EECS470 Computer Architecture Out-of-Order Processor Design Report

Haoyang Zhang, Juechu Dong, Xiangdong Wei, and Chen Huang

Abstract

This is the project report for University of Michigan course EECS470 Computer Architecture. We designed a 3-way scaled, R10K based out-of-order processor with advanced branch predictor, prefetching and non-blocked DCache with system verilog. The issue stage uses FIFO to help reduce clock cycle during reservation station selection.

I. Introduction

Building a processor of our own is the best introduction to computer architecture. It was a hard journey, especially with a time limit of 6 weeks. Our out-of-order processor is not perfect and we haven't realized all our wonderful advanced features we planned when we started, but we still managed to make 3-way scalar, an advanced branch predictor, prefetcher and non-blocking DCache working.

In this report, we will elaborate design choices, implementation and performance of the processor in detail. In part II, we will give a overview of the processor and the design choices we made in this processor. In part III, details about each module implementation is provided. In part IV, we provide some test data on and performance analysis on bottlenecks, potential improvements and reflections.

II. Design Overview

We built an out-of-order, 32-bit processor based on the 3-way scaled R10K microarchitecture, with advanced branch predictor, non-blocking cache design and prefetcher, as shown in Fig. 1. Our processor support 32 bits RV32IM ISA, without fences, division, CSR operations and system calls.

In center is the main pipeline. Pipeline stages represented as blue boxes are separated by yellow pipeline registers. From left to right there is fetch stage, dispatch stage, issue stage, functional units, complete stage and retire stage. Apart from the pipeline, there are map table, physical register, free list, reorder buffer (ROB) and architecture map table from the R10K design to record necessary data during out-of-order processing. On top of that, there are instruction cache (ICache), store queue, data cache (DCache) and memory controller to deal with memory related operations. Finally there is a branch predictor that feeds prediction to fetch stage and reduce precise stage interruption cause by faulty branch direction. Critical specs of our design are listed in TABLE I.

TABLE I: Critical Design Spec

RS	ROB	PR	FUs	ICache	Prefetcher	SQ	DCache	BP
16 entries	32 entries	64 regs	3ALU, 2Load, 1Branch, 2Mult	256 Bytes	12 lines ahead	8 entries	256 Bytes	32 entries

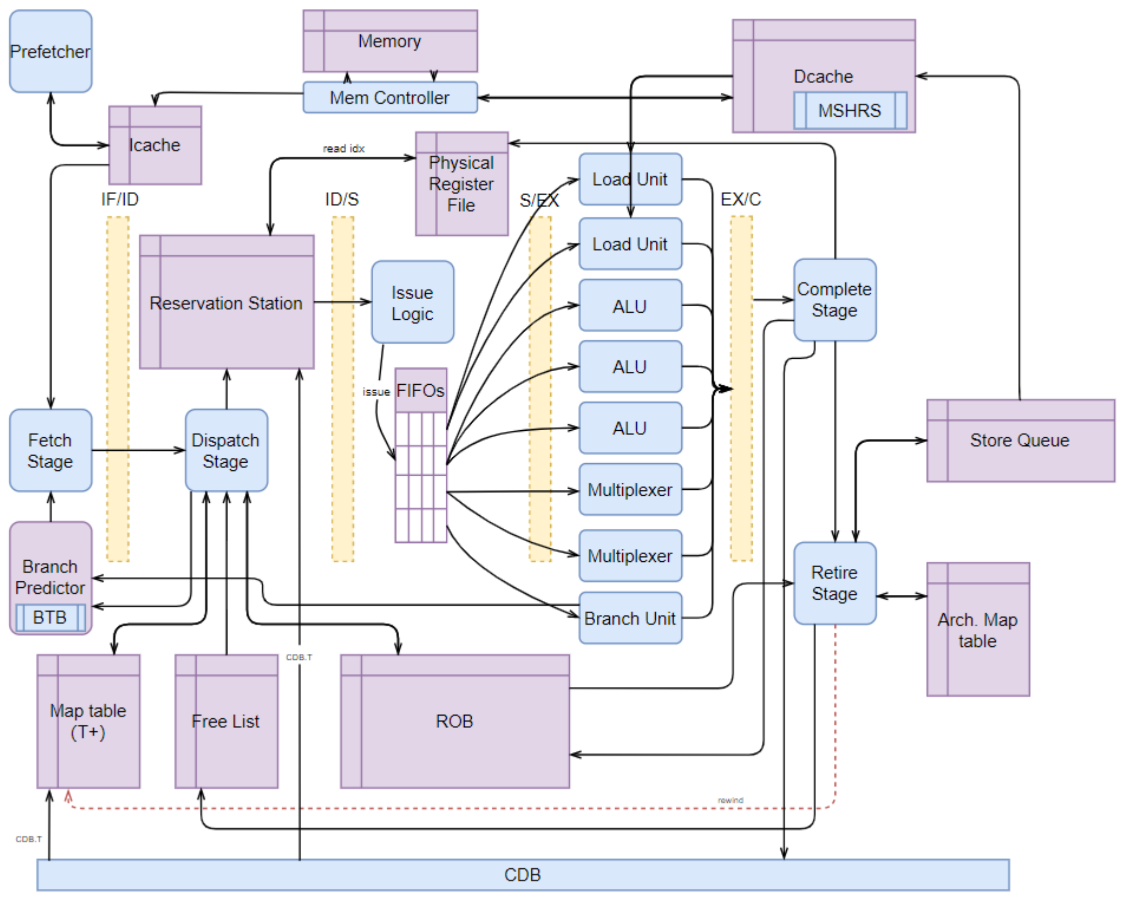


Fig. 1: Top-level Overview of our R10K based Processor

III. Basic Out-of-order Features

A. Fetch Stage

Our fetch stage has an embedded PC counter that records the next three instructions it is going to fetch. It forwards its PC counter to the non-blocking ICache to require for the data. In return, the ICache gives fetch stage the data and a valid signal immediately if it's a cache hit, or otherwise returns a cache miss signal. Most of the time, the fetch stage would increment each of the three entries of its PC counter by 12. However, there are exceptions. When there's a cache miss or the dispatch stage stalls, the fetch stage would set the first unfetchable instruction as the next first instruction it will fetch. For example, if fetch stage wants to fetch instructions with PC 0, 4, 8 this cycle but PC 8 turns out to be a cache miss, it will attempt to fetch instructions with PC 8, 12, 16 in the next cycle. What's more, branch predictors can also overwrite the PC counter by asking the fetch stage to jump to a non-sequential PC.

B. Reservation Station

Our reservation station has 16 entries with no internal forwarding (An entry can't be issued and dispatched into in the same stage). The RS allocates new entries for newly dispatched instructions, watches on the tag broadcast from complete stage and identifies the instructions that has no dependency that are ready to go and issue them. The dispatch instructions has priority to be allocated at the rop of the reservation station while the issue selection logic prioritise from the bottom of the reservation station. In that case, RS is more likely to issue older instructions first.

The 3-way scaled selection logic for 16 entries RS that consider multiple functional units assignment and stall is very complicated and has a delay of 14ns. To reduce the delay without conpromising on the number of entries, we only consider the type of functional unit and leave the assignment to particular functional unit in issue stage. In issue stage, we build a first in first out (FIFO) queue for each kind of functional unites, and assign instructions at the head of the FIFO to each free functional units. The FIFO has internal forwarding, therefore the instructions don't need to wait one cycle before they are sent to the FUs when the queue is empty. These queue are 32 entries each and are impossible to stall because they are larger than our ROB.

