EECS 470 Final Project Report

Hudson Hall
University of Michigan
hallhu@umich.edu

Owen Park
University of Michigan
owenpark@umich.edu

Zehua Wang
University of Michigan
oscarhua@umich.edu

Weijia Wu
University of Michigan
weijiawu@umich.edu

Enming Xu
University of Michigan
enmingxu@umich.edu

***Abstract*—Our project is an implementation of an R10K-style processor based on the RISC-V ISA. Our processor consists of two "versions": one for single-threaded programs and the other for simultaneous multithreading (SMT). We have designed both versions to be as efficient as possible, trying to achieve the best clock period with the lowest cycles per instruction (CPI) possible. Our single thread version is 2-way superscalar, capable of fetching, dispatching, issuing, and retiring two instructions per cycle. It has a clock period of 7.85ns and an average CPI of 1.05. Our SMT processor is capable of fetching four instructions, dispatching and issuing two instructions, and retiring four instructions per cycle. It has a clock period of 9.34ns and an average CPI of 1.14. This report covers our architecture design, components, and advanced features we implemented in greater detail, followed by an analysis of how each feature affected the performance of our system.**

I. Introduction

A large portion of our semester was spent designing and implementing a fully functional and well-optimized single thread processor. From our base design, we would then make the architectural jump to SMT. This report will follow similarly to our engineering process. It will begin by covering the base features and architecture of our single thread processor. We will then cover the advanced features and improvements we made to increase single-threaded performance. Finally, it will conclude with the changes made to make SMT fully functional.

II. Single Thread Design

This section will examine the design and implementation of our single thread processor. We managed to achieve a synthesized clock period of 7.85 ns with an average CPI of $\approx$1.05. The critical path is each stage in *mult.sv*.

A. Design Overview

We have designed an out-of-order 2-way superscalar R10K-style processor, with advanced features to improve performance such as a tournament branch predictor, instruction prefetching, a non-blocking 4-way set associative data cache, early tag broadcast, a return address stack, and a load buffer-/store queue with internal forwarding. A high level overview of the flow of data in our processor can be seen in [2, Fig. 1].

B. Instruction Fetch

Since we are 2-way superscalar, we fetch two instructions (or one word) at once which conveniently fits within the one block of data that our memory module can provide in a cycle. Once an instruction is fetched, the next PC is determined based on the instruction type. For conditional branches predicted not taken and non-branch instructions, the next PC will be PC+8. Conditional branches predicted taken and *jal* instructions will change the next fetch PC to current PC + instruction immediate. *jalr* instructions will use the top of the return address stack or the BTB output as the next PC to fetch, depending on whether the instruction is a function return or a function call.

C. Instruction Cache

We have implemented an instruction cache (Icache) with a victim cache. Our Icache is 32 lines direct-mapped, with a 4-line fully associative victim cache. It is also non-blocking, and can continuously prefetch to achieve a low latency for the next instruction fetch.

D. Branch Target Buffer

Our branch target buffer (BTB) is a 2-way set-associative 64-line specialized cache used for enhancing the efficiency of *jalr* instructions. The branch target buffer module will output the predicted target address to the fetch module for the *jalr* instructions. We have chosen to update the BTB on instruction retire.

E. Reservation Station

Our reservation station (RS) has 16 entries and is implemented as an array of RS entries. The RS will allocate new entries for dispatched instructions if no structural hazards are present and will issue instructions that are ready to go. The RS also watches the Common Data Bus (CDB) to identify if renamed registers have received new data, and will issue them if they resolve all dependencies and "wake up." A priority selector for each functional unit selects from these instructions. Our processor is capable of issuing as many instructions as possible, limited only by the number of woken-up instructions and functional unit availability.

