Chig Bungus CPU

3 May 2024

—

ECE 411 - Dr. Rakesh Kumar

mp_ooo

Richard Bi, Kevin Liu, Bryan Zhang - CA: Pradyun Narkadamilli

Richard Bi, Kevin Liu, Bryan Zhang

Dr. Rakesh Kumar

ECE 411 – Computer Organization & Design

3 May 2024

Chig Bungus CPU Final Report

Introduction

For our final out-of-order processor, we created a parameterizable, superscalar, explicit register renaming (ERR), single-core processor that supports most RISC-V instructions and includes several advanced features such as a pipelined instruction cache, combinational multiplier, a branch predictor and a disambiguated load-store memory structure. As an educational project, our primary focus was to implement as many unique and broad features as possible to explore how various changes to parts of our CPU will impact performance. Due to the nature of our superscalar design, our CPU targeted relatively higher IPC, meaning more useful work is being done in each clock cycle, rather than higher clock frequencies in which there are more clock cycles per unit of time.

We implemented our CPU to work with the RISC-V instruction set architecture (ISA) because the course focused its discussion of CPU architecture in the context of RISC-V, specifically, the RV32I instruction set. On top of its open-source nature, one of the main reasons this ISA was chosen is its simplicity in terms of the development cycle. The RV32I base integer instruction set consists of fixed length instructions (32 bits each) that operate only on signed and unsigned integers. This simplicity allowed us to focus more on the architectural challenges and gave us the flexibility to tackle advanced features we were interested in, such as superscalar.

Additionally, we were provided a shift-add multiplier that allowed the processor to be compatible with the multiplication extension of the ISA. There was also an option to extend the compatibility of our processor to include compressed instructions, division instructions and floating point instructions, but we did not integrate these extensions into our design. Our processor also does not support FENCE and CSR instructions as those are more focused on multi-core and multithreaded tasks that are not relevant to our single-core processor.

Project Overview

The Chig Bungus out-of-order processor is a RISC-V explicit register renaming based processor with several advanced features through which we were able to learn the basics of the CPU design process and exercise what we learned during lectures. We completed the aforementioned advanced features to explore areas of optimization that computer architects utilize when designing their own CPUs and to compete in the MP_OOO competition by optimizing our instructions per cycle (IPC), maximum clock frequency, area, and power. Code is often written during group meetings in the lab or individually if the module can be isolated. The majority of the debugging is done then during group meetings when we integrate new or improved components into the top level. This also ensured that everyone in the group was familiar with the other modules, improving understanding of the processor as a whole. In this report, we discuss how we decided our features, the design choices we made in our implementation, the timeline of our development process, some challenges and improvements that can further optimize our CPU, and finally some closing thoughts on the project as a whole.

Design Description

Detailed discussion of our process will be split into the following sections: CP1, CP2, CP3, and advanced design features. Under each section, we will discuss specific design choices we made when implementing a checkpoint/feature, its testing procedures, and its performance measurements if it applies. For the advanced features, we will also discuss any design trade-offs we made, any improvements or diminishment in performance, and any potential workloads that the feature would be well or ill suited for. We will start with the essential checkpoints required to get the out-of-order processor working before moving on to the advanced features section.

CP1

CP1 was the simplest checkpoint in terms of implementation, so most of the time was spent determining what features our CPU was going to have and a rough plan and timeline going forward. During this week, we made two major design decisions, namely that we were going to implement an explicit register renaming CPU and that it will have superscalar capabilities. We chose ERR over the implicit register renaming from Tomasulo's algorithm since it would provide better long-term performance benefits for power and also made supporting superscalar simpler. Integrating superscalar capabilities into our design was the key focus and feature of our CPU from the start because we believed it would significantly increase IPC, so these are the foundations of our proposal. Below is the design block diagram of our baseline design that we submitted as part of our CP1.

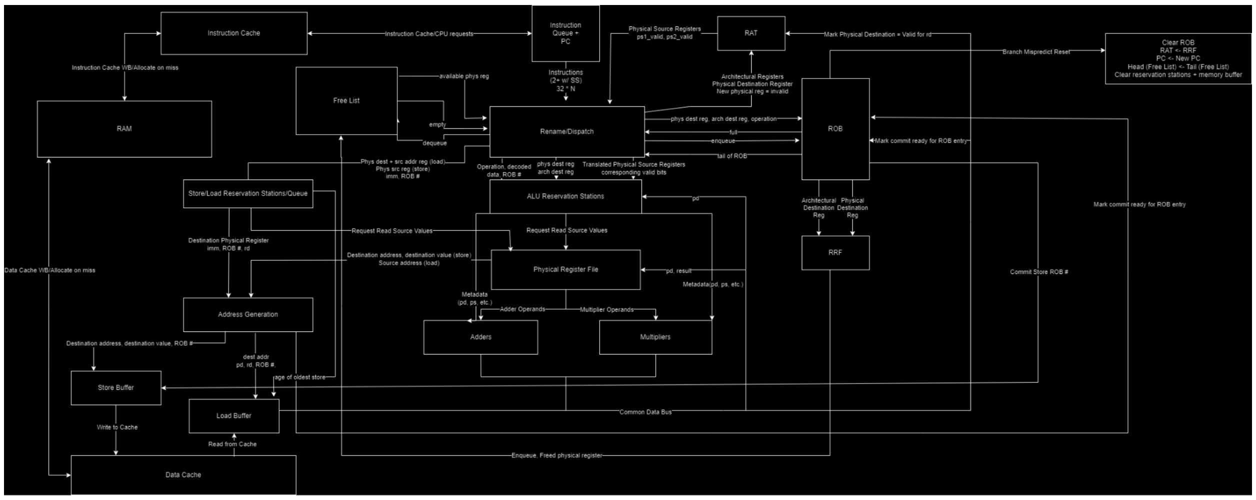

Instruction Cache
Instruction Cache/CPU requests
Instruction Queue + PC
Instructions (2+ w/ SS) 32 * N
Physical Source Registers ps1_valid, ps2_valid
RAT
Mark Physical Destination = Valid for rd
Branch Mispredict Reset
Clear ROB
RAT <- RRF
PC <- New PC
Head (Free List) <- Tail (Free List)
Clear reservation stations + memory buffer
Instruction Cache WB/Allocate on miss
RAM
Free List
available phys reg
empty
dequeue
Rename/Dispatch
Architectural Registers
Physical Destination Register
New physical reg = invalid
phys dest reg, arch dest reg, operation
full
enqueue
tail of ROB
ROB
Mark commit ready for ROB entry
Phys dest + src addr reg (load)
Phys src reg (store)
imm, ROB #
Operation, decoded data, ROB #
phys dest reg
arch dest reg
Translated Physical Source Registers
corresponding valid bits
Store/Load Reservation Stations/Queue
ALU Reservation Stations
pd
Architectural Destination Reg
Physical Destination Reg
Mark commit ready for ROB entry
Request Read Source Values
Request Read Source Values
Destination Physical Register
imm, ROB #, rd
Destination address, destination value (store)
Source address (load)
Physical Register File
pd, result
RRF
Data Cache WB/Allocate on miss
Metadata(pd, ps, etc.)
Commit Store ROB #
Metadata (pd, ps, etc.)
Address Generation
Adder Operands
Multiplier Operands
Adders
Multipliers
Destination address, destination value, ROB #
dest addr
pd, rd, ROB #,
age of oldest store
Store Buffer
Load Buffer
Common Data Bus
Write to Cache
Read from Cache
Enqueue, Freed physical register
Data Cache

Figure 1: Original block diagram of proposed CPU design

Additionally, we implemented two basic components of the CPU: a basic instruction fetch stage and a parameterizable queue. The fetch stage was a naive implementation assuming no control instructions (PC will continuously increment by 4), no superscalar fetch, and it interfaced with the magic memory module (all of these would later be changed to support the full instruction set, advanced features, and realistic memory). In this design, the instruction memory output matched the PC, and incremented with each memory response. The parameterizable queue served as the baseline design for the instruction queue, a FIFO structure where new instructions read from instruction cache or RAM are kept until the dispatch stage pops them off. The queue is an unpacked array of packed logic (registered). The FIFO behavior was realized with a head and a tail pointer, each a register with $\log_2(\text{depth})+1$ width. The head pointer always points at the oldest instruction in the queue, which is popped should a pop be asserted, and the tail points at the next available position where an instruction (or bundle, for superscalar) can be pushed. The width of the head/tail registers is $\log_2(\text{depth})+1$ so it's easy to tell which entries are valid when wraparounds happen (tail goes beyond the end of the queue and comes back to the beginning).

