

An out-of-order, super-scalar implementation of the RISC-V ISA in the style of the P6 micro-architecture

Wenjie Geng^a, Peijing Li^a, Haowen Tan^a, Yunjie Zhang^a and Yunqi Zhang^a

^a*Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, United States*

ARTICLE INFO

Keywords:

RISC-V ISA
out-of-order processor
P6 microarchitecture
superscalar
prefetching
scalable memory hierarchy

ABSTRACT

This report describes an implementation of the P6 microarchitecture using the RISC-V instruction set. Our design includes several features that contribute to high performance, including out-of-order execution, superscalar instruction fetch and dispatch, a non-blocking instruction cache, instruction prefetching, and a branch target buffer, among others. We have ensured the correctness of our processor through rigorous testing and validation, even if it comes at a cost of fewer *claimed* advanced features, and we believe that our implementation provides a powerful and flexible CPU design suitable for a wide range of applications. As part of our report, we include detailed documentation of our functionalities, testing results of how our advanced features impact the performance of the processor, and relevant discussion and reflection on the impacts of our project.

1. Introduction


Intel's P6 microarchitecture, first debuting in 1995, can be considered as one of the company's most influential products. Among the advanced features it introduced are speculative, out-of-order execution and microcode translation (Gwennap, 1995). The former enables the processor to execute subsequent instruction even if a prior instruction stalls or otherwise requires long latencies to execute. The latter essentially translates x86 complex instructions into simpler, RISC-like micro-instructions; this also enables the actual execution pipeline of P6 processors to be built similar to RISC processors. This means that the P6 can take advantage of the simplicity, higher clock frequencies, and other benefits previously only enjoyed by dedicated RISC systems.

This report presents our implementation of the P6 microarchitecture using the RISC-V instruction set. We pay particular attention to the P6 features of out-of-order execution and superscalar instruction fetch, and developed a two-way fetch, five- or six-way issue, out-of-order pipeline that aspires to combine some of the inherent advantages from P6 with the simplicity of the RISC-V ISA into a powerful, flexible package.

The rest of the report is organized as follows. Section 2 provides detailed documentation on our microarchitecture design, both on a system level and for each of the substantial modules we implemented. Section 3 presents the testing methodology for our processor as well as performance figures given different configurations of our processor. Section 4 presents a brief discussion on the implication of our processor and a brief reflection upon the significance of our project. Section 5 summarizes and concludes this report.

2. Design and Implementation

This section provides both an overview of our P6-style microarchitecture design and documentation of each of our modules.

 wenjieg@umich.edu (W. Geng); peijli@umich.edu (P. Li); thwthw@umich.edu (H. Tan); yunjiez@umich.edu (Y. Zhang); yunqizh@umich.edu (Y. Zhang)
ORCID(s): 0000-0001-8517-6109 (P. Li)

2.1. Microarchitecture Overview

We implemented our out-of-order processor in the style of the Intel P6 microarchitecture primarily for the intuitiveness and ease of maintaining precise states, as well as the more straightforward register-renaming scheme of tying reorder buffer entries to renamed destination registers. This subsection presents both the configuration of a “**baseline**” processor that is in the `final_submit` branch of our BitBucket register and additional functionalities not included in that final submission.

2.1.1. List of features

Apart from the standard modules and features present in a P6-style processor (i.e., the reservation station, reorder buffer, map table, standard functional units, common data bus, architectural register file, instruction fetch and decode modules, etc.), we also implemented a number of advanced features and modules, most but not all of which are present in the baseline processor. Many but not all of our modules also have configurable parameters to enable us to find their optimal values through testing and evaluation. The following Table 1 lists the advanced features that we implemented.

Table 1

List of advanced features implemented for the project

Advanced features	Included in baseline processor
2-way superscalar fetch, dispatch, and retire	Yes
Instruction cache with sequential prefetching	Yes/Can be parameterized
Set associative data cache	Yes/Can be parameterized
Store to load forwarding	Yes
32-stage pipelined divider	Tested and completed but not included
Branch predictor	Yes
Branch target buffer	Yes/Can be parameterized
Decode branch instructions during fetching	Yes

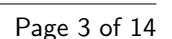
2.1.2. Top-level System Diagram

The following Figure 1 contains an overview of the structure of our processor, including both basic and advanced modules, as well as some of the critical connections between individual modules. Note that modules related to instruction fetch and decoding are depicted in red, modules related to instruction dispatch, issue, and retirement are depicted in blue, modules related to instruction execution are depicted in green, the Common Data Bus is depicted in orange, and modules related to memory accesses are depicted in yellow.

Some of the most important organization traits of our processor’s organization are listed as follows.

1. The memory interfaces with both an instruction cache and a data cache.
2. Both a branch target buffer and a branch predictor are involved in the instruction fetch and program counter update process.
3. Each functional unit contains a separate bank in the reservation station, where only instructions associated with the functional unit are stored.
4. A load-store queue is responsible for store-to-load forwarding and other actions that interface with the memory.
5. Pipeline registers are inserted between the instruction fetch module and the decoder module (IF/ID), the reservation station banks and their corresponding functional units (RS/FU), the functional units and the common data bus (FU/CDB), and the store queue and the data cache (SQ/D\$).

