

EECS 470 Final Project Report

Group 1

Shibo Chen, Zhen Feng, Chin-Wei Hsu, Yueying Li, Wenhao Peng

ABSTRACT

In this project, we have designed a 3-way superscalar R10K processor together with performance-enhancing features, including: LSQ (load store queue) with internal forwarding, prefetch, set associative caches, and others. Design choices, testing plans, and justifications are presented. It was designed, implemented, and tested by our group, and achieved a clock period of 11.3ns.

INDEX TERMS

3-way Superscalar, R10K, LSQ with Internal Forwarding, Hybrid Branch Resolver, ICache with Prefetching, Associative DCache

I. INTRODUCTION

A. Project Background

EECS 470 uses a subset of Alpha64 ISA to design microarchitectures. The design is done in teams of five. Serving as a major design experience, students implement in System Verilog some of the processor designs discussed in class.

B. Design Choices

We implemented an R10K MIPS 3-way superscalar pipelined processor. The basic technical requirements include the following four items, all of which we have met in the project:

- 1) An instruction cache and a data cache cache. Since the base memory will have 100ns latency associated with it, we implemented cache system improve this.
- 2) Multiple functional units (FUs) with varying latencies. In order to improve the cycle time, we implemented multiple FUs. Branch target and address calculations is split into separate units to enhance performance.
- 3) An out-of-order implementation. In order to make use of the wasted cycles in in-order processors, we implemented out-of-order execution (or dynamic execution). In computer engineering, out-of-order execution is a paradigm used in most high-performance central processing units. Instructions are sent to the execution stage in an order other than the program order. It is noteworthy that an in-order retirement is ensured and exceptions are correctly handled. We implemented the out-of-order scheme with the assistance of reservation station (RS) and reorder buffer (ROB).
- 4) Dynamic branch prediction. We implemented a advanced branch prediction mechanism for better hit rates.

We implemented **10 advanced features** which include 1 major feature and 9 minor features:

- 1) 3-way Superscalar execution
- 2) Partial Early Branch Resolution for instruction fetch
- 3) A load-store queue (LSQ)
- 4) Store to load forwarding in LSQ
- 5) Out-of-order memory access issue
- 6) A tournament branch predictor in Branch Resolver (TBP)
- 7) A branch history table in Branch Resolver (BHT)

- 8) A return address stack in Branch Resolver (RAS)
- 9) Instruction prefetching in Icache
- 10) Associative Dcache

The design of these features will be discussed in our design section and in the context of their host-module. The evaluation will be demonstrated in the Evaluation section.

Branch Misprediction and recovery is a significant factor in the performance of wide-issue dynamically scheduled superscalar machines. We implemented a sophisticated branch prediction mechanism with several advanced features. In case of misprediction, while a full-scale early branch resolution requires us to squash instructions in RS and FUs and also recover from a dedicated state buffer, we deem that as way too complicated based on our design. However, we still want to fetch instructions from the memory earlier to hide memory access latency, so we implemented early branch resolution for instruction fetch to start fetching instructions earlier when there is a outstanding misprediction.

Moreover, we implemented icache prefetch to speed up the memory read time and a set associative data cache to increase cache hit rate.

Finally, we implemented a Load-Store Queue (LSQ), and it will have a size of 20% of the size of our Reorder Buffer (ROB). The LSQ has load to store forwarding, which allows loads to execute faster when they are dependent on a pending store.

Our goals are summarized as follows: 1) A correct implementation of out-of-order Alpha64 processor; 2) Have a good IPC; and 3) Have a good clock period.

