EECS470 Final Project Report

Group 17: Yinwei Dai, Tianchi Zhang, Xiaoxue Zhong, Ramchandra Apte

April 20, 2022

1 Introduction

The project has built a P6-structure pipeline to handle RISC-V instructions. Some advanced features such as superscalar execution are added to enhance processor performance. This report covers the design details of our project, the final implementation and testing results, and evaluation and analysis of performance for different optional features.

2 Design Overview

We implemented a 2-way superscalar P6 pipeline in this project. The top level design of the entire architecture of our processor is showed in the diagram below. It contains all the basic components for a P6 processor to support out-of-order implementation: a 32-entry Map Table, a 32-entry Reorder Buffer, a 20-entry Reservation Station, a 2-entry Common Data Bus and a Load Store Queue (4-entry load queue and 12-entry store queue). To handle memory latency, we also implemented a I-cache and a D-cache. And a dynamic branch predictor is added for branch and address predictions.

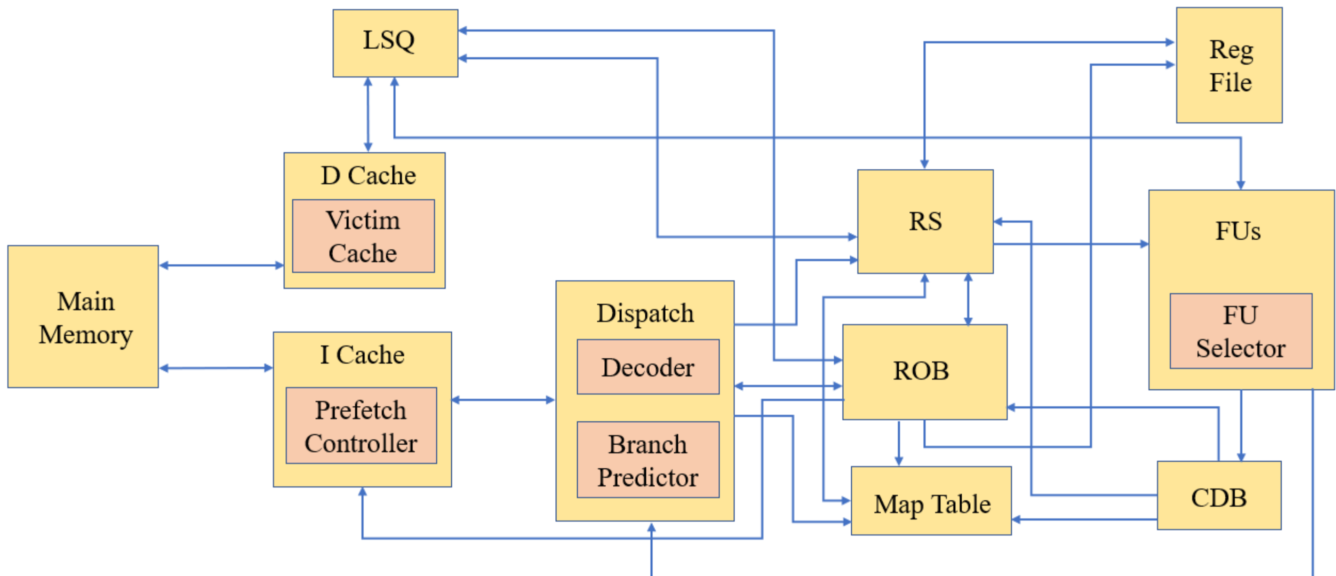

Figure 1: High-level Processor Design Overview.

Besides the basic modules. below are all the optional and advanced features our group added to enhance the processor performance. The details and evaluation of these features would be discussed in the following sections.

1. 2-way Superscalar pipeline.
2. 2-way associative I-cache with prefetching and early branch resolution.
3. 2-way associative dual-ported non-blocking D-cache with a 16-byte victim cache.
4. Dynamic branch predictor with 2-bit saturating counter and branch target buffer.
5. Return address stack to handle *jalr* instructions.
6. A GUI debugger to visualize pipeline status (ROB, RS, caches *etc*) in every cycle.

The following subsections covers the design details about several important modules/components in our pipeline, and the evaluation of their performance would be in the analysis section.

2.1 Load Store Queue (LSQ)

The LSQ have the same number of port as the number of load-store unit and they work in parallel with no blocking against each other. We have four load-queue entries work in out-of-order mode and sixteen store-queue entries work in in-order mode. Both of them are lined to D-cache and allow loading and storing at the same time. A load can go to the memory when there is no matching store that is older and having same address and size or not having address or value resolved. A store is eligible to go to the memory when been retired.

