R10K-Based 2-Way Superscalar Out-of-Order Microprocessor

EECS 470 Group 9

Ram Srivatsa Kannan
ramsri@umich.edu

Kartik Joshi
kartikj@umich.edu

Salessawi Ferede Yitbarek
salessaf@umich.edu

Thomas Zachariah
tzachari@umich.edu

Walter Zarate
wezarate@umich.edu

I. INTRODUCTION

A. Overview

This report describes the out-of-order core implemented by team TWRKS for the EECS 470 term project. We have based our core on modified version of Tomasulo's R10K micro-architecture. The core has been designed giving priority to correctness and then overall performance. None of our design decisions took area and power into account. Furthermore we were successful in implementing many advanced optimizations. The basic microarchitecture of our processor is an out of order 2-way Superscalar system. The processor has the following design features:

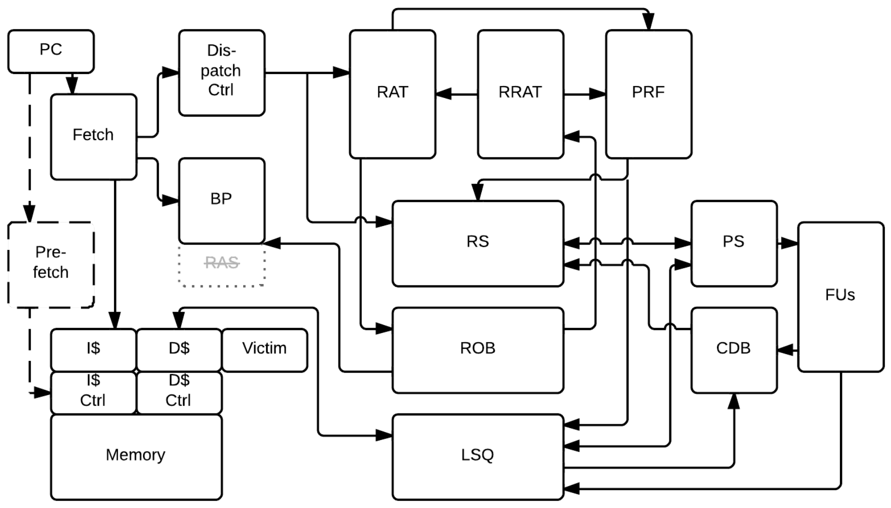
1. 2-Way Superscalar
2. Load Store Queue (OoO Loads & Store-to-Load Forwarding)
3. Tournament Branch Predictor (Local and Global)
4. Data Cache (N-way Set Associative, Non-blocking, Dual-ported)
5. Victim Cache
6. Instruction Prefetcher
7. Custom Built GUI Debugger

More details on the individual components as well as the microarchitectural configuration will be provided in the later part of the manuscript. We had a focused approach for design verification which included the development of a visual debugger and automated testing suite for fast and efficient debugging and regression analysis.

B. Who Did What

1. Ram Srivatsa Kannan(20%): Branch Prediction, PS, RS, Testing, Top Level
2. Kartik Joshi(20%): RoB, RS, Cache, Testing, Synthesis Guy, Top Level
3. Salessawi Yitbarek(20%): GUI Debugger, RS, Testing, Non-blocking Cache Controller, Fetch/Prefetch and dispatch logic, Regression Test Scripts, Top Level
4. Thomas Zachariah(20%): PRF, Load-Store Queue, RS, Testing, Top Level
5. Walter Zarate(20%): RAT, RRAT, Load-Store Queue, RS, Testing, Top Level

II. IMPLEMENTATION



A. *Reservation Station*

The processor has 16-entry unified reservation station with concurrent 2-way dispatch, issue and broadcast. A free list keeps track of the free entries of the reservation. The no of free slots is combinationally determined by the head and tail pointers of the free list. When the reservation station indicates that it has not free entries left, the instruction dispatch is stalled. By keeping the size of the free list as rob_size+1, it is ensured that the free list is never full, and thus does not require additional list_full flag bits.

B. *Reorder Buffer*

The processor has 32-entry reorder buffer with concurrent 2-way dispatch, issue, complete and commit. The reorder buffer stores track of ARN and PRN of the destination operand, PC value, store queue index, branch target address, and has flags for whether the instruction is halt, store or branch. The ROB is responsible for the following actions:

- Signaling the store buffer to commit stores
- Detecting branch mispredicts by comparing computed target addresses with the PC of the instruction following the branch
- Update the RRAT when instructions hit the head of the ROB

C. *Caches*

Our design has an 8-line arbitrarily-associative (configured for 4-way associativity) non-blocking, dual ported data cache. We also have 4-line fully associative victim cache appended to the data cache. The data-cache and the victim cache follow pseudo-LRU replacement policy with LRU tree flattend and represented an array of length ASSOCIATIVITY-1. Even if we also experimented with an associative instruction cache, the final submission was configured to a 32-line direct-mapped instruction cache.

The non-blocking cache controller is a separate module from the associative memory has 16 miss status handling registers(MSHRs) and directly communicates with the LSQ, cache memory and main memory. This same controller and associative cache was used to implement next-line instruction prefetching in the fetch stage.

D. *Functional units*

The processor has 2 ALUs and one 4-stage multiplier. The ALU is used for address calculation as well. In addition to operating on the operands, the ALU also propagates the destination PRN and ROB indexes for broadcast to the CDB.

E. *Physical Register File*

The processor has a physical register file (PRF) containing 64 physical registers. The registers are renamed by the RAT. If an instruction requiring data from a register is dispatched, the PRF is checked for values in the corresponding entries and are provided to the Reservation Station if present. If not present, the index of the PRF entry is provided to the Reservation Station, so it knows to listen for the resolved value on the CDB. Instructions that require a new register are assigned a PRF location by the RAT. Additionally, the PRF free list is maintained by the RAT, and updated by the RRAT on branch mis-predicts.

F. *Register Aliasing Table*

Register Aliasing Table, also known as the Map Table, was implemented as a set of 32 registers. This memory units holds the current valid pointers to the physical register file. Every time an instruction is dispatch, the RAT makes a translation for the two operands given. It also assigns a free PRF pointer to the destination register. All this information is pass down to the reservation station and the reorder buffer for further instruction execution. In order to guarantee the RAT can assign a free pointer to the destination register, an internal free list queue is used to keep track of the availability of these pointers.

One key feature the RAT is that it supports different dispatch and complete instructions sizes. It also implements internal forwarding to make sure all possible dispatch cases are covered.

- Variable width Dispatch: *Map Architectural Register to Physical-Register*
  Ex. Dispatch width of 2
  R2 + R4 = R5 -----renamed----- P1 + P8 = P10 *(free list == free list –P10)*
  R4 + R3 = R8 -----renamed----- P8 + P7 = P9 *(free list == free list –P9)*

  Ex Dispatch width of 2 and internal Forwarding
  R2 + R4 = R5 -----renamed----- P1 + P8 = P10 *(free list == free list –P10)*
  R5 + R3 = R8 -----renamed----- P10 + P7 = P9 *(free list == free list –P9)*

In order to have a sustainable system where we can re-used unused pointers to the PRF, the RAT adds entries to the free list when a committing instruction overrides some PRF pointer no longer needed.