II. DESIGN AND IMPLEMENTATION

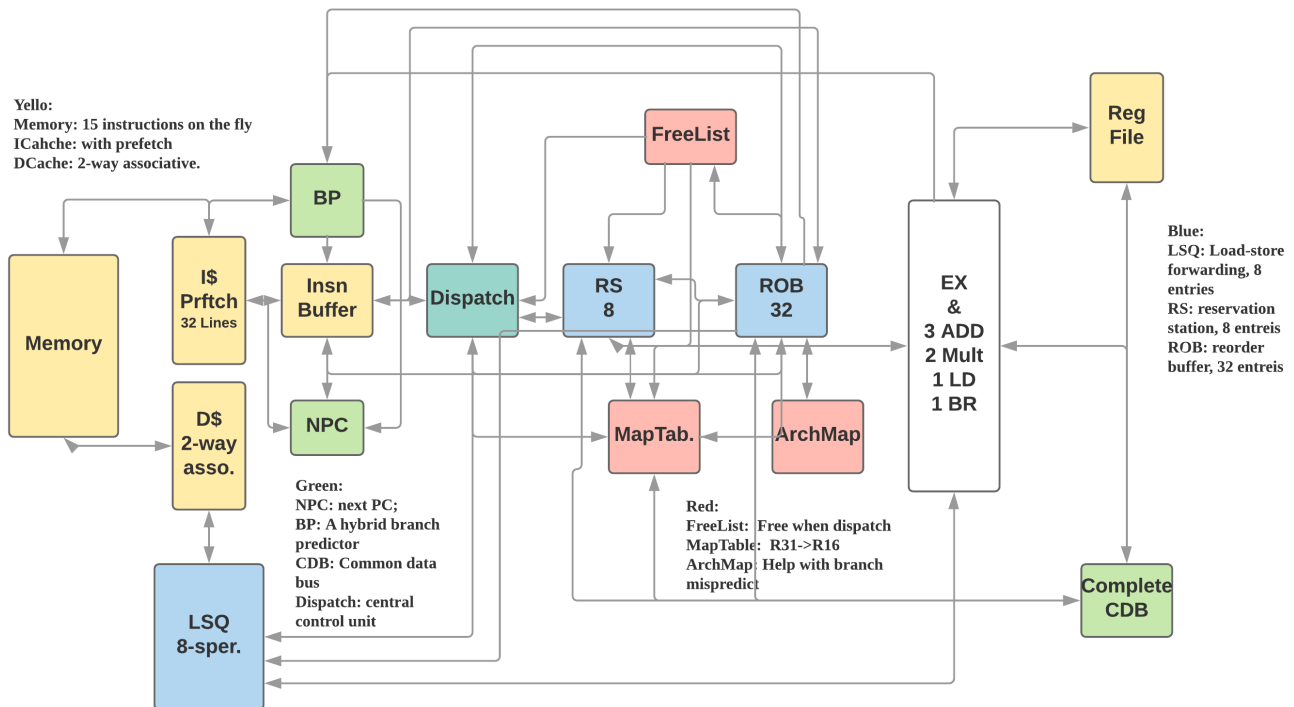


Fig. 1. High level overview of our architecture

As shown in Fig. (1), we implemented our processor design in SystemVerilog using Synopsys tools for simulation and synthesis. When designing our processor, we made strategic choices of the additional features in order to better our design.

A. R10K

R10K is a popular out-of-order algorithm. We choose to implement R10K instead of P6 for its elegance. The R10K only needs tag to be passed between physical register file and function units, whereas the P6 need to pass the entire data from architectural register file, reorder buffer, reservation station, function units, etc.

B. Execution and Functional Units

We used 7 Functional Units (FUs): 3 ALUs, 2 pipeline multiplier, 1 load/store address calculator and 1 branch address calculator. The reason for this assignment is that usually there are not many load/store and branches executed at the same time. There are four types of functional units used in our processor. Each of these units reads its operands from one input buffer named as RS content from RS and reads its register value the register file, performs its operation, and then sends the result to the complete stage. The complete stage then back to the register file through the common data bus (CDB). The latency of the functional units vary depending on the type of operation. It is worth noting that the LD/ST unit has an dynamic latency due to the potential misses we have in the ICache/DCache. The ALU unit is used for every arithmetic or logical operation except multiplication. The LD/ST unit will calculate corresponding address for LD and store instruction and send the data and address to the load-store queue. Also, the load-store queue will send its complete signal and load data to it. The Branch unit calculates the target for a branch instruction and determines whether the branch predictor mispredicted the location or direction of the branch. Finally, the MULT unit is used for multiplication instructions. It completes multiplications in four cycles, and is pipelined to allow for four multiply instructions to be in flight at a time. Also, one scheduler will compute from valid bits in the third stage of two multipliers and together with the "will complete" signal from LSQ to get totally how many instructions of mult and LSQ will finish next cycle. Reservation station will reduce issued instructions from this number to avoid structure hazard in CDB in next cycle. For the critical path analysis, we find that there is a long path from EX stage to register file and then come back with value and goes into multiplier. It results in 11.3ns period. Since there are several critical path in different module in 11.3ns, so we do not add additional latch to break the critical path.