The following subsections will provide detailed documentation of each of the modules involved.



Two branch target buffers (BTB) are instantiated for conditional branches and unconditional branches, respectively. A two-bit saturating counter is also attached to each entry of the conditional BTB entry. This

module is connected to the instruction fetch stage and accepts raw instruction bits before they pass through the IF/ID pipeline registers into the decoder, hence the “pre-decoding” of branch instructions.

The branch target buffers determine whether inputs from the fetch stage belong to conditional or unconditional branches (hence the “pre-decoding” of branch instructions), and look up their program counter values in the BTB tables. If the PC value is found in the BTB for unconditional branch, or if the PC value is both in the BTB for conditional branch and the entry predicts either “taken” or “weakly-taken,” the instruction fetch module would reference the target address in the BTB entry and start fetching instructions from there.

Upon the retirement of a branch instruction, the outcome of conditional branches and/or the branch target address are compared with its corresponding BTB entry. If there is either a misprediction of conditional branches or a misprediction of branch target addresses, the pipeline would be flushed with the BTB entry updated with the branch outcome and/or the branch target address. If the retired branch does not yet correspond to any BTB entry, a new entry would be allocated in the BTB table, using a random eviction policy if the BTB table is already full.

2.3. Basic P6 Modules Involved in Instruction Dispatch, Issue, and Retire

2.3.1. Reservation Station

The key trait of our reservation station is that it is split into five (or six) discrete banks. Each bank is of parameterized size (four in the baseline processor) and corresponds to one functional unit (either integer ALU, multiplier, branch resolution unit, load and store address resolvers, or the divider as an optional feature). The reservation station can dispatch two instructions at once and store them into the correct reservation station banks. Some other noticeable implementation details of the reservation station are discussed below.

One of the improvements is the dispatch stage, in which we need to dispatch the two packets from decoder into the corresponding bank. The naive solution to it is to check the functional unit types for both instruction with complex nested structure, which may lead to a huge `always_comb` block and a long clock period. Instead, we use the divide-and-conquer algorithm. Each bank will check whether the fetched instructions can be dispatched. We break the task originally done in one combinational block into separated ones to allocate new entries. It results in shorter critical path and parameterized code.

Since the reservation station structural hazard signal is used by many modules in our processor, we want to get the reservation station structural hazard signal in each cycle as fast as possible. So we test each bank and check whether there are two empty entries. If all the six banks has two empty entries, the structural hazard signal is negative. This method to check structural hazard also helps with the dispatch stage improvement above since we know that there must be empty entries to put the new instructions when there is no structural hazard.

Separate consideration was also given to the bank storing load instructions. We need to issue the youngest load in the load bank to ensure the correctness. The naive solution is to traverse the whole load bank and find the entry with the smallest age index. But traversing and comparing age bits will lead to a longer delay. We solve this problem by introducing a head and tail index, which is similar to those in the reorder buffer (to be discussed below). Each time we dispatch or issue a load instruction, the reservation station will change the head and tail index, so that we can ensure that the youngest load instruction is always at the head.

2.3.2. Reorder Buffer

The reorder buffer (ROB) is used to keep track of the instructions in program order. It is also of parameterized size, with a default value set to 16 in the baseline processor, and it is implemented as a circular queue, with head and tail pointers for control. It also supports dual port searching, allocation, and output, enabling the handling of two instructions dispatched at once that may be interdependent upon each other.

In Dispatch stage, ROB allocates two entries at a time, getting packets from decoder and interacting with map table and RS. In complete stage, it marks the status of the instruction as complete according to the

broadcast packets from the common data bus and forwards the data to RS if it requires in the same cycle. ROB retires the head instruction if it's completed and write the value to register file, twice at most. ROB also supports detection for whether to halt, to read or write memory at the same time. By keeping a record of number of instruction retired in each cycle, it counts the processed instruction to the pipeline as well as the testbench.

2.3.3. Map Table

Map table is implemented as a content-addressable table containing 32 entries corresponding to each architectural register that supports dual-port searching and outputs. Each entry maps an architectural register to a tag in the reorder buffer, with an addition ready bit to indicate whether an actual value is ready in the corresponding ROB entry.

The input of map table includes ROB dispatch packet, CDB packet, ROB retire packet and decoder packet, and the output sends tags and ready bits to reservation station. The map table uses the architectural register numbers as indexes to search the current state information of one register. In dispatch stage, it needs to search and determines the states of the instruction's opa register and opb register, and sends the corresponding information to reservation station to update the entries. At the same time, it receives ROB dispatch packet to set the tag of the destination register as the corresponding ROB index. In complete stage, Map table searches if it has an entry contains the same tag from CDB and turns the ready bit on. In retire stage, Map table searches if it has an entry contains the same tag from ROB and turns the corresponding tag bit be zero. In addition, when there is branch mis-prediction, map table clears all its entries.

