A Two-way Superscalar Processor with Out-of-Order Execution and Early Branch Resolution

Ruiyang Zhu, Meitang Li, Tianrong Zhang, Yuan Shen, Kangjia Cai

***Abstract*— Nowadays, out-of-order pipeline processors with advanced features outperform pipeline processors in many aspects. However, there are too many advanced features to consider when designing an out-of-order processor and the performance improvement of these features is not analyzed thoroughly. This report describes an out-of-order processor design with several advanced features. The processor is a two-way superscalar processor with early branch resolution. The processor consists of different individual modules and a main pipeline module using a hardware description language (HDL) named SystemVerilog. The processor achieves about 160% speedup compared to a traditional 5 stage in-order pipeline processor in various test conditions.**

***Index Terms*—Processor design, Processor Architecture**

I. Introduction

Computer architecture is the foundation of designing computers. Computer architecture bridged the gap between electrical engineering and computer science, and it has always been a hot topic in both academic research and industry. In this report, we demonstrate that our design of a two-way superscalar out-of-order processor with early branch resolution can achieve better performance by analyzing different metrics and comparing to the traditional design of a 5 stage in-order pipeline processor.

In this report we propose a design of a pipelined processor for fast branch misprediction recovery and a superscalar width of two. High-frequency pipelined processors experience huge performance degradation under branch misprediction because branch misprediction exposes the long latency associated with instruction memory. And a superscalar processor can provide better instruction level parallelism (ILP), which means the processor can execute more than one instruction at a time and thus achieve a higher speed.

The processor is based on RISC-V (RV32) instruction set architecture and is implemented in R10k algorithm [3] because of its elegance compared to other out-of-order algorithms. This processor is evaluated by mainly four metrics. They are relatively correct implementation of out-of-order processor, cycle per instruction (CPI), a good clock period and program completion time. In later sections, these metrics are calculated and analyzed to prove the efficiency of the processor design.

II. Project Design

The project is designed to implement a fully working processor with advanced features. We then analyze how these features help contribute to the processor's performance enhancement. And if the advanced features do not provide the expected improvement, reasons need to be explained why the expected performance improvement is unmet.

Among various advanced features, our team chose to implement mainly three large categories, with several specific features in each category. The first category is fast branch misprediction recovery, which includes early branch resolution (also known as branch stack). The second category is novel memory access schemes such as non-blocking cache, load-store queue and prefetching. The last category is better instruction level parallelism by expanding superscalar width.

This section will begin by describing the overall system architecture of the processor, and then explain in detail how each feature works and why it is beneficial.

A. System Architecture

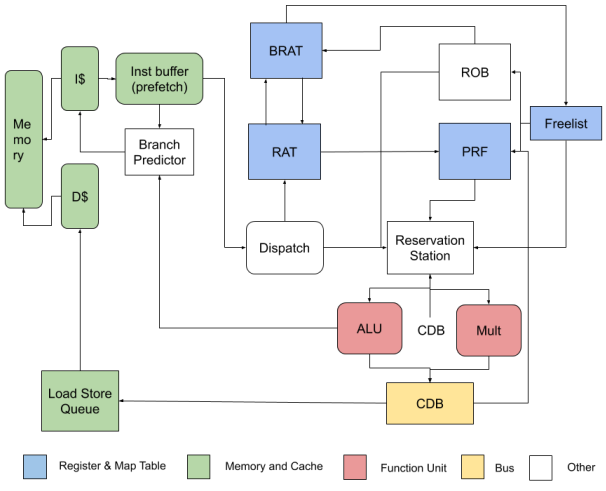

Figure 1. High Level Design Diagram

The processor is based on the R10k algorithm, which is an out-of-order algorithm extending *Tomasulo's algorithm*. The processor has 6 stages, fetch/prefetch (Figure 1), decode, dispatch (figure 1), issue, execution and retire. Similar to a typical processor, the processor has memory/cache interface (green color in Figure 1), register files and map table (blue color

in Figure 1) and function units (red color in Figure 1). What is different in our design is the Branch Register Alias Table (BRAT, Figure 1). BRAT serves as an important structure to help the processor recover quickly from branch misprediction. More on this will be discussed in part B.

B. Early Branch Resolution

The reorder buffer (ROB, Figure 1) and Register Alias Table (RAT, Figure 1), also known as Map Table, are the vital structures to understand the benefit of early branch resolution. In an out-of-order processor, instructions are dispatched and committed in-order but executed out-of-order. ROB is a structure that records the instructions at in-order dispatch time and commits the instructions in-order after the out-of-order execution is finished. Map tables map architecture registers (produced by compiler) to physical registers in the hardware. It provides register renaming to modify the two instructions that have the same destination register to two different hardware components so that they can be executed in parallel.

