EECS 470 F16 Final Report
MIPS R10K 3-Way Superscalar Out-of-Order Processor

Group abc

Yuchen Fan fanyuch@umich.edu
Xiangxuan Ge gxx@umich.edu
Ruohan Luo luoruoh@umich.edu
Yuxiang Mu muyx@umich.edu

ABSTRACT

This report summarizes our EECS 470 design project: a 3-way superscalar out-of-order processor based on MIPS R10K scheme. To improve the processor's performance, we add some advanced features to this processor, such as early branch recovery, next-line prefetching and store-to-load forwarding in LSQ. Finally, the test cases can be passed with low CPI as well as low execution time.

KEYWORDS

MIPS R10K, superscalar, out of order, GUI, early branch recovery

1. INTRODUCTION

Nowadays, the trend in computer architecture fields is to pursue higher performance while keeping the power consumption low. For the course project of EECS470, we designed a 3-way pipelined out-of-order MIPS R10K processor based on the Alpha 64-bit ISA. It contains 6 stages for an instruction to implement: instruction read, fetch, dispatch, issue/execute, complete and retire. Besides the superscalar, we also implemented early branch resolution, cache enhancement and branch predictor to optimize the performance with a decent clock period. The overall architecture is shown in Figure 1.

The following parts gives the detailed explanation of our design of each module, advanced features and the analysis methods, and it is organized as follows. Section 2 gives mapping from every module to the verilog file name. Section 3 summarizes the features of our processor. Section 4 gives the details of the module designs. Section 5 summarizes the performance of the processor. Section 6 describes our GUI debugger. Section 7 analyzes the impact of each advanced features. Section 8 shows our work distribution.

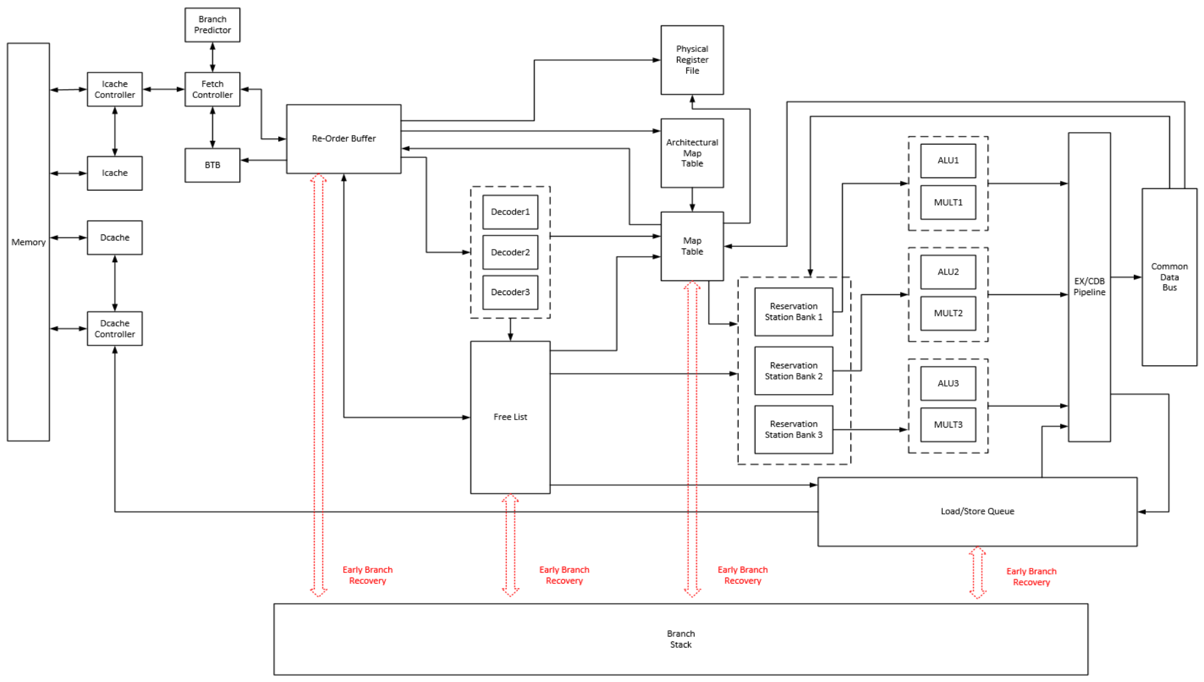

Figure 1: Overall architecture of our processor

2. MODULE NAME MAPPING

The following table illustrates the mapping of our modules to the verilog files. Our final integration of all modules is completed in “part_cmfr.v”. The final testbench is “part_cmfr_testbench.v”.