Testing consisted of hooking the fetch stage to the magic memory and running the following cases: pushing without popping was tested by hardwiring the pop signal of the queue to 0 and pushing instructions. We verified the behavior by making sure instructions were arriving into the queue in order, and that the fetch stage was no longer pushing when the queue was full. Popping without pushing was tested by hardwiring the push signal to 0, and trying to pop. Simultaneous push and pop are tested by manually controlling the pop signal using hierarchical references from the testbench. We tested cases of simultaneous push and pop when the queue is empty and when it's full.

CP2

CP2 was significantly more involved in terms of implementation compared to CP1, and the objective was to implement support for all but control and memory instructions. For this week, most of the initial implementation was done individually and in parallel. We divided the key components from our original block diagram and split them into modules (i.e. register alias table, register file, reservation stations, functional units, etc.) that can be implemented individually. Substantial parts of the CPU were implemented with superscalar compatibility in mind by this point, so many of the parts had utilized a SS_FACTOR parameter that was set to one. After the individual modules were believed to be completed, we got together at the lab to put them all together starting with the rename and dispatch module. The rename dispatch module's primary purpose is to facilitate the distribution of instructions given enough resources (free registers, reservation stations, reorder buffer entries, etc.) are available. For example, if there are not enough physical free registers in the free list, the CPU needs to ensure no instructions are dispatched and we don't try to add or update invalid entries to other structures like the register alias table (RAT). The rename and dispatch module begins by requesting the necessary free registers from the free list to remap the RAT for instructions with a destination register. At the same time, the physical source registers are requested so that the reservation station and reorder buffer (ROB) have a valid mapping between architectural and physical registers for both source and destination. The mapping within the RAT is speculative, so even though a previous instruction/mapping may still be in-flight/incomplete, we want to map new instructions to use this speculative physical register. Once the previous instruction is completed, we can use the now precise register for the new instruction. If there are available reservation stations at this point, then the instructions along with the physical register indices are forwarded to begin execution. A new ROB entry will also be pushed, but will be marked as not ready-to-commit until the execution stage is finished (Note: if the ROB is full, instructions will not be dispatched even when there are free reservation stations, and vice versa). The execution stage of the CPU consists of a set of reservation stations, an issue arbiter, ALU functional units, MUL functional units, and a CDB arbiter. Each reservation station is a struct that contains the data and metadata necessary for executing an instruction and correctly writing back the instruction. Each reservation station struct contains the 32-bit instruction data, the indices of the physical registers that serve as the source(s) and destination of the instruction, the index of the ROB entry corresponding to the instruction, and the address in memory where the instruction resides. The reservation station struct also contains a bit to indicate whether it is occupied (busy). For CP2, we used two reservation stations. The issue arbiter searches for instructions in the reservation stations whose sources are ready and searches for available functional units. Then, ready instructions are issued to available functional units. For CP2, our issue arbiter could issue one non-MUL ALU instruction and one MUL instruction per cycle. To do so, the issue arbiter did an associative search over the reservation stations. For each reservation station, the issue arbiter checks whether the station is occupied, whether both the ps1_s (the physical register to which rs1_s is mapped) and ps2_s registers are ready, and whether the instruction is a MUL. To check whether ps1_s and ps2_s are ready, the issue arbiter receives a bitmask from the register file that indicates the status of each register (i.e. valid/invalid). (For instructions without a second source register, ps2_s is set to be p0, which corresponds to architectural register x0 and is always valid.) And to check whether the instruction is a MUL, the issue arbiter examines the opcode and the funct7 fields of the instruction. Every cycle, the issue arbiter notes whether there exists a non-MUL ALU instruction that is ready to issue and the reservation station index of the ready instruction. It similarly notes the existence and reservation station index of ready MUL instructions. The issue arbiter also receives a bitmask indicating which ALU and MUL functional units are busy, and it performs an associative search of the ALU and MUL functional units, noting which ALU/MUL functional units are available, if any. If the issue arbiter finds an instruction that is ready to be issued and a functional unit of the corresponding type that is free, it combinationally reads the source register values from the register files and passes these values along with the other data needed to execute the instruction to the functional units in a struct we call issue_fu_data_t. Additionally, it updates a bitmask rs_to_free to indicate which reservations should be emptied. The ALU and MUL functional units then register the inputs from the issue arbiter and set their busy bits. At this point, the functional units decode the instructions to obtain the opcode, funct3, funct7, and immediate values needed to execute the instruction. We chose to decode the instructions immediately before they are used in the functional units to save area that would otherwise be needed to register the components of the decoded instruction in earlier stages of the pipeline. After the functional units finish their calculation, we register their output and metadata such as the destination physical register and rob number in a struct we call fu_cdb_data_t. The fu_cdb_data_t structs for all functional units in addition to bitmasks indicating which ALU functional units and which MUL functional units have completed execution are then used as inputs for the CDB arbiter. For CP2, we had a single CDB that contained a valid bit, the destination physical register, the ROB number corresponding to the instruction, the data to write to the physical register, and the RVFI data. The CDB arbiter worked by performing an associative search over the ALU functional units and the MUL functional units. If there existed an ALU (MUL) functional unit that finished executing, the CDB arbiter took note of the index of and the data present in the corresponding fu_cdb_data_t struct for the first finished ALU (MUL) functional unit it found. Since there was only one CDB for CP2, if there existed both an ALU and MUL functional unit that finished executing in a given cycle, the MUL functional unit was given priority to avoid the scenario where ALU functional units constantly finished executing, causing the MUL functional unit to be starved from the CDB. Finally, the CDB arbiter copies over the data from the ready ALU or MUL functional unit onto the CDB. The CDB arbiter also raises an ack signal for the functional unit it services, telling the functional unit that provided the data placed on the CDB that it can reset and begin executing a new instruction. For CP2, the data on the CDB is sent to the physical register file for updating register values and the ROB for setting the ready-to-commit bit to be valid. If the ROB entry at the head of the queue is valid, then we commit the instruction which completes the process of fetching, decoding, executing, and committing that particular instruction. The CPU then needs to complete some cleanup for the completed instruction by updating the register retirement file, which holds the precise, non-speculative mapping between architectural and physical registers. The old physical register that is replaced will then be sent to the free registers list for future instructions. This process for executing non-memory and non-control instructions stays mostly the same for the rest of our project.