For a typical out-of-order pipeline processor, branch misprediction signal is broadcasted when a branch retires from the ROB (Figure 1). And the entire ROB is flushed to empty because any instruction in the ROB is on the wrong path. This may cause the processor essentially doing nothing since the executed instructions after the mispredicted will be squashed afterwards. In ideal scenarios, we want branch misprediction signals to be sent as long as the branch instruction is calculated by the function units so that the processor can squash unnecessary instructions and recover to the correct program path as soon as possible. And early branch resolution achieves this property by introducing BRAT checkpoints (Figure 2a).

BRAT checkpoint [2] has the same structure as a map table, and only functions when branch mispredict happens. When a branch instruction gets dispatched at the dispatch stage, one BRAT takes a checkpoint of the map table by copying all of its contents. When later the branch instruction is resolved by the function units and is a misprediction, the processor will revert the state to the corresponding BRAT checkpoint by copying back the contents in the BRAT since that state is the latest time where a misprediction has not happened. An analogy to this is to avoid screwing up drawing a graph, you first make a copy of the current drawing state and continue drawing. If at any time you find it unsatisfactory, you can go back to the prepared copy to redraw the graph.

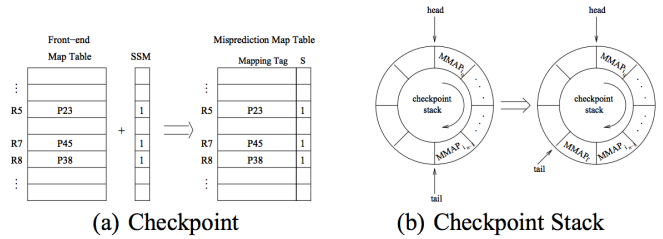

Figure 2. BRAT Checkpoint to Handle Misprediction [2]

It is possible that there are multiple branch instructions currently in the processor. With only one BRAT, one branch instruction can get dispatched only if there are no dispatched branch instructions before it or the previous dispatched branch instruction retires. This will make the most part of the processor unavailable which is known as structural hazard. To address this problem, BRAT checkpoint stack (Figure 2b) is used. BRAT stack contains multiple BRATs, and each of the BRAT is a checkpoint of the map table. When a misprediction signal asserts, the checkpoint stack outputs the corresponding checkpoint to recover for the processor.

C. Non-blocking Cache, Load-Store Queue and Prefetching

A processor communicates with memory to load and store data that cannot be filled in the registers. However, memory latency is usually ten times larger than the processor clock period, which means after querying the memory, the processor will wait about 10 cycles to get back the data. To mitigate this problem, cache is used to store some frequently visited data in the processor. When the memory address requested by the processor core is in the cache, the cache can directly give it back. This scene is called a cache hit.

However, not every data access will result in a cache hit. When the address requested by the processor is not in the cache, the cache is responsible to bring the data from the memory. This scenario is called a cache miss. Traditionally if the cache encounters a cache miss, it cannot accept another query from the processor core before the cache miss is fully resolved. A cache with this property is called blocking cache. Blocking cache sometimes creates the head of line (HOL) problem where a cache miss could block the later cache hit for a long time. Therefore, a non-blocking cache can accept query from the processor even if there is an unresolved cache miss in progress.

Here we use a specific scenario to explain the benefits of having a non-blocking cache over a blocking cache. Consider the following memory accesses: {miss to cache block 1, miss to cache block 2, hit to a cache block}. Assuming the memory accesses are issued from left to right. Figure 3 shows the timing diagram of a blocking cache. The blocking cache uses 18 cycles of time to finish all of the three memory accesses. While a non-blocking cache takes only 12 cycles of time to resolve all of the accesses (Figure 4). The performance improvement is 33% in this case.

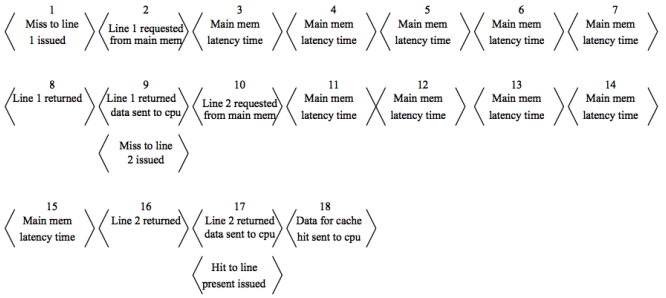

Figure 3. Performance of a Blocking Cache for {miss1, miss2, hit} Sequence

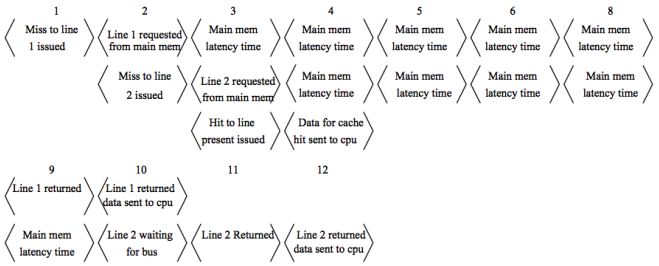

Figure 4. Performance of a Non-blocking Cache for {miss1, miss2, hit} Sequence

Our design includes a non-blocking data cache. In order to implement a non-blocking cache, a new structure named Miss Status Handling Register (MSHR, Figure 5) is needed. A MSHR keeps track of the current cache miss requests and updates correspondingly when memory gives the data back. A queue of MSHR is needed to take down the information of multiple cache misses. The size of the queue is parameterizable so that the performance improvement of different capacity of non-blocking cache can be compared and analyzed. There will be experiment outputs on this in the analysis section.