- Free a unused pointer
  Ex commit two instructions
  Commit a P10 = result (overrides p20) *(free list == free list +P20)*
  Commit a P9 = result (overrides p15) *(free list == free list +P15)*

During Branch miss prediction, the RAT copies all the information including free list, free list pointer and memory tables from the RRAT.

G. Retirement Register Aliasing Table

Retirement Register Aliasing Table, also known as the Architectural Map Table receives inputs from the ROB during retire stage. The RRAT was implemented as a set of 32 registers. Just like the RAT, these memory units hold PRF pointer information.

The key difference between the RRAT and the RAT is that the former is like a past version of the RAT and serves as a safety net in case of a branch miss prediction since only instructions that are valid are allow to make changes to this structure.

Another important feature of the RRAT table is to process and output the correct free PRF pointers to the RAT.

- Update memory and free pointers
  Ex commit two instructions
  Commit a P10 = result (overrides p20) *(free list == free list +P20)*
  Commit a P9 = result (overrides p15) *(free list == free list +P15)*

During a branch miss prediction (BMP) the RRAT just needs to make sure all instructions in front of the BMP (valid instructions) are allowed to make changes to the Architectural state. Once this part is complete the RRAT send out all its information to be copy by the RAT.

H. Load Store Queue

The load store queue was implemented as a split store queue and load buffer, each having 16 entries. The store queue and the free list structure required for the load buffer are implemented as circular queues.
The key advantage of this implementation of the load store queue is the capability to execute loads out of order and to forward data from uncommitted stores to loads. This functionally drastically improves the performance of the processor since loads and stores typically produce the largest instruction latencies.

Load Buffer:

- *Dispatch* - During dispatch the load entry will get as much information as possible from the dispatcher since the format of the loads contain one source operand and one immediate value, it is possible that we can get the actual value or just a pointer to the PRF. In either case the load queue provides an index to the RS and the ROB where all the entries being store. once the RS sends the load address to be calculated, the RS also attaches the index so the load can later match it with the corresponding row.

- *Execute* - Once the load have their corresponding address resolved, there are two key stages that happened during execution: the loads must resolve who are the relevant stores in front of him and save this information in a structure we called store map and the load should make a resolution of their next step, the options are to forward the data (if we have a relevant store and their address matched), wait until the relevant store resolves the address or go and fetch the load data data from memory.

- *Complete* - The load has a ready bit that request the system (PS)permission to use the CDB, once these permission is granted the load controls the CDB to set the corresponding data result and PRF pointer. The load entry is clear and the index is added to the free list.

- *Commit* - Once the load instruction gets to the head of the ROB and the instruction becomes ready for commit, the load execution has already completed and the only change left is to update the RRAT.

Store Queue:

- *Dispatch* - During dispatch the stores get the destination PRF pointer or the actually data to be store. If the data is not available during dispatch by default it will come with the address calculation. Another key aspect to the store queue is that it will provide index to the RS and the ROB to be able to track any results back to its correct memory location.

- *Execute* - During this stage the RS will collect the correct data for the address calculation units. once the correct information is store in the RS and the priority selector grants use of the functional units, the address result will be send to the store queue.

- *Complete* - When the store's address is resolved, it updates the store map in the load queue so that loads which are dependent on them can get their data forwarded and loads which are not dependent on them can go fetch their associated data from memory.

- *Commit* - A store can commit whenever the store instruction reaches the head of the store queue and its corresponding entry in the ROB reaches the head of the ROB, it will request memory from storing its information. After it is granted the information, they commit , update architectural state and removes their associated entry from LSQ and ROB.

I. Prefetching and Front End

After the fetch unit has requested the instruction at the current PC, the prefetching logic automatically places a request for the next instruction as well. If the instruction that is intended to be prefetched is already in the I-cache, the request is not sent to memory. We also experimented with prefetching upto six extra instruction but did not see any performance gain beyond prefetching two instructions.

J. Branch Predictor

- Global Predictor - The Global branch predictor (BP) is implemented as 4 bit branch history buffer with an array consisting of 16 entries of pattern history table consisting of 2-bit saturating counters that are indexed based on the branch history. We have a 16 entry direct mapped branch target buffer that predicts the address of the branch when it is predicted taken. The branch predictor updates itself after committing every branch instruction. It can service two branches retiring and fetching simultaneously.

- Local Predictor - The Local branch predictor (BP) is implemented as a table consisting of 16 entries each entry indexed by the last 5 bits of the PC. Each entry in the PC contains 4 bits of the branch history of all branches indexed to that entry. Each entry in branch history table is indexed to the pattern history table consisting of 2-bit saturating counters. The configuration for the BTB and the update logic is same as global predictor.

- Tournament Predictor - The Tournament branch predictor (BP) is implemented as an arbiter that arbitrates between the global branch predictor and the local branch predictor. The configuration of the individual predictors are same as the ones mentioned above.

K. *Priority selector and CDB arbitration*

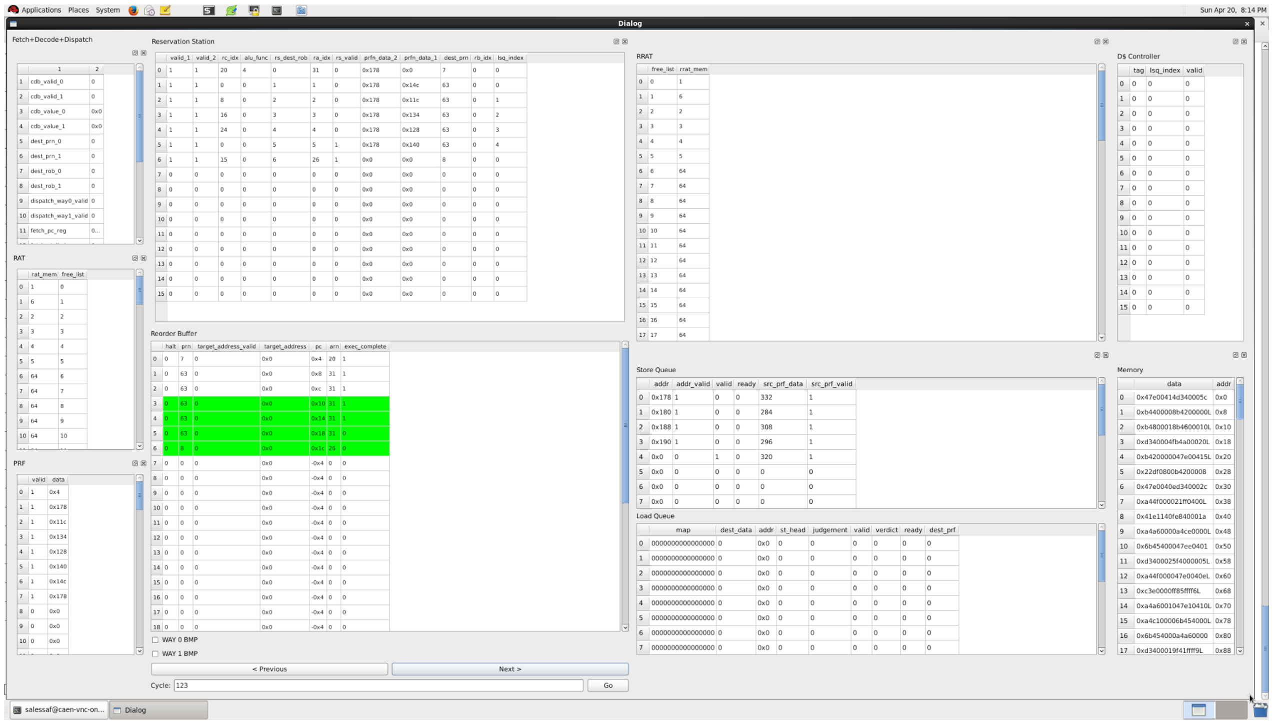
The CDB can only broadcast 2 outputs at a time from the buffers that wanted to use it. The priority selector is responsible for choosing the 2 outputs which needed to be broadcasted. The inputs to the PS are the entries in the Reservation Station as well as those in the Load Store queue that has completed execution whose values are ready to be broadcasted. We have given preference to the entries in the load store queue since there would be more operations dependent on memory outputs

