EECS 470 Computer Architecture Microprocessor Project Design Report

Yuqing Qiu
qyuqing@umich.edu

Shixin Song
shixins@umich.edu

Zesheng Yu
jensenyu@umich.edu

Chenyan Zhang
zchenyan@umich.edu

Zimeng Zhang
zimengzh@umich.edu

Abstract

Out-of-order pipeline structure is a common and efficient way to design the processor. In this report, we are going to introduce the basic implementation and advanced features of our pipeline and evaluate its performance under different cases. Our group implemented an out-of-order, 3-way superscalar microprocessor [3] based on the MIPS R10K processor architecture with several advanced features [1][2]. We measured and compared the performance of the processor with different features on multiple test programs.

1. Introduction

EECS 470 leads us to the deeper principles of computer architecture. We have learned multiple techniques to optimize instruction flow, branch resolution and memory accesses. We have learned a simplified version of MIPS R10K processor [4] architecture in class and would like to explore its whole functions.

In this project, we are implementing an out-of-order, 3-way superscalar microprocessor based on the MIPS R10K processor architecture with advanced branch predictor, instruction prefetcher, 4-way set associative blocking data cache, FIFO Instruction buffer and complete buffer. The programming language we used is System Verilog. Our processor supports 32 bits RV32IM ISA.

In this report, we will introduce the details of our design, the performances of different features, the group logistics and possible future optimizations. In design, we will focus on how each module works and what motivates our decisions. In performance part, we will focus on evaluating performances and their possible relationships with additional features.

[1]The source code of basic Gcat pipelne can be found at `https://bitbucket.org/eecs470staff/group12w22/src/final_submit`

[2]The source code of advanced Gcat pipelne can be found at `https://bitbucket.org/eecs470staff/group12w22/src/advanced_manual_fix`

2. Processor Design

The graph (Figure 1) describes our design of the processor and how it interacts with the memory. Generally, an instruction goes from the top of the graph to the bottom. As the bar in the left showed, each instruction goes through six stages which are Fetch (F), Dispatch (D), Issue (S), Execute (E), Complete (C) and Retire (R). The physical register file keeps data values. Functional units do arithmetic calculations. The RS, ROB, Map Table, Freelist and CDB help maintain an out-of-order pattern. The LSQ, DCache and ICache controls the data flow from or to the memory. Prefetcher, BP, BTB and Arch. table handles prediction and resolution of branch instructions.

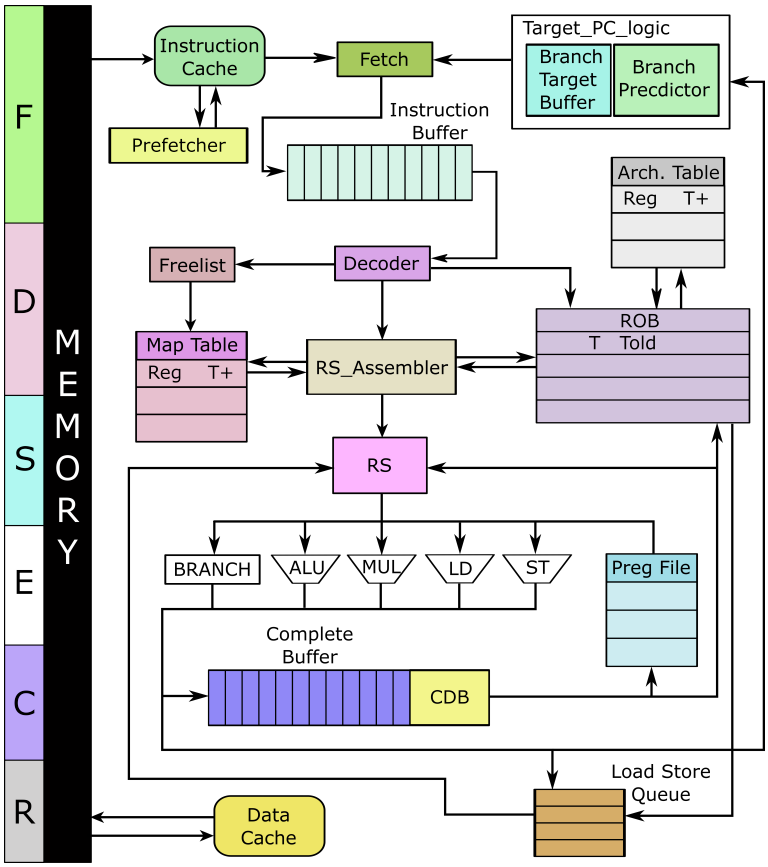

Figure 1: R10K Architecture

Our critical design specifications are listed in Table 1. The reasons why we chose these parameters will be illustrated in the following sections.

2.1. Fetch Stage and Instruction Buffer

The fetch stage fetches instructions from icache. It requests for a cache line at a time, which is two instructions, and writes the result to the instruction buffer. The fetch stage will continue fetching until the instruction buffer is full, even if there is traffic in the pipeline. The top three instructions in the instruction buffer will be dispatched on request. The advantage of having an instruction buffer is to supply enough instructions to our 3-way superscalar pipeline.

Module	Num	Unit
RS	13	entries
ROB	32	entries
Arch. Reg	32	regs
PReg	64	regs
ALU	3	units
Mult	3	units
LD	3	units
ST	3	units
Branch	1	unit
ICache	256	bytes
LSQ	32	entries
DCache	256	bytes
BP	256	entries
BTB	32	entries
MULT	4	stages

Table 1: Critical design specifications.

2.2. Instruction Cache

The instruction cache (icache) is a direct mapped cache. When icache receives a request from the fetch stage, it checks if there is a cache hit. If that is the case, icache will output valid and the resulting data; otherwise, icache will schedule a request for memory. To reduce icache misses, we designed a prefetcher, which will be covered in the advanced features.

2.3. Dispatch Stage

The dispatch stage decodes raw instructions, requests physical registers from map table and freelist, and sends decoded packets to RS and ROB. Firstly, it receives three raw instructions from the fetch stage. Then, it passes the raw instructions into its embedded decoder, which outputs the registers, offsets, function units, branch information and error status of the instructions. After that, the dispatch stage dispatches instructions in FIFO order according to the free count of ROB and FUs. We ensure that our freelist never meets structural hazards so we don't need to worry about the free count of it.

Finally, our dispatch stage outputs virtual registers to map table to get their corresponding physical registers. It outputs the count of destination registers to freelist to request new physical registers. It outputs PC, FU and error status to ROB to prepare for the retire stage. It outputs incomplete RS entry, including PC, NPC and FU, to RS to prepare for further stages. It outputs the number of instructions it dispatches to fetch stage to fetch the instructions from the correct location in the next cycle.