Module Name	Verilog File
Icache Controller	icache.v
Icache	cachemem.v
Architectural Map Table	amt.v
Branch Predictor	bpred.v
Branch Stack	bstack.v
Branch Target Buffer	btb.v
Common Data Bus	cdb.v
Dcache Controller	dcache_ctrler.v
ALU/MULT	ex_stage.v
Module Name	Verilog File
Map Table	mt.v
Physical Register File	prf.v
Priority Selector	ps16.v
Re-Order Buffer	rob.v
Reservation Station	rs.v / rs_base.v
Free List	freelist.v
Decoder	id_stage.v
Fetch Controller	if_stage.v
Load/Store Queue	lsq.v

Table 1: Module name/Verilog file mapping table

3. FEATURE SUMMARY

The features of our processor are listed below:

1. 3-way superscalar R10K design
2. Early recovery from mispredicted branches
3. Graphical debugging tool
4. Memory System Enhancements
    a. Next-line prefetching
    b. 2-way associative
    c. Victim cache
    d. 2-ported cache
5. Store-to-load forwarding in LSQ

4. MODULE DESIGN

4.1 Re-Order Buffer

Our re-order buffer (ROB) has 64 entries, which can accept at most three instructions from IF stage and 3 instructions for dispatch. According to our design, each entry mainly contains instruction IR, PC, destination reg index, tag and old tag as well as branch information for recovery such as mispredict and target PC.

The ROB is implemented as a circular buffer with head and tail pointers. While ROB is not full, it allows up to three instructions each cycle and place basic information in empty entries in order. Otherwise it will send stall signal to fetch stage to stop PC counter. The instructions in ROB must dispatch in order with tail incrementing. When instructions gain permissions to dispatch from reservation station (RS) and free list (FL), ROB will send corresponding instruction IR to decoder for further renaming work and finally pass the necessary information of dispatching instructions to RS and LSQ. Meanwhile, the tail will move to the last dispatch entry. Similarly, retirement of instructions must be in order as well with head incrementing. Once an instruction is completed, ROB will mark its corresponding entry as completed. When the ROB entry that head points to suggests it has already completed, this instruction will be retired and this entry will be flushed with head pointing to next entry. Meanwhile ROB will send physical and architectural tags to update architectural map table (AMT) and FL. When a branch instruction is mispredicted and this branch is not in fast recovery process, the ROB will send branch information to IF stage for recovery.

4.2 Reservation Station

We divide our reservation station (RS) into 3 banks and each reservation station bank has 16 entries. In the dispatch stage, ROB sends the information of three instructions to each bank. Then we use the 16 to 1 priority selector to select the entry of each bank to store the information. We will broadcast CDB tag with the physical register number of the operands every time an instruction is completed. Once they are matched, we will set the ready bit of the operand to 1. If the ready bits of all operands in one entry are set to 1, the entry will wait to be selected to issue. We use two 16 to 1 priority selectors for MULT operations and ALU operations separately. We decide to issue the multiplication operation earlier than ALU operations since we would like to eliminate the delay of the dispatch of the instructions dependent on the multiplication operations which will take eight cycles to finish. The entry will be free once it is issued. Each entry will also store the b-mask bit for branch recovery, the entry with certain b-mask bit will be free when a branch operation needs to be recovered.

Though this structure minimizes the hardware complexity and the potential dependency hazard, the performance will be affected when one of the reservation bank is full since we need to stall the fetch stage. Fortunately, this situation is uncommon.

4.3 Map Table

The map table (MT) has 32 entries. It is used to rename the architectural registers to the updated physical registers, which are provided by free list. It also stores the information of ready states of these physical registers. After receiving the decoding information, it will output the physical register information to reservation station and output the old tag information to ROB. When CDB is broadcasting, MT will receive the information and set the corresponding ready bit to 1. Every time a branch is mispredicted, MT will be replaced with the current architectural map table (AMT).

4.4 Architectural Map Table

The architectural map table (AMT) has 32 entries. It is used to store the retirement information of the mapping from architectural registers to physical registers. Every time an instruction is retired, AMT will receive the tag and destination register information from ROB to update the corresponding physical register information. Every time a branch is mispredicted, AMT will be used to recover the state of MT.

4.5 Free List

The free list (FL) is a circular FIFO buffer. In the dispatch stage, at most 3 free physical registers will be dispatched for destination registers and the head pointer will increase. In the retire stage, at most 3 retire physical registers will return to free list and the tail pointer will increase. Since the size of physical register file is 96 and the size of map table is 32, we set the size of free list to be 64, which makes sure the free physical registers available in the circular FIFO from head to tail will never be used out.

When the branch recover is enabled, the recover head will overwrite the current head, which backs the head pointer up and restores all free physical registers that have been allocated after the branch operation which needs to be recovered.