2.2 I-cache

The I-cache is a 2-way set associative cache with 16 lines, each storing a 64-bit value (containing 2 instructions). For each instruction, the I-cache fetches it and the following 6 lines, to reduce the number of I-cache misses. This works well when the sequence of instructions executed is linear. Whenever ROB detects an early-branch wrong prediction, the I-cache will give up the current fetching and start prefetching the branch target.

2.3 D-cache

The D-cache is a write-through 2-way set-associative dual-ported non-blocking cache with 16 lines, each storing two 64-bit values. It works in write-though mode and it can respond to read and write instruction in the same time. We chose a write-through design to keep the D-cache simple considering variable length writes. It also has a 16-byte victim cache, and we use LRU policy for eviction. There are eight entries to record the response from the memory. If there is new data loaded from the memory and the target entry is filled, the evicted entry will be pushed into the victim cache.

2.4 Multiple Function Units

We have split the function units into 4 types: **the ALU**, **the load-store unit**, **the branch execution unit** and **the multiplier**. The ALU completes all the arithmetic calculations except multiplication. The branch execution unit calculates whether the branch condition is satisfied

as well as the corresponding target address. The load-store execution unit is used to calculate the load/store address for LSQ. All the three function units require one cycle to complete. The multiplier would handle all the signed and unsigned multiplications. And it would take four cycles to complete. Instead of connecting four *mult_stage* serially together, we updated the multiplicand and multiplier every cycle to reduce the circuit area for synthesis.

For the 20-entry Reservation Station, there are correspondingly **8 ALUs**, **4 load-store execution units**, **4 multipliers** and **4 branch calculators**. Using a priority selector, the function selector decides which of FU result to send to the CDB.

2.5 Re-Order Buffer (ROB)

The Re-order buffer is one of the most important modules in our pipeline. It stores the destination register index and when an instruction is dispatched and allocates an ROB entry and updates its value when valid. To support the 2-way superscalar implementation, it contains two head pointers to the top two valid ROB entries as well as two tail pointers pointing to the last two valid entries in the ROB. For branch prediction recovery, each ROB entry would also store the predicted branch result and corresponding target address when a branch instruction is dispatched. When the branch function unit get the correct branch result, it would pass to ROB and compare with the data stored in ROB. If the prediction stored in ROB does not match the branch execution unit result, ROB would send the correct target address to I-cache for early prefetching. When the mispredicted instruction retires, it would trigger the squash signal to clear the map table, reservation station and the return address stack. We tested with different ROB sizes and fixed it to 32.

2.6 Branch Prediction

In terms of the branch prediction mechanism, our group has implemented a 32-entry directed mapped **Branch Target Buffer**, a 32-entry **Pattern History Table (local predictor)** and a 8-entry **Return Address Stack**.

- **Branch Target Buffer(BTB):** The BTB we implemented would store the target address for all PC-relative branch instructions (including unconditional *jal* instruction and conditional branches). And the BTB would be updated once the function unit has the result of the target address. We tested with different sizes of BTB and fixed the size to 32.

- **Pattern History Table(PHT):** We have implemented a 32-entry PHT that each entry contains a 2-bit saturating counter as a local predictor for per PC prediction. And it would only be updated when an branch instruction is retired from ROB. We also tried other branch predictors: Gshare, PAg and Hybrid branch predictor, while the direct local predictor shows the best performance. More detailed analysis would be discussed in Section 4.

- **Return Address Stack (RAS):** Every time, a *jal* jump instruction is fetched, it would push the address of PC+4 into RAS. And the top stack pointer is circulative so it would always store the latest 8 return addresses. If the return address does not match the result from the function unit, when the *jalr* instruction retires, it would clear all the entries in RAS. We also add a register that always stores the latest return address in RAS so if the RAS is cleared due to a squashed signal, it still still provides a speculative address prediction.

2.7 GUI debugger

We have created a GUI debugger in Python using ncurses which displays the current pipeline status for each cycle. It displays the ROB, RS, CDB, map table, I-cache and D-cache for each cycle. Also, it allows the user to navigate using the arrow keys to change the current cycle. The user can also press the home and end button to move to the first and last cycle respectively. It also shows the change in the values with a purple color.

