4-way Superscalar R10K Out-of-Order Processor

EECS 470: Computer Architecture

Fall 2019

Group 8

OoO

Xiuneng Lu
University of Michigan
xiuneng@umich.edu

Siyu Niu
University of Michigan
siyuniu@umich.edu

Yunjie Pan
University of Michigan
panyj@umich.edu

Runyu Zheng
University of Michigan
runyuz@umich.edu

1 Introduction

Out-of-order pipelining is widely used in high-performance processors to avoid some kinds of stalls that decreasing CPI. This report describes a 4-way Superscalar Out-of-Order Processor in SystemVerilog implemented by Group 8 OoO for EECS 470 final project. Our goal is to design a core with several advanced features and high performance while maintaining correctness.

2 Features

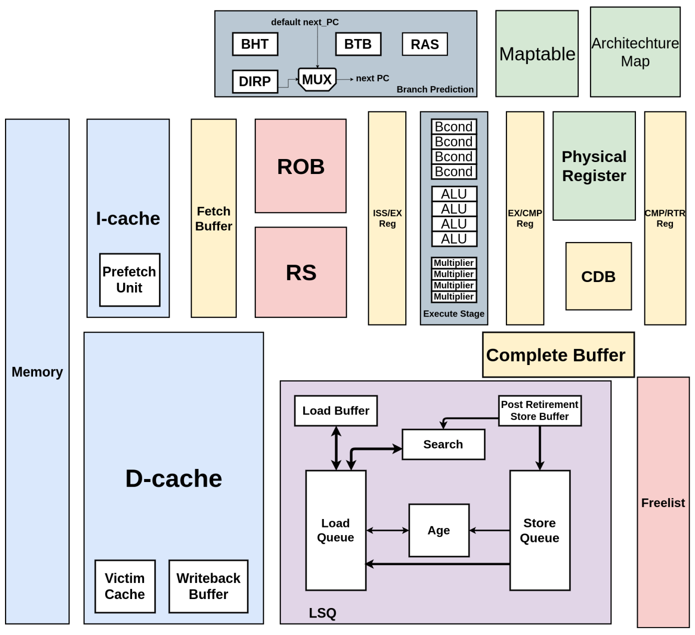
Feature	Included	Comments
RISC V R10k OoO Processor	Yes	
Graphical debugging Tool	Yes	Visualize pipeline information with ncurses.
Automated regression testing infrastructure	Yes	Automatically test the correctness of our design with shell.
Superscalar	Yes	Fix superwidth to be four.
Store-to-load forwarding in LSQ	Yes	Forward SW to LW, LH and LB from store queue or post store buffer.
Loads issue out-of-order past pending stores (non-speculative)	Yes	Once all previous load have their address ready and no data can be forwarded, load can be issued.
Post-retirement store buffer	Yes	FIFO buffer size of 16. When store retired, data and address are stored here, waiting to be stored in cache.
Multiple outstanding load misses	Yes	Have non-blocking cache
Next-line or stride prefetching for instructions and/or data	Yes	When fetch stage requires a specific instruction, search the next 16 instructions, if they are not in icache, issue a fetch request.
Write-back data cache	Yes	Dcache is a write-back, write-allocate cache
Associativity > 1	Yes	Both Icache and Dcache are N-way associative caches
Victim cache	Yes	A victim cache is fully associative cache placed in the refill path of Dcache.
Return address stack	Yes	When the destination or the source register of a jump instruction is a link register, push or pop to RAS.
Loads speculatively issue past pending stores	No	Implemented and tried on trivial cases. Still some bugs about rolling back and recovering.
Load dependence predictor	No	Implemented, basically two way cache. Store the address of any load-store forwarding pair to predict load dependency , the structure is too inefficiency and impact on our critical path.

Table 1: Features

3 Design

3.1 Top Level Diagram

The top-level diagram is shown in Figure 1.