4.6 Common Data Bus

The common data bus (CDB) is 3-way, which can support our 3-way superscalar execution. It is used to broadcast resulting data from the function units, including 3 MULT modules, 3 ALU modules, 1 Load Queue (LQ) and 1 Store Queue (SQ). Among these 8 function units, the 3 MULT modules have the highest priority. This is because that the MULT instruction takes 8 cycles to finish, which means the finished MULT instructions are most likely to be the oldest instructions among the 8 instructions. The LQ module and SQ module have the second and third highest priority, respectively, because the latency of memory access is long. Finally, the 3 ALU modules have the lowest priority because ALU instructions only need 1 cycle to calculate.

If there are more than 3 instructions that can be completed, CDB can choose 3 instructions with highest priority to complete and broadcast. For other ready but not completed instructions, we need to keep their current resulting data in their function units. Therefore, these function units should be invalidated until these instructions are completed and broadcast in future cycles.

4.7 Icache

Our instruction cache is a 256-byte, 2-ported, non-blocking, write-through, 2-way associative cache. Since we will fetch at most 3 instructions every cycle, original 1-ported cache will cause a huge delay. Therefore, we decided to increase the number of read ports to 2 so that we can read at most 4 instructions every cycle. In the fetch stage, we will choose 3 instructions from 2 data reads from Icache. We implement a Miss Status Holding Register Buffer (MSHR), which stores all the outstanding misses and sends one by one each cycle at head. In order to eliminate the structural hazard of the MSHR buffer, we set the size to 16. This design greatly improves our performance by hiding the memory latency.

The next line prefetching is also implemented in the Icache controller. Each cycle, we will first check if the request of PC misses. We will put it into the MSHR buffer if it is a miss and it doesn't exist in the MSHR. If the request of PC hits or it is already in the MSHR, we will check the request of PC+8. If it is still a hit or already in the MSHR, we will send the next line of PC+8 to the MSHR buffer, which means we have prefetched the PC+16 into the MSHR.

The Icache also cooperates with a fully-associative victim cache, which contains 4 64-bit blocks. The lines which are evicted from Icache due to collision are stored into the victim cache with LRU replacement policy. If the instruction misses in Icache but hits in victim cache, the corresponding hit line will be moved back to Icache again.

4.8 Dcache

Our Dcache is a 256-byte, 2-way set associative, 64-bit line non-blocking, 2-read-port and 2-write-port write-through cache. It is composed of two parts: Dcache controller and Dcache memory.

DCache Controller has a 16-entry Miss Status Holding Register Buffer (MSHR), which stores all the missed load and store instructions and sends one by one each cycle at head. This design greatly improves our performance. It is full when there are less than 3 empty slots available. Besides, it has 2 read ports and 1 write ports, which can accept 2 load branch instructions and 1 write instructions each cycle.

The two write ports are used for a memory read miss and LSQ write hit separately. We use the LRU replacement policy to make the cache work in an efficient way. Since there are only two ways in one block, we use one bit to record which way is least recently used, which will be replaced when either a memory read miss or a LSQ write hit happens.

4.9 Function Units

Our function units have 3 ALU function units and 3 multiplier function units. Each RS bank corresponds to 1 ALU and 1 multiplier. This design simplifies our design and reduces our clock period. The ALUs are used to calculate normal ALU calculation, branch target PC calculation, and load/store address calculation. The multipliers are used to calculate multiplication only. We use an 8-stage pipelined multiplier to reduce the clock period. To make the auxiliary information go along with MULT instructions, we add another 8-stage pipelined registers inside the multiplier.

4.10 Branch Target Buffer

Our branch target buffer (BTB) has 8 entries, which stores the PC and target address of branch instruction if this branch is ever taken. Every time the result of local predictor is taken, the BTB will search its entries to find if there is a hit with branch PC. If there is a hit, the BTB will send corresponding target address to IF stage as next PC. Otherwise we just predict it to be not taken since we cannot obtain target address until EX stage. BTB also helps prefetching process in Icache. If the prefetched PC is a hit with one of the branch PC in BTB, Icache will prefetch target address instead of next-line block.

4.11 Branch Predictor

Our branch predictor (BP) is correlated, which allows us to maintain prediction per (PC, history). The branch history table (BHT) has 3 bits per PC, which records the branch outcomes of last 3 branch instructions. The entry size is 256, which means we use the [9:2] bits of PC to index the BHT. The pattern history table (PHT) has 2 bits, which contains 4 states: 11 as strong taken, 10 as weak taken, 01 as weak not taken and 00 as strong taken. Hence, the most significant bit of these 2 bits can be used as the branch prediction result. (1 as predict taken, 0 as predict not taken) The prediction result will be output to BTB. Every time a branch instruction is resolved, the actual branch result will be passed to BP to update the BHT and PHT. After several tests and comparisons, we decided to set the initial PHT state as 10, which can mostly reduce the mispredict rate.