L. *Testing*

Unit tests were written for all modules before integration. However, post integration tests have exposed more bugs that were not caught by our unit tests. Integration tests were initially made with all modules outside of the core turned off. Extra modules(non-blocking caches, branch predictors and LSQ features etc...) were turned on gradually after debugging the module intrinsic to the core.

The correctness of our processor was verified using tests scripts that run a given program on the provided inorder core as well as our out-of-order core. The script compares register values of the inorder core at the end of each cycle with register values of the out-of-order core after each instruction was retired. If there is a discrepancy between the two, we pin-point at which clock cycle the error first surfaced. Our test scrip also final memory contents of the inorder and out-of-order core - similar to the way the autograder verifies our design.

After identifying at which cycle the error occurred, use our custom built visual debugger to jump to the desired cycle and investigate the problem further. We used a library named cocotb to hook into the vcs simulation and prob signals and did not need to add any extra code to our HDL code. We decided to build our visual debugger for two reasons. First, debugger provided on the course website uses a an old library which makes it hard to prob signals that rely on certain features of SystemVerilog. Second, we believed that it would be much easier for us to modify a code we wrote ourselves as the semester progressed. As shown in the figure below, the debugger shows the states of different units (RS, ROB, RAT, RRAT, LSQ, Cache Controller, and memory) for every cycle.



Fetch+Decode+Dispatch

	1	2
1	cdb_valid_0	0
2	cdb_valid_1	0
3	cdb_value_0	0x0
4	cdb_value_1	0x0
5	dest_prn_0	0
6	dest_prn_1	0
7	dest_rob_0	0
8	dest_rob_1	0
9	dispatch_way0_valid	0
10	dispatch_way1_valid	0
11	fetch_pc_reg	0...

RAT

	rat_mem	free_list
0	1	0
1	6	1
2	2	2
3	3	3
4	4	4
5	5	5
6	64	6
7	64	7
8	64	8
9	64	9
10	64	10

PRF

	valid	data
0	1	0x4
1	1	0x178
2	1	0x11c
3	1	0x134
4	1	0x128
5	1	0x140
6	1	0x14c
7	1	0x178
8	0	0x0
9	0	0x0
10	0	0x0

Reservation Station

	valid_1	valid_2	rc_idx	alu_func	rs_dest_rob	ra_idx	rs_valid	prfn_data_2	prfn_data_1	dest_prn	rb_idx	lsq_index
0	1	1	20	4	0	31	0	0x178	0x0	7	0	0
1	1	1	0	0	1	1	0	0x178	0x14c	63	0	0
2	1	1	8	0	2	2	0	0x178	0x11c	63	0	1
3	1	1	16	0	3	3	0	0x178	0x134	63	0	2
4	1	1	24	0	4	4	0	0x178	0x128	63	0	3
5	1	1	0	0	5	5	1	0x178	0x140	63	0	4
6	1	1	15	0	6	26	1	0x0	0x0	8	0	0
7	0	0	0	0	0	0	0	0x0	0x0	0	0	0
8	0	0	0	0	0	0	0	0x0	0x0	0	0	0
9	0	0	0	0	0	0	0	0x0	0x0	0	0	0
10	0	0	0	0	0	0	0	0x0	0x0	0	0	0
11	0	0	0	0	0	0	0	0x0	0x0	0	0	0
12	0	0	0	0	0	0	0	0x0	0x0	0	0	0
13	0	0	0	0	0	0	0	0x0	0x0	0	0	0
14	0	0	0	0	0	0	0	0x0	0x0	0	0	0
15	0	0	0	0	0	0	0	0x0	0x0	0	0	0

Reorder Buffer

	halt	prn	target_address_valid	target_address	pc	arn	exec_complete
0	0	7	0	0x0	0x4	20	1
1	0	63	0	0x0	0x8	31	1
2	0	63	0	0x0	0xc	31	1
3	0	63	0	0x0	0x10	31	1
4	0	63	0	0x0	0x14	31	1
5	0	63	0	0x0	0x18	31	0
6	0	8	0	0x0	0x1c	26	0
7	0	0	0	0x0	-0x4	0	0
8	0	0	0	0x0	-0x4	0	0
9	0	0	0	0x0	-0x4	0	0
10	0	0	0	0x0	-0x4	0	0
11	0	0	0	0x0	-0x4	0	0
12	0	0	0	0x0	-0x4	0	0
13	0	0	0	0x0	-0x4	0	0
14	0	0	0	0x0	-0x4	0	0
15	0	0	0	0x0	-0x4	0	0
16	0	0	0	0x0	-0x4	0	0
17	0	0	0	0x0	-0x4	0	0
18	0	0	0	0x0	-0x4	0	0

RRAT

	free_list	rrat_mem
0	0	1
1	1	6
2	2	2
3	3	3
4	4	4
5	5	5
6	6	64
7	7	64
8	8	64
9	9	64
10	10	64
11	11	64
12	12	64
13	13	64
14	14	64
15	15	64
16	16	64
17	17	64

Store Queue

	addr	addr_valid	valid	ready	src_prf_data	src_prf_valid
0	0x178	1	0	0	332	1
1	0x180	1	0	0	284	1
2	0x188	1	0	0	308	1
3	0x190	1	0	0	296	1
4	0x0	0	1	0	320	1
5	0x0	0	0	0	0	0
6	0x0	0	0	0	0	0
7	0x0	0	0	0	0	0

