch Technique

3.6.1 Tournament Branch Predictor & Branch Target Buffer

Our analysis about branch prediction is composed of two parts: direction hit rate analysis and target address hit rate analysis. First comes direction hit rate analysis.

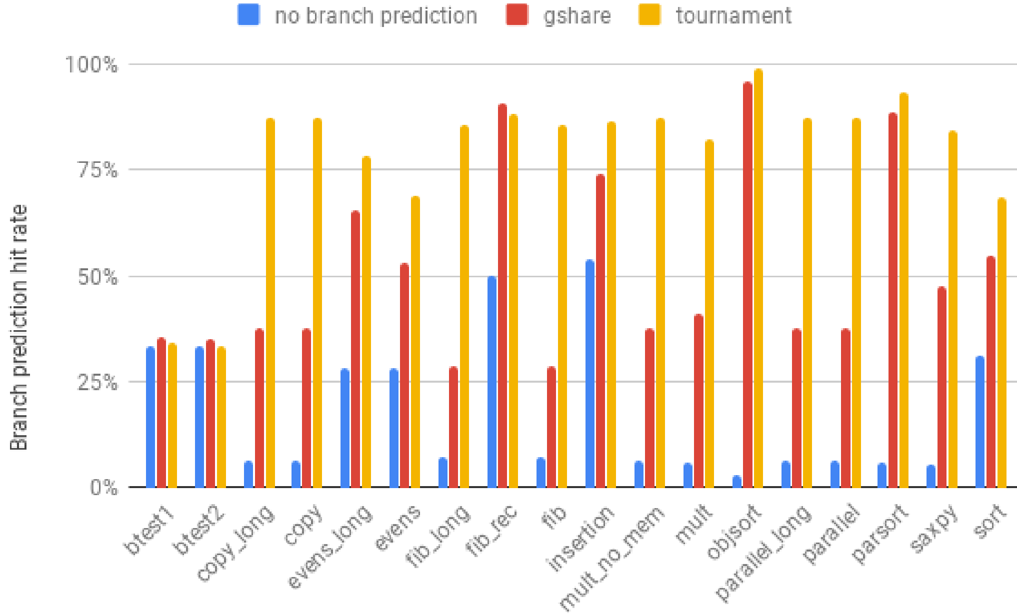

Figure 12: Branch Direction Hit Rate with No Prediction/Gshare/Tournament

Gshare predictor and tournament predictor increase direction hit rate sharply, which means direction prediction is very necessary for us to improve the performance of processor. For most test programs, tournament predictor has a higher direction hit rate than Gshare predictor. Therefore, we decide to use tournament predictor in our design.

Then we analyzed the influence of BTB size on branch prediction hit rate and ultimate performance. The final branch prediction hit rate depends on BTB hit rate greatly. We found that branch prediction hit rate in *ojbsort* is very low. Then we alternated the size of BTB and Figure 13 shows the result. (The size of BTB is originally 16.)

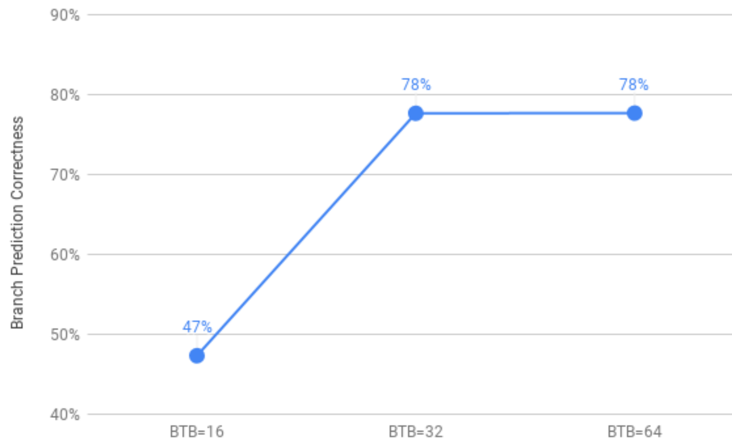

Figure 13: *Ojbsort* Branch Prediction Correctness vs. Different Sizes for BTB

We concluded that there are many branch instructions in *ojbsort* program so that the size 16 isn't sufficient for BTB to store all the valid branch target addresses. Therefore, we tried to increase its size to 32 or 64 and tested the CPI when the processor runs all of the programs. Figure 14 shows the result.