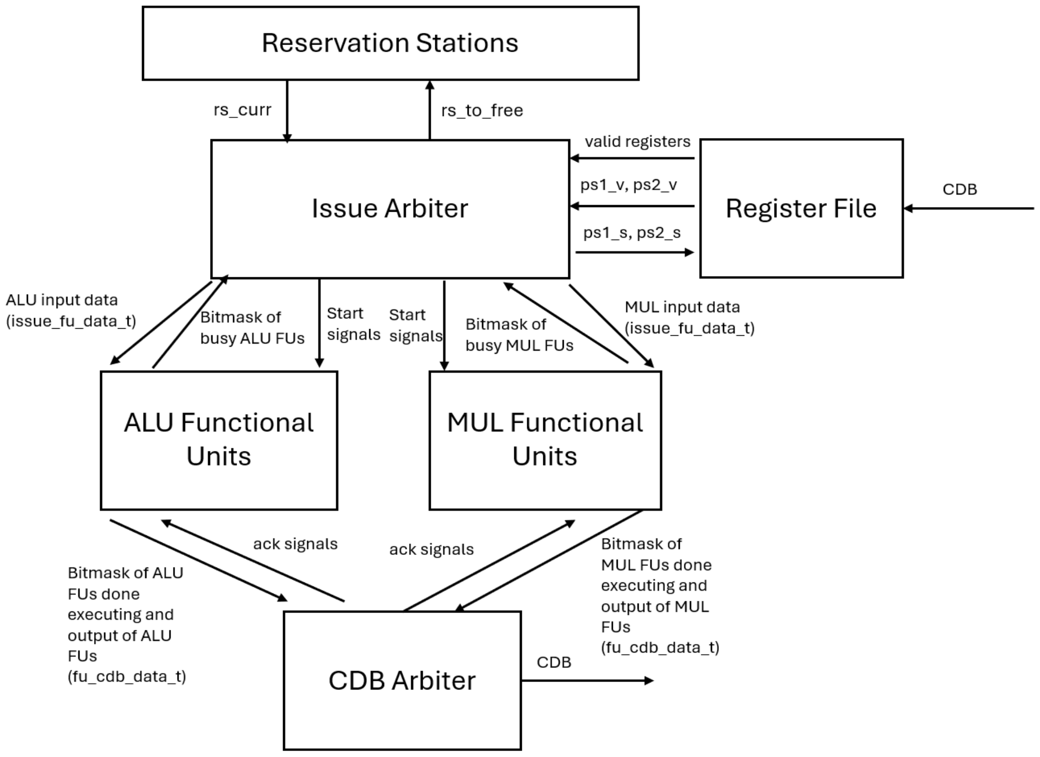
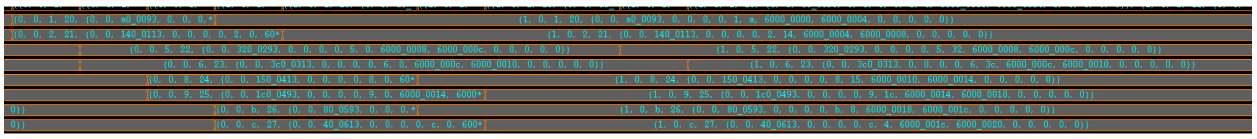


Figure 2: Detailed diagram of the execution stage developed in CP2

We mainly relied on the given test cases, ooo_test.s and dependency_test.s for testing CP2. The test cases were simple and we were able to easily verify the intended behavior within the waveform in RVFI. For example, in the ooo_test.s case, we can clearly see that the ALU instructions that came after the multiply instruction are marked as ready-to-commit in the ROB before the multiply has finished. This demonstrates that our instructions can execute out-of-order, although we still have to wait for the multiply to finish to commit in order. Below is a waveform showing that valid bits are being set out-of-order.



```
{0, 0, 1, 20, {0, 0, a0_0093, 0, 0, 0,*        {1, 0, 1, 20, {0, 0, a0_0093, 0, 0, 0, 0, 1, a, 6000_0000, 6000_0004, 0, 0, 0, 0, 0}}
{0, 0, 2, 21, {0, 0, 140_0113, 0, 0, 0, 0, 2, 0, 60*        {1, 0, 2, 21, {0, 0, 140_0113, 0, 0, 0, 0, 2, 14, 6000_0004, 6000_0008, 0, 0, 0, 0, 0}}
    {0, 0, 5, 22, {0, 0, 320_0293, 0, 0, 0, 0, 5, 0, 6000_0008, 6000_000c, 0, 0, 0, 0, 0}}        {1, 0, 5, 22, {0, 0, 320_0293, 0, 0, 0, 0, 5, 32, 6000_0008, 6000_000c, 0, 0, 0, 0, 0}}
    {0, 0, 6, 23, {0, 0, 3c0_0313, 0, 0, 0, 0, 6, 0, 6000_000c, 6000_0010, 0, 0, 0, 0, 0}}        {1, 0, 6, 23, {0, 0, 3c0_0313, 0, 0, 0, 0, 6, 3c, 6000_000c, 6000_0010, 0, 0, 0, 0, 0}}
        {0, 0, 8, 24, {0, 0, 150_0413, 0, 0, 0, 0, 8, 0, 60*        {1, 0, 8, 24, {0, 0, 150_0413, 0, 0, 0, 0, 8, 15, 6000_0010, 6000_0014, 0, 0, 0, 0, 0}}
        {0, 0, 9, 25, {0, 0, 1c0_0493, 0, 0, 0, 0, 9, 0, 6000_0014, 6000*        {1, 0, 9, 25, {0, 0, 1c0_0493, 0, 0, 0, 0, 9, 1c, 6000_0014, 6000_0018, 0, 0, 0, 0, 0}}
0}}            {0, 0, b, 26, {0, 0, 80_0593, 0, 0, 0,*        {1, 0, b, 26, {0, 0, 80_0593, 0, 0, 0, 0, b, 8, 6000_0018, 6000_001c, 0, 0, 0, 0, 0}}
0}}            {0, 0, c, 27, {0, 0, 40_0613, 0, 0, 0, 0, c, 0, 600*        {1, 0, c, 27, {0, 0, 40_0613, 0, 0, 0, 0, c, 4, 6000_001c, 6000_0020, 0, 0, 0, 0, 0}}
```

Figure 3: ROB entries show ready-to-commit bits (the first bits in the structs) are being set to 1 out of order (5th entry/row is set before 4th entry/row)

We then verified with spike to make sure that each of our commits correctly matches with the golden sample. Similarly for dependency_test.s, we made sure that instructions that were not ready to be executed were held in the reservation station until the dependent instruction was completed. This was also further confirmed through spike and RVFI, which showed that we weren't committing wrong values to registers (which would happen if we did not deal with dependencies correctly). Although the ALU/MUL instructions were working by the end of CP2, we later had to make some minor changes and bug fixes in certain modules to make them compatible with memory and control instructions (i.e. mispredict signal from ROB on branches).

CP3

In our final checkpoint, we finished implementing the complete out-of-order CPU by adding support for control instructions (BR, JAL, JALR) and memory instructions (ST, LD). Additionally, we integrated our cache design from MP_Cache into this CPU as the instruction cache and memory cache as two separate instances.