Load Queue

	map	dest_data	addr	st_head	judgement	valid	verdict	ready	dest_prf
0	0000000000000000	0	0x0	0	0	0	0	0	0
1	0000000000000000	0	0x0	0	0	0	0	0	0
2	0000000000000000	0	0x0	0	0	0	0	0	0
3	0000000000000000	0	0x0	0	0	0	0	0	0
4	0000000000000000	0	0x0	0	0	0	0	0	0
5	0000000000000000	0	0x0	0	0	0	0	0	0
6	0000000000000000	0	0x0	0	0	0	0	0	0
7	0000000000000000	0	0x0	0	0	0	0	0	0

D$ Controller

	tag	lsq_index	valid
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0

Memory

	data	addr
0	0x47e00414d340005c	0x0
1	0xb4400008b4200000L	0x8
2	0xb4800018b4600010L	0x10
3	0xd340004fb4a00020L	0x18
4	0xb420000047e00415L	0x20
5	0x22df0800b4200008	0x28
6	0x47e0040ed340002c	0x30
7	0xa44f000021ff0400L	0x38
8	0x41e1140fe840001a	0x40
9	0xa4a60000a4ce0000L	0x48
10	0x6b45400047ee0401	0x50
11	0xd3400025f4000005L	0x58
12	0xa44f000047e0040eL	0x60
13	0xc3e0000ff85ffff6L	0x68
14	0xa4a6001047e10410L	0x70
15	0xa4c100006b454000L	0x78
16	0x6b454000a4a60000	0x80
17	0xd3400019f41ffff9L	0x88

III. ANALYSIS AND RESULTS

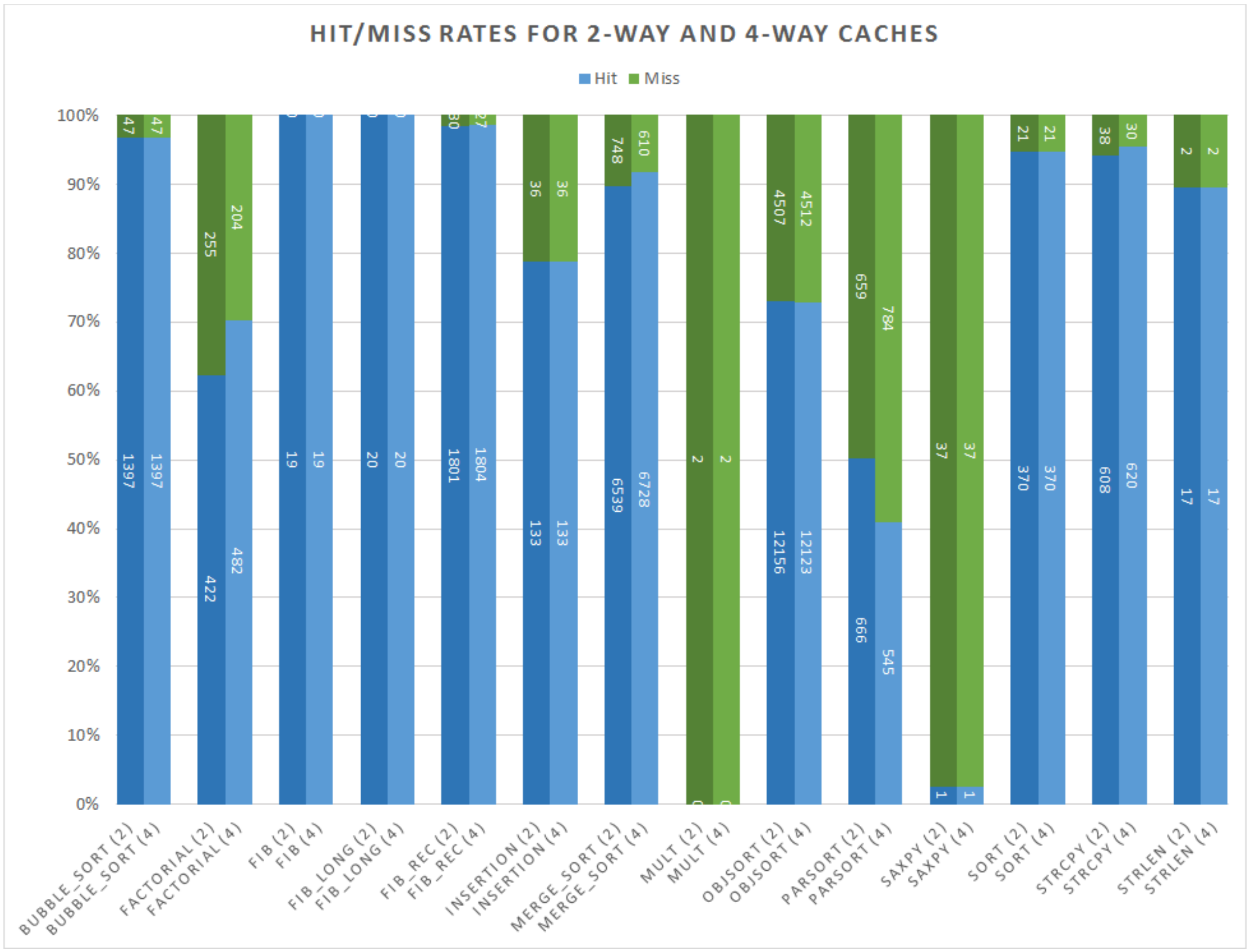
A. *Basic Performance Validation*

To study if our 2-way processor is able to take advantage of instruction level parallelism available to it, we measured the CPI of the processor with a program that consists of 20 independent addition instructions that loop 8000 times. For this program, we observed a CPI of 0.502 which very close to the ideal value for a 2 way system.