C. ROB

Our ROB has 32 entries with internal forwarding (an entry can be engaged when its previous instruction retires in the same cycle). It enables the pipeline to handle precise states and saves all the information that retire stage needs to know. It uses a head and a tail pointer to mark the first and the last non-empty entry. Both head/tail pointer can move three entries at most per cycle to support three-way superscalar.

When a new cycle starts, the head pointer will move based on the number of completed instructions that are going to be retired in this cycle. The tail pointer will then move based on the its distance from the updated head pointer and sends structural stall if the left entry space is smaller than the number of instructions given by the dispatch stage.

The ROB saves and removes instructions based on the movement of head/tail pointer. When an instruction is dispatched and saved in ROB, it sends out the entry index of that instruction. After an instruction is completed, ROB uses its attached ROB entry index to set the complete flag high. Completed instructions are sent to the retire stage in the next cycle.

The ROB saves the physical register and architecture register for the retire stage updating Map Table and Arch Map Table. It also saves predicted direction/PC when the instruction is added to compare with the results of branch execution. It sends precise state signal to retire stage if two results mismatch.

When a precise state happens, all ROB entries are cleared and head/tail pointers are set to zero by default.

D. Freelist

Our Freelist has 32 entries with internal forwarding. (a Physical Register can be used when it is freed in the same cycle.) Its basic logic is the same as ROB with a head and a tail pointer pointing the first and the last available Physical Register. Both head/tail pointer can move three entries at most per cycle to support three-way superscalar.

The Freelist size is designed to avoid structural hazards that only due to itself. Since the ROB has 32 entries at most, it is not possible that the Freelist is empty before the ROB is full. So Freelist's structural hazards are mostly covered by RS/ROB's structural hazards.

When a precise state happens, the tail pointer will move as usual to handle instructions that retire in that cycle. The head pointer will set to the entry one after the tail pointer marking there are 32 available Physical Registers by default.

E. Functional Units

Our design include 8 functional units: 1 branch resolves (1 cycle), 2 multiplying units (5 cycles), 3 arithmetic-logic units (1 cycle) and 2 load units (3+ cycles). The branch resolver handles branches (B and J branches), the multiplying unites handle multiplications (Our processor doesn't support divide.), the load units handle load instructions and the ALUs handle everything else. Note that our processor direct store instructions towards ALUs. The reason will be explained in the following section talking about store queue.

F. Map Table & Arch Map Table & Physical Registers

A Map Table and an Architecture Map Table are implemented in order to perform register renaming in R10K style. We have 32 architectural registers and 64 physical registers. The Map Table hold the register renaming map of all dispatched instructions, and all incoming instructions can look up their operands. When a new cycle starts, the Map Table entries will be updated one by one using dispatch results. It also support internal forwarding of the CDB broadcast. (One update by CDB broadcast can be read by dispatch stage in one cycle.) The Architectural Map Table holds the architectural state of Map table, it only updates for instructions that have retired.

When a branch misprediction happens, the Architectural Map Table will help rewind the state of the Map Table. However, since we are implementing a 3-way superscalar processor, it's a little more complicated because not all instructions in this cycle may be cleared. So we reprocess the Architectural Map Table entries in retire stage and then send it to Map Table, as shown in Fig. 1.

IV. Advanced High-performance Features

A. Prefetcher

Due to the fact that it takes multiple cycles to require data from memory and ICache and DCache might send request to memory at the same cycle, it always takes a long time to get a new instruction. To improve this, a prefetcher is implemented. It is triggered on every ICache reference. It automatically gets the next 12 lines of data from the memory one by one, and writes them back to the ICache. The number 12 here is defined as a macro in our project. The detailed analysis on the choosing of this number will be discussed in the analysis section later.

To prevent duplicate fetching from the same address, prefetcher always sends a signal to ICache when ICache wants to request for data from the memory. This signal tells the cache whether a memory request of that address has been sent to the memory.

B. Branch Predictor

Our branch predictor has a 32-entry directed-mapped buffer. It uses the least significant 5 bits as the entry index, and the most significant 27 bits as the entry tag. It predicts branch instructions based on two-bit saturating counters that consider new instructions as non-taken at the beginning.

When a branch instruction enters the fetch stage, it is passed to the branch predictor. If the instruction already exists in the branch predictor buffer, the predicted direction and PC will be output to the if_id_register. Otherwise, it is considered as non-taken. After the instruction is decoded in the dispatch stage, it will be passed to the branch predictor if it is a branch instruction. The buffer will saves the new branch instruction if its corresponding entry is empty or engaged by a different older instruction. The branch function unit will update the predicted direction and PC when it completes a branch instruction and the branch instruction exists in the buffer.

To handle misprediction case, the prediction result of the branch predictor will be attached to the instruction and saved in the ROB through the dispatch stage. The predicted direction and PC will be compared to the executed direction and PC when the branch instruction is completed. If they don't match, ROB will give priority to the execution results and start a precise state process.

To handle dispatch stall and multiple branch prediction cases, the prediction results will be processed in the if_id_register. If a branch is predicted taken, the same cycle instruction after it will be abandoned and the if_id_register will not input new instructions from the fetch stage until the branch instruction finishes dispatching. If a precise state or multiple branch prediction happens in the same cycle, only one branch prediction or precise state branch will be processed. The fetch stage will give priority to the precise state, then the oldest branch instruction.

C. Store Queue

Store Queue is where memory store operations are stored and reordered to support precise state. In our design, we chose 8-entry store queue with no internal forwarding. (One entry cannot be retired and then allocated in the same cycle.) This is a rather simple implementation of store queue. Because of this implementation, every load instruction need to wait until all the store instructions before it to execute. An interface with the reservation station ensures that any load instructions won't be issued until all the stores older than them are executed.

This simple implementation of store queue can be the system bottleneck especially for data heavy programs. Fortunately, as we mentioned in the FU section, stores are handled by ALUs and need only one cycle to execute. The

ALUs are unlikely to stall because there are three ALU who can take new instructions every cycle unless it needs to wait other ALUs to complete first.

V. Memory Interface

A. Non-blocking Instruction Cache

The ICache is non-blocking and directly mapped. It's size is 256 bytes with 8 bytes per line and 32 lines in total. On a cache miss when the fetch stage requires data from it, it will send a request to the memory.

B. Non-blocking Write-back Write-allocate Directly Mapped DCache

We implemented a non-blocking, write-back, write-allocate, directly mapped data cache in this project. The data cache uses a block size of 8 bytes, and has 32 entries in total, so it can contain at most 256B of data. We implemented a miss status handling registers (MSHRs) to make the DCache be able to handle multiple misses at the same time. The design of this MSHRs is explained in section V.C.

When a memory block to be written is found in DCache, the DCache is updated correspondingly. However, this change does not go to the memory immediately. Instead, DCache will mark the block it just written as dirty. Dirty blocks will be written back to the memory either when they are evicted or the whole program halts. This write-back policy increases the hit rate and reduces the memory bandwidth consumed by storing. When a memory write misses, the DCache will first load the corresponding block from the memory, then override the the part that we want to write (since the RISC-V ISA supports different width of store), finally write this block data to DCache, allocating a new block if needed. The specific method of doing this is explained below.