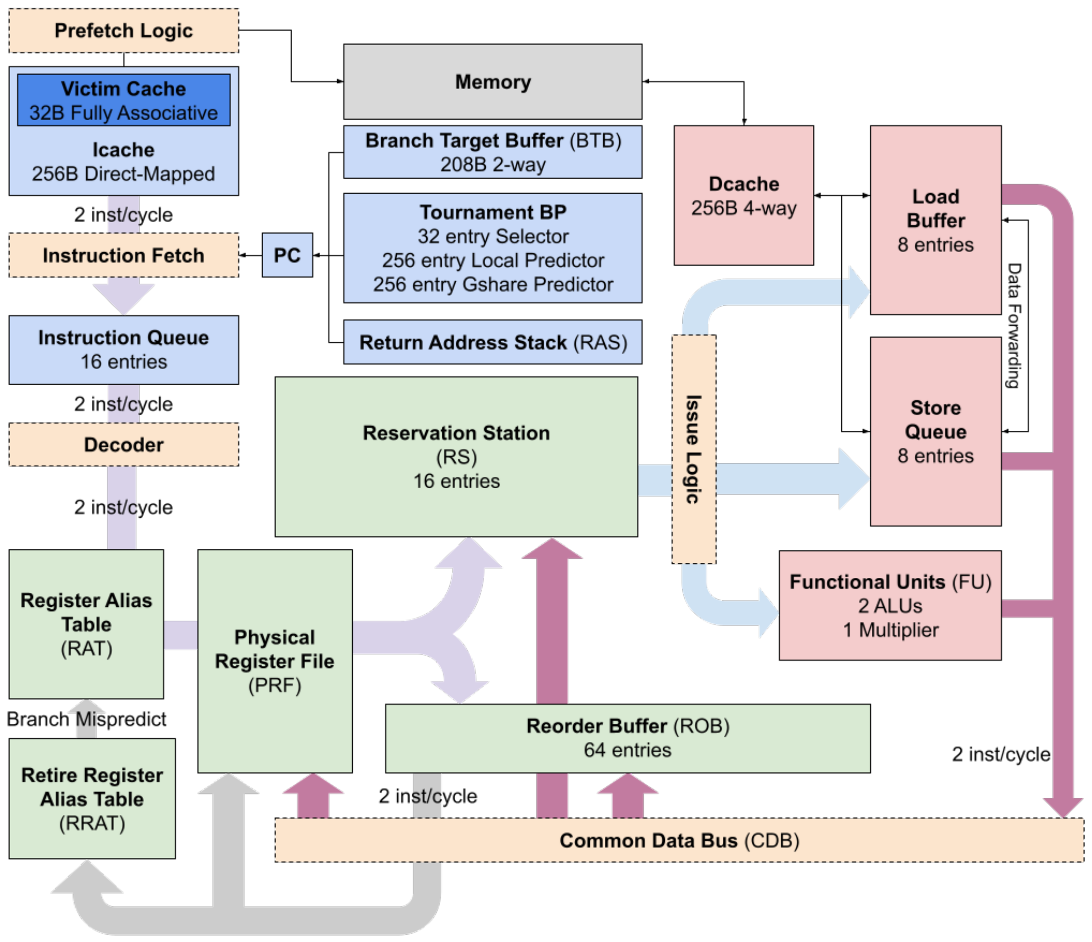

Fig. 1. Detailed single thread processor design.

F. Reorder Buffer

Our reorder buffer (ROB) has 64 entries and is implemented as a circular buffer. Since we are 2-way superscalar, we keep track of the head, head+1, tail, and tail+1. The ROB head will move based on how many instructions we retire each cycle, and the tails will move based on how many instructions we dispatch each cycle. The ROB, along with the rest of our processor, is capable of receiving zero, one, or two instructions on dispatch. It is also capable of retiring zero, one, or two instructions.

G. Register Alias Table and Retire Register Alias Table

Our Register Alias Table (RAT) and Retire Register Alias Table (RRAT) are critical components for supporting out-of-order processing as they handle register renaming so that we can deal with data dependencies. We have 32 architected registers and 96 physical registers. The RAT keeps track of the mapping between architected and physical registers, as well as the states of each architected register to send to RS. The RRAT is only updated with values that have been retired. When a branch misprediction occurs, the system needs to be nuked and the RAT copies its data from the RRAT.

H. Physical Register File

Our physical register file (PRF) has 96 entries and includes a free list to use for register renaming. A priority selector is used to select a free physical register number (PRN) and on dispatch, the free PRN assigned to the dispatched instruction will be made not free. Upon instruction execution, the PRN entry will be marked valid and its value will be updated. Finally, when an instruction retires, the PRN entry that it overwrote on dispatch will be made free.

III. Advanced Features

A. 2-Way Superscalar

Our processor is 2-way superscalar, meaning it is able to fetch, dispatch, issue, execute, and retire two instructions in one cycle. This allows our processor to exploit more instruction-level parallelism. Executing multiple instructions in parallel helps reduce our clock per instruction greatly.

B. Early Tag Broadcast

We have implemented early tag broadcasts for our processor. Without it, when the CDB broadcasts the physical register number it writes to, the reservation station will first clock it in, and on the next clock cycle, the instruction will be ready to issue. By implementing early tag broadcast, the instruction will be ready to issue in the cycle CDB broadcasts so that we don't need to wait one more cycle, helping to reduce CPI.

C. Instruction Prefetching

Given that we can only make one memory request each cycle and we want to give load/store instructions priority, we have to wait quite a few cycles if the Icache misses. To improve this, we implemented instruction prefetching. For our processor, we will prefetch as many instructions as possible until the address for the prefetching has a conflict with the PC requested by the fetch module in the Icache.

D. Non-Blocking 4-Way Set-Associative Data Cache

For the data cache (Dcache), we have implemented a non-blocking, write-back, write-allocate, 4-way set-associative cache. We have implemented miss status handling registers (MSHR) for the Dcache so that we can continuously accept loads when an outstanding miss occurs. We use pseudo least recently used bits (LRU) for each set in Dcache.

When a load or store instruction is trying to access memory, our processor will first check whether the access is a cache hit. If it is a cache hit, we update our pseudo LRU bits and dirty bits of the cache line accordingly. If it is a miss, then the processor will go to MSHR entries to check whether there is an MSHR hit. If MSHR hits, we append the access to the MSHR entry. If MSHR misses, the Dcache will allocate an MSHR entry for the memory access and record the requesting load/store in the MSHR entry. When data comes back from memory, Dcache will update the data according to all pending stores, broadcast the updated data to all pending loads, and write the updated memory block to Dcache.

E. Tournament Branch Predictor

We have implemented a tournament branch predictor for improved branch prediction accuracy. We are using a 256-line local predictor with 8 bits of local history, as well as a 256-line gshare predictor with 8 bits of global history. Both the local and gshare predictors have 2 bits for prediction state, initialized to weakly not taken for local and weakly taken for gshare. The tournament predictor has 32 selectors, all initialized to weakly global. All predictors are updated when a branch instruction retires, based on the retired PC in the ROB entry for local and the current state of branches for global. If both local and global were found to predict the same result, then the tournament predictor is not updated. However, if they differed, the tournament predictor would change to reflect the correct prediction. The tournament branch predictor sends a branch prediction to the fetch stage to determine the predicted PC for the next instruction fetches, greatly reducing the performance impact of branches if they are predicted correctly.

F. Return Address Stack

The return address stack (RAS) exploits the fact that we can tell whether a *jalr* instruction is a function call or a function return for a C program by checking whether the destination register is zero register or not. The implementation of RAS makes the processor capable of predicting *jalr* target addresses better and makes recursive function calls more efficient.

In the fetch stage, once we have fetched a *jalr* instruction from memory, we will decide whether it is a function call or a function return. For function calls, we will allocate an entry in the RAS and increment the head of the stack. For function returns, the fetch module will use the top of the RAS as the next fetch PC and decrement the head of the stack.

G. Load Buffer and Store Queue

In our processor, we have implemented a load buffer (LB) and store queue (SQ) with forwarding capabilities. Load instructions will be dispatched to both the RS and LB. Once all operands are ready, the load instruction can be issued and obtain forwarding data from the SQ. Store instructions will directly be appended to the tail of the SQ without entering the RS because they must be executed in order.

