ECE 411: Final Report

November 29, 2021

Qingyi Wang / qwang83
Mingyuan Han / mh24
Shitao Liu / sl53
TA: Youjie Li

Contents

1 Introduction

After the invention of Intel 4004, the central processing unit (CPU) has become the most important component of a computer, as it is not only the brain of a computer but also the main unit doing most of the calculation work. In a few decades, generations of engineers explored and innovated various genius architectures and algorithms to speed up the CPU and balance between performance and power consumption. In this MP, *Outel* (mentioned as "our team" in the following content) implemented a pipelined $RV32I$ processor with several advanced features, including a uniquely designed parameterised cache system, a hardware prefetcher, a branch predictor module, and a multiplier/divider module supporting the M-Extension instruction set.
The project includes a CPU with embedded branch predictor and M-extension supporter module, connected with a parameterised L-1 cache system, an arbiter unit with hardware prefetcher, a parameterised L-2 cache system, and an eviction write buffer. The buffer is connected to the cacheline adaptor, which provides a standardized interface to the lower memory.

2 Overview of the Project

2.1 Overall Structure

In our five-stage pipelined design, our CPU has five stages: IF, ID, EXE, MEM, and WB. The instructions and data from the previous stage is temporarily stored in latches, and the pipeline stalls when it waits for data from the cache system or inserts a bubble in the MEM stage when it has data hazard that can not be solved by data forwarding.

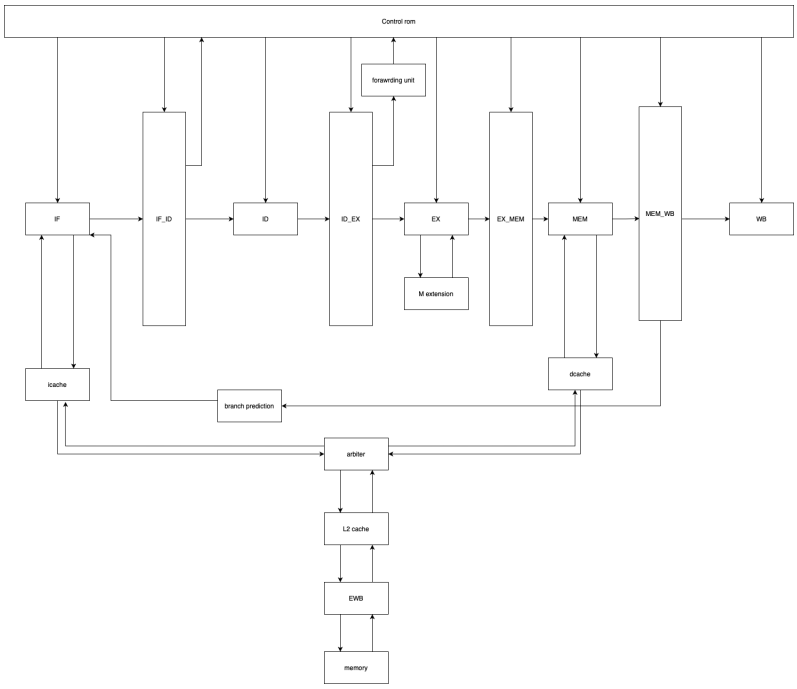


Figure 1: Overall Structure of the Project

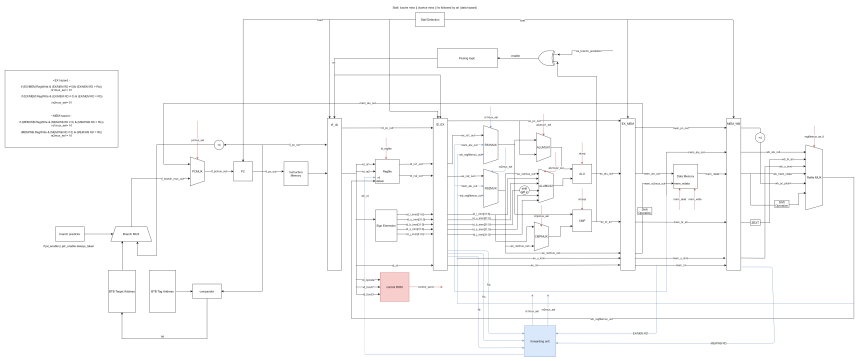


Figure 2: Circuit Layout

2.2 ISA Support

In addition to the standard RV31I base instruction set, our CPU also supports the RV32M standard extension [4] with the implemented multiplier/divider module using the modified Booth-Wallace tree algorithm.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 3: Instruction Set Support List

