EECS 470 Final Project Report

Group No: 4 (Team: Bugs Makers)
Zhuonan Li, Fan Wu, Bohan Wang, Xiaowei Wang, Matt Wojick
Department of Electrical Engineering
{zhuonan, fanwudy, bohanw, xiaoweiw, mwojick}@umich.edu

***Abstract*— in this paper, we propose a 2-way superscalar out-of-order processor with R10K scheme designed. It includes basic modules of R10K scheme such as Reservation Station, Reorder Buffer, Map Table and Free List. In addition, we designed advance feature such as Early Branch Resolution, Forwarding Load Store Queue, Prefetching Instruction Cache and Non-Blocking Data Cache in order to have better performance.**

***Index Terms*— 2-way superscalar out-of-order processor, R10K, Early Branch Resolution, Forwarding Load Store Queue, Prefetching Instruction Cache, Non-Blocking Data Cache.**

I. INTRODUCTION

THIS EECS 470 course project is a 2-way superscalar out-of-order processor implemented in MIPS R10K scheme. The motivation of choosing MIPS R10K scheme but not P6 scheme is to achieve register renaming while keeping less overhead of copying data value. In order to make the processor achievable and less complicated, we chose to use 2-way superscalar instead of N-way. In order to compensate the drawback of small width of the parallel instructions, we implemented Early Branch Resolution, Separate Load Store Queue with Forwarding and Non-Blocking Data Cache.

II. DESIGN

The high level diagram of our 2-way superscalar out-of-order processor with R10K scheme is shown in below. The individual modules design are introduced below. The detail analysis will be provided in the following section.

A. Fetch stage

a) Instruction Cache

We had planned to implement an I-cache prefetcher which fetches one more line from the memory whenever it detects one instruction miss. Ideally, this would reduce half of the latency of loading instructions at the beginning of the execution. However, due to the time limit, we did not finish debugging it in time. The performance would see an improvement had the prefetcher been finished.