In the LB, each entry records the stores that the load instruction can depend on. When the RS issues a load instruction, the LB will send the calculated address and the size of the load instruction to the SQ in order to get the forwarding data back. Also, load instructions can obtain forwarding data when store instructions retire.

When all store instructions that a load instruction is waiting on are resolved and there is no forwarding data from the SQ, the load instruction will access memory for data.

Store instructions will be appended to the tail of the SQ when dispatched. Once the RS issues a load instruction, the SQ will use the load address and output the store entries which the load instruction is still possibly dependent on. Also, store instructions will only access memory after it retires and is the head of the SQ to make sure store instructions do not change the memory speculatively and are executed completely in order.

IV. Simultaneous Multithreading Design

Several changes were made to make the jump from our single-threaded processor toward our SMT processor. The following section highlights some components that underwent significant change. In the end, we achieved a clock period of 9.34 ns with an average CPI of $\approx$1.14. The critical is the path from *alu.sv* $\rightarrow$ CDB $\rightarrow$ instruction wake up in *rs.sv* $\rightarrow$ and issue back to *alu.sv*.

A. Design Overview

From a high-level perspective, the design of our SMT processor does not differ extremely from our single thread processor. There are only two new structures necessary: a dispatch arbiter to arbitrate which instructions from each thread should be dispatched, and reservation sets which threads can use to achieve synchronization (locks). Additionally, the Icache was adapted to handle two concurrent threads requesting instructions. Other than these additions, most of the changes were simply expanding our existing modules to support taking instructions from two threads and adding logic to distinguish instructions from each thread. These structural changes can be seen in [5, Fig. 2].

B. *Fetch and Instruction Cache*

For our SMT processor, the fetch module is almost identical to the one in our single thread processor, except that we now have two fetch modules for each thread which both will request instructions from the Icache. The Icache is capable of processing both requests at the same time and utilizes a prefetch buffer for each thread (each holding up to 8 instructions) to further enhance performance. Upon a cache miss, the Icache will go to memory for the instructions. It will additionally start prefetching for that thread. While this is occurring, the other thread is still able to fetch from the Icache, hopefully still getting hits. In the case that both instructions are in the “missed” state — either going to memory for the missed instruction or for further prefetching — the Icache will alternate between the two threads every cycle to prevent thread starvation, prioritizing the thread that has just missed. Each fetch module has its own instruction queue to store fetched instructions.

C. *Dispatch Arbiter*

We now have a dispatch arbiter that sits between the decoder and the out-of-order system. The dispatch arbiter is responsible for taking up to four decoded packets (two from each thread) and selecting up to two packets to send to the rest of the system. The dispatch arbiter takes into consideration structural hazards for each thread, as well as their instruction queue availability when making the decision to dispatch. If possible, it will try to dispatch one instruction from each thread. In addition, there is extra logic implemented to prevent starving by simply rotating priority in the event we have to make an arbitrary decision to choose one thread over the other.

We attempted to make a more complex dispatch arbiter that also takes branch mispredicts and Dcache misses into consideration. However, the new dispatch arbiter didn’t provide better scheduling for our processor. Hence, we decided to stick to the original simple dispatch arbiter.

D. *Control and Status Registers*

To run multithreaded programs on our processor, we have to add support for one instruction, which is *csrr*.

The CSR instructions stand for control and status registers. Our SMT test programs uses our own custom C runtime file which reads the value of *mhartid*, to determine which thread the program should jump to. Since we only need to support *csrr* for this case, we treat the *csrr* instruction as a normal *addi* instruction, with the source register set to zero register, and the immediate set to 0 or 1, depending on the hardware executing the instruction.

E. *Locks*

In order to implement locks in the SMT processor, we added support for two more instructions, which are load reserved (*lr.w*), and store conditional (*sc.w*).

For locks, load reserved will reserve an address for its thread, and store conditional will write success (0) to the destination register only when the address it stores to is reserved by a previous load reserved from the same thread and is not invalidated by the other thread. Otherwise, it will write an error code to the destination register, which is 1 in our implementation. It will also invalidate the reserved address of the other thread if the other thread reserved the same address as the store instruction stores to. The main program we used in testing was *inc50.c* which is shown in [4, Code 1]. The goal of *inc50.c* is for each thread to increment the global *counter* by 50. With load reserved and store conditional implemented, we can achieve pseudo mutual exclusion in our hardware and pass *inc50.c*, successfully incrementing *counter* to 100 with two threads running at the same time.

F. *Physical Register File*

The PRF must be expanded to allow for two threads. Our first approach with the PRF was to let both threads share the same pool of $64 + 2 * \text{ROB Size} = 192$ physical registers. This seemed like the obvious solution for us since this would allow processes such as instruction wake up to be transparent to thread information. By simply broadcasting the PRN on the CDB, the proper instructions would get the data forwarded correctly since every PRN was unique. This approach would prove inefficient, greatly increasing the clock period of *prf.sv* to 19 ns. Seeing this, we then pivoted to use two separate PRF’s for each thread, each with $32{+}\text{ROB Size} = 96$ physical registers. Since each PRF would contain PRNs 0 through 95,

```c
#define LOAD_LOCK(lock, lock_status) asm volatile ("
    lr.w %0, (%1)" : "=r" (lock_status) : "r" (lock)
    );
#define STORE_COND(lock, write, success) asm
    volatile ("sc.w %0, %1, (%2)" : "=r" (success) :
     "r" (write), "r" (lock));

volatile int lock=0;
volatile int counter = 0;
int main(int argc, char * argv[])
{
    int local_count=0;
    int fail;
    int in_use;
    while (local_count<50)
    {
        LOAD_LOCK(&lock, in_use);
        if (in_use) continue;

        STORE_COND(&lock, 1, fail);
        if (fail) continue;
        counter++;
        lock=0;
        local_count++;
    }
    return 0;
}
```

Code 1. Program code of inc50.c for multithread testing.