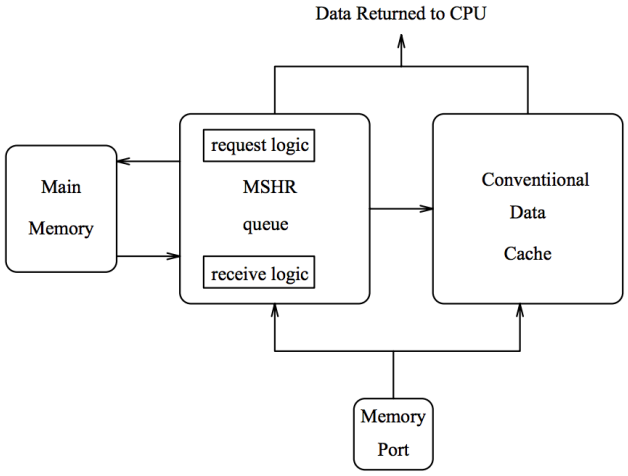

Figure 5. Block Diagram of a Non-blocking Cache

Before the cache receiving any load or store requests, the processor needs to make sure these memory operations are ready to be sent. Moreover, sending loads out-of-order is preferred because there might be dependent instructions on such loads waiting in the pipeline.

The processor uses a double-list structure to manipulate load and store operations separately (Figure 6). When a store instruction gets dispatched, it goes to the end of the store queue. Later load operations memorize the stores that are in front of them by taking down the tail of the store queue. This information is called the age of a load. When a load finds that all of the previous stores are resolved, it will transfer to the ready state and sent to the data cache. If it happens that one or more of the previous stores have the same memory address as the load operation. The load can take the latest store value and get back to the processor immediately. This scheme is called store-to-load forwarding.

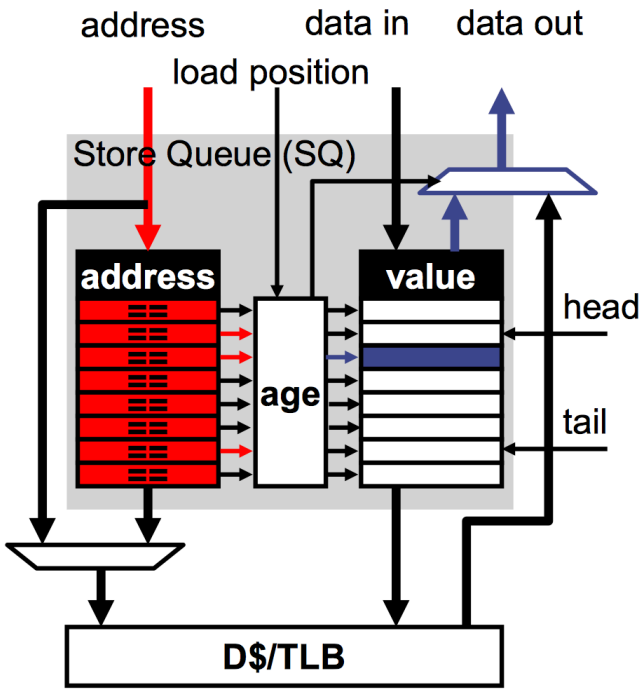

Figure 6. Load-Store Queue Structure

The last advanced memory feature included in the processor is prefetching. The idea of having a prefetching unit is straightforward. If the processor asks for one instruction, it is likely that the processor will ask for several instructions after that as well. Instead of waiting for the fetching unit of the processor to ask, the prefetching unit will query the instruction memory beforehand. An analogy to this is that say you are writing an article and you know that it's likely the article will be multiple pages. You then prepare several pages of paper and start writing on one of them instead of grabbing a new page after you finish one.

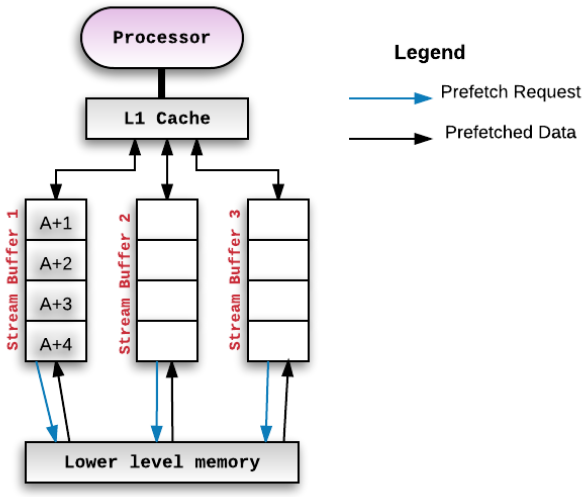

Figure 7. Block Diagram of Prefetching Unit

The prefetching unit contains a stream buffer (Figure 7)

which is essentially a small cache to store prefetched instructions. When the processor requests for an instruction, the requests go through the instruction cache and stream buffer. If it finds the corresponding instruction, it can pass the instruction down to the pipeline stages directly. A comparison analysis of the processor with (w/) prefetching unit and without (w/o) prefetching unit is performed in the analysis section.