3 Written Description

3.1 Features Overview

We finished the implementation of our final project within four checkpoints. For each checkpoint, each team member was responsible for designing, implementing, and integrating his/her part into the basic five-stage pipeline backbone. In the final deliverable, we have a L-1 cache of which the number of index is parameterised, a L-2 cache system with the numbers of ways and index adjustable, a tournament branch predictor of which we can customise the number of index and the number of last branches used, a 3-cycle multiplier using Wallace Tree and modified Booth recoding, a divider using classic long-division, a hardware prefetcher, and an eviction write buffer, both implemented as finite-state machines.

3.2 Forwarding in Pipelining

The forwarding unit consists of two MUXs and one controlling unit. The first MUX, which we named as $RS1MUX$, takes in the value of $RS1$ from the ID stage, the output of ALU from MEM stage, and $wb\_regfilemux\_out$ from the $WRITEBACK$ stage. The second MUX, which we named as $RS2MUX$, takes in the value of $RS2$ from the ID stage, the output of ALU from MEM stage, and $wb\_regfilemux\_out$ from the $WRITEBACK$ stage. The two select signals, $rs1mux\_sel$ and $rs2mux\_sel$, are generated by the control unit and used to select the outputs of two MUXs, as introduced in our lecture [5]. The logic of the control unit is shown below. Because we are not able to handle both EX and MEM hazards at the same time, we decided to give EX hazards a higher priority.

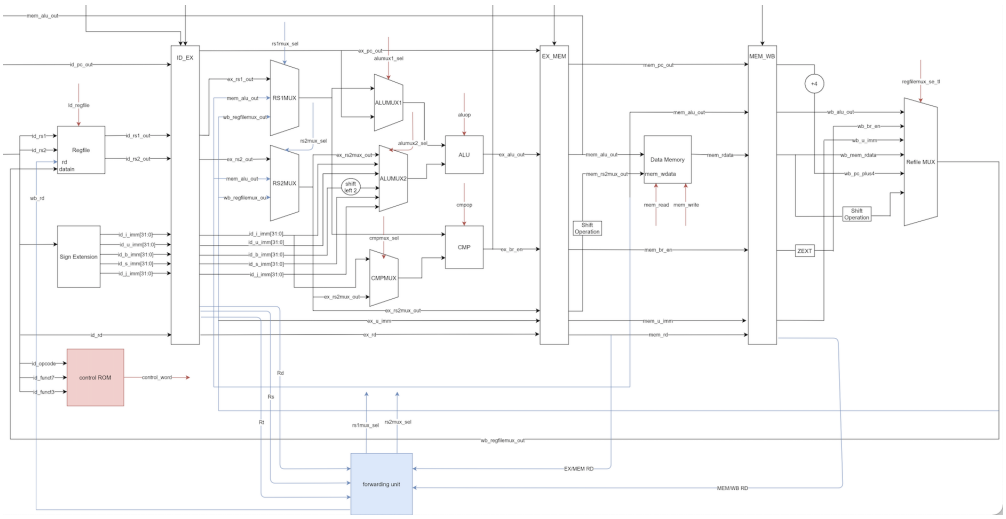


Figure 4: Forwarding Unit Layout

3.3 Milestones

3.3.1 Checkpoint 1

We split Checkpoint 1 of our final MP4 into four subtasks: control_rom, datapath, interconnection, and testing. Specifically, each of us was assigned at least one task: Qingyi was responsible for migrating the control module from the previous MP2 to our final project and making it compatible with our pipeline design; she was also responsible for writing the testbench files. Shitao implemented the skeleton of the pipelined datapath and other necessary modules. Finally, Mingyuan finished the interconnection between modules on the CPU level and solved signal errors.
For this checkpoint, we finished a basic pipelined CPU connecting to the provided "magic memory"