For all those look up functionalities, we use combination logic to update Map table next state. One thing needs to notice is that since retire stage input and CDB input and dispatch stage are handled simultaneously, there are possibility that the updates are pointing to the same entry and they may replace the next state information by order. However, at the same time, we need to pass through the dispatch instruction's opa and opb register information and give outputs to the reservation table. In order to give outputs with desired information, we set up a special middle state to specifically deal with the output to reservation table.

2.4. Functional Units

We instantiated five (5) Functional Units (FU) in total in our baseline processor, listed as follows.

1. One 32-bit arithmetic logic unit, in charge of executing addition, subtraction, bitwise and, bitwise or, arithmetic shift, and logical shift operations.
2. One 32-bit, pipelined multiplier with 8 pipeline stages.
3. One branch resolution unit, containing comparators to resolve conditions branches and an adder to compute branch target addresses.
4. Two 32-bit adders to compute the addresses of load and store instructions, respectively.

We also developed an additional 32-bit, pipelined divider not included in our baseline processor, to be described in detail below.

Each of our functional units is connected to a corresponding reservation station bank. The outputs of each functional unit contain the result of the operation, the destination tag in the reorder buffer for the instruction executed, and a done bit. All functional units except the multiplier and divider are purely combinational. The sequential multiplier and divider modules each have additional input bits of clock and reset signals.

The latency of functional units varies due to their operation. The ALU and adders to resolve branches and memory operations an finish within 3 ns, while the multiplier needs 7 ns and the divider needs 6 ns, respectively, for their clock cycles.

2.4.1. Pipelined Divider

Among the more experimental of our advanced features is a 32-bit, 32-stage, pipelined divider. The divider utilizes an algorithm similar to the SRT algorithm in resolving one bit of the quotient and the remainder at each pipeline stage and progressively performs left shifts on both the dividend and the partial remainder. The divider would resolve all divisions with the assumption that the dividend and divisor are unsigned; the correct sign would be assigned to the output quotient and remainder wires by appropriate multiplexers.

This divider is not present in our baseline processor due to unknown bugs that may or may not be associated with the module itself during initial synthesis testing. These bugs appear to have been resolved by the time of writing this report, and after the deadline for code submission.

2.5. Common Data Bus

Since we have six functional unit in total, but we could only have two CDB packet out in one cycle according to the project requirement, so we give write a CDB selection pipeline register to realize this function. In each cycle, the pipeline register will receive six packet from the six functional unit. Then the pipeline register will calculate the number of packets needed to be broadcast. If there are less or equal to two entries, just broadcast these entries and record the six inputs into the next stage. Else if more than two entries needed to be broadcast, we turn on a CDB structural hazard signal and broadcast two of the entries, and we will stall the functional units and prevent them from providing the CDB with new instruction results.

2.6. Modules Involved in Memory Instructions

2.6.1. Store Queue

We implemented a store queue and relevant load scheduling functionalities to realize store-to-load forwarding and conservative load scheduling. This means that we are to only issue load instructions if all older stores have calculated their address. If there the address of the load instruction matches entries in the store queue, we shall forward the value from the store queue. If there is not a match in the store queue, the module would issue the load to the data cache and potentially the system memory.

The store queue mainly deal with three parts of mission, giving an age index for every load or store instruction, recording value and address and sending retired store instructions to data cache, checking the status for the load instruction and get the load value from correct space.

At the dispatch stage, each load instruction is provided with an “age index” to facilitate conservative load scheduling. For store instruction, the age index will be directly written into the store queue at the same time we allocate it. For load instruction, the age index will be bounded with this load from the dispatch stage and finally enter the load register.

Every entry in the store queue consists of ROB index, address, value, age index, retired bit and entry valid bit. The store queue first allocate on entry for store instruction in dispatch stage, get the value to store in issue stage, get the target address from complete stage and turn on the retire bit in retire stage. After that, the store queue will check whether the entry at head has been retired. If this store instruction has been retired by ROB, the store queue will directly send the address and value to data cache and raise a flag telling the load register do not send load request to data cache in this cycle. When the store queue gets a flush signal indicating a reset or misprediction, it will flush all entries with a negative retired bit to ensure the correctness.

The load register complete the data forwarding task and send load request if this load need the data from data cache. It will first traverse the store queue and decide whether the load can get the value from forwarding or from data cache. If the store flag in this cycle is high, the load register will try to send the load request to data cache in the next cycle. After the load gets its value, the load register sends the load result to CDB pipeline register just like other functional units.

The size of the store queue is parameterized, and we finally chose the size of four in the baseline processor due to the following reasons. According to the lecture slide, the store queue size is at about 15-20% of the

ROB size. The other reason is that we need to traverse the store queue in each cycle, and the clock period will largely increase if we increase the store queue size. And we also find that the structural hazard signal for store queue is low for most of the time, so we finally make the store queue size four.