When a memory load hits, the data is directly transferred from the DCache to the Load functional unit. When a memory load misses, the DCache will first load the corresponding block from the memory, then write this block into DCache. Finally, the load hits and the data is transferred form the DCache to the Load functional unit.

Preserving the dependency of the memory accesses when designing a non-blocking write-allocate cache is a bit more complicated. We used a series of strategies to prevent errors. Here is the strategy of handling the update of DCache entries.

When a block of data should be written to the DCache, we will first look if the target cache entry(set) is valid. If it's still invalid, the data will be directly written to the target entry. If it's already valid, we will then look if the current tag in this target entry is the same as that of the data block we are going to write. If they are the same, then we don't need to evict the old block, we only need to override the write part of current update if it's a store miss (if it's a load miss, do nothing). If they are not the same, we should evict the old block if it's dirty by sending a store request to memory, and then write the new block data to the target entry.

Furthur more, we use a special ”load hazard” detecting logic to guarantee that if a load access and any older missed store access have the same block address, the load shouldn't get the data until that store miss is handled. This design is explained in section V.C.

C. DCache Controller and Miss Status Handling Registers (MHSRs)

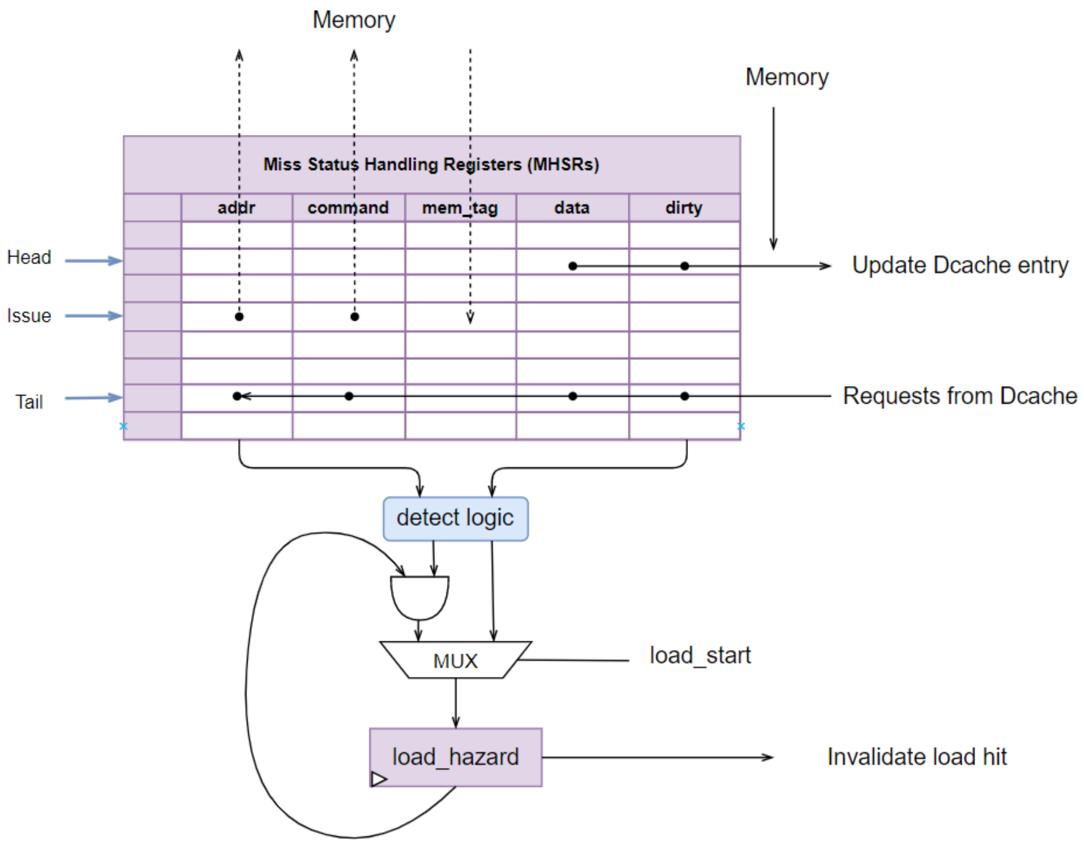


Fig. 2: Data Cache Controller for Memory Accesses

Since we have 2 load functional units, and at most 3 store requests per cycle, at most 6 memory requests (2 load miss, 3 store miss and 1 evicted dirty block to write back) can come from DCache in one cycle but the memory can only accept one, we implemented a MHSRs to help handle these memory accesses. Figure 2 is the data cache controller we implemented to perform the memory access sequentialization and the special “load hazard” detecting.

The MHSRs is a first-in-first-out (FIFO) queue with three pointers: head, issue, and tail. In our design, it has 16 entries. The memory accesses between the tail pointer and the issue pointer are those which has been listed but not yet sent to memory. The memory accesses between the issue pointer and the head pointer are those which has been sent to memory but not yet committed. When a memory access is requested from DCache, a MSHR entry will be allocated for it, and the tail pointer will increase 1 (if MSHRs is not full). When the entry that issue pointer is pointing to is not empty, it will be send to memory and the memory responding tag will be stored in this entry. And the issue pointer will increase 1 to issue the next entry in future cycles. When the entry that head pointer is pointing to is a load and gets data from the memory, the processed data will go to DCache and update the corresponding DCache entry. Then this entry will be cleared in the next cycle and the head pointer will increase 1 to commit the next entry in future cycles. (If the entry that head pointer is pointing to is a store, it will directly be committed.)

The “dirty” bit in the MHSRs table is used to distinguish load accesses caused by load miss or store miss. If dirty bit is 1, it’s a store miss, and the data to write is also stored in this MSHR entry, waiting to override the data loaded from memory. If dirty bit is 0, it’s a load miss, and the data from memory will be directly send to DCache.

We also implemented a “load hazard” detecting logic to prevent getting wrong data if a load is coming after a store miss with same block address. When a load request is sent from load functional unit to DCache, this logic will compare all addresses in the MHSRs table with the current load, and it also detect the dirty bit to see if the entry is a store miss. The load_hazard register has 16 bits, every bit is mapped to an entry of MHSRs table. So this logic can keep detecting “load hazard” every cycle, and by adding an and gate, it could mask out all the younger store misses and only keep tracking on the store misses existed before the current load comes.

However, in our final code submission before due, this logic still has a problem so it can't handle a tricky case. This caused the test case "alexnet.c" failed when doing post synthesis simulation. Now we have fixed this bug and run all the public test cases correctly.

D. Memory Controller

As discussed in previous sections, ICache and DCache both send reading requests to the memory, and DCache can also write data to the memory. To decide which address is going to be read from or write to, the memory controller is implemented. It places DCache's priority before ICache. That is, when both ICache and DCache send a request to the controller, it would forward DCache's request to the memory and send a stall signal to ICache to tell it that the memory is busy. In the reverse signal propagation direction, the controller just directly forwards the memory response to ICache and DCache.

VI. Testing

This part gives an overview of our testing strategies and debug flow. Here future students doing this project might find some advice and the starting point.

A. Test Strategy

Since team members implement different modules, each module is individually tested after they are implemented. Both simulation and synthesis tests are applied to make sure the module functions correctly and doesn't have over complex logic.