C. DCache (*Features Included: Set Associative Cache*)

The DCache is coded to be a programmable 2^N -set-associative, write-through, allocate-on-write cache. We allowed the set associativity N to be tuned because we might need to trade hit rate for clock period. A Least-Recent-Used logic is implemented to allow evicting the least used block. Thanks to the LSQ internal forwarding, the DCache is able to handle read and write separately while functioning correctly. In other words, write requests are performed in sequence in one cycle from the DCache point of view. At the same time, load instructions are performed in sequence, so load misses will block subsequent instructions. The load response and tag management is the same as ICache, resulting in a non-blocking operation of our DCache. However, we did not have time to redesign the LSQ logic to allow non-blocking load. Because of our speculative execution with branches, sometimes mispredicted branches result in invalid read addresses coming to DCache from LSQ. In these cases, if we allow the invalid read requests to memory, memory will not respond and the whole system will stall forever. Instead of allowing the bad address to propagate through DCache, we added a logic mask to the addresses of load requests. Acting like a bandpass filter, the invalid loads are reflected back with a default data deadbeef. This piece of data, along with the invalid address, will be flushed by branch mispredict recovery logic. With this, we have avoided unnecessary eviction of our data in DCache and have saved many cycles on this invalid load request.

DCache was synthesized for direct-mapped, 2 ways, 4 ways, 8 ways, and 16 ways when it was first designed. The clock periods vs number of ways is summarized in Table IV.

TABLE I
DCACHE GOING BEYOND 2-WAY ADDS TO TOO MUCH OVERHEAD ON CLOCK PERIOD CONSIDERING OTHER MODULES.

associativity	clock period/[ns]
direct	8
2-way	9
4-way	10
8-way	14
16-way	28

We tried direct-mapped, 4-way and 8-way in behavioral simulation to see the effect of increased associativity. The effects are only seen in the three large public cases: fib_rec, objsort, and parsort. We also borrowed the merge sort sample from decaf470. The CPIs are compared in Table II.

TABLE II
CPI FOR THREE PUBLIC CASES AND MERGE SORT SHOWS DIMINISHING EFFECTS GOING BEYOND 2 WAY ASSOCIATIVITY

	merge_sort	fib_rec	objsort	parsort
direct	1.64	1.78	2.38	1.09
2-way	2.02	1.82	2.26	1.05
4-way	2.13	1.81	2.26	0.98
8-way	2.13	1.82	2.20	0.97
16-way	2.00	1.81	2.21	1.02

Because added associativity does not significantly bring down CPI, taking clock period into consideration, we settled with the 2-way design.

D. ICache, Instruction Buffer, and NPC generator (**Features Included: Icache Prefetcher**)

We have three modules dedicated to instruction fetch stage: icache, instruction buffer and NPC generator.

ICache is important in order to lower the memory access cost. In this project, we implemented 256 bytes direct-mapped icache with 32 blocks. Since we tend to execute sequentially, we will prefetch the next outstanding instruction request within the next 16 address window and it can be easily reconfigured to prefetch less or more entries. The CPI is significantly reduced by the prefetch functionality, which we will demonstrate in the later sections.

The request results returned from icache will go into instruction buffer. It's fairly important to start buffering instructions early after misprediction or structural hazards so that we can recover to full speed after hazards have been resolved.

NPC generator will decide what instructions need to be fetched in the next cycle. It makes decision based on how many instructions have been fetched in the last cycle, whether there is a mispredict or not and whether there is a taking branch in the last fetched few instructions.

E. Dispatch

Dispatch is the central module between instruction buffer, reservation station, reorder buffer, load-store queue, free list and map table. Its functionality is to maintain the three-way pipeline, and to stall on the occurrence of any structural hazard from ROB/RS/LSQ/Instruction Buffer.

It is important to allow the pipeline to function smoothly, and so when we see a potential stall on any other unit, we will reduce the number of instructions to dispatch in that cycle.

In this project, we used a Dispatch module with a decoder embedded, so we could send out the operation type and their corresponding register numbers at this stage to the reservation stations.

F. Map Table, Free List and Architectural Map

Free list, Map table and Architectural map are important modules that enable R10K structure to work without heavy data transmission. In R10K structure, data is stored in physical register rather than architectural registers. Map table is the module that keeps track on the mapping between architectural registers and physical registers. Map table also is responsible to keep track of the states of each architectural register and send the ready bits to RS. Architectural map is a backup version of Map table, which only updates when instructions retire. If a branch misprediction happens, Map table will be updated as what architectural map is. Free list keeps track on the physical registers that are free to use. It keeps a list of free physical registers and update in the dispatch stage and retire stage. It is responsible to give out a physical register for mapping when a instruction that needs destination registers is dispatched.

Map table module stores the physical register index corresponding to each architectural register, and a ready bit to indicate whether the data is ready in that physical register. When a instruction dispatch, Map table will update the physical register index and set the ready bit to 0. When a instruction is complete, Map table will get the index of the physical register and set the corresponding ready to 1 if found.

Architectural map has a similar structure as Map table, but no ready bits. It updates when an instruction retires, mapping the retired physical register to the retired architectural register.