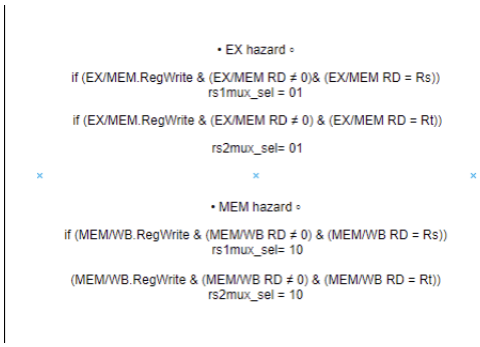


Figure 5: Forwarding Unit Logic

and it could execute most of the RV32I instructions specified in the documentation without handeling the data and control hazards. The internal latches in our datapath pass the control words, calculation results, and memory data between stages to make the pipelining feasible.
We tested our implementation with the provided assembly files. We implemented the RVFI monitor on our own to compare the spec data with our results. The RVFI monitor is committed every time we have valid data passed through the MEM_WB register and we connect the signals at the WB stage with the RVFI monitor to do the comparison. Therefore, all signals necessary for debugging such as instruction, rs1_addr and rs2_addr need to be carried by the internal latches to the last stage as well. To make sure that the pipelining design is capable of handling all instructions, we also checked both the data in the register files after execution and the waveform clock-by-clock to verify the working status of each internal signal.

3.3.2 Checkpoint 2

We generally split the task for Checkpoint 2 into three parts: cache and arbiter (memory interaction part), branch predictor, and forwarding logic part. Specifically, everyone is responsible of implementing his/her design as described in the last proposed roadmap: Shitao designed, implemented, and tested the robustness of the memory interaction part, including a D-Cache and a I-Cache and the arbiter. Qingyi is responsible for the static branch predictor and the testing of the whole design after completion. And, finally, Mingyuan is responsible for design, implement and testing of the forwarding logic part.
We relied on both the provided test cases and the autograder to examine our design. To test the L-1 caches and the arbiter, Shitao replaced the magic memory for CP1 with the ordinary param memory, connected the L-1 caches and the arbiter to the CPU, and reran the CP1 test to check if the CPU works properly. Mingyuan manually creates codes with data hazard to test if the forwarding par is working as designed. Qingyi finishes the monitor files in hvl folder and checks the waveform to ensure the correctness of her design.

3.3.3 Checkpoint 3

For this checkpoint, as usual, we split the main task into three different parts, and each member was responsible for implementing and testing his/her designed as planned in our last roadmap. Specifically, Shitao implemented the whole cache system and tested the overall performance of the whole design; Qingyi designed and implemented the local branch history predictor and global branch history predictor, and based on them developed a tournament branch predictor. Afterwards, Qingyi also developed the m-extension for our CPU using the simple add-shift algorithm. Mingyuan finished the hardware prefetching part. We also kept an eye on the timing test along the way implementing our design to make sure our design can run at 100MHz without having a too long critical path.

3.4 Advanced Features

3.4.1 Parameterised Cache System

To make our design more portable and customisable, we make both L-1 and L-2 parameterised. Specifically, the user can set the size of both L-1 and L-2 cache by changing the number of set and the number of ways in the toplevel. Our L-2 cache supports both 4-way and 8-way settings using the tree-structure pesudo-LRU algorithm to decide which line should be evicted when there is a cache miss. Also, we used masked arraies to simplify bit-wise arithmetics.