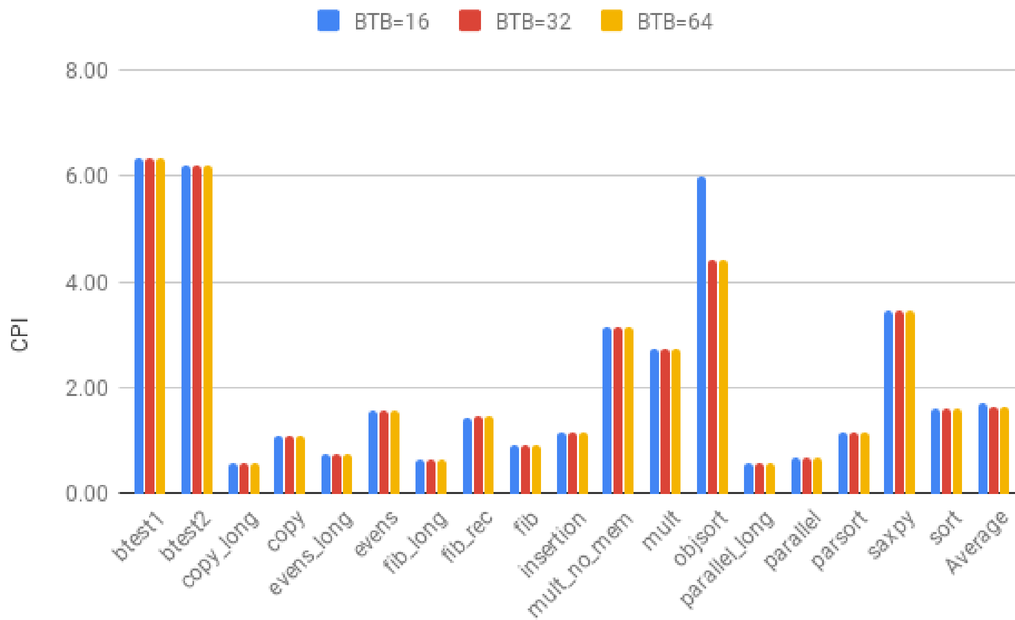

Figure 14: CPI vs. Different Sizes for BTB

When the size of BTB increases to 32 or 64, CPI of *ojbsort* program decreases drastically. The change doesn't affect the performance of processor when running other test programs, either. Therefore, we decided to set the size of BTB to 32 in our design.

3.6.2 Return Address Stack

Return address stack works specially for return instructions. We found that CPI of *ojbsort* program is large in our design, so we decided to add return address stack to our processor. Figure 15 shows the result of analysis of function of RAS.