The control instructions utilize the same rename/dispatch stage as any other instructions, are dispatched into the ALU/MUL reservation stations, and are then issued into the combined compare/ALU units for branch enable and branch target calculation. To track the branches during commit, we chose to add a separate branch queue besides the ROB. Without the branch queue, we would have had to add an additional 32-bit field to each of the ROB entries as the branch/jump target field of the ROB entry. This would have been inefficient, though, as the majority of the test bench instructions are not control instructions. The branch queue contained a smaller number of entries (parameterizable, 16 as default) to save space, and each entry contained a 32-bit branch-target address and a ROB index pointer that points to the ROB entry that corresponds to the current control instruction. During dispatch, only control instructions are dispatched into the branch queue (control instructions are also dispatched to ROB). After a branch or control instruction is evaluated in the functional unit, a branch enable bit (also added in this checkpoint) will be asserted/deasserted on the CDB, which will update the ROB (branch enable) and branch queue (target address) entries. During a commit, the ROB asserts a mispredict signal and informs the branch queue that a branch taken instruction is being committed, which will let the branch queue use the head of the queue to send the correct target address back to the fetch stage to update the PC. The ROB's mispredict signal will also be broadcasted to many other structures within our CPU to ensure that we don't commit or work on any instructions that came after the mispredict. This includes completely clearing the reservation stations (both ALU/MUL instructions and memory instructions which will be discussed later), instruction queue, dispatch instructions, ROB entries, and instructions executing in the functional units. We also need to restore the RAT to the precise state stored by the RRF and free up the free list entries that were used by invalid in-flight instructions. At this point in the MP, we used a simple static-not-taken branch predictor since we wanted to ensure functionality before all else, so whenever we take a jump and on most branches, there will be a mispredict. Note that when a branch instruction is at the head in the ROB, the corresponding instruction is guaranteed to be at the head of the branch queue.

For this checkpoint, we implemented a load/store combined queue to act as the "reservation stations" or issue queue of memory instructions. In this implementation, loads and stores were dispatched and executed in order relative to each other, so even if a load's operands are ready, it must wait for the next older load or store. This queue was attached to the rename/dispatch stage, so all memory instructions went into the load store queue (LSQ). To execute memory instructions, we designed a memory arbiter module to issue the instructions. Because memory instructions in this checkpoint are in order, we didn't need any sort of scheduler or priority system. The arbiter checks the head of the LSQ to determine if an instruction is ready to be issued (operands ready) and also checks if the ROB number stored in the entry at the head of the LSQ matches the ROB head. If the memory is not busy (encoded as a 2-state state machine in the arbiter: busy/not busy), the instruction is issued, and the instruction is popped only after a response is received from D-cache. At the same time, the output of the instruction is placed on the CDB.

There were two other considerations for handling memory systems: competing reads between I-cache and D-cache and burst memory reads/writes. To handle these, we also designed a cacheline adaptor. This module is a state machine with 3 state variables and a counter variable. One variable encodes whether there's a current memory operation in flight, the second encodes whether the adapter is reading from or writing to cache, and the third encodes the choice of cache (i_cache or d_cache). The counter is required because each memory operation contains 4 transaction bursts of 64 bits each. Because there are always instruction reads, we made the cacheline adaptor always prioritize data memory instructions. During a mispredict, the downwards facing port of either cache may change its address, so the cacheline adaptor compares the cache's address to the memory response's address and only asserts a response to the cache when the addresses match.

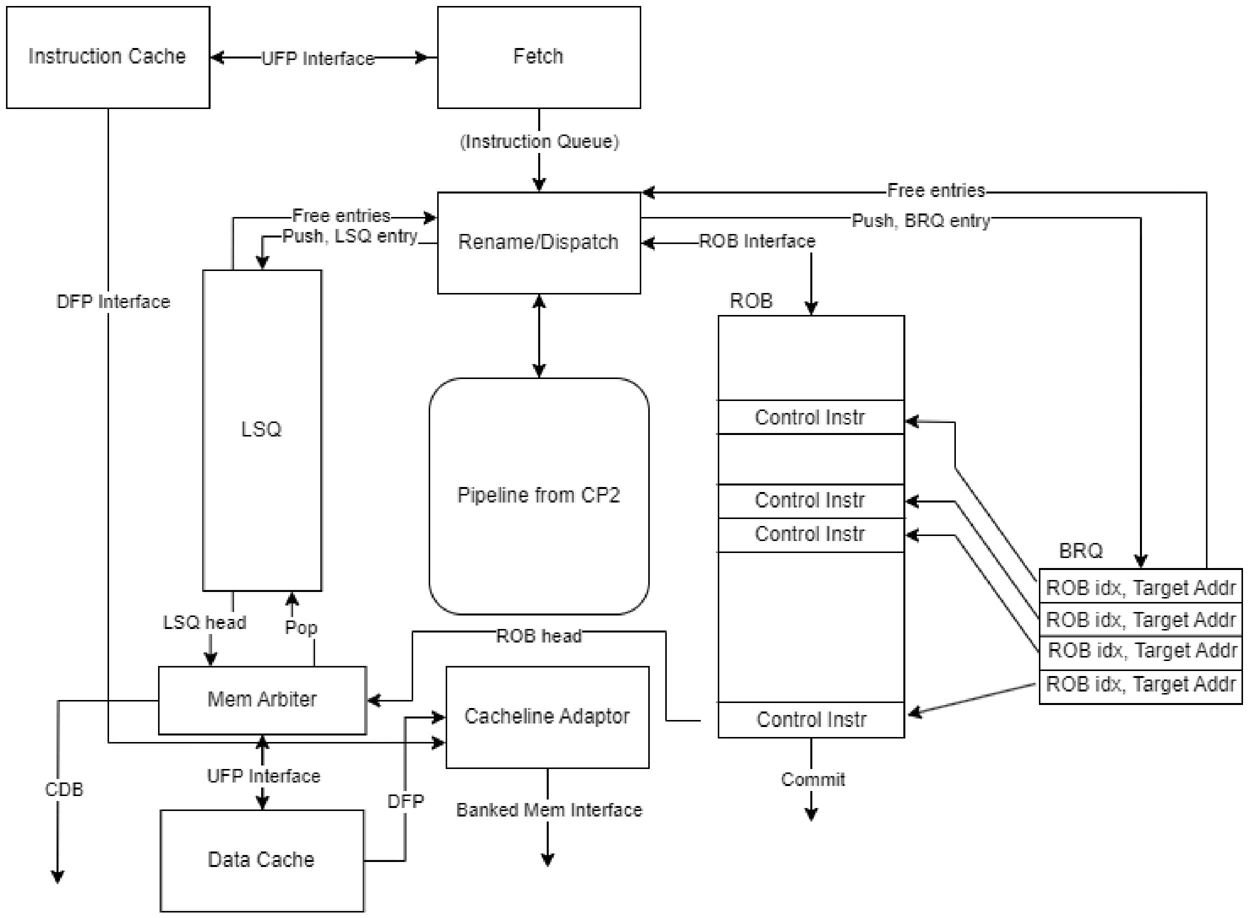


Figure 4: New structures added: LSQ, memory arbiter, caches, cacheline adaptor, branch queue (BRQ)

Because the CPU was fully functional by the end of this checkpoint, we tested the design by connecting the competition memory to the CPU in the testbench and running test programs. We first made sure that the new features did not break any old programs by running simpler test programs from the previous checkpoints. Next, we wrote two test programs in assembly that focus on repeated load/stores and branches. Afterwards, we moved on to quicksort, which was much more complex than anything we wrote previously, and finally, we verified our design on coremark. When running coremark, there were occasionally bugs caused by instructions that were difficult to interpret, especially memory instructions that might pass through on RVFI but become incorrect in spike, so we tried to replicate the incorrect behavior using smaller test cases to accelerate the debugging process. Obviously, there were numerous bugs during each stage of the testing process, but using this multistage debugging process greatly improved our efficiency as we were able to fix one bug at a time with smaller programs instead of jumping to coremark in the beginning.

With this baseline design, we achieved an IPC of 0.298 on coremark clocked at a maximum frequency of 400MHz. Area was measured to be roughly 300,000 μm^2.