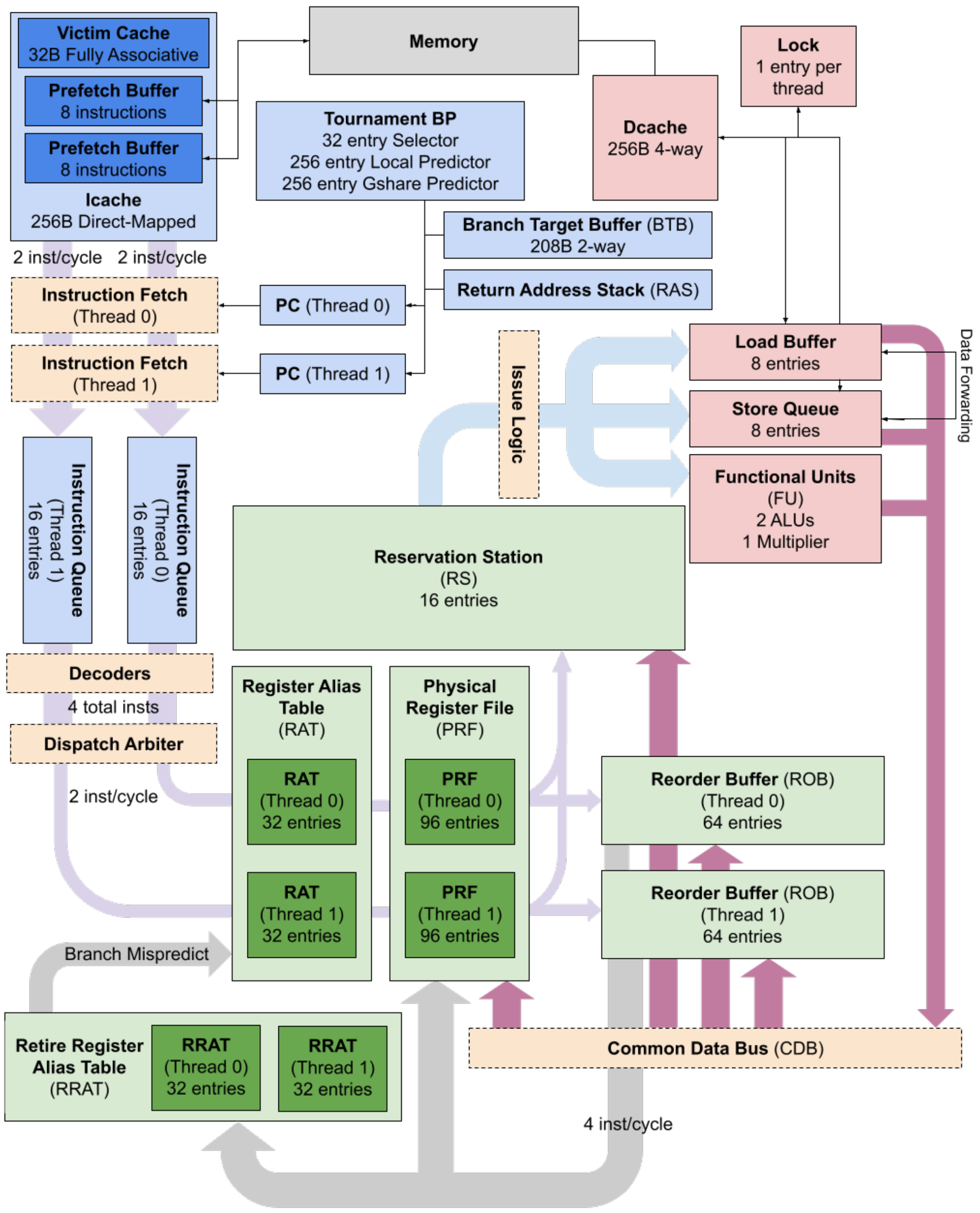
Victim Cache
32B Fully Associative
Prefetch Buffer
8 instructions
Prefetch Buffer
8 instructions
Icache
256B Direct-Mapped
Memory
Lock
1 entry per thread
Tournament BP
32 entry Selector
256 entry Local Predictor
256 entry Gshare Predictor
Dcache
256B 4-way
Branch Target Buffer (BTB)
208B 2-way
Return Address Stack (RAS)
2 inst/cycle
2 inst/cycle
Instruction Fetch (Thread 0)
PC (Thread 0)
Instruction Fetch (Thread 1)
PC (Thread 1)
Issue Logic
Load Buffer
8 entries
Store Queue
8 entries
Data Forwarding
Functional Units (FU)
2 ALUs
1 Multiplier
Instruction Queue (Thread 1) 16 entries
Instruction Queue (Thread 0) 16 entries
Reservation Station (RS)
16 entries
Decoders
4 total insts
Dispatch Arbiter
2 inst/cycle
Register Alias Table (RAT)
RAT (Thread 0) 32 entries
RAT (Thread 1) 32 entries
Physical Register File (PRF)
PRF (Thread 0) 96 entries
PRF (Thread 1) 96 entries
Reorder Buffer (ROB) (Thread 0) 64 entries
Reorder Buffer (ROB) (Thread 1) 64 entries
Branch Mispredict
Retire Register Alias Table (RRAT)
RRAT (Thread 0) 32 entries
RRAT (Thread 1) 32 entries
Common Data Bus (CDB)
4 inst/cycle

Fig. 2. Detailed simultaneous multithreading processor design.

this meant that two in flight instructions from separate threads could share identical destination and op PRNs. To prevent conflicts, we simply send a bit along with PRN data so that the system can distinguish which thread the PRN corresponds to. This change resulted in a *prf.sv* clock period of 8 ns, a much more reasonable time.

