DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF MICHIGAN, ANN ARBOR

A R10K based RISC-V Superscalar Architecture

2022 Fall Group 6 Final Project Report for EECS 470

Professor: Ron Dreslinski
Group member: Jianping Shen, Xinchao Zha, Junyi Cao, Anchen Xue, Guangze Zu, Ning Liang

1 Project Overview

The project implements a 3-way superscalar processor supporting RISC-V Instruction Set Architecture (ISA) with integer multiplication extension and using a MIPS R10K style architecture. This design intends to maximize instruction level parallelism by using real register renaming and out of order (OoO) execution. In order to improve the computing performance, several different optimizations were attempted to achieve a balance between cycle per instruction (CPI) and clock period. This report includes top level and each module design overview, methodology for testing and verification, testing results and performance analysis, individual contribution and conclusion regarding potential improvement or optimization as well as broader societal impacts.

2 Design Overview

The top-level design is shown in Figure 2.1 which is based on the MIPS R10K architecture. There is no specific pipeline structure but a normal ALU instruction can take the minimum of 6 cycles from being fetched to retirement. This is an OoO architecture which means that all instructions can be executed OoO, while they are fetched, dispatched and retired in order. It usually takes more depending how full each buffer is and how many unresolved instructions in front of it. This is a 3-way superscalar design so it can at most dispatch, issue, execute, complete and retire at most 3 instructions in a single cycle. True register renaming is used to resolve WAR, WAW dependency and improve instruction level parallelism. Early branch recovery is also implemented with a 4-entry branch stack, so mispredicted branches can be recovered in as fast as one cycle. The load and store queue modules ensure that load and store instructions are executed in order while their address and data may be resolved OoO. Once a terminated instruction WFI is retired, the error status will set the WFI_detected signal and wait for the store queue to be flushed out. When all data in the store queue are stored in D-cache, the error status will be set as WFI, which lets the testbench know that execution is terminated.

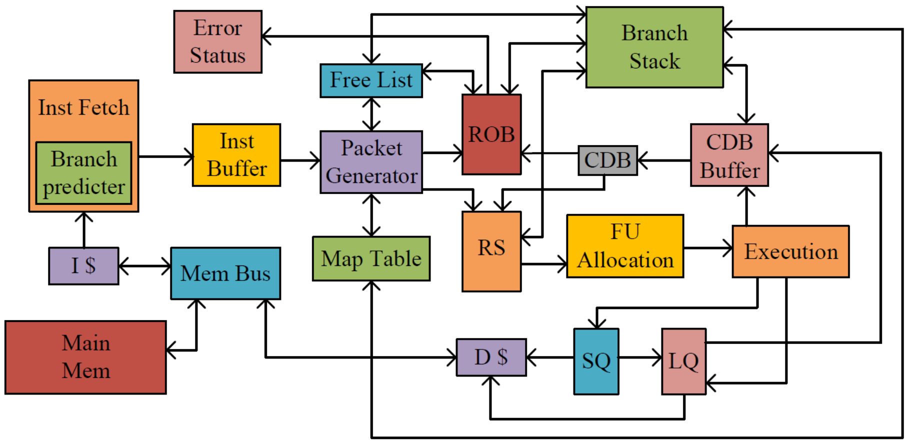


Figure 2.1 R10K top level design

2.1 Specification

ROB	32 entries
RS	8 entries
Freelist	32 depth FIFO
Maptable	32 entries
Inst buffer	8 entries FIFO
CDB buffer	32 entries FIFO
I-cache	256 byte direct map
D-cache	256 byte 4-way associate
Function units	3 ALU and 1 8-stage multiplier
LSQ	16 entries each

2.1.1 Advanced Features

Advanced features	Integrated in the final submission
3-way superscalar	Yes
Early branch resolution	Yes
Non-blocking D-cache	Yes
Multiport D-cache	Yes
4-way associativity D-cache	Yes
Data forwarding	Yes
Prefetch in I-cache	Yes

2.2 Module design

2.2.1 Instruction Fetch, I-Cache and Instruction Buffer

These modules make up the IF stage together.

2.2.1.1 Instruction Fetch

The instruction fetch module fetches data from I-cache based on the next PC, parses it into aligned instructions and sends it into the instruction buffer. Instruction fetch module can at most send four instructions into the instruction buffer. It is also responsible for assigning branch stack entry for predicted branch and branch mask for instruction after predicted branch. Once the data is fetched from I-cache, the instruction fetch module will align and predecode the data.

Predecode module inside the instruction fetch module will determine whether the instruction is a conditional or unconditional branch and calculates potential branch address depending on the imm and current PC. If a branch instruction is detected, the instruction fetch will make it the last instruction sending to the buffer in this cycle. If this is a conditional branch, the branch predictor will decide the branch direction and assign a branch entry for this prediction. If the branch stack is full or it is a jalr instruction, no prediction will be made and the instruction fetch module will stall until the branch is resolved. There is no branch target buffer, because most branch targets can be calculated by current PC and imm directly except jalr instruction.

2.2.1.2 I-Cache

The I-cache is a 256-Byte direct map cache with instruction prefetch function. The I-cache can at most prefetch 16 cache lines from memory in advance of the current requested PC fetched by the processor. The prefetch function effectively eliminates continually cold miss. During a cache hit, I-cache can send one line or two lines to the instruction fetch module depending on whether the second line is also a hit, so at most 4 instructions can be fetched in a cycle.

2.2.1.3 Instruction Buffer

The instruction Buffer is used to store the instruction fetched from I-cache, the main function of it is to store the fetched instruction which cannot be sent to execute in the same cycle. Instruction Buffer smooths the procedure between fetch and execute, it can be used to hide the latency of i-cache miss. All the instructions in the instruction buffer will be executed except there is branch misprediction. When branch mis-prediction happens, flush the whole instruction buffer.