4.12 Load/Store Queue

Our Load/Store Queue (LSQ) is used to deal with memory instructions, which can communicate with Dcache controller and enable store to load forwarding. It is implemented as two 16-entry separate circular buffers: one for load instructions and one for store instructions. They are full when there are less than 3 empty slots available. Each buffer is similar to the ROB but only has load or store instruction in it. According to the ROB size, we set the size of load queue and store queue to 16 to balance between CPI and clock period.

When memory instructions are dispatched, new entries are allocated in load queue and/or store queue. These memory instructions will also enter RS and get issued to EX-stage to calculate their addresses. When addresses are calculated, LSQ will get the result from EX/CDB pipeline and update load and store addresses. They will not enter CDB to broadcast. Then LSQ will check if there is any load-store pair that data can be forwarded based on their relative order and addresses. To speed up, when the addresses of loads are resolved, LSQ will send load instruction to DCache Controller immediately, before all previous stores' addresses are resolved. If LSQ figures out that this load should be forwarded from another store later, the data loaded from Dcache controller will be discarded and new data from store will be saved. This should make load complete earlier but occupy the memory bus more. Loads are completed only at load queue head when all of previous stores' addresses are resolved and its data is either loaded from memory or from forwarding. Stores are completed only at store queue head when it meets the head of ROB. When completed, store will send its data to Dcache controller to store into memory.

4.13 Physical Register File

The Physical Register File (PRF) has 96 entries, which equals to the sum of ROB entries and map table entries. It is used to store the actual numerical data of each physical register.

4.14 Branch Stack

Our 8-entry Branch Stack (Bstack) is implemented for fast branch resolution. When processor dispatches branch instructions, branch stack will allocate an entry to keep copy of current ROB/LQ/SQ tail, map table, free list head as well as other checkpoint information for possible recovery and change b-mask bits to keep track of pending branches. When branch instruction is completed, branch stack will free the entry and give out the checkpoint information for recovery depending on if it is mispredicted or not. Notice that the branch stack might be full, and we choose to continue dispatch branches and solve them in retire stage instead of stalling branch instructions in dispatch stage.

5. PERFORMANCE

5.1 Performance Analysis

According to Iron Law, the processor performance is related to "Time/Program", which is "Code Size * CPI * Clock Period". Since the code size of each test case is fixed, hence we want to minimize "CPI * Clock Period" of each test case, which is "Time/Instruction".

After synthesis, we find that the smallest clock period we can achieve is 9.6ns. Since the equation to calculate memory latency in cycles is (100/Clock Period + 0.49999), hence the clock period of 9.6ns corresponds to the memory latency of 11 cycles. To achieve the memory latency of 9 cycles and 10 cycles, the corresponding smallest clock period is shown below.

Clock Period (ns)	Memory Latency (cycles)
9.6	11
10	10
11.2	9

Table 2: Different clock periods and mem latency cycles

In order to find the best clock period to achieve the minimum "CPI * Clock Period" for each test case, we will discuss these three conditions. The test result is shown below.

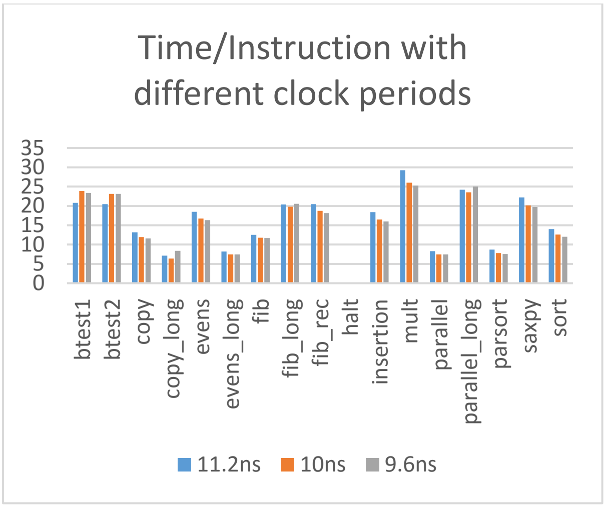

Figure 2: Time/Instruction with different clock periods

From the above figure, we conclude that generally, setting the clock period as 9.6ns will achieve the minimum "Time/Instruction" for most test cases. For some test cases with more branch instructions, setting the clock period to 10ns or 11.2 ns will be better because of the reduction of memory latency in cycles. The average Time/Instruction for these 3 conditions is shown below.

Clock Period (ns)	Average Time/Instruction (ns)
9.6	15.74
10	16.31
11.2	19.09

Table 3: Average time/instruction with different periods

As shown above, the clock period of 9.6ns will achive the lowest average time/instruction. Therefore, we decide to use 9.6ns as the clock period of our processor.

5.2 Performance Result

As we use 9.6ns as the clock period, the resulting CPI for test cases is shown below. The average CPI for these test cases is 1.64.