D. Superscalar and Instruction Level Parallelism

Non-dependent instructions in a program can be executed in parallel. Doing so will raise the full speed of the processor and result in a faster program completion time. This notion is called instruction level parallelism (ILP). To extend the full speed of the processor, superscalar is implemented to take advantage of instruction level parallelism.

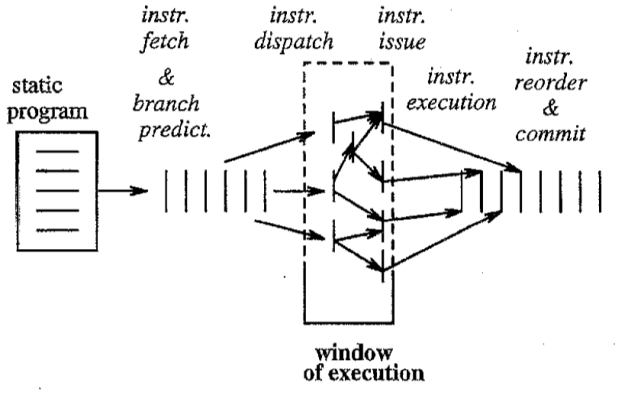

Figure 8. A Conceptual Figure of Superscalar Execution [1]

Our processor design is a two-way superscalar design, which indicates that at most two instructions can enter into each stage. Figure 8 shows a conceptual flow diagram of the superscalar out-of-order design. Multiple instructions (in our case, two) can get dispatched and calculated at the same time, ideally resulting in a 100% speedup.

III. Evaluation & Outcomes

The evaluation policy of the processor conforms with the metrics discussed in the introduction section. There are 30 benchmark test programs for testing the correctness and CPI of the processor. The output from the processor implementation is compared against an instructor solution of an in-order pipeline processor.

A. Correct Implementation

For a correct implementation, the programs should be successfully executed in our out-of-order processor and produce the same program state and result as an in-order pipeline processor. Testing of correctness is done using a bash script which compares the outputs from the out-of-order processor with the instruction solutions. Experiments on local machines indicate all tests are successfully passed.

B. Cycle Per Instruction

Cycle per instruction (CPI) is a typical metric that evaluates the performance of a processor. A smaller CPI indicates that the time for a processor to execute one instruction is less. Table I shows the CPI statistic for all of the 30 test programs. Most of the programs finish with a CPI between 1 to 2.8. For programs that can largely be executed in parallel like rv32_copy_long, the CPI is smaller and for programs that are non-predictable, the CPI is as large as 5. The average CPI of our processor is 2.420.

TABLE I
CPI (CYCLES/INSTRUCTION) FOR TEST PROGRAMS

Program	CPI	Program	CPI
rv32_mult	1.911	*bfs*	2.773
rv32_parallel	1.600	*basic_malloc*	3.458
rv32_copy	2.113	*insertionsort*	2.469
rv32_copy_long	1.094	*dft*	2.143
rv32_fib_long	1.136	*fc_forward*	2.409
rv32_fib	1.773	*alexnet*	2.090
rv32_saxpy	2.839	*backtrack*	2.878
rv32_evens_long	1.453	*matrix_mult_rec*	3.944
rv32_evens	2.794	*omegalul*	2.635
rv32_btest1	5.277	*priority_queue*	3.490
rv32_btest2	5.262	*quicksort*	2.373
rv32_insertion	1.538	*sort_search*	2.025
mult_no_lsq	2.092	*outer_product*	1.478
sampler	2.752	*graph*	3.114
rv32_fib_rec	2.025	*mergesort*	2.686

C. Clock Period

The clock period is verified using a synthesis tool named *Synopsis Tool*. The minimum working clock period for this processor is 10.6 ns, indicating that the processor has a 100 MHz frequency. The success of synthesis is checked through the Slack signal generated by the synthesis report (Figure 9).

```
clock clock (rise edge)                                          11.00      11.00
clock network delay (ideal)                                       0.00      11.00
clock uncertainty                                                -0.10      10.90
cdb_instance/cbd_buffer[5]/dest_result_reg[30]/CLK (dffcs2)       0.00      10.90 r
library setup time                                               -0.33      10.57
data required time                                                          10.57
-------------------------------------------------------------------------------------
data required time                                                          10.57
data arrival time                                                           -8.51
-------------------------------------------------------------------------------------
slack (MET)                                                                  2.07
```

