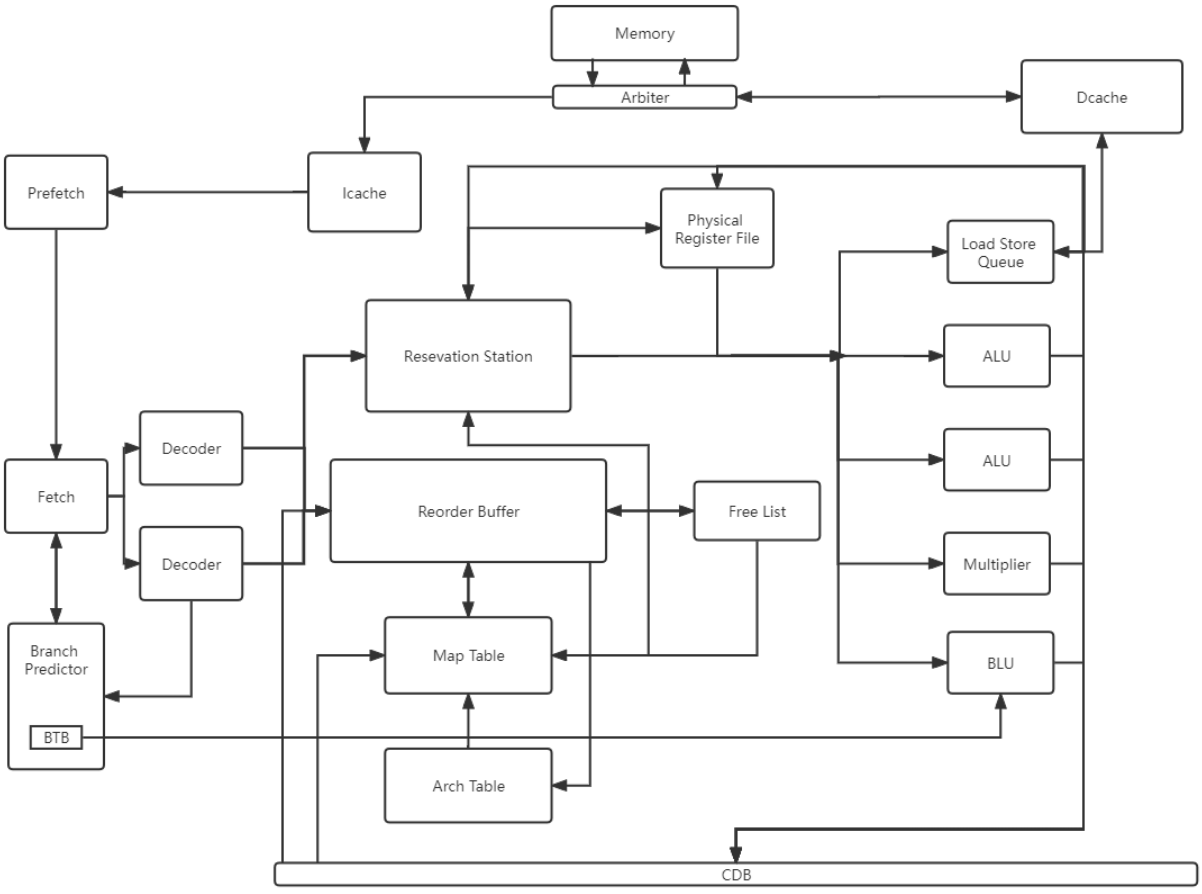
EECS470 Final Project Report

Group 18
Shiqi Yu, Zhijie Xu, Leiqi Ye, Mingxiao Su, Zeqi Han

INTRODUCTION

The project implements a MIPS R10K style processor design based on RISC-V Instruction Set Architecture(ISA). The processor has been designed to achieve Instruction Level Parallelism along with maintaining a balance between cycles per instruction(CPI) and the synthesized clock period. The design that is implemented is capable of fetching, decoding, issuing, completing and retiring 2 instructions per cycle and also includes speculative execution to improve the performance. The report covers the various aspects of the architecture, the components we designed, the features we implemented, including optimal forward, associative cache, prefetch, and visual debugger.

DESIGN AND IMPLEMENTATION



2.1 Design Overview

The project implements a uniform 2-wide superscalar machine with SystemVerilog HDL. The design specification with the top-level diagram of the entire architecture is shown in Figure. Our basic architecture is a 2-way superscalar machine. The basic technical requirements include the following four items, all of which we have met in the project:

1) An instruction cache and a data cache.
2) Multiple functional units (FUs) with varying latencies.
3) An out-of-order implementation
4) Branch prediction with address prediction

We implemented 5 advanced features which include 3 major features and 3 minor features:

1) 2-way superscalar
2) Speculating on load dependencies and forwarding
3) hardware prefetch
4) associative cache
5) a pretty good visual debugger
6) SMT (design only)

2.2 Module Specification

R10k is a popular out-of-order architecture. We choose to implement R10K instead of P6 for its elegance. The R10k only needs tags to be passed between Reservation station, register file and function units, whereas the P6 needs to pass the entire data from architectural register file, reorder buffer, reservation station, function units, etc.

Fetch Stage

1. Fetch
   The fetch module is responsible for fetching instructions from the memory hierarchy. The internal PC register keeps track of the execution flow of the program and is updated accordingly when the fetched instruction is successfully dispatched, a branch misprediction needs to be recovered, or the branch predictor enables a prediction. To simplify the design, the fetch always requests 8 bytes from Icache. The fetched instruction is stored in the fetch buffer and there is a pointer associated with the buffer itself. This PC pointer will keep the correct dispatching order.

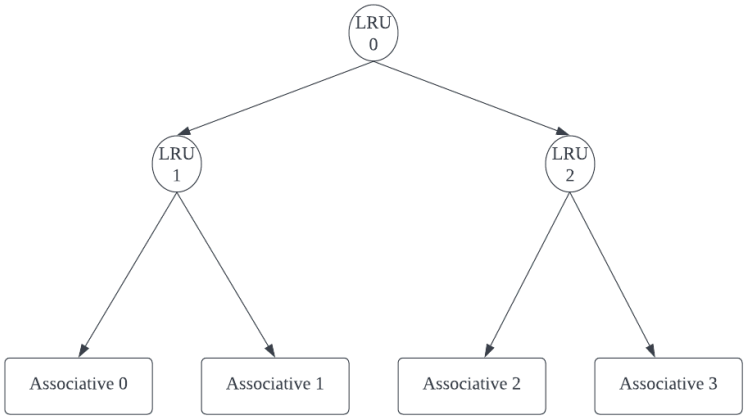
2. Instruction Cache
   The instruction cache is where we store instructions that are of high temporal or spatial locality. In our memory hierarchy, we have only one L1 instruction cache with 32 cache lines and an 8-byte block size. Reading from the instruction cache is significantly faster than reading data from the main memory which greatly lowers the idle waiting time of function units.