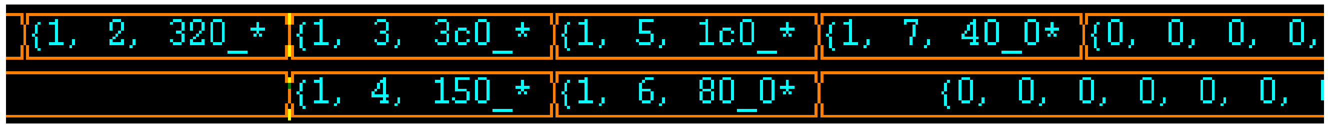
Advanced Design Features

We implemented a series of advanced feature optimizations after completing the baseline design to increase performance. Because we had already made the decision to implement superscalar at the beginning of the MP, portions of the design already had some level of superscalar compatibility, i.e., some modules had a parameterizable superscalar factor that determined the number of simultaneous operations (dispatch, commit, etc.) that could be done. Therefore, we tackled superscalar as our first optimization. After implementing superscalar, we noticed that without a good branch predictor, large portions of the simultaneous instruction commits would be wasted due to pipeline flushes caused by the static-not-taken mispredictions, so we then implemented a two-bit saturating counter branch predictor, which dramatically reduced the misprediction rate. Afterwards, we were recommended to implement a better multiplier than the one provided, as the default multiplier took 32 cycles to finish an operation. At the same time, we added performance benchmarks into our testbench to help us locate bottlenecks in our design, by which we noticed that the instruction queue was empty for a large proportion of time (up to 50%). We also noticed that instruction fetches from the instruction cache often had an extra cycle of delay due to the non-pipelined nature of cache (the cache state machine needed to return to IDLE before accepting a new request). This prompted us to implement a pipelined cache, where a new request can be given on the same cycle as the response of the previous request. Finally, we noticed that the LSQ for memory instructions was a bottleneck during cases where load instructions were holding each other up even when there were no dependencies, so we separated the LSQ into two different structures that allowed loads to execute out of order with respect to each other.

Superscalar

At the end of CP3, we had already built the foundations for superscalar capability in several of our modules. As mentioned earlier, implementing a superscalar processor was our main goal from the beginning, as it has the potential to improve our IPC by a factor of up to the value we choose our SS_FACTOR to be. We were aware of some tradeoffs of having a superscalar processor, such as an increase in area due to having more ports and registers, as well as a lower max frequency due to fanout. However, the significant IPC improvement and lower power consumption (due to lower frequency) would help balance out the negatives.

Starting from modifications in the fetch stage, we needed to pull more instructions per clock from the cache, as we would get no performance improvement if we could only fetch one instruction per cycle. Our fetch stage was changed so that it can pull INSTR_FETCH_NUM * 32 (instruction width) bits from the instruction cache every cycle, which would represent INSTR_FETCH_NUM consecutive instructions. The INSTR_FETCH_NUM instructions will then be sent to the instruction queue along with some necessary metadata. One issue we had to deal with was ensuring that the instruction memory address sent to the instruction cache is always aligned with INSTR_FETCH_NUM and also correct. For example, each of our cacheline holds 256 bits (8 instructions), so the cacheline could hold instructions with address 0x6000 - 0x601C. If a branch sets our PC to 0x601C with INSTR_FETCH_NUM = 2, we might try to fetch instructions at addresses 0x601C and 0x6020. This will cause us a lot of trouble since the instruction at 0x6020 will not be in this cacheline when we try to fetch two instructions. To solve this issue, we made sure that all of the instructions fetched from the cache were INSTR_FETCH_NUM*4 byte-aligned. This means if we fetch 0x601C, we would get 0x6018 and 0x601C instead of 0x601C and 0x6020. Another issue this caused is we don't want to execute 0x6018 if our branch branches to 0x601C, so we also send a valid bit with each of our instructions to the instruction queue, and we would set valid to 0 for 0x6018 but 1 for 0x601C. The structure of the instruction queue stayed mostly the same other than adding parameters and width compatibility with the new instruction bundles being retrieved from fetch. Once the instructions are in the instruction queue, the dispatch stage has to determine whether or not a new bundle of instructions can be retrieved from the instruction queue and also break up the INSTR_FETCH_NUM instruction bundle into digestible parts for the rest of the pipeline. Our implementation of superscalar was also unique in that not only can we commit a variable amount of instructions in one clock cycle, but we can also dispatch a variable number of instructions from the current instruction bundle in one clock cycle. The dispatch module maintains how many instructions in the instruction bundle have already been dispatched and how many still need to be dispatched. When dispatching, the module takes the minimum of the available resources and SS_FACTOR. The amount of available resources are kept track of and sent from other modules such as the ROB and reservation station. For example, if the SS_FACTOR is 4 and there are 6 available reservation stations, 12 available free registers, 30 available ROB entries, and 2 instructions left in the bundle, then we will send the last 2 instructions to the LSQ or ALU/MUL reservation stations. If we only have 1 available reservation station, then we will only send 1 instruction from the bundle and mark that instruction as sent before sending the final instruction at a later time. In order to efficiently support parallel execution and writeback of instructions, some modifications were made to the execution stage of our processor. The primary change is that we created multiple "execution groups," each of which contained two ALU functional units, one MUL functional unit, four reservation stations, and an issue arbiter. In essence for each of these execution groups, the issue arbiter would conduct an associative search over the four reservation stations and dispatch ALU and MUL instructions with ready operands to the ALU and MUL functional units in the group. Unlike our issue arbiter for CP2, the issue arbiters we used for superscalar execution tried to issue as many instructions as possible in one cycle, which it accomplished by searching for up to two ready ALU instructions, two empty ALU functional units, one ready MUL instruction, and one empty MUL functional unit every cycle. We decided to split our execution resources into multiple groups to avoid conducting associative searches over all reservation stations, all ALU functional units, and all MUL functional units. We were concerned that conducting these large associative searches would lead to significant area overhead or additional combinational delay. By splitting our execution resources into groups, the associative searches conducted could be smaller and more efficient with respect to area and delay. A similar idea was used for our CDBs. To accommodate superscalar, we increased the number of CDBs. To avoid conducting associative searches over all ALU functional units and MUL functional units, each CDB was assigned to two ALU functional units or to two MUL functional units. Then, the CDB arbiter for each group of functional units would perform an associative search over its corresponding functional units, searching for ready functional units similar to what was described in the CP2 section. Slight modifications were also needed in various other modules such as the ROB, RRF, and free list, as superscalar commits means that we might retire multiple registers or pop multiple free registers in a single clock cycle. We simply accomplished this by moving the tail/head of queues by the amount requested by dispatch/committed by the ROB and forwarding the information to the modules that needed it. One specific consideration we had to make for the ROB/branch queue was dealing with multiple branches in a single commit. We want to make sure the first mispredict that fires will override all future instructions and mispredicts that also want to be committed on that cycle. Because we ensure that the ROB entries are in order, when we loop through the instructions we want to commit starting from the ROB head and we hit a misprediction, we will break out of the loop and not commit the rest of the instructions even if instructions afterwards are marked as valid. A misprediction is then treated the same way as our non-superscalar CPU.



```
{1, 2, 320_* {1, 3, 3c0_* {1, 5, 1c0_* {1, 7, 40_0* {0, 0, 0, 0,
             {1, 4, 150_* {1, 6, 80_0*       {0, 0, 0, 0, 0, 0,
```

Figure 5: We see that we can commit 0, 1, or 2 instructions per clock cycle depending on the status of the ROB with SS_FACTOR = 2. If we set SS_FACTOR = 4, we would be able to commit anywhere between 0 and 4 instructions per clock cycle. We are also able to dispatch and process less than SS_FACTOR instructions per clock cycle, improving flexibility. On a good day where all instructions are independent, in the cache, and are ALU instructions, we can achieve an IPC of up to SS_FACTOR!