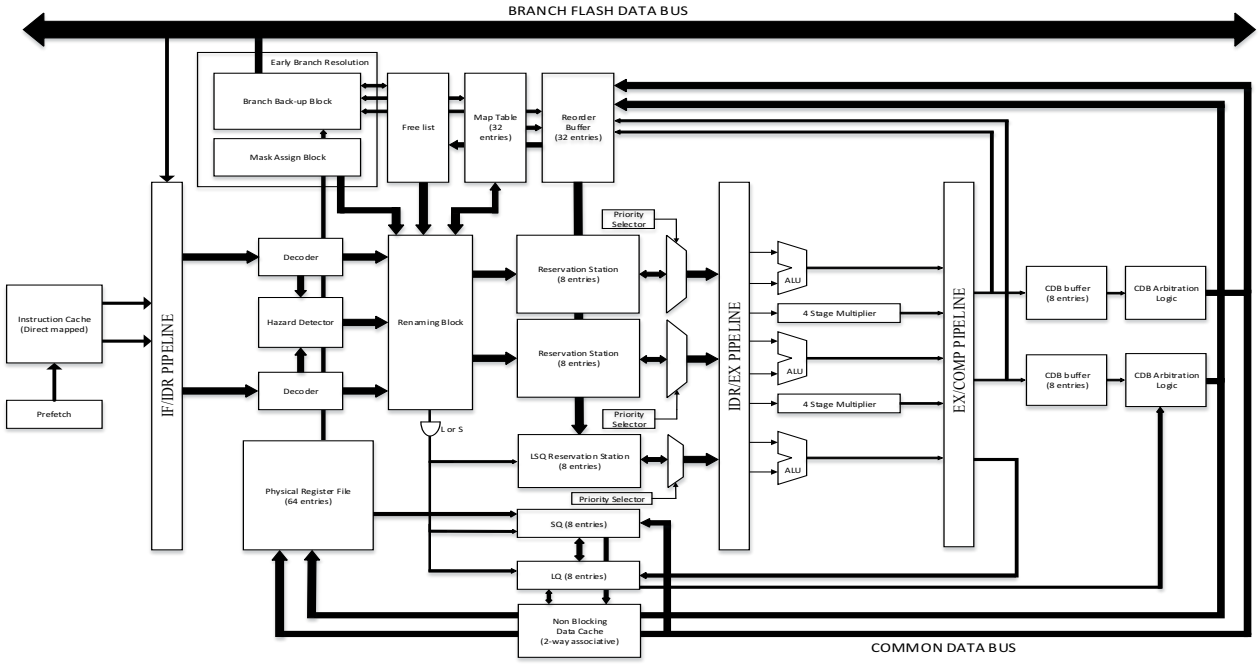

Figure 1:high level design diagram

B. Dispatch Stage (IDR)

There are several modules present in the Dispatch Stage: the Decoder, Data Hazard Detection (for 2 ways), Renaming Block, Reservation Stations, Reorder Buffer, Map Table, Free List, Separate Forwarding Load Store Queue and Early Branch Resolution.

a) Decoder

There are 2 Decoders to decode 2 instructions coming from the IF/IDR pipeline register, and each decoded instruction is sent to each Reservation Station (except the load & store instructions, which have their own unique Reservation Station). In the Decoder, we separated the instructions into 2 types: load & store instructions and non-load & store instructions, in order to determine whether the instructions can be allocated into the Forwarding Load Store Queue or not. In the non-load & store instructions type, we also separated this into 3 smaller types: ALU, MUL and BRANCH. This is for setting the issue priority in the Issue Stage to choose which instruction is to be sent to the Execute Stage.

b) Data Hazard Detection

In the R10K scheme, the data dependency can be solved by the Tomasulo algorithm and Register Renaming. However, when there are data dependency between instructions in way-1 and way-2, R10K cannot solve this kind of data hazard and that is why we implemented the Data Hazard Detection.

There are 3 kinds of data hazards between instructions in way-1 and way-2: Read after Write (RAW), Write after Write (WAW) and Write after Read (WAR). These hazards are detected in this block and will be sent to Renaming Block to have a correct mapping between architecture registers and physical registers.

c) Renaming Block

The Renaming Block is designed to solve the Write after Write (WAW) and Write after Read (WAR) data dependency in a single way and 3 kinds of data dependency occur between the 2-way instructions.

In order to differentiate the valid destination register and non-destination register, we used an extra bit to tell whether the destination register of the instruction is valid or not. If the instructions such as store or some special unconditional branch whose destination is none, the valid bit is 0, otherwise is 1. Also, we used an extra bit to tell whether the source registers are ready or not. If the source is an immediate or ready in Map Table/CDB, the ready bit is 1, otherwise is 0.

The steps are as follows: first Renaming Block and Reorder Buffer will get the new destination tag from Free List first and the corresponding source tags from Map Table; second Map Table will send the old destination tag to Reorder Buffer; third the new updating destination tag will be sent to Map Table. This can solve the data dependency in a single way. When there is RAW dependency between 2-way instructions, Renaming Block get the tag from the Free List (the same as the destination tag for way-1) to the source in way-2 who has RAW dependency with the destination register in way-1. When there is WAW dependency between 2-way instructions, both instructions will get the destination from Free List. But only the destination tag in way-2 will be sent to Map Table to update the information since it is the old tag for the future instructions. And the WAR dependency can be solved by the above renaming steps.

d) Reservation Station (RS)

There are 2 banks of Reservation Stations with 8 entries for the non-load & store instructions and 1 specific Reservation Station with 8 entries for the load & store instruction. The motivation of such design is that we want the load & store instructions can be done as soon as possible and they are not affected even if the other 2 Reservation Station are full. Therefore, when the Reservation Stations for non-load & store instructions are full, they will not stop allocating the load & store instructions to the specific Reservation Station and vice versa. We analyzed the effect of increasing the number of entries, CPI will not have better performance since the CDB can only send 1 instruction out to write back. It's the same result adding more Reservation Station since we only have 2 CDB to broadcast.