2.6.2. Data Cache

We implemented a 256-byte set-associative data cache, the specifications of which are listed as follows.

The cache is divided into 32 blocks of 8 bytes, or 64 bits. This block size is equal to the size of the data packet provided by the memory interface. The associativity of the cache is parameterized, with the baseline associativity in our `final_submit` branch being 2.

In the event of cache misses, the cache is configured as “write-allocate.” This means for any memory operation (both read and write) that causes a cache miss, the data cache will fetch the entirety of the 8-byte block referenced by the memory access from the system memory. When storing a newly fetched block into a full set in the data cache, the eviction policy is configured to randomly evict an occupied block in the set. A rotating priority selector is implemented to facilitate this functionality.

When executing memory store instruction, the cache is configured as “write-through.” This means that the store instruction would first write to a cache block (either already present in the cache on a cache hit, or recently fetched on a cache miss) and to its corresponding block in the system memory.

The data cache is configured as “blocking.” This means that only one memory operation (and the cache lookup, block fetch, block eviction, and write-through routines associated with it) is permitted to be executed at any given time. Memory operations that result in cache hits would complete in 1 clock cycle. Load operations that result in cache misses would complete in the same number of cycles as the memory latency. Store operations that result in cache misses would complete in twice as many cycles as the memory latency.

Given the above specification, the data structure of a cache block includes the 64-bit data, a tag, and a valid bit. The data structure of a cache set includes a number of blocks equal to the parameterized associativity of the cache and a “full” bit that is the conjunction of all the valid bits of its sets. Note that no dirty bit is necessary due to the write-through nature of the cache, and no “least-recently-used” bit is necessary due to the randomized eviction policy.

3. Performance

This section provides a discussion on the performance of our processor. Both the baseline processor included in our `final_submit` branch and variations of the processor with different advanced features and/or module parameter configurations are included in the performance analysis. The remainder of this section would present, in that order, our testing methodology, the implementation of our main test bench, an overview of the performance of our baseline processor, the impact of various novel advanced features on the performance, and the impact of different module parameters on the performance of the processor.

3.1. Testing methodologies

We implemented unit tests for all of our modules and made sure that each module is passing the unit tests in both simulation and synthesis before proceeding to implement the next module. During integration testing, we are also making sure that our entire pipeline is passing all test programs for both `make simulate_all` and `make simulate_all_syn` before adding each new advanced module into our pipeline. **Note that both the baseline processor and all variations based on it can produce correct results on all instructor provided test files, thus more detailed discussion on the correctness of our processor is not included in this report.**

3.2. Testbench Implementation

We modified the `testbench.sv` from Project 3 in the following manners to fit our out-of-order pipeline. We retained the feature to print the state of the pipeline during every clock cycle to a `.pp1n` file; however, we modified this feature such that more detailed information about the state of each module, pipeline registers and cache files are dumped to the output at each cycle to facilitate debugging. We also specified that after a `halt` instruction retires, the test bench would wait for all store queue entries to write to the data cache, then manually dump the entire contents of the data cache back into the system memory before reaching a `$finish` operation. This is to ensure correctness of the system memory by the time a program terminates.

We used Synopsys VCS to simulate our designs for correctness, and used Synopsys Design Compiler to synthesize our design and produce reports on clock periods and performance. For the latter toolchain, we specified “medium” mapping and area effort and “none” in terms of power effort during gate-level optimization in our Synopsys DC Shell scripts.

3.3. Performance Overview of "Baseline" Pipeline

Our submitted baseline processor can successfully run every provided test programs both in simulation and synthesis. Due to the different data flows and branch pattern, the outcome CPI varies, as shown in the following Figure 2.