The load-store portion of our CPU was not upgraded to superscalar at this point, so only one memory instruction can be executed in each cycle (to ensure memory operations don't execute out of order and cause data hazards).

Testing superscalar capabilities was not difficult since we just had to ensure that the competition suite and all of our previous test benchmarks passed both RVFI sim and spike. To make sure we were committing in superscalar, we looked at the RVFI waveform to verify that there were many instances of multiple instructions being committed in the same clock cycle. It was also during testing that we discovered having superscalar by itself worsened our performance on almost all fronts. Using SS_FACTOR = 2, the IPC barely improved and sometimes was worse than the non-superscalar version. We were also unable to clock our CPU as high (less than 300 MHz) due to the increase in logic and fanout in each stage of our processor, and our area bloomed to over 350,000 µm^2. This led to us conducting several performance benchmarks for bottlenecks and allowed us to determine the rest of the advanced features to really make our superscalar processor shine. First, we knew a basic two-bit saturating branch predictor would be a relatively straightforward way to boost our performance, as a mispredict hurts our superscalar processor a lot more than a non-superscalar processor (more unnecessary cache to memory requests on cache misses since we go through instructions faster, more wasted instructions being executed, etc.). Decreasing the frequency in which we need to pay a misprediction penalty would greatly help with this to decrease the amount of wasted work. Our performance benchmarks also showed that most of the bottlenecks when running our test programs came from the instruction queue/fetch stage being empty and the LSQ being full. When looking at the RVFI waveform, we discovered that the cache takes two cycles to fetch an instruction bundle while we can dispatch up to an entire instruction bundle every cycle. This meant the dispatch stage (and everything after) did little to no work half the time when there were no instructions coming in. We were also not dealing with loads particularly well, since one load can hold up the entire processor if it is waiting for a long or highly dependent instruction such as a multiplication instruction. To address the fetch bottleneck, we decided that a read-only, pipelined instruction cache would allow us to at the very least remove the instruction fetch bottleneck since a pipelined cache would be able to respond with an instruction bundle every cycle. Then, we would target the LSQ bottleneck by creating separate structures for loads and stores. Stores would be treated the same as before since there are no easy ways around committing out of order stores, and loads would be allowed to commit out of order as long as all stores that came before it have been completed. Thus, our next steps became obvious — add a branch predictor and then go through the rest of the features that can resolve bottlenecks with the time we have left.

After implementing the remaining features, we found that having superscalar capabilities did improve our performance by ~20% across the board compared to running the same benchmarks with the same advanced features but only having single commits. This is a major improvement in demonstrating the capabilities of a superscalar processor, as it shows that as we refine the other parts of our processor, the superscalar factor would provide additional gains. In our case, we believe that the changes we made, specifically the branch predictor, increased the proportion of time in which instructions can execute in parallel.

Our superscalar processor would perform well on workloads with a lot of instruction-level parallelism. For workloads that consist primarily of ALU/MUL instructions with few dependencies, perhaps a workload like matrix multiplication where the operands are stored in registers, our processor would be able to fetch, execute, and commit approximately SS_FACTOR instructions per cycle, the maximum speedup possible with SS_FACTOR-way superscalar. On the other hand, our processor would likely perform poorly on workloads with many dependencies. A contrived example of such a workload is a program that maintains an accumulator register and does ALU/MUL operations where the accumulator register serves as both a source and the destination for every operation. In this case, instructions are dependent on all preceding instructions, which means that a particular instruction cannot begin to execute until all instructions earlier in program order have completed. In this manner, instruction execution would become serialized, which means that our superscalar processor would not be able to utilize its ability to execute and commit multiple instructions at a time.

Branch Predictor

To decrease the frequency with which we would need to pay the performance penalty for mispredicted branches and unexecuted jumps, we decided to implement a branch predictor and partially execute jumps directly in the instruction fetch (IF) stage.

To support a branch predictor and partial execution of jumps in the IF stage, we changed how we calculated the next value of the PC. For simplicity of discussion, assume that our PC is aligned such that the next INSTR_FETCH_NUM instructions lie within a single cache line (as described in the superscalar section). Upon receiving the packet of INSTR_FETCH_NUM from the instruction cache (I-cache), the IF stage partially decodes these instructions, determining their opcodes as well as the immediate values used for JAL instructions and branch instructions. If there exists in the packet a JAL or a branch instruction that is predicted to be taken, then the IF stage sets the next value of the PC to be the target of the JAL or predicted-taken branch instruction earliest in program order. Additionally, before the IF stage pushes the packet of instructions to the instruction queue, it invalidates all the instructions that follow the JAL or predicted-taken branch instruction earliest in program order. For JALR instructions, the IF stage "predicts" that the JALR instruction's target is its address plus four. While this behavior will probably not be an accurate prediction, we felt that given the low frequency of JALR instructions (only <3% in regular coremark), the additional overhead needed to support JALR instruction such as an additional read port on the register file and logic needed to check the validity of the JALR source operand would not be worth implementing. Additionally, it is possible that the source register of the JALR instruction is not ready when the JALR is fetched, which means that the JALR instruction might need to be speculatively executed anyway.

Our branch predictor consisted of a table of 64 two-bit saturating counters. We used bits [7:2] of the instruction address to index into our table two-bit counters. Every cycle the IF stage provides the branch predictor with the address of the first instruction (in program order) that it will fetch, and the branch predictor indexes into the table of counters to provide a prediction about whether a branch at a particular address whose instruction the IF stage will fetch should be taken or not.

One design choice we made was to update the counters in a branch predictor by connecting the branch predictor to the CDBs and updating the branch predictor immediately after branches are finished executing. To support updating the branch predictor using the CDBs, we added fields to the CDB struct to indicate whether the instruction in a given CDB is a branch, whether it is a taken branch if it is a branch, and what the address of the instruction is. Immediately updating the branch predictor after branch execution allows us to save some area by avoiding storing extra data in the branch queue, and it allows us to update the branch predictor more quickly than we could if we only updated the branch predictor on committing branch instructions, which allows the branch predictor to adapt to new patterns with respect to taken/not-taken branches more quickly. On the other hand, updating the branch predictor immediately after branch execution may lead us to update the branch predictor based on branches that are only speculatively executed and never actually committed, potentially leading to less accurate predictions.

Another design choice we made within the branch predictor was how to increment the two-bit counters. Because our processor is superscalar, multiple branch instructions may enter the CDBs in one cycle. Therefore, with the ordinary implementation of the two-bit counter, we might need to change the value of the counter by more than one. In our design, we still only change the value of a given counter by at most one in a single cycle. To change the value of the counter, we count the number of taken branches and not-taken branches in the CDBs whose addresses index to a given counter. If both the number of taken branches and the number of not-taken branches are equal, the counter value does not change. If the number of taken branches exceeds the number of not-taken branches, then we increment the counter by one (e.g. weakly not-taken to weakly taken). Finally, if the number of not-taken branches exceeds the number of taken branches, then we decrement the counter. By only incrementing/decrementing the counter by at most one every cycle, we simplify the logic needed to update the value of the counters. Additionally, we concluded that the likelihood that two branches whose addresses index to the same counter finish executing in the same cycle is small (since this would require at least two instances of a single branch instruction completing execution at the same time or at least two branch instructions with 60+ instructions in-between them completing execution at the same time). Hence, the behavior of our design would not differ from the behavior of the ordinary implementation of a two-bit counter in most cases.