G. Expansion

Certain modules were expanded to accommodate two threads. We now have two ROB's, two RAT's, two RRAT's, and two PRF's. There is one ROB for each thread, which means that our system is now capable of retiring four instructions (two from each thread's ROB) every cycle. There is also one RAT for each thread, which also means that we need two RRAT's. This is so that we can account for the additional registers and rename each thread accordingly. The *prf.sv* and *rat.sv* modules were expanded to handle both threads in the existing modules. Under the hood, however, each contains two separate tables for each thread (a detailed dive of *prf.sv* was given in IV-F). The *rob.sv* and *rrat.sv* modules were simply duplicated in *cpu.sv* with minimal logic to parameterize which module corresponded with each thread. The reservation station and functional units have no major architectural differences. The largest change to each was handling proper squashing of a certain hart.

V. What Works

All of our major components synthesize and work exactly as expected, and we successfully pass all test cases for single-threaded performance. We managed to achieve a synthesized clock period of 7.85 ns for our single-threaded processor with an average CPI of ≈1.05.

For multi-threaded performance, our processor also worked as expected, producing the correct output against ground truths we created and synthesizing at a 9.34 ns clock period with an average CPI of ≈1.14. We created ground truths by modifying crt.s to run two different programs back to back in single thread mode and then compared these with our multithreaded output. During tests, we found that there would sometimes be stack overflow when running certain large programs together, but multi-threaded works if given enough stack space for every program, and our modules do not appear to have any problems. Also, we can successfully add up to 100 when running two threads of *inc50.c* program [4, Code 1], as expected.

For our branch predictor, we tested global versus gshare to use in a tournament predictor and found that gshare provided better prediction accuracy, so we decided to use gshare in our final version and did not include global. We also tried speculatively updating our branch predictor but found that it produced poor results and decided to update on retire.

VI. Evaluation & Testing

A. Branch Prediction

We were able to achieve an 84.00% branch prediction accuracy for all programs and a 92.14% accuracy for larger programs (programs with greater than 4000 conditional branches). Ultimately, our design featured a tournament branch predictor, selecting from a local predictor with a local history table, and a global gshare predictor.

We found that it was most optimal to update on retire for both our local predictor and global predictor, although the differences in execute and retire were minimal. We also performed testing for speculatively updating our predictor, although the implementation did not integrate well with our current design, as fixing the speculative update was difficult. Thus, in our single-threaded design, we elected to update our entire predictor on retirement. However, in our multithreaded design, we were able to update up to 4 instructions at once, due to having two ROBs. Thus, to simplify the design, we decided to update the predictor on execution for our multithread design.

We also tested the performance differences in initialization, testing whether it was better to initialize our two-bit saturating counter state to weakly taken (*2'b10*) or weakly not taken (*2'b01*) and for our tournament selector, we tested weakly global versus weakly local.

Lastly, we tested different sizes for our tournament predictor but noticed that the size differences provided minimal differences in performance once the tournament selector reached a sufficient size. It should also be noted that all tests were performed without multithreading enabled.

In our testing, we first compared the overall prediction accuracy, then we considered the prediction accuracy for large programs only, shown in the table below in [6, Tab. 1]. Although we considered both these statistics, we ultimately decided to prioritize branch prediction accuracy for large programs, as we believed this would yield more conclusive results.

A large part of our testing was performed to analyze the most optimal way to update our predictor. First, we performed individual tests on both the local predictor and global predictor, updating them at different stages: (1) Updating speculatively at fetch, (2) Updating on execution through the CDB, and (3) Updating on retire.

a) Testing the Global Predictor: After testing different global predictor schemes, we found that the best scheme for the gshare predictor was to initialize to weakly taken, with updating on execute or retire, as these two schemes produced the best results. When updating speculatively, these yielded the worst results, although we believe that under ideal conditions, this may not be the case. It was difficult to accurately fix wrong speculations, especially with the shift registers used for branch history. In our speculative implementation for gshare, we kept track of each branch instructions index in the gshare table and then would fix the entry on retire if there was a misprediction. However, we could not find an effective way to update the global history shift register, which we believe was a contributor to the poor results yielded by speculatively updating the predictor. As a result, we decided to scrap the idea of updating speculatively moving forward. The results for the global predictor are shown in the table below [6, Tab. 2], showing the prediction accuracy for each scheme with all of our testbenches and the prediction accuracy for just the large programs in [6, Tab. 1].

TABLE I
Large Program Conditional Branch Count

Program	**Conditional Branches**
dft	4376
insertionsort	10252
sort_search	10588
outer_product	10856
matrix_mult_rec	11364
alexnet	14743

TABLE II
Performances of Different Global Predictor Schemes

Global Predictor Scheme	**Overall**	**Large**
Weakly Taken, Update Speculatively	69.90%	76.04%
Weakly Taken, Update on Execute	80.78%	89.69%
Weakly Taken, Update on Retire	81.61%	89.33%
Weakly Not Taken, Update Speculatively	35.83%	27.42%
Weakly Not Taken, Update on Execute	62.46%	88.40%
Weakly Not Taken, Update on Retire	63.94%	87.70%

b) Testing the Local Predictor: Similar tests were done for the local branch predictor. After analyzing the results shown in [7, Tab. 3], we concluded that the best configuration for our Local predictor was to initialize to weakly not taken, with updating occurring on retire. Again, we noticed fairly similar results for schemes updating on execute vs. schemes updating on retire, although retire ultimately performed the best.

c) Testing the Tournament Predictor: After finding the most optimal configurations for our local and global predictors, we implemented them both using a tournament selector. We tested different sizes of our tournament selector as seen in [7, Tab. 4] and noticed that the size of the tournament selector did not influence the performance in the way that we expected. Ultimately, we decided to use 32 lines, as it gave us the best performance out of all schemes and did not impact our clock period.

d) Evaluating the Tournament Predictor: It is evident from the chart [7, Fig. 3] and table [7, Tab. 5] that the tournament predictor was successful in integrating the local and global predictors. In some programs, the local predictor performed better than the global predictor, and in other programs, the global predictor performed the best. The tournament selector successfully balances these two predictors and performs the best in both branch prediction accuracy and instructions per cycle (IPC) in almost all cases.