2.2.2 Instruction Decoder and Packet Generate

Instruction decoder and packet generation module together form the Dispatch stage in the R10K design.

2.2.2.1 Instruction Decoder

The Instruction Decoder is used to decode the valid instruction sent from the instruction and send the decoded instruction to Packet Generate.

2.2.2.2 Packet Generate

The main function for Packet Generate is resource allocation. It takes the decoded instructions from Decoder. It will consider the structural hazard from ROB, RS, Freelist, Load Queue and Store Queue. Through comparing the structural hazard from different modules and the number of valid instructions which are sent from the instruction buffer, it will determine how many instructions can be sent to the pipeline in the same cycle. The Packet Generate also generates the packets to RS, ROB, LQ, SQ of corresponding instructions which have been sent to the pipeline.

2.2.3 Reorder Buffer

The reorder buffer has 32 entries which is implemented by a circular buffer. It has a head and a tail to record the instructions which are dispatched but not retired. When an instruction is dispatched, it will reserve an entry in ROB. The ROB will also receive the data from the CDB buffer, and mark corresponding ROB entries as completed. In one cycle, the ROB can retire at most three instructions. When a branch is mispredicted, the tail of ROB will be reset from the branch stack.

2.2.4 Reservation Station

The Reservation station has two stages, the first one is issued from RS. In this stage,RS will choose the oldest instructions which are ready for both operands and issues to the execute stage. In this stage, the structural hazard needs to be considered because in our design we only have one multiply function unit and one branch function unit. The second stage is compress and wake-up. In this stage, it will compress the older instructions to the top of the RS and reserve the new instructions at the following which can ensure that the older instruction can be issued first from the RS to alleviate the influence of RAW. For the wake-up part, RS will receive the wake-up signal from both CDB bus and back to back forwarding. The back to back forwarding means that when an instruction which needs to write back to physical register A has been issued from RS, in the same cycle, make the instructions in RS which has physical register A to be ready in advance.

2.2.5 Freelist, Maptable and Physical Register File

Freelist, maptable and physical register files make up the real register renaming system which is very essential for R10K architecture.

2.2.5.1 Freelist

Freelist is a FIFO with 32 entries recording physical registers which are ready to be renamed. It is connected to the ROB, in which registers to be freed are sent to the freelist, and to the packet generator, in which the top three available physical registers will be packed into a ROB_IN_PACKET, which will then be sent to the ROB for them to issue new instructions. It is also connected to the branch stack so that the freelist could send its state to it for it to take snapshot and recovery to the precise state during branch recovery.

2.2.5.2 Maptable

Maptable links logical registers to their renaming physical register. It also holds a valid bit representing whether data in the logical register is ready or not. The valid bit will be cleared when a new physical register is renamed to the logical register and set when detected as a completed signal broadcasted by CDB. Maptable also sends its current state to the branch stack for making copy and recovery to the state sent by the branch stack during branch recovery.

2.2.5.3 Physical Register File

In our design R10K system, one big physical register file holds all data, which means there is no copying and pasting. The total size of this register file is 64*32 bits. The register file implements six reading ports and three writing ports due to requirement.

2.2.6 Execution Stage

The execution stage has two parts. The first part takes the instruction issued from the RS, distinguishing the type of instructions, and sends to the corresponding function unit. The second part uses the corresponding function unit to calculate the result. In this part, the data forwarding is also implemented for each functional unit. After the result has been calculated in the execution stage, it will be sent to the CDB buffer for broadcasting and the early branch resolution will be supported in all function units including alu, multiplier and branch function unit.

2.2.7 Load Store Queue and Data Cache

2.2.7.1 Load Store Queue

The LSQ unit is divided as split LQ and SQ. When load and store instructions are dispatched, the new entries will be reserved in LQ and SQ. In LQ, it will also store the current SQ tail for each load instruction which is challenging when multiple load and store instructions are dispatched in the same cycle. For SQ, the main responsibility of it is to forward value for load instruction and issue the store instruction to D-cache when the store instruction is retired from ROB. SQ supports the fully forwarding which means that load instruction can forward a byte/half-word/word from multiple LQ entries. For LQ, it will select and send at most three load instructions to SQ for forwarding, at most three load instructions to D-cache and at most three load instructions to CDB buffer.

2.2.7.2 D-cache

Our D-cache is a 4-way set associative, non-blocking D-cache with three read ports and two write ports. When load or store instructions have cache-miss in D-cache, it will reserve a new entry in MSHR according to their public address. The MSHR has eight entries and for each MSHR entry there is a table to record the load and store instructions which are cache-miss for this address. Because there could be three load instructions and two store instructions can be sended to the D-cache at the same time, the load instruction will always have the higher priority to reserve new MSHR entries. We also use the pseudo-LRU as our evict policy for one cache set. When a dirty cache line is evicted, it will have the top priority to write into memory.

2.2.8 Common Data Bus

The Common Data Bus (CDB) is responsible for broadcasting the number of the physical register whose value is ready. It is noted that a CDB buffer is inserted in the CDB module. The reason is that the number of the function units is larger than the superscalar way number, which means there may be more than three function units that completes the execution in one cycle, but only three physical register numbers can be broadcasted. The CDB buffer is used to handle such circumstances. It stores the surplus physical register numbers and pop them out in later broadcasts.

