EECS 470 Final Project Report

Daniel Yu (dddaniel@umich.edu), Hsiang-Yang Fan (seanfan@umich.edu), Yan-Ru Jhou (yanruj@umich.edu), Hsin-Ling Lu (hsinling@umich.edu), Eli Muter (innitir@umich.edu)

Abstract

We aim at addressing one of the challenging tasks in microarchitecture, namely, designing an Out-of-Order pipeline microprocessor. In addition to the basic features required in the final project, we integrated advanced features (i.e., 3-way and even partially implemented N-way arbitrary superscalar) in order to leverage the pipelining performance. Thus, we present the 3-Way Superscalar R10K processor, a microarchitecture based on VeriSimpleV RISC-V.

Table of Contents

Introduction

We implemented the Out-of-Order R10K processor based on VeriSimpleV RISC-V. It utilizes a 3-Way Superscalar architecture, which allows it to manage up to three instructions in each of the fetch, dispatch, complete, and retire stages simultaneously. This allows the processor to significantly improve its performance and increase the speed at which it can complete tasks. Additionally, our R10K processor includes out-of-order execution and capabilities, further improving its ability to efficiently execute instructions.

The motivation for R10K over P6 is that we would like to achieve fast implementation and avoid the memory overhead of copy-based register renaming like P6. Meanwhile, to compensate for the strain on the main memory, we integrate Store Queue, I-Cache and D-Cache, into our memory hierarchy. To the best of our knowledge, this design has the potential to significantly improve performance and efficiency.

The main contributions of this paper are:

- **Out-of-Order R10K Pipeline Processor:** Our R10k-based uarch design extends the in-order pipeline processor from project 3 to increase overall performance by allowing out-of-order execution.
- **3-Way and theoretical N-way arbitrary superscalar:** We implemented 3-Way superscalar execution to allow up to three instructions in each of the fetch, dispatch, complete, and retire stages simultaneously. This improves the overall performance of the processor, especially for programs with fewer dependencies and/or dependencies that are separated by many instructions, since, theoretically, if instructions do not need to be stalled for any reason (*i.e.*, there are no structural hazards and no dependency hazards), our pipeline could complete a program three times faster than a non-superscalar implementation. Furthermore, in all modules except the FU, ICache, and SQ, we avoided hardcoding the superscalar value, so our implementation of superscalar execution only depends the abstract value of N-way superscalar defined by the Macro "SUPERSCALAR_WAYS". In theory, if the FU, ICache, and SQ were updated to also avoid such hardcoding, our pipeline could support arbitrary N-way superscalar execution.
- **Demonstration:** We showcase results of our microarchitecture processor on a variety of testbench from c programs and assembly codes.

The rest of this report is organized as follows. We first present our primary contribution of this paper, 3-Way Out-of-Order R10k microarchitecture design in *Design Overview*. We would provide comprehensive and detailed illustrations for each module in this section. Our verification methodology for R10k is illustrated in *Verification*. We would demonstrate how we verify both individual and overall modules against the functional correctness. We would discuss our advanced features for our microarchitecture in *Advanced Features.* Teamwork is depicted in *Teamwork*. Finally, the conclusion and acknowledgements are listed in *Conclusion* and *Acknowledgement*.

