MP4 Progress Report
Tingkai Liu, Zhi Cen

1 Introduction

This project is mainly about the design of a basic RV32I processor and various improvements we have made on it. This project is an integration of instruction processing (CPU part) and memory access (cache part). To improve the performance of our processor, we enhance the processor from different aspects using the knowledge of computer architecture: splitting CPU into 5-stage and pipelining, solving hazards to avoid stalls, increasing the accuracy of branch prediction, using advanced cache to reduce miss penalty. In Part 2, we will give an overview to the whole design. In part 3, detailed design descriptions are provided to each part of our processor, including designs, testing and performance analysis. Part 4 is a conclusion of the whole project.

2 Project Overview

2.1 Goals and Constraints

The goal of the project is to make an efficient RV32I processor. Efficient is not only about running test codes in less time, but also less power consumption under the constraint of 100 MHz Fmax. To run test codes in less time, we tried to pipeline the CPU, increase memory hit rate and branch prediction accuracy. To reduce power consumption and Fmax constraint, we minimize the resource utilization and make decisions to strike a balance between different designs. For example, we use a 2-way L1 cache instead of 4-way to avoid long critical path when it is both a L1 and L2 cache miss.

2.2 Work Distribution

This project is completely designed and tested by two students from ECE411 as shown in Table 2.2.

Functionality	Notes	Work Distribution
a 5-stage pipeline CPU which can handle RV32I Instructions	1. no hazard detection 2. full RV32I Instructions except FENCE*, ECALL, EBREAK, and CSRR	Zhi: split mp2 CPU into 5 stages, unit test each stage and test the whole CPU with provided tb. Tingkai: design control_rom, assign control signals to each stage and connect all stages in datapath.
Hazard, static branch prediction	1. data hazard: forwarding 2. control hazard: static-not-taken branch prediction	Tingkai
L1 cache, arbiter	1. arbiter connected to I-Cache, D-cache and cacheline adaptor 2. no cache coherence	Zhi
L2 cache		Tingkai

4-way cache	1. only used in L2 cache	Zhi
2-level Branch predictor		Zhi, Tingkai
Prefetch	1. only used in instruction cache	Zhi

Table 2.2 Work Distribution

3 Design Description

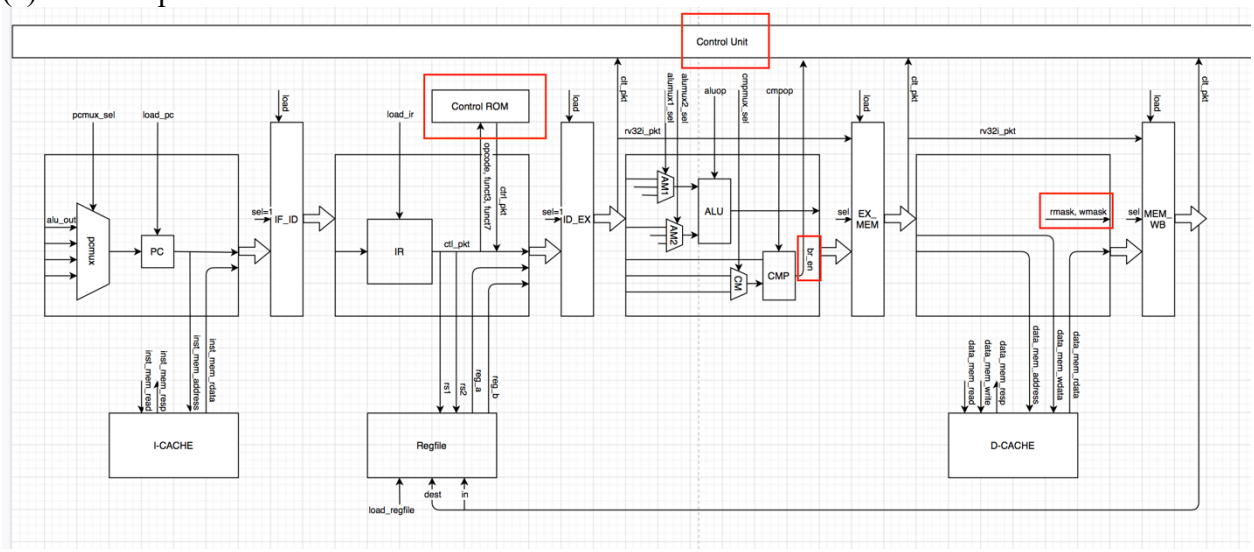
3.1 Overview

To achieve the goal described in 2.1, we divide the whole project into 3 milestones: (1) basic pipeline, (2) L1 cache and hazards, (3) advanced designs.