Figure 9. Synthesis Report of Clock Period 10.6 ns

D. Program Completion Time

Although CPI and clock period are two generic metrics that people use to evaluate the performance of a processor, a small CPI or a small clock period does not definitely imply that the processor is good. Program completion time, however, is the truth of a processor's performance on a particular test program. Program completion time is calculated using Iron Law by multiplying the CPI, clock period with instruction numbers. Table II shows the program completion time (in milliseconds) of all 30 test programs. For a program with larger CPI, it does not imply that the program will have a longer execution time because its instruction number might be small (Table II). Since the programs vary largely in instruction numbers, calculating average program completion time alone is useless, so average program completion time is omitted.

TABLE II
PROGRAM COMPLETION TIME FOR TEST PROGRAMS

Program	Time (ms)	Program	Time (ms)
rv32_mult	0.006	*bfs*	0.101
rv32_parallel	0.003	*basic_malloc*	0.034
rv32_copy	0.007	*insertionsort*	1.940
rv32_copy_long	0.003	*dft*	1.304
rv32_fib_long	0.008	*fc_forward*	0.016
rv32_fib	0.003	*alexnet*	4.565
rv32_saxpy	0.006	*backtrack*	0.217
rv32_evens_long	0.005	*matrix_mult_rec*	0.876
rv32_evens	0.003	*omegalul*	0.002
rv32_btest1	0.013	*priority_queue*	0.053
rv32_btest2	0.025	*quicksort*	0.674
rv32_insertion	0.009	*sort_search*	2.426
mult_no_lsq	0.006	*outer_product*	6.086
sampler	0.003	*graph*	0.358
rv32_fib_rec	0.257	*mergesort*	0.265

IV. ANALYSIS

In this section, we analyze the performance improvement over different advanced features and the reasons for finalizing the parameters for the pipeline processor.

A. Non-blocking Cache vs. Blocking Cache

The average CPI of the processor with a blocking data cache is 3.11 while the average CPI of the processor with a non-blocking data cache is only 2.42. The CPI improvement of having a blocking cache is **22%**. For all of the 30 test programs, having a non-blocking cache with MSHR size 4 will result in a better CPI than using a blocking cache (Figure 10).

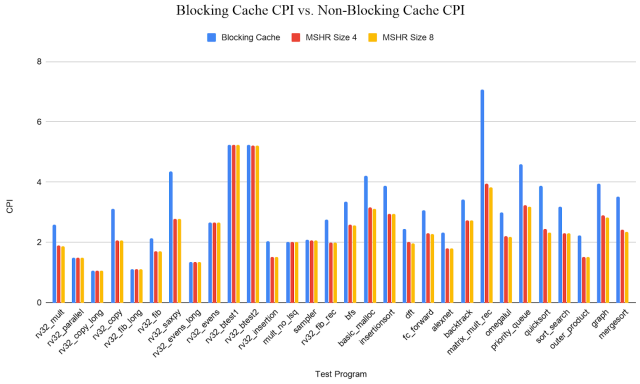

Figure 10. CPI Comparison of Blocking Cache and Non-blocking Cache

However, we didn't find much further performance improvement by increasing the MSHR size. Increasing the MSHR from size 4 to size 8 only gives us about 0.1% CPI improvement. This is normal because for a processor with ROB size 32, the scenario that over 4 cache misses exist at the same time is very rare. Therefore, we finalize the MSHR size to be 4 to shoot for a smaller synthesis clock period.

B. Optimal BRAT Size

Early branch resolution is probably the most important feature of this out-of-order processor. Unfortunately, because early branch resolution's implementation details vary largely from traditional way which resolve branch misprediction at instruction retirement, we are not able to keep two copies of the scheme and compare the performance improvement of full early branch resolution with a processor that does not implement early branch resolution. With simple qualitative analysis, the processor starts fetching and prefetching new PCs as soon as the CDB broadcasts the branch misprediction signal and other structures will squash the instructions on the wrong path, this is inherently better than resolving the branch misprediction at the head of the ROB. Therefore, having early branch resolution is strictly better than no early branch resolution for most of the time (except for when the BRAT stack is full, which is very rare).

Although directly comparing the effect of early branch resolution is not feasible, finding a suitable BRAT size for optimal CPI is worth analyzing.

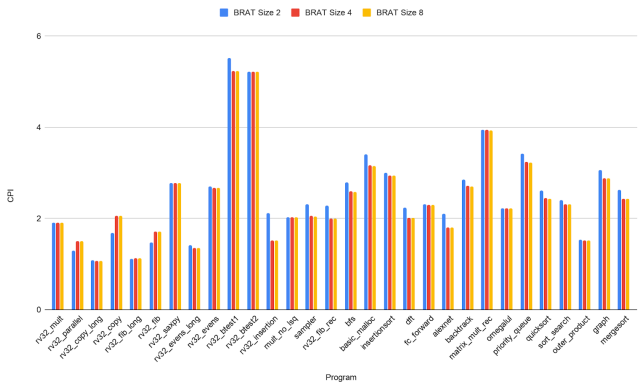

Figure 11. CPI of different size BRATs

For programs in which branch instructions are dense or close to each other (e.g. betst1, rs32_insertion and bfs), BRAT size 2 is too small and will experience performance degradation (Figure 11). After switching to BRAT size 4, the overall CPI improvement is about **12.94%** (Figure 11).