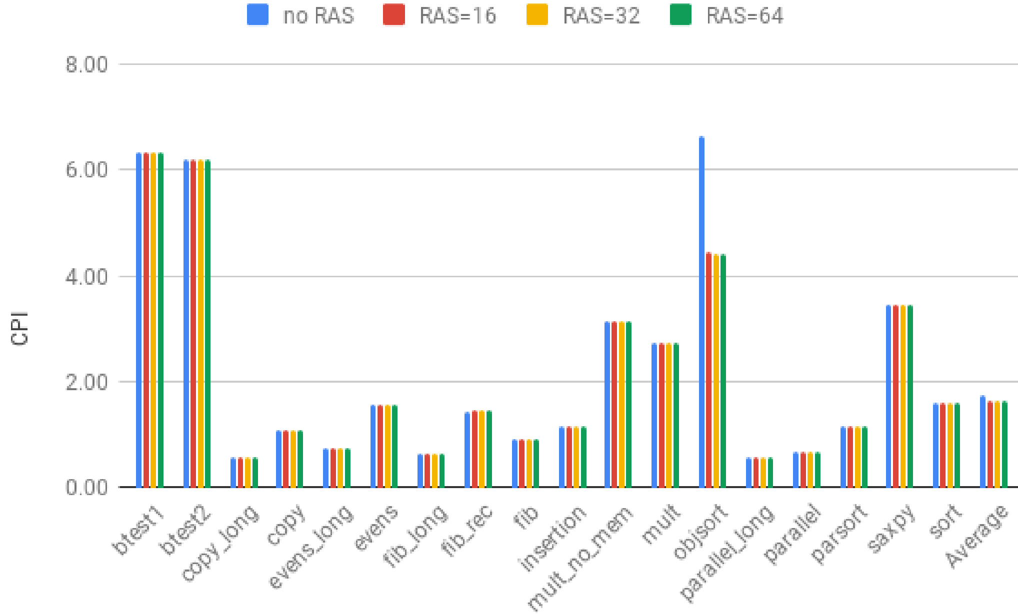

Figure 15: CPI vs. Different Sizes for RAS

It's apparent that CPI of *ojbsort* program decreases sharply with RAS compared with that without RAS, and the size of RAS has little effect on performance of the processor. Therefore, we decided to add RAS with size 64 to our processor.

3.6.3 Instruction Buffer

We simply fixed the Instruction Buffer size twice as large as ROB because it is large enough to avoid the fetch stage be bottle-necked by the size of instruction buffer. Making the size too large is meaningless because it causes buffer to fetch a lot of useless instructions.

3.7 Instruction Prefetch

Prefetch upperbound refers to how many lines of instructions (one line of instruction contains 2 instructions) are prefetched ahead of current PC. Since we decide to make use of instruction prefetching in our design, we wanted to know the effect of *prefetch upperbound* on Icache hit rate and CPI (pre0 means no instructions are prefetched, which is equivalent to no prefetch). Figure 16 and Figure 17 show the results respectively.