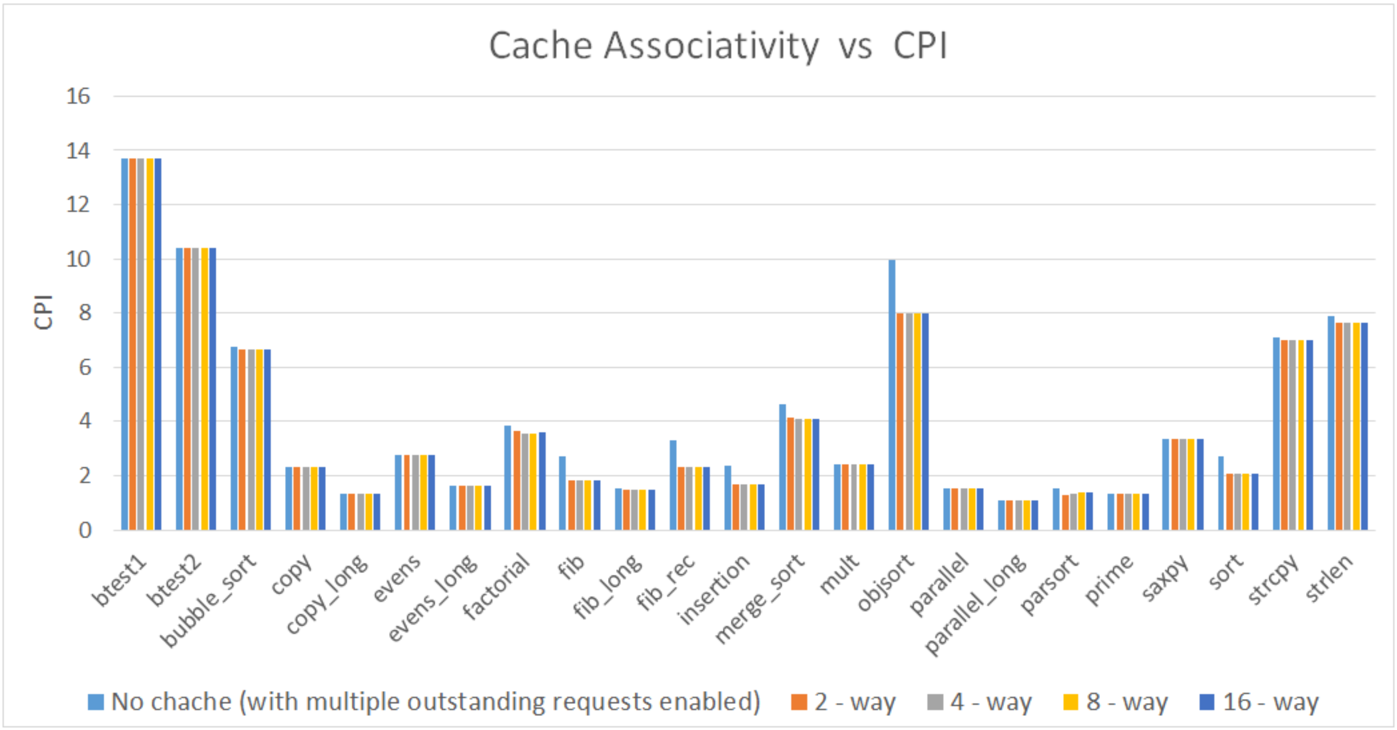
B. *Data cache performance*

Our analysis of different cache set-associativity values indicated that there is virtually no gain in going beyond 4-way. As increasing the associativity beyond 4-way. The graph given below compares the cache hit and miss rates for 2-way and 4-way set associative caches.

One of the interesting things that can be seen from the graph is that **as the hit rate increases, the number of load requests sent to the cache also change.** This is mainly due to the interaction of the cache controller with the LSQ. Different hit rates affect how long loads stay in the load queue - which affects the rate at which instructions are retired among other things.

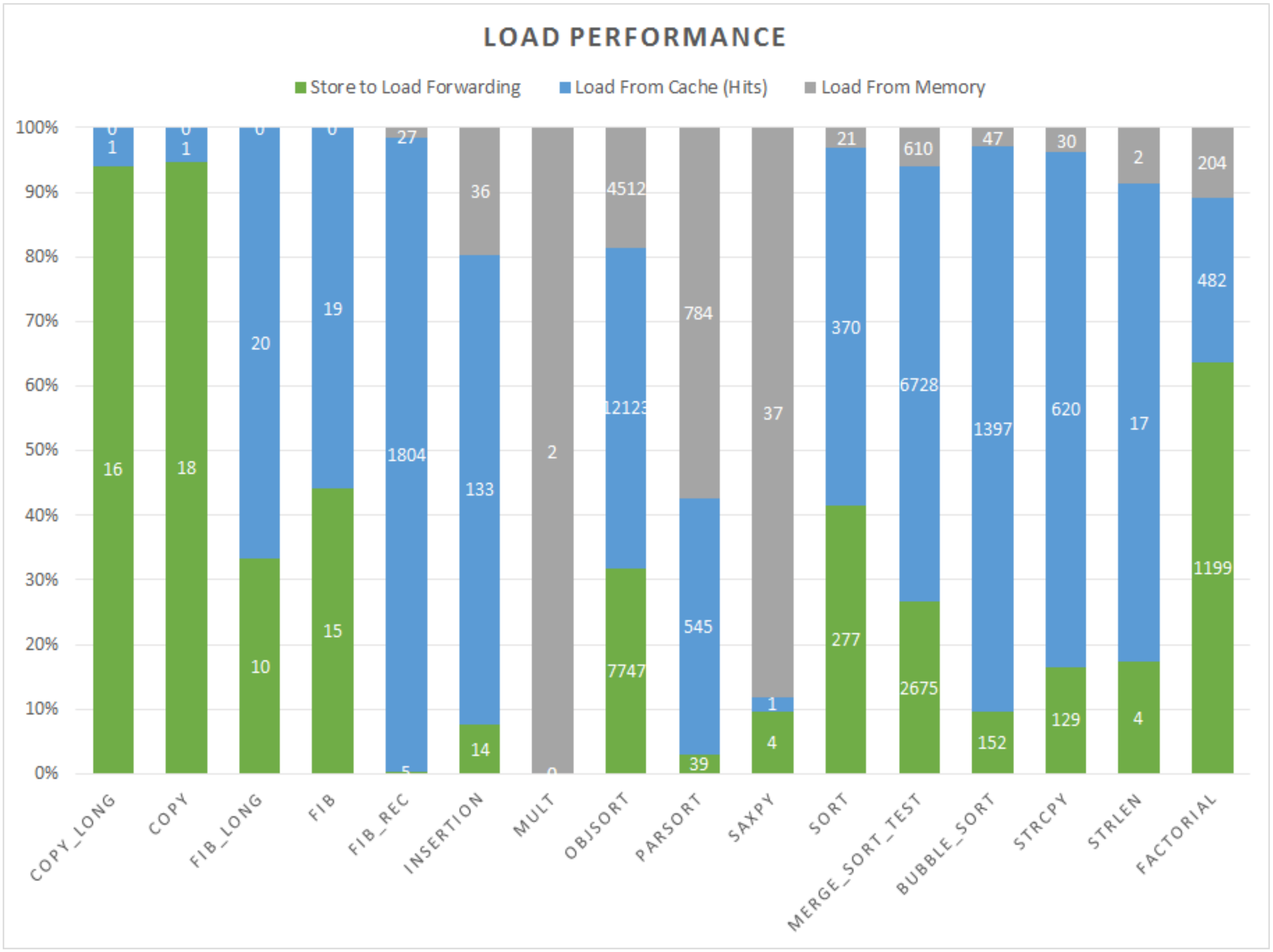


To see how increased hit rate translates into CPI increase, we also analyzed the CPI of the different programs under different cache associativity values. As it can be seen, not appreciable change is see beyond 2-way. A more interesting insight is that for the smaller programs, the large load and store buffers(16 each) with load-to-store forwarding and multiple outstanding requests, removing the cache will have little effect. This can be corroborated by looking at the next section which analyzes overall load performance.