1 Design Overview

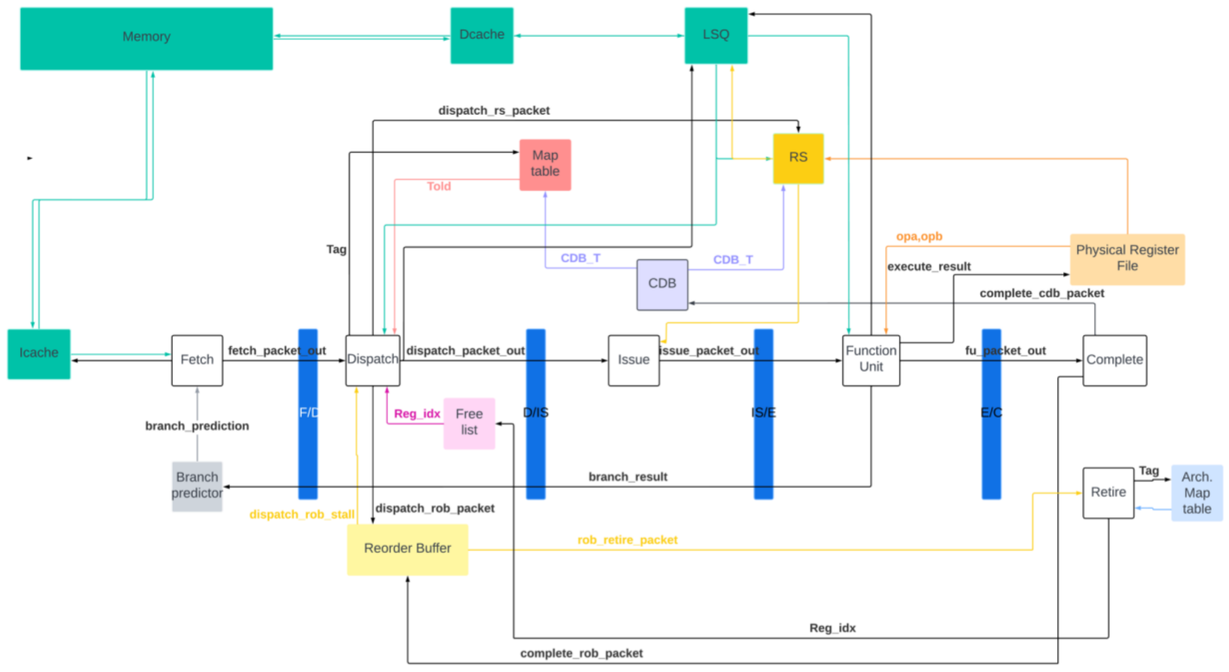


Figure 1. Design diagram for R10K processor

1.1 Stages

Our processor has 6 stages: Fetch, Dispatch, Issue, FU (Execute), Complete, and Retire.

1.1.1 Fetch

The fetch stage serves to forward instructions read from memory to the dispatch stage and to record the current PC of the instruction being fetched from memory. The current PC is held in an array of PC values, one for each superscalar way. The fetch stage receives input from the dispatch stage and the retire stage, and sets the PC according to these signals.

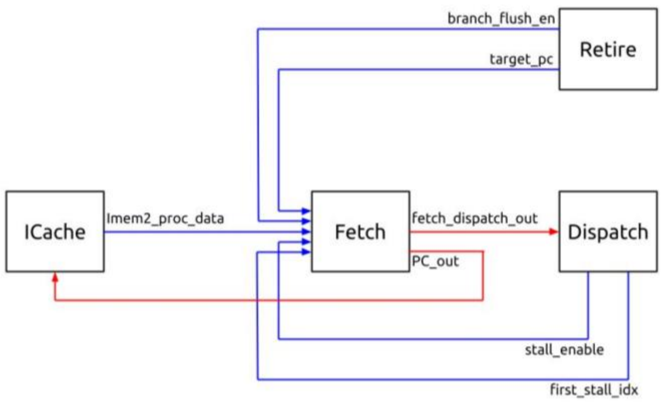


Figure 2. Connections with the fetch stage

The input from the retire stage indicates whether or not a branch was falsely predicted not-taken. If there was a taken branch, the fetch stage invalidates the current instructions being fetched, and sets the next PC values according to the branch's target PC.

If the input from the retire stage does not signal a branch correction, the fetch stage will consider the input from the dispatch stage. These signals indicate whether or not the fetch stage needs to stall the instructions due to a stall in the dispatch stage, and it increments the PC values according to the number of instructions being stalled. If there are no stalls, the fetch stage increments each PC value once every clock cycle according to the number of superscalar ways (*e.g.*, a 3-way superscalar implementation would have fetch increment each by 12). However, if there are stalls, the fetch stage only increments the PC values by the number of new instructions that can be dispatched. For example, a 3-way superscalar implementation where dispatch is stalling 2 instructions would lead the fetch stage to increment each PC value by 4, since the first instruction will be dispatched.

1.1.2 Dispatch

The dispatch stage decodes instructions and allocates the necessary physical registers, reservation station entries, and rob entries for each instruction. It receives input from the fetch stage, the architectural maptable, the RS, the register freelist, the ROB, the CDB, and the retire stage.