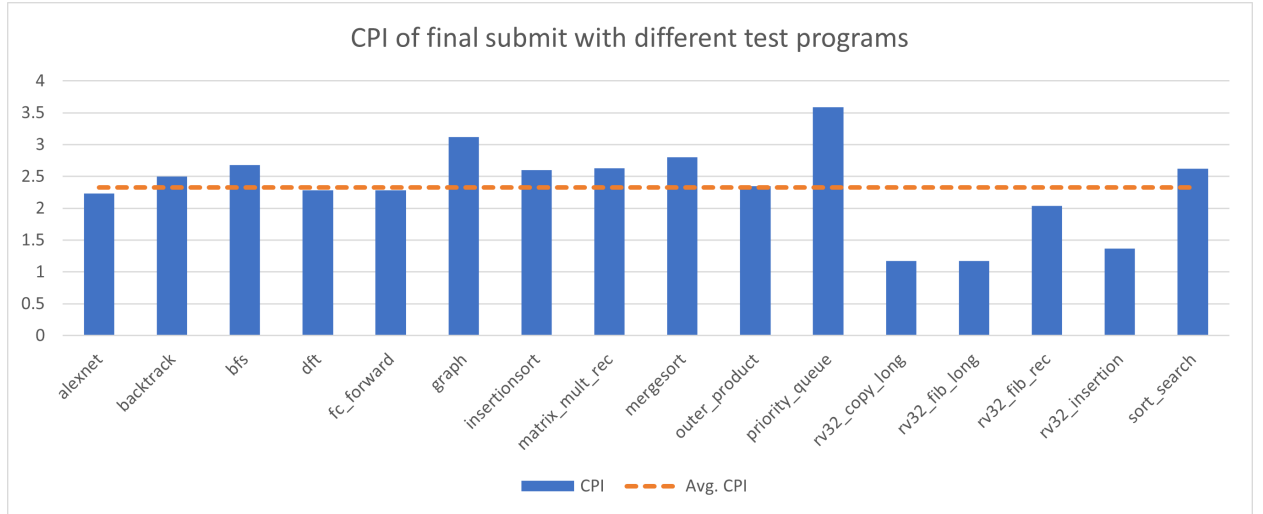


Figure 2: CPI of our final submit with different test programs.

For programs with appropriate use, our design achieves an average CPI of **2.32**. It should also be noted that our baseline processor achieves a clock period of **20 nanoseconds**.

3.3.1. Critical Path and Performance Bottleneck

From the `pipeline.rep` file generated by the Design Compiler, we find that the critical path in our processor is from ROB to BTB to ROB to Map table. This critical path occurs at retire stage. At retire stage, ROB sends its expected retired instructions to BTB. BTB then needs to confirm whether there is a branch prediction and generates flush signal. The flush signal will send to every module in the pipeline except for the instruction cache and BTB itself. The flush signal decides whether ROB needs to give up retiring and flush itself or ROB can retire normally. After retire packet is determined to be normal or flushed as empty, then the map table can know whether it needs to update or does nothing in retire stage.

3.4. Impact of the Presence of Advanced Features

This subsection investigates how the presence or absence of two advanced features – store-to-load forwarding and 32-stage pipelined divider – impact the performance of our processor. Note that store-to-load forwarding is present in our baseline processor, while the divider is not.

3.4.1. Store-to-load forwarding

An important feature implemented during the execution of memory operations is the store-to-load data forwarding. When dispatching a load instruction, it would be assigned with a age index. Later in the memory stage, the load would check whether the address of stores older than it are valid in the store queue. Once the older stores' addresses are valid, and is the same as the specific load, the data would be forwarded from the store queue. In this way, load instructions can save many cycles waiting for the stores to write through. In many C programs, a store is usually followed by a load from the same address, so the forward can help decrease the CPI.

We captured the rate of load forward in our processor. The result is presented in Table 3. The average cycle needed for a forwarded load valid in LSQ is 3. In contrast, without this function, the average cycle is more than 6 cycle as it have to wait for all older stores to send through the BUS. The CPI corresponding to both cases are also presented in Figure 3. Since the improvement in CPI is quite obvious, we kept this function in our final design.

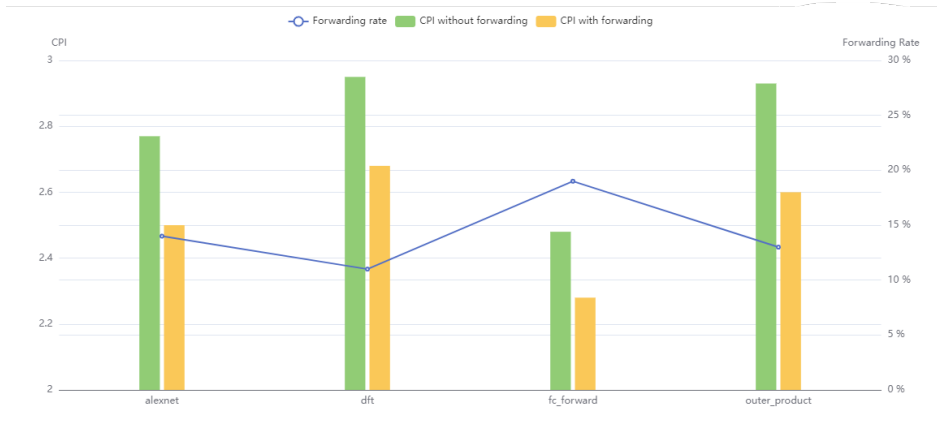


Figure 3: Store-to-load forwarding rates and comparison of CPI numbers with and without forwarding

3.4.2. Divider and the Full RV32 “M” ISA Extension

One of the major reasons of the exclusion of the divider module from the baseline processor is that it presented far less-than-expected performance benefits to the processor as a whole, while at a nontrivial cost to the correctness of other logic in the pipeline. Given that the intention of implementing the divider module was to support native division and remainder instructions from the “M” extension to the RV32 ISA, it was disappointing to see that only one of all the instructor-provided test programs utilize such native instructions at all. Namely, it was `insertionsort.c` that contains a single signed remainder operation in its entire source code.

The following Table 2 contains a comparison between the baseline processor with and without the divider module while compiling and executing the `insertionsort.c` test case.