3.2 Milestones

3.2.1 Checkpoint 1: RV32I ISA and basic pipelining

(a) Datapath



(b) Instruction Analysis

The opcode of basic RV32I ISA instructions includes IMM, REG, LUI, MEM, AUIPC, BR and JAL(R). The Table 3.2.1(b) shows the use of each stage by different types of opcode.

Opcode	IF	ID	EX	MEM	WB	Note
IMM	1	1	1	0	1	
REG	1	1	1	0	1	
LUI	1	1	0	0	1	Place u_imm into register
MEM	1	1	1	1	1/0	Not need to write back for ST
AUIPC	1	1	1	0	1	Add something to PC and place the result into register.

BR	1	1	1	0	0	Conditional branch
JAL	1	1	1	0	1	Unconditional jump.
JALR	1	1	1	0	1	Unconditional jump.

Table 3.2.1(b) 5-stage utilization of different opcode

(c) Specific Design Description

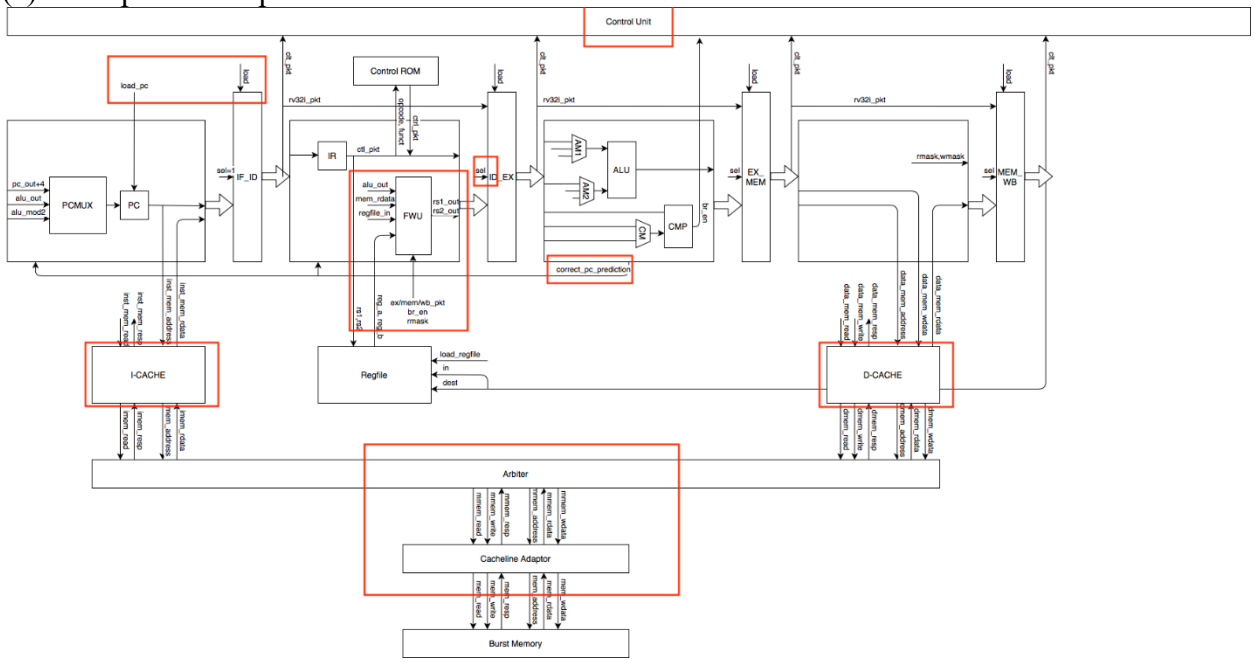
As shown in 3.2.1(a) and 3.2.1(b), we divide a basic processor into 5-stage pipelined processor: IF (fetching instruction), ID (decode), EX (execute), MEM (reading/writing to memory) and WB (write to regfile). To move the pipeline, we wrap the information of each instruction in a packet and pass it down through the pipeline. The packet contains four subsets: valid (indicate if it should be committed), CTRL packet (controls which stages/components are used), INST packet (decode by control rom) and DATA packet (saved data, e.g. pc). There are buffers between the stages to save the former information and signals (load and sel as shown in 3.2.1(a) datapath) to determine if it should be updated or dropped.