2.4. Map Table

The map table maps virtual register to physical register. It uses virtual register as the index, and in each entry, it stores its corresponding physical register and its value ready bit. Since RV32IM ISA has 32 virtual registers, then the size of our map table is 32. Specifically, it has three functions.

Firstly, the dispatch stage sends the virtual registers (V_1 and V_2) to it and it outputs their corresponding physical registers (T_1 and T_2) and their ready bits to RS. Secondly, the dispatch stage sends the destination virtual registers

(V) to it and the freelist sends the newly allocated destination physical registers (T) to it. Then, it outputs the original destination physical registers (T_{old}) to ROB and replace them with the newly allocated ones (T). Thirdly, the complete stage sends completed destination physical registers to it and it set their corresponding ready bits as true. We also add data forwarding in the map table in case of register conflicting.

2.5. Freelist

The freelist is a critical module of register renaming. [2] It keeps track of all registers that has not been used in the physical register file or has been freed after an instruction has retired. Whenever an instruction with a destination register is dispatched, to avoid WAW hazard and WAR hazard, dispatch stage will require a new free physical register from the freelist. We have adjusted the number of physical registers so that the freelist will never encounter structural hazards. When an instruction with a destination register is retired, freelist will add the T_{old} of that instruction for it becomes a free register.

When we are squashing the pipeline, freelist will add all T_{old} registers from rob because these tags are incorrectly fetched.

2.6. Reservation Station Assembler

The Reservation Station assembler assembles data from different modules and gives unified packages to the RS. We designed this assembler to make the organization of modules nicer and cleaner. During our implementation, we found that data we need for building an entry in the RS comes from multiple modules including the freelist, the map table and the decoder. The source code of dispatch stage has already been very complicated since it contains the decoder. Additionally, the RS is also a complicated module. Our code and wires will be clearer if RS can receive packages ready for processing. Therefore, we created this module to assemble data into packages which only contains combinational logic to avoid extra cycles.

2.7. Reservation Station

At the dispatch stage, all instructions are stored in the reservation station, ready to be issued when the functional unit is available. When they enter the execute stage, they will be removed from the reservation station. Our reservation station entries are classified by functional units. We have 3 ALUs, 3 Mults, 3 LD, 3 ST and 1 branch unit. When a functional unit is available, a corresponding instruction could be dispatched to the reservation station.

Since we are doing a 3-way superscalar processor, we need to find an efficient way of choosing the instructions to issue in each cycle. For better performance, we used a priority selector to select the candidate instructions.

2.8. Reorder Buffer

Since we are executing instructions out-of-order, we need to reorder them to have the correct register values and branch results. Therefore, we implemented a reorder buffer that keeps all the instructions in the pipeline in order. On CDB broadcast, the corresponding instructions are marked to be complete, but only the first three of the reorder buffer are allowed to retire.

If a store is ready to retire, the reorder buffer should wait for the load store queue to finish storing. As discussed below, we only allow one store to retire at the same time, which could be a bottleneck of the performance. If a mispredicted branch is ready to retire, the pipeline should be squashed, and the map table and freelist are re-calculated.

2.9. Execute Stage

The execute stage supervises all functional units. Specifically, all functional units are built inside this module. It receives instructions and tags from the RS and values from the physical register file. Although at most 3 instructions will be given to this module each cycle, execute stage may send more than 3 results to the complete buffer so that we can reduce chances of encountering structural hazard. It is possible that more than 3 functional units finished their calculations in one cycle since they have different latencies.

If we are squashing the pipeline, all reset signals will be sent to functional units to terminate ongoing calculations.

2.10. Functional Units

We have five different types of functional units. They are the ALU (Arithmetic Logic Unit), the multiplication unit (MULT), the load unit (LD), the store unit (ST) and the branch unit. Each unit is able to perform calculations required by the corresponding group of instructions. ALU is responsible for basic arithmetic calculations. MULT unit performs integer multiplications. LD and ST unit calculates the address for loading and storing values. The branch unit determines whether a branch instruction is taken or not.

Our pipeline contains 3 ALUs, 3 MULTs, 3 LDs, 3 STs and 1 branch unit in total. The reason for this is that we are building a 3-way superscalar processor. The functional units will be capable of handling all instructions when the processor works at full speed (uses all 3 ways) without getting the pipeline slowed down. We only have 1 branch unit since it is safer and more straight forward to have only one branch instruction resolved at a time. It is more complicated and less safer for implementation to try to resolve more than one branches. The latency for MULT unit is longer than other FUs and the MULT unit is properly pipelined to reduce the clock period. In previous projects of EECS 470, it is found that more stages reduces the clock period but increases the total latency because it requires more cycles. However, we finally found that the critical path of the whole pipeline does not belong to execute stage. Therefore, it is efficient to set the number of stages of MULT unit smaller in our case. During testing process, we found that 4 is the smallest number of stages with which we could achieve a correct output. As a result, we set the number of stages as 4.

2.11. Physical Register File

The physical register file stores the value of all physical registers. We set its size as 64, which is the sum of the number of virtual registers and the size of ROB, so that we will never run out of physical registers. Since we are building a 3-way superscalar processor, we supports at most 6 read requests (3 T_1 and 3 T_2) and at most 3 write requests (3 T) concurrently. We also implemented data forwarding if receiving write and read requests of the same registers in the same cycle.

2.12. Complete Stage

The complete stage stores completed instructions in the complete buffer and broadcast their results to ROB, RS and physical regsiter file. In each cycle, it receives the results of newly complete instructions from the execute stage. Then it stores them into its embedded complete buffer, which is implemented using the head and tail method. After that, it outputs the results of at most three instructions to CDB based on the FIFO order from the complete buffer. The results in CDB are then broadcast to ROB for updating its retire ready bits, to RS and map table for updating its tag ready bit, to physical register file for writeback.

2.13. Load Store Queue

Since we are building an out-of-order processor, we need to reorder memory operations at retire time. Therefore, we need a load store queue to keep all the loads and stores in order.

We implemented a unified load store queue with forwarding. LSQ entries are allocated in the dispatch stage and removed in the retire stage. Each entry of the queue stores the load/store type, address, data and the reorder buffer index. When a load is the incomplete first load in the queue, we issue it by sending a signal to the reservation station. Note that this load doesn't wait all previous stores to complete. In the execute stage, load will send its resolved address to the load store queue. The load store queue will then figure out if the target data is stored in the queue, and send back the data to the execute stage. When load completes, the load store queue will mark it as complete, and the next load will be qualified for issuing.

When a store is the head of the reorder buffer, the load store queue will store its data to the data cache. If there is a cache miss, the load store queue will continue issuing the same request. After storing the data, the load store queue will notify the reorder buffer that the store has been retired.