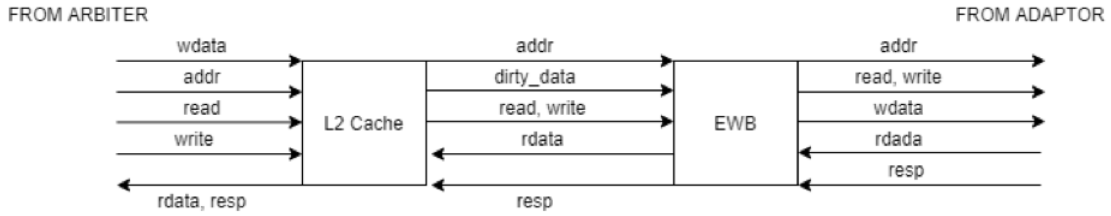


Figure 6: API of L-2 Cache

While testing our design, we noticed that a larger L-1 cache could significantly boost the overall performance. Thus, we also designed a optimised L-1 system with parameterisation. Specifically, to ensure that our L-1 cache is one-cycle hit, we need to balance between performance and power, as the power consumption of a cache with array units increases non-linearly. At last, we designed an asymmetric L-1 cache with a relatively larger I-Cache and a smaller D-Cache, which not only maximally utilised available design units on our FPGA board but also put the overall dynamic power dissipation in a reasonable range.

3.4.2 Eviction Write Buffer

The Eviction Write Buffer can exchange the order of Read and Write in the WRITE_BACK stage, by storing the dirty data along with its address in the inside register of EWB and send a read request to the physical memory first, then receive the read data and pass it back to the upper-level cache, and lastly write the dirty data back to the physical memory. Our EWB is implemented as a finite-state machine.

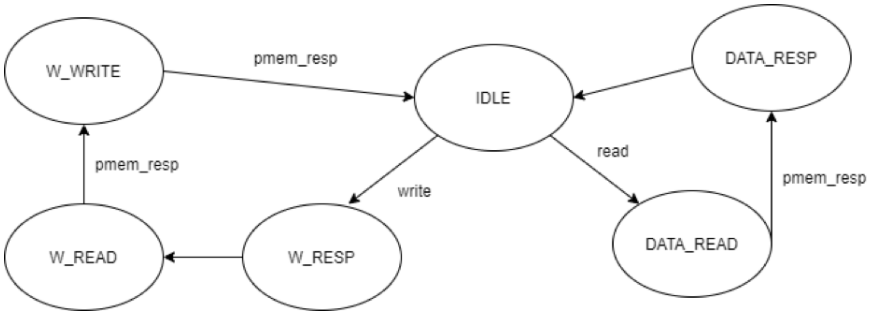


Figure 7: Finite-State Machine of EWB

3.4.3 Branch Predictor

We designed a local branch predictor, a global branch predictor, and a tournament branch predictor for our processor. The local branch predictor consists of a 2-bit branch history table and a branch target buffer. The BTB can be indexed by some bits in the PC and the rest of the PC would be saved in the tag array. The number of the indexes used is parameterised.

In the IF Stage, we keep comparing the current PC with the PC stored in the tag array. Every time we receive a BTB hit and the predictor predicts taken, we would fetch the address stored in the BTB instead of the PC+4. The branch update logic is implemented in the MEM stage to avoid timing issues. If there is a mismatch between the predicted branch direction and the actual branch direction, or between the predicted next PC and the actual next PC, we would flush the stage latches and update both the branch history table and BTB.
The global branch predictor uses a 2-level branch history table. It saves the results of the past few branches into a branch history register and uses the BHR to choose among several pattern history tables. Depending on the prediction given by the pattern history table, we would decide whether we will fetch the PC+4 or fetch the PC stored in the BTB. The size of the BTB and the number of the last branches are both parameterised. Based on the local branch predictor and the global branch predictor, we build a tournament branch predictor and use a 4 state FSM to choose between which of the two is the best predictor to use for a branch.

3.4.4 M-Extension Support

Our processor supports the M-extension. For the integer divider, we use the classic long division scheme. For the integer multiplier, we first used the simple add-shift algorithm but realised that it takes a maximum of 32 cycles to finish the multiplication. Therefore, we re-designed the multiplier using the Modified Booth-Wallace tree to reduce the number of cycles needed. We use the radix 4 modified Booth recoding to halve the number of partial products to be added. This reduces both the amount of area needed for the hardware and the time needed for the execution. The Radix-4 modified booth recoding algorithm is shown below:

Y_{i-1}	Y_i	Y_{i+1}	Recoded Operation on Multiplicand X
0	0	0	0X
0	0	1	+X
0	1	0	+X
0	1	1	+2X
1	0	0	-2X
1	0	1	-X
1	1	0	-X
1	1	1	0X