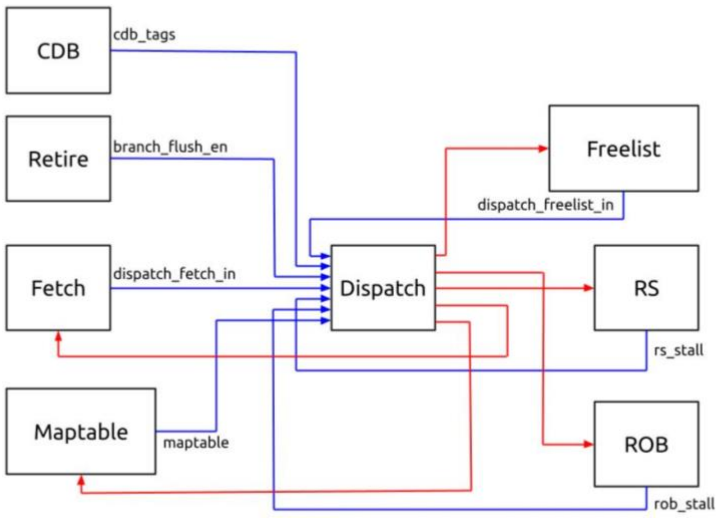


Figure 3. Connections with the dispatch stage

The dispatch stage first decodes each instruction received from the fetch stage. Then, if the instruction has a destination register, dispatch will assign the corresponding architectural register a physical register according to the input from the freelist, send one signal to the freelist that the physical register is no longer free, and another to the architectural maptable indicating the values that need to be updated.

Then, dispatch will check through the CDB tags, as well as the previous instructions being dispatched in the same cycle, for matches in the source register indices, and update the instruction data accordingly. For example, if the maptable indicates that a source register is incomplete, but the register is included in the CDB tags, dispatch will mark it as ready. Then, if a source register is the destination register for a previous instruction dispatched in the same cycle, dispatch will update the source register's recorded physical register index to correspond to that assigned to the previous instruction, and mark it as not ready.

Finally, the dispatch stage will send signals to the RS and ROB allocating new entries in both for the new instruction.

However, if the input signals from (a) the freelist indicate that there are not enough free registers, (b) the RS indicate that there is not room in the RS, and/or (c) the ROB indicate that there is not room in the ROB, dispatch will stall the necessary number of instructions, output a signal to the fetch stage indicating that there's a structural stall, and disable all changes (the output to the maptable, freelist, RS, and ROB) for all instructions that must be stalled. The only exception to this stall decision is if the input from the retire stage indicates that a branch was falsely predicted not-taken, in which case the output of the dispatch stage does not matter, since all the modules it sends output data to will be cleared regardless.

1.1.3 Issue

The issue stage serves to pass control signal from the reservation station to the fu (execute) stage. It receives input from the RS and the PRF. The issue stage propogates the instructions' data from the RS to the FU, and decodes the register value by getting the real value from the physical register file.

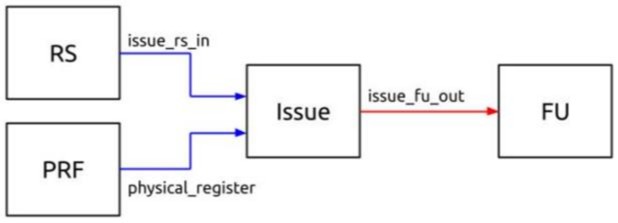


Figure 4. Connections with the issue stage

1.1.4 FU (Execute)

Our function unit includes one branch resolver, two load units, three arithmetic-logic units, and two multiplier units.

The function unit will distribute the input issue packet to the different units due to its function select signal. Different units require differing numbers of cycles to complete. The arithmetic-logic units (including the branch resolver) take one cycle to execute, and the multiplier units take four cycles to finish. The function unit module will choose a total of three execution results from each unit as a packet, make the packet in the order of their rob entry index, and then pass the instruction data to the complete stage.