3. Branch Target Buffer and Branch Predictor
   The branch target buffer stores the target address of branch instructions that are determined by the Branch Unit to be taken. The stored target address will be used as the

next PC address when the predictor predicts that the current branch instruction will be taken. It will also be stored in the Branch Unit to check if the address prediction is correct. As for the branch predictor, it is a simple saturation counter, aka a two-bit branch predictor. The internal state of the predictor will be updated when it receives feedback, i.e. branch recovery, or retiring without recovery. Compared with the single-bit branch predictor, the saturation counter is surprisingly more stable and more accurate. In our design, we do not store the tag of the PC address, and thus the prediction and predicted address may be totally irrelevant because it may be actually from other blocks of the program. Besides, the two-bit predictor in nature is very simple. Even though, the performance is pretty well. The number of entries is 32. More details will be discussed in the analysis section.

4. * Instruction prefetch
   In our fetch stage, we implement an instruction prefetcher. Without the prefetcher, the processor will frequently be stalled due to high memory latency. Our instruction prefetcher is a simple stride prefetcher that will prefetch the next block of memory based on the current PC address. We choose this because instruction memory tends to be accessed much more sequentially and thus yields more performance improvement rather than a waste of resources. The prefetched instruction is stored in the fetch buffer. In the fetch module, we mention that we have a PC pointer that keeps track of the dispatch order of the program. Now we add another pointer to the fetch buffer, the prefetch pointer. It can help maintain the right order of prefetched instruction. It will be updated when instructions are successfully prefetched or a branch recovery occurs. The number of entries of the fetch buffer is 16.

5. * Associative Icache
   On top of the provided instruction cache, we implement a 4-way associative cache. Compared with a direct-mapped cache and fully associative cache, a 4-way associative cache maintains a certain degree of both temporal locality and spatial locality. A direct-mapped cache tends to evict content mapping to the same address more frequently and a fully associative cache tends to evict content least recently used. In our case, the 4-way associative combines their strength which makes it more stable. Furthermore, we use true LRU as the replacement policy. We use three bits per cache line to maintain the LRU status. Below is a figure that shows how the LRU bits work.



Each bit stores that information that which part, left or right, is accessed more recently. Every time the associative entry is accessed, the LRU bits are updated accordingly to reflect which entry is most recently accessed. By flipping the bit and tracing down the hierarchy, we can find the least recently used entry.

Dispatch Stage

1. Reorder Buffer
   Reorder buffer is part of bookkeeping part of out-of-order execution. It keeps track of the status of dispatched instructions and determines when they can be committed in order. There are 32 entries in our ROB and each entry carries necessary information for bookkeeping. There are two pointers associated with the buffer which indicate status in two stages, dispatch (tail), and retire (head). These two pointers are updated accordingly when there are instructions dispatched or retired. When there is a branch recovery, the tail will be forced to align with the head which means discarding everything the pipeline has already dispatched but not retired yet.

2. Freelist
   Freelist keeps track of the usage of physical renaming registers. There are also two pointers associated with the list. The head pointer is to track registers that are going to be allocated. The tail pointer is to track the register returning from the pipeline. The size of the freelist is the same as the size of ROB, 32.

3. Map Table
   The Map table is used to update the ready status of the input physical register in the dispatch stage and the ready status of the physical register in the competing stage. The most important function of the Map Table is to build the link between the physical register and the architecture table. In the dispatch stage, the Map Table gives the ready status of the physical register to RS as initial the ready status and sets the ready status of the new physical register index to 0. In the complete stage, the map table updates the ready status of the physical register to 1, preparing for use in the retire stage in the ROB. If it receives the branch recover signal, the Map table updates all table values from the

Architectural table. The number of map table entries is 32, the same as the number of architecture registers.

4. Decoder
   The decoder is part of the dispatch stage, decoding the instruction to the operation code, corresponding reg number, and control value. Packing them to the decoder packet as output and giving it to ROB, RS, BTB, Map Table, Free list, and even other modules. We have two decoder modules to build a two-way superscalar pipeline.

Issue Stage

1. Reservation Station
   The reservation station is the key point of out-of-order execution. It will store the instructions that are not ready yet because of hazards and issue them to function units when it is their time to be executed. In our design, we implement a fully associative reservation station rather than a FIFO that can only be connected to a certain function unit. The price is that the routing logic between RS and FU becomes way more complex. As for issue logic, the older instructions will be prioritized.

Execute Stage

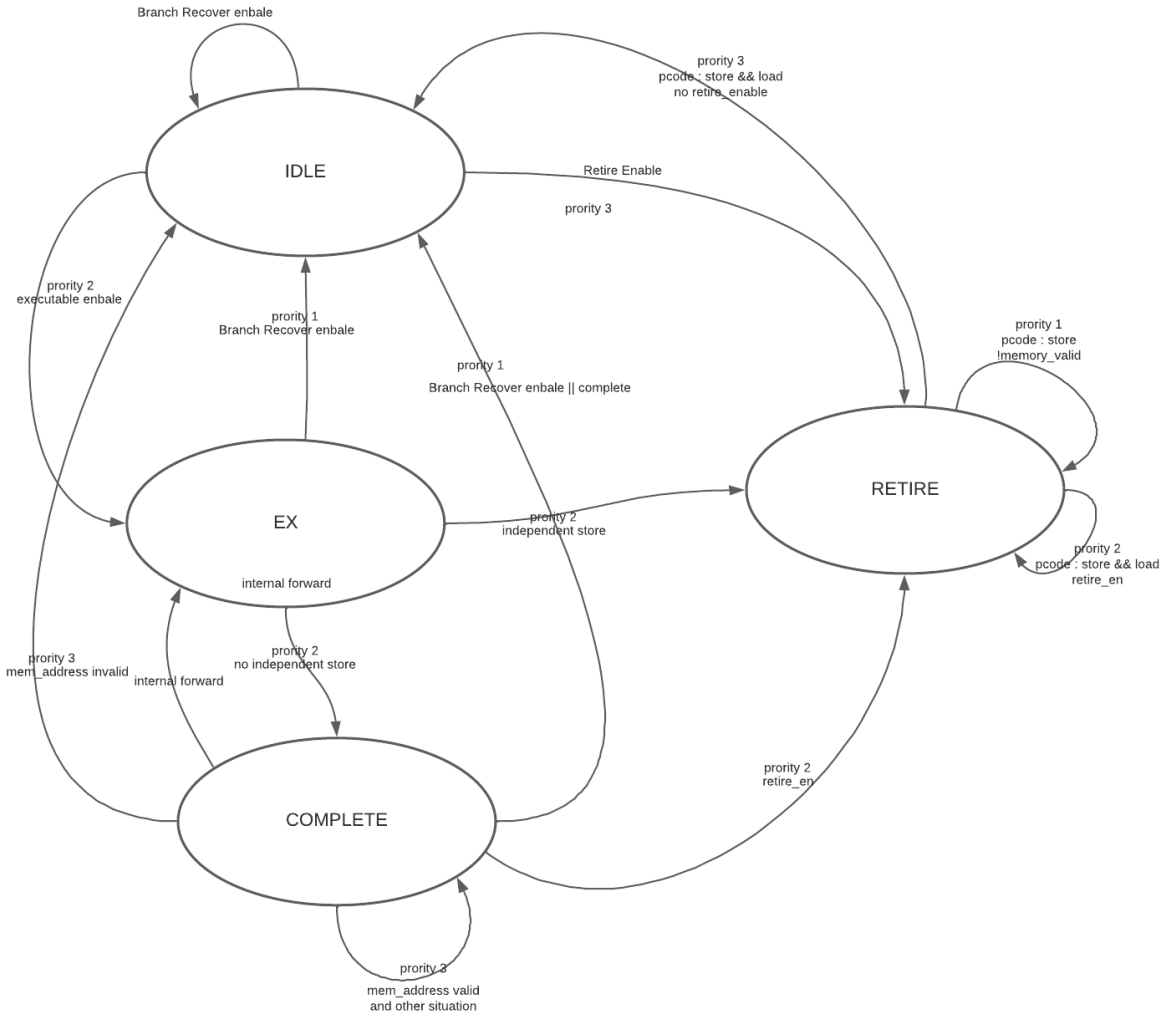
1. ALU
   We used 5 Functional Units(Fus): 2 ALU, 1 multiplier, 1 load/store address queue, and 1 branch unit. There are 4 types of functional units used in our processor. Each of these units reads its operands from RS and reads its register value from the register file, performs its operation, and then sends the result to CDB. The latency of the functional units varies depending on the type of operation. What's more, branch units and the load/store queue have more operations than calculations, which will be discussed later. When each function unit is ready to accept data from RS and the register file, it will send a ready signal to RS. When a function finishes processing, it will send a done signal to CDB. Using these two signals, function units communicate with other modules.