B. Instruction Prefetching

We choose to prefetch at most 16 instructions in our Icache module. This is mainly because the memory latency is 13 cycles so any value below 13 could result in an idle Icache. For programs that have a lot of loops, the size of the prefetcher should be small so that all instructions of the loop are still in the cache when the instruction at the end of the loop is executed. For programs that have a lot of if statements, the size of the prefetcher should be large so that the branch target is prefetched and can be used immediately. We chose the prefetch size to be 16 and we have an overall speedup of 2.587. This speedup can be seen in [7, Fig. 4] where ”standard” is our processor.

C. Non-blocking Data Cache

A non-blocking data cache can respond with an ACK to the load buffer under a cache miss and deal with other requests before it receives value from the memory. It improves the performance when the program has large and continuous loads, such as *matrix_mult_rec*. Implementing a non-blocking data cache results in an overall speedup of 1.182. This can be seen in [8, Fig. 5].

TABLE III
PERFORMANCES OF DIFFERENT LOCAL PREDICTOR SCHEMES

Local Predictor Scheme	Overall	Large
Weakly Taken, Update on Execute	72.79%	88.57%
Weakly Taken, Update on Retire	72.80%	88.78%
Weakly Not Taken, Update on Execute	80.98%	88.56%
Weakly Not Taken, Update on Retire	81.15%	88.79%

TABLE IV
PERFORMANCES OF DIFFERENT TOURNAMENT PREDICTOR SIZES

Size	Overall	Large
16 Lines	84.31%	92.06%
32 Lines	84.00%	92.14%
64 Lines	83.85%	92.05%

TABLE V
BEST IPC OF EACH BRANCH PREDICTOR FOR LARGE PROGRAMS.

Program	Local IPC	Global IPC	Tournament IPC
dft	0.6764	0.6959	0.6985
insertionsort	0.9979	1.0550	1.0636
sort_search	0.7678	0.7407	0.7554
outer_product	1.0171	1.0311	1.0311
matrix_mult_rec	1.1429	1.1856	1.2357
alexnet	0.9170	0.7792	0.9217

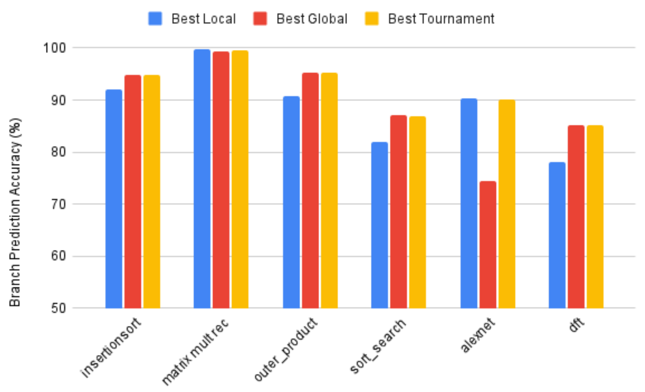

Fig. 3. Prediction accuracy of each branch predictor for large programs.

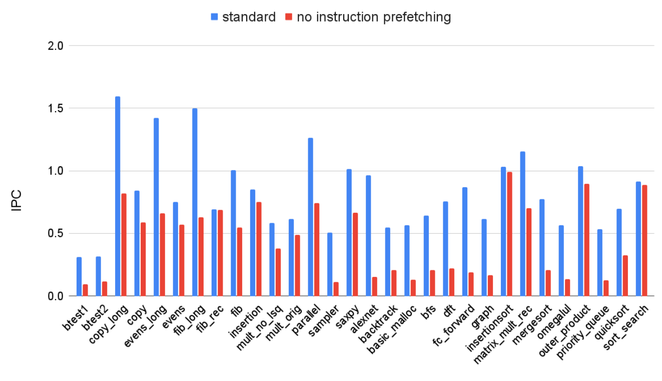

Fig. 4. Performance of instruction prefetching.

D. Early Tag Broadcast

Initially, we do CDB selection and broadcast, RS selection and issue, and alu execution in one cycle, but it becomes our critical path. Early tag broadcast allows the RS to know which PRN numbers are going to be on CDB in the next cycle and the result of CDB is removed from the input of the priority selectors in the RS. It reduces the clock period by 0.24 ns. One alternative is to do CDB broadcast and issue in different cycles, but there would be stalls between back-to-back dependent instructions. The performance is much worse when the program has a long dependency chain. Compared with this option, early tag broadcast results in a speedup of 1.203 as shown in [8, Fig. 6].

E. Return Address Stack

For *jalr* instructions, we use the destination register to differentiate between function calls and function returns. If the instruction writes to architecture register 0, we view it as a function return, otherwise it is a function call. For *fib_rec* and *omegalul*, BTB performs better than RAS because speculative and squashed function calls and returns mess up the RAS. For *alexnet*, *fc_forward*, *matrix_mult_rec*, and *outer_product*, the implementation with RAS is slightly better. Overall we get a speed up of 1.010 as shown in [8, Fig. 7].

F. Data Forwarding from Stores to Loads

In our processor, if the latest dependent store for a load instruction is still in the store queue when the load is issued, the store queue will forward the value to the load buffer. This greatly improves the performance when the program often stores to and then reads from the same address, which happens a lot when we specify no optimization during the compilation. By implementing the data forwarding, we have an overall speedup of 1.153 visible in [8, Fig. 8].