Test Case	CPI	Test Case	CPI
btest1	2.43	halt	Inf
btest2	2.41	insertion	1.67
copy	1.21	mult	2.63
copy_long	0.87	parallel	0.78
evens	1.7	parallel_long	2.61
evens_long	0.78	par_sort	0.79
fib	1.22	saxpy	2.05
fib_long	2.14	sort	1.25
fib_rec	1.89		

Table 4: Result of CPI for different test cases

6. GRAPHIC DEBUGGING TOOL

We use Python to build a GUI debugger to simplify our debugging process. In our debugger, all the signals can be monitored in arbitrary cycles.

Our GUI debugger has 11 pages and each page displays various signals in different modules. At the head of each page, we display all the instructions and their NPC in each stage in our pipelined CPU. Since our processor is a 3-way superscalar processor, we display 3 instructions at each stage. Clock cycle number is also shown at the head of each page together with three buttons: previous clock, next clock, and jump clock, which can navigate between arbitrary clock cycles. Besides, we can monitor the status of all physical registers at the right side of each page. Memory data is also shown in a separate page.

The biggest benefit is that we can add any signals in any modules to the visual debugger easily by adding them in "config" file. Figure 3 is a snapshot of our visual debugger.

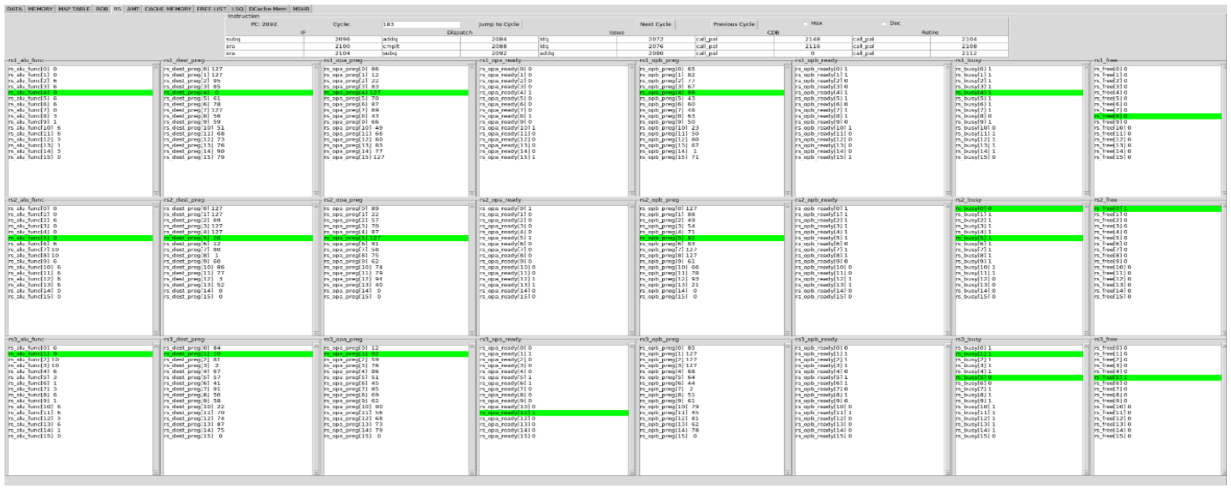

Figure 3: Snapshot of our GUI debugger

7. ADVANCED ANALYSIS

7.1 Branch Predictor Initial State

As mentioned before, the pattern history table (PHT) has 2 bits to represent 4 states: 11 as strong taken, 10 as weak taken, 01 as weak not taken and 00 as strong taken. With the different initial state of PHT, the CPI of each test case can be different. The testing result is shown below.

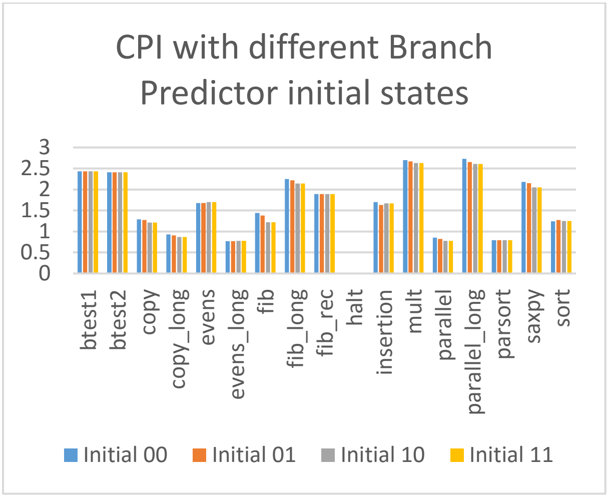

Figure 4: CPI with different BP initial states

From the above figure, we conclude that generally, setting the initial state of PHT as 10 (weak taken) will achieve the lowest average CPI for all general test cases. This is because in most test cases, the branches are more likely to be taken, rather than no taken. Therefore, we decide to set the initial state of PHT as 10 (weak taken).