2.2.9 Branch Predictor and Branch Recovery

2.2.9.1 Branch Predictor

Branch predictor locates in the instruction fetch module. When a branch prediction is needed, the branch predictor will assign a branch stack entry to the branch instruction and make a prediction. There are 64 local branch history tables (BHT) and a 64-entry pattern history table (PHT) with 2-bit saturation, so it uses per PC branch history and global pattern history. An average of 81.18% prediction accuracy is achieved by this design.

2.2.9.2 Branch Recovery

The early branch recovery is realized by using branch stack. When a predicted branch instruction is dispatched, the branch stack will take a copy of the current state of maptable, freelist, ROB tail and LSQ tail. When a branch is resolved, branch stack will check whether a branch recovery is needed. If branch recovery is needed, branch stack will send the recovery signal and the copy of the state to maptable, freelist, ROB and LSQ. RS, CDB buffer and execution stage will need to flush certain instructions inside based on their branch mask and instruction buffer should clear all its entries. Branch stack also needs to send the correct branch PC to instruction fetch. If the branch was correctly predicted, branch stack will broadcast a branch correct signal and release the stack entry. Instruction buffer, RS, execution pipeline, and LSQ may need to clear the branch stack bit on the instructions’ branch mask. The branch predictor may also need to update its entries when a branch is resolved.

3 Verification and Debug Methodology

3.1 Verification Methodology

3.1.1 Unit test

For each individual module, unit testbench with hard code driver and ground truth are used to test that it can perform its basic function. Each module was also synthesized individually to ensure that it was synthesizable and can pass 10 ns STA check. For some modules that are highly related to each other, they can perform some combined tests before assembling them into the top level. For example, I-cache, instruction fetch and instruction buffer were tested together. In I-cache and D-cache testbench, where ground truth can be easily obtained, the random function was used to generate a long pressure test to cover every corner case.

3.1.2 Top Level Testbench

Testbench is combined by a driver and several monitors. The driver is just the given mimic memory module. The testbench will load binary code into the memory and connect the device under test (DUT) to the memory module. The monitors will record the retired instructions with their writeback data and registers, count mispredicted branch and cache miss times. At the end of the test, testbench will print the final memory data with d-cache dirty data in program.out with CPI, cache miss rate, predicted rate statistics. Ground truth was generated by the in order 5-stage processor given in project 3. Both writeback data to register and final memory content are used to check the correctness of execution. During the last phase of verification, an autotest script was used to automatically execute all test programs and check their correctness.

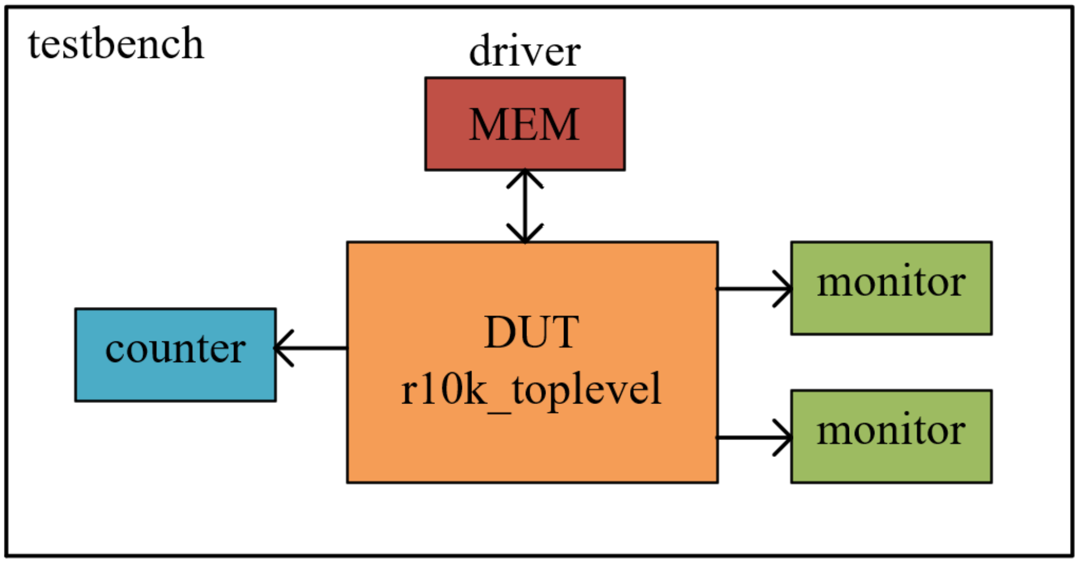


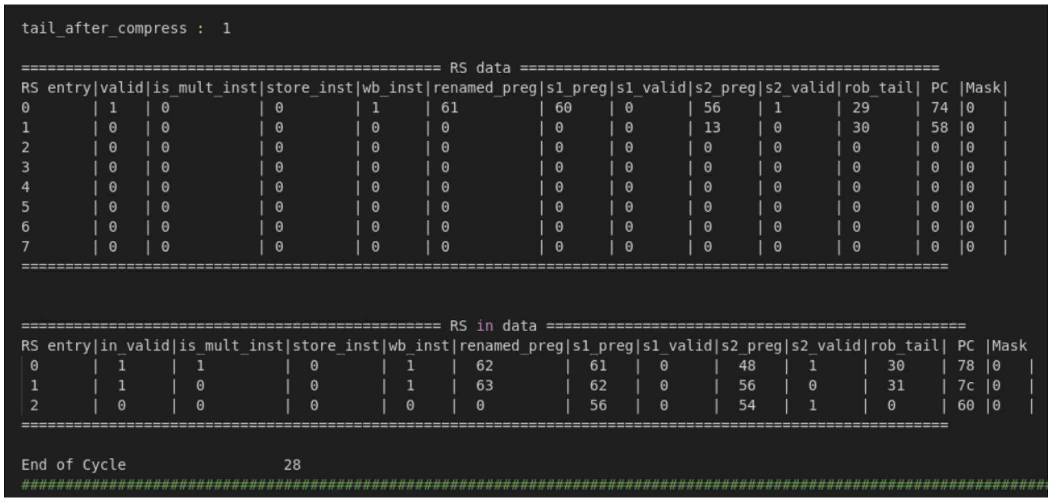
Figure 3.1 Top–level testbench

3.1.3 Post synthesis verification

After synthesizing the top-level design, post synthesis verification was performed. The gate file was used as DUT in the same testbench with similar process.