Modules are integrated and tested spatially, for example Complete/Retire Stage, Map Table, Freelist, and ROB are connected to see the changes when a instruction finishes execution. Testing spatially makes debug easier for later overall integration.

The integrated pipeline is tested with the provided testcases and a visual debugger. After pipeline is able to pass single testcases, a autotest shell script is designed to verify its performance on all testcases automatically.

B. Testbench

During the development, we had four types of testbenches: module tester, simulator, pipeline debug tester and the final platform.

We started the project by developing each blocks and their corresponding testbenches. In these testbenches a basic debug print task for the module is implemented. The testbenches feed in naive instructions and control signal to make sure the modules work before we connect them to the pipeline.

To have the pipeline running and start testing before we have all the modules, we implemented several simulators in cpp. Later we compared the result with the potentially buggy new module. Architectural map table, cache and memory and the prefetcher are three main simulators that helped during our development.

The pipeline debug test is what we use for debugging and optimization. Here we combine the print tasks from the module testbenches and add pipeline prints. We also generate debug output in the same style as project3 pipeline.out so that we can diff the writeback values with the correct output from project 3 and locate bugs quickly.

The final platform is the one our processor runs on during final tests. It is a slim techbench that only outputs the memory result and a debug print every 10000 cycles so that we know the simulation is still running.

C. Autotest Shell Script

To test multiple testcases at one time, a shell script is written to compare our results and the correct outputs. It compiles each testcase into binary code, runs simulation/synthesis, and only compares memory data.

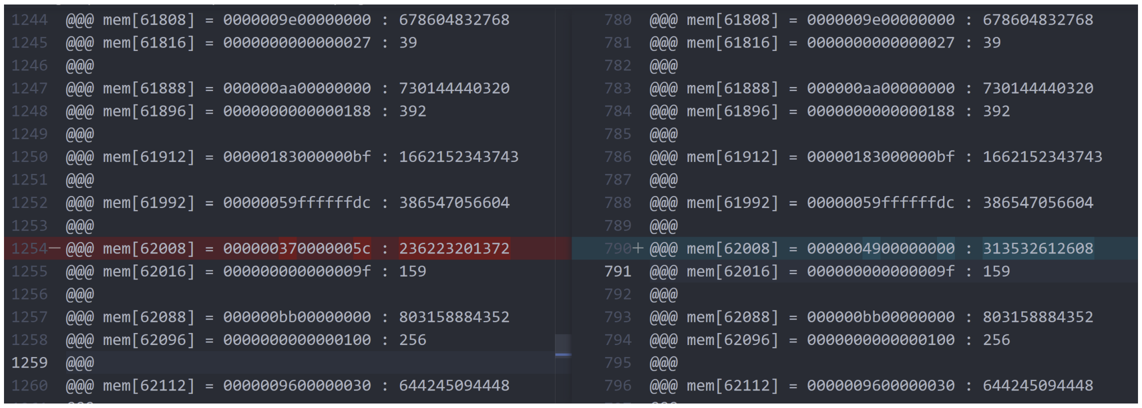
It prints pass/fail outcomes to a autotest_result.txt file and saves our results of each testcase in an individual folder for later debug.

D. Testing Results

For simulation, We run our pipeline on all testcases listed in Appendix. All runned testcases pass except alexnet.c.

For synthesis, We run our pipeline on all testcases except outer_product.c because it takes too long to run. All runned testcases pass except alexnet.c.

Looking into our output of alexnet.c, we find that only one memory data is incorrect. We print out all the register writeback value and they are also correct, so we infer that the problem is due to DCache's interaction with memory. Our decache fails to correctly handle some tricky scenarios and saves a wrong value to the memory.



```
@@@ mem[61808] = 0000009e00000000 : 678604832768
@@@ mem[61816] = 0000000000000027 : 39
@@@
@@@ mem[61888] = 000000aa00000000 : 730144440320
@@@ mem[61896] = 0000000000000188 : 392
@@@
@@@ mem[61912] = 00000183000000bf : 1662152343743
@@@
@@@ mem[61992] = 00000059ffffffdc : 386547056604
@@@
-@@@ mem[62008] = 000000370000005c : 236223201372
@@@ mem[62016] = 000000000000009f : 159
@@@
@@@ mem[62088] = 000000bb00000000 : 803158884352
@@@ mem[62096] = 0000000000000100 : 256
@@@
@@@ mem[62112] = 0000009600000030 : 644245094448
```

```
@@@ mem[61808] = 0000009e00000000 : 678604832768
@@@ mem[61816] = 0000000000000027 : 39
@@@
@@@ mem[61888] = 000000aa00000000 : 730144440320
@@@ mem[61896] = 0000000000000188 : 392
@@@
@@@ mem[61912] = 00000183000000bf : 1662152343743
@@@
@@@ mem[61992] = 00000059ffffffdc : 386547056604
@@@
+@@@ mem[62008] = 0000004900000000 : 313532612608
@@@ mem[62016] = 000000000000009f : 159
@@@
@@@ mem[62088] = 000000bb00000000 : 803158884352
@@@ mem[62096] = 0000000000000100 : 256
@@@
@@@ mem[62112] = 0000009600000030 : 644245094448
```

Fig. 3: Wrong Memory Data of alexnet.c

VII. Performance Analysis

A. Performance Metrics

The most effective rule to evaluate the performance of a program is the iron law:

$$\begin{aligned} Performance &= \frac{\#Instructions}{Program} \times \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle} \\ &= CodeSize \times CPI \times ClockPeriod \end{aligned} \tag{1}$$

B. Overall Performance

We calculated Time Per Instruction of all testcases for our pipeline and the in-order pipeline of project 3. Here we assume the clock period of the in-order pipeline of project 3, which is the latency of the memory access. The real latency of the in-order pipeline should be the latency of the memory access plus the critical path of the pipeline itself.

$$\begin{aligned} Time/Instruction &= \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle} \\ &= CPI \times ClockPeriod \end{aligned} \tag{2}$$

The average time per instruction for the two versions are $Time/Instruction_{complete} = 71.07ns$ and $Time/Instruction_{P3} = 179.64ns$. It can also be seen from Fig. 4 that the Time Per Instruction of all programs are decreased significantly in our pipeline, indicating that our pipeline has a much faster instruction processing speed.