(d) Testing

Testing Type	Files
Unit test	if_unit_test.sv, id_unit_test.sv, ex_unit_test.sv, wb_unit_test.sv
Using source code	mp4_cp1.s

3.2.2 Checkpoint 2: L1 caches, hazards and static branch prediction

(a) Update Datapath



(b) Specific Design Description

(i) Arbiter

Arbiter is connected to data cache and instruction cache. It determines which cache is interacted with memory. Below table shows the state machine design of arbiter.

State	Description	Next	Output
WAIT	Wait until a access request happens from either I-cache or D-cache	If I-cache requests, to I_MEM. If D-cache requests, to D_MEM. I-cache request is check first (Prioritized)	0
I_MEM	Access memory according to the signals from I-cache	If mem_resp, to I_DONE	Pass signals of I-cache to physical memory
I_DONE	Finish I-cache request, and check whether need to do D-cache request	To WAIT	inst_mem_resp = 1
D_MEM	Access memory according to the signals from D-cache	If mem_resp, to D_DONE	Pass signals of D-cache to physical memory
I_DONE	Finish D-cache request, and check whether need to do I-cache request	To WAIT	data_mem_resp = 1

Table 3.2.2(1) Arbiter state machine design

(ii) Forwarding

At ID stage, when reading the register values, the latest version of register value may not be write-back. As a result, we design the forwarding to be getting the latest value of register at ID stage if necessary, which means instead of connecting forwarded value to EX, MEM, WB, we only forward the value to ID stage and store the forward value in the packet and pass it along the pipeline.

Note that for register dependency happens when the previous instruction is a load instruction, the pipeline need to stall since the next instruction needs the value at EX at the same time MEM is getting the value. The way we solve that is when such cases is detected, the IF, IF_ID, and ID are stalled by keeping the load to be 0, and an invalid instruction is inserted in ID_EX and keep ID_EX, EX, EX_MEM, MEM, MEM_WB, WB moving.

(iii) Control Hazard

Opcode	PC
BRANCH	PC <- PC + (br_en? b_imm : 4)
JAL	PC <- PC + j_imm

JALR	PC <- {(rs1_out + i_imm) [31:1], 1'b0}
Other opcodes	PC <- PC + 4

Table 3.2.2(2) PC update table

Mark the mis-fetched instruction to be invalid and stop them from updating memory and registers. Since we are using static no-taken branch prediction, when we see a br/jal/jalr at EX stage calculated the correct branch target which is not pc+4, we will mark the instruction in IF_ID and ID_EX to be invalid since they are mis-fetched. After those instructions are marked as invalid, they won't be able to write to memory or registers.