2. Branch Unit
   The branch unit receives operand and opcode from RS and physical register file and then uses them to calculate branch conditions. Internally, there is a buffer that will allocate entries for the dispatched branch instructions and deallocate entries for the retired branch instructions. This FIFO helps keep the track of the order of branch instructions. It can also be merged into ROB entries but we think putting it in BLU makes more sense in the term of decoupling. Meanwhile, the branch unit also records predictions from BTB, which happens at the dispatch stage, and outputs recovery instructions if there are any mispredictions in the retire stage. When the execution stage is enabled, it will calculate the branch target address, compare the result with the prediction, look up the corresponding entry, and update it with the calculated result. The size of the buffer is 32, the same as the ROB size.

3. Multiplication Unit
   The mult unit reads operands from the register file and executes instructions in 4 cycles when the execution stage is enabled. Both the length and execution cycles are parameterized and can be changed. Internally, it calculates signed multipliers and multiplicands in 64 bits and returns 32 bits versions of them according to specific instructions.

4. Physical Register File
   Register file has 4 ports and supports internal forwarding as register write happens, which is on the complete stage enabled by CDB signals. Register file uses CDB tags as index to write corresponding values, which come from different function units. The total volume is 64, which equals the sum of architecture register number and rob size.

5. * Load-store Queue with Speculation on Load Dependencies & Forwarding
   The load-store queue is responsible for dealing with out-of-order access to data memory. Below is a diagram of the finite state machine of our load-store queue.



Our load-store queue supports out-of-order load execution, store to load forward, and faster execution on blocked load. It is compatible with our pipeline design. When the pipeline retires two memory instructions at the same time or consequently, the LSQ is able to handle the situation. There are 4 stages in our load-store queue. Below is a detailed explanation of each stage.

The first stage is IDLE. In this stage, the load-store queue is able to send a ready signal to RS and receive an execute enable signal from RS. Once it receives the enable signal, it will enter the EXECUTE stage.

In the EXECUTE stage, it will upload the internal buffer based on the current instructions. Also, if the current instruction is load, it will traverse the buffer to check if there is an unissued store before or potential forwarding. If there is an unissued store, the LSQ will record the tag of the store and put it in the corresponding entry so that when this store completes, the load will be triggered again to see if it can be executed. As for forwarding, the LSQ does not only check the address but also takes size into consideration so that the store can be optimally forward in nearly all cases, like store word to load byte, multiple store byte to load word, or store half to load half. As long as it

is the same thing in the memory, it can be forwarded. If nothing above occurs, the load can be executed right now. After that, LSQ will enter either the IDLE, if the load is blocked, or the COMPLETE stage, if the load is completed.
In the COMPLETE stage, LSQ behaves like other function units. It will wait for the CDB signal and store the result in the register file. When there is a blocked load in the buffer and it is blocked because of the current store, the LSQ will go back to EXECUTE stage to execute that load again and check if it can be executed now. If the LSQ receives a retire enable signal, it will enter the RETIRE stage rather than IDLE because the LSQ has to deal with all retired instructions first to avoid potential physical tag conflict.
In the RETIRE stage, the LSQ will write back to memory until there are no retired instructions left in the LSQ.

Complete Stage

1. CDB
   The Common Data Bus (CDB) takes results from 5 different function units and outputs them on a bus. The CDB gives priority first to multiplication results, then load-store queue results, then BLU results, and finally ALU results. If there are more than two inputs coming into CDB, it chooses two, and then sends out a stall signal to the function units whose values are not going to be broadcasted. The only result that is not broadcast on the CDB is store. Stores do not have a destination PRN, but they have a ROB number. Instead of putting this ROB number on CDB, we forward it directly to the ROB.

Retire Stage

1. Architecture Table
   The architecture table serves as the status register of committed instructions. It stores the physical tags of architecture registers at a specific time. It will be updated when there are retired instructions. When a branch recovery occurs, the content of the map table will be overwritten by the content of the architecture table to restore the in order commit status.
2. Data Cache & Arbiter
   The major difference between data cache and instruction cache is that there is a evict stage in data cache. If the entry being accessed is dirty and needs to be replaced, the data cache will firstly store it back to the main memory. After that, the data cache will load data from corresponding address in before anything else happens based on write allocate policy.
   The arbiter works as the bus between main memory and two cache. It will determine the ownership of current communication with main memory and route the signal accordingly.