3.2 Debug Technique

Debugging the huge design was the most challenging work in the project. However, several techniques were developed to make the work more efficient and smooth. Besides the monitor functions for checking the correctness that mentioned before, other monitor functions were used to check internal state of essential modules such as ROB, RS, and LSQ. In most cases, the first error appears after thousands of cycles, so it is not easy to locate the error. If the processor is stuck somewhere, for example, RS is full but not making any issue or a load never returns. We search in the log and find the last time it made a retirement and check ten or twenty cycles before it until find out where the error actually started. If the cause of error is obvious, we can locate the bug in the code and fix it. Some bugs may require modification in several modules or even redesign some function. If the cause of error cannot be found, we use Verdi to check how the error is generated.



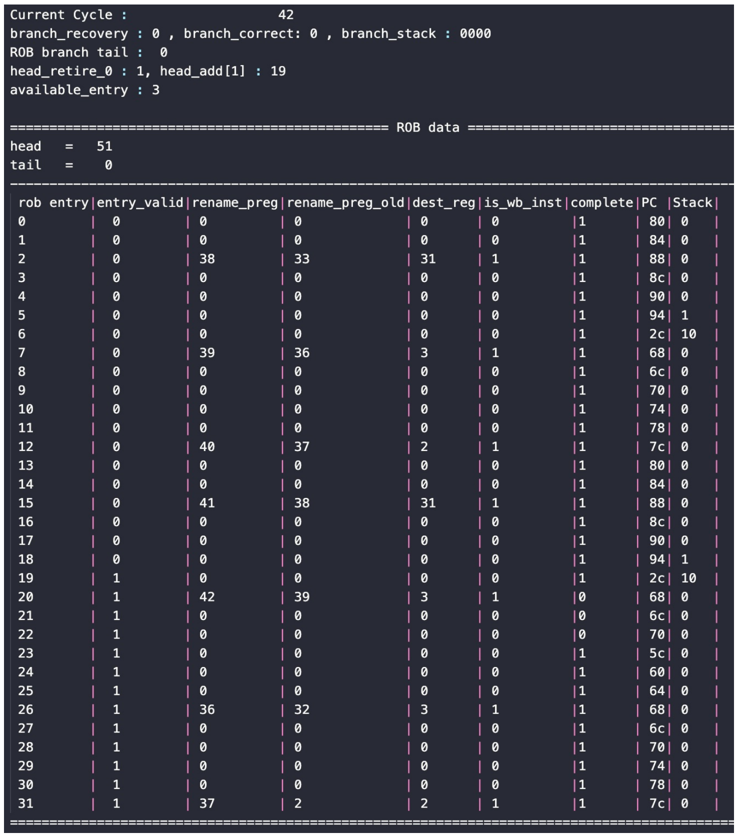
```
tail_after_compress :  1

============================================== RS data ==============================================
RS entry|valid|is_mult_inst|store_inst|wb_inst|renamed_preg|s1_preg|s1_valid|s2_preg|s2_valid|rob_tail| PC |Mask|
0       | 1   | 0          | 0        | 1     | 61          | 60    | 0      | 56    | 1       | 29      | 74 |0   |
1       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 13    | 0       | 30      | 58 |0   |
2       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
3       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
4       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
5       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
6       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
7       | 0   | 0          | 0        | 0     | 0           | 0     | 0      | 0     | 0       | 0       | 0  |0   |
=====================================================================================================


============================================== RS in data ==============================================
RS entry|in_valid|is_mult_inst|store_inst|wb_inst|renamed_preg|s1_preg|s1_valid|s2_preg|s2_valid|rob_tail| PC |Mask
 0      |  1     |  1         |  0       |  1    |  62        |  61   |  0     |  48   |  1      |  30     | 78 |0   |
 1      |  1     |  0         |  0       |  1    |  63        |  62   |  0     |  56   |  0      |  31     | 7c |0   |
 2      |  0     |  0         |  0       |  0    |  0         |  56   |  0     |  54   |  1      |  0      | 60 |0   |
=====================================================================================================

End of Cycle                28
##########################################################################################################
```