2.14. Data Cache

For this project, we implemented blocking, write-allocate, write-back data cache (DCache). It supports only one load or store request at each time. To balance the temporal locality and the spatial locality, we design our DCache as 256-byte large and 4-way set-associative.

It receives load request from the execute stage and receives store request from the store retire module. The priority is given to store request if receiving both. Then, it indexes into the corresponding cache set and searches for the tag of given memory address. If it's a hit, it will update the lru bit correspondingly. To simplify the implementation, we use the bit pseudo lru (PLRU), where each cache way only stores one bit as lru.

If it's a miss, our DCache will find the first available slot in current cache set. If all slots are occupied, it will evict one slot according to the lru bit. If the evict one is dirty, our DCache will send store request to the memory first to store the data of the dirty slot. Otherwise, it will directly send the load request to the memory to get the data of the outstanding request address. our DCache blocks until getting the response from the memory. Therefore, requests should be sent continuously to check whether it can be served or not. After getting the data from the memory, our DCache will allocates a new slot for it. If the outstanding request is load, our DCache will output the corresponding value. If the outstanding request is store, our DCache will change the corresponding cache block and set its dirty bit as true.

2.15. Architecture Map Table and Branch Resolution

Branches are resolved at the retire stage. When we come into a branch misprediction at the retire stage, we will squash our pipeline and start to fetch from the branch target. When squashing, the map table is replaced by a copy of architecture map table, and the freelist is directly computed by the architecture map table. For future improvement, resolving the branch early at the execute stage would result in a performance gain.

3. Advanced Features for High Performance

After the code deadline, we fixed synthesis bugs related to the Next-line Prefetcher and GShare Branch Predictor and optimize the cycle length from 19ns (basic version) to 17.5ns. The advanced version can be found in the same project repository on the branch `advanced_manual_fix`.

This section describes the advanced features we chose to implement. They are the Next-line Prefetcher, the Gshare Branch Predictor, the Branch Target Buffer and the Instruction Pre-decoder.

3.1. Next-line Prefetcher

On an icache miss, icache will send a request to memory, and allow for multiples cycles for memory to reply. The icache isn't doing anything in the waiting process, which is highly inefficient. Therefore, we came up with the design of next-line prefetch.

The idea of next-line prefetch is that when icache has received a valid tag for the current fetch PC that evokes a cache miss, it could issue a request of the next-line after the fetch PC. If the program is running in order, soon the processor would ask for this next-line. We save much time here since the next-line request is already sent or even finished, with the latter resulting in a cache hit.

For example, suppose fetch stage is requesting PC = 0 from icache, which is a cache miss. Icache will then send a request to memory, and get a valid response tag. While icache is waiting for the data at PC = 0, it continues to request for PC = 8; and when PC = 8 has a valid tag, icache will request for PC = 16 ... When the data for PC = 0 is returned and fetch stage starts to request PC = 8, icache has already asked memory and may get the data in store. In this case, icache won't issue another request for PC = 8, and could provide the data to fetch stage much faster.

In our implementation, we kept the original functionality of the icache on a cache miss. We added a prefetcher module to figure out when to issue a prefetch request to memory. We won't prefetch if the current fetch PC results in a cache hit or if the current fetch PC is requesting to memory.

Now that we have multiple tags waiting for their data, we choose to store the tag to address relationship in a tag-to-address map. When memory returns data with a specific tag, we can use the tag to find the address and update icache correspondingly.

Whenever the prefetcher sees a valid memory response, it would record the tag and address in the tag-to-address map at the next positive clock edge. It would also start to prefetch the line after the recorded one. To avoid eliminating useful entries from icache, we limit the maximum prefetch count by a parameter. Different parameter choices have various performances, as we will discuss in the performance section.

3.2. Gshare Branch Predictor

The main goal of branch predictor (BP) is that given an PC, it can output taken or not taken. If predicting correctly, we can then fetch the correct PC in the next cycle, which saves the cycles needed for branch squash.

In this project, we use Gshare BP [1] as it can achieve relatively high accuracy while requiring small memory space. Our Gshare BP consists of two parts: 8-bit long branch history register (BHR) and 256-entry large pattern history table (PHT). The BHR records the directions, taken or not taken, of the last 8 resolved branches. 1 stands for taken and 0 stands for not taken. The PHT is used for making predictions, where in each entry, it stores a two-bit saturation counter.

When our Gshare BP receives the results of new resolved branch from the execute stage, it will firstly shift BHR one bit and update its lowest bit. Secondly, it will xor the resolved branch PC with BHR and use the result to index into the PHT. Then it will update the corresponding two-bit counter using the resolved branch direction. To make prediction, similarly, it will xor branch PC with BHR and indexing into PHT to get the branch prediction.

3.3. Branch Target Buffer

The branch target buffer, or BTB, is designed under the consideration that a branch instruction located in a certain PC, if ran multiple times, is likely to branch to the same target. Situations like this frequently appear in loops. We can make the branch prediction faster if we predict the correct target in advance. Typically, BTB allows our processor to remember the target of resolved branch instructions. When it encounters an instruction with the same PC as one of the recorded ones and the branch is predicted by BP as taken, BTB gives out the last resolved target of this branch instruction. The BTB is structured out of a direct-mapped cache with 32 cachelines (BTB entries). It keeps the [6:2] bits of PCs as index of entries and the [31:7] bits of PCs as tags. If a branch is resolved, BTB overwrites the entry and the target according to its PC. When we need to predict new branches, BTB searches the corresponding entry and compare the tags. If it is a hit, BTB gives out the target. If it is a miss, BTB gives out nothing. A miss case will be similar to the case where the BP gives not-taken.

3.4. Target PC Logic and Instruction Pre-decoder

The Target PC Logic module (Figure 2) manages the input and output of BP and BTB with an internal instruction pre-decoder. When new instructions are fetched, the pre-decoder performs a simple decoding process to determine whether there are branches. The instructions are also sent to BP and BTB to get branch direction and branch target respectively. If the instructions are branches, BP predicts taken and BTB gets a hit, Target PC Logic will output the BTB target as the predicted branch target. Otherwise, it will output PC + 4, i.e. the next sequential PC, as the predicted target. By performing branch detection in advance, the pre-decoder increases the accuracy of branch prediction.