Another bottle neck is the specific Reservation Station. For some special test cases, there are lots of load and store instructions and only a small amount of other kinds of instructions. This may lead to the Reservation Station for load & store is highly used and the other 2 Reservation Station are empty at most of the time. This make the processor works as 1 way but not 2-way superscalar, and will greatly affect the CPI.

When Reservation Station are not enough place for new instructions, it will send a stall signal to stall fetching new instructions in IF stage, sending instructions information into Load Store Queue and Reorder Buffer.

e) Reorder Buffer (ROB)

The reorder buffer (ROB) keeps track of the order that instructions were dispatched in, and, in the R10K implementation, only stores the tags, valids, and other metadata, with no need to store the result data. In the context of head and tail movement, it is implemented in the form of a circular buffer with 32 entries. 2 tail and 2 head pointers are used to allocate and retire up to 2 instructions per cycle, respectively. It can retire instructions once the complete bit at the head is set to 1 by the CDB, and it can allocate instructions if it's not full. In order to determine if the ROB is full or not, each entry has an extra 'busy bit' to show that it has been allocated. Thus once the 2 tails point to two entries that are busy, the whole ROB has been allocated, and it must wait for some instructions to retire. An "ROB full" signal is ORed with all other structural hazard signals in order to stall the fetch stage.

The ROB also stores information regarding the instructions OP type (particularly important for branch instructions), and write-mem signals, which tell the LSQ if a store is retiring. For updating the branch predictor, it stores completed branch information, such as the actual branch direction and partial branch target address.

The ROB also handles early branch resolution by restoring the tail location to the backed up tail, and by cleaning busy and complete bits from the restored tail to right before the head. Also, if there is an incoming 'halt' or illegal instruction, when allocating these into the ROB, it immediately sets the complete bits in their respective entries to 1, since they do not have to do any actual computation.

f) Map Table

This is implemented in the form of a 32 entries vector table. There are 4 read ports, 2 write ports and writing enable signal to tell whether to write the new destination tag into Map Table or not. Besides, when there is WAW data dependency between 2-way instructions, the old tag sent to ROB of way-2 instruction should be the same as the new destination tag, which newly comes from Free List, of way-1 instruction.

There is one corner case need to be considered. When CDB broadcast the data value to Map Table, if the Renaming Block updates the same architecture register, the ready bit in Map Table should be set 0 instead of 1. And the information of this ready data can be found in the Reservation Station.

g) Free List

Free List is implemented into 32 entries circular FIFO to allocate the physical registers for the input instructions and get back retire physical register. When there are new instruction, if the destination register is none, Free List will allocate 6'b111111 to the destination tag in Renaming Block. If the destination register is valid, then Free List will allocate the head of it to the destination tag in Renaming Block. If there are not enough physical registers in Free List to allocate, it will send a stall signal to stall fetching new instructions in IF stage, sending instructions information into Load Store Queue and Reorder Buffer.

When Reorder Buffer retires instructions, it will send the old tag of the corresponding retire instructions to Free List.

h) Load Store Queue (LSQ)

Loads and Stores are dispatched to a separate Load Queue and Store Queue.

Store Queue is designed into an 8 entries circular FIFO. The sizing is selected based on 30% of ROB sizing. From the head pointer to the tail pointer, instructions in Store Queue are in program order. Whenever the retire instructions from Reorder Buffer are stores, the head pointer will move forward. Also, whenever there are valid store instructions are dispatched into Store Queue, the store will be allocate inside and the tail pointer will move forward. In the data structure of Store Queue, there are tag and age. Tag is the position in the Store Queue and age is for indicating the order relation with the loads in Load Queue. When the store instructions retire, the corresponding age will be set to 0.

Load Queue is implemented in the form of 8 entries vector table. The sizing is selected based on 30% of ROB sizing. There are also tag and age in Load Queue. When a load is dispatched into Load Queue, if Store Queue is empty, the tag and age of the load instruction are 0, which means there are no pending stores instructions before head. If it's not empty, the tag and age of the load instruction are equal to the tag and age of the last store instruction in Store Queue. In this case, the equal tag represents which store instruction is the youngest older than the load instruction.