Figure 3.2 Example of RS state log



```
Current Cycle :                   42
branch_recovery : 0 , branch_correct: 0 , branch_stack : 0000
ROB branch tail :  0
head_retire_0 : 1, head_add[1] : 19
available_entry : 3

================================================ ROB data ========================================
head   =   51
tail   =    0
--------------------------------------------------------------------------------------------------
 rob entry|entry_valid|rename_preg|rename_preg_old|dest_reg|is_wb_inst|complete|PC |Stack|
 0        |  0        | 0         | 0             | 0      | 0        |1       | 80| 0   |
 1        |  0        | 0         | 0             | 0      | 0        |1       | 84| 0   |
 2        |  0        | 38        | 33            | 31     | 1        |1       | 88| 0   |
 3        |  0        | 0         | 0             | 0      | 0        |1       | 8c| 0   |
 4        |  0        | 0         | 0             | 0      | 0        |1       | 90| 0   |
 5        |  0        | 0         | 0             | 0      | 0        |1       | 94| 1   |
 6        |  0        | 0         | 0             | 0      | 0        |1       | 2c| 10  |
 7        |  0        | 39        | 36            | 3      | 1        |1       | 68| 0   |
 8        |  0        | 0         | 0             | 0      | 0        |1       | 6c| 0   |
 9        |  0        | 0         | 0             | 0      | 0        |1       | 70| 0   |
 10       |  0        | 0         | 0             | 0      | 0        |1       | 74| 0   |
 11       |  0        | 0         | 0             | 0      | 0        |1       | 78| 0   |
 12       |  0        | 40        | 37            | 2      | 1        |1       | 7c| 0   |
 13       |  0        | 0         | 0             | 0      | 0        |1       | 80| 0   |
 14       |  0        | 0         | 0             | 0      | 0        |1       | 84| 0   |
 15       |  0        | 41        | 38            | 31     | 1        |1       | 88| 0   |
 16       |  0        | 0         | 0             | 0      | 0        |1       | 8c| 0   |
 17       |  0        | 0         | 0             | 0      | 0        |1       | 90| 0   |
 18       |  0        | 0         | 0             | 0      | 0        |1       | 94| 1   |
 19       |  1        | 0         | 0             | 0      | 0        |1       | 2c| 10  |
 20       |  1        | 42        | 39            | 3      | 1        |0       | 68| 0   |
 21       |  1        | 0         | 0             | 0      | 0        |0       | 6c| 0   |
 22       |  1        | 0         | 0             | 0      | 0        |0       | 70| 0   |
 23       |  1        | 0         | 0             | 0      | 0        |1       | 5c| 0   |
 24       |  1        | 0         | 0             | 0      | 0        |1       | 60| 0   |
 25       |  1        | 0         | 0             | 0      | 0        |1       | 64| 0   |
 26       |  1        | 36        | 32            | 3      | 1        |1       | 68| 0   |
 27       |  1        | 0         | 0             | 0      | 0        |1       | 6c| 0   |
 28       |  1        | 0         | 0             | 0      | 0        |1       | 70| 0   |
 29       |  1        | 0         | 0             | 0      | 0        |1       | 74| 0   |
 30       |  1        | 0         | 0             | 0      | 0        |1       | 78| 0   |
 31       |  1        | 37        | 2             | 2      | 1        |1       | 7c| 0   |
==================================================================================================
```

Figure 3.3 Example of ROB state

4 Performance Statistics and Analysis

4.1 Overall Performance

	Weighted average	Worst case	Best case
Synthesized clock period	12.5 ns	12.5 ns	12.5 ns
CPI	1.20	1.924	0.844
I-Cache miss rate	1.61%	7.03%	0.26%
D-Cache miss rate	10.04%	50%	2.44%
Branch predicted rate	81.18%	40%	91.5%
Running time	930.769 ms	5513787 ms	1237 ms
Instruction per ms	66621 inst	41580 inst	94787 inst

All average numbers were calculated by weighted average among running all .c programs in the provided test program.

Detailed performance statistics can be found in these links:
r10k performance or
https://docs.google.com/spreadsheets/d/1xvaij53vJ3hth8UQjO93swWhMj4tMwaN55wMY2g8Pho/edit?usp=sharing