(c) Testing

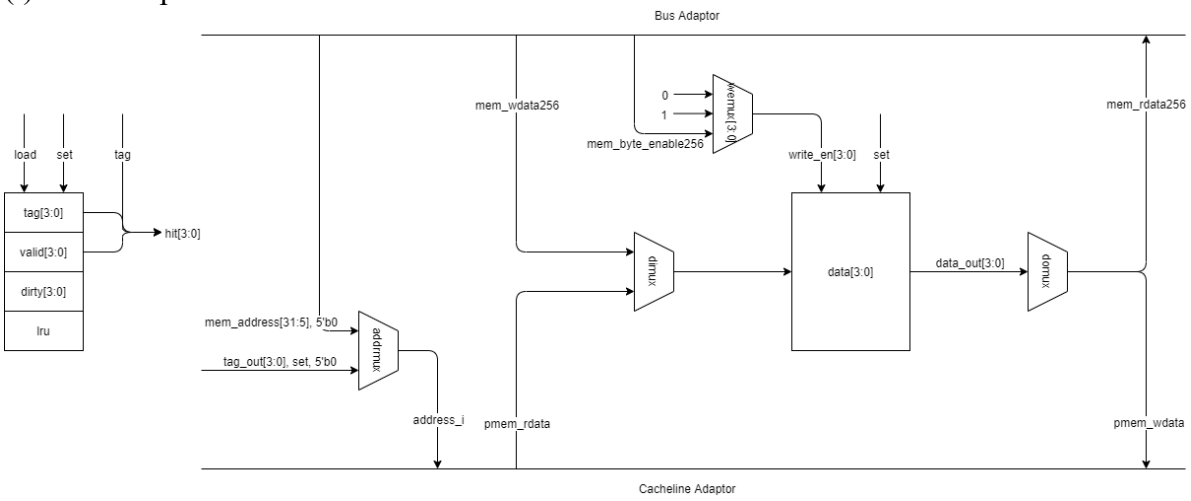
Testing Type	Files
Unit test	cache_unit_test.sv, arbiter_unit_test.sv, cache_top_unit_test.sv
Using source code	mp4_cp2.s, factorial.s, mp4_cp3.s

3.3 Advanced Design Options [11]

3.3.1 4-way cache [2]

(a) Design

(i) Datapath



(ii) Description

After successfully implemented the 2-way cache, by analyzing the performance, we find that it takes (10 + burst delay) cycles for a cache miss without a write back and (19 + 2*burst delay) for a cache miss with a write back. Thus, it is a great penalty for miss and we can improve the performance by increasing the hit rate by changing the 2-way cache to 4-way cache.

The datapath of the 4-way cache is similar to the 2-way cache, except for the number of ways. As for the control part, we use the pseudo LRU algorithm to determine which way we should replace for a miss. Below are the LRU update table (Table 3.3.1(a)) and LRU replacement table (Table 3.3.1(b)).

Way hit	L2	L1	L0
1	x	0	0
2	x	1	0

3	0	x	1
4	1	x	1

Table 3.3.1(a) LRU update table

Way to replace	L2	L1	L0
1	x	1	1
2	x	0	1
3	1	x	0
4	0	x	0

Table 3.3.1(b) LRU replacement table

(b) Testing

Testing Type	Files
Unit test	cache_unit_test.sv
Using source code	mp4_cp3.s, comp1.s, comp2.s, comp3.s

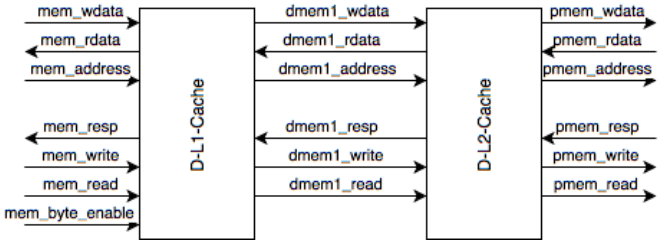
(c) Performance Analysis

4-way cache generally reduce the miss rate. See 3.3.5 for detailed data.

3.3.2 L2 cache [2]

(a) Design

(i) Datapath



(ii) Description

After finishing the 4-way L1 cache, we find that using 4-way L1 cache results in very long critical path. The main reason is that we use combinational logic in array reading in order to response a memory hit in one cycle.

To fix this Fmax problem and further accelerate our compiler, we decide to add a L2 cache. The design of a L2 cache is similar to L1 cache, the only difference is that we don't need a bus adaptor to covert 32-bit data to a 256-bit cacheline.