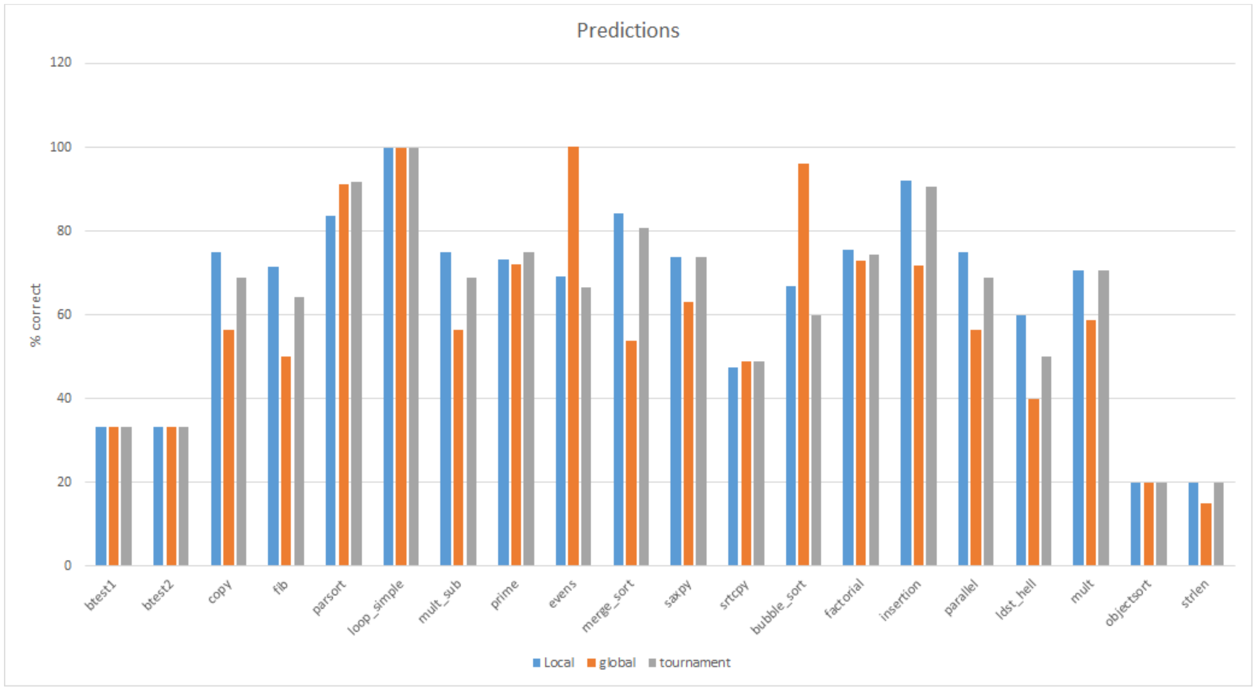
C. *Overall Load Performance*

To gauge the effectiveness of our LSQ implementation, we analyzed what percentage of the loads are forwarded directly from the store buffer. The graph given below clearly shows that a significant number loads are directly forwarded from the store buffer. Furthermore, the caches also do a good job of reducing memory traffic. Overall, it can bee seen that for most programs, with our set-associative cache and load-to-store forwarding, **less than 20% of the loads actually go all the way to memory**.   However, in this project, where the cache is just one cycle away, such effective load-to-store forwarding is less likely have significant effects on CPI. Since disabling forwarding in our LSQ is a non-trivial task, we have not shown the CPI gains from forwarding.



D. Branch predictor performance

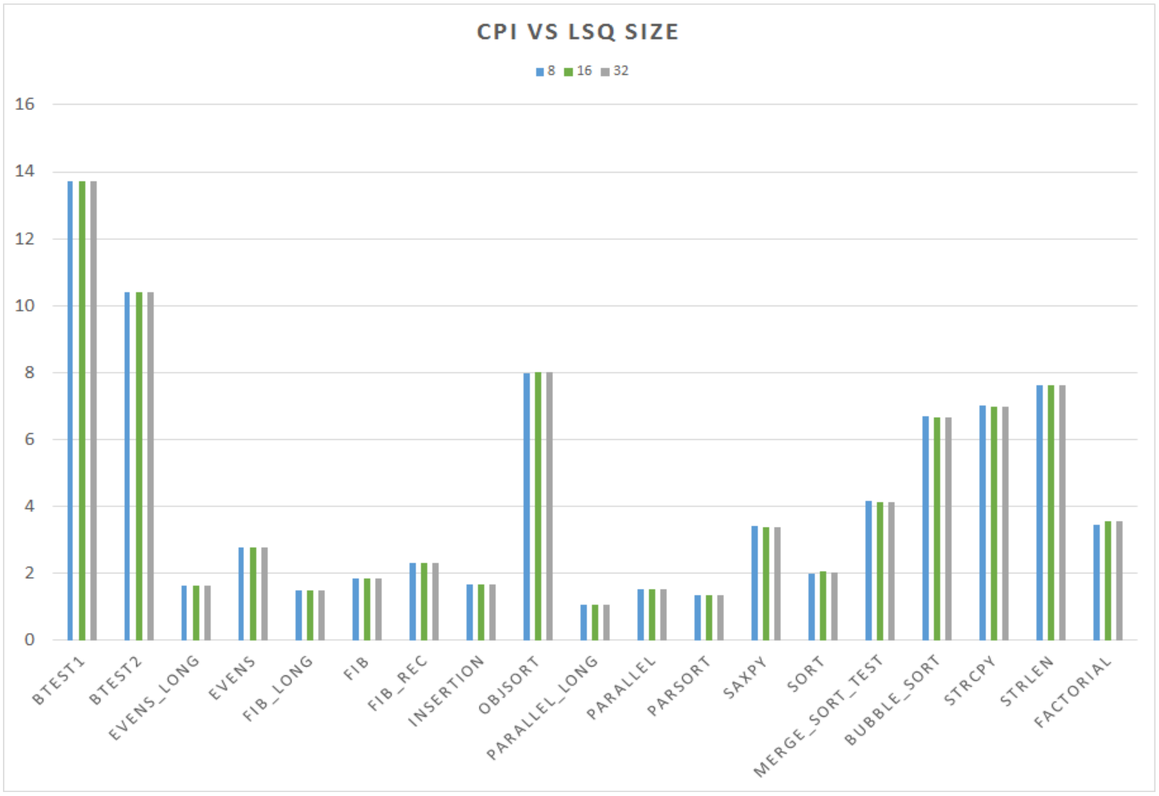
The graph below shows the percentage of correct predictions on different programmes for the branch predictors that we have implemented. A common trend that we can see from the graphs is that the local predictor does better in most of the cases. We also implemented a tournament branch predictor and observed that it performed worse on our benchmarks as compared to a local predictor but it seemed to perform better than global predictor. One uncommon result that we saw was that the global predictor performed better on bubble sort. This can be attributed by the fact that aliasing among branches helps improve performance.