The age of store instructions in Store Queue are changing dynamically. In every cycle, the age will be reset from the head pointer to the tail pointer -1 and the value of age should be the distance between the position and head pointer + 1. The age of load instructions in Load Queue are also changing dynamically, following the age of the instructions with the same tag. Therefore, through the age of Store Queue and Load Queue, we can tell which store instructions are older than the load instruction.

We implemented Load Store Forwarding in this module as well. Through the tag and age of Store Queue and Load Queue, we can search the Store Queue for every load instructions in Load Queue whose address is already calculated in the execute stage. We first find the youngest store instruction which is older than the load instruction. Then we search from the store instruction to the head pointer. If the address of the store instruction has not been calculated yet, then stop searching this load instruction and start searching the next load instruction. If the address of the store instruction is ready, then start to compare their address. If the address is the same, then forward the value of the store instruction to the load instruction, set the bit to indicate the data is forwarded to the load instruction and then stop searching this load instruction and start searching the next load instruction. If the address is not the same, keep searching in the same way.

When finish searching all the older store instructions and there are no data forwarding, the Load Queue can send load request to Data Cache. Since the load instructions do not have to complete in order, we used an 8 bit rotate select to pick the ready load instruction and send to Data Cache. When we receive the data from Data Cache, we need some information to tell which load instruction should receive the data. Therefore we attach the index and calculated address with the data to indicate the position. The address information can avoid the data disorder if 2 requesting load have the same index and the first one is a mis-predict one. It might be the case that this load instruction is after a mis-predicted branch instruction. In this case, we used a busy bit to show whether the load instruction is still valid.

i) Early Branch Resolution (EBR)

We implemented Early Branch Resolution to recover the architecture state as soon as the branch is mis-predicted. Without EBR, the processor will keep fetching and executing instructions which will be flushed later because of the mis-predicted branch, until that branch hits the head of ROB. EBR is an efficient way to avoid this behavior and expose ILO in the right direction. There are two major parts in EBR, one is Mask Assignment Block, and the other is Branch Back-up Block.

Mask Assignment Block is implemented in the purpose of sequence of branch instructions and other instruction so that unneeded instructions in RS, pipeline registers, multiplier, LSQ and CDB buffer can be flushed. There are three parts here: mask free list, mask active table and mask history table. The mask free list contains four 4-bit masks which are 4'b0001, 4'b0010, 4'b0100, 4'b1000. These will be assigned to the initial mask field of the instruction. The mask history table and mask active table contain the masks which are active and have left the mask Free List. They will be assigned to the true mask field of the instruction. By telling the dependency of the mask, all the instructions depend on the mis-predicted branch will be flushed, or all the masks of the instructions depending on the correctly predicted branch will be updated to reflect the new dependency on the branch. The initial mask will be returned to mask Free List and the mask active/history table will be updated when a branch is solved.

Branch Back-up Block is for the purpose of copying and storing the necessary values for recovering. The following values will be stored in the Branch Back-up Block.

- ROB tail pointer – the tail pointer which points to the next ROB entry after the branch.
- Free list – Containing tags, read/write pointer and empty/full signal is stored and will be updated when instructions with retire.
- Map Table – The architected register to physical register mapping and ready bits at the moment of dispatching the branch instructions is stored and will be updated when instructions which have no dependency on branch complete.
- SQ head/tail pointer and empty/full signal. The place where branch arrived and will be updated as the older store instructions retire.

Our EBR has four initial masks totally so that it can allow four branch instructions to be active in the pipeline a time. 4-bit initial mask increase the difficulty in implementing the EBR, but offer more capability for branch instruction and reduce the EBR structure hazard.

j) Data Cache

The data cache is a 2-way set associative cache. It has a size of 256 bytes, and each cache block is of 8 bytes. The cache replacement strategy is LRU. Its write policy is write-through, and write allocate. The data cache is mostly non-blocking except when it blocks all subsequent store instructions if they have the same target address as one of the load instructions being processed by the cache. D-cache always has the privilege over I-cache when they contend for a memory bus.