Figure 8: Radix-4 Modified Booth Recoding [1]

Multiplier consumes most of its power and delay when adding generated partial products. The Wallace tree method is a very efficient way to do partial product addition in high-speed circuit designs because it needs fewer addition stages, thus improving the speed. Here, we use the 4:2 Compressors to reduce the partial product layers because it lowers the latency of accumulation stage. The block diagram of the 4:2 compressor is shown below. It takes 5 inputs and generates 3 outputs, which is more compact than the 3:2 compressor. The Wallace tree structure using 4:2 compressors is shown below. The final addition is done using a carry lookahead adder because it has a smaller delay compared to Ripple Carry Adder. By using this Modified Booth-Wallace tree Multiplier, we can reduce the number of cycles from a maximum 32 cycles to 3 cycles without having timing issues.

3.4.5 Hardware Prefetcher

We have implemented the basic hardware prefetch, which is a one block lookahead prefetcher. The mechanism of prefetching is that every time we encounter a cache misses, we not only fetch the current line but also the next line of data. We implemented this feature within our arbiter unit by adding an extra state after the fetching is done, which is the state diagram labeled as $I\text{-}Cache\ prefetch$ and

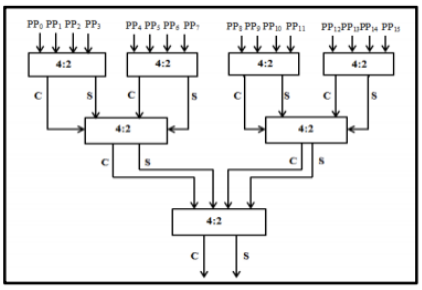


Figure 9: Wallace Tree Structure using 4:2 Compressors [1]

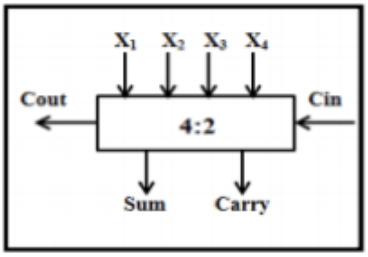


Figure 10: 4:2 Compressor [1]

D-Cache prefetch. Even though we implement it for both instruction cache and data cache, because of its mechanism, it works better for instruction cache comparing to the data cache.

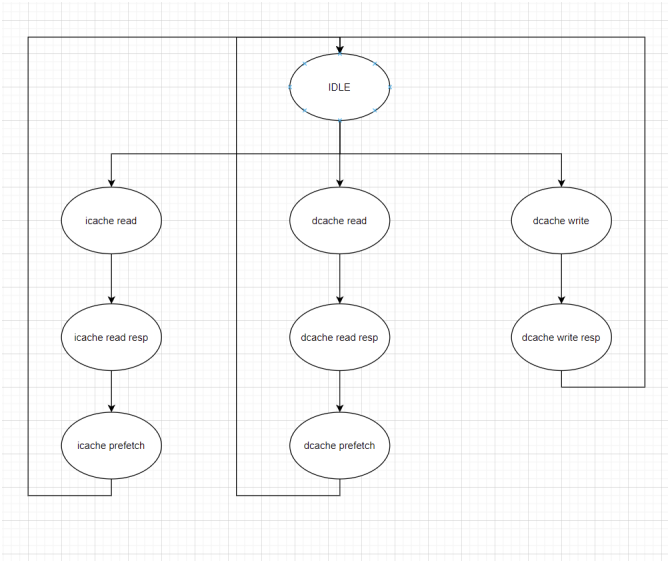


Figure 11: Finite-State Machine of Prefetcher

4 Testing and Performance Analysis

4.1 Testing Strategy

We mainly relied on the provided testcases to check if our design works properly. We hooked the RVFI monitor at the very beginning, which makes the debugging process much easier. For the provided testcodes, we checked whether we had any error messages from the monitor. For the testcodes we wrote on our own for the branch predictor and data forwarding logic, we compared the data stored in the register file after the program halts with the results we calculated manually. We also checked the waveform to ensure that the logic modules, such as the PLRU tree for the L-2 cache system, work as expected.