(b) Testing

Testing Type	Files
Unit test	cache_unit_test.sv
Using source code	mp4_cp3.s, comp1.s, comp2.s, comp3.s

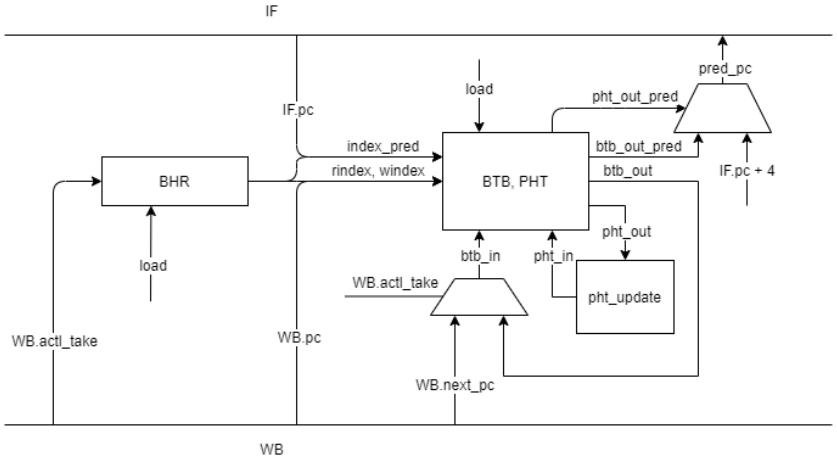
(c) Performance Analysis

L2 cache increases memory access performance in some category memory access pattern. Specifically, L2 cache will boost the performance (significantly decrease the L1 miss penalty) when more than 2 cache lines are repeatedly accessed, in which case some cache lines are evicted from L1 cache but needed later. See 3.3.5 for detail data and analysis.

3.3.3 Global 2-level branch predictor [3]

(a) Design

(i) Datapath



(ii) Description

A wrong branch prediction will cause stalls or waste cycles. Therefore, if we can predict the next pc with higher accuracy, we can save cycles and promote the performance.

A local branch history table use (PC xor BHR) as index to look up tables (Branch Target Buffer, a look up table for predicted PC, and PHT, a 2-bit state machine for each entry of BTB), to produce predicted PC and whether it should be taken. Since the PC itself doesn't contain much information about the instruction for the predictor, we added a fast decode in IF stage to correct the predicted PC to be PC+4 whenever the instruction is not a branch. After the branch instruction finished its execution, its information, including what is the exact branch target and whether the branch is taken, is used to update the tables in the branch predictor for later usage.

(b) Testing

Testing Type	Files
Unit test	bp_unit_test.sv
Using source code	mp4_cp3.s, comp1.s, comp2.s, comp3.s

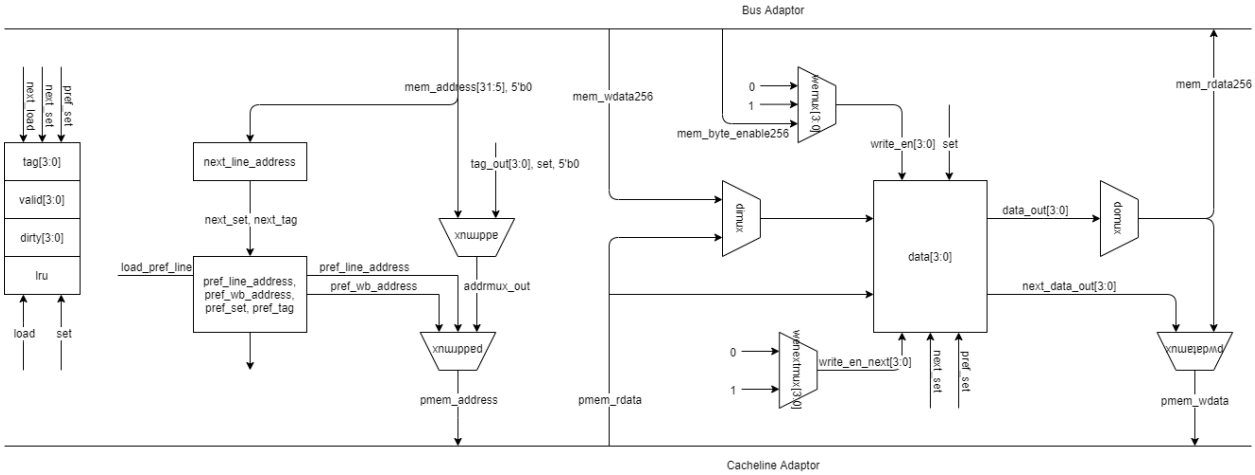
(c) Performance Analysis

Generally, our branch predictor reached the prediction accuracy of about 80%. See 3.3.5 for detailed data.