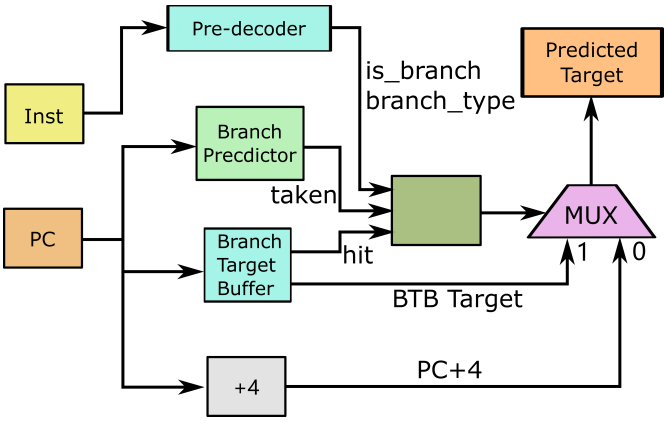

Figure 2: Target PC Logic Design

4. Testing Methodology

To test and debug our processor, we used multiple testing methodologies. We wrote a testbench for each module with simple cases, edge cases, and random cases. After passing the individual testbenches, we connected the related modules and wrote testbenches for the combined block. For example, branch predictor, branch target buffer, and our target pc logic are connected and tested. Then we joined all modules in our pipeline and tested them with .s and .c programs. We first wrote our testing programs before experimenting with the course-provided programs, as attached in the appendix.

We first use the rv32_gcat_simple.s to test the basic pipeline, the multiplication execution, and the halting instruction. Then rv32_load.s is used to check all types of loads; likewise, rv32_store.s is used to test all kinds of stores. We also wrote some .c files: swap.c is used to test basic register storing, and lshift.c is used to test branch predic-

tion and branch squash.

To enable auto testing and collect data for performance analysis, we wrote our script auto_test.sh, as attached in the appendix. When running the script with the base and test directory name, it will run the base project and the test project, compare the result to see if the cases are passed and collect information in files.

5. Performance Analysis

This section covers our effort in performance analysis including our test strategy, performance with basic design, performance with instruction prefetch and performance with Target PC Logic. In general, our basic design achieves 19ns cycle period, and our advance design with prefetcher and TPL achieves 17.5ns cycle period.

5.1. Test strategy

We divided our project into multiple tasks. Each member implemented different modules and applied both simulation test and synthesis test to them. Some complicated modules were completed by more than one group members.

Then we assembled modules which works together and test them. For example, we assembled dispatch stage and decoder, execute stage and functional units, and Target PC Logic and BP/BTB. Testing modules as small groups may simplify the testing of the whole pipeline.

Finally we tried to test the pipeline as a whole. We added multiple output ports and functions to the testbench so that we were able to check the status of critical modules in each cycle. To avoid infinite loops, we limited the number of cycles according to the number of instructions of the testcases.

We wrote some basic testcases ourselves to check whether the pipeline works correctly. To make debugging easier, we also wrote smaller programs with the same function as the given programs.

The criteria of evaluating the performance of our processor follows the Iron Law:

$$Performances = \frac{\#Instructions}{Program} \times \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle}$$

We also use time per instruction:

$$\frac{Time}{Instruction} = \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle} = CPI \times ClockPeriod$$

5.2. Performance with basic design

Figure 3 describes the performance differences between in-order pipeline in project 3 and our out-of-order pipeline. For project 3, the clock period should be larger than one memory access latency. Thus, we assume that the clock period is the same as final project, which is 100ns.

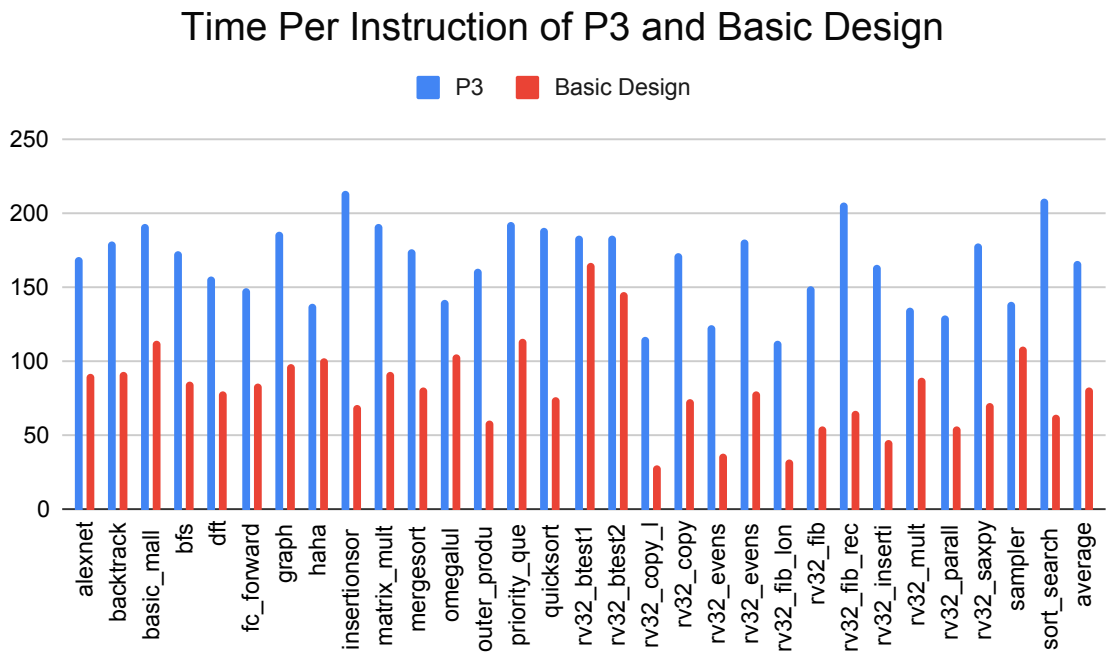

Figure 3: Comparing TPI of P3 in-order pipeline design and basic design on test cases

This figure shows that the average time per instruction for our basic design is significantly smaller than for the P3 design in all cases. However, for branch tests rv32_btest1.s and rv32_btest2.s, the performance gain is relatively small. This is because our basic design always predicts branch not taken, and it will squash at the retirement stage if a branch misprediction occurs.

Figure 4 shows the performance differences between our basic out-of-order pipeline and the advanced version with branch predictor and prefetcher.