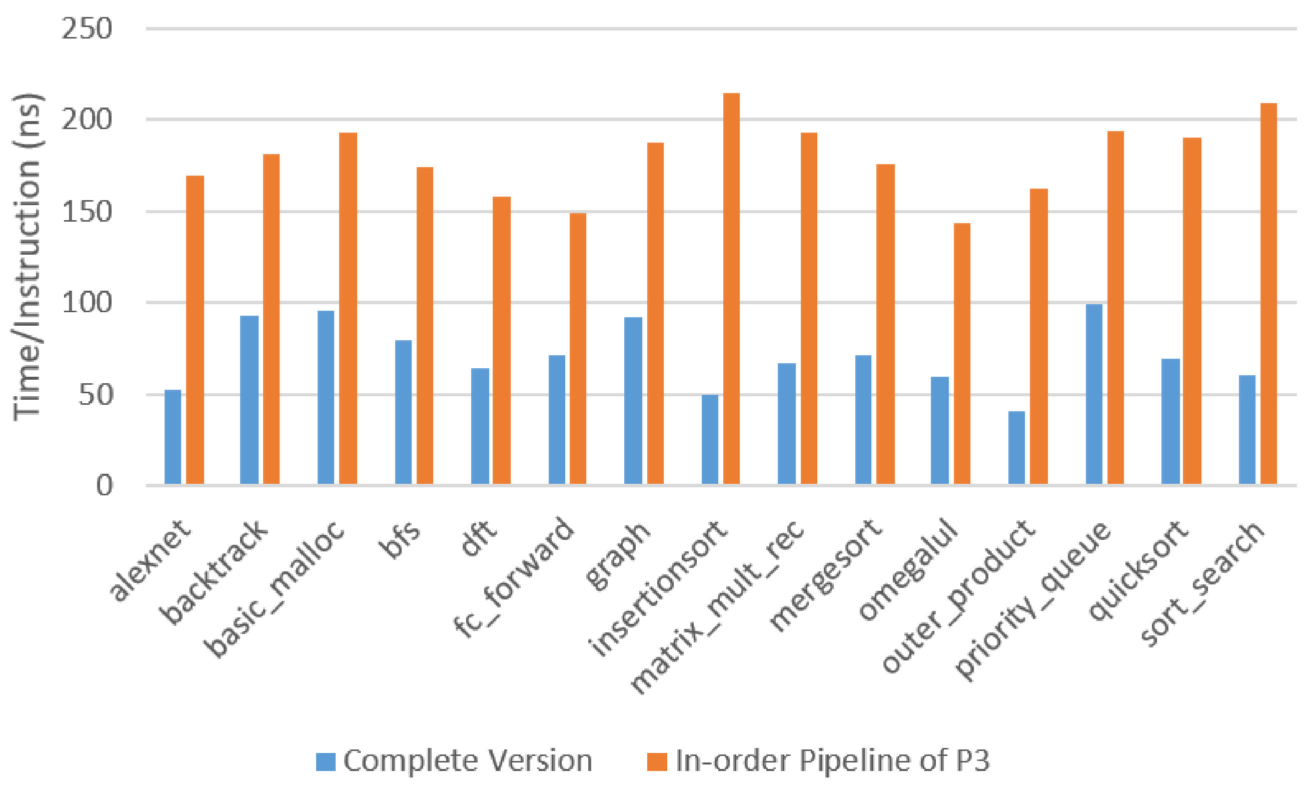


Fig. 4: Time Per Instruction of Our Pipeline and In-order Pipeline

C. Prefetch

As described in Design Choices part, we implemented the number of lines to prefetch as a macro. By changing the number to different value, we get different average CPI to finish running the public test cases. As shown in Fig. 5, when the number of prefetching lines increments from 2 to 4, the CPI drops significantly, which indicates a large improvement in the performance. However, as the number keeps increasing, the CPI changes very slightly. It is due to the fact that as the prefetcher accessing more to the memory, the larger probability it will interfere with the DCache. And since in our design, the requests from DCache always have higher priorities than those from ICache and prefetcher, the prefetcher cannot be fully utilized. Therefore, we should better change our prefetching lines to 4 or 6 so as to keep a good fetching performance and reduce the chances of conflicts to occur between ICache and DCache.

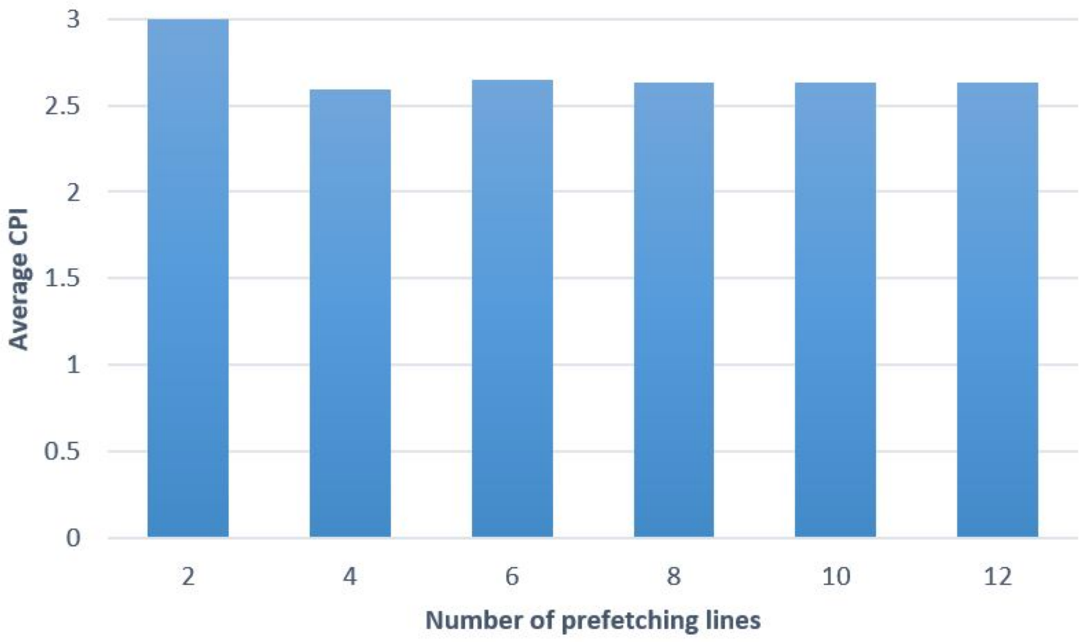


Fig. 5: Average CPI Versus Different Numbers of Lines to Prefetch

We select and run some long .c testcases on our final pipeline version and a version without Prefetch and records their CPI. It can be seen from Fig. 6 that with Prefetch, the CPI of most programs are decreased significantly, espically for testcases like backtrack, bfs, dft, etc. The average CPI for the two versions are $CPI_{complete} = 2.22$ and $CPI_{no\_prefetch} = 2.48$. Though some testcases like insertionsort and omegalul have equal or higher CPI with our final version, the increased CPI due to misprediction is relatively small. Those higher CPI are due to the probabilities that prefether might overwrite old ICache blocks, thus changing an ICache access that should have been a hit into a cache miss.

Overall, our Prefetch makes a distinguishable improvement on our pipeline CPI.