Since chip area is not a major concern in this project, we would like to have a cache of as much size as possible, which is restricted to 256 bytes. The provided memory bus only provides 8 bytes per response so it is natural to select 8 bytes as the smallest storing unit in the cache for simplicity. We adopt LRU as the replacement policy since it is straightforward to implement for a 2-way associative cache, while still preventing the frequently used cache from being evicted easily. Write through is used due to the relatively abundant memory bandwidth. Write allocate policy may also result in a better hit rate, since loading after storing is common in programs. The memory bus arbiter will always serve the command from D-cache first, since the missing data possibly could stall many other units in the processor. We choose to implement the non-blocking data cache to make the full use of a LSQ which may request two memory accesses at peak time. All the design choices above are made at design time, mainly based on our static analysis of the processor and the programs.

The data cache mainly consists of two modules, the cache blocks where the data are actually stored, and the buffers for temporarily storing the load and store instructions that are still waiting for the memory. The cache blocks have one read port and three write ports, so that the two potential store instructions and one data loading from memory may be performed in the same cycle. The write function for storing data into cache would replace the old data in the cache if it finds one to eliminate the discrepancy between the dirty data in the cache and those last written by the core. The index of the victim of one set in the cache is updated whenever a load hits the cache or a data is written into the cache, so that the most recently accessed data would not get evicted.

The major outside signals and the responses of the Dcache controller may be summarized as follows. When a load instruction comes in, it is immediately sent to the cache blocks to see if there exists one block that matches the address. If so, the data is returned to the processor at the same cycle; otherwise, it is queued into the memory request buffer, along with all additional information including its target address and index within the load queue. When a store instruction arrives, it updates the data in the cache at the same cycle while queuing itself into the request buffer. The request buffer keeps sending the entry at the head to the memory until it receives a response from memory. The load instructions are then buffered at another place waiting for the memory's data. The controller snoops on the memory bus to receive and match tags with current outstanding load instructions. After such a match is found, the data is written to the cache at the corresponding place where the load instruction may finally receive it.

C. Issue Stage (IS)

The order of priority from high to low is branches, multiplications and ALU instructions. The motivation is branch may affect the flow of the program, therefore it has the highest priority. Multiplications need 8 cycles to finish calculation, if we issue multiplication first, the instructions depend on that do not have to wait so long.

When the instruction is issued, the corresponding entry will be empty in the next cycle. Therefore, the new instruction can be allocated in that entry in advance.

The Physical Register File is read in the Issue Stage and is written in the Complete Stage. It supports 2 write ports and 6 read ports. When the read address is the same as the write address, the data can be forwarded.

D. Execute Stage (EX)

There are 2 functional units each one dedicated to one of the Reservation Station for non-load & store instruction. Each functional unit has 2 components. (ALU & 8 stage pipeline multiplier) There is a select unit to decide to use which component base on the operation type of the instruction and the one we do not use will be set to invalid. That means we handle 1 instructions execution in every cycle. Since the multiplier is designed in 8 stage pipeline, there may be the case that both ALU and multiplication finish executing, then both of the instructions will be sent to Complete Stage and wait for CDB broadcasting.

Besides, there is a specific ALU for calculating the load & store address and this result will be sent to Complete Stage and wait for CDB broadcasting as well.

E. Complete Stage (C)

There are 2 Common Data Bus (CDB) for 2-way superscalar processor. Since there may be more than 1 instructions needed to be sent through CDB, we design an arbitration logic to select 1 instruction to CDB. The arbitration logic is implemented in the form of a vector buffer. Load and store instructions have the highest priority, whenever a load & store instruction finish executing, it will be forwarded to CDB and do not have to wait in the buffer. For the other instructions, if the buffer is empty,

the instruction can be forwarded to CDB. Otherwise, we used a rotate priority selector to select one instruction to broadcast.