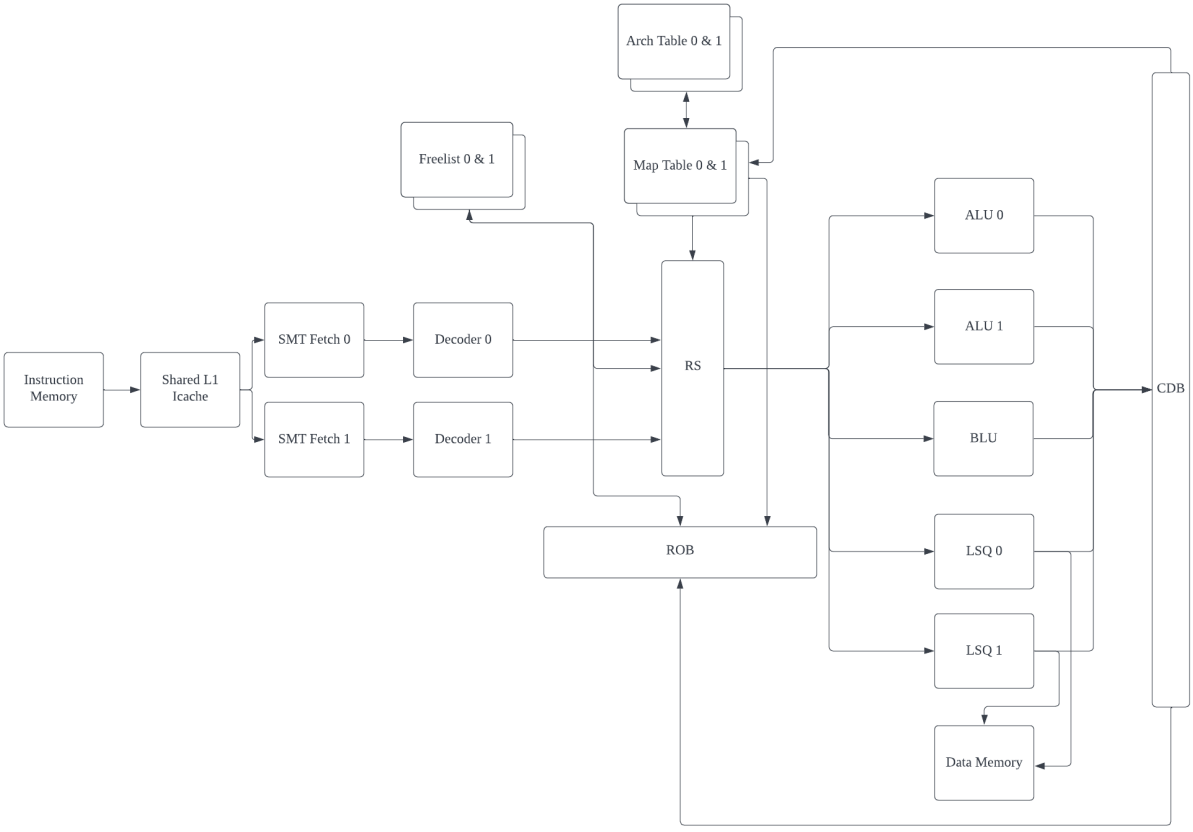
2.3 Two-way Superscalar Design

As our main advanced feature, we implement a two-way superscalar pipeline that can dispatch, issue, complete, and retire two instructions at the same time. This superscalar design greatly improves the performance because it breaks the bottleneck of the traditional pipeline where the theoretical highest CPI is 1. With great improvement comes great overhead. There are tons of

dependency checking because the pipeline dispatches two instructions at the same time. Inside most bookkeeping modules, we have an internal forward to solve the dependency between two instructions. During the development, we always found something that we ignored during design by tracing back bugs. Although the superscalar sounds like an ultimate solution, the performance is not as high as expected. The lack of instruction-level parallelism greatly impairs the performance. However, this issue can be solved by SMT which will be illustrated further in the next section.

2.4 SMT Design

In our proposal, we proposed to implement a Simultaneous Multithreading mechanism to hide memory latency, reduce the probability of true dependency, and improve FU utilization. However, due to a variety of limitations, we do not have enough time actually to implement this part. The good thing is that our modules are designed to be compatible with SMT so we do not need large-scale rewriting to make it work. Below is a simplified diagram for the pipeline with SMT. The major change is that there are two sets of bookkeeping modules.



1. SMT Bookkeeping
   The main difference between SMT and the traditional pipeline is that we need more bookkeeping parts to keep track of the status of each hardware thread. Currently, our configuration is 2 simultaneous threads. So, we need two freelists, two map tables, two

architecture tables, and two sets of physical registers. Other modules, like ROB, RS, and CDB will remain the same.

2. SMT Memory
   As for memory hierarchy, we could use completely isolated two L1 instruction caches, two L1 data caches, and two main memory modules for these two threads. In this way, there is no need to worry about cache hit rate and coherence issues. There is also another design choice which is using the same memory hierarchy but adding a thread tag to every cache entry. Based on the assumption that these two hardware threads will not interfere with each other, the thread tag is capable of tacking the ownership of the cache. If configured properly with associative cache, the performance should be acceptable.

3. SMT Dispatch Logic
   The bottleneck for processors like this project lies in the memory latency and dependency stall. The core of SMT is that it can hide high latency memory operations, lower the possibility of true dependency in program flow, and thus improve FU utilization. Its nature is to make the incoming instruction irrelevant so that there is no need to stall. For the SMT pipeline, there are two fetch stages with an output signal indicating if there are two instructions in the fetch module that are ready to be dispatched (2-way superscalar). If both fetch modules are able to dispatch, we will dispatch one instruction per fetch module. If they are not both ready, just dispatch whatever they can. The fetch modules can be stalled by instruction memory access and the structural hazard of a single thread. In this case, the other fetch module will be able to dispatch. In this way, the FU utilization is greatly improved.

4. SMT Routing & Dependency-Check
   This is the main reason why we do not have time to implement SMT. Even though our modules, like map table and freelist, support individual dispatch signals, we still have to put some dependency-checking work in the pipeline. To make it work with SMT, we need to add a thread tag checking part in every line of dependency-checking code because instructions from two threads do not have any dependency. We have also considered using index 0-31 for one thread, and 32-63 for another. In this way, in some cases. it may work flawlessly, and there is nothing much to worry about. As for the routing, the selection logic will also be more complex than before. Although this is an implementation difficulty rather than a design issue, it is a considerable obstacle.

5. Potential Problem
   In a paper about SMT, we learned that the physical registers for SMT are way too large to be regarded as registers, especially for a larger number of threads. In this case, access to physical registers has to be split into multiple cycles to avoid a long critical path. Besides, we also need to consider how to efficiently rewrite our modules so that they can support clearing specific entries instead of everything when there is a mispredicted branch recovery. Furthermore, the combinational logic for routing may also

be a serious issue. The paper proposed a method such that certain threads can only be connected to a limited number of RS and FU to reduce the complexity.

TEST

Our test is based on each module before and after the synthesis. Each module is designed and tested using test cases to make sure that each module has basic functionality. For some important modules like RS and ROB, we build very complicated test cases to make sure they perform very well. And most modules are in this order. 1. We build the basic code of one module and test its functionality correctly under the simulation. 2. We test its functionality correctly under syn. After finishing the basic module, we combine all the modules together and carry on new tests. Firstly, we test the pipeline without LSQ. In this process, except for the test case provided, we also test some test cases we write. After designing and testing the load-store queue, data cache, instruction cache, and memory arbiter, we integrated them and tested them with more complicated test cases provided. To compare the correctness of the pipeline work, we make the pipeline print out each instruction's writeback.out with the in-order version of Project3, which is more strict than the basic requirement, i.e. program.out. On top of that, we also modified the in-order pipeline to print the address and content associated with the load and store instructions. We also wrote more programs to test the functionality of the whole pipeline and figure out more bugs that the public test cases cannot figure out or are too hard to find. Additionally, we developed a visual debugger to monitor the pipeline, which can help us to find the wrong cycle and wrong data quickly. Below is a screenshot of the test cases we developed for RoB and RS. There are generally hundreds of lines of code in testbench and we have testbench for every single module, including CDB. As for debugging, we print out the difference between the writeback.out between in-order pipeline and our design. It contains detailed information about each instruction. We use this to locate the bug. We also print out the writeback.out with $realtime to make our life easier when looking for it in the DVE. To better understand the program behavior, we look through the program.debug.dump to track the execution of instructions.

```
// Retire two, request two, forward to request
dispatch_en_in = 2'b11;
retire_en_in = 2'b11;
retire_tag_in[0] = 2;
retire_tag_in[1] = 3;
#1;
/*
+---+----+
|   | T  |
+---+----+
|   | 0  |
+---+----+
|   | 1  |
+---+----+
|   | 2  |
+---+----+
| t | 3  |
+---+----+
| h | 12 |
+---+----+
|   | 13 |
+---+----+
|   | 14 |
+---+----+
|   | 0  |
+---+----+
*/
`CHECK(free_reg_out[0], 2);
`CHECK(free_reg_out[1], 3);
`CHECK(free_reg_en_o[0], 1);
`CHECK(free_reg_en_o[1], 1);
```

ANALYSIS

Due to the limitation of time, we did not aggressively push our clock period. In the final version, the synthesizable clock is 24ns. Based on the synthesis result before, we think 19ns would be an adequate approximation for the real clock period. Below is a table of synthesizable clock period of each module.