4.2 Statistical analysis for different design

4.2.1 Branch Prediction

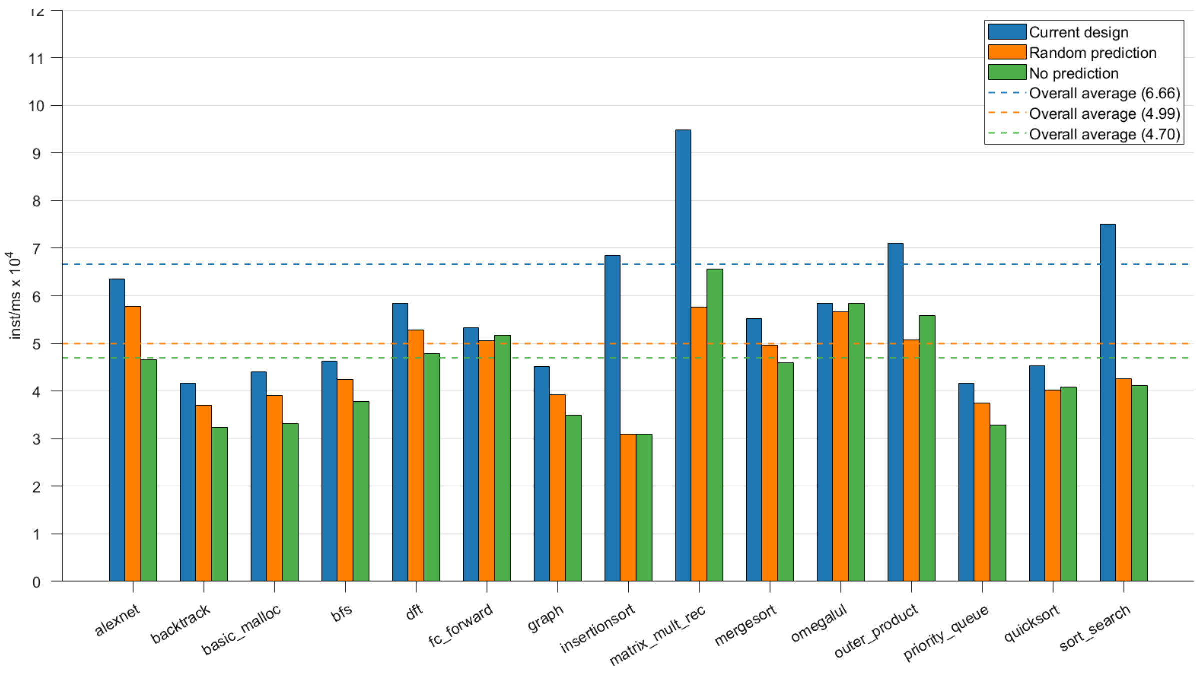


Figure 4.1 Current Design v Random Prediction v No Prediction

From figure 4.1, we can see how the branch predictor with early branch recovery improves the performance of our design. There is a 41.7% average speed up by using the PAg 2-bit saturation predictor with 81.7% prediction accuracy. There is also a 6.2% speed up by only using the early branch recovery with random prediction compared to no prediction design. In some cases, making no prediction may even have better performance than random prediction, because too many wrong predictions may cause unnecessary I-cache eviction. Using way-associated I-cache, larger I-cache, or victim buffer in I-cache may decrease the impact of wrong prediction and improve overall performance even in current design with only 20% mispredicted rate.

4.2.2 I-Cache Design

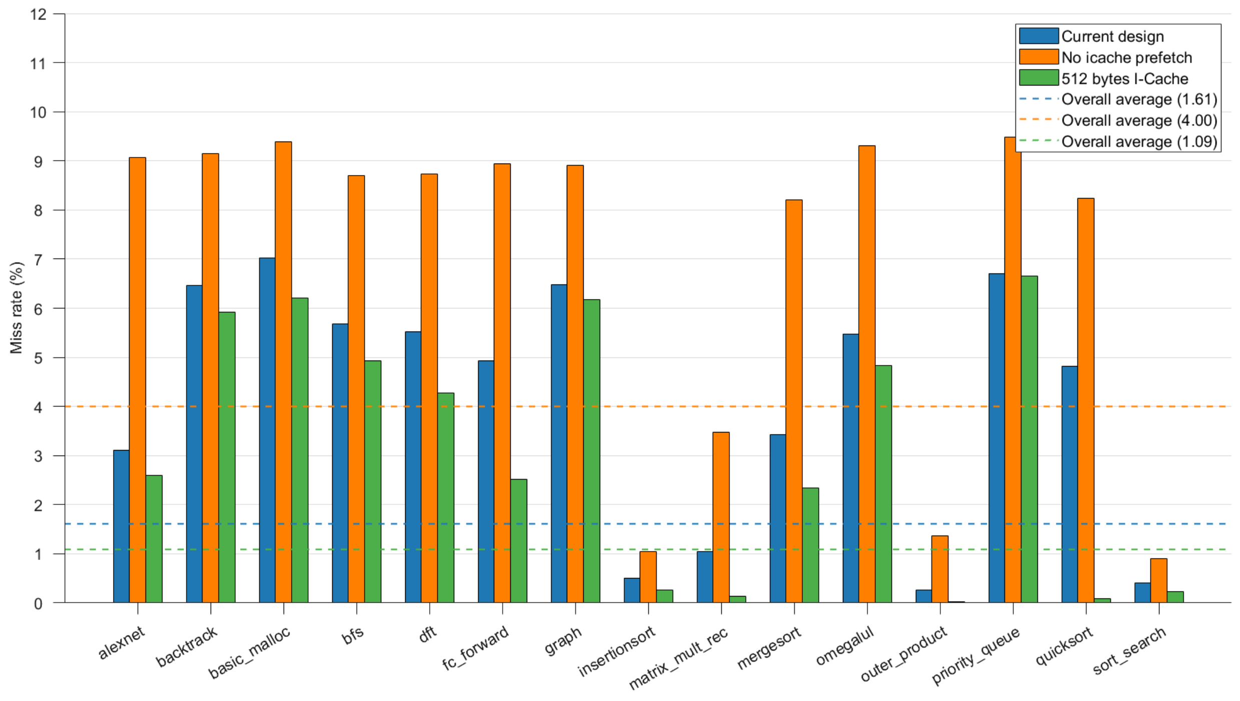


Figure 4.2 Current Design v No prefetch v 512 Byte I-cache

We are using a 256 byte direct-map I-cache with automatic prefetching. When can prefetch at most 16 cache lines or 32 instructions in advance of the PC address fetched by the processor. Figure 4.2 shows that current design with prefetch decreases I-cache miss rate by almost 60%. Without prefetching, consecutive cache misses may happen when executing a section of instructions for the first time while it just happens once with prefetch. Prefetch brings the design almost 50% speed up in overall IPC.

Also, using 512 byte I-cache will further reduce the I-cache miss rate by 47% because there are still too many conflicts misses in such a small cache. However, larger cache may bring more chip area, power consumption and longer critical path that decreases frequency.