Free list module has a list that contains the indices of physical registers. The size of the list is same as that of ROB because the maximum number of free physical register is equal to the size of ROB. Free list use a head and a tail to indicate the range of free physical registers. The head will update on dispatch, which means that the physical register is now in use. The tail will update on retire, which means that the released physical register is free now. To flush back when there is a branch misprediction, Free list will give out the head to ROB, which is then used to indicate the Free list status that needs to be recovered when the head comes back.

G. Branch Resolver (Features Included: Tournament Branch Predictor, Branch History Table, Return Address Stack)

We have a pretty sophisticated branch prediction mechanism to increase prediction accuracy. In branch resolver, there are three parts that help with branch prediction: a tournament branch predictor, a branch history table and a return stack.

In tournament branch prediction, we keep track of both global branch history and local branch history. For each entry, we keep a 2-bit branch predictor along with a one bit choice predictor to decide whether global prediction history or local branch history may achieve a high prediction accuracy for this entry. It helps us to decide whether we should take the branch on conditional jumps/branches.

Branch history table keeps track of where we will likely jump to on a each-location-basis. It gives hints on where this branch instruction may end up with and helps us with jumps on a value in the register which we may not have resolved during instruction fetch stage.

Return stacks yield higher prediction accuracy on return instructions because there is a good pattern to follow: we will likely return to the last place we called `jump_to_subroutine` or `branch_to_subroutine`. We push a return address when `jump_to_subroutine` or `branch_to_subroutine` are called and pop an address and branch to it when a return is called.

When we are unable to decide the address we need to jump to because it is the first time we execute this PC, we use the value in the register to speculatively branch. It works when we have already resolved this value in the register or it stays intact after last time.

We also tested different number of entries in branch predictor and branch history table. The difference is not significant so we land with 32 entries.

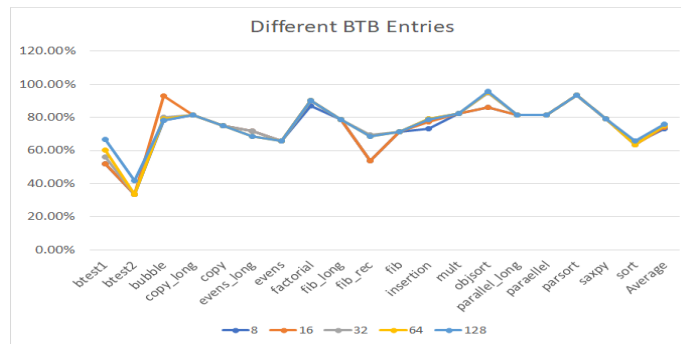


Fig. 2. There is no significant difference in prediction accuracy with different number of entries in predictor or branch history buffer.

H. Load Store Queue (features included: LSQ, store to load forwarding, out-of-order memory access issue)

Load-store queue is critical to ensure that memory access is in-order. We implement a store queue with store-to-load forwarding. To ensure correctness, load instructions can only be completed when all older stores have calculated their address. Also, since there is no load queue, only one load instructions is allowed to be in execution.

The store queue records the information of a store instruction, which contains ROB index, address, data value, address ready bit and a valid bit for each row. It use head and tail to indicate the valid rows of the store queue. When a store instruction is dispatched, the store queue update the head, meanwhile put the ROB index into the row. The address and data value is known in the stage of execution, where the address bit will also be set to 1.

When a load instruction is dispatched, LSQ module will tell RS the load position, or *age*, of that load instruction. This age is used to determine the address ready bits that have to be checked when the load is executed. When a load comes in during execute stage, the load busy will first be set to 1. This signal is sent to RS and be used to ensure that no load will be issued until this one is completed. If all SQ address before this load is ready, the module tries to find out if there is any matched address in SQ and thus internal forwarding can be performed. Otherwise, LSQ will ask D-Cache for the address and wait for it to come back. If there is a hit in LSQ forwarding, then one cycle can be saved for this load instruction.

LSQ is designed to retire one store maximum to ease the structure of D-Cache. Because the size of the store queue is much less than ROB, there is a possibility that store is full and cause structural hazard. Since the maximum number of store instructions can be dispatched is 3, whenever the number of empty entries of the store queue is less than 3, LSQ will need to tell dispatch module to stall.

The size of LSQ is a parameter that can be changed. We try the size at 4, 8 and 16 for LSQ. The only effect of this size change is stall caused by structural hazard. However, we observed that the store queue is rarely full when the size is greater than 8. Therefore, the CPI or IPC improvement is very little for size greater than 8. We choose to use size 8 in the end. To our surprise, we found that the clock period actually is quite constant for different size. The synthesized clock periods are 3.5ns, 3.6ns and 3.7ns for size 4, 8 and 16 respectively.