Cycle 671 Press q to quit, ←, →, Home, Page Up, Page Down, End to navigate

ROB

	pos	valid	PC	ready	dest_reg_idx	value	store	mis_pred	branch_target	take_branch	halt
0		0	0	0	0	0	0	0	0	0	0
1		0	0	0	0	0	0	0	0	0	0
2	ht	0	0	0	0	0	0	0	0	0	0
3		0	0	0	0	0	0	0	0	0	0
4		0	0	0	0	0	0	0	0	0	0
5		0	0	0	0	0	0	0	0	0	0
6		0	0	0	0	0	0	0	0	0	0
7		0	0	0	0	0	0	0	0	0	0

Regfile

reg	value
0	xxxxxxxx
1	00001000
2	bd1e2104
3	87b0b0fd
4	b504f32d
5	00000001
6	00000001
7	xxxxxxxx
8	xxxxxxxx

CDB

tag	value	valid
0	0	0

RS

	busy	T_dest	Tag0	V0	V0_ready	Tag1	V1	V1_ready
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0

MT

	mt_tag	tag_ready
0	32	0
1	32	0
2	32	0
3	32	0
4	32	0
5	32	0
6	32	0
7	32	0
8	32	0
9	32	0

ICache

	valids	tags	data
0	1	00	0140016f000010b7
1	1	00	27bb2ee687b0b0fd
2	0	00	0000000000000000
3	1	00	0081220300012183
4	1	00	0012829300000293
5	1	00	023105b30102a313
6	1	00	02358633004585b3
7	1	00	023606b300460633
8	1	00	02368133004686b3
9	1	00	0005d59300410133

DCache

	data	addr	lru_counter	valid
0	27bb2ee687b0b0fd	1	1c	1
1	b504f32d	2	1d	1
2	f28a7b15	200	1e	1
3	4def06ee	201	1f	1
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0

Figure 2: Debugger using ncurses

3 Final Results and Testing

The final version of our P6 pipeline has a synthesized clock period of **11 ns** and it was able to pass all the public assembly and C test cases with the CPI performance below. During the implementation progress, to ensure the correctness of the pipeline, we first wrote unit testbenches that covered most of the possible cases for the major components like ROB and Map Table. Then, after we integrated the pipeline, everytime we added a new feature, we would run the script to make sure it pass all the public assembly and C programs in both simulation and synthesis with an improved CPI result. After implementing the base features, we also adopt the incremental integration testing strategy, especially for 2-way superscalar feature. We first make the output logic to be two way and fully test it after each integration. We incrementally implement and test our pipeline to build a complete 2-way superscalar pipeline.

4 Performance Analysis

This section would discuss how we choose the size of different components for our pipeline and how the implemented advanced features impact the performance of our processor.

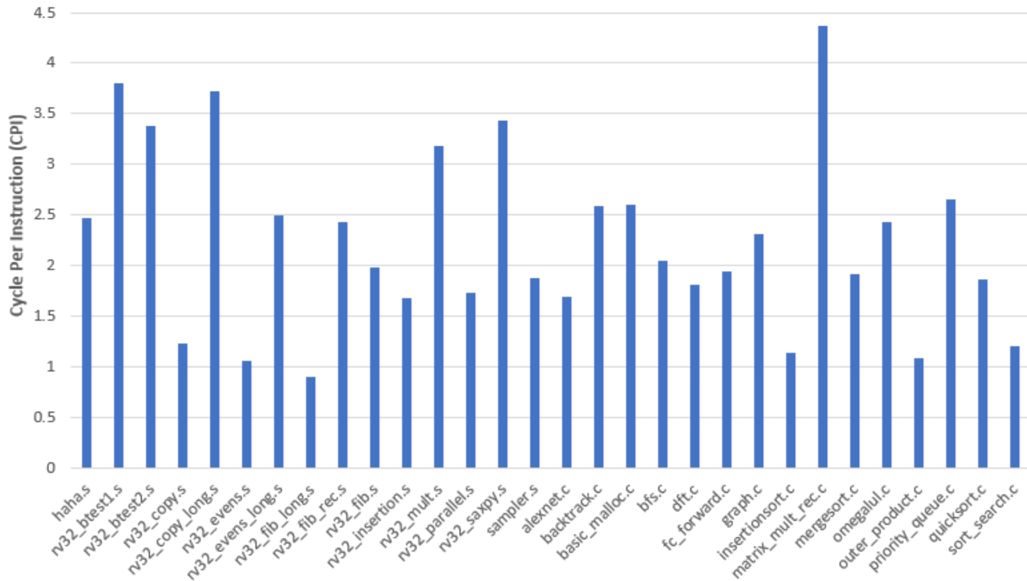

Figure 3: CPI performance of all test cases.

4.1 Impact of Different Module Size

Before we implemented any advanced features, we first checked how the size of the major module components would affect the IPC performance of our pipeline. We tried different combinations of ROB sizes (16 and 32) and SQ sizes (8 and 12). According to the results we got, increasing the size of ROB would result in a generally higher IPC performance as it removes some structural hazards when ROB is full. And when fixing the ROB size, increasing the SQ size also leads to a slight IPC improvement (not obvious from the plot but would save several dozens of cycles for large C programs).