Additionally, to support branch prediction and partial execution of JAL, we added metadata to the structs that we use to pass instructions through our processor. For example, a bit indicating whether the instruction is a branch predicted to be taken or a JAL is passed from the

IF stage through the rename/dispatch stage to the execute stage so that the execute stage can use this bit to determine whether the prediction for a particular branch matches its actual behavior (and consequently indicate a misprediction has occurred if there exists a mismatch). We also repurposed the branch taken field in the ROB as a control instruction mispredicted field.

As a result of implementing our branch predictor, the proportion of control instructions for which we need to flush the pipeline (e.g. JALRs and mispredicted branches) decreased from 65% to 30% when running regular coremark with no multiplications. Additionally, when running regular coremark on a 2-way superscalar processor, our IPC increases from 0.269 to 0.433 after implementing the branch predictor. Our branch predictor would likely work well with workloads where a particular branch instruction can be predicted accurately using its behavior in the previous few times that it was executed. One such situation is a branch instruction that determines whether a for-loop with many iterations should repeat. In anticipation of such a workload, our counters are initialized to weakly taken, so on workloads with many for-loops each with many iterations, our branch predictor only mispredicts once in the final iteration of each for-loop and correctly predicts the whether each for-loop should repeat for all other iterations. Our branch predictor would likely not work well in workloads where a branch instruction alternates between taken and not-taken. One such example is an if-statement inside of a for-loop where this if-statement evaluates to true only if the loop variable is odd. Then, every iteration, the branch instruction alternates between taken and not-taken. Because of how we initialize the branch predictor, the branch predictor will alternate between weakly taken and weakly not-taken and will always predict incorrectly.

Multiplier

One feature that we didn’t originally plan on making was improving the shift-add multiplier, since our performance metrics did not detect that they were a bottleneck. Although our performance tests kept track of how often the multipliers were being used (they weren’t usually filled up), it did not track other metrics such as its impact on dependent instructions. The original shift-add multiplier took 32 clock cycles (one shift per cycle for 32 bits) to finish executing a multiplication instruction. This meant that if other instructions were dependent on a multiplication instruction, it would have to wait 32 clock cycles, holding up other parts of our CPU and potentially reducing superscalar parallelism. Changing to a combinational multiplier was an upgrade recommended by our mentor, Pradyun, as a quick and easy way to improve our IPC. Since the multiplier wasn’t a part of our critical path, we could perform more than a single shift/add operation in each clock cycle. To do this we used retiming and DC to reduce a fully combinational multiplier (which would be terrible for the critical path) to a 4-stage pipeline rather than a default 32-stage pipeline. In the first clock cycle, the values would be latched into a register and in the subsequent clock cycles, we would perform multiple shift/add operations in each clock cycle divided evenly between each stage by DC and output the value on the final stage. This module directly replaced our original shift-add multiplier, so no further modifications were needed in the other modules.

Functionality testing was more or less the same as the other advanced features we implemented after CP3. We added our new multiplier and checked to see if it can run the benchmarks in the competition suite. The new multiplier surprised us and gave us a significant boost in performance. With the multiplier and branch predictor running coremark_im and

SS_FACTOR = 2, we achieved a ~50% performance improvement without increasing area or power, from an IPC of 0.32 to 0.46. The new multiplier was mainly impactful for benchmarks with many multiply instructions as is the case with portions of coremark_im. In benchmarks that had little multiply instructions or had many instructions that didn't depend on multiplication outputs, the improvements were less obvious, which makes sense because the multiplier only targets the multiplication heavy parts of a program.

Pipelined Cache

During performance testing, we noticed that the instruction queue remained empty up to half the time, causing regular dispatch-stall patterns (one cycle of dispatch, one cycle of stall). This was caused by our cache being non-pipelined, which meant that the CPU could only issue a new request to the cache one cycle after the cache had responded with the previous instruction bundle. Even if every cache request was a hit, the average response time would be 2 cycles instead of the ideal 1 cycle. With a superscalar rename/dispatch stage that could dispatch 2, 4, or 8 instructions per cycle, the instruction queue could not keep up. After pinpointing this bottleneck, we designed a read-only pipelined instruction cache that could take new requests the same clock cycle as the response of a previous request. This halved the average latency of hit requests to the instruction cache, which alleviated the instruction queue to dispatch bottleneck.

The original cache state machine consisted of 4 states: IDLE, COMPARE_TAG, ALLOCATE, and WRITEBACK.

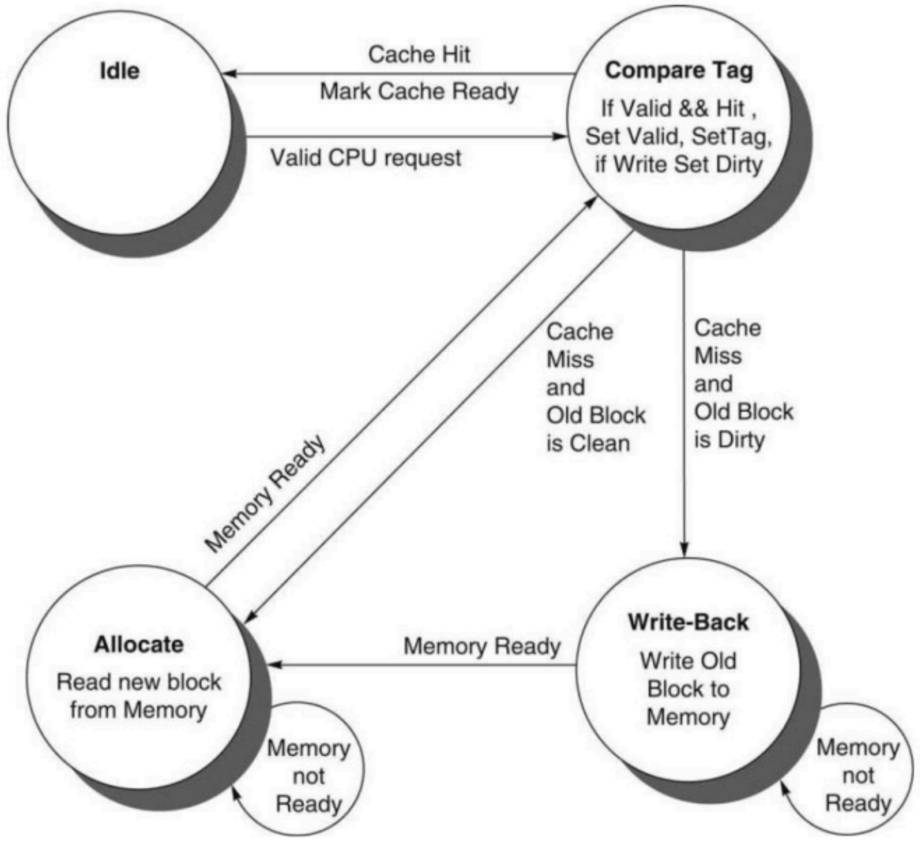


Figure 6: baseline cache state machine, taken from lab slides

Because we never write using the instruction cache, we first removed the WRITEBACK state and transition logic. Next, we split the transition from COMPARE_TAG to IDLE into two different transitions: Hit + no request, which went to IDLE, and hit + request, which returned to COMPARE_TAG. If the cache hits, the fetch module is allowed to make a new request immediately, which keeps the cache in the COMPARE_TAG state. Theoretically, if the CPU does not request another instruction from the instruction cache, the cache is allowed to transition to the IDLE state, but realistically, that does not happen in the Chig Bungus CPU, as we permanently assert the imem_rmask output of the fetch stage.