It is worth noting that the major advantage of implementing the divider and native division and remainder operations is the decrease in code size. One single native remainder operation, instead of assembly-level workaround functions, can reduce the size of the code stored in memory by more than 6%. The actual number

Table 2

Comparison of baseline processor with and without the divider

Configuration	Assembly file size	Cycles	Instructions	CPI
With divider	9501 bytes	193272	73141	2.642
Without divider	10145 bytes	194511	74610	2.607
% improvement	6.35%	-0.64%	1.97%	-1.34%

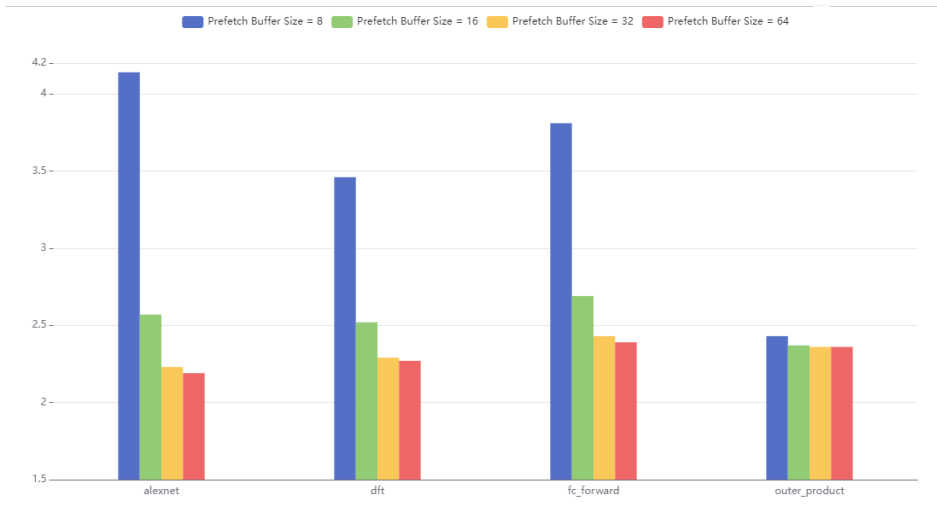
of instructions executed also decreases by almost 2%. However, the difference in total number of cycles, cycles per instruction, and execution time becomes negligible. The minimum clock period of the processor also stays the same at 20 nanoseconds. This is probably due to the fact that the SRT division algorithm would take 32 cycles to complete a single division or remainder operation, which may even cause additional stalls due to the reorder buffer waiting for the division to complete and filling up with later instructions. It should also be noted that it is very rare in practice for 32 consecutive division and/or remainder instructions to completely fill the 32-stage pipeline in the divider to achieve maximum performance and amortized CPI values, except for highly specialized hashing or cryptographic operations.

3.5. Impact of Different Module Parameters

This subsection investigates how changing the parameters and sizes of some of our modules impacts the processor's performance. The parameters of interest include the size of the prefetching buffer, the number of reservation station entries per bank, the size of the reorder buffer, associativity in the data cache, and the size of the branch target buffer.

3.5.1. Size of the Prefetching Buffer to the Instruction Cache

We tested the size of prefetch buffer for 8 bytes, 16 bytes, 32 bytes, and 64 bytes. The outcomes shows prefetch buffer with a air size can largely improve performance. It can be identified that only diminishing returns can be achieved for prefetching buffer size of over 32 bytes, while the cost in terms of area, power, and complexity of lookup logic may not justify those returns. As such, we keep the buffer size as 32 bytes (8 instructions). The result is shown in Figure 4.

**Figure 4:** Relationship between the size of the prefetching buffer and the cycles per instruction

3.5.2. Number of Reservation Station Entries

We have tried the size of RS for 4, 6, 8 and 10, with CPIs in the Table 5. The result showed that CPIs has a obvious improvement from 4 to 6. This is because our specific struct hazard computing method. when reservation station size increases to 6, new instructions can be dispatched if each banks has four busy entries, which is nearly double compared to the size of 4. But we finally choose size of 4 to preserve the clock period.

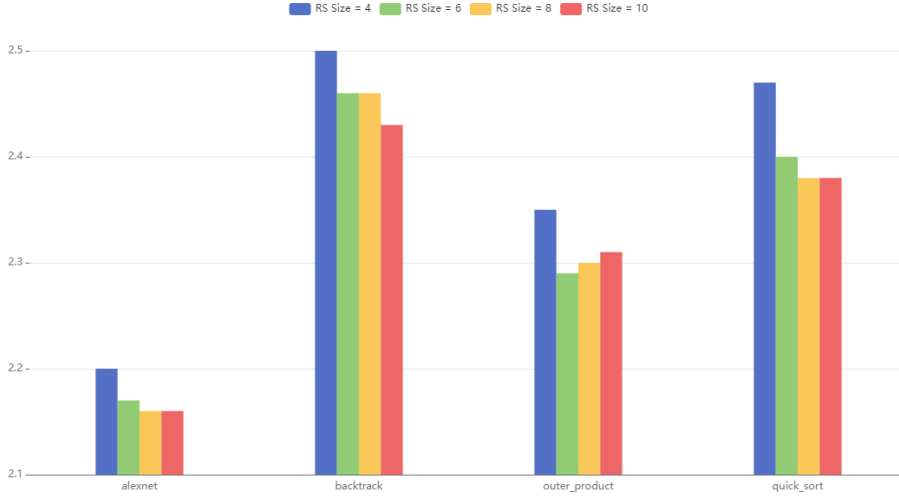


Figure 5: Relationship between size of the reservation station and the cycles per instruction