Also notice that for small test programs such as rv32_parallel and rv32_copy, having a smaller BRAT size of 2 will even produce a better CPI than using a large BRAT. This is probably because the branch predictor behaves badly because the predictor is not warmed up. And stall to wait for branch instructions to resolve are actually better than going on and squashing everything later.

The early branch resolution doesn't provide the intended performance improvement as we expected. Part of the reason might be on the branch predictor side. A too good or too bad branch predictor will influence the early branch resolutions performance either by there are no mispredicted branches or stalling is even better when there are too many mispredicted branches. Analysis of branch prediction is in section C.

C. Predictor Accuracy and Predictor Size

Branch prediction plays an important role in total performance as a misprediction can waste a lot of efforts. To better account for the early branch resolution feature, we dig further into the branch predictor and analyze the tradeoff and potential problems in the branch predictor performance and its

design problems.

Branch Target Buffer (BTB) stores the address for branch instructions and allows the processor to identify the fetching program counter corresponding to a branch instruction and its target address. A local history predictor makes use of the branch position in the program to make predictions. Local History Table (LHT) contains the branch information of the last several bits of the program counter and Pattern History Table (PHT) contains the 2-state predictor (00, 01 for not taken and 10, 11 for taken).

The potential values for BTB size and branch history table size are 8, 16 and 32 while for pattern history table size are 8 and 16.

However, adjusting these three parameters doesn't greatly affect the CPI of the processor. The CPI change is in magnitude of 0.01 (Figure 12). By calculating the average CPI among all test cases, the best average CPI is 2.52, achieved by the setup of **8-entry BTB, 16-entry BHT and 8-entry PHT**, which is used as the final parameter for the branch predictor in the pipeline processor.

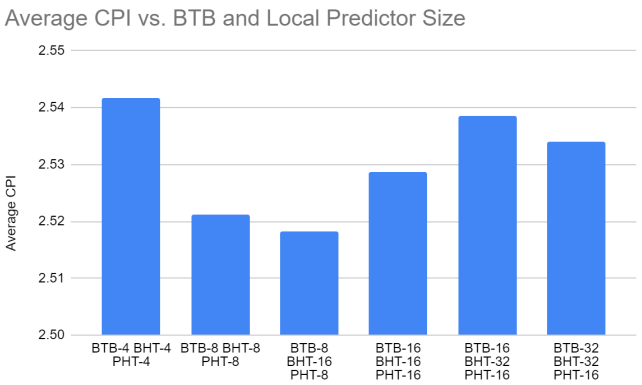

Figure 12. Average CPI of different size BTBs and Local Predictors

Meanwhile, the actual accuracy of prediction our design obtained is also an interesting factor. The total prediction accuracy for each setup is derived along with the CPI information (Figure 13).

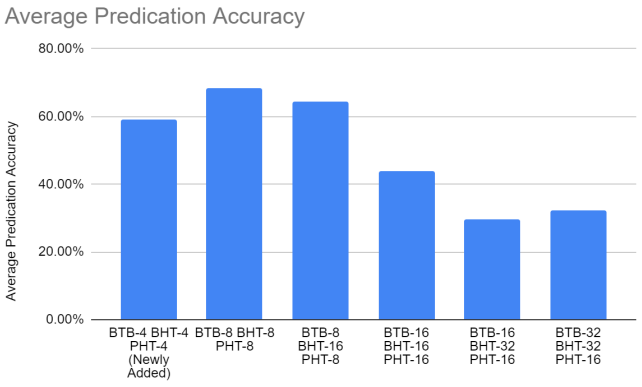

Figure 13. Average accuracy of different size BTBs and Local Predictors

The prediction rate is surprisingly decreased per increasing size. The best accuracy is **68.36**% from 8-entry BTB, 8-entry BHT and 8-entry PHT. However, this result points out an unexpected trend that CPI is not affected very much while the accuracy is dropping as the size grows. So, a new set of data is added to fix all variables to be 4-entry. Since it has the lowest CPI and a lower accuracy, the optimal size is set as: 8-entry BTB, 16-entry BHT and 8-entry PHT.