Components	Comments
Superwidth	4 for all stages
ROB entry number	32
RS entry number	16
Load queue entry number	16
Store queue entry number	16
Post Retirement number	32
Load buffer number	4
Victim cache size	8 Bytes
Writeback buffer number	8
Branch History table size	64
Branch target size	256
Return address stack size	16
Fetch buffer size	8
Complete buffer size	52
Dcache size	64 Bytes
I cache size	64 Bytes
Synthesis clock period	17.0ns
Average CPI	1.13

Figure 1: Top level design.

3.2 Design Choices

3.2.1 Post store retirement buffer

When a store instruction is retired (worst case 4 per cycle), instead of sticking them in the store queue, waiting Dcache to store them in order, we just put the data and address in the FIFO buffer, so the retire stage won't be stalled. Because both data and address are ready, if a later load use it, data can be forwarded through our "SEARCH" logic. We allow the head of buffer to be stored into Dcache to ensure in order store. In this way, we get away with store miss penalty.

3.2.2 Load queue issue & complete policy

Since load instruction can be issued (send address to Dcache or forwarding data from other place) and completed (send data to complete stage) out of order, we could make load queue a subset of RS. We make load queue a subset of ROB because we want to know the relative order of load instruction. We believe that earlier load should have larger priority to be issued and to complete buffer because later instructions are likely to be dependent on previous load instructions, especially after load miss. So our issue and complete policy for load queue is searching from head to tail, and find the first super-width number of qualified load instructions.

3.2.3 Store load forwarding logic

In our implementation, we allow store word forward data to any type of load instruction(LW, LH & LB). Though store half and store byte can forward data to load instruction in some special cases, but that's rare and would cause cost more latency.

3.2.4 Complete buffer size & input priority

Since our super width is 4, with 4 ALU, 4 Multiplier and 4 load buffers, in the worst case, we will have 12 instruction completed. However, we have only 4 CDB. So the complete instruction (data and tags) must be put into a buffer to solve this problem. Unlike post store retirement buffer, which a larger size could degrade the performance a lot, (because later instructions need to judge if data needs to be forwarded. Larger size would made the judgement complex and slow.) we can make complete buffer large enough so that our pipeline won't be stall on execute stage. After our estimation, (not precise but must be greater than the upper bound) we make the complete buffer size equal to Superwidth $\times$ (load miss cycles $+$ mult stage cycles $+ 3) = 52$.

We believe instructions with longer latencies should have larger priority to complete, because later instruction is more likely to be dependent on them. For our case, we make priority Multiply $>$ Load $>$ ALU.

3.2.5 Dcache

Our Dcache is a N-way set associative, write-back, write-allocate cache. We apply least-recently-used(LRU) replacement policy to decide which way of data to be evicted if N ≥ 2. Because of LSQ forwarding, Dcache is able to deal with load and store instructions separately. For write-allocate cache, both load and store requires to read data from main memory. Only when the required data is loaded on Dcache, can load queue get data or store queue write data. For write-back cache, if the evicted data from cache is dirty, which means previous store instructions has written new value to that address, the evicted data need to be write back to main memory.

Our Dcache is a non-blocking cache, which means it can deal with multiple miss load instructions. In particular, our Dcache can deal with SUPERWIDTH load instructions from load queue at the same time. Once a load instruction is hit in Dcache, it can be replaced with a new load instruction in Dcache in the next cycle. Since Dcache can deal with SUPERWIDTH load instructions, if more than one load instruction is miss in Dcache, Dcache can send load request to memory in pipeline manner. In such case, Dcache can hide memory latency.

To hide memory write latency, we use a writeback buffer to store all the dirty evicted data, waiting for being granted for memory bus. A victim cache is also implemented in our Dcache design. The victim cache is fully associative, 8 byte (a cacheline) cache placed in the refill path of Dcache. Load instructions can refer to victim cache for the latest evicted data to save the time to load data from memory(100ns).

There may be multiple load/store request from cache to memory, the memory grant priority in our design is as follows: Icache fetch miss $>$ Dache load miss $>$ Dcache store miss $>$ Dcache writeback buffer store to memory $>$ Icache prefetch.