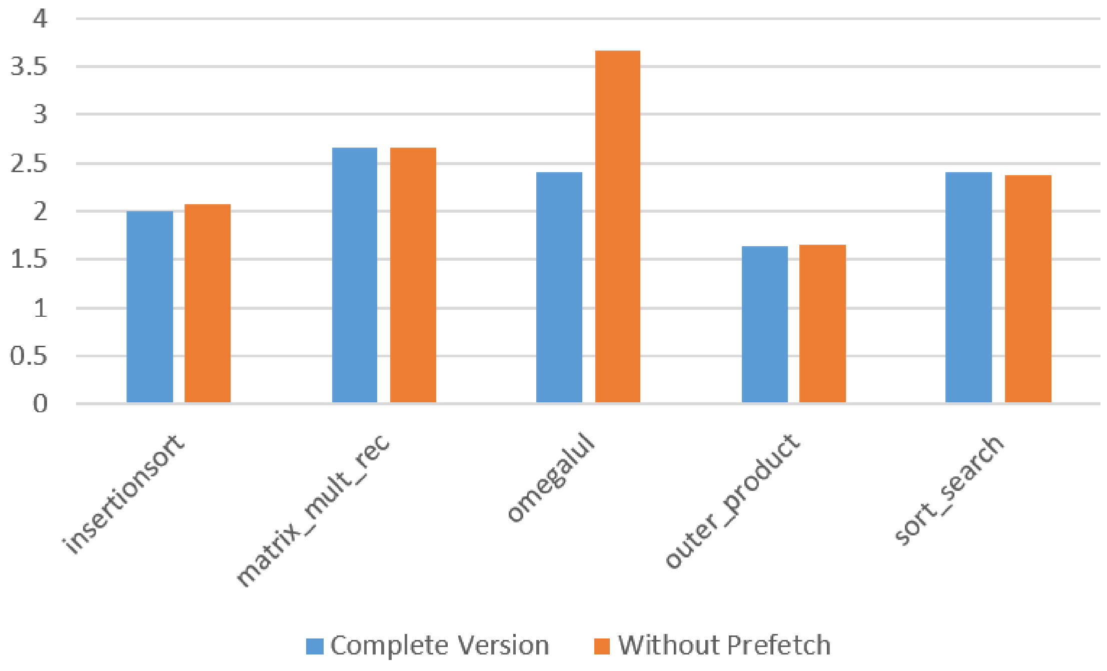


Fig. 6: Testcase CPI of Pipeline with/without Prefetch

D. Branch Predictor

We run all the .c testcases on our final pipeline version and a version without Branch Predictor and records their CPI. It can be seen from Fig. 7 that with Branch Predictor, the CPI of most programs are decreased significantly. The average CPI for the two versions are $CPI_{complete} = 2.84$ and $CPI_{no\_BP} = 3.30$. Though some testcases like backtrack and basic_malloc have equal or higher CPI with our final version, the increased CPI due to misprediction is relatively small. These testcases may have more complex branch history that can not be handled by a two-bit saturating counters, so a more advanced branch predictor can be used in the future.

Overall, our branch predictor makes a distinguishable improvement on our pipeline CPI.

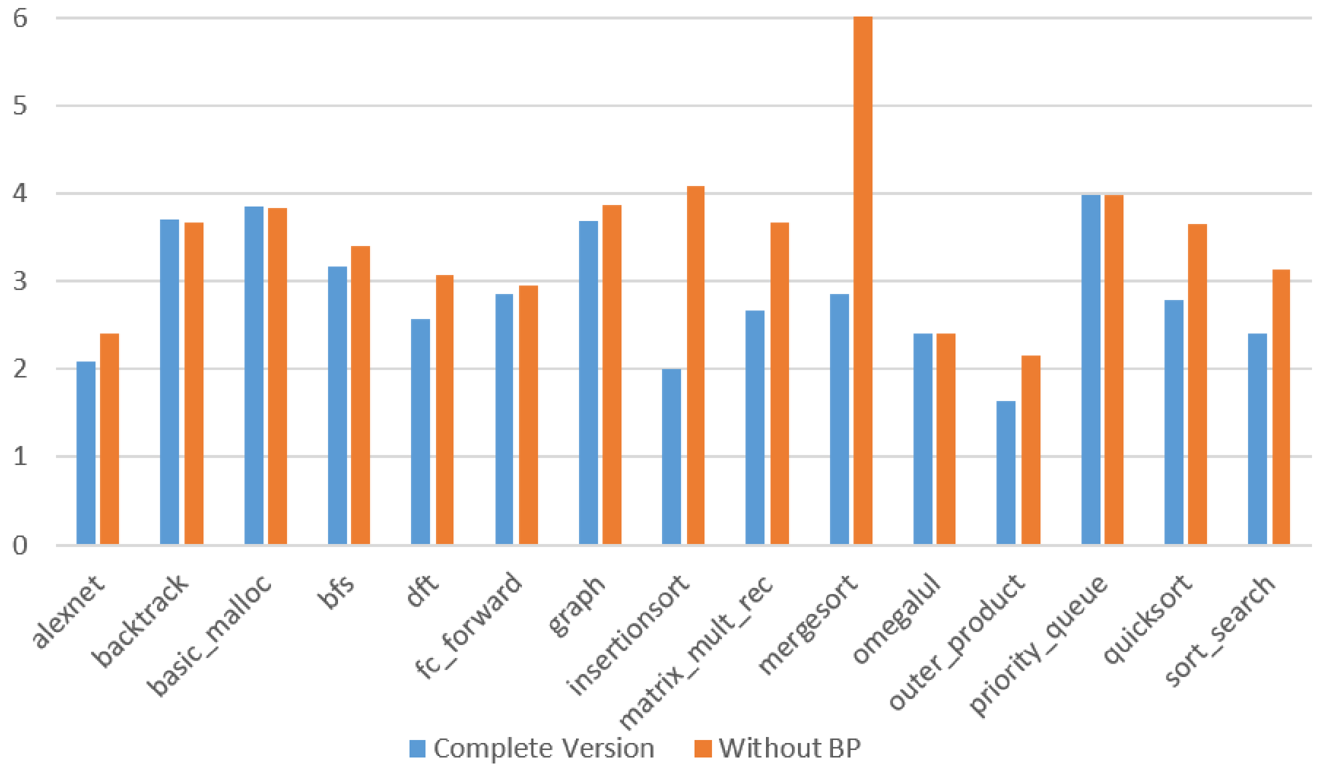


Fig. 7: Testcase CPI of Pipeline with/without Branch Predictor

E. ROB/RS Sizes

Our ROB has 32 entries, while RS has 16 entries. Since RISCV ISA has 32 architectural registers, we have 64 physical registers in our design. The ROB and RS are the bottle neck of the pipeline. ROB often causes structural stall when executing high latency instructions such as multiply, load, and store, while RS causes structural stall when adjacent instructions have RAW dependencies on each other, especially when these instructions have high latency.

After adding the branch predictor, the density of instructions and the possibility of dependencies are both increased. ROB and RS causes structural stalls more frequently.

However, increasing the size of ROB and RS will result in longer clock period because these two units communicates with many other units back and forth. The final size we select is a balanced version between CPI and Cycle Time.

VIII. Our Test Case

We wrote a test case called testcase.c to test on multiple tricky memory accesses patterns to ensure our DCache and LSQ is working correctly. We also tested on heavy branches to ensure that our ICache, branch predicter and prefetcher works well.

The test case is 6489 instructions long, and our processor get the correct output with a CPI of 1.907. The test case code is attached in the appendix and it's also submitted through git. Its name is "testcase.c" and it can found under the folder "test_progs" in branch "our_testcase".

This test case tests these listed tricky situations:

- load after store miss
- load - store - load to the same block
- load before store miss
- multiple store miss to the same block but writing different parts
- heavy dependency memory access series
- high density memory accesses
- frequent branch instructions
- frequent jumps to a certain function

IX. Further Optimization

After the final submission, we were able to fix some bugs and achieved a major clock period optimization on our processor with little CPI compromise. We improved the clock period from 25 ns to 13.1 ns.

This version is on final-opt branch of the repository.

A. Fix Data Cache Load-Store-Load Bug

In the submitted version, public case Alexnet.c will fail at tricky DCahe access case: A load-store-load to the same address. In the original version, cache hit will happen for the second load instruction while the store data is waiting to be written in. We decided to stall any new load accesses if there is an awaiting store to the same address until the store instruction in written to cache. After this quick fix, the new DCache is able to handle all public cases.