However, we notice that some test cases have very low accuracy (Figure 14). For those with small instruction numbers, it might be due to cold start, where most branches are executed once or twice. On the other hand, there are test cases with large amounts of instructions. For those with low accuracy through all setups, like rc32_fib_rec, this is mainly because we are executing a recursive algorithm, which is full of function calls and each return address is different. Therefore, the address of an unconditional jump representing a return may never be correctly predicted. Meanwhile, there are test cases with low accuracy for large predictors but high accuracy for smaller ones, like insertionsort and matrix_mult_rec. Local predictors with small BHT and BTB somewhat degenerate to global predictors. So, for test cases with chains of branches, they might work much better (Figure 14).

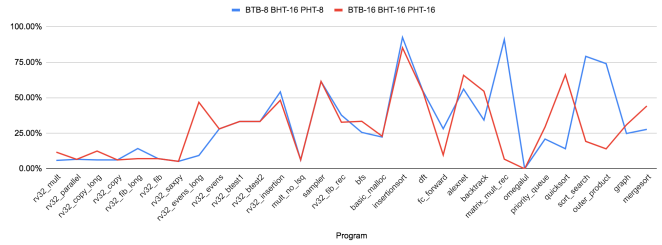

Figure 14. Accuracy of Different Size BTBs and Local Predictors

The branch predictor is not very ideal in this processor. Due to lack of time, a more sophisticated branch predictor is not implemented. A more complex predictor such as a tournament branch predictor or a return call stack can outperform a single local branch predictor for sure. With a better branch predictor, the performance improvement of having early branch resolution in the processor will be better.

D. Performance Improvement of Different Prefetching Size

Instruction prefetching counts as an important factor to improve processor's performance, especially at the condition where memory latency is about 10 cycles. We take down the CPI information before adding the prefetching module and analyze the performance improvement with different prefetching sizes of 4, 8 and 16.

Figure 15 indicates that adding a prefetching unit to the pipeline can largely improve the performance of our processor. To be more specific, going from no prefetching to prefetching 4 instructions results in an average CPI reduction of **33.7%**! Enlarge the stream buffer in the prefetcher to size 8 can further improve the performance of 8.01%.

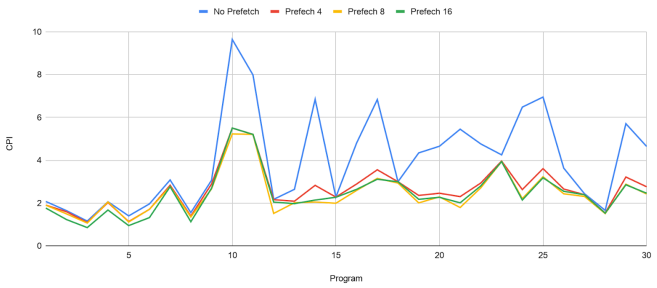

Figure 15. No Prefetching CPI vs. Different Prefetching Size CPI

Using a prefetching of size 16 actually hurt performance a little bit, as shown in Figure 3. While prefetching stream buffer size 16 gives us the best CPI 0.86 on test rv32_copy_long and CPI 0.95 on rv32_fib_long, notice that it actually hurts the processor's performance on some large C testcases such as insertion sort and quicksort. This is probably because the prefetching unit is doing extra work to fetch unnecessary instructions on mispredicted branches. Above all, prefetching size of 8 is chosen ultimately since it improves the overall CPI.

E. Cache Hit Rate & Load-Store Forwarding in MSHR

Conventional load to store forwarding is implemented in the processor. Except for that, a tiny trick feature is added to the non-blocking cache. The data that's in the MSHR but not yet committed to memory can be forward to later load operations as well. Since the point of load-to-store forwarding is to perform load operations quickly, we combine the cache hit and forwarding from MSHR as our total cache hit rate because these load operations can finish in just one extra cycle even if they are not forwarded from the store queue directly. Surprisingly this mechanism works pretty well and gives back a good cache hit rate of **93.6%** with only a 32 * 8 Byte direct mapped cache (Figure 16).

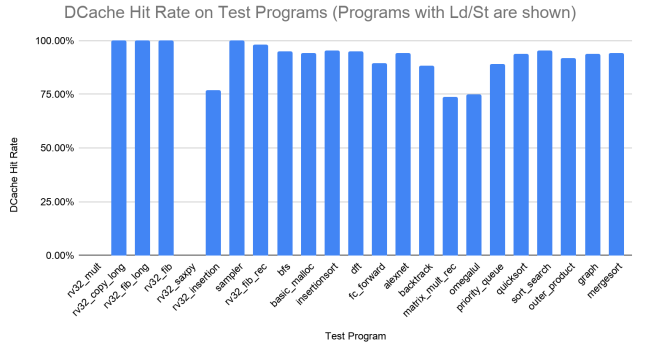

Figure 16. Data Cache Hit Rate on Different Test Programs

F. Further Refinement on Reservation Station Size

Lastly, reservation station size (RS) is analyzed to address the problem that RS might be full to create structural hazard, causing the processor to stall for too long. A small RS will more easily be filled up, so in theory a larger RS will result in fewer stalls and thus better performance. RS size of 2, 4, 8 and 16 is tested and the corresponding CPI is compared (Figure 17).