Module	Synthesizable Clock
Fetch	9
ROB	5
Regfile	2.8
CDB	3.1
Arch Table	5
RS	10
Map Table	2.3
Freelist	5.3

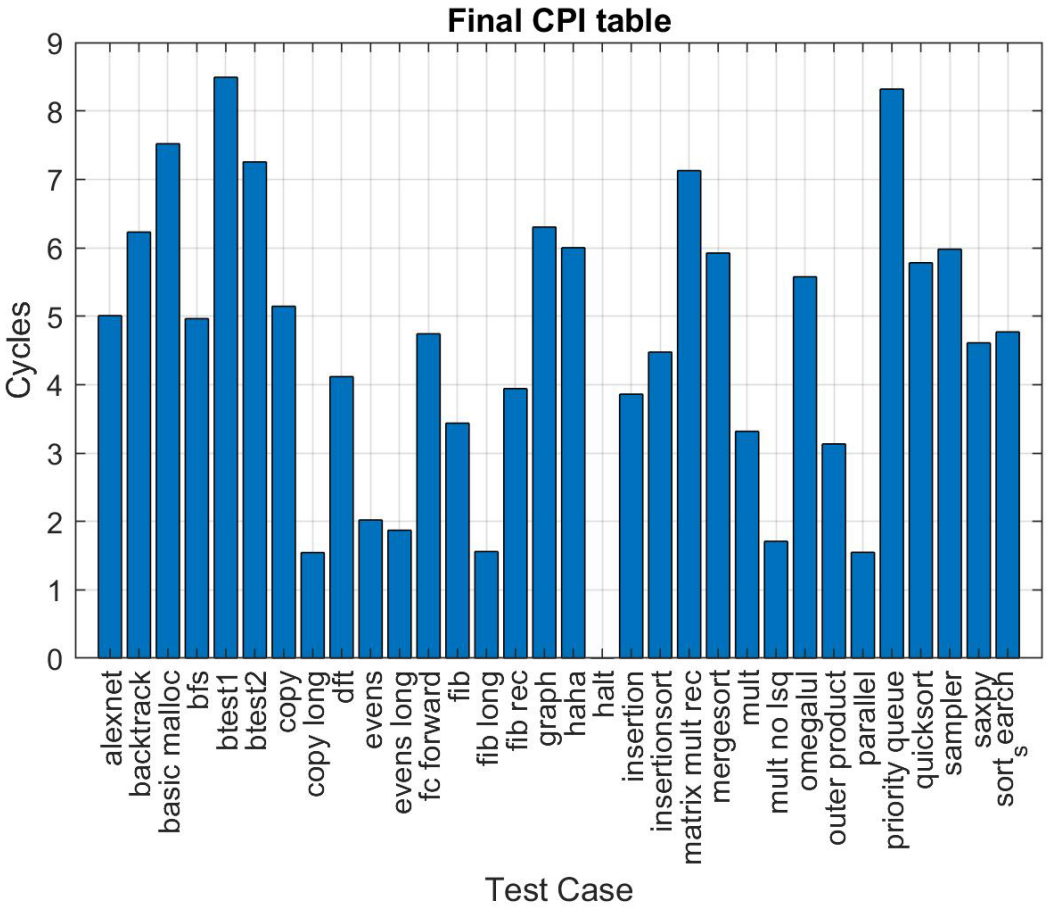
```
/*
Allocate 4 instructions in sequence.
+---+----+------+---+
|   | T  | Told | C |
+---+----+------+---+
|   | 32 | 1    | 1 |
+---+----+------+---+
| h | 33 | 2    |   |
+---+----+------+---+
|   | 34 | 3    |   |
+---+----+------+---+
|   | 35 | 4    |   |
+---+----+------+---+
|   | 36 | 5    |   |
+---+----+------+---+
|   | 37 | 6    |   |
+---+----+------+---+
| t |    |      |   |
+---+----+------+---+
*/
`NEXT_CYCLE;
`CHECK(head_debug, 1);
`CHECK(tail_debug, 6);

CDB_i[0] = 6'd33;
CDB_i[1] = 6'd34;
CDB_en_i[0] = 1'b1;
CDB_en_i[1] = 1'b1;
dispatch_en_i[0] = 1'b0;
/*
```

Branch Unit	10
Load Store Queue	12
Decoder	5
ALU	3

The critical path lies in the execute stage where LSQ reads a value from the register file and access data cache in the same cycle. The configuration table is below.

Module	Size
Superscalar	2
Fetch Buffer	16
Reorder Buffer	32
Reservation Sation	4
Freelist	32
Tables	32
LSQ	32
Branch Unit	32
Branch Target Buffer	16
Icache	8 * 32 * 4
Dcache	8 * 32