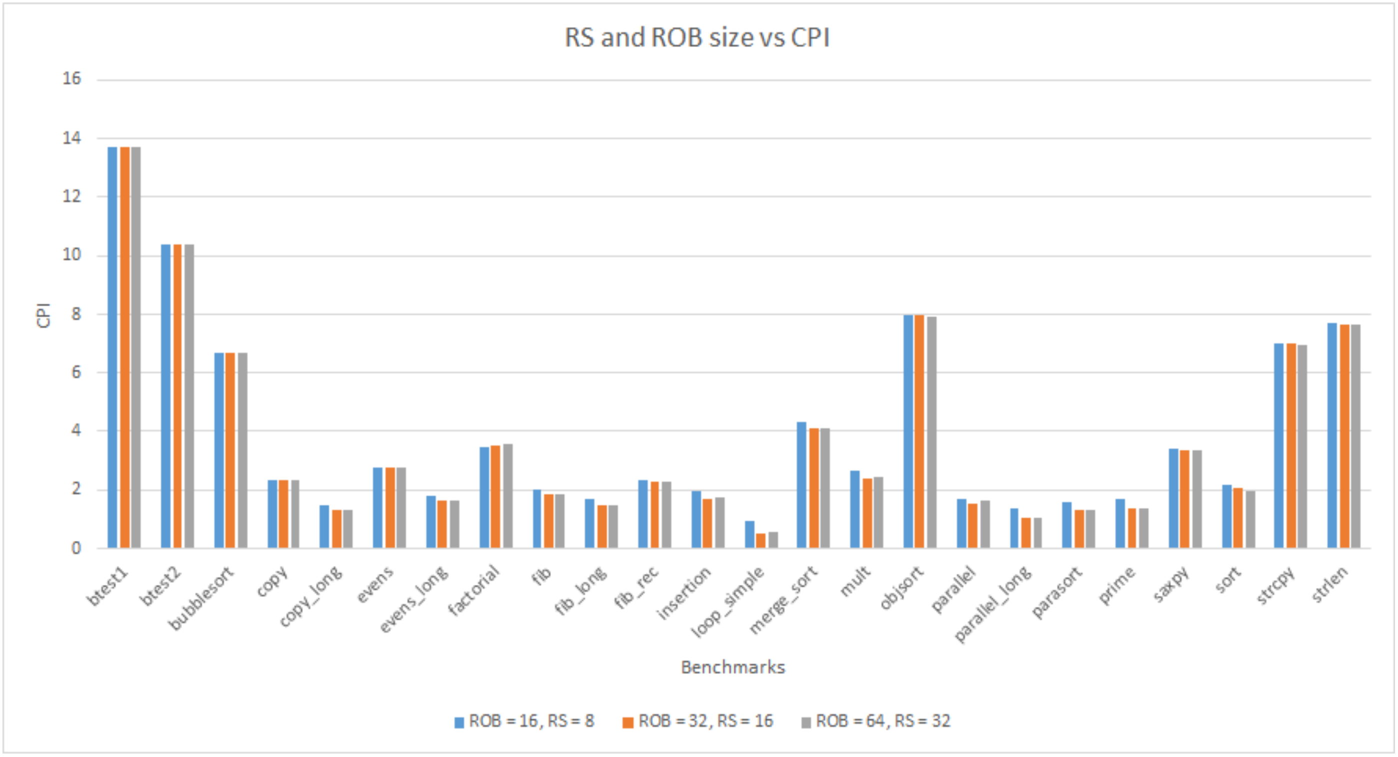
E. *Buffer sizes (ROB, RS and LSQ)*

LSQ Size

Store and load queue sizes of 8, 16, and 32 each had roughly the same CPI. Having small queue sizes are likely to cause excessive stalls due to structural hazards. Since the LSQ is not part of our critical path, we left the store and buffer sizes to be of size 16(which was the default value were testing under) even if it has almost no appreciable benefit compared to a queue size of 8.

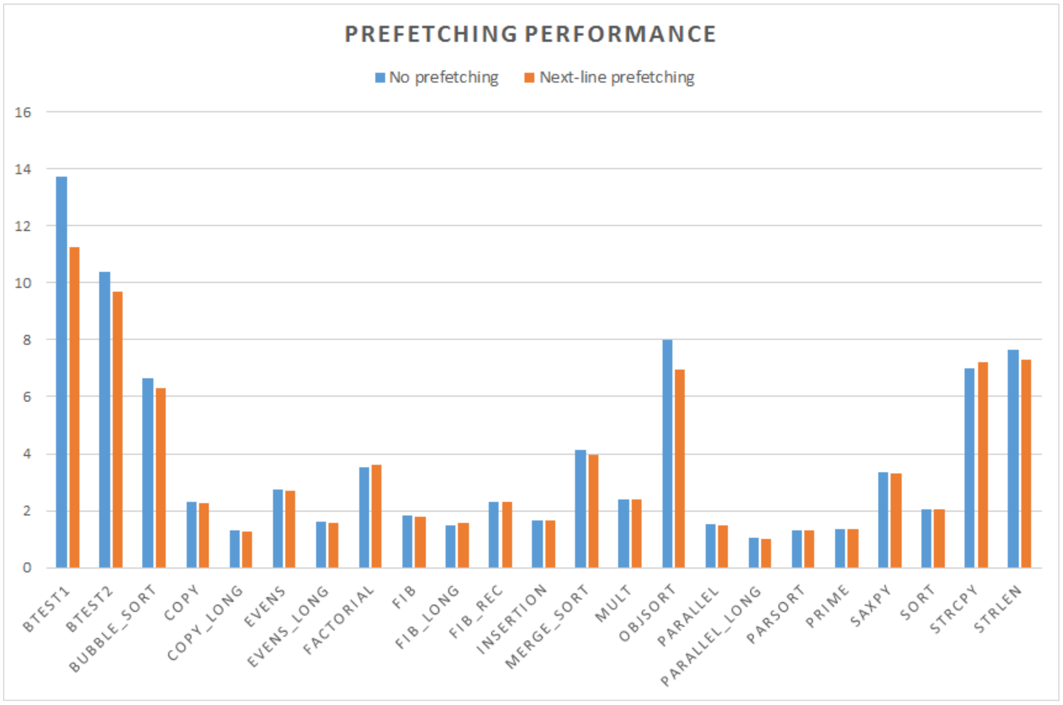


ROB and RS size
Increasing the RS size from 8 to 16 and the ROB size from 16 to 32 shows slight reduction in CPI. Any further increase does not being performance gains. The effect of the different sizes on clock period was not studied due to time constraints.



F. *Instruction Prefetching*

We studied the possible performance gains from instruction prefetching. For this experiment, we configured our cache to be 4-way associative. The graph below shows that most of the programs have very small gains from prefetching. The programs that benefit the most from instruction prefetching seem to be the ones with a high branch-mispredict rates. On the other hand, the performance of the strcpy and factorial programs degraded – possibly because instructions are being evicted by prefetched instructions. We were able to observe that prefetching beyond two instructions did not bring any performance gains. Integrating the prefetch required the clock period to be increased to 10.1ns due to the associative cache.
Since prefetching was added at the last minute and synthesis did not finish in time for the final deadline, the processor in the submission branch does not have prefetching turned on.