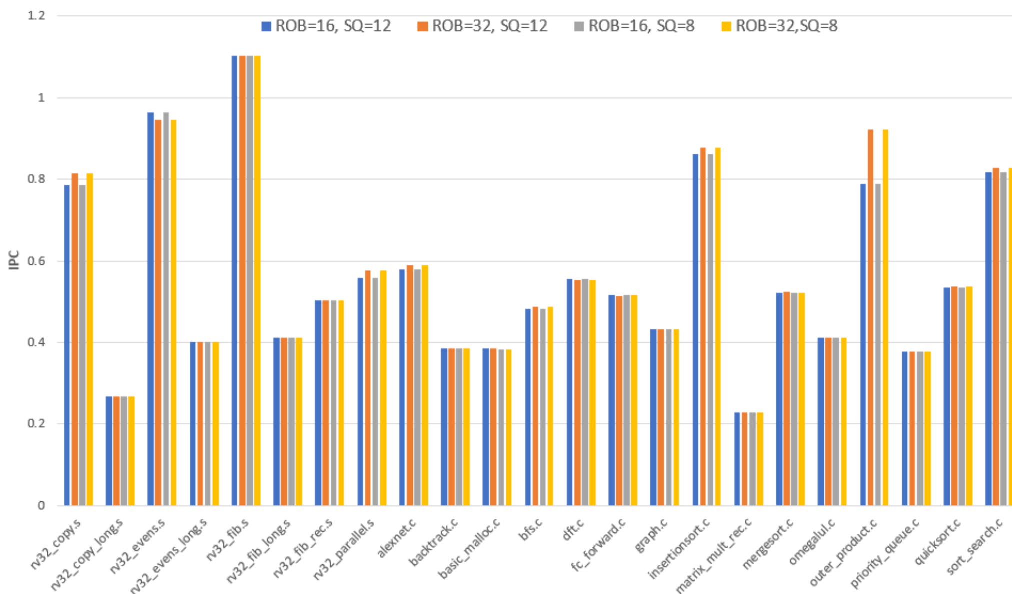

Figure 4: IPC performance with different ROB and SQ size.

4.2 Impact of Different Instruction Prefetching Size

To save more cycles from memory latency, we implemented the prefetcher inside the I-cache. It is able to do the early prefetching if there is a valid predicted target PC address. We also tried different sizes for instruction prefetching and compared the corresponding CPI results as shown in the figure below. We found that when we change the prefetcher size to 7, the CPI would reduce and have the best performance for most of the test cases, which also verified the functionality of our prefetcher. So we finally fixed the prefetcher size to 7.

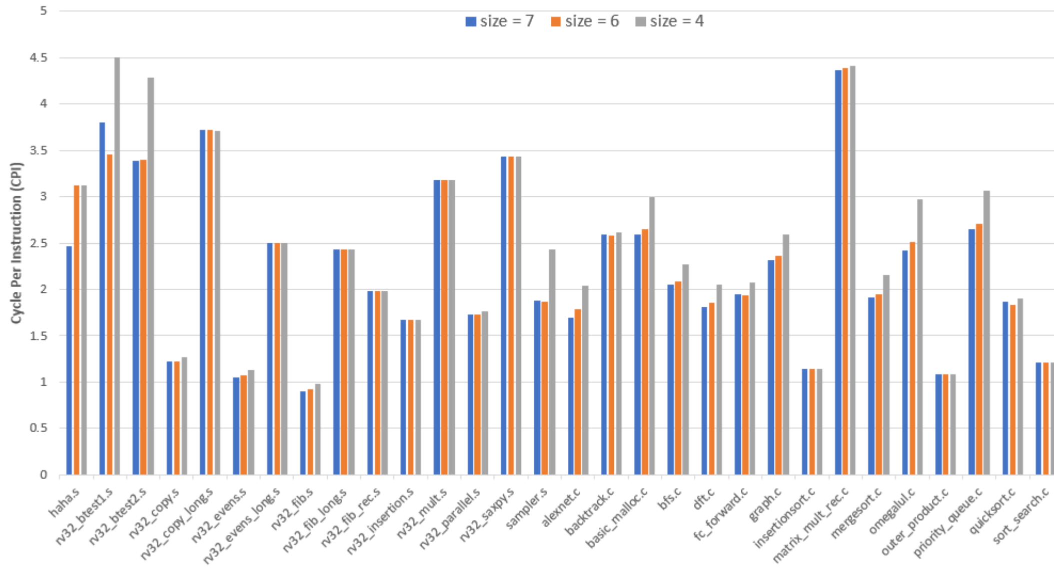

Figure 5: CPI performance with different instruction prefetcher size.