I. Reservation Station

We built a generic reservation station that can store all kinds of instruction in any reservation station. This could reduce structure hazard in the RS compared to specified with function types in each RS unit. Each cycle, at most 3 instruction will go into RS and the corresponding ready status will comes from map table. The CDB will update ready bit in every valid RS unit in every cycle. To decrease latency in issue stage, we use next ready bit instead of ready bit together with busy status bit to judge issue logic so that instruction could be issued right in the cycle that CDB broadcast completion of the dependent register. The allowed issue numbers depends on the EX stage as mentioned above. The RS also send

numbers can be dispatched back to dispatcher to avoid structure hazard in RS. The issue priority is as instruction types from branch, load or store, multiply to alu, because branch is critical in instruction flow and load or store usually takes longer cycles to finish. RS will send information to each FU with valid bit to tell them if there will be work next cycle. The RS will always issue the youngest load instruction among all load instruction to avoid inter dependence loop between load and store in LSQ, this may stop the program in some edge case such as LD ST LD. This is achieved by comparing age in LSQ of all ready load instruction. It can issue load instruction with same ages in LSQ out of order. Also, it will stop issue load logic when LSQ has not finished the last load to avoid structure hazard in LSQ . To avoid long combination logic across different module, we latch input of load busy status from LSQ to cut the long path. This will introduce one cycle latency if there are continuous load instruction. Also, we have to add one register to forbid issue load in two continuous cycle because we latch the busy status of LSQ.

For the size selection, We try the size at 8, 12, 16. Due to its generic unit type and only 7 function units, the increasing size does not surely decrease CPI and increase CPI in some cases such as "objsort". This is because the issue priority is based on instruction types. In some case, the continuous high priority instruction will stall the low priority instruction with younger ages for long time, and result in bottle-neck in retire stage. This effect may be worse when RS is larger. Also, in many cases, the CPI does not change with RS number. The synthesized period is 8.33ns in size of 8 and 11.33ns in size of 16. To get better clock period, we choose size of 8.

J. Reorder Buffer

The reorder buffer helps to buffer the instructions and keeps track of the instructions in the processor in program order. Each cycle, three instructions (maximum) are retired from the processor, dictated by the head of the ROB. The ROB also keeps track of which instructions have completed. We have swept ROB size across 32, 64, 128, and 256 and have found that the CPIs are almost the same, except that in copy_long, the CPI improved a little when going from 32 entries to 64 entries of ROB. CPI vs. ROB size is summarized in Table III. Going beyond 32 does not help, but having only 16 entries will make ROB a bottleneck for performance.

TABLE III
CPI FOR ROB SIZE SHOWS NO IMPROVEMENT BEYOND 32 ENTRIES

	copy_long	fib_long	merge_sort	objsort
16	0.52	0.49	2.05	2.32
32	0.47	0.49	2.02	2.26
64	0.44	0.49	2.02	2.26
128	0.44	0.49	2.02	2.26
256	0.44	0.49	2.02	2.26

K. Feature: Early Branch Resolution for Instruction Fetch

We broadcast misprediction signal to instruction fetch unit as soon we received a instruction misprediction signal from the completion unit. While we continue fetching new instructions starting with the new NPC, we need to lock down the instruction buffer and stop dispatching before the standard misprediction signal is issued. Comparing to the generic design where misprediction signal is issued only after the mispredicted branch has been retired, we are able to recover to full speed more quickly.

III. TESTING

Our test is based on unit testing before and after synthesis. Each modules are design and tested using simple test cases to ensure the functionality. After integration of the pipeline, our tests are performed in the following steps: first, to ensure the correctness of some of the basic test cases, we tested using simple program "mult.s". This program is a test case without a lot of ld and st instructions, and could ensure our

functionality before milestone 2. Then, after designing and testing the load-store-queue and DCache, we integrated them and tested with more sophisticated programs. To obtain measurements of performances, we created a automatic script to run through the simulations for multiple programs. Our processor was able to pass all the test programs that were provided. Then further improvements were made to styles to correct post-synthesis simulations.

We also wrote some more programs to test the functionality of certain features. For example, we created programs with fixed-pattern accessed memory to check the functionality of branch predictor. We also created hard branch prediction programs to check if the squash (serial roll back) is working correctly. For DCache, we simulated with one hundred store requests and then one hundred load requests to make sure that everything comes out intact.