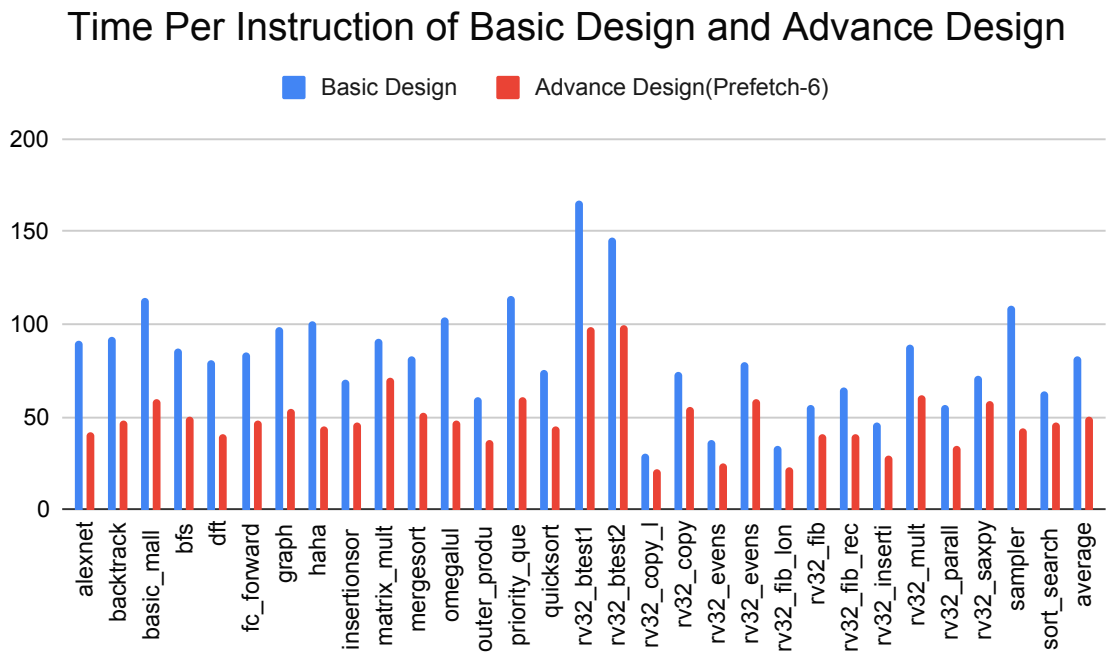

Figure 4: Comparing TPI of basic design and advanced design on test cases

This figure shows that the average time per instruction for our advanced design is smaller than for the basic design in all cases. The performance gain is more remarkable for longer cases like alexnet, where the prefetcher prefetches more valid instructions. For cases with many branches, the performance gain is also larger since our branch predictor has more correct predictions.

5.3. Performance with Target PC Logic

Figure 5 shows the CPI comparison for all test programs between our basic out-of-order pipeline (predicting not-taken) and the one with TPL. In smaller .s cases, the CPI varies, but the design with TPL performance better in more cases. In larger .c cases, the CPI for the basic design is slightly larger than that for the design with TPL. Therefore, our branch predictor and branch target buffer in TPL do have a small performance gain.

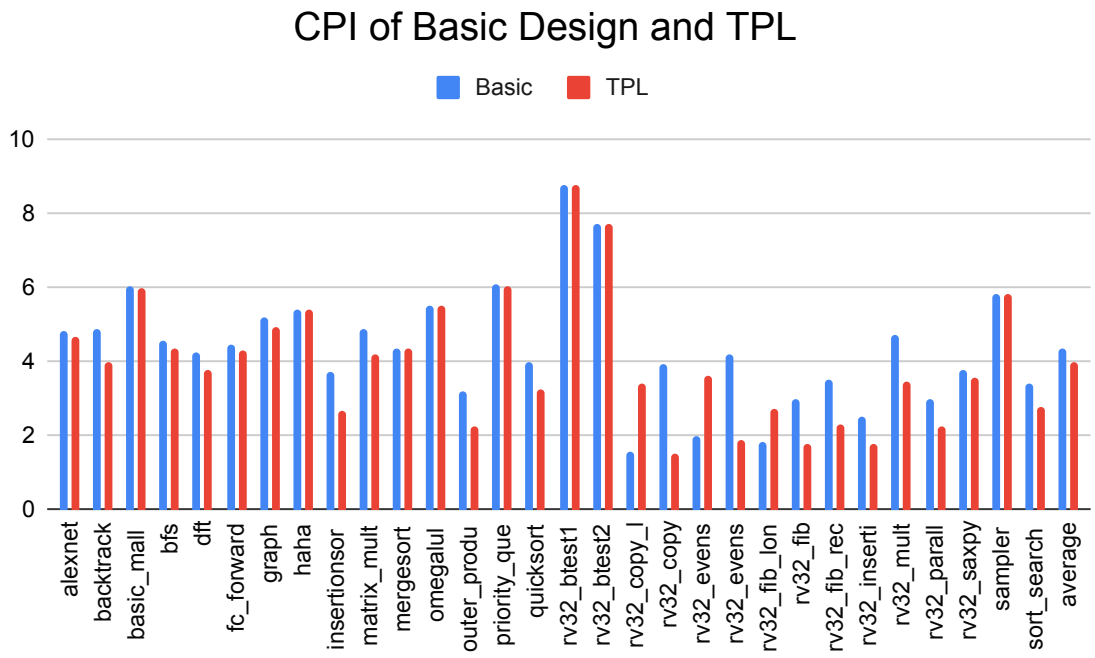

Figure 5: Comparing CPI of basic design and design with TPL on test cases

To evaluate the effectiveness of our TPL, we also measure the branch MPKI with and without GShare predictor and BTB. As shown in Figure 6, our TPL can reduce Branch MPKI in most cases.
For our target pc logic (with branch predictor and branch target buffer), we conclude that it provides a small performance gain. Further optimization, such as changing history register's size and set associative BTB with a better replacement policy, could be done for better performance.

5.4. Performance with instruction prefetch

Figure 7 shows the CPI comparison for .s test programs between the design with TPL and the design with TPL and a 6-line prefetcher. (We chose 6-line after comparing several choices, as we will discuss below.) As shown in the graph, the design with the prefetcher performs better in all cases with a significantly smaller CPI. Therefore, the design of perfetcher provides a remarkable performance gain.