4.3 Impact of Superscalar Implementation

To reduce the CPI for a better program, we implemented the 2-way superscalar execution so the pipeline can dispatch, issue, complete and retire up to two instructions at the same time. By comparing the CPI results of the test cases in the figure below, it shows that the implementation of 2-way superscalar had a significantly positive affect on the processor performance. And based on the test cases we have, it would have an average **1.32** speedup of the original pipeline.

If evaluated by IPC for every test cases, we got that our 2-way implementation would improve the average IPC for test cases from **0.34** to **0.43**.

4.4 Impact of Different Branch Predictor Strategies

The first thing we evaluated was how the **size of BTB** would affect the pipeline performance. So we fixed the branch prediction strategy and then changed the BTB size to check the corresponding CPI results as shown below. For the assembly cases, as the number of the instructions is not quite large, the BTB size does not affect the CPI result. While for C test cases, changing of BTB size would lead to a just small change in CPI. By considering the circuit area and also the affect for larger programs like alexnet.c, we fix the BTB size to 32.

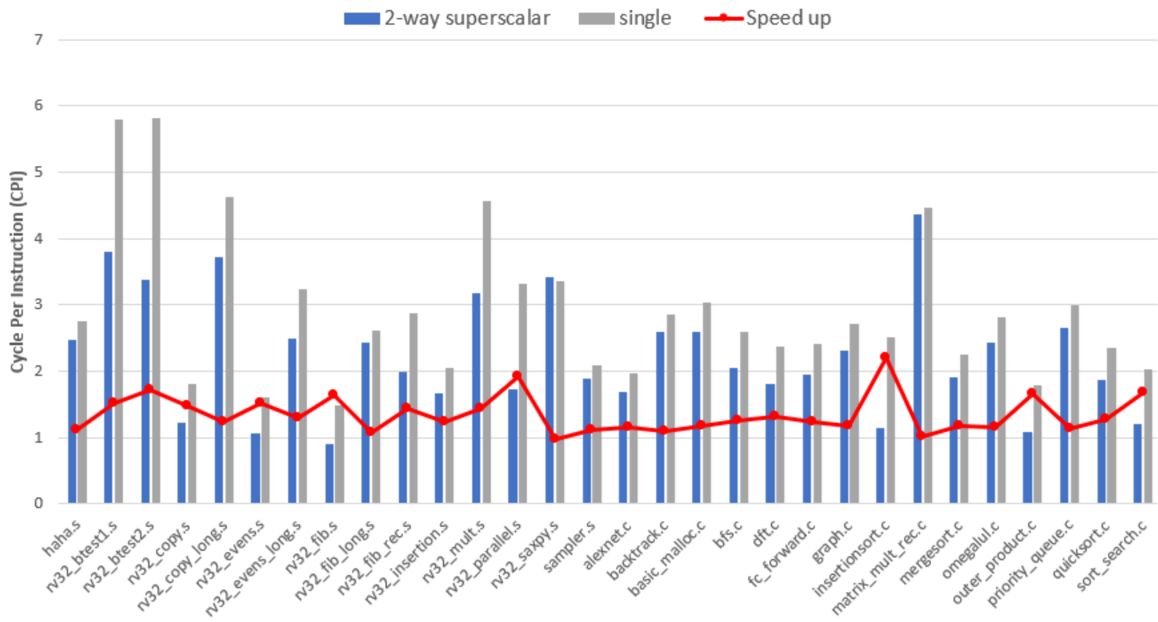

Figure 6: CPI performance of 2-way Superscalar implementation.