4.2 Performance Analysis

We also added a performace counter to each advanced feature implemented to check how well they worked in our design.

4.2.1 Cache Performance

For the cache system, we recorded the total number of memory calls, including *D-Cache read*, *D-Cache write*, and *I-Cache read*, from the CPU, and the total number of *resp* signals sent by the physical memory. The performance of our cache system is measured by its ability to increase the cache-hit ratio, or to decrease the ratio between the aforementioned values.

Cache Performance Analysis

Testcase: comp3	# of mem calls	# of pmem access	Cache hit ratio
4-Way L2	53146	414	0.992
Without L2	53146	802	0.985
Without Optimised L1 (CP2 version)	53146	8730	0.836

Figure 12: Performance of Cache System

From the figure above, we notice that both an enlarged L-1 cache and a L-2 cache can significantly boost the overall performace of our CPU when running the *comp*3.*s* testcase. With our optimised L-1 cache and a 4-way L-2 cache, the cache-hit ratio can achieve 99%, which means that making the two caches even larger will not bring too much advantages. Rather, solely making the L-1 cache larger by adding its associativity and using BRAM to decrease power dissipation could no doubt further exploit performace, since accessing L-1 is always faster than accessing lower-level caches.

4.2.2 Branch Predictor Performance

This table shows the performance of each branch predictor, given by our performance counter. We counted the number of branches/jumps and the number of correct predictions and calculate the prediction accuracy rate for each testcode. As the table shows, by implementing the local branch predictor, the global branch predictors and the tournament branch predictor, we are able to boost the prediction accuracy rate from 30% in average to 85% in average.
The global branch predictor performs a little worse than the local branch predictor and tournament branch predictor because of the weak correlation in the testcodes. Therefore, in the final version for the competition, we decided to use the local branch predictor as it reaches a good balance between performance and power.

	Static-Not-Taken		Local (index = 10)		Global (index = 10, # last branches = 2)		Tournament (index = 10, # last branches = 2)	
	# of Correct Prediction	Prediction Accuracy Rate	# of Correct Prediction	Prediction Accuracy Rate	# of Correct Prediction	Prediction Accuracy Rate	# of Correct Prediction	Prediction Accuracy Rate
Comp1.s (12371 branches/jmps)	5853	47.31%	10703	86.52%	9986	80.72%	10549	85.27%
Comp2_i.s (34621 branches/jmps)	11756	33.96%	27866	80.49%	28152	81.31%	27959	80.76%
Comp3.s (4392 branches/jmps)	614	13.98%	3874	88.21%	3878	88.30%	3882	88.39%

Figure 13: Performance of Branch Predictors

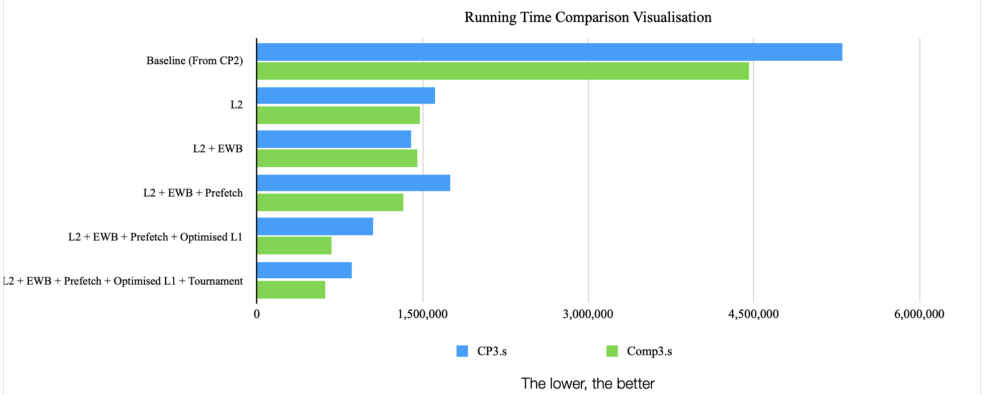
4.2.3 Hardware Prefetcher Performance

This table shows the performance of the basic hardware prefetcher running *cp*3.*s*, which could be considered as the worst-case performance. We recorded the number of misses that happened in the cache system as an indication of whether prefetching is effective. A decrease in the number of misses indicates that prefetching successfully brings the target data to the cache system. We can see that there is a decrease in miss rate in *I-Cache*, which highly depends on the test case. The more branch instruction, the less effective prefetcher will be. However, the miss rate of *D-Cache* did not change, which makes sense since data access is not sequential.