We also used some script to create a visualized output of the internal states of some of our modules in the pipeline to check the functionality when doing unit testing.

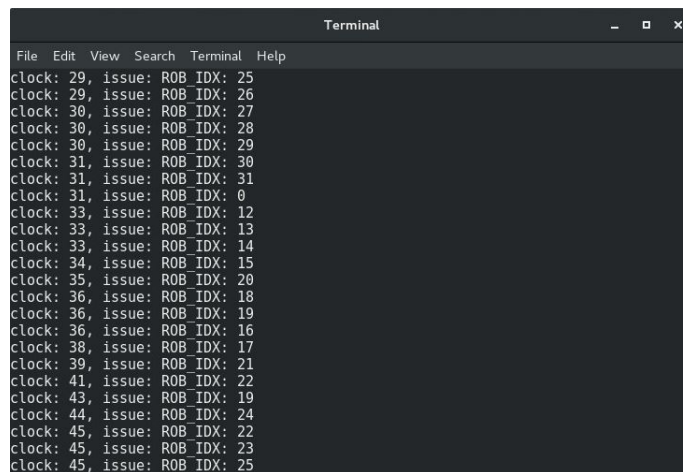
To verify correctness, memory contents after program execution are compared with those coming from the in-order version of project 3.

IV. ANALYSIS

A. Three Way Superscalar

We implement three way superscalar aimed to drastically increase our IPC. In same cases with long loop and weak dependency, the 3 way superscalar shows $CPI < 0.5$ such as "copy_long.s" which is surly much better than 2 way or 1 way. For test cases with lots of short branch and strong dependence, the 3 way superscalar has less advantage compared to 1 way or 2 way but still exhibit some speed up in faster dispatch, faster retire. Also, the fully usage of 3 way relies largely on the performance of branch prediction. Thanks to our good branch prediction ability, we achieve good CPI in some complex cases such as "objsort" with 2.3 CPI. One of our bottle neck in superscalar is that there is only one port to memory. Also,we have only one load-store function units, one port in LSQ and one port from LSQ to dcache to get better clock period. This will limit our CPI when load and store have high density in program. Another disadvantage of 3 way is more complex logic in every stage which results in higher clock period. We optimize critical path by adding latch to break up critical path to sacrifice CPI. Due to the time limit, the trade off of CPI and clock period may not be optimal.

Figure 3 shows that our processor can issue up to 3 instructions out-of-order.



```

Terminal
File Edit View Search Terminal Help
clock: 29, issue: ROB_IDX: 25
clock: 29, issue: ROB_IDX: 26
clock: 30, issue: ROB_IDX: 27
clock: 30, issue: ROB_IDX: 28
clock: 30, issue: ROB_IDX: 29
clock: 31, issue: ROB_IDX: 30
clock: 31, issue: ROB_IDX: 31
clock: 31, issue: ROB_IDX: 0
clock: 33, issue: ROB_IDX: 12
clock: 33, issue: ROB_IDX: 13
clock: 33, issue: ROB_IDX: 14
clock: 34, issue: ROB_IDX: 15
clock: 35, issue: ROB_IDX: 20
clock: 36, issue: ROB_IDX: 18
clock: 36, issue: ROB_IDX: 19
clock: 36, issue: ROB_IDX: 16
clock: 38, issue: ROB_IDX: 17
clock: 39, issue: ROB_IDX: 21
clock: 41, issue: ROB_IDX: 22
clock: 43, issue: ROB_IDX: 19
clock: 44, issue: ROB_IDX: 24
clock: 45, issue: ROB_IDX: 22
clock: 45, issue: ROB_IDX: 23
clock: 45, issue: ROB_IDX: 25

```

Fig. 3. Multiple instructions issued Out-of-order, when running program evens.s, where mult is a long-latency instruction compared to others

B. Branch Prediction Rate (*Related Features: tournament branch predictor, branch history table, return address stack*)

Thanks to our advanced branch predictor design, we are able to achieve a 75.19% prediction accuracy. This is more than 3 times better than the baseline design with no prediction which is 17.36%.

We performed our analysis in an incremental manner: (1)baseline with no prediction, (2) with only tournament branch predictor(TBP), (3) with TBP and branch history table (BHT), (4) Our design (with TBP, BHT and return address stack). We find that all test cases benefit much from the tournament branch predictor. factorial, fib_rec and objsort benefit from the addition of the branch history table. Only objsort benefits from return address stack (from 89% to 95%). This is due to the fact we speculatively return on the value present in the register, which correctly predicts the target even the return address stack is absent. However, this doesn't undermine the value of return address stack. Our high performance in objsort is impossible without return address stack.