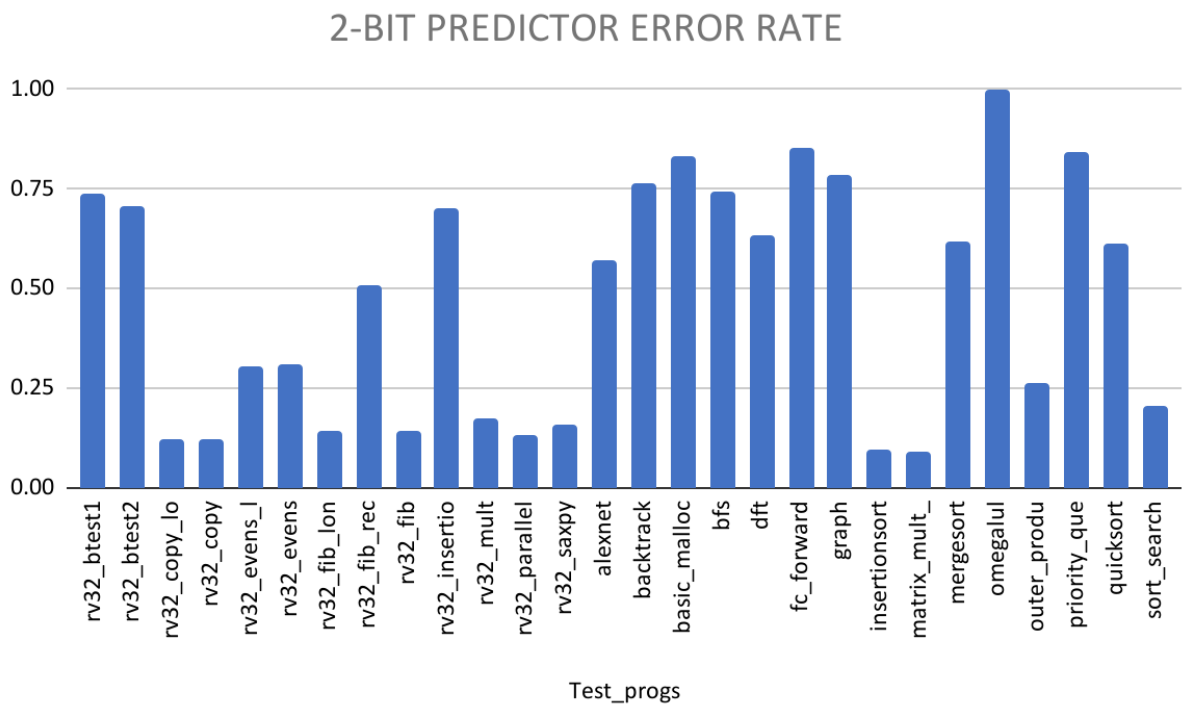
4.1 2-Way Superscalar

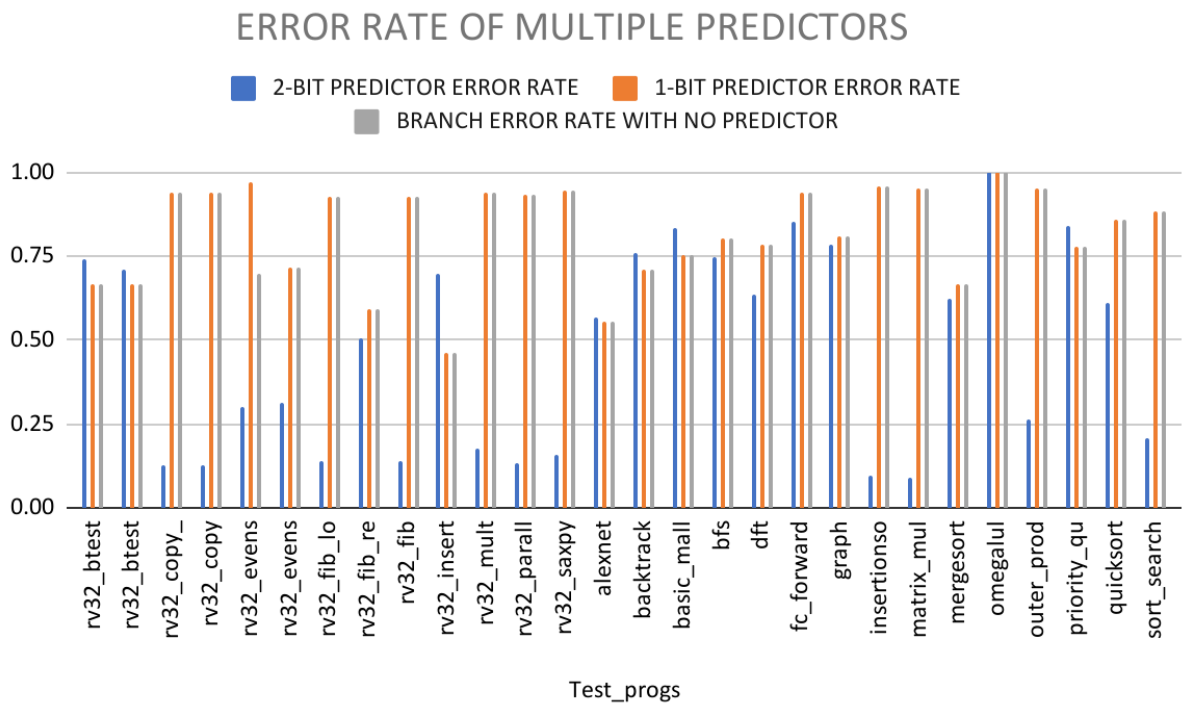
We do not have a really good way to show how 2-way superscalar improves the performance except for showing the CPI. To demonstrate we have a working 2-way superscalar pipeline, we collect the data that total cycles that we retire two instructions at the same time.

4.2 Branch Prediction

As described in the BTB and branch predictor section, we use a saturation counter, aka a two-bit branch predictor. Although the design is simple and there are some oversimplifications, the performance is surprisingly good. Serving the purpose of comparison, we also test the static not taken predictor and one-bit branch predictor. The one-bit predictor simply repeats the same branch behavior. From the figure, we can see that a two-bit branch predictor significantly reduces the prediction error in some programs, such as insertion sort. However, the program that does not contain many branch instructions is still unpredictable. From the comparison, we can see a two-bit predictor is generally more accurate than one-bit predictor or condition with no

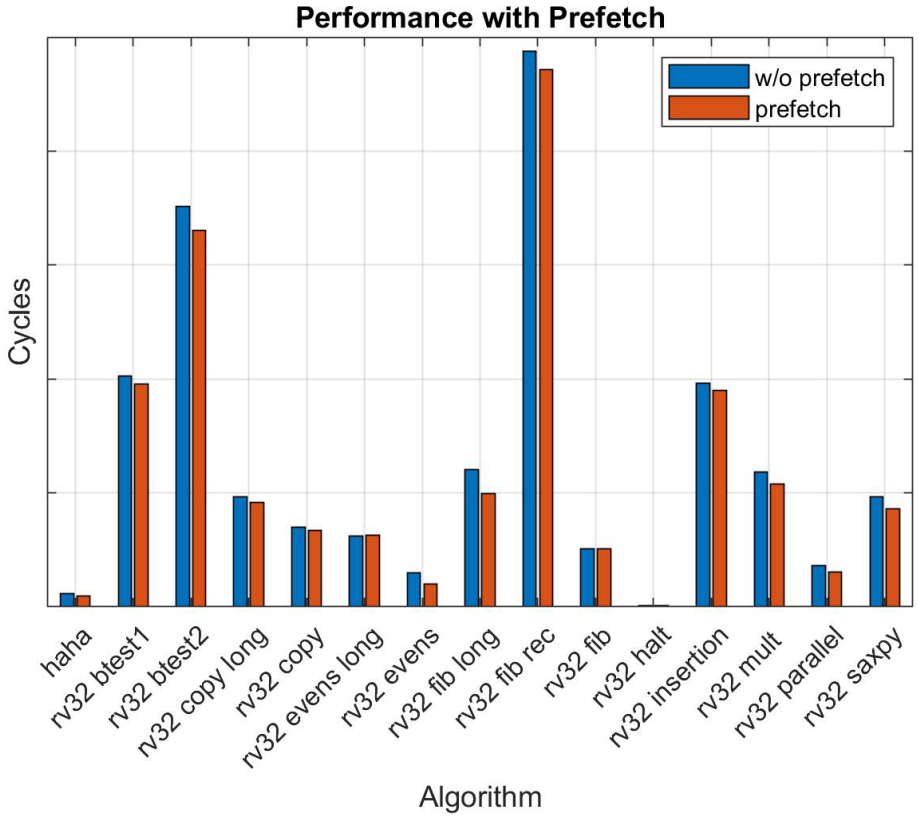
prediction at all.