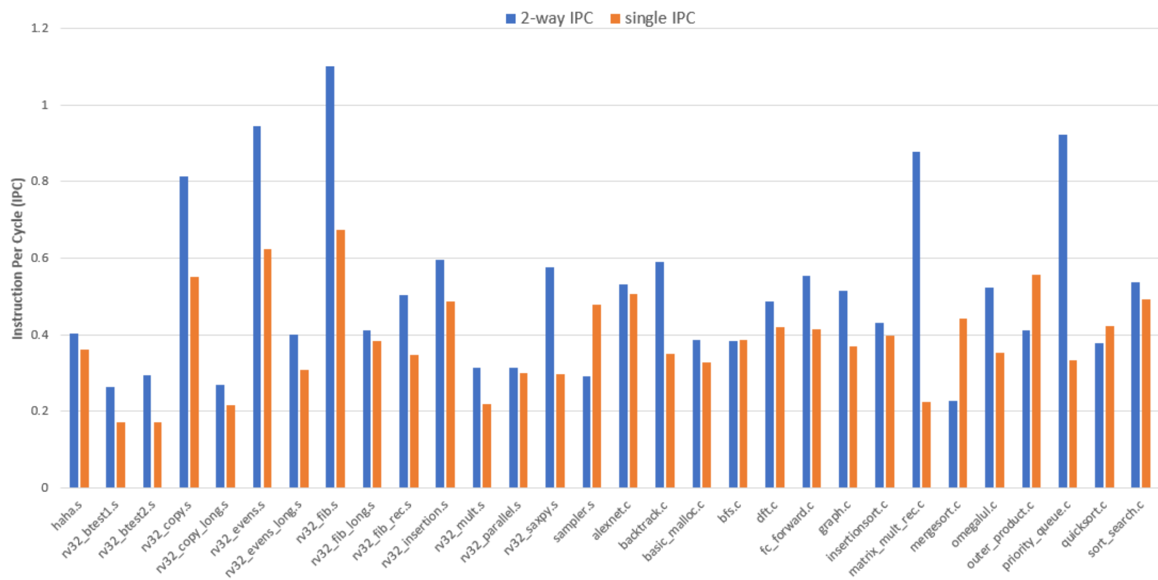

Figure 7: IPC performance of 2-way Superscalar implementation.

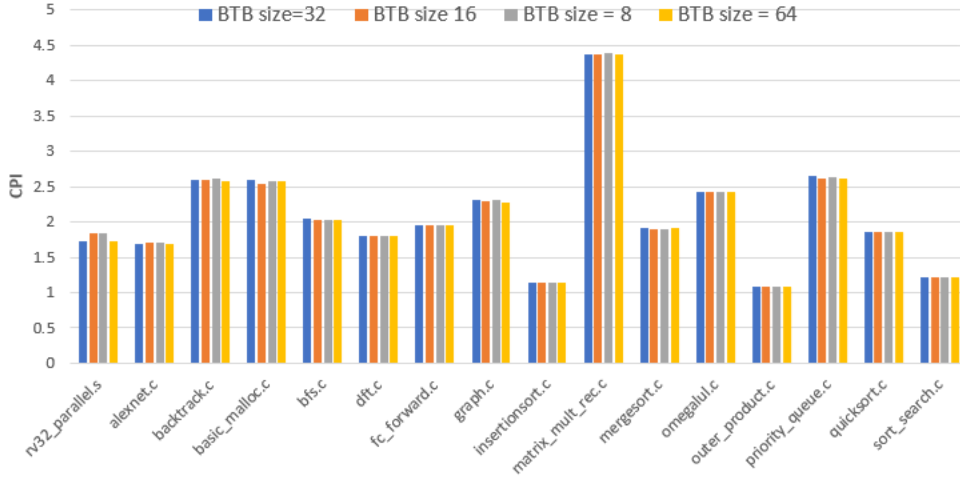

Figure 8: CPI performance of related test cases for different BTB sizes.

The second thing we evaluated was how different **branch predictor strategy** would affect the pipeline performance. We tried four different branch predictors: **Gshare** predictor (5-bit global BHR and a 32-entry PHT), **PAg** predictor (32-entry with 2'b10 as the initial value for saturating counter), **local** predictor (32-entry PHT) and **hybrid** predictor (a local predictor plus a global predictor inside with 32-entry PHT). We used the IPC results when all branches are predicted as not taken as the base line and compared the IPC performance of four different branch predictors. All of the four predictors have an improvement for IPC performance in most of the testcases. We also found that the IPC improvement of the two simpler predictors (local and PAg) surprisingly performed better in most of the test cases than the rest two predictors with a smaller combinations logic block. And the IPC difference between local predictor and PAg was little, so we decided to choose the local predictor with a smaller logic block. We thought the reason that the local predictor has the best performance is that most of the public test cases are related with single for loops.