3.5.3. Size of the Reorder Buffer

We have tried the size of ROB for 16, 32 and 64, with CPIs showed in the Table 6 . We can also identify diminishing returns in cycles per instruction when the size of the ROB is greater than 16, while the potential for critical path due to more complex lookup logic no longer justifies the marginal additional returns. As such, we keep the size as 16 to save area and energy and preserve performance.

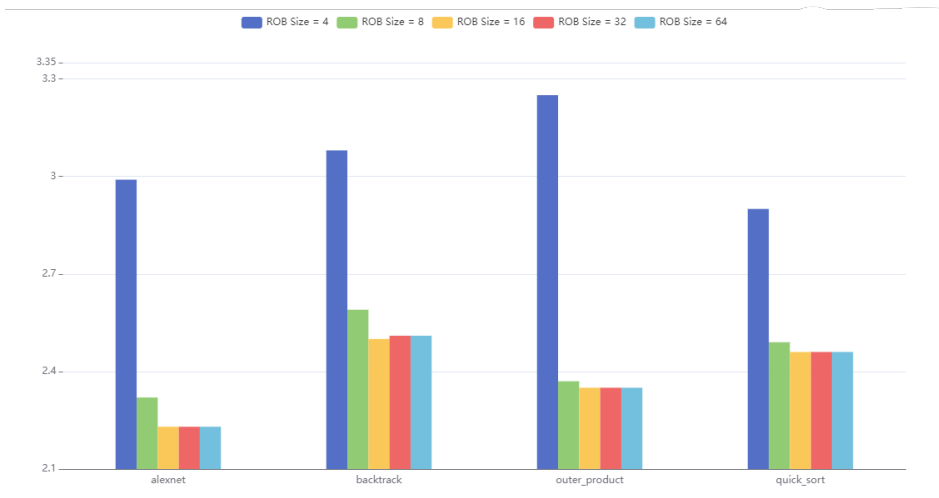


Figure 6: Relationship between size of the reorder buffer and the cycles per instruction

3.5.4. Associativity in the Data Cache

Since our data cache is configured as randomized evicting, a small set size could affect the locality, which could lead to unnecessary eviction of blocks. Therefore, we tried different associativity in our data cache to find a proper number of blocks per set. In general, the direct-mapped, 2-way, 4-way, 8-way and 16-way caches are tested. Since the phenomenon of frequent eviction usually appears in large programs, we picked several programs with most cycle. The result is presented in Figure 7. We found that CPI decreases when switching from direct-mapped to 2 way associativity, but the performance are similar when continue increasing associativity, so we kept a set size of two in our final design.

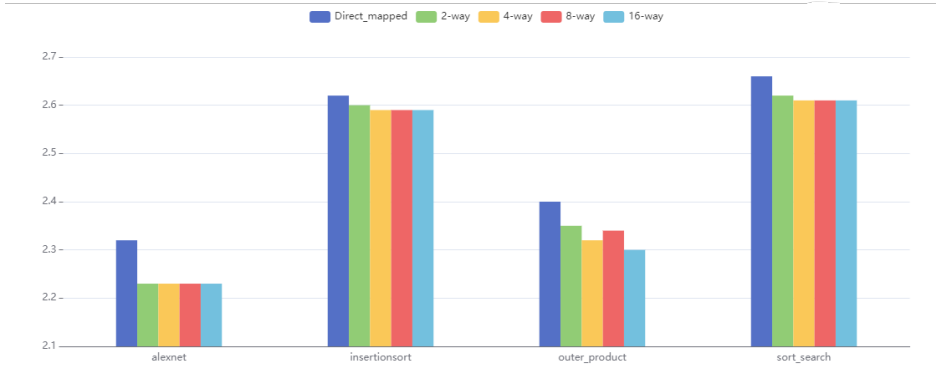


Figure 7: CPI for different dcache associativity.

3.5.5. Size of branch target buffer

Theoretically, larger size of branch target buffer will boost processor performance with more accurate branch prediction. However, the branch target buffer is a source of delay since it involves table entries traverse and compare. Too large size will result in significantly increased clock period. We tested our branch target buffer size for 4, 8, 16, and 32. Some test programs are very sensitive to the size of BTB, such as priority queue. While some programs have almost the same performance with different BTB size. Figure 8 shows the relationship between branch accuracy and BTB size. To enhance performance while keep clock cycle short, our design has a BTB size of 8 entries.