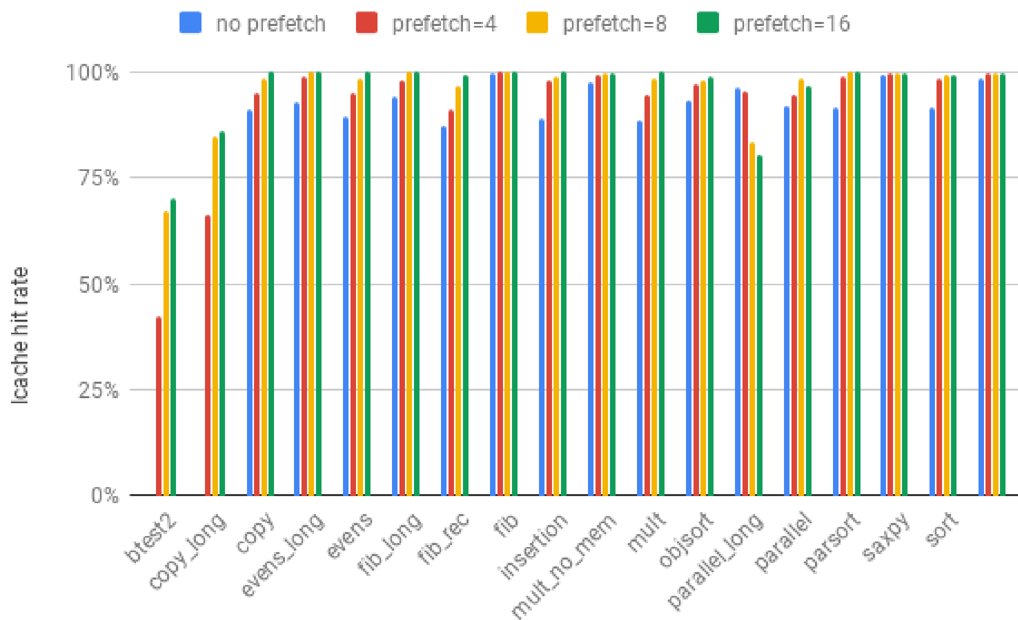

Figure 16: Icache Hit Rate vs. Different Prefetch Upperbound

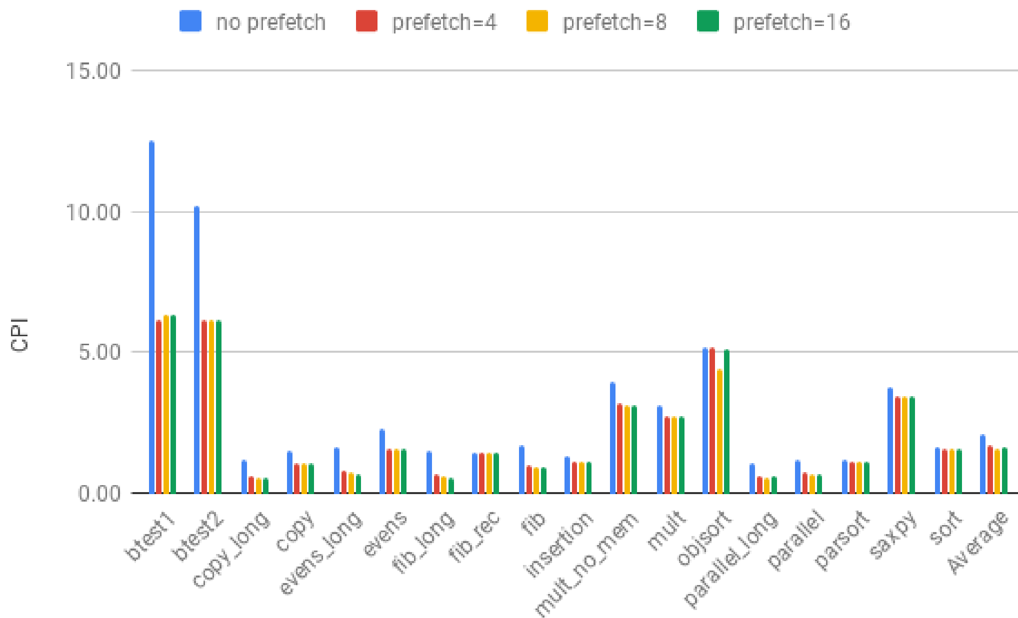

Figure 17: CPI vs. Different Prefetch Upperbound

We concluded from the data that instruction prefetching can boost Icache hit rate and decrease CPI for all of the programs. Among 3 different *prefetch upperbounds*, we found

CPI is the lowest when it's 8. Therefore, we set *prefetch upperbound* to be 8 in our design.