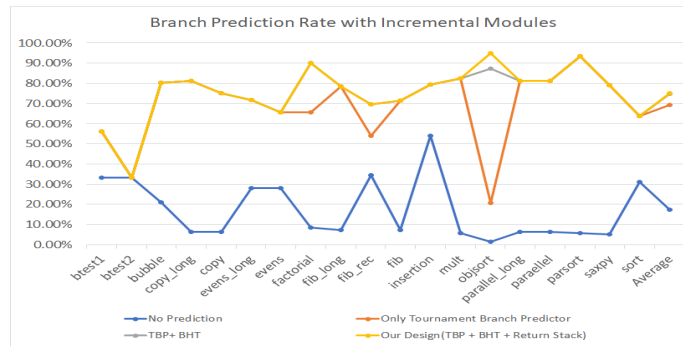


Fig. 4. This figure shows all features included in our design improve the prediction accuracy.

C. IPC Difference due to Different Prefetcher Size (*Related Features: Cache Prefetching*)

The implementation of the prefetcher is of size $2^n - 3$. The reason for this size is due to the priority encoder's implementation. We tried with size 1, 5, 13, and see that the improvement of IPC is program-dependent. For example, we see that the btest1 and insertion performs better under smaller prefetcher size. The prefetcher size performs best at 1 for "parallel long" program. It is due to the extensive noop in this program, which degrades the potential benefits of fetching instructions in advance. In consequence, fetching unnecessary instructions will kick out the good instructions in a small loop and increase the data hit rate. For other programs, we generally see an increase on IPC for larger prefetcher size, which validates the functionality of our prefetcher.

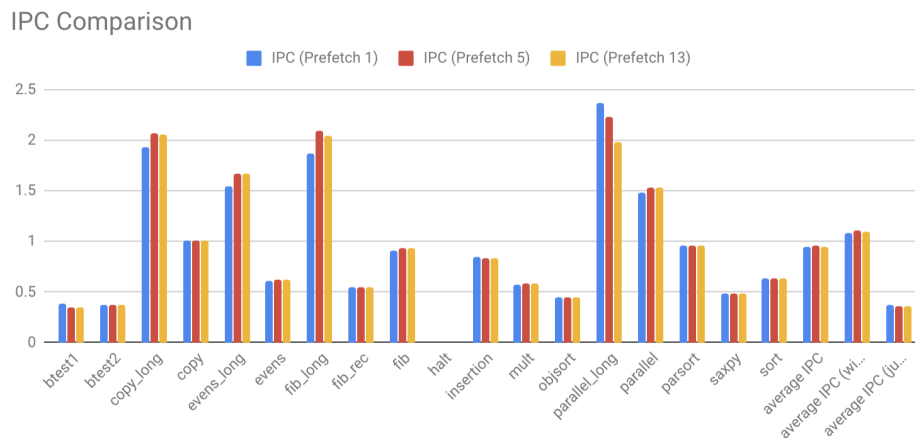


Fig. 5. IPC analysis for prefetcher size

D. Early Branch Resolution for Instruction Fetch (**Related Features: Early Branch Resolution for Instruction Fetch**)

Early branch resolution for instruction fetch boosts the performance by fetching and buffering instructions early. It increases the overall performance by 4%.

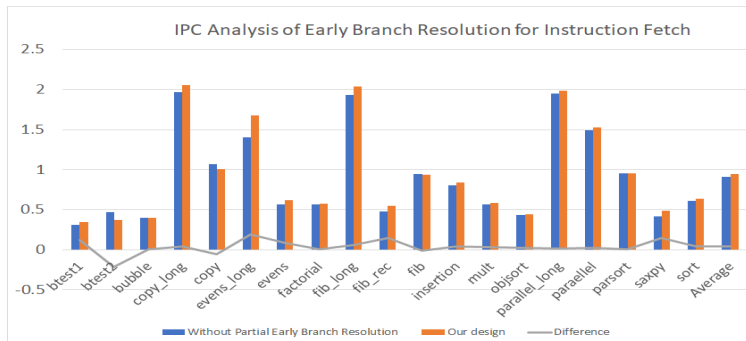


Fig. 6. Performance Improvement of different test cases due to early branch resolution for instruction fetch.

However, we notice that rather than benefiting from this feature, copy and btest2 suffer from the implementation of this feature. This is because we only update branch predictor once the instructions retired. However, due to the fact we are fetching instructions while there are some instructions waiting in ROB, we are unable to use the most up-to-date information to predict, thus we suffer from a drop in prediction accuracy. But as a trade-off, even we may mispredict more, we also recover faster, so most test cases still have net gain because of this feature.