IV. FINAL THOUGHTS

A. *What we did right*

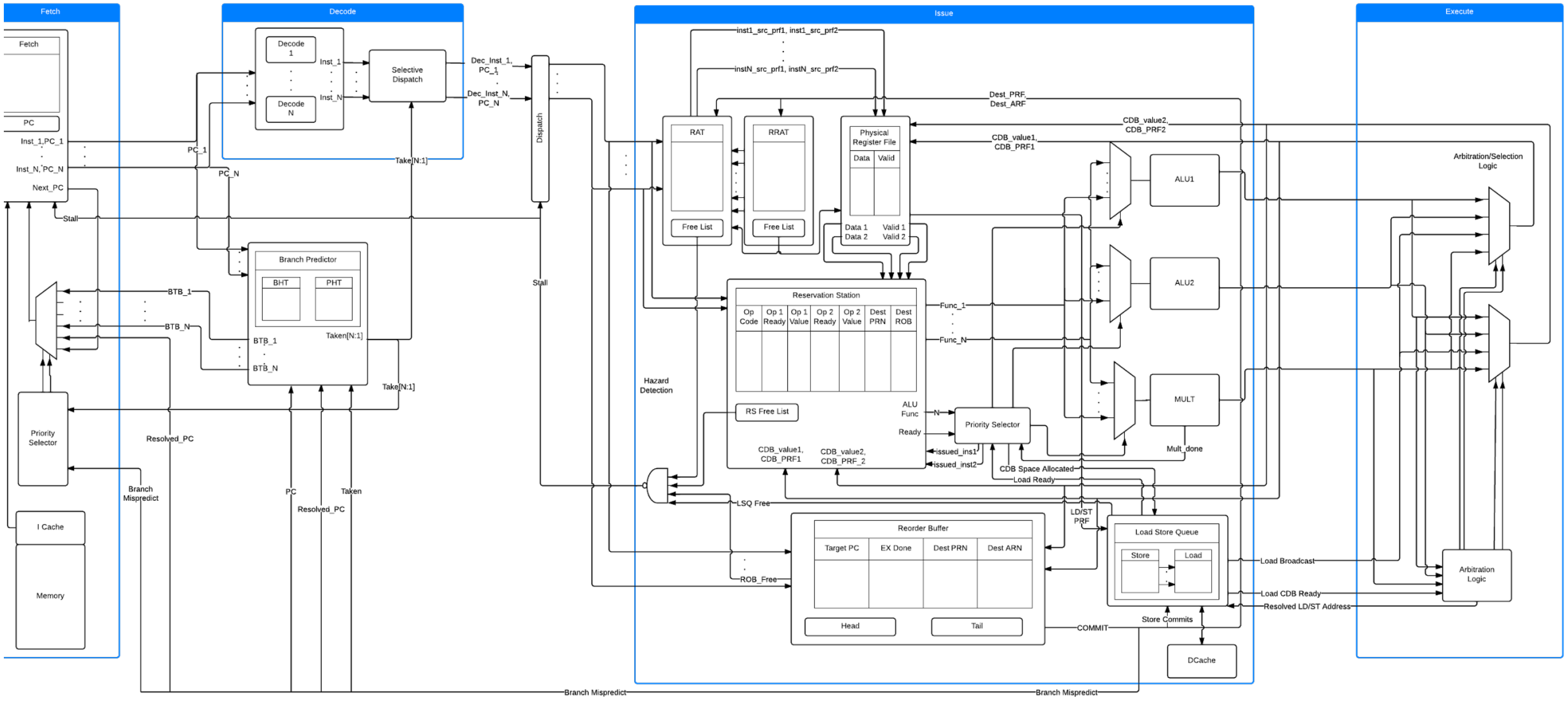
- Modules were unit tested to a reasonable degree before integration. Even if this did not eliminate all the bugs, it saved us a lot of post-integration grief
- During integration testing, we did turned of most of the modules outside of the core until we were eliminated a significant number of the bugs internal to the core.
- We had automated regression testing suit which enabled us to quickly catch changes that broke the design
- We met very often and updated one another on our progress and plans frequently

B. *What we did wrong*

- Some of the individual modules were not synthesized separately. As a result, we did not discover some inefficient implementations until the latest phase of the design.
- Some of our modules did not follow the best-practices outlined by the course staff. For examples, our reset logic was gated and we did not have sync_set_reset compiler directives at the necessary places. As a result, we had to spend hours trying figure out why our design was failing some of the test cases post-synthesis.
- Towards the end of the project we spent time optimizing the clock cycles by adding pipeline stages. However, we did not properly analyze what modules were limiting our CPI and take corrective measures. As a result, our design was not tweaked for get the maximum possible CPI for our design. We also did not consider ways to handle back-to-back dependencies.
- The instruction prefetcher was not sufficiently tested by the submission deadline and hence we decided to turn it off on our submission to the autograder in favor of guaranteeing correctness.
- We did not integrate our RAS and retirement RAS in time which led us to submit our final design without a RAS.

GROUP 9

Ram Srivatsa Kannan
Kartik Joshi
Salessawi Ferede Yitbarek
Thomas Zachariah
Walter Zarate


Fetch
Decode
Issue
Execute
Fetch
PC
Inst_1,PC_1
Inst_N, PC_N
Next_PC
Stall
Priority Selector
I Cache
Memory
BTB_1
BTB_N
Resolved_PC
Branch Mispredict
PC_1
PC_N
Decode 1
Decode N
Inst_1
Inst_N
Selective Dispatch
Take[N:1]
Branch Predictor
BHT
PHT
Taken[N:1]
PC
Resolved_PC
Taken
Dec_Inst_1, PC_1
Dec_Inst_N, PC_N
Dispatch
Stall
inst1_src_prf1, inst1_src_prf2
instN_src_prf1, instN_src_prf2
Dest_PRF, Dest_ARF
CDB_value2, CDB_PRF2
CDB_value1, CDB_PRF1
RAT
Free List
RRAT
Free List
Physical Register File
Data
Valid
Data 1
Data 2
Valid 1
Valid 2
Reservation Station
Op Code
Op 1 Ready
Op 1 Value
Op 2 Ready
Op 2 Value
Dest PRN
Dest ROB
RS Free List
ALU Func
Ready
CDB_value1, CDB_PRF1
CDB_value2, CDB_PRF_2
Func_1
Func_N
N
Priority Selector
issued_ins1
issued_inst2
CDB Space Allocated
Load Ready
Hazard Detection
LSQ Free
ROB_Free
Reorder Buffer
Target PC
EX Done
Dest PRN
Dest ARN
Head
Tail
COMMIT
LD/ST PRF
Load Store Queue
Store
Load
Store Commits
DCache
ALU1
ALU2
MULT
Mult_done
Load Broadcast
Load CDB Ready
Resolved LD/ST Address
Arbitration Logic
Arbitration/Selection Logic
Branch Mispredict