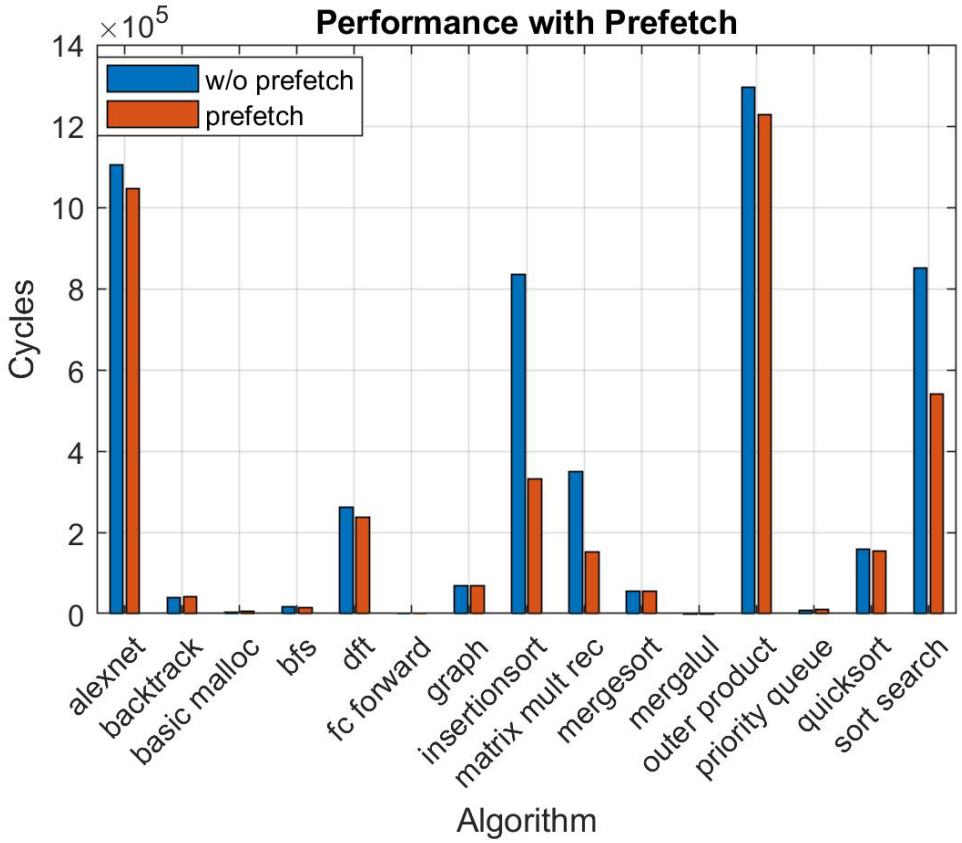




4.3 Prefetcher

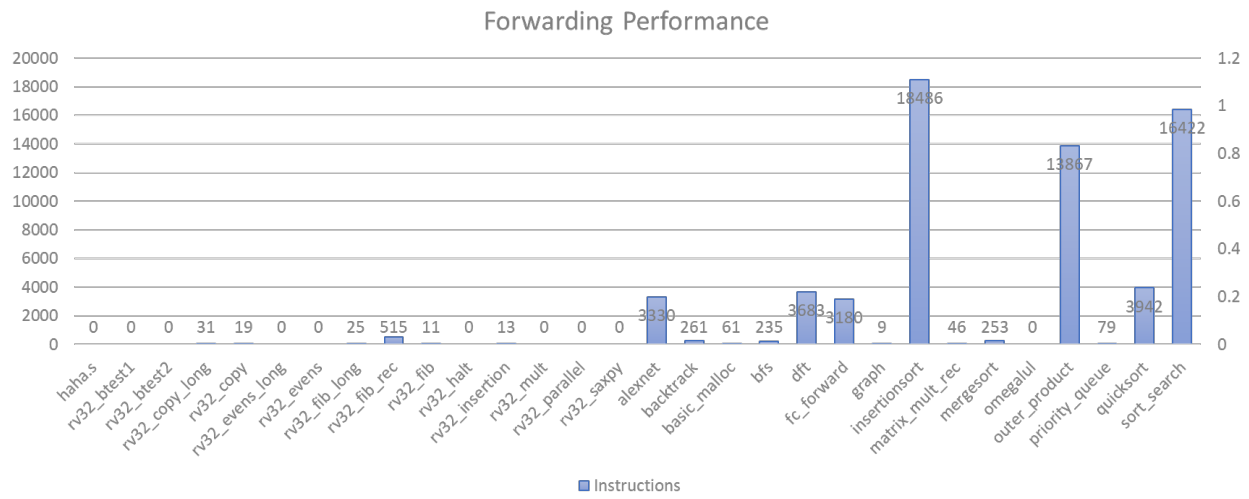
To test our prefetcher, we test it with the size of 16. Currently, we do not have synthesis results to guild us to make the tradeoff. But it will not affect the real critical path. From the result, we can tell that our sride prefetcher is effective for programs with predictable execution flow, i.e. less branch instructions. This performance is also influenced by the prediction rate of our branch predictor because branch prediction may thrash prefetched instruction.