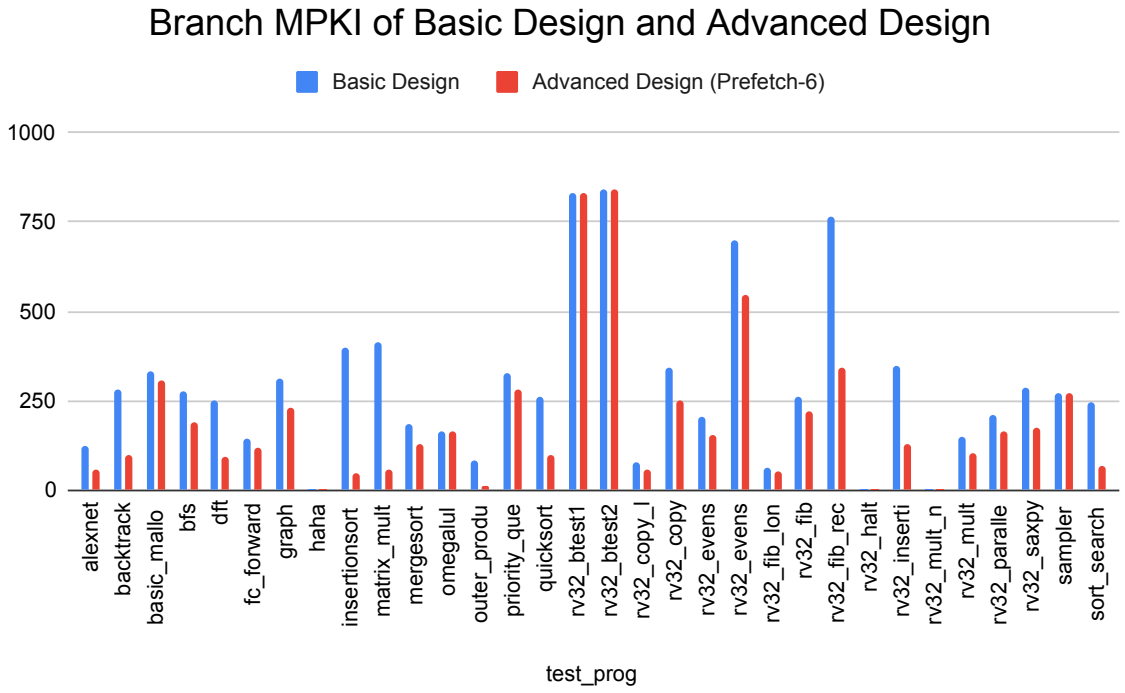

Figure 6: Comparing Branch MPKI of with and without GShare predictor and BTB

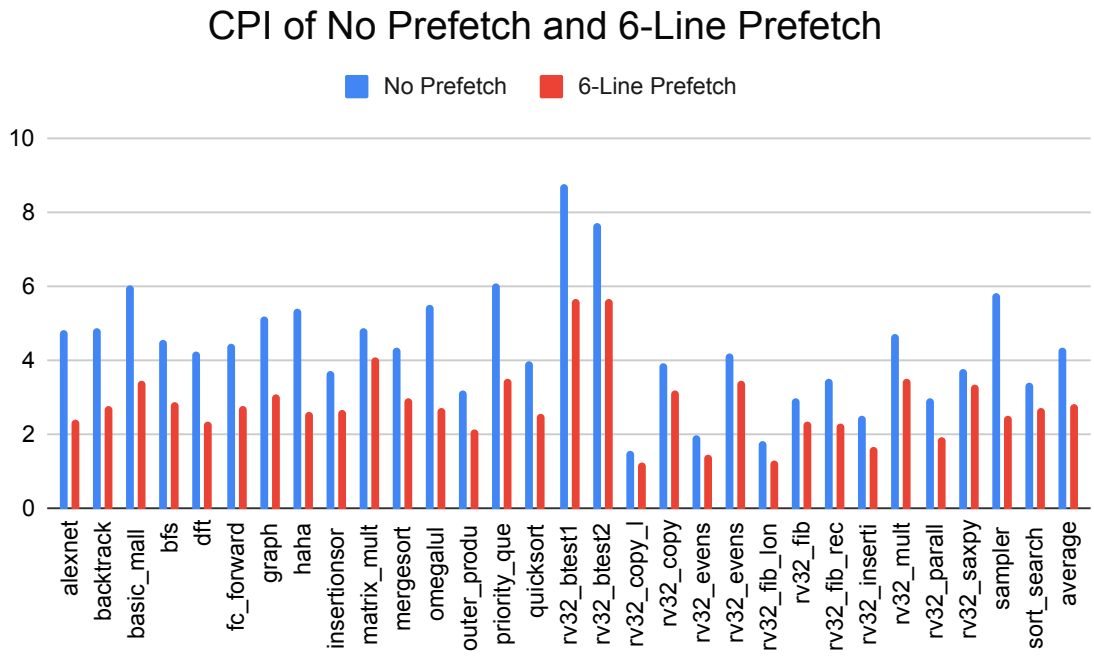

Figure 7: Comparing CPI of basic design and design with TPL and prefetch on test cases

Figure 8 shows the CPI comparison for .s test programs with different numbers of instruction prefetch. As shown in the chart, in rv32_branch.s, rv32_btest1.s, rv32_btest2.s, 3 lines prefetch performs better than 6 or 9 lines prefetch. This is because there are many branches in those three cases, presenting less spacial locality. Therefore, if more next-lines are prefetched into icache, some useful entries might be eliminated.

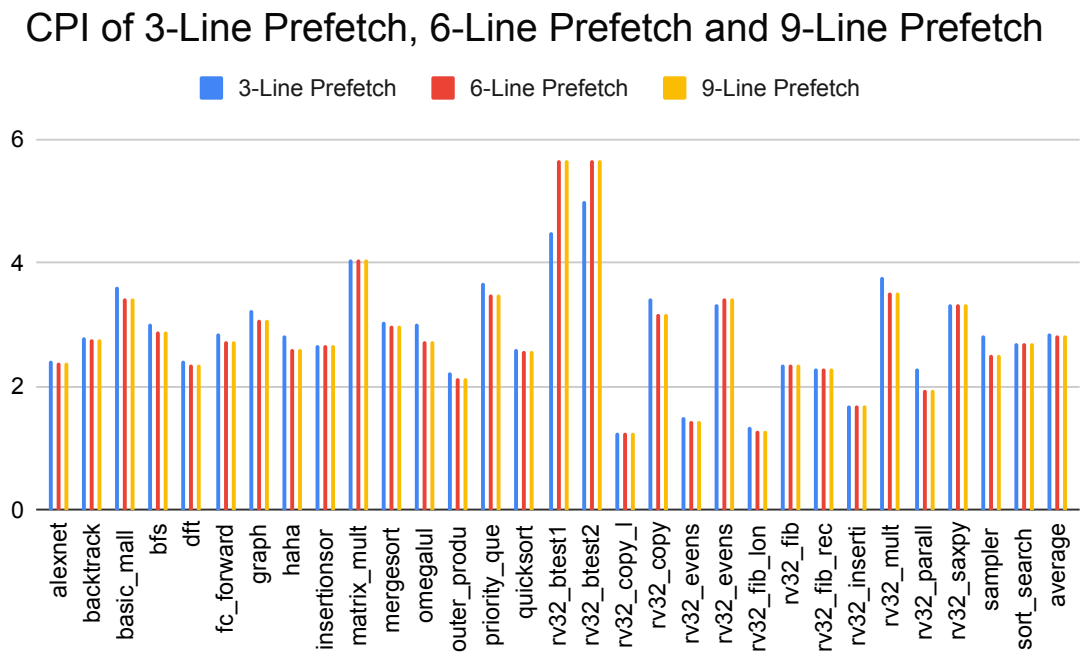

Figure 8: Comparing CPI of different lines of instruction prefetch

For most other cases, 6 lines and 9 lines prefetch performs slightly better with a smaller CPI. This is because when the programs have strong spacial locality, prefetching next-lines to icache will remarkably reduce icache misses. We calculated the average CPI and chose 6-line prefetch as the optimal one.