III. Performance Summary

A. CPI Performance Analysis

Program name	Instructions	CPI	Cycle time(ns)	Tcpu(ns)
btest1	229	10.38	10	23770
btest2	455	8.567	10	38970
copy_long	590	1.207	10	7120
copy	130	2.115	10	2740
evens_long	318	1.66	10	5270
evens	82	3.378	10	2770
fib_long	627	1.399	10	8770
fib_rec	12942	2.009	10	260000
fib	147	2.251	10	3310
insertion	598	1.619	10	9680
mult	325	2.609	10	8480
objsort	19704	6.463	10	1273470
parallel_long	910	1.09	10	9920
parallel	194	1.768	10	3430
parsort	11106	1.287	10	142930
saxpy	185	3.216	10	5950
sort	1349	2.249	10	30340

The actual performance of the processor differ for different benchmark programs. In general, the processor performs well except for btest1, btest2, and objsort, which involve frequent branch instructions that are hard to predict. Therefore the large overhead of misprediction and a less accurate branch predictor may heavily degrade the overall performance. On the other hand, our processor generally perform well for the programs with suffix "long", since those programs contain a large portion of nop instructions, which are effectively prevented from taking up computing resources.

Finally, although we designed a two-way superscalar processor, the actual CPI never reaches 0.5, which can be due to the large amount of overhead inside the processor pipeline, as well as the significant memory latency.

B. LSQ Analysis

The following graph shows the number of store to load forwarding for different test cases. The LSQ forwarding will have more improvement when the test case are the decaf complied program since these programs use more load & store instructions than other basic test case.

When we search for forwarding, it will stop searching if we find the address of the older store has not been calculated yet. Until we get the store address, we can keep searching the store queue. It might be the case that during the waiting time, some potential forwarding store may retire, which may lead to the decrease of LSQ forwarding number.

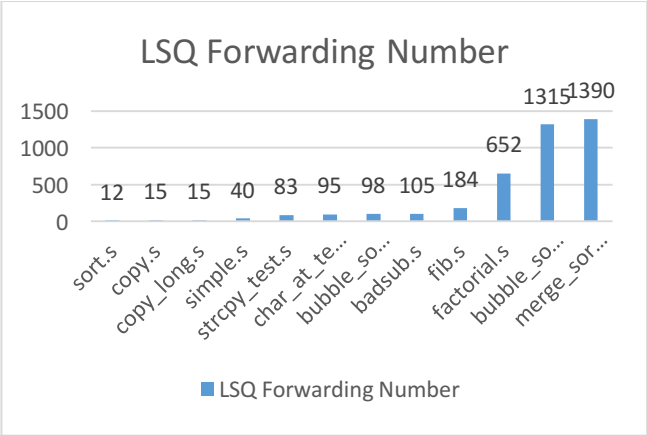

Figure.2 LSQ Forwarding Number

C. ROB Sizing Optimization Analysis

We sweep the sizing of Reorder Buffer and to find a optimize sizing for lower CPI. From the graph below, even though having a smaller sizing may lead to more stalls, in most test cases the CPI does not change much. Besides, having a smaller size (like 16 entries) of ROB is area and power efficient.

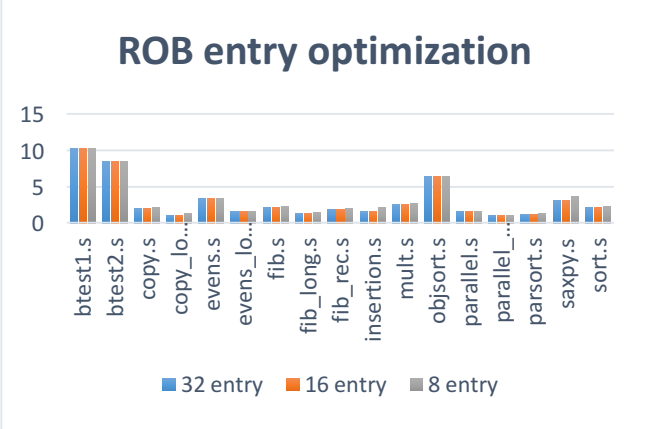

Figure.3 ROB entry optimization

D. Early Branch Resolution Analysis

The purpose of adding EBR is to recover the processor as soon as a branch is solved and found mispredicted. Without EBR, the average latency between resolving a mispredicted branch and retiring it is estimated to be three cycles, and one more cycle is needed to flush the instructions depending on the mispredicted branch, so that totally four cycles are needed. With EBR, only one cycle is needed for recovery.
The figure below shows how much performance is improved with EBR. The analysis is based on the Predict-not-taken baseline processor.