4.2.3 LSQ Size

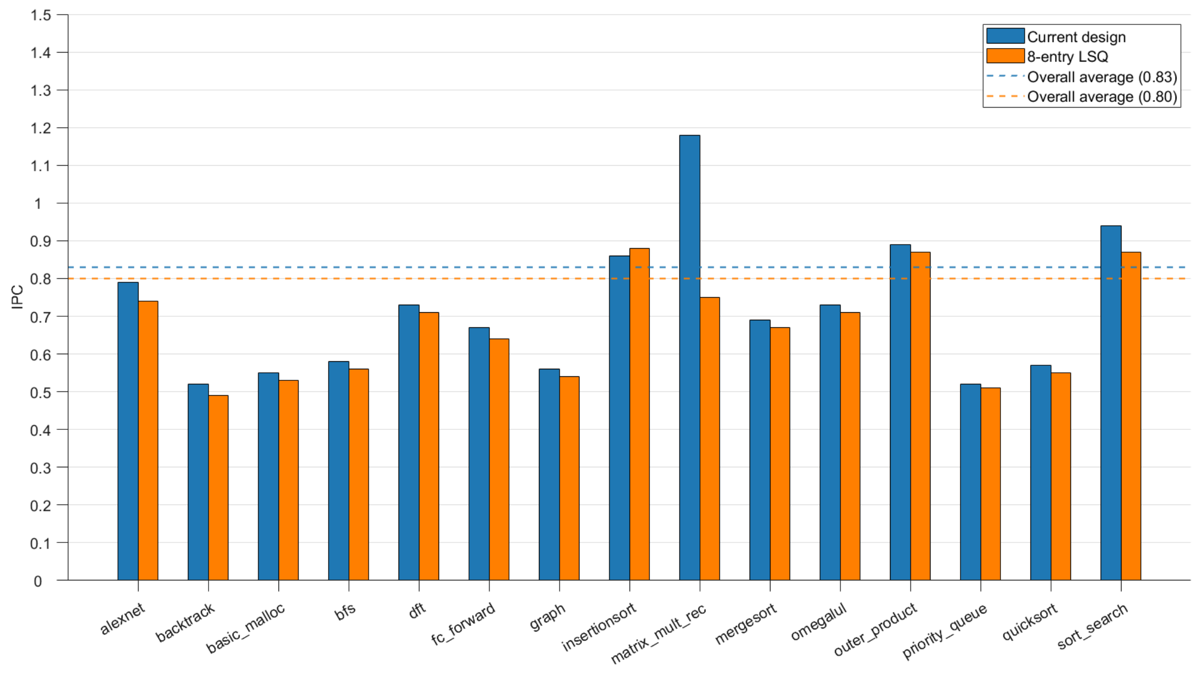
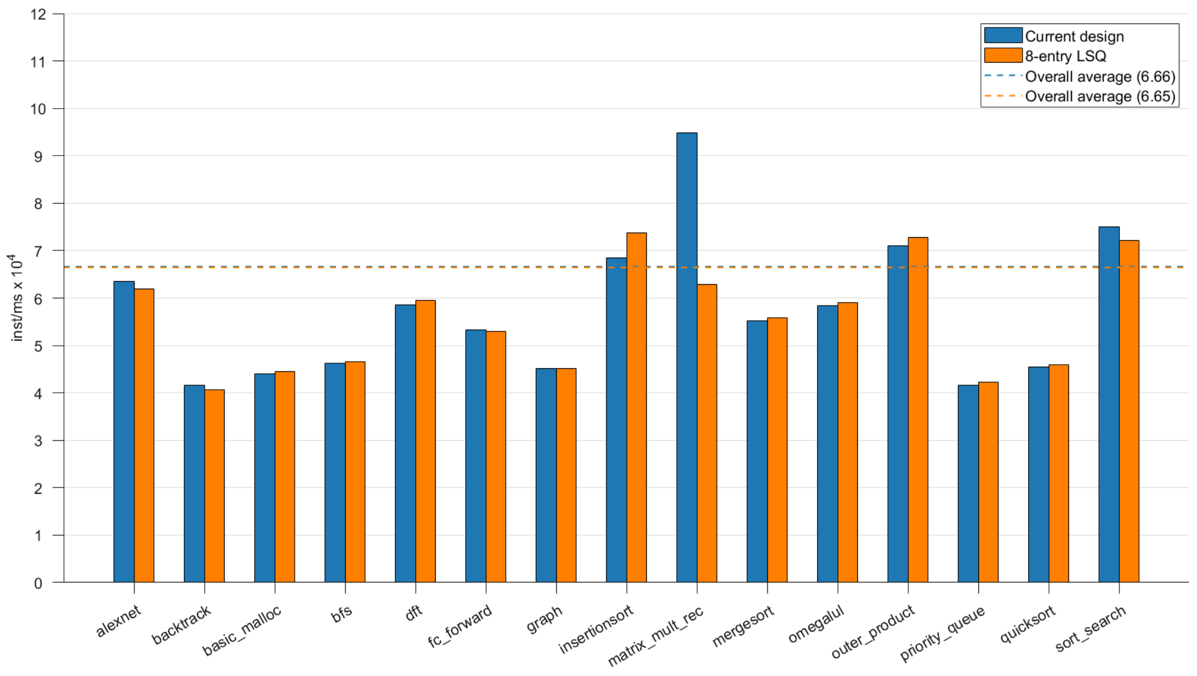


Figure 4.3 IPC of 16 entry (current) v 8 entry LSQ



2

Figure 4.4 Inst per ms of 16 entry (current) v 8 entry LSQ

When LSQ size is 16, the clock cycle of our processor is 12.5 while clock cycle is 12 when LSQ size is 8. If the SQ size increases, it will increase the critical path of load-store forwarding. However, if the LSQ size is larger, it can reduce the chance of structural hazard in LSQ. From the Figure 4.3, it can be seen that for the program which has many load and store instructions in a sequence, the larger LSQ size can tremendously improve the IPC. For test program matrix_mult_rec.c, the IPC improves from 0.75 to 1.19. Although larger LSQ size will increase the clock cycle, according to our statistics, the average Inst per ms under 16 entries LSQ is 6.66 which is slightly better than the average Inst per ms under 8 entries which is 6.65. So we choose 16 as our LSQ size in our design.