To evaluate the effectiveness of our instruction prefetcher, we measure number of cycles when the instruction buffer is empty, and divide it by the total number of cycles to get instruction buffer empty rate. In our design, reorder buffer can dispatch instruction from instruction buffer as long as instruction buffer is not empty and reorder buffer is not full, so this value can also be interpreted as reorder buffer empty rate. As shown in Figure 9, our instruction prefetcher reduces the number of cycles when instruction buffer is empty significantly, so it successfully break the performance bottleneck of fetching instructions.

6. Team Logistics

Yuqing Qiu(20%): dispatch, issue, execute, complete, branch predictor, prefetch, data cache
Shixin Song(20%): fetch, prefetch, target pc logic, reservation station, reorder buffer, map table, freelist, synthesis debug
Zesheng Yu(20%): dispatch, execute, map table, freelist, branch resolving, synthesis debug
Chenyan Zhang(20%): fetch, prefetch, target pc logic, reservation station, load store queue
Zimeng Zhang(20%): dispatch, execute, branch target buffer, function units

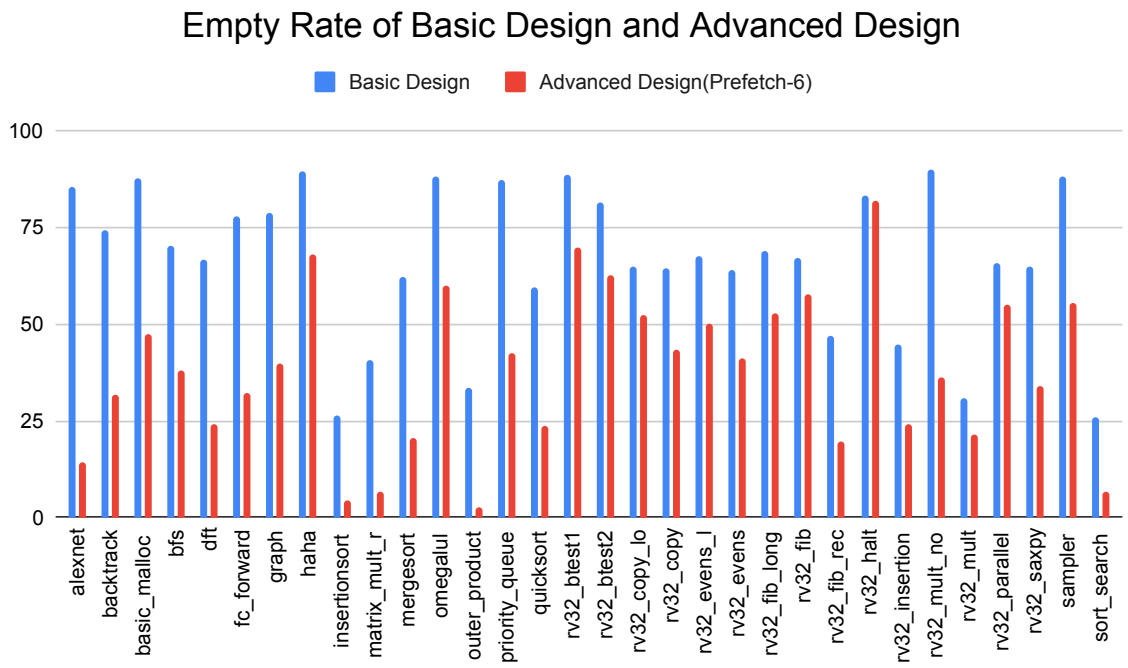

Figure 9: Comparing Instruction Buffer Empty Rate of with and without Next-Line instruction prefetcher

7. Future improvements

7.1. Non-Blocking Data Cache and separated LSQ

A certain amount of time of running a program is taken by fetching data from memory since accessing memory access is much slower than accessing reg files. In our design, we chose to implement a blocking data cache to guarantee a steady version before submission. Non-blocking caches allow certain number of missed requests and check whether they come from the same block.

Implementing non-blocking cache will also allow us to retire multiple store instructions and request data for multiple load instructions. We may separate our LSQ to LQ and SQ to support this feature. Adding non-blocking feature to our data cache and separate our LSQ might be a good method to further increase the performance.

7.2. Early branch resolution

At present, a branch instruction will be resolved when it retires. However, we can know whether the branch is taken or not earlier than that just after execute stage. We might let execute stage start the roll back process so that we can resolve the branch faster. [5]

References

[1] MCFARLING S. Combining branch predictors, technical report. Digital Equipment Corporation WRL TN-36, 1993.

[2] D. Sima. The design space of register renaming techniques. IEEE Micro, 20(5):70–83, 2000.

[3] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. Proceedings of the IEEE, 83(12):1609–1624, 1995.

[4] K.C. Yeager. The mips r10000 superscalar microprocessor. IEEE Micro, 16(2):28–41, 1996.

[5] Peng Zhou, Soner Önder, and Steve Carr. Fast branch misprediction recovery in out-of-order superscalar processors. Proceedings of the 19th annual international conference on Supercomputing - ICS '05, 2005.