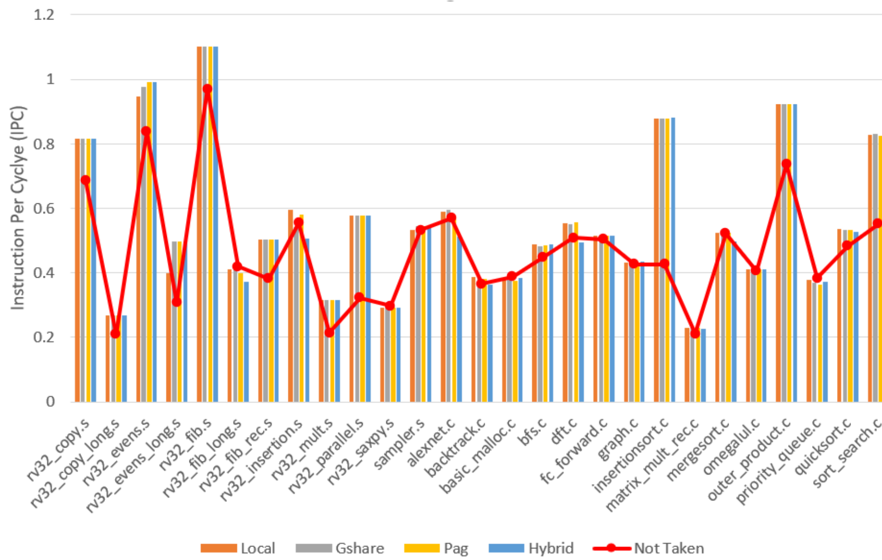

Figure 9: IPC performance of related test cases for different branch predictors.

Another thing we evaluated was **the efficiency of the RAS**. The CPI results with and without the RAS of all test cases are shown. It was evaluated when the 2-way superscalar was disabled (due to time limitation, we did not find a sophisticated strategy to push and pop into/from RAS when there are instructions dispatched in the same cycle). For larger C programs with sub functions calls, the RAS indeed saved some cycles with right return address prediction. The improvement is not quite significant from the plot as expected, since the amount of cycles saved is relatively small with respect to the comparatively large number of total instructions. For example, for the test case *outer_product.c*, it would take 511962 cycles without RAS while 508893 cycles with RAS, which saved around 3100 cycles.

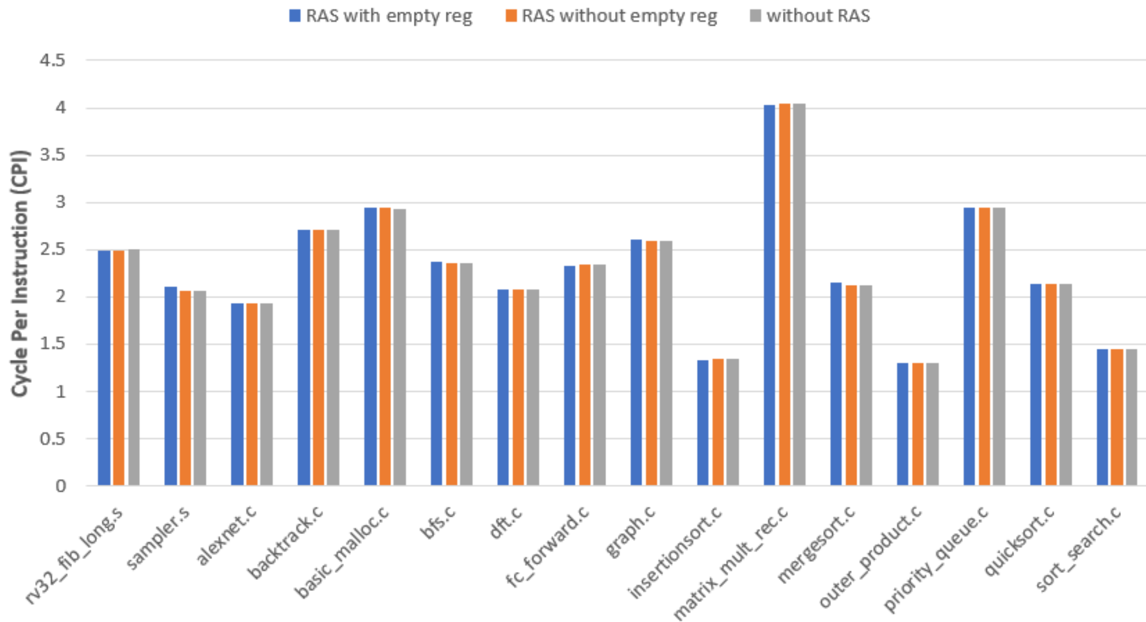

Figure 10: CPI performance with RAS.

4.5 Impact of Different Synthesis Time

We tested the CPI and CPI time cycle time with cycle time of 11ns, 12ns, 13ns, and 14ns. As shown in Figure 11 and Figure 12, despite of the CPI increase with the cycle time, the total running time of each program decrease as the result of CPI × Cycle Time decrease. We chose 11ns in the final submission.