B. Critical Path Optimization

After careful clock cycle testing, we identified the critical path in our design. It starts with the complicated DCache retire stall logic, which ensures that only one store miss instruction is allowed to retire. This logic needs data from SQ, DCache and RoB and is quite complicated.

1) SQ Internal Forwarding Optimization: The critical path in our 25ns-clock submission starts from SQ head packet that is sent to identify store miss in DCache, which decides Data Cache retire stall at ROB, and then goes to Retire Stage, back to SQ head update logic, and then to dispatch SQ structural stall and ends in Branch Predictor.

$$SQ \Longrightarrow DCache \Longrightarrow ROB \Longrightarrow RetireStage \Longrightarrow SQ\ head\ update \Longrightarrow SQ\ allocate\ stall \Longrightarrow dispatchstall \Longrightarrow BP$$

In order to break this chain, we cancelled SQ internal forwarding logic so that dispatch SQ structural stall doesn't depend on the complicated DCache retire stall logic. After applying this logic, we are able to reduce clock period from 25ns to 17ns.

2) DCache Repetitive Logic Elimination: The second longest path also starts with the DCache retire stall logic but ends at updating MHSRS entries in DCache. After the SQ receives signal from retire stage that tell it how many instructions to retire, it sends corresponding entries to DCache to store the data to memory. The DCache then checks for cache misses and put missed instructions to MHSRS.

$$SQ \Longrightarrow DCache \Longrightarrow ROB \Longrightarrow RetireStage \Longrightarrow SQ\ retire\ entries \Longrightarrow DCache\ miss \Longrightarrow MHSRS$$

However, the retire entries sent to DCache are actually guaranteed to the first a few entries at SQ head, which we've already checked for cache miss in DCache retire stall logic. The DCache only need to know how many instructions to store at the SQ head.

By eliminating redundant cache miss logic, we are able to reduce clock cycle down to 13.1ns. Compared to the 25ns clock version, our critical path optimization managed to achieve almost 2x performance improvement.

C. Performance Analysis

We select and run a couple of long .c programs to examine the improvement of our optimization. The average time per instruction for the two pipelines are $Time/Instruction_{Before} = 71.07ns$ and $Time/Instruction_{After} = 38.35ns$. As shown in Fig. 8, the Time/Instruction of our pipeline after optimization is almost half of the original pipeline's, which means the performance of the pipeline is twice as good as the old one.