8. Appendix

.1. Our auto-testing script

This is the script we used for auto-running all public test cases.

```
#!/bin/bash

if [ -z "$3" ]; then
  echo "Usage: $0 BASEREPO TESTREPO OUTPUT_FILENAME"
  exit 1
fi

BASEREPO=$1
TESTREPO=$2
RESULTDIR=$(pwd)/test_result
OUTPUT_FILE=$(pwd)/$3
mkdir -p $RESULTDIR

function generate_outputs {
  repo=$1
  prefix=$2
  make_option=$3
  output_option=$4
  echo $repo
  mkdir -p ${RESULTDIR}/${prefix}
  cd "${repo}" || exit 1
  for file in test_progs/*.s; do
    file=$(echo $file | cut -d'.' -f1)
    filename=$(basename $file)
    echo "Assembling $file"
    make assembly SOURCE=${file}.s > /dev/null
    echo "Running $file"
    echo $make_option
    echo $output_option
    make $make_option > /dev/null
    echo "Saving $file output"
    grep "@@@" ${output_option}program.out >
    ↪  "${RESULTDIR}/${prefix}/${filename}.program.out"
    grep "CPI" ${output_option}program.out >
    ↪  "${RESULTDIR}/${prefix}/${filename}.cpi.out"
    grep "ns total time to execute" ${output_option}program.out >
    ↪  "${RESULTDIR}/${prefix}/${filename}.time.out"
    mv pipeline.out "${RESULTDIR}/${prefix}/${filename}.pipeline.out"
    mv writeback.out "${RESULTDIR}/${prefix}/${filename}.writeback.out"
    if [ -f "performance.out" ]; then
        mv performance.out "${RESULTDIR}/${prefix}/${filename}.performance.out"
    fi
  done
  for file in test_progs/*.c; do
#      if [[ $file -ef "test_progs/alexnet.c" ]]; then
#        continue
#      fi
      file=$(echo $file | cut -d'.' -f1)
      filename=$(basename $file)
      echo "Compiling $file"
      make program SOURCE=${file}.c > /dev/null
      echo "Running $file"
```

```
      make $make_option > /dev/null
      echo "Saving $file output"
      grep "@@@" ${output_option}program.out >
      ↪  "${RESULTDIR}/${prefix}/${filename}.program.out"
      grep "CPI" ${output_option}program.out >
      ↪  "${RESULTDIR}/${prefix}/${filename}.cpi.out"
      grep "ns total time to execute" ${output_option}program.out >
      ↪  "${RESULTDIR}/${prefix}/${filename}.time.out"
      mv pipeline.out "${RESULTDIR}/${prefix}/${filename}.pipeline.out"
      mv writeback.out "${RESULTDIR}/${prefix}/${filename}.writeback.out"
      if [ -f "performance.out" ]; then
          mv performance.out "${RESULTDIR}/${prefix}/${filename}.performance.out"
      fi
    done
  cd "$RESULTDIR" || exit 1
}

function compare_result {
  correct_dir=${RESULTDIR}/$1
  tested_dir=${RESULTDIR}/$2
  pass_count=$((0))
      fail_count=$((0))
      total=$((0))
  echo "test_prog,time,cycle,instr,cpi,retire,branch_squash,frontend_stall" >
  ↪  "$OUTPUT_FILE"
  for file in $correct_dir/*.program.out; do
    testname=$(basename $file .program.out)
    echo "Compare result for ${testname}"
    diff $correct_dir/${testname}.program.out $tested_dir/${testname}.program.out >
    ↪  /dev/null
    status1=$?
    diff $correct_dir/${testname}.writeback.out $tested_dir/${testname}.writeback.out >
    ↪  /dev/null
    status2=$?
    if [[ "$status1" -eq "0" ]] && [[ "$status2" -eq "0" ]]; then
      echo "${testname} PASSED"
      pass_count=$(($pass_count + 1))
    else
      echo "$0: Test $testname FAILED"
      fail_count=$(($fail_count + 1))
    fi
    echo "BASE PERF `cat $correct_dir/${testname}.cpi.out`"
    echo "BASE PERF `cat $correct_dir/${testname}.time.out`"
    echo "TEST PERF `cat $tested_dir/${testname}.cpi.out`"
    echo "TEST PERF `cat $tested_dir/${testname}.time.out`"
    echo ""
    total=$(($total + 1))
    time_ns=`sed -nr 's/@@  ([0-9]+\.?[0-9]*) ns total time to execute/\1/p'
    ↪  $tested_dir/${testname}.time.out`
    cycles=`sed -nr 's_@@  ([0-9]+) cycles / ([0-9]+) instrs = ([0-9]*\.?[0-9]+)
    ↪  CPI_\1_p' $tested_dir/${testname}.cpi.out`
    instrs=`sed -nr 's_@@  ([0-9]+) cycles / ([0-9]+) instrs = ([0-9]*\.?[0-9]+)
    ↪  CPI_\2_p' $tested_dir/${testname}.cpi.out`
    cpi=`sed -nr 's_@@  ([0-9]+) cycles / ([0-9]+) instrs = ([0-9]*\.?[0-9]+) CPI_\3_p'
    ↪  $tested_dir/${testname}.cpi.out`
    if [ -f "$tested_dir/${testname}.performance.out" ]; then
```

```
      retire=`cat $tested_dir/${testname}.performance.out | tr -s ' ' | sed -nr
      ↪  's/Cycle: ([0-9]+) retire: ([0-9]+) branch_squash: ([0-9]+) frontend_stall:
      ↪  ([0-9]+)/\2/p'`
      branch_squash=`cat $tested_dir/${testname}.performance.out | tr -s ' ' | sed -nr
      ↪  's/Cycle: ([0-9]+) retire: ([0-9]+) branch_squash: ([0-9]+) frontend_stall:
      ↪  ([0-9]+)/\3/p'`
      frontend_stall=`cat $tested_dir/${testname}.performance.out | tr -s ' ' | sed -nr
      ↪  's/Cycle: ([0-9]+) retire: ([0-9]+) branch_squash: ([0-9]+) frontend_stall:
      ↪  ([0-9]+)/\4/p'`
    else
      retire=""
      branch_squash=""
      frontend_stall=""
    fi
    echo
    ↪  "${testname},${time_ns},${cycles},${instrs},${cpi},${retire},${branch_squash},${frontend_stall}
    ↪  >> "$OUTPUT_FILE"
  done
  echo ""
  echo "PASSED $pass_count/$total tests ($fail_count failures)."
}

#generate_outputs $BASEREPO "base" "" ""
generate_outputs $TESTREPO "test" "" "sim_"
compare_result "base" "test"
```

.2. Our testcases

rv32_gcat_simple.s

```
li          x1, 0x5
li          x2, 0x6
mul         x3, x1, x2
wfi
```

rv32_load.s

```
li        x2, 4
lw        x4, 8(x0)
lh        x4, 8(x2)
lb        x4, 8(x2)
lhu        x4, 8(x2)
lbu        x4, 8(x2)
wfi
```

rv32_store.s

```
li        x2, 2048
li        x4, 1231894
sw        x4, 8(x2)
sh        x4, 16(x2)
nop
```

```
nop
nop
nop
li        x1, 1
wfi
```

swap.c

```c
#include<stdio.h>
int main() {
    float opa, opb, temp;
    opa = 0;
    opb = 1;
    temp = opa;
    opb = opa;
    opa = temp;
    return 0;
}
```

lshift.c

```c
#include<stdio.h>
int main() {
    float ops[100];
    float temp;
    for (i = 0; i < 100; i++) {
        ops[i] = i;
    }
    temp = ops[0];
    for (i = 0; i < 100; i++) {
        if (i == 99) {
            ops[i] = temp;
        } else {
            ops[i] = ops[i+1];
        }
    }
    return 0;
}
```