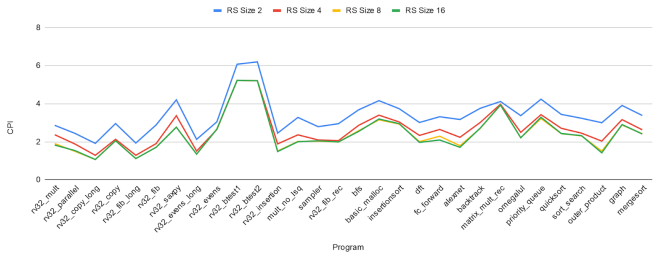

Figure 17. CPI with different RS sizes

For most test programs the CPI decreases with the size of RS. Note that there is a strange point at test program matrix_mult_rec. The program is a large matrix multiplication program written in recursion and requires a lot of store operations and incurs a lot of store misses in the data cache. Increasing RS sizes does not help in this case because nearly all of the stalling comes from the store queue.

G. Overall Performance Against In-Order Pipeline

To evaluate the improvement and effectiveness of the designed processor. A detailed comparison with the five-stage pipelined processor is performed thoroughly. All the 30 testcases are run on the provided 5 stage pipelined processor. The final metric used here is program completion time, which is the most suitable metric here to evaluate the performance improvement. Because the original five-stage pipeline processor omits the actual memory latency, 30ns memory latency is used for memory access in only one cycle for the in-order pipeline processor.

TABLE III
PROGRAM COMPLETION TIME COMPARISON OF OUT-OF-ORDER PIPELINE PROCESSOR WITH FIVE-STAGE IN-ORDER PIPELINE PROCESSOR

Program	Out-of-Order Completion Time (ms)	In-Order Completion Time (ms)
rv32_mult	0.006	0.013
rv32_parallel	0.003	0.008
rv32_copy	0.007	0.007
rv32_copy_long	0.003	0.021
rv32_fib_long	0.008	0.022
rv32_fib	0.003	0.007
rv32_saxpy	0.006	0.010
rv32_evens_long	0.005	0.013
rv32_evens	0.003	0.005
rv32_btest1	0.013	0.013
rv32_btest2	0.025	0.025
rv32_insertion	0.009	0.030
mult_no_lsq	0.006	0.010
sampler	0.003	0.005
rv32_fib_rec	0.257	0.742
bfs	0.101	0.182
basic_malloc	0.034	0.054
insertionsort	1.940	4.766
dft	1.304	2.743
fc_forward	0.016	0.029
alexnet	4.565	10.667
backtrack	0.217	0.392
matrix_mult_rec	0.877	1.256
omegalul	0.002	0.003
priority_queue	0.053	0.084
quicksort	0.674	1.544
outer_product	2.426	7.121
graph	0.358	0.625
mergesort	0.624	0.498

Figure 18 illustrates the program completion time improvement of large test programs. Small testcases are omitted in the figure because of scale problems since their completion time is too small to show on the graph. Detailed completion time is shown in Table III. The average program completion time for our out-of-order processor is 0.643 *ms* and the average program completion time for a five-stage pipeline processor is 1.665 *ms*. The performance is inversely proportional to the completion time. Thus, the performance of the two-way superscalar out-of-order processor is 2.590 times the performance of an in-order pipeline processor, which indicates a **159%** performance improvement.

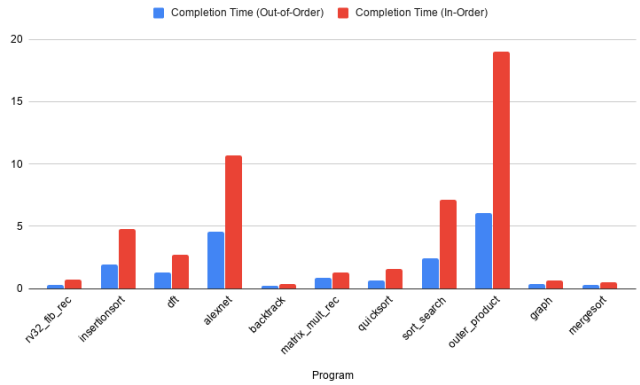

Figure 18. Program Completion Time Improvement

V. Conclusion

We have successfully implemented a two-way superscalar out-of-order RISC-V processor using the R10k algorithm. The processor synthesis in a clock period of 10.6 ns. This report provides a detailed analysis of the processor's advanced features and their effectiveness. The processor is fully integrated, tested in SystemVerilog HDL. We demonstrate our processor is a fully working prototype of an out-of-order processor design with early branch resolution and a superscalar width of two.

References

[1] JAMES E. SMITH, "The Microarchitecture of Superscalar Processors," *PROCEEDINGS OF THE IEEE*, VOL. 83, NO. 12, DECEMBER 1995.
[2] Peng Zhou et al, "Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors," ICS '05, June 20-22, Boston, MA, USA.
[3] Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEEMicro, 1996.