3.3.4 Basic hardware prefetch [4]

(a) Design

(i) Datapath



(ii) Description

We only use prefetch in instruction cache, because most instructions are sequence accessed. If cacheline k is accessed and it is a hit, but cacheline k+1 is a miss, then the cache will request a read of cacheline k+1.Below table shows the state transition.

State	Description	Next
HIT CHECK	Serves curr_line hit and deals with misses.	if curr_line miss, to WRITE BACK/READ BACK. if curr_line hit but next_line miss, to PREFETCH WB/RB. if curr_line hit and next_line hit, to HIT CHECK.
WRITE BACK	Writes the dirty cacheline back to memory.	if (pmem_resp), to READ BACK
READ BACK	Reads curr_line back to cache.	if (pmem_resp), to HIT CHECK
PREFETCH WB	Writes the dirty cacheline back to memory. Serves curr_line hit.	if (pmem_resp), to PREFETCH RB
PREFETCH RB	Reads next_line back to cache. Serves curr_line hit.	if (pmem_resp), to HIT CHECK

Table 3.3.4 State machine of prefetch

(b) Testing

Testing Type	Files

Using source code	mp4_cp3.s

(c) Performance Analysis

Prefetch can only pass mp4_cp3.s. Since most instructions are cached in L2 and L1 I-cache, it does not accelerate the whole design distinctly.

3.3.5 General Performance Analysis

We used comp1.s, comp2_i.s, and comp3.s as benchmarks to evaluate our design performance comparing the "baseline" provided, as the table shows.

	Comp1	Comp2	Comp3
Order	44437	87863	47243
Data memory access	2813	4399	11134
Runtime (ns)	495055 (1.5x speedup)	1308735 (3.5x speedup)	926585 (4x speedup)
Power (mW)	830.57	831.56	772.76
I-Cache L1 miss	57/44437 = 0.12%	6524/87863 = 7.5%	6246/47243 = 13%
I-Cache L2 miss	27/57 = 47%	171/6524 = 2.6%	130/6246 = 2.1%
D-Cache L1 miss	14/2813 = 0.50%	82/4399 = 1.9%	567/11134 = 5.1%
D-Cache L2 miss	7/14 = 50%	27/82 = 33%	283/567 = 50%
Branch prediction accuracy	10683/12370=86%	27506/34620 = 79%	3665/4391 = 83%

Table 3.3.5 Design Performance

For comp1 code, since the memory access pattern is not that complicated, the L1 cache miss rate is low and the speedup mainly comes from branch prediction. For comp2 and comp3 code, L2 cache has helped a lot, as they access more than 2 cache lines memory frequently. One of the surprising discoveries is the miss rate of L1 cache is pretty high. I think that's because there are long loops in a code that the code goes cross several cache lines repeatedly. In this case, the L2 I Cache increases the performance quite a lot.

4 Additional Observations

We started to think about implementing Out-of-Order execution at cp1, that is also a reason why we used "packet" between stages. It would be very convenient if we broadcast the whole packet on common data bus. However, due to time limitation and the difficulty to implement dynamic register dependency in hardware, we didn't manage it.

Another possibility is the advance features of power. In lecture we talked about powers a lot, and it is indeed a real challenge. I was surprised that in the advance feature list power is not mentioned at all. I think we can at least implement the clock gating and use the simulation of Quartus to evaluate the performance. Of course, there will be additional complexity on understanding what is going on with the power simulation, but I do suggest putting it into the list for future students.

5 Conclusion

In conclusion, this project successfully designs a 5-stage pipelined RV32I processor and improves its performance by using 4-way L2 cache, branch predictor and prefetch method.