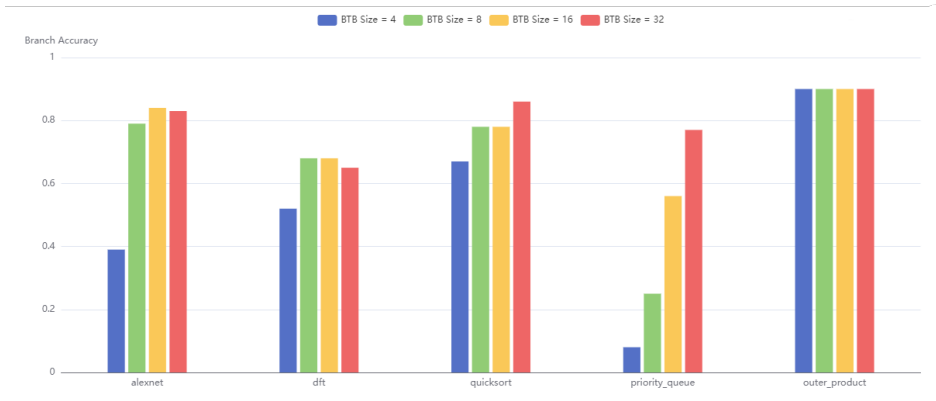


Figure 8: Branch Predict Accuracy for different BTB size.

4. Discussion

This section discusses both the societal implications of our processor and offers some reflection upon our project.

The out-of-order, superscalar processor implementation based on the RISC-V ISA in the style of the P6 micro-architecture has the potential to impact a variety of industries. On broader terms, the out-of-order and superscalar features of our processor can increase the efficiency and performance of embedded systems that utilize programs based on the RISC-V ISA. The configurable parameters for some of our modules also present opportunities to fine-tune our register-transfer-level design to specific applications. In the field of scientific research, faster and more efficient computing systems are crucial for running complex simulations and processing large datasets, which can help accelerate discoveries in areas such as physics, chemistry, and genomics. In finance, high-performance computing is essential for running simulations and performing data analysis, which can improve trading strategies and risk management. In the entertainment industry, faster computing systems can enable the development of more realistic and immersive virtual reality experiences, as well as advanced visual effects in movies and games. Overall, our implementation of the processor based on the open-source RISC-V ISA has the potential to drive innovation and progress in a variety of industries by enabling faster and more efficient computing systems that can transform how we work, learn, and live.

Our project has provided valuable insights into the challenges and complexities of processor design. We gained a deep understanding of the RISC-V ISA, micro-architecture design principles, and modern processor optimization techniques, and encountered challenges such as instruction ordering, power consumption, and timing constraints. Despite these challenges, we successfully implemented a functional processor that achieved significant performance gains and demonstrated the feasibility and benefits of out-of-order, superscalar processor designs based on the RISC-V ISA. Our project has provided valuable lessons in processor design, optimization, and trade-offs, which will be beneficial for our future careers in computer engineering and related fields.

Other takeaways from the project include different philosophies for implementing our processor and its individual modules. A top-down approach is conceived for module implementation, where our workflow goes from conceptualizing a module based on EECS 470 course content, to specifying how it should be implemented with detailed text descriptions and flow chart diagrams, to the actual realization in SystemVerilog code. In contrast, a bottom-up approach is utilized for building the entire pipeline. We strive to first implement individual modules, often working in parallel with different members tasked with different components, resolving bugs and other problems through unit testing. We would then put together the modules into clusters and pipelines, while ironing out any larger-scale idiosyncracies with integration testing. We find the combination of these differing approaches be most helpful in ensuring the most precise, robust, and efficient design workflow.

In retrospect, we also identified several improvements that can be made to further improve performance.

- In our final submission, the pre-decoder at BTB only determine whether the instruction is a branch instruction. However, For unconditional branch instruction "JAL", its branch target can be predecoded at BTB. The target can be sent to IF stage immediately to increase branch accuracy. We realized this issue when reading RV32 document after the final submission. It taught us a lesson that document reading is an essential part during a complex system design.
- As shown in figure 5, larger RS size can boost performance to some extent. We keep every RS bank size as 4 and never changed it during design. We did not realize that some function unit's performance (such as ALU) can be improved with larger RS size and the RS size can be different for each unit. We clearly learn a lot about the importance of testing during design.

5. Conclusion

Our group have designed and implemented a two-way superscalar out-of-order P6-style processor. Our minimum clock period reaches 20 ns and the average CPI of public testcases is 2.32. During the project, we established a strong interest towards designing, testing and improving a processor and were extremely excited when it worked after tons of debug operations. All included modules are integrated, implemented and tested in SystemVerilog and we can program proper C programs with our processor. The report keeps track of the features and attempts we made and how effective they were.

References

Gwennap, L., 1995. Intels p6 uses decoupled superscalar design. Microprocessor Report 9.