4 Performance Analysis

4.1 N-way Superscalar

We implement 1-way, 2-way, and 4-way superscalar of our out-of-order CPU. The CPI for testcases is shown in Fig 5. The average CPI of 1-way, 2-way, 4-way CPI are 2.07, 1.73, 1.63, respectively. Superscalar is able to drastically decrease the CPI, since it can exploit instruction-level parallelism. But for test cases with lots of strong dependence (like matrix_mult_rec), the 2-way or 4-way superscalar has less advantage over 1 way but still gain some speedup because of faster fetch, dispatch and retire. Besides, the RAS has contributes a lot to the performance of superscalar CPU. Since RAS is able to predict jump and return address, we can achieve good CPI in many recursive programs like mergesort. And thanks to our branch prediction, programs with long loops are able to achieve low CPI since they can save the time for the rollback from branch mispredict.

One of the most important bottleneck in our design is the limitation of memory bus. In other words, Icache and Dcache share the same memory bus, which is only 64 bits. As stated in 3.2.5, we have a sophisticated grant priority over different memory and cache units. This will limit our CPI when there are dense load and store instructions in the program. Another disadvantage of superwidth CPU is that as the superwidth increase, the clock cycle need to increase to avoid negative slack. Because we need to update superwidth variables in the same cycle, the latency of it will contribute to the critical path. Therefore, we need to tradeoff for CPI and clock cycle considering different superwidth.

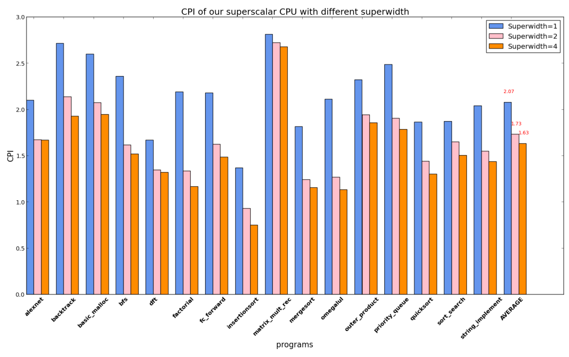


Figure 2: CPI of our CPU with different Superscalar

4.2 Branch history table

As shown in Figure 3, prediction of always taken has already gives us decent performance, but prediction aided by BHT also gives us slight performance boost.

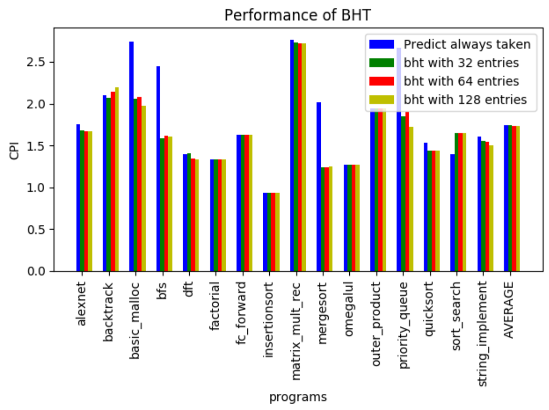


Figure 3: CPI influenced by branch history table.

4.3 Next-line prefetch

As shown in Figure 4, instruction prefetch gives us huge performance boost in lots of test cases. However, prefetching next 8 instructions or next 16 instructions does not make much difference.

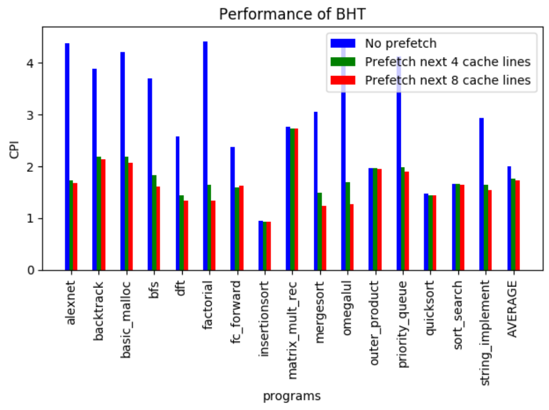


Figure 4: CPI influenced by instruction prefetch.