G. Different Sizes of RS, ROB, LB, and SQ

The combination of RS Size = 16, ROB Size = 64, LB Size = 8, and SQ Size = 8 is enough for not having structural hazards that hurt performance. We can observe from the graph that *saxpy*, *insertionsort* and *outer_product* are slowed down when RS Size = 8, ROB Size = 32, or LB Size = 4. *saxpy* and *matrix_mult_rec* are slowed down when SQ Size = 4. However, it is not the case that a larger buffer/queue size is better. Since we are not able to choose the oldest instruction in the RS to issue, a larger buffer/queue size will let more later instructions come into the out-of-order system and the instructions in the longest dependency chain can not be issued one by one as soon as possible. The performance for *outer_product* is worse when RS Size increases from 16

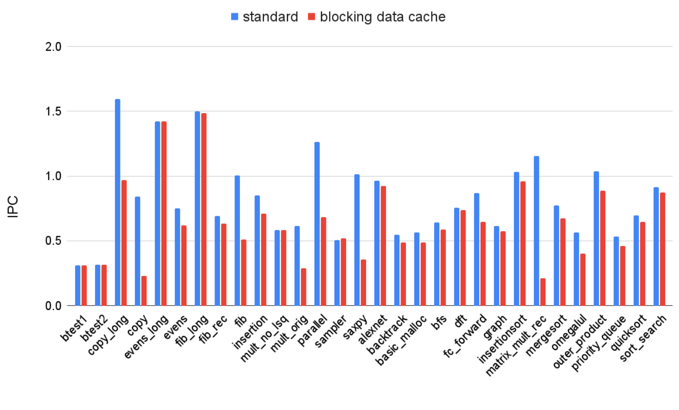

Fig. 5. Performance of a non-blocking data cache.

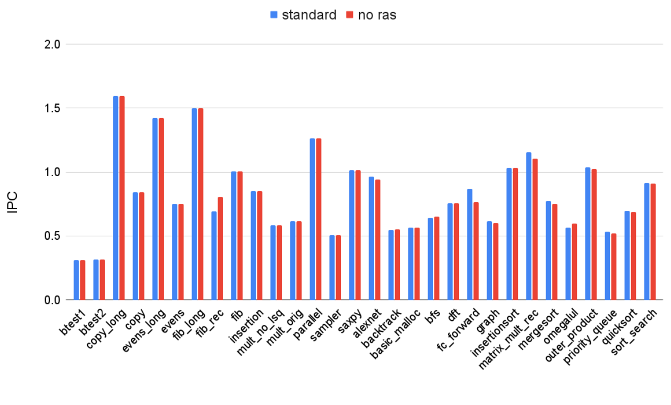

Fig. 7. Performance of a return address stack.

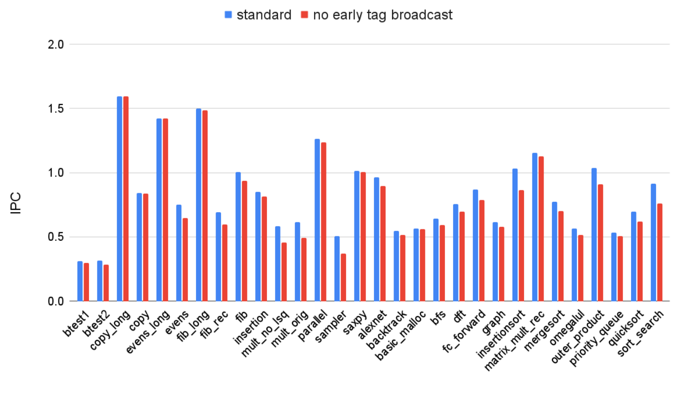

Fig. 6. Performance of early tag broadcast.

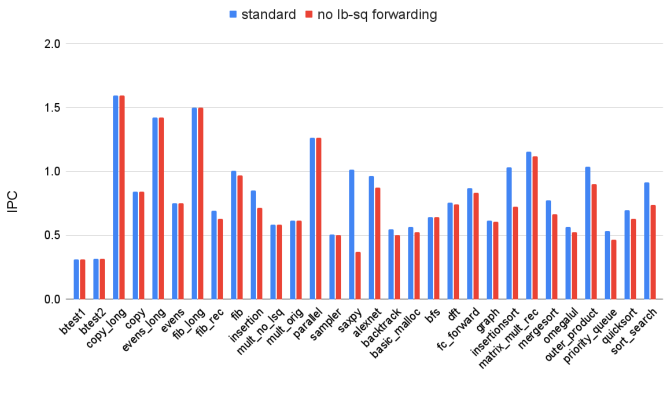

Fig. 8. Performance of data forwarding from stores to loads.

to 32 or ROB Size increases from 64 to 128. The performance for *fib* is worse when LB Size increases from 4 to 8. Overall the performance with the parameter RS Size = 16, ROB Size = 64, LB Size = 8 and SQ Size = 8 is the best. This data is shown in [9, Fig. 9, 10, 11, 12].

H. *Homework Testcases*

The goal of Homework 4a [9, Code 2] was to achieve an IPC greater than 1, which was achieved by having two groups of instructions that have their own dependency branches. Lines 2 through 4 are one dependency branch, and lines 5 through 7 are another dependency branch. The goal of Homework 4b [9, Code 3] was to achieve back-to-back execution of dependent instructions, which was achieved by looping through instructions with subsequent dependencies.

I. *Simultaneous Multithreading Performance Gain*

We saw significant improvement from our single thread to our multi-thread performance, as shown in [10, Tab. 6]. We saw an average speedup of 1.1895. Our non-SMT time was calculated by immediately halting the second thread and fetching from one program until completion. We then summed the individual times for these programs and compared them to the time spent when SMT was enabled. We decided to solely use statistics from our SMT processor instead of our single-threaded processor because our SMT processor lacks some performance optimizations that were implemented in the single-threaded processor. We decided to run programs in pairs, swapping each program's thread in subsequent tests to

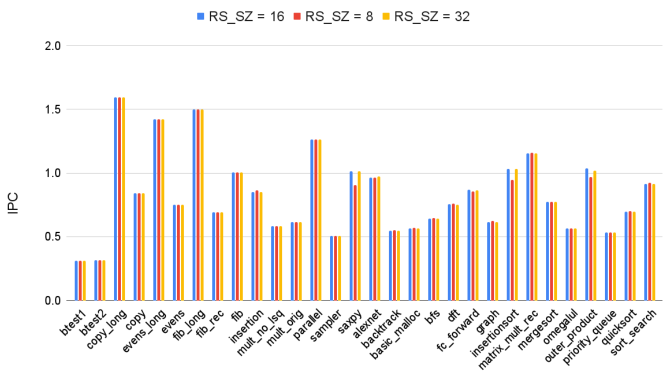

Fig. 9. Performance of different RS sizes.