4.2.4 RS-EX Forwarding

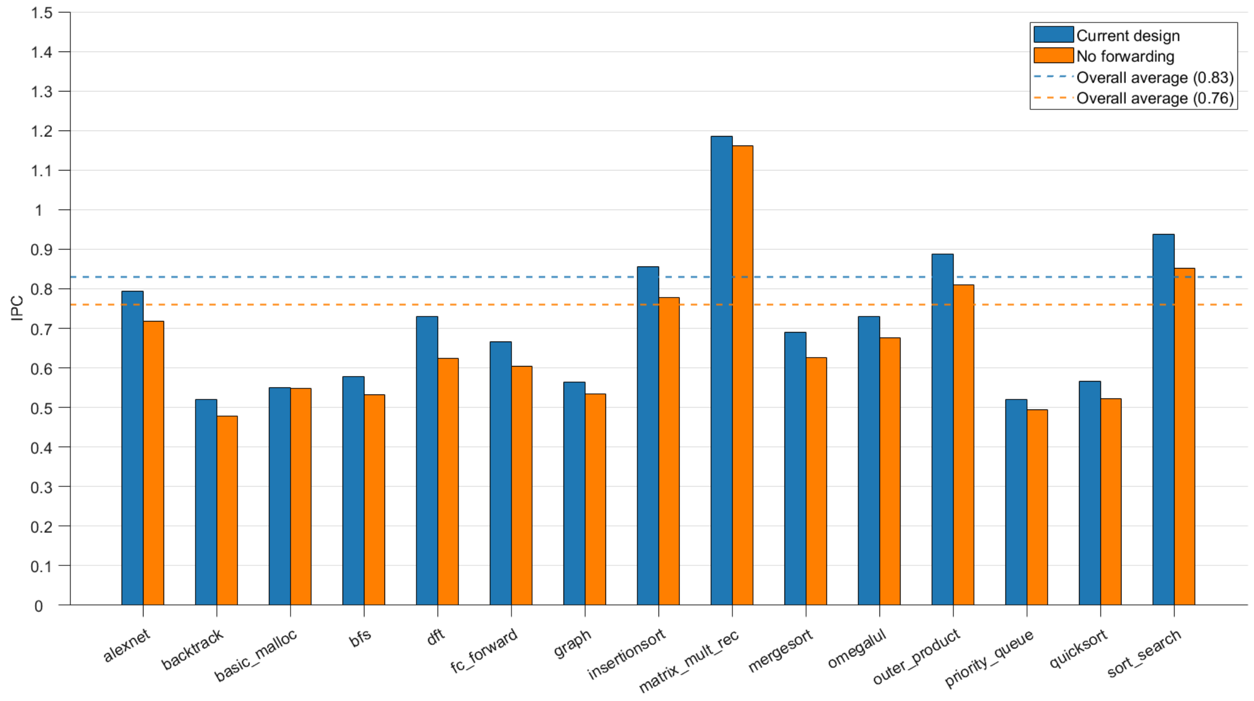


Figure 4.5 Current Design v no RS-EX forwarding

It can be seen from Figure 4.5 that the RS-EX forwarding will improve our overall cpi by 0.07. The RS-EX forwarding can send instructions for execution and free RS entries earlier. Especially for the test programs which have many RAW dependencies between instructions, the RS-EX forwarding can improve performance of our processor.

5 Potential Improvement

After running all the .c in testprog, we find that the performance of our processor in backtrack, bfs is relatively poor compared to other test programs. We think that it is because we do not do the branch prediction on jalr instruction, so we will get a large penalty for the program which uses recursive heavily. So the first potential improvement we want to do is to add a Return Address Stack to our branch prediction part to predict the jalr instruction. The second thing we found is that the d-cache hit rate of out_product.c is 50% which is harmful to our overall performance. It is because that the data needed in out_product.c can not take the spatial locality. So we think that if a stride D-cache prefetcher can be added to our processor is another potential improvement we can implement.

6 Conclusion and Final Remarks

6.1 Conclusion

This project shows the advantage of OoO architecture compared to normal in order structure. With multi-way superscalar and instruction level parallelism, the R10K architecture can achieve a higher IPC. In fully OoO architecture, the pipeline does not need to stall for any single instruction which makes it possible to use 8-stage multipliers and potent floating point units with even longer latency. While long latency computation in an in order structure will stall any following instructions despite data dependency, an OoO structure can successfully resolve the dependency. The usage of multi-cycle functional units boost the clock frequency. The OoO architecture can also benefit from implementing more functional units which is almost impossible in an in order structure. We also investigated how different designs will affect the computation performance in different ways and how to find the balance between CPI and clock frequency. Most importantly, we learnt how great the power generated by a group of people with different strengths working together and the importance of it in the integrated circuits industry which cannot come true without the collaboration of thousands of people from different fields.

6.2 Broader Societal Impacts

Since the well-known prediction was made by Gordon Moore 50 years ago, the size and price of a single transistor has become much cheaper than 50 years ago. As the transistors are so cheap today, we don’t want to be parsimonious in the number of transistors in the design. Instead, we are trying to exploit the greatest potential of computational power with almost infinite resources of transistors. We hope our work can fully utilize the advantages given by current semiconductor technology instead of wasting it. Only high computational power can realize more advanced artificial intelligence and make every people’s life and work easier and more efficient.

6.3 Individual Contributions

Member Name	**Contribution**	**Percentage of Work**
Xinchao Zha	ROB, RS, Packet, EX stage, CDB, LSQ, D-cache, predictor, testbench, optimization, debug	27%
Jianping Shen	Top level, IF, I-cache, predictor, branch recovery, maptable, testbench, optimization, debug	25%
Junyi Cao	Inst buffer, testbench, documentation	13%
Guangze Zu	CDB, documentation, figures	12%
Ning Liang	Freelist, testbench	11%
Anchen Xue	Register file, testbench	12%