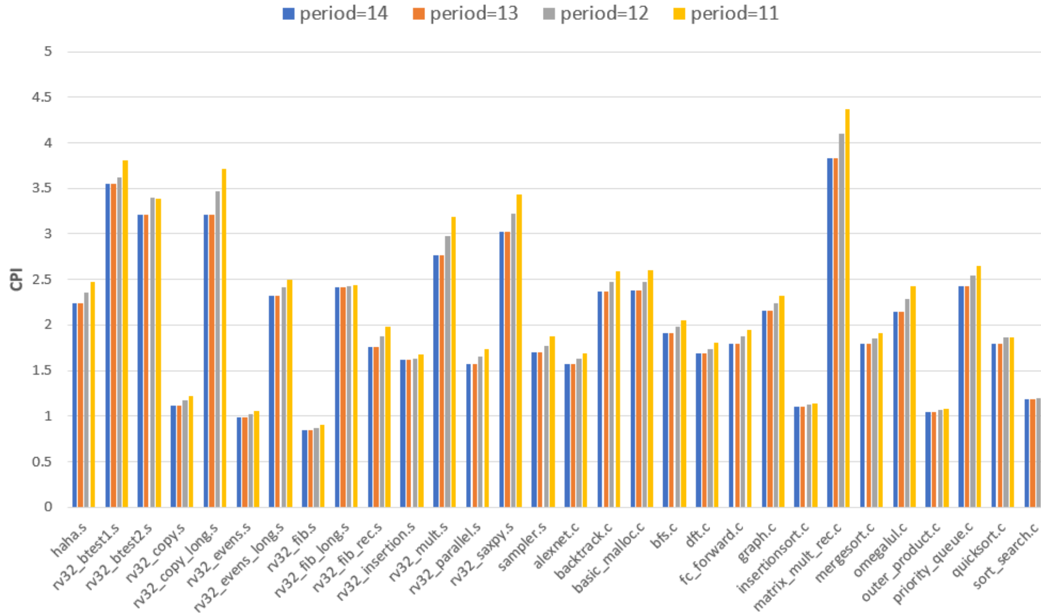

Figure 11: CPI results with different synthesis period.

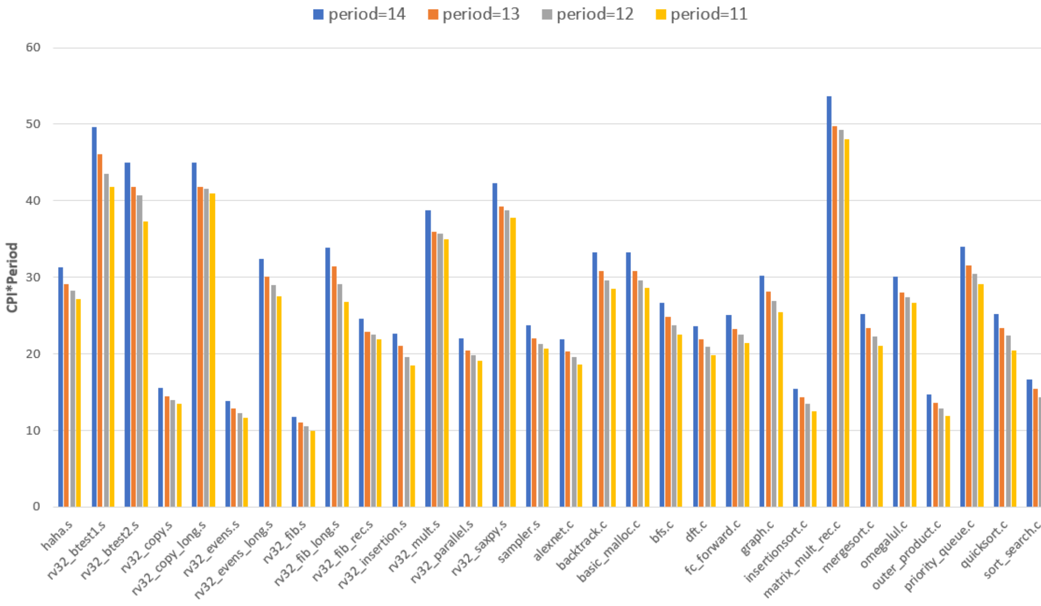

Figure 12: Total execution time with different synthesis period.

5 Conclusion

In summary, we have implemented a 2-way superscalar P6 pipeline in this project that can handle out-of-order executions of RISC-V instructions with improved I-cache and D-cache to solve memory latency, as well as a dynamic branch predictor to enhance prefetching. The pipeline produces the correct output for all the testcases. Generally speaking, we managed to implement almost all the features as we expected in our project proposal.