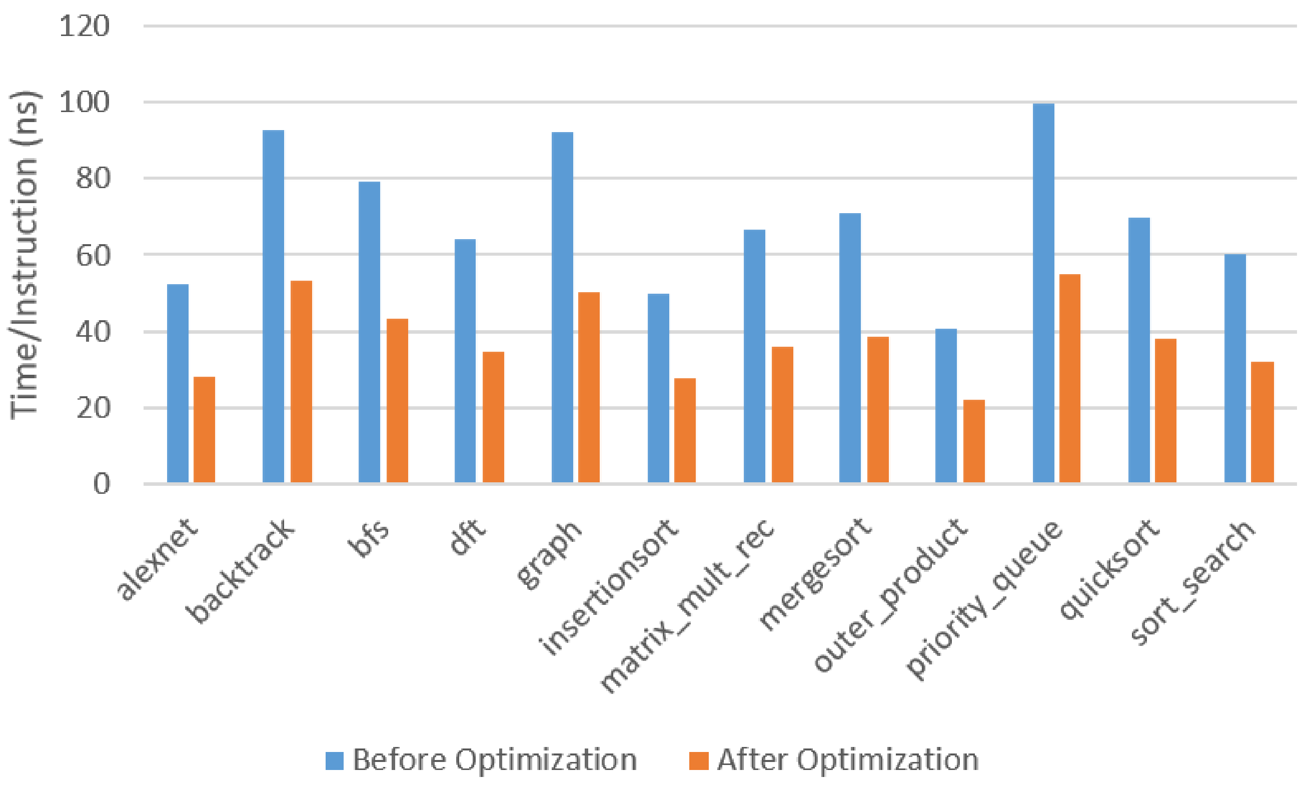


Fig. 8: Time Per Instruction of Pipeline Before/After Optimization

X. Team Logistics

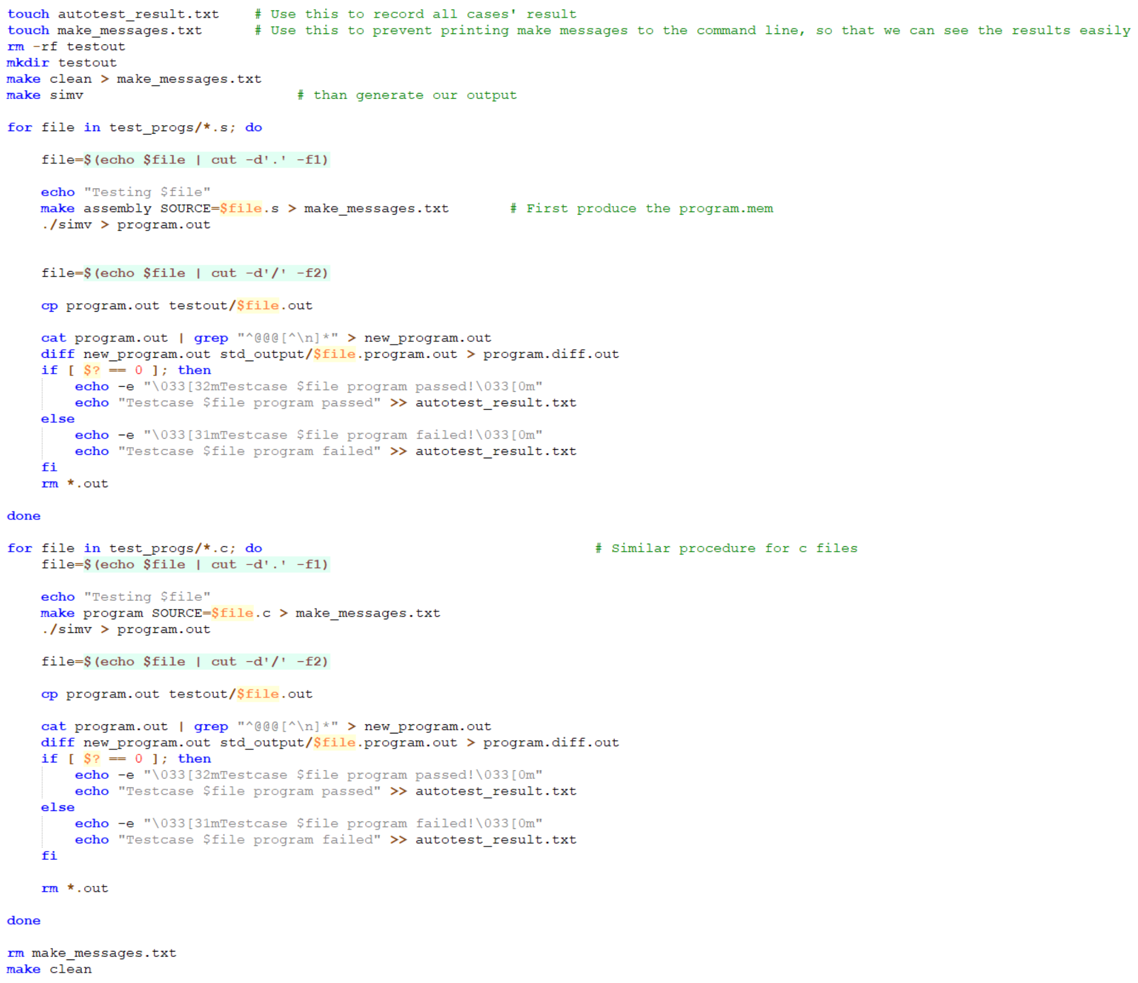
- Juechu Dong (25%): Reservation Station(Optimization), Store Queue, Dispatch Stage, Issue Stage, Functional Units (Except branch solver), Test Infrastructure, and Debugging
- Haoyang Zhang (25%): DCache, Memory Controller, Retire Stage, Map Table/Arch. Map Table, Branch FU, and Debugging
- Xiangdong Wei (25%): Reservation Station(Issue Logic), Reorder Buffer, Freelist, Branch Predictor, Pipeline Integration, and Debugging
- Chen Huang (25%): Reservation Station(Issue Logic), Fetch Stage, ICache, Prefetcher, Complete Stage, Pipeline Integration, Test Infrastructure, and Debugging

Appendix A
Public Testcase List

Here lists all the assembly and C programs we use to test our final pipeline.

- haha.s
- rv32_btest1.s
- rv32_btest2.s
- rv32_copy.s
- rv32_copy_long.s
- rv32_evens.s
- rv32_evens_long.s
- rv32_fib.s
- rv32_fib_long.s
- rv32_fib_rec.s
- rv32_halt.c
- rv32_insertion.s
- rv32_mult.s
- rv32_parallel.s
- rv32_saxpy.s
- sampler.s
- tj_malloc.h
- alexnet.c
- backtrack.c
- basic_malloc.c
- bfs.c
- dft.c
- fc_forward.c
- graph.c
- insertionsort.c
- matrix_mult_rec.c
- mergesort.c
- omegalul.c
- outer_product.c
- priority_queue.c
- quicksort.c
- sort_search.c

Appendix B
Autotest Script Code



```
touch autotest_result.txt    # Use this to record all cases' result
touch make_messages.txt      # Use this to prevent printing make messages to the command line, so that we can see the results easily
rm -rf testout
mkdir testout
make clean > make_messages.txt
make simv                         # than generate our output

for file in test_progs/*.s; do

    file=$(echo $file | cut -d'.' -f1)

    echo "Testing $file"
    make assembly SOURCE=$file.s > make_messages.txt       # First produce the program.mem
    ./simv > program.out


    file=$(echo $file | cut -d'/' -f2)

    cp program.out testout/$file.out

    cat program.out | grep "^@@@[^\n]*" > new_program.out
    diff new_program.out std_output/$file.program.out > program.diff.out
    if [ $? == 0 ]; then
        echo -e "\033[32mTestcase $file program passed!\033[0m"
        echo "Testcase $file program passed" >> autotest_result.txt
    else
        echo -e "\033[31mTestcase $file program failed!\033[0m"
        echo "Testcase $file program failed" >> autotest_result.txt
    fi
    rm *.out

done

for file in test_progs/*.c; do                                  # Similar procedure for c files
    file=$(echo $file | cut -d'.' -f1)

    echo "Testing $file"
    make program SOURCE=$file.c > make_messages.txt
    ./simv > program.out

    file=$(echo $file | cut -d'/' -f2)

    cp program.out testout/$file.out

    cat program.out | grep "^@@@[^\n]*" > new_program.out
    diff new_program.out std_output/$file.program.out > program.diff.out
    if [ $? == 0 ]; then
        echo -e "\033[32mTestcase $file program passed!\033[0m"
        echo "Testcase $file program passed" >> autotest_result.txt
    else
        echo -e "\033[31mTestcase $file program failed!\033[0m"
        echo "Testcase $file program failed" >> autotest_result.txt
    fi

    rm *.out

done

rm make_messages.txt
make clean
```

Fig. 9: Autotest Script Code

Appendix C
Our Test Case

testcase.c

```c

int a[100];
int b[100];

int getSomeInt(int n) {
    if (n < 2) {
        return 100;
    } else {
        return -100;
    }
}

int main(){

for (int i = 0; i < 100; i++)
{
    b[i] = i;
}

// load after store: b[0]
b[0] = 9;
a[0] = b[0];

// load - store - load to the same block
b[3] = b[2];
a[1] = b[3];

// load before store:
int c ,d;
 c = b[4];
 b[4] = 0;
 a[4] = c;

// multiple store miss to the same block but writing different parts
a[6] = 7;
a[7] = 8;
b[7] = a[6];
b[6] = a[7];

//heavy dependency
a[9] = b[10];
a[10] = a[9] + 9;
b[10] = a[9];
a[10] = b[10] - 8;
a[11] = b[10] + 89;
a[15] = a[11];
a[16] = a[15] - a[11];
a[15] = a[16] + a[15];

//high density memory accesses
for (int i = 0; i < 100; i++)
{
    b[i] = a[i]+b[i]-a[i]*b[i];
}

// branch
int n = 0;
for (int i = 0; i < 30; i++) {
        if (n == 1) {
                a[n] = getSomeInt(n);
                n++;
        } else if (n == 2) {
                a[n] = getSomeInt(n);
                n++;
        } else {
                a[n] = getSomeInt(n);
                n++;
        }
}

return 0;
}
```

Acknowledgment

The authors would like to thank Professor Jonathan Beaumont, Alireza Khadem, Scott Smith, John Campbell for their help and advice on the project.