3.8 Victim Cache

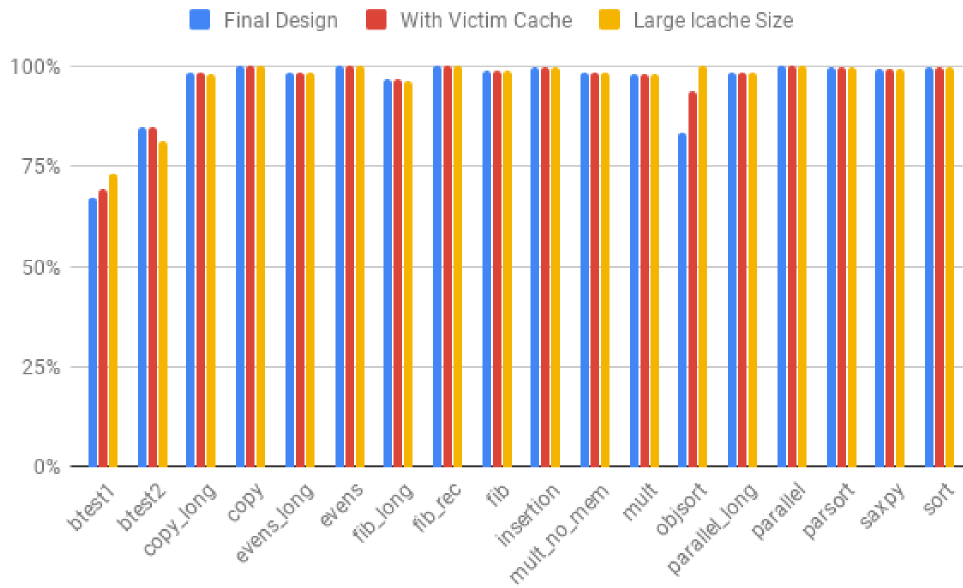

Figure 18: CPI for Different Design

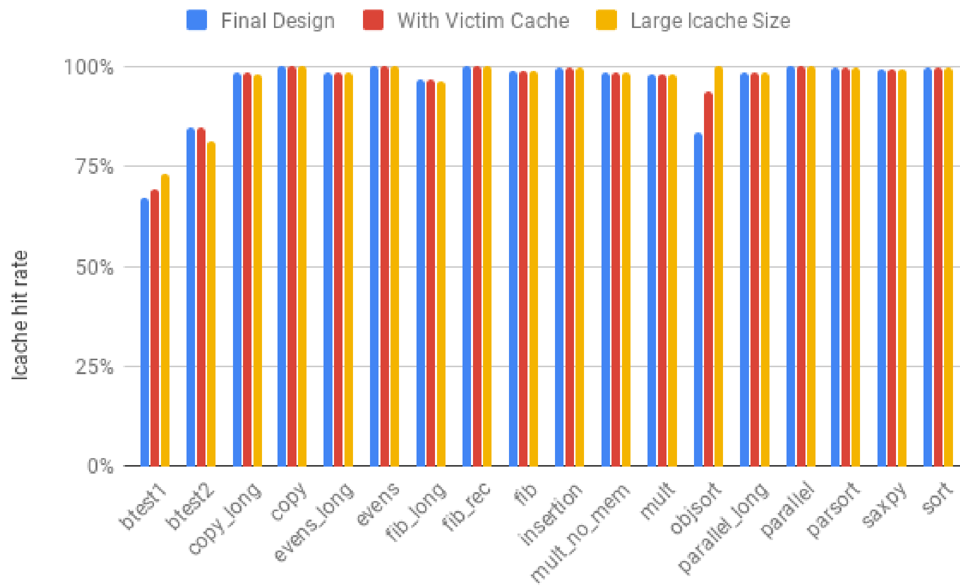

Figure 19: Icache Hit Rate for Different Design

We implemented victim cache with 4 entries and it is attached to the Icache. Here are the CPI and Icache hit rate for with and without victim cache, and we also collected the data for very large Icache, which has 1024 entries.

Victim cache does help boost our Icache hit rate, but the CPI for *ojbsort* goes down. After further investigation, we found that the branch prediction hit rate goes down after we applied victim cache rate to it.

Table 2: Branch Prediction Hit Rate for *objsort* under Different Icache Design

	Final Design	With Victim Cache	Large Icache
Branch Prediction Hit Rate	78%	46%	45%

We found that the branch prediction hit rate goes down for *objsort*, while the direction prediction didn't change much. The same situation appears when we used very large size of Icache. We have a possible explanation for this situation. As the Icache is larger, the instruction is less likely to be kicked out from the cache. The instruction may be fetched directly from the cache for the second time before the branch target has been resolved, which causes additional miss branch prediction. As the victim cache hurts the performance, we decided not to use it in our final submission.