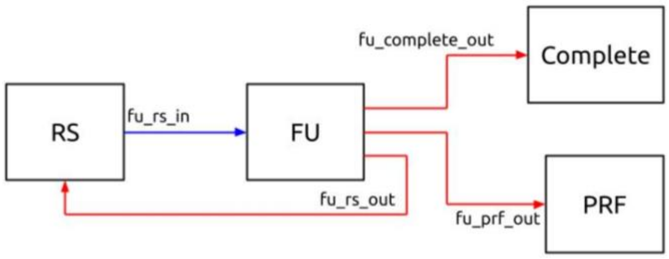


Figure 5. Connections with the FU (execute) stage

When more than three units want to complete in one cycle, the function unit will choose three to pass to the complete stage, store the rest packets in the register, and pass to the complete stage at the next cycle. When the unit's result is stored in the register or the unit is still executing something, it will output a busy signal to the RS indicating that it should not pass a new packet into the unit.

1.1.5 Complete

The complete stage sends a signal to the ROB for each valid, completed instruction, and broadcasts the value of each completed instruction that has a destination register (*i.e.*, instructions other than branches, noops, and stores) across the processor, to ultimately be used by the dispatch stage and the reservation station.

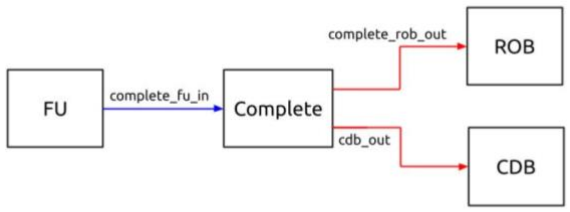


Figure 6. Connections with the complete stage

1.1.6 Retire

The retire stage serves to retire instructions from the reorder buffer. It receives input signals from the ROB and the architectural maptable, and records the processor's precise state as an architectural maptable.

Each clock cycle, retire checks the input from the ROB for complete instructions. If any instructions are ready to retire, the retire stage checks if those instructions are halts or branches, stopping on the first of either that it comes across. If a halt is to be retired, the retire stage outputs a signal indicating that the processor has been halted. If a branch is to be retired, the retire stage outputs a signals indicating that a branch has been falsely predicted not-taken, the branch's target PC, and the architectural maptable containing the precise state that should be used to recover from the branch.

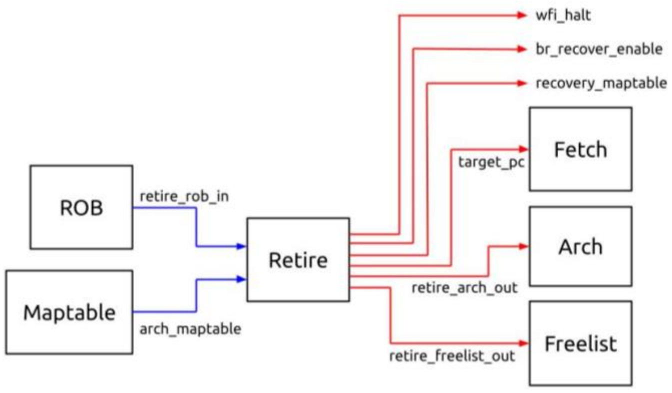


Figure 7. Connections with the retire stage

For each retired instruction, the retire stage sends a signal to the architectural maptable and the freelist indicating the registers that can be marked done or as free, respectively.

1.2 Memory

Our processor has 3 main memory modules: the instruction cache (ICache), the data cache (DCache), and the physical register file (PRF).

1.2.1 Instruction Cache

Our instruction cache is non-blocking and directly mapped. The total size of the ICache is 256 bytes, with 32 lines and 8 bytes per line. The basic behavior of the ICache is pretty simple; if the cache hits, the cache will send the data and valid bits to the fetch stage. If the cache misses, the fetch stage will first require data from the cache, the cache will then send the request to the memory. Finally, the memory will send the data or address back to where it's needed.

The interesting part of the ICache is the connection between the cache and the fetch stage. The fetch stage encounters a stall situation from the dispatch stage. The fetch stage will send the choose address signal to Icache. For example, if the dispatch stage stalls at the first stall index equal to 2. The fetch stage will then send the chosen address signal to the cache in order to select the accurate address. The connection between the memory and the cache is important, as well. Hence, we created a module that can control whether the address is written to or read from the memory. After receiving the write or read index from the output of the cache. The controller will send the output data and valid bits back to the cache for further application.

1.2.2 Data Cache

The DCache in our project is non-blocking and directly mapped. The total size of the DCache is 256 bytes, with 32 lines and 8 bytes per line. If the memory that is to be written is found in the DCache, the DCache will be updated. The DCache is designed to follow a write-back procedure. Therefore, data written in the cache won't go directly back to the memory, but will be marked as dirty. If the data is selected to be evicted, the lines that are marked dirty will then be written back to memory. If the memory encounters write misses, the DCache will load the address' data from the memory, and then overwrite the data that was missed, so DCache will eventually get the value.

1.2.3 Physical Register File

The PRF records the value stored in each physical register index. It receives input signals from the FU (Execute) stage indicating registers that need to be updated, as well as the new value to be stored in the registers. It outputs a signal to the issue stage containing the values stored in each physical register.

1.3 Modules

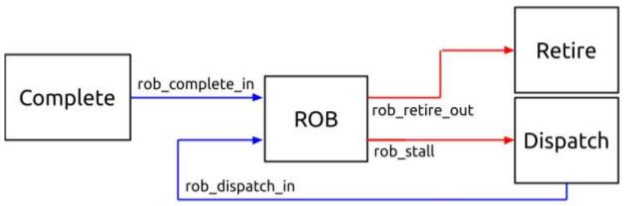
Our processor has 6 non-stage, non-memory modules: The reorder buffer (ROB), the reservation station (RS), the freelist, the map table, the architectural map table, and the store queue (SQ).

1.3.1 Reorder Buffer

The ROB records instruction data for all dispatched, but not yet retired instructions. Our ROB can hold up to 16 entries. It functions as a circular queue with head pointer and a tail pointer which will only be equal to each other when the ROB is empty or contains only one instruction. The ROB receives input from the dispatch and complete stages.

Each clock cycle, the ROB first determines the instructions that can be retired, if there are any.

Figure 8. Connections with the reorder buffer



Beginning at the head pointer, the ROB outputs a signal to the retire stage for each instruction that is complete, stopping when it reaches a halt, a branch falsely predicted as not-taken, or it's retiring the maximum number of instructions allowed by the number of superscalar ways, whichever occurs first, and incrementing the head pointer for each instruction that is retired.

Then, the ROB iterates through the input signal from the complete stage, and marks each valid, completed instruction as complete in the ROB.

Finally, the ROB handles dispatched instructions. If there is not enough room in the ROB to add all the dispatched instructions, the ROB sends a signal to the dispatch stage indicated that it must stall the corresponding number of instructions (*e.g.*, in a 3-way superscalar implementation, if the ROB only has one unused entry, it's output to the dispatch stage would indicated that the latter 2 instructions must be stalled). For each valid instruction that can be allocated a ROB entry, if the input from dispatch indicates that the instruction is enabled (*i.e.*, it does not need to be stalled for any other reason), the ROB does so, and increments the tail pointer for each instruction that is allocated an entry. The only exception to the tail pointer incrementation is if the ROB was previously empty, either because it was empty the previous cycle or because all the entries previously in it have been retired and none have yet been dispatched.

1.3.2 Reservation Station

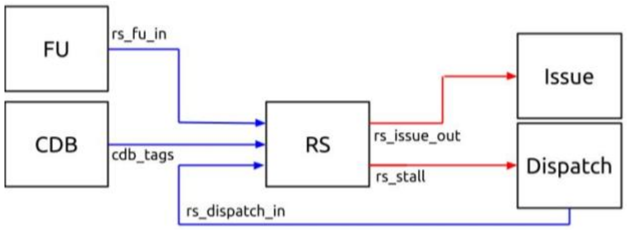


Figure 9. Connections with the reservation station

There are 16 entries for RS. There is no internal forwarding (*i.e.*, an entry cannot be issued and dispatched at the same time). The RS assigns fresh entries for freshly dispatched instructions, keeps an eye on the tag broadcast from complete stage, finds the instructions that are independent and ready to proceed, and issues those. Our RS logic uses a priority selector to allocate entries from the top of RS to entries from the dispatch stage and issue instructions from the bottom of RS to the issue stage. So, RS would be in favor of issuing older instructions.

RS is capable of issuing at most 3 instructions to Issue stage. Since we have 3 ALUs, 2 Mult units, and 1 Branch unit in the FU stage, our RS logic would try to select at most 3 ready instructions. However, Mult units take varying amounts of time to be complete, so there is a good chance that both Mult units are full. To address the Mult capacity issue, we take a conservative approach and allow only at most one Mult instruction to be issued when there is only one instruction being executed in Mult to ensure there is always enough space in Mult.

1.3.3 Freelist

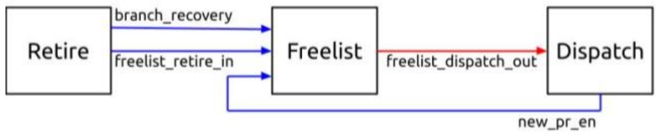


Figure 10. Connections with the freelist

We implemented a 32-entry Freelist with internal forwarding. In each cycle, it will send three physical register indexes to the ROB, and then the ROB will decide whether or not the register will be used. The logic consists of three priority selectors. In addition, our freelist supports N-way superscalar execution.

The size of the Freelist is designed to avoid structural hazards which are only due to itself. Since we have 32 architectural registers and the ROB has 32 entries, it is not possible that the Freelist would be empty before the ROB is full. Therefore, there is no structural risk to the Freelist only.

When a precise state exception occurs, the Freelist uses the recovery maptable from the retire stage to recover the Freelist.

1.3.4 Map Table and Architectural Map Table

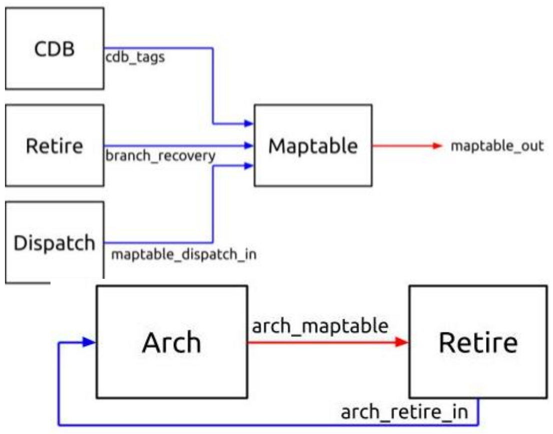


Figure 11. Connections with the map table and architectural maptable

The map table and architecture map table are the modules used to implement register renaming in our R10K style microprocessor. We have 32 architectural registers and 64 physical registers.

The architecture map table holds the actual value of physical registers and will update its value after an instruction with a destination register is retired.

The map table holds the index of the physical register, which is assigned by the freelist. Map table also holds a done status for the physical register in the module, and the done status is used to track whether the value of the physical register is done or not. The dispatch stage will get the corresponding done status of the physical register. The map table supports internal forwarding when the CDB sends the done signal in the same cycle.

When branch misprediction happens, the map table will copy the architecture map table to resolve the precise state exception.

1.3.5 Store Queue

Memory store operations are stored and reordered in the store queue. In our design, we implement an 8-entry store queue with internal forwarding. Our store queue is connected to the dispatch stage, RS, FU, DCache, and retire stage.

First, the dispatch stage will send a store signal to the store queue. Then, when the instruction is completed in the ALU, the FU (execute) stage will pass the value into the store queue. When the store instruction retires, it will send the data to the DCache.

2 Verification

Hardware verification can help ensure the functional correctness of the specification. However, such verification would require both manual and automated verification strategies, which are tedious and error-prone. In response, we designed three types of verification: module verification, progressive integration and verification, and pipeline verification with automated testing.

2.1 Independent Module Verification

We first began implementing the pipeline microarchitecture from scratch. After we completed the verification phase, *i.e.*, module verification. We ran comprehensive test suites against each module. Without connecting each module, we instead used fake data to simulate the input and output from the module and verified that each module's output was as expected.

2.2 Progressive Integration and Verification

After verifying the behavior of all individual modules, such as the ROB and RS, we then connected all modules one after another. Specifically, we first tested the fetch stage. We then connected the fetch with the dispatch stage, and then the fetch and dispatch stages to the ROB, etc. We called this approach "progressive integration," since we were slowly building our pipeline, verifying its behavior after each successive module integration. This strategy allows us to catch bugs much more easily than directly verifying the entire pipeline processor.

2.3 Pipeline Verification and Automated Testing

Once our progressive verification was completed, we finally moved to the entire pipeline verification. We use print tasks to print all pipeline stage outputs, including output from individual modules. These debugging information would be generated in pipeline.out.

Furthermore, automating the verification procedure is very useful. As a result, we wrote 2 scripts to auto test our pipeline's output, "testSim.sh" and "testSyn.sh," which simulate and synthesize the pipeline, respectively. Both scripts compare the pipeline's output, specifically the "writeback.out" and "program.out" files, to that of the P3 pipeline for each public test case, as well as some simple test cases we wrote ourselves. Verification condition is based on the exact match for the "writeback.out," and for lines in "program.out" beginning with "@@@" (*i.e.*, the lines in "program.out" which detail the final memory state and the stop condition of the pipeline (wfi halt, illegal instruction exception, etc.).

2.4 Testing Results

We have verified our processor against all the public cases and some assembly programs that we wrote ourselves. We have achieved a 100% pass rate on the programs we wrote, and we have passed the public cases "rv32_halt.s" and "mult_no_lsq.s." We achieve timing closure of the design at 12.0 ns. The average CPI for our processor is 4.77. The program CPI is presented in the following table:

Table 1. Processor CPI for Public Test Cases

Program	Number of Cycles to Complete	Number of Instructions	CPI
rv32_halt.s	7	1	7
mult_no_lsq.s	719	283	2.54

3 Advanced Features

We fully implemented and integrated 3-way superscalar execution in our processor, and partially implemented N-way arbitrary superscalar execution. More detail on the advanced features of out pipeline can be found below.

3.1 3-Way Superscalar Execution

We implemented 3-Way superscalar execution to allow up to three instructions to be handled in each of the fetch, dispatch, complete, and retire stages simultaneously. This improves the overall performance of the processor, especially for programs with fewer dependencies and/or dependencies that are separated by many instructions, since, theoretically, if instructions do not need to be stalled for any reason (*i.e.*, there are no structural hazards and no dependency hazards), our pipeline could complete a program three times faster than a non-superscalar implementation.

3.2 N-way Arbitrary Superscalar Execution

We partially implemented N-way superscalar execution in our pipeline. In all modules except the FU, ICache, and SQ, we avoided hardcoding the superscalar value, so our implementation of superscalar execution only depends on the abstract value of N-way superscalar defined by the Macro "SUPERSCALAR_WAYS". In theory, if the FU, ICache, and SQ were updated to also avoid such hardcoding, our pipeline could support arbitrary N-way superscalar execution, although we have not fully integrated this feature into our pipeline, and therefore have not tested the feature.

4 Teamwork

All members of our group contributed equally to the project, and while our efforts often overlapped, certain members were primarily responsible for different modules and features. The breakdown of primary responsibilities is as follows:

1. Yan-Ru Jhou: Reservation Station, Decoder, Precise State, Pipeline and testbench
2. Daniel Yu: Map Table, Physical Register File, Issue Stage, Function Unit
3. Hsing-Ling Lu: Freelist, Complete Stage, Store Queue
4. Hsiang-Yang Fan: Fetch Stage, Retire Stage, Architecture Map Table, ICache, DCache
5. Eli Muter: Dispatch Stage, Reorder buffer, verification, 3- and N-way superscalar

Conclusion

Based on VeriSImpleV RISC-V, we designed and implemented a R10k Out-of-Order processor that supports 3-way superscalar execution. This microarchitecture design paved the way for exploiting more advanced processors in the future such as parallel computer architecture. Since we chose 3-way superscalar execution, we optimized our processor to compensate for low IPC.

We believe that our efficient and performant processors can have a positive impact on society by enabling faster and more powerful computers and devices. This can improve access to information and technology and enable people to accomplish more in less time. This could further enable the development of new and innovative applications. Additionally, out-of-order pipeline processors can be more energy-efficient than other types of processors, which can help to reduce greenhouse gas emissions and contribute to a more sustainable future.

Acknowledgements

We would like to express our gratitude to the Computer Aided Engineering Network (CAEN), Michigan Engineering at the University of Michigan—Ann Arbor (UMich), for their server equipment with this project. Also, we are grateful to the following people: Prof. Ron Dreslinski, Brendan Freeman, Nevil Pooniwala, and Mason Nelson for their useful discussion and insights.