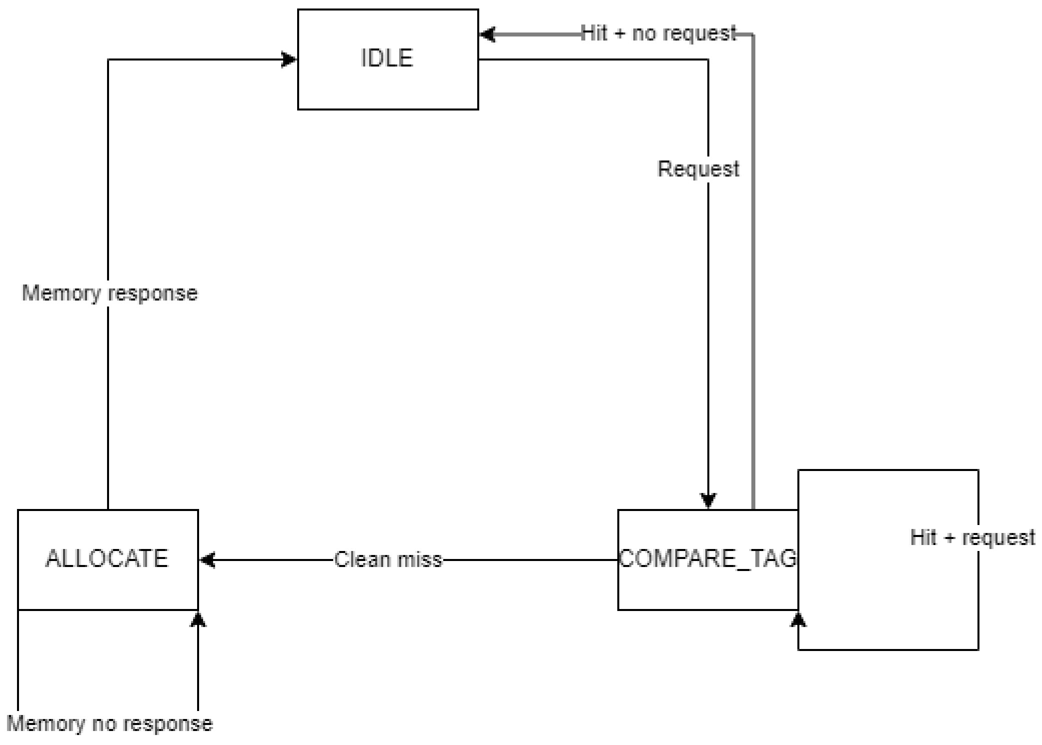


Figure 7: modified cache state machine

This change, however, caused a combinational loop to occur in the CPU. Because matching the tag required the upper-facing port's memory address, the response from the cache depended on the imem_addr output from the fetch stage. Simultaneously, in a pipelined cache, the imem_addr is allowed to change when the fetch stage receives a response from the cache, causing a circular dependency between imem_addr of the fetch stage and the ufp_resp of the cache. To resolve this combinational loop, we created a register ufp_addr_reg that latches the ufp_addr field (imem_addr of the fetch) of the cache every cycle, effectively keeping the imem_addr, specifically the tag and offset fields, of the previous request for the cycle of the response when the fetch stage has moved on to the next address, breaking the circular dependency.

After implementing the pipelined cache, the average instruction cache full time dropped to below 10%, which allowed the rename/dispatch stage to display superscalar dispatch more frequently, and it improved the IPC by up to 0.05 on certain programs. With this optimization, the logic for the cache actually becomes simpler because of the lack of a WRITE_BACK state, which should improve overall performance for any program. However, because the dispatch stage is now capable of dispatching more instructions per cycle on average, this put a higher load on other components in the pipeline, including the execution (ALU/multiplier) and the load/store units, transferring the bottleneck to them. The load store queue, in this case became more of a bottleneck due to the higher rate of dispatch and the in-order nature of the LSQ, and this was apparent in load-heavy sections of our test programs, shown by an increase in the percentage of time that the LSQ was full.

By enabling pipelined cache reads, we expect that on most workloads our processor's performance will increase compared to our processor's performance with a non-pipelined cache because we are able to fetch more instructions per fixed amount of time. As with an ordinary cache, we would expect to see the largest performance increase when running a workload that repeatedly uses the same instructions, for example a workload that includes many loops.

Load Store Disambiguation

With regards to memory disambiguation, we implemented load reservations stations that are separate from the store queue, and we modified the memory instruction issue arbiter to allow for issuing loads out-of-order with respect to each other.

To implement the store queue, we essentially repurposed the LSQ from CP3, and added two output signals: exists_store_dependency and store_dependency. For a particular clock cycle, exists_store_dependency indicates to the load reservation station whether there exists a store instruction that must be committed (i.e. have written to D-cache) before the instructions that are entering the load reservation station in that cycle can be issued. And the store_dependency signal is the ROB number of the store that must be completed if the signal exists_store_dependency is high. We explain how these signals are calculated in our discussion of the load reservation stations.

To implement the load reservation stations, we created a module describing a single load reservation station and wrapped multiple instances of this module in a larger module. The module for a single load reservation station contains a struct that contains the data necessary for executing load instructions (e.g. the physical source and destination register indices, the corresponding ROB number, the 32-bit load instruction). Additionally, the struct contains a state from {EMPTY, WAIT_FOR_STORE, WAIT_FOR_REG, READY} that indicates whether there exists a valid instruction in the load reservation station and whether this instruction is ready to be issued. In general, when a load instruction enters the load reservation station, the state will change from EMPTY to WAIT_FOR_STORE if there exists a store earlier in program order that has not completed. In particular, we store the ROB number corresponding to the store latest in program order that must be completed before the load can be issued. Then, after all stores earlier in program order have completed, the reservation station enters the WAIT_FOR_REG state to wait for the load instruction’s source operand to become ready. After the source operand becomes ready, the reservation station enters the READY state to indicate to the memory arbiter that the load can be issued. To allow the load reservation stations to change their state, each load reservation station is connected to the CDB dedicated for memory instructions (so that the load reservation stations can check for the completion of store instructions with specific ROB numbers) and receives the bitmask indicating which physical registers contain valid data. There is some subtlety involved with determining whether a load instruction must wait for a store instruction, which we explain now. When a load instruction enters the load reservation station, it enters in either the WAIT_FOR_STORE or WAIT_FOR_REG state depending on the value of the exists_store_dependency signal. Suppose a load instruction enters the load reservation station at the beginning of cycle X (i.e. immediately after the positive clock edge that begins cycle X). If at the end of cycle X-1, the store queue is empty or the store queue contains one entry that will be popped out of the store queue at the positive clock edge that begins cycle X, then the exists_store_dependency signal is low, and we place the load instruction in the WAIT_FOR_REG state to indicate that there are no store instructions that the load instruction must wait for before the load instruction can be issued. Otherwise, we place the load instruction in the WAIT_FOR_STORE state and store the ROB number of the store instruction that is at the back of the store queue to indicate that this store instruction must complete before the load instruction can issue (this ROB number is the signal store_dependency mentioned above).

One downside of this method for resolving dependencies between load and store instructions is that the rename dispatch stage cannot dispatch a store and dispatch a load later in program order in the same cycle. For example, if a store is at address 0x6000, and there is a load at address 0x6004, the rename/dispatch stage will not be able to dispatch the store and load in the same cycle since the logic we have implemented cannot indicate that the load at 0x6004 may be dependent on the store at 0x6000. While this limitation could decrease the rate at which memory instructions are dispatched, we feel that the delay imposed on the load instruction by the fact that the load cannot issue until the previous store reaches the head of the ROB and is committed is sufficiently large that the the one extra cycle of delay for the load, which is needed to ensure that the load is not issued in the same cycle as the store, would not be a significant additional delay.