7.2 Victim Cache

As mentioned before, we use a victim cache to cooperate with the Icache because a victim cache can help Icache more than Dcache. The following figure illustrates the CPI of running general test cases with/without victim cache.

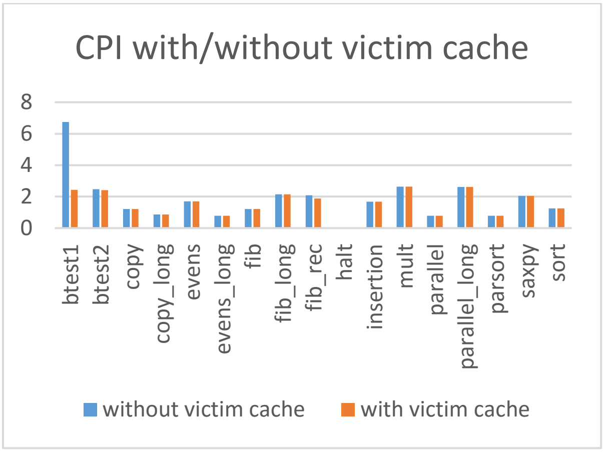

Figure 5: CPI with/without victim cache

From Figure 5, we conclude that using victim cache can notably reduce the CPI in some test cases, such as *btest1*, *btest2* and *fib_rec*. Therefore, we decide to use the victim cache to improve our performance in these test cases.

7.3 Branch Recovery

Applying R10k design means a huge effort on precise state and branch recovery. At the very beginning, we decide to follow traditional way to recover from mispredicted branches. When an instruction is fetched, branch predictor will give out a prediction as well as target address if it is a hit in BTB. The prediction will be stored in ROB and passed through following. Once a branch instruction is finished in the execution stage, processor is able to judge if this branch is mispredicted or not according to the original prediction along with the instruction. If the branch is mispredicted, the processor set mispredict bit to be high and store correct address for recovery in corresponding ROB entry. Every time the head pointer goes to such instructions that are marked mispredicted, the processor will use single cycle to recover. The efforts contain restoring free list tail, copying current architectural map table to map table, flushing ROB/RS/ LSQ, and finally sending target address to IF-stage.

Considering CPI promotion, we decide to add fast branch resolution in addition to regular ways as mentioned above. We establish an 8 entries branch stack to collect information of current status of processor when dispatching a branch instruction, which allow us to recover from misprediction as long as branch finishes execution. Instead of just flushing modules such as ROB, RS and LSQ, we need more complicated efforts such as squashing particular instructions in RS and EX-stage by comparing b-mask reg, restoring LSQ and ROB tails according to the checkpoint in branch stack (Bstack). The storage of checkpoint information in Bstack is a hurt definitely on area and slightly on clock period. However, it should have a notable improvement on CPI, as shown in figure6.

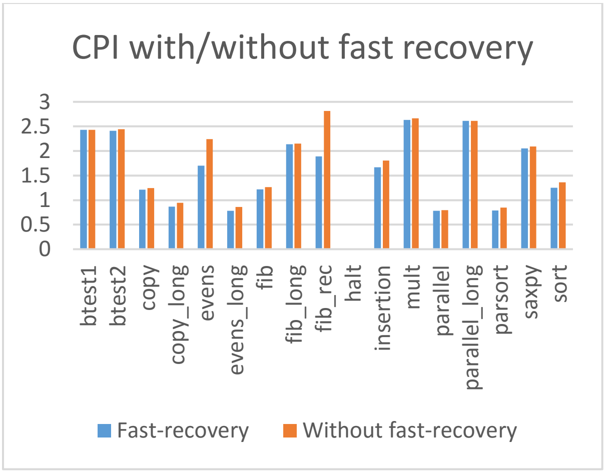

Figure 6: CPI with/without fast recovery

When the Bstack is full, we have an option to rather rewind extra branches through regular ways (in retire stage) or just stall branch instructions in dispatch stage. Research suggests that stalling branch instruction is more efficient in situations that true hazard frequently occurs. As long as we already have copy of mispredict information in ROB, we decide to continue resolving branches that are not in Bstack.