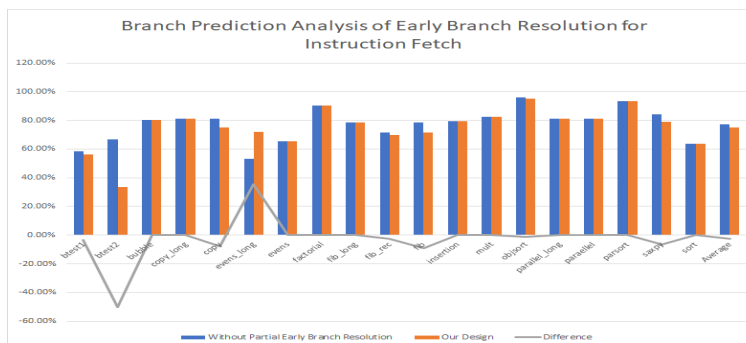


Fig. 7. The effect of early branch resolution has on prediction Accuracy.

E. LSQ and store-to-load forwarding. **Related Features: LSQ, out-of-order memory access issue, store-to-load forwarding**

The implementation of a load-store queue (LSQ) allows the memory-related instruction to be issued out-of-order. Without a LSQ, all the load and store instructions have to be executed in-order, and the next instruction can only be issued after all the previous memory access instructions have been retired. In this case, the benefits of the out-of-order execution processor will be limited by the memory access instructions.

Our LSQ also allow memory access instructions to be issued partially out-of-order. The store queue keeps the data and only put them into D-cache when the corresponding store instruction is retired. By doing this the store instructions can be issued out-of-order without having address ambiguity problem. Although we did not implement a load queue, the load instructions generally can also be issued out-of-order. However, the out-of-order issue can only be allowed when there are multiple load instructions have the same youngest load position with respect to store. If they don't, the load instructions have different

age, then the oldest load must be issued first. Otherwise, some edge case like (LD, ST, LD) will make the second load stuck in LSQ forever and thus the whole program will be stuck. Therefore, to be accurate, we might say that the load instruction can be issued out-of-order *weakly*.

Another important feature of our LSQ module is that we also implement store-to-load forwarding. Since the data can only be written into D-cache and memory when the instruction is retired, the data forwarding for load before the store instruction retires is very important. Without store-to-load data forwarding, every load instruction may have to wait much more cycles before it can be completed.

We analyze the number of store-to-load forwarding and the percentage of the forward hits in the load instructions. The program that has utilized store-to-load forwarding are listed in Table IV. These numbers indicate the importance of store-to-load forwarding. However, it is hard to calculate the cycle number reduction because of this feature since we don't know how many cycles will need to retire all previous store instructions.

In addition, this store-to-load forwarding also potentially make load faster. The cycle number needed for forwarding is 2, while it takes 3 cycles if we have to get the data from D-cache. In summary, by doing store-to-load forwarding in LSQ, the processor benefits from two aspects: 1) no need to wait for store retire. 2) reduce cycle need for loading data.

TABLE IV
LSQ STORE-TO-LOAD FORWARDING NUMBER AND RATIO.

Program	copy	copy_long	fib_long	fib_rec	fib
forwarding number	16	16	11	240	1
forwarding ratio	100%	100%	37.93%	6.29%	3.45%

F. Bottleneck of Pipeline

We did the synthesis using a divide-and-conquer hierarchical approach. Individual modules are pushed first, and then the pipeline is assembled together. Although most modules can be done within 9 ns, there are some critical path extending across modules. We found that the bottleneck of our processor lies in 1) the LSQ, DCache, and memory 2) the Ex/mult and regfile. It is possible to add more pipeline stages to cut critical combinational logic if more time were allowed.

V. CONCLUSION AND ACKNOWLEDGEMENT

To sum, we have designed a three-way superscalar pipelined out-of-order processor, and our clock period achieved 11.3 ns, and the IPCs are summarized in Table V.

TABLE V
IPC FOR PUBLIC TESTS.

btest1	0.347	btest2	0.371	copy_long	2.06
copy	1.01	evens_long	1.67	evens	0.616
fib_long	2.04	fib_rec	0.549	fib	0.936
insertion	0.839	mult	0.588	objsort	0.442
parallel_long	1.98	parallel	1.53	parsort	0.956
saxpy	0.483	sort	0.637		

The report provided a detailed analysis of the features and their effectiveness. Our processor was fully integrated, tested and demonstrated in SystemVerilog. We worked together closely and solved many challenges in this process.

We would love to extend our sincere gratitude for every team member's equal contribution and the EECS 470 faculties and staffs for their support in this project.