4.4 Store Load forwarding

Since many of the program involves lots of array computation, data is frequently load and stored from the same place. Having a load store forwarding would help hide the cache miss penalty. To verify this, we turn off load store forwarding unit, using o0 to run all the .c test, here's the result:

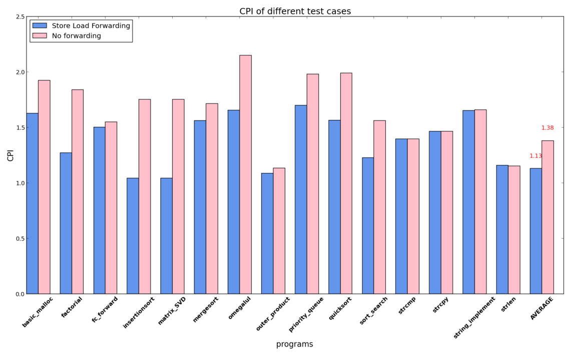


Figure 5: CPI of our CPU with/without store load forwarding

As we can see, with store load forwarding, our CPI drops from 1.38 to 1.13, which is a great improvement.

4.5 Cache Associativity

Since fetched instructions are usually continuous, but store or load addresses are usually not, in this section we focus on discussing how the associativity of Dcache will infect out CPI. Figure 6 shows the result of CPI with different associativity in Dcache. The higher the number of way, the higher of associativity of cache. The result shows that when the way of Dcache is set as 2 or 4, our CPU can achieve optimial performance.

If the associativity is 1, which means that Dcache is a direct-mapped cache, there may be multiple load or store address mapped to the same index, and the data will be evicted from Dcache, and load back to Dcache back and forth, which will increase the time waiting for memory read/write (100ns). On the other hand, if the associativity is relatively high, it takes time to iterate through all the lines, and it has to iterate over entire cache set to locate a block.

To make a trade-off between direct mapped and fully associative cache, we fix our Dcache as a 2-way set-associative cache.

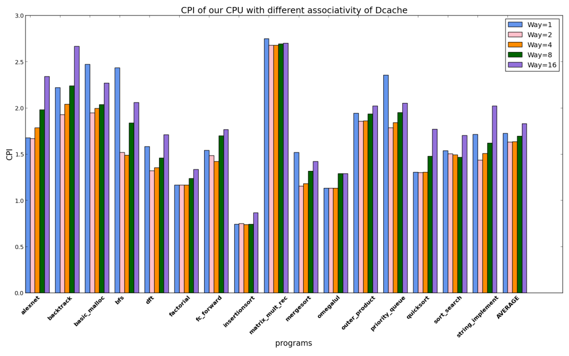


Figure 6: CPI of our CPU with different associativity in Dcache

4.6 Critical Path

Before we modifying our Dcache, the critical path is the communication between load queue, Dcache and memory. Because we use asynchronous signal to tell whether the load instruction is hit in Dcache or not, Dcache need to wait for memory send back new value before knowing the load instruction is hit or not. The previous minimum clock cycle for synthesis is around 33 cycles. After we modifying the signal to be synchronized, the clock cycle can be reduced by half, to 17 cycles.

The critical path in our final design is the communication between Dcache and memory. Because our design is a 4-way superscalar CPU, updating superwidth values in a clock cycle contribute to the latency in critical path. One possible solution to reduce critical path latency is to reduce superwidth in our design. But it will increase the CPI, as shown in Figure 5.

5 Contribution

WHO	WHAT
Xiuneng Lu	Execute Stage, Complete Stage, Retire stage, Map Table, Load Store Queue, Pipeline Integration & Debug
Siyu Niu	Architecture Map, Issue Stage, ROB, Physical Register, Pipeline Integration & Debug, synthesizing & Debug
Yunjie Pan	Fetch Stage, ICache, Branch Prediction, Freelist, DCache, Pipeline Integration & Debug
Runyu Zheng	Dispatch Stage, Issue Stage, ROB, RS, Branch Prediction, ICache, Visualized Debugger, Pipeline Integration & Debug

Table 2: Contribution Taable