7.4 Cache Associativity

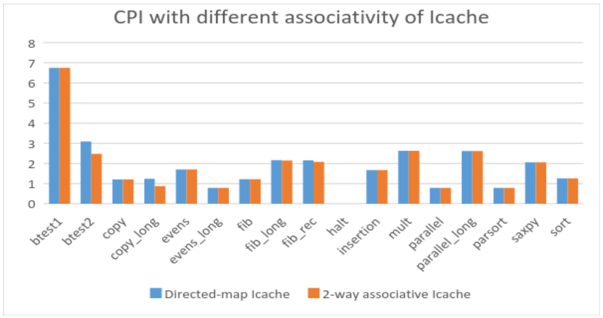

Figure 7: CPI with different associativity of Icache

At the beginning, we use the given direct-mapped instruction cache memory. Considering the block size and cache size of Icache is not too large, we think increasing the cache associativity will increase the overall Icache hit rate and decrease the average CPI. As shown in the above Figure 7, the CPI of some test case like *btest2* and *copy_long* decreases significantly after the enhancement.

7.5 Prefetching

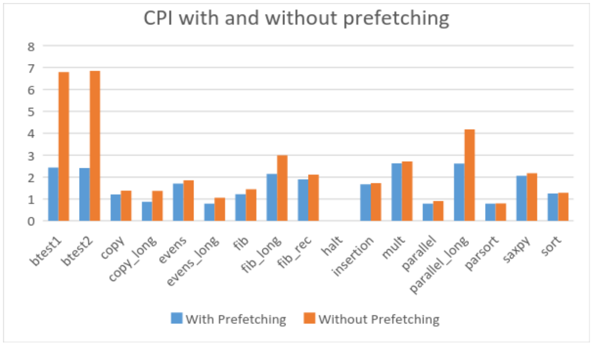

Figure 8: CPI with/without prefetching

Using prefetching can reduce compulsory cache miss greatly and hence give a great boost to the overall performance. As shown in the figure above, *btest1* and *btest2* get a huge improvement because of the prefetching, the CPI is reduced by 200%. The possible reason is that these two test cases contain several blocks with a certain pattern, thus next-line prefetching will increase the hit rate when the first instruction in a block hits.

7.6 Speculative Load to Memory Issue

Instead of sending load instructions to memory and Dcache when all previous stores are ready, we send them in advance. When the addresses of load instructions are ready, LSQ will send load instructions immediately. When loads are forwarded from stores, the data loaded from memory will be discarded. This speculative design is a simpler version of speculative load issue. It issues loads to memory earlier but completes at the same time as before. It can overlap the time needed for loads to grab data from memory and the time waiting for all previous stores to be resolved. It occupies memory bus more, but makes load instructions complete earlier. According to test statistics, there's an average of only 10% loads which will be forwarded from stores. Thus, only 10% of loads whose data loaded from memory are discarded and 90% of loads can benefit from our design and complete earlier. Figure 9 shows the comparison of CPI between these two designs. We can see that our new design has lower CPI especially when the program has more load instructions.

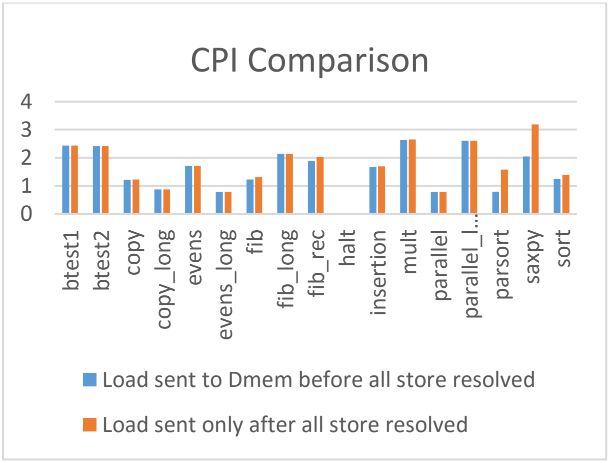

Figure 9: CPI comparison between these two designs

7.7 Store to Load Forwarding

In LSQ, we enable store to load forwarding. This greatly improves the performance. The forwarding rate (portion of load instructions which is forwarded) is shown in Figure 10. We can see that when the program is larger, more forwarding happens. Those programs with 0% load to forwarding rate is because some of them do not have much load or store instructions, or the distance between loads and stores is too big so that they cannot stay in the buffers at the same time.

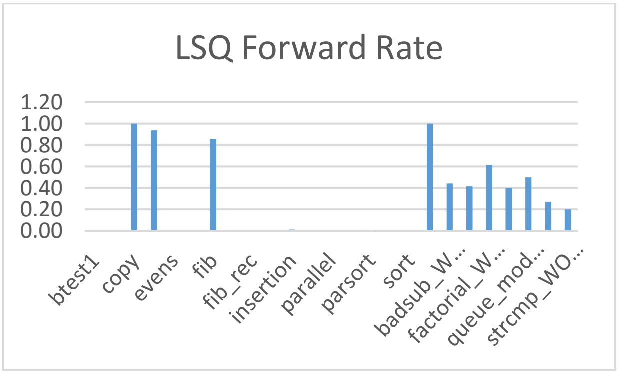

Figure 10: LSQ forwarding rate for different test cases

7.8 Dcache Hit Rate

Figure 11 shows the comparison of data cache hit rate for different programs. We can see that except several test cases with very few load instructions, most programs have good data cache hit rate, varying from nearly 100% to about 20%.