	number of miss in icache	number of miss in dcache
without prefetch	6484	1413
with prefetch	6413	1413

Figure 14: Performance of Hardware Prefetcher

4.2.4 Overall Performance



Running time

Running Time Unit: ns	CP3.s	Comp3.s	Speed Up from Baseline (CP3.s)	Speed Up from Baseline (Comp3.s)
Baseline (From CP2)	5,300,135	4,458,455	1.00	1.00
L2	1,610,615	1,474,505	3.29	3.02
L2 + EWB	1,398,475	1,451,745	3.79	3.07
L2 + EWB + Prefetch	1,751,045	1,324,575	3.03	3.37
L2 + EWB + Prefetch + Optimised L1	1,058,305	676,295	5.00	6.59
L2 + EWB + Prefetch + Optimised L1 + Tournament	864,805	617,365	6.13	7.22

Figure 15: Overall Performance

As we shall see, the advanced features we designed significantly boost the overall performace of our CPU compared with the baseline. Among all features, the cache system contributes the most to the overall performance, leading the branch predictor and the hardware prefetcher.
We also notice that the prefetcher affects the overall performace negatively when running some specific programmes. We believe that the performace of the hardware prefetcher is highly influenced by each specific testcase. Since one block lookahead prefetcher is going prefetch every time there is a cache miss, it will load a lot of data into the cache. Some of them are useful, but most of them are not, which will lead to cache pollution. Also, the number of branch instructions is also going to lead to a negative effect on hardware prefetcher. Since we are only capable of prefetching the next cacheline, we cannot prefetch the previous data. The last possible reason is that prefetching is blocking. A blocking mechanism is going to increase the running time.

4.2.5 Competition Performance

Our Final version can run all three competition testcases successfully and smoothly, and the power dissipation is measured with the *.vcd* file. Our CPU can run at $103.68MHz$, with no timing issues. While the power dissipation varied in a small range each time we run the analyser, we used the mean value after multiple measurements to calculate the final score.

Unit: Time (ns); Power (mW)	Running Time	Power	Score	Fmax	Utilised Regs
Comp1	506275	607	7.877E-11	103.68MHz	21937
Comp2_m	511700	680	9.111E-11	103.68MHz	21937
Comp3	608255	611.9	1.377E-10	103.68MHz	21937
Geometric Mean			**9.960E-11**		

Figure 16: Competition Score

5 Conclusion

This project is challenging but interesting. Throughout the whole month, our teammates have deployed what we've learned in the lectures to a complicated project and explored other aspects of CPU design. Specifically, we gained a deeper understanding of how the cache system, the pipelining system, the branch prediction module and the prefetcher work comprehensively in a CPU with other units. We also learned how to implement advanced arithmetic algorithms other than add-shift.
However, we also understand that there's still a lot of room for improvements. For example, we didn't pay enough attention to the power dissipation, and it finally became a major factor impeding us from further speeding up our CPU. Also, we made some compromises in the design of our branch prediction module to avoid the critical path being too long. We learned that translating theories into real circuit is much more difficult than we expected, and a good testing strategy is critical to a successful project. Overall, we believe that we have demonstrated a good understanding of the course materials and practised our ability to develop a large project through teamwork. Every team member has contributed a lot to the final deliverable and our cooperation was very smooth and brainstorm active. Throughout the whole process, our mentor TA, Youjie Li, helped us a lot with the details of designing each component and performance analysis. This project laid a solid foundation for our future career in computer architecture area.

References

[1] Design and Analysis of Various 32-bit Multipliers

[2] CMPEN411 from PSU: Multiplier Design

[3] EE486 from Stanford: Advanced Computer Arithmetic Design

[4] The RISC-V Instruction Set Manual

[5] ECE411 Lecture5: Pipelining