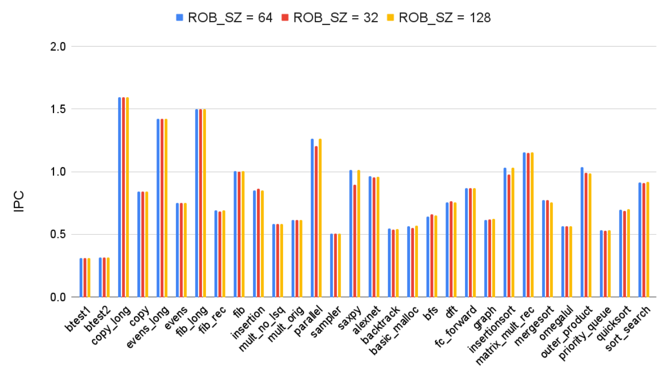

Fig. 10. Performance of different ROB sizes.

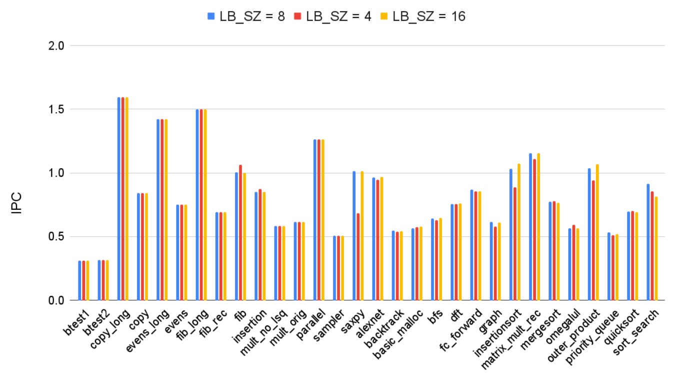

Fig. 11. Performance of different LB sizes.

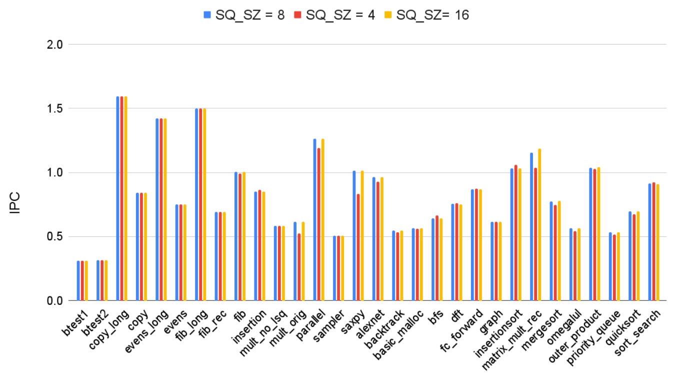

Fig. 12. Performance of different SQ sizes.

```
        li      x1, 0x32
loop:   addi    x2, x1, 1
        addi    x3, x2, 1
        addi    x4, x3, 1
        addi    x5, x1, 1
        addi    x6, x5, 1
        addi    x7, x6, 1
        addi    x1, x1, -1
        bnez    x1, loop
        wfi
```

Code 2. Homework 4a program code

```
        li      x1, 0x0
        li      a0, 0x100
loop:   addi    x2, x1, 1
        addi    x3, x2, 1
        addi    x4, x3, 1
        addi    x5, x4, 1
        addi    x6, x5, 1
        addi    x7, x6, 1
        addi    x8, x7, 1
        addi    x1, x8, 1
        ble     x1, a0, loop
        wfi
```

Code 3. Homework 4b program code

TABLE VI
PERFORMANCE GAIN OF SMT

Thread 0	Thread 1	CPI	Time (ns)	Non-SMT Time (ns)	Speedup
matrix_mult_rec	mergesort	1.0429	303578.02	337295.42	1.1111
mergesort	matrix_mult_rec	1.0423	303409.90	337295.42	1.1117
graph	backtrack	1.5888	272186.28	346345.88	1.2725
backtrack	graph	1.6162	276874.96	346345.88	1.2509
mergesort	graph	1.4882	286541.86	360916.28	1.2596
graph	mergesort	1.4797	284916.70	360916.28	1.2667
dft	quicksort	1.2520	994719.34	1197602.82	1.2040
quicksort	dft	1.2398	985015.08	1197602.82	1.2158
outer_product	insertionsort	1.1438	4979518.26	5349531.7	1.0743
insertionsort	outer_product	1.1295	4920900.42	5349531.7	1.0871
backtrack	mergesort	1.4152	220620.14	289652.08	1.3129
mergesort	backtrack	1.4354	223767.72	289652.08	1.2944
outer_product	dft	1.1009	4624729.02	5081268.22	1.0987
dft	outer_product	1.1057	4645006.16	5081268.22	1.0939

make sure that our prefetching behavior was evenly distributed for both programs. Since we saw nearly equivalent performance for pairs of programs, we can conclude that our processor is not biased towards one thread.

It should also be noted that with the ability to run two programs, we must be very careful with the allocation of stack space to each program. The table [10, Tab. 7] shows the maximum stack depth of the larger test benches.

TABLE VII
STACK DEPTH OF RELEVANT PROGRAMS

Program	Stack Depth (B)
matrix_mult_rec	44380
outer_product	20480
dft	572
quicksort	572
insertionsort	492
mergesort	480
graph	300
priority_queue	284
bfs	236

ACKNOWLEDGMENT

We would like to thank Professor Mark Brehob and the rest of the EECS 470 staff — Mustafa Miyaziwala, Bradley Schulz, and Ian Wrzesinksi — for their help throughout the semester.