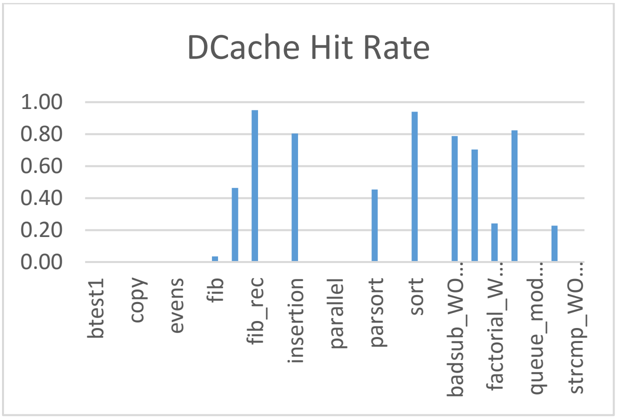

Figure 11: Dcache hit rate for different test cases

7.9 Store Queue Usage

Figure 12 shows the usage of store queue buffer running *factorial_WORKS.s* We can see that in most of the time the buffer has less than 6 instructions, but there are still many situations that the buffer is full. Thus, our store queue size of 16 is reasonable. Notice that we set the store queue to full and stall dispatching when there are less than 3 empty slots. Thus, in the figure below, our store queue is never absolutely "full".

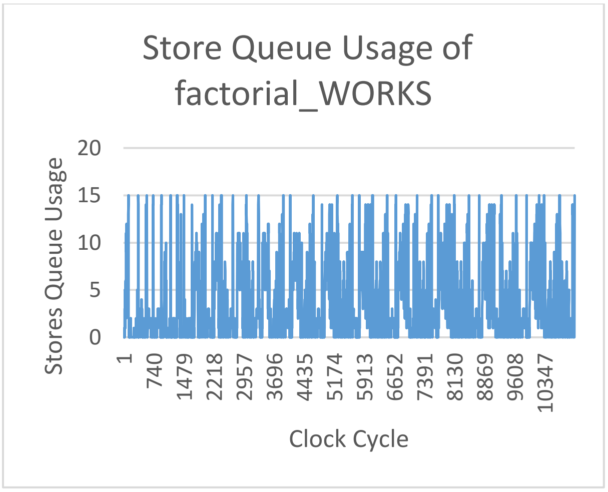

Figure 12: Store Queue usage of factorial_WORKS

7.10 Load Queue Usage

Figure 13 illustrates the usage of load queue buffer running fib_rec.s We can see that in most of the time the buffer has less than 2 instructions, but there are still many situations that the buffer is full. Thus, our load queue size of 16 is reasonable. Notice that we set the load queue to full and stall dispatching when there are less than 3 empty slots. Thus, in the figure below, our load queue is never absolutely "full".

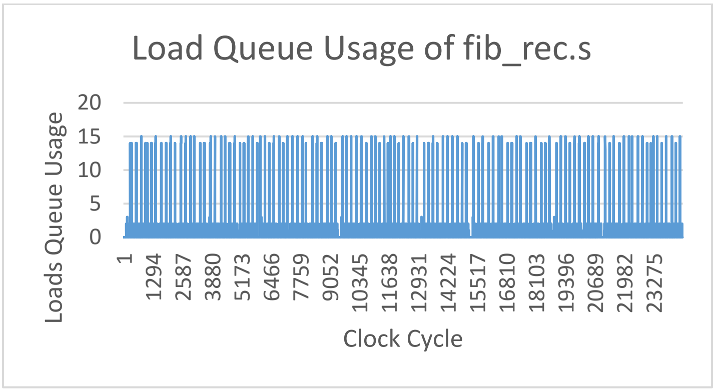

Figure 13: Load Queue usage of fib_rec

8. WORK DISTRIBUTION

Every group member has even contribution to the project.

Yuxiang Mu: Map table, AMT, CDB, Local predictor, Victim cache

Xiangxuan Ge: RS, EX-stage, PRF, LSQ, Dcache, Dcache controller, GUI debugging tool

Yuchen Fan: Icache, Icache controller, Dcache, RS, Free list, IF-stage

Ruohan Luo: ROB, BTB, Bstack, IF-stage, EX-stage

In addition, all group members do system integrating and debugging together.

9. ACKNOWLEDGEMENTS

First, we would like to thank Prof. Thomas Wenisch for his interesting and impressive lectures. Second, we would like to thank our GSIs, Steve Zekany and Marshall Versteeg, for their tremendous help and suggestions on our labs and final project.