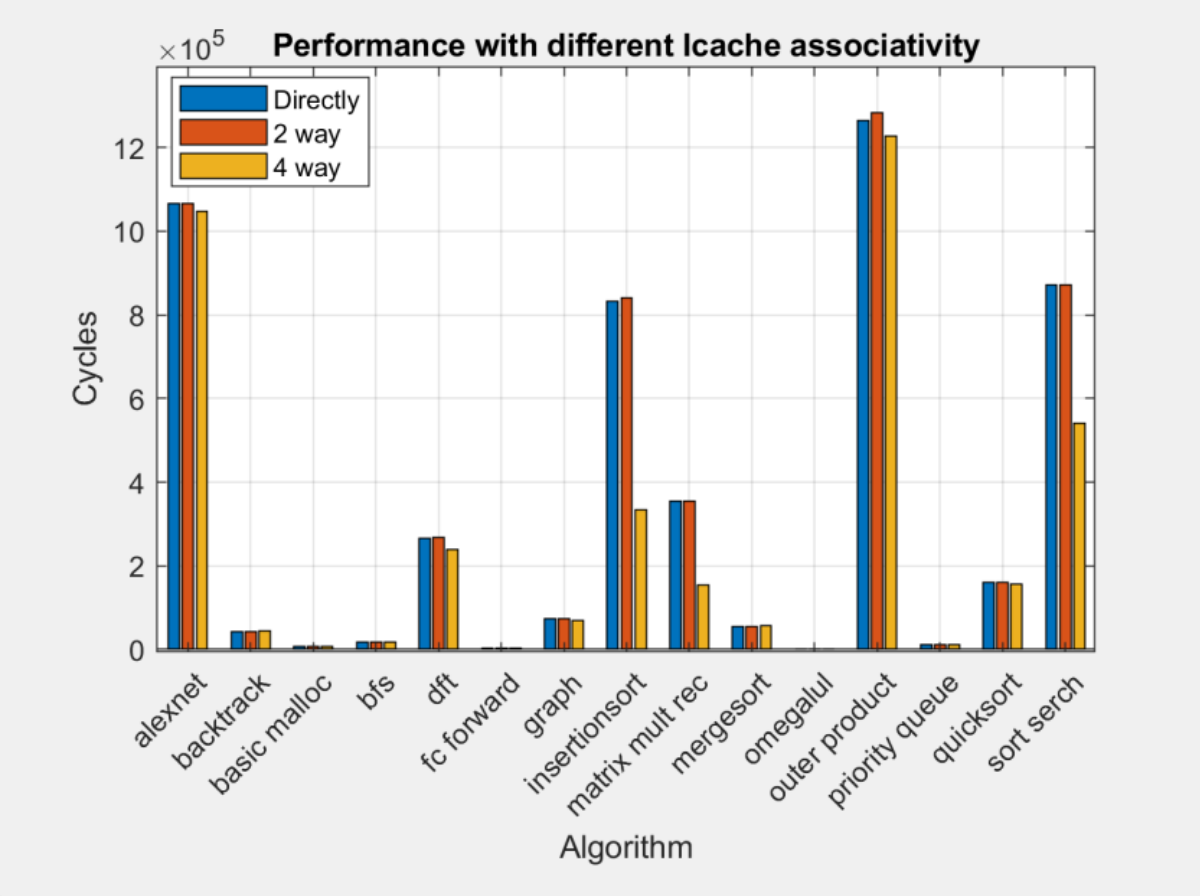
4.4 LSQ with forwarding

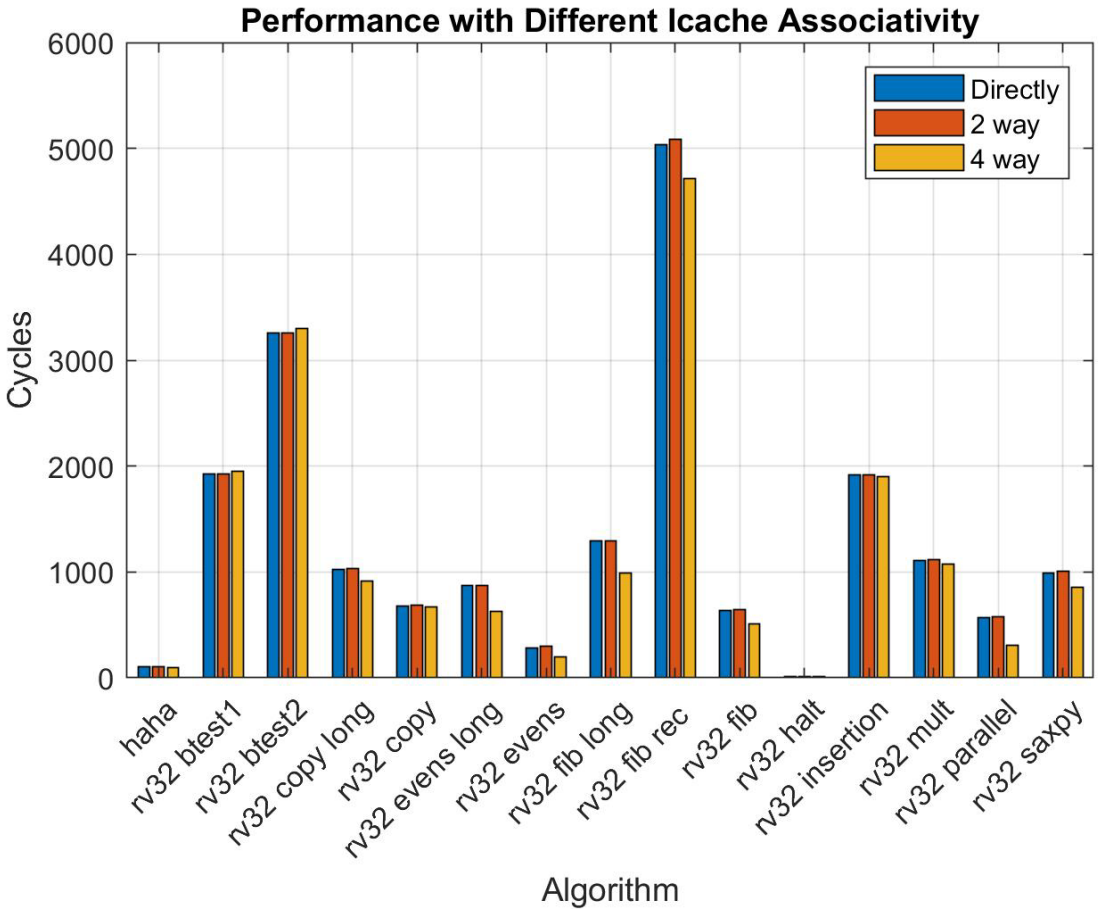
Our LSQ supports speculation on load dependencies and forwarding. With this feature, the load instructions will not be blocked by the previous store such that they can only be executed when there is no previous store. Additionally, some of the load instructions may not need to access data memory because the data will be forwarded inside the LSQ. This avoids much unnecessary and expensive memory latency. The most intuitive way to measure this is to show how many times the data is forwarded during the execution. Form the result, we can see that some memory heavy case like insertionsort, outer_product, and sort_search benegfits a lot from internal forwarding in load-store queue.



4.5 Cache

To analysize our cache, we test instruction cache with different associativity and size, and data cache with different size. From the result ,we can tell that 4-way associative cache generally has better performance than two-way and direct-mapped cache, especially in the cases that the program flow is more predictable, like insertion sort, matrix multiplication recursion, and sort search.





FUTURE IMPROVEMENT

1. If more time is given, we will definitely implement SMT. SMT is a brand novel way to improve FU utilization and it is a great way to learn more about the real problem in the processor design, like coherence, clock period balancing, and performance tradeoff.
2. Due to the fast pace of development, we do not have enough time to take a look at our design with a high-level analysis. In the future, we will focus more on the critical path in our design and try to find out which path extends through several modules and impair the clock period greatly.
3. For many buffers, the sizes are relatively large because we want to simplify the design and take care of the structure hazard after we implement the function. Sadly we were always hurrying to the next part till the end.
4. Our LSQ is not fully optimized so it introduces unnecessary latency. We will eliminate this extra latency by writing more advanced code.
5. Our data cache is developed based on the given instruction cache. The coding convention is awesome. However, unfortunately, we do not really know how to write

code liek that. If possible, we want to develop our own cache with the traditional FSM technique.

6. Inside ROB and RS, there is some redundant information that is not necessary for the proper functioning of the pipeline. We will try to remove them from the entries to improve the performance.
7. Our coding convention across several modules is not very similar. The main reason is that we kept improving our coding technique to write more clear and efficient code. Although we have completely rewritten ROB and RS before, we still need to optimize the code more.

ENCOUNTERED PROBLEM

1. The most troubling problem to us is that three members out of five wrote 0 lines of HDL before. Considering the difficulty of this project, it is hard for our team members to learn what they need to write working code and catch up with the fast pace of development. Besides, there are also three members who have not taken computer organization or equivalent courses before, which makes it more difficult to apply the lecture content in the real project.
2. Before milestone 1, we had an extremely weird bug. Our ROB cannot be synthesized without a totally irrelevant assign statement. This assign statement connects the head pointer of the ROB buffer with a debug signal, which means there is absolutely nothing that may affect the ROB itself. However, with this, the synthesis works perfectly fine.
3. One problem that gives us a lot of trouble is that synthesis and test take too long. In last week, we have to wait for 12 hours to fully test our synthesized version.

CONCLUSION

In conclusion, we successfully made a 2-way superscalar processor with out-of-order execution and add-on advanced features. Through this project, we learned a lot about RTL design, verification, performance tradeoff, and basic techniques required for developing HDL in real world environment. Besides, we get to gain deeper knowledge about the lecture content after hands-on analysis and experiment. Although this project is not optimized, we think it still reflects our hardwork and passion to this course.