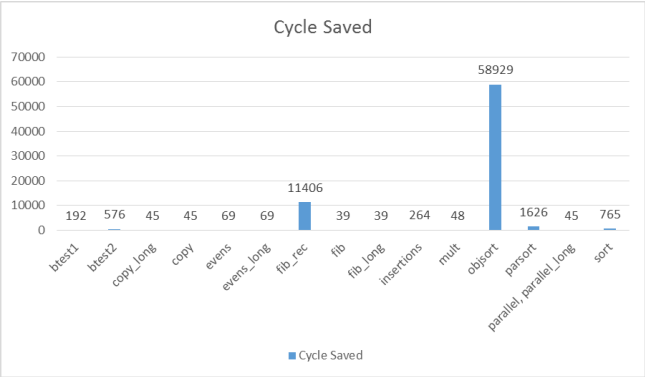

Figure 4 Number of saved cycle using early branch resolution

For program which has high misprediction rate, EBR can improve the performance obviously.

E. Data Cache Analysis

We observe significant performance improvement (CPI reduction 15% ~ 32%) from the addition of a data cache for several programs (fib_rec, fib, insertion, objsort, parsort, sort). These programs also have a large number of cache hits. In contrast, the programs that contain nearly no memory operations do not experience much performance improvement from data cache.

The cache hit rates also vary among different programs. The programs that uses few memory addresses (fib_long, fib) would see a higher hit rate since there are fewer evictions in the cache. Interestingly, the program "copy" only causes one hit and no misses in the cache despite containing much more load instructions actually. This reduction in memory access is due to the effective internal forwarding inside the LSQ. The program "objsort" does not have a very high hit rate (0.58), but it sees a performance improvement since the absolute cache hit count is high. One program, "saxpy", has a cache hit rate of zero because it is loading data from all different memory addresses so there are a large number of compulsory misses. The overall hit rate of the data cache is 0.62. Data Cache hit rate table as follow:

	hit	miss
copy_long	1	0
copy_long	1	0
fib_long	28	0
fib_rec	1778	49
fib	28	0
insertion	108	30
mult	0	2
objsort	8117	5945
parsort	544	639
saxpy	0	36
sort	249	17
total	10854	6718

F. Instruction Cache with Prefetcher Analysis

Although the I-cache prefetcher we have devised may not function correctly for all programs, we still observe a significant CPI reduction (28% ~ 41%) for certain programs like btest1 and btest2. Because these programs tend to execute a large number of different instructions, an instruction prefetcher is helpful in reducing the number of total cycles by pipelining the fetching of two new instructions. Other few programs (fib_rec) that mostly execute the same set of instructions repeatedly do not see much improvement provided by I-cache prefetcher.

IV. GROUP DYNAMIC

Zhuonan Li (32%) – hazard detection, renaming block, reservation station, issue stage, execution stage, complete stage (with CDB), free list, map table design, separate store queue & load queue, whole core debugging

Fan Wu (32%) - Integration and debug, Early branch resolution (mask assignment block, branch back-up block), LSQ design and test, RS deisign and implementation, Hazard detection, renaming block, multiplier, Fetch stage

Bohan Wang (13%) – map table, reservation station, fetch stage

Xiaowei Wang (18%) - Integration of Dispatch/Decode Stage; Design of Dedicated Reservation Station for Load/Store; Design of Data Cache

Matt Wojick (7%) - ROB, ICache Prefetch

V. CONCLUSION

We have successfully implemented our 2-way superscalar out-of-order R10K processor with the advanced feature we proposed. This course project gives us much insight to the development and verification of computer architecture.