3.9 Critical Path Reduction

In this project, we eliminated 2 critical path to optimize our clock period.

In our original design, the store queue sends data and address to the memory during the same cycle when this instruction is retired from ROB. We decided to add one cycle delay after the instruction is retired from ROB, thus reducing the clock period from 9.2 to 8.2.

The second path appears from the output of RS to the first stage of multiplier. We added a input flip flop before the multiplier, which reduced the clock period from 8.2 to 7.7.

After these modification, a lot of paths in our design have almost the same delay, which is ideal.

4 Results

Our design achieves 7.7ns clock period with following CPI for each test bench .

Table 3: Final Result

Test Bench	CPI	Clock Period	Num of Inst	Time (ns)
btest1	6.33	7.7	229	11161
btest2	6.18	7.7	455	21651
copy_long	0.58	7.7	590	2634
copy	1.08	7.7	130	1081
evens_long	0.73	7.7	318	1787
evens	1.56	7.7	82	984
fib_long	0.63	7.7	627	3041
fib_rec	1.47	7.7	12942	146490
fib	0.93	7.7	147	1052
insertion	1.15	7.7	598	5295
mult_no_mem	3.16	7.7	278	6764
mult	2.74	7.7	325	6856
objsort	4.43	7.7	19704	672123
parallel_long	0.56	7.7	910	3923
parallel	0.69	7.7	194	1030
parsort	1.14	7.7	11106	97488
saxpy	3.47	7.7	185	4943
sort	1.61	7.7	1349	16723
Average	1.62			

5 Group Contribution

- Yichen Yang (20%) - ROB, RS, LQ, Fetch, Decode, RAS, Free List, Pipeline, Debugging.
- Yuhan Chen (20%) - ROB, RS, Dcache, Fetch, Ibuffer, Pipeline, Debugging.
- Zixuan Li (20%) - ROB, Prefetch, Dcache, Execute, Icache, Victim Cache, Map Table, Arch Map, Pipeline, Debugging.
- Haiyang Jiang (20%) - BTB, RAS, Prefetch, Victim Cache, Free List, Tournament Branch DIRP, Pipeline, Debugging.
- Tian Pang (20%) - ROB, SQ, Visual Debugger, Script, Pipeline, Debugging.

6 Conclusion

We have successfully implemented a 2-way superscalar out-of-order processor based on R10K structure and Alpha64 ISA. This practical project gives us fundamental experience of designing a modern processor by ourselves with SystemVerilog. The **Design and Components** section has already reflected our deep understanding of the modules that a modern processor has. By implementing those modules and finally integrating them

together, we have developed our knowledge on computer architecture from top level all the way to specific details. Our analysis and thinking on the design methods and choices have been shown in the **Analysis** section. Dealing with such an open-ended project, we have strengthened our critical thinking skills. The final achievements of our 10-week hard work have been listed in the **Results** section in terms of CPI and optimal clock period. After this project, not only the technical knowledge and proficiency, we have also learned how to do efficient work collaboratively, organize the highly complicated project and balance between optimal results and time consumption. In the end, the experience of this project has provided the necessary knowledge, capability and chances so that we can explore the field of computer architecture further in the future.

7 Future Improvements

Given the focus on the 2-way superscalar, we should have implemented Early Branch Resolution for better performance. We can implement the speculatively updating branch DIRP. Besides, a speculative Load-Store Queue is worth trying if more time is offered. Also, design with the other number of superscalar ways needs comparing with our current design and implementing multi-core processor will be valuable practice in the future.

Acknowledgments

We thank Mr. Jon Beaumont, Mr. Vivek Venkatraman, Mr. Zhiheng Yue and Mr. Carson Boden for their help and support. We really enjoy the time with you.