Design and Development of spenserCPU, an Out-of-Order RISC-V 32I Processor

Final Project Report

Hassan Farooq, Spenser Fong, Timothy Vitkin
hfaroo9, sfong5, tvitkin2

Professor Jian Huang
ECE 411, Spring 2022

Table of Contents

I. Introduction 3
II. Project Overview 3
III. Design description 4
A. Overview 4
B. Pipeline Stages 4
IV. Milestones 6
V. Advanced design options 7
A. Tomasulo Algorithm 7
B. Parameterized Cache 8
VI. Conclusion 9
Appendix 10

I. Introduction

This project was the design and development of a speculative Tomasulo processor that uses the RV32I base integer instruction set from RISC-V, an open standard instruction set architecture. Our motivation for this project was to develop a deeper understanding of computer architecture and the implementation details, and we chose to design an out-of-order processor over an in-order processor to learn more about how many modern processors are made today.

The rest of the report will be organized into four main sections: (a) design description, (b) milestones, (c) advanced design options, and (d) conclusion.

II. Project Overview

The Tomasulo algorithm is a very advanced design feature, so we wanted to focus all of our time to make sure of the basic Tomasulo datapath functionality first. Our sharing of work was a mix of pair programming and work distributed to each member.

III. Design description

A. Overview

We implemented an out-of-order design following the speculative Tomasulo algorithm and the RV32I instruction set. RV32I is a load-store architecture that is byte-addressed. We loosely based the design off the original Tomasulo paper[1], but added an additional instruction decode/instruction dispatch stage to our pipeline in order to increase our Fmax.

B. Pipeline Stages

1. *Instruction Fetch*

   Our instruction fetch pipeline stage utilized a static, not-taken branch predictor in order to constantly prefetch the next instruction. Instructions were loaded into an instruction queue with eight entries, however in practice we found that instruction queue entries were never populated since we were not prefetching fast enough to require it.

2. *Instruction Decode/Instruction Issue*

   Every clock cycle, the decoder stage reads the head of the instruction queue, decodes the instruction, and sends information to specific units based on the instruction data. Loads/stores were issued to the load/store queue, branches and arithmetic comparisons were issued to the comparator reservation stations, and all instructions requiring non-comparison arithmetic operations were issued to the arithmetic reservation stations. Concurrently, it sent information about register tag information to the reorder buffer in the `rob_values_t` struct (Appendix). If there was no space in any of these data structures, the decoder stopped accepting new instructions from the instruction fetch stage, effectively stalling instruction issues.

3. *Execute*

   We had two six entry reservation stations - one pair for our ALU's and another for our comparators. Each entry in the reservation stations had its own ALU/comparator unit, and thus in total we had 6 ALU's and 6 comparators. The load-store queue had 8 entries because we found that it was very common in the provided test code to have multiple loads/stores in a row and having a larger load-store queue meant that we did not have to stall decode as often while waiting for a memory operation to complete.

4. *Commit*

   Commits were handled with a reorder buffer (ROB). Most instructions are committed when they meet two criteria: (a) the instruction is at the front of the queue, and (b) a valid value has been loaded into that instruction reorder buffer entry. Additional logic was

[1] Tomasulo, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." *IBM Journal of Research and Development*, vol. 11, no. 1, 1967, pp. 25–33., https://doi.org/10.1147/rd.111.0025.

added to the ROB for branch, JAL, and JALR instructions. This was needed since we don't need to commit branch instructions to the register file and may need to change the PC based on the branch calculation. With JAL/JALR instructions, we need to edit the PC in addition to committing data to the register file.

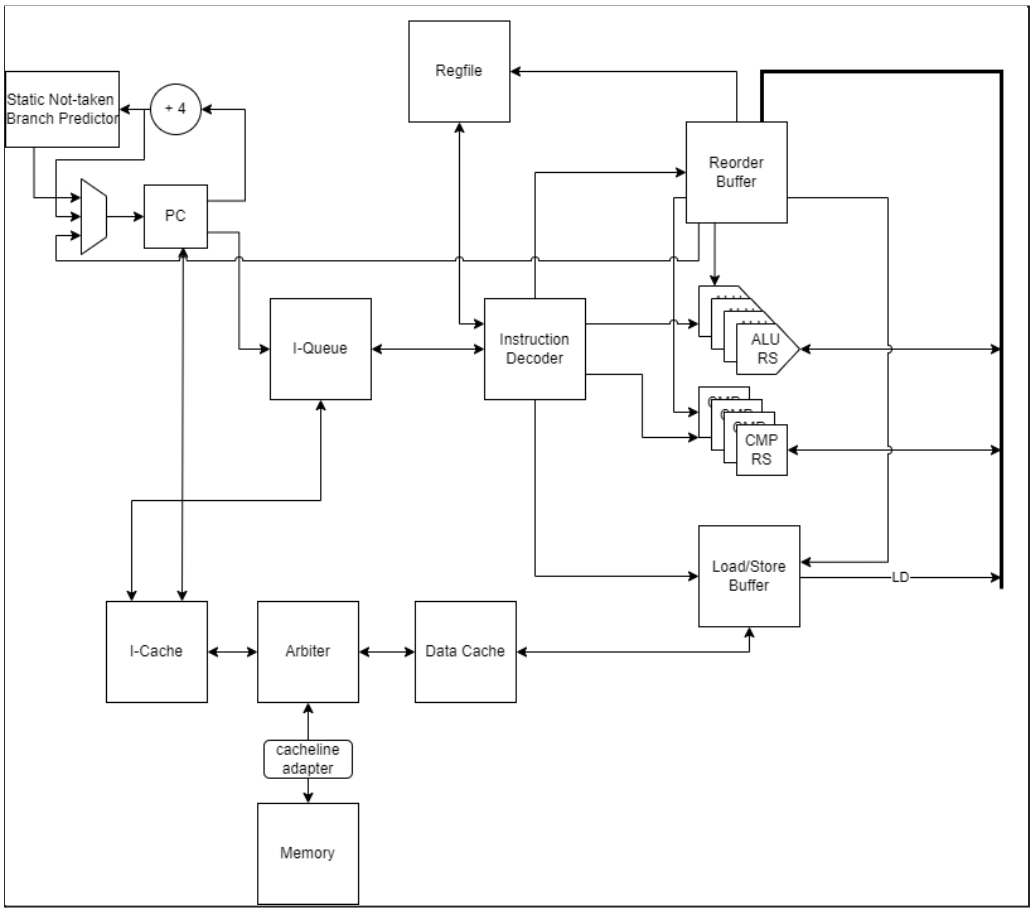

Fig. 1. Datapath

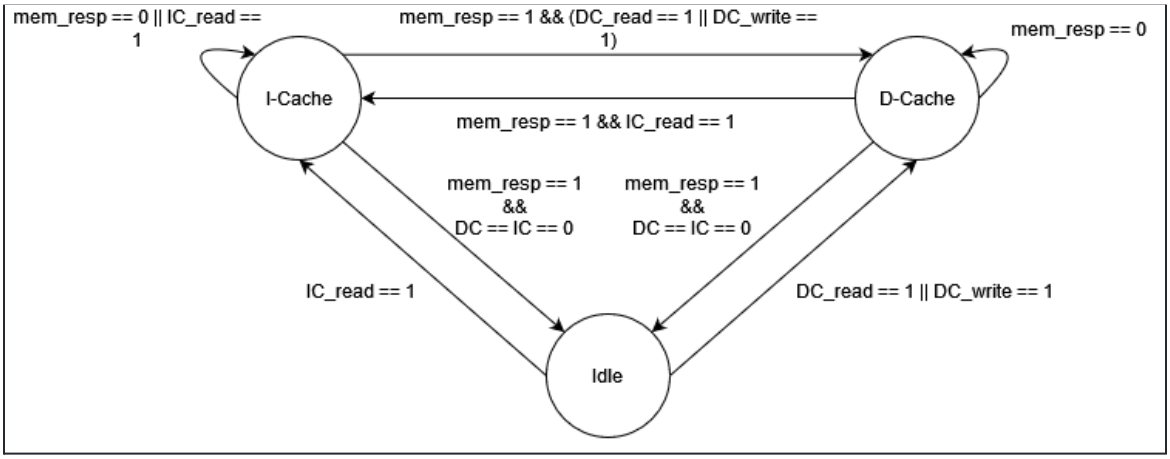

Fig. 2. Cache arbiter state machine

IV. Milestones

A. Design Checkpoint

A paper design of the datapath was made similar to that shown in Figure 1. This was later modified as we implemented the design and realized certain parts were missing or did not function as we initially thought.

B. Checkpoint 1

The instruction queue was implemented and fully verified for this checkpoint. Testbenches with modelsim were our main tools for debug and verification throughout the project. We verified modules based on the assumptions made in our paper design of how each should individually work. Modules for the ALU and compare logic unit were created, along with a draft of the PC register code and reservation stations.

C. Checkpoint 2

Our workflow was connecting all the modules and debugging from there. Functionality was made in an incremental manner. We got instruction fetching working first. We then implemented most of the reorder buffer, reservation stations, load-store buffer, and decoder, as there were still some edge cases that had not been accounted for. An arbiter design was also completed as seen in Figure 2. By this point, our datapath did not fully function.

D. Checkpoint 3

Our code was able to run the CP1 test code without caches. We integrated the split caches and arbiter (Fig. 2) afterwards, and we were able to run the CP1 test code with caches. The cache we used was a direct-mapped, single-cycle hit cache, which was given to us by course staff. We continued to debug our code for the CP2 test code.

E. Checkpoint 4

We finished adding JAL/JALR issuing logic in our decode/issuer and continued debugging. We were able to get CP2 test code running. While CP3 test code partially ran, our processor ended up stalling on it about halfway through. We were unable to fix this issue and thus were unable to run CP3 or any of the competition codes correctly, nor any of the competition codes. While debugging we realized the issue was with our tag array updating a clock cycle later than we expected, or in some instances never at all.

V. Advanced Design Options

A. Tomasulo Algorithm

1. Design

Explained in the design description section.

2. Testing

As mentioned in the milestones section, we first verified modules with an assumption of how each should work individually. We then wrote targeted tests to address edge cases, and compared the golden output waveform of the testcodes to our own waveforms.

3. Performance analysis

Our processor performed better than the MP2 processor for all the codes we could run (CP1 and CP2). This was due to the many load and store instructions in those test codes.

The Fmax values for our processor were very similar to those of MP2. We expected our Fmax values to be lower than MP2 since there is a lot more hardware to potentially increase the critical path. The high Fmax values are probably due to our decode and issue stages being clocked. The processor probably would have had lower Fmax values if the decode and issue stages were combinational. The clocked decode and issue stages essentially created an extra stage for the pipeline, which was different from our intended design of having decode and issue as one stage.

The power consumption was considerably higher than MP2, which was expected since there is a lot more hardware and control overhead in an out-of-order processor, mainly the reorder buffer and the reservation stations.

Table 1
Performance Comparison for Checkpoint Codes

	MP2	**MP4**	**Speedup**
CP1	33.505µs	7.345µs	356.16%
CP2	15.955µs	6.406µs	149.10%

The MP2 processor was a multi-cycle implementation using magic memory instead of caches.

Table 2
Performance Comparison with Fmax

	MP2	**MP4**
Slow 900mV 100C	105.15MHz	104.8MHz
Slow 900mV -40C	112.08MHz	111.54MHz

Table 3
Performance Comparison with Power

	MP2	**MP4**
Total Thermal Power Dissipation	392.53 mW	699.94 mW
Core Dynamic Thermal Power Dissipation	33.18 mW	327.22 mW
Core Static Thermal Power Dissipation	318.97 mW	322.39 mW
I/O Thermal Power Dissipation	40.39 mW	50.32 mW

B. Parameterized Cache

1. Design

For our second advanced design option, we decided to parameterize the number of sets of the given cache. We did this in a hierarchical fashion, allowing one to only need to set the number of sets at the cache's top level instead of at each submodule that deals with sets.

2. Testing

Testing for this design option was very simple, since parameterization does not affect functionality. As a sanity check, golden outputs were used to determine whether the processor with the parameterized cache was working correctly.

3. Performance analysis

There are no performance benefits here, but this design option allows one to more easily change the parameters inside the processor when doing optimization, speeding up the workflow.

VI. Conclusion

During the past few weeks, we designed and developed a speculative Tomasulo RISC-V 32I processor. Over the course of the project, our design changed multiple times as we better understood how Tomasulo's algorithm worked and better understood the signals and modules we would need. Our final processor consisted of a 6 entry ALU reservation station, with each entry tied to its own ALU, a 6 entry comparator reservation station, with each entry tied to its own comparator, a 8 entry load store buffer, and an 8 entry reorder buffer. This was in addition to our issuer/decoder, register file, instruction queue, instruction and data caches, and arbiter. While we worked hard to debug, we were unable to get all checkpoint code and competition code to run as expected on our processor. In the end we were able to only get checkpoint 1 and 2 code running correctly. When looking back at our code and discussing it with our assigned TA's, we realized that the issue likely lies somewhere within our decode and tag array logic. Since our tag array was not updating as we expected, we were sending incorrect ROB tag information to our logic units which thus caused our processor to stall at random parts of the program. Had we had more time, we would have further debugged this issue, and would have likely switched our decode/issuing logic to be combinational instead of clocked to try and fix this bug. In the end, even though our processor didn't function fully as expected, we learned a great deal and understand Tomasulo's algorithm much better than before.

Appendix

```
// array.sv
`define ARRAY_S_INDEX 3
`define ARRAY_WIDTH 1


// i_queue.sv
`define I_QUEUE_ENRTRIES 8


// pc_reg.sv
`define PC_REGISTER_WIDTH 32


// reg.sv
`define REGISTER_WIDTH 32


// alu_reservation_station.sv
`define ALU_RS_SIZE 6


// cmp_reservation_station.sv
`define CMP_RS_SIZE 6


// alu_reservation_station.sv
`define LDST_SIZE 8


// ro_buffer.sv
// 8 + 1 because entry 0 is reserved
`define RO_BUFFER_ENTRIES 9


`define NUM_CDB_ENTRIES 13
```

```
`include "macros.sv"


package structs;
import rv32i_types::*;


typedef logic [$clog2(`RO_BUFFER_ENTRIES)-1:0] tag_t;


typedef struct packed {
   logic type_of_inst; // 0 = load, 1 = store
   rv32i_word vj;
   rv32i_word vk;
   tag_t qj;
   tag_t qk;
   rv32i_word addr;
   logic [2:0] funct;
   tag_t tag;
   logic valid;
   logic can_finish;
} lsb_t;


typedef struct packed {
   rv32i_word pc;
   rv32i_word next_pc;
   rv32i_word instr;



} i_queue_data_t;


typedef struct packed {
   rv32i_word instr_pc;
   rv32i_opcode opcode;
   rv32i_reg rd;
} i_decode_opcode_t;


typedef struct packed {
```

```
    rv32i_word value;
    logic can_commit;
} rob_reg_data_t;


typedef struct packed {
    tag_t tag;
    logic valid;
    i_decode_opcode_t op;
    rob_reg_data_t reg_data;
    rv32i_word target_pc;
} rob_values_t;


typedef struct packed {
    logic valid;
    rv32i_word value;
    tag_t tag;
} rs_reg_t;


// for doing internal calculations in the alu reservation station
typedef struct packed { // when alu_rs needs to send data to the alu, it uses this struct
    logic valid;
    rs_reg_t rs1;
    rs_reg_t rs2;
    rs_reg_t res;
    alu_ops op;
    tag_t rob_idx;
    jmp_t jmp_type;
    rv32i_word curr_pc;
} alu_rs_t;


typedef struct packed { // when alu_rs needs to send data to the alu, it uses this struct
    logic valid;
    logic br;   // high if opcode is a branch, some non-branch opcodes also use cmp
    rs_reg_t rs1;
    rs_reg_t rs2;
    rs_reg_t res;
    rv32i_word pc;
    rv32i_word b_imm;
```

```
   rv32i_word result;
   branch_funct3_t op;
   tag_t rob_idx;
} cmp_rs_t;


typedef struct packed {
   rv32i_word vj_out;
   rv32i_word vk_out;
   tag_t qi_out;
   tag_t qj_out;
   tag_t qk_out;
} regfile_data_out_t;



typedef struct packed {
   rv32i_word value;
   tag_t tag;
   rv32i_word target_pc;
} cdb_entry_t;


typedef cdb_entry_t[`NUM_CDB_ENTRIES-1:0] cdb_t;
typedef rob_values_t[(`RO_BUFFER_ENTRIES)-1:0] rob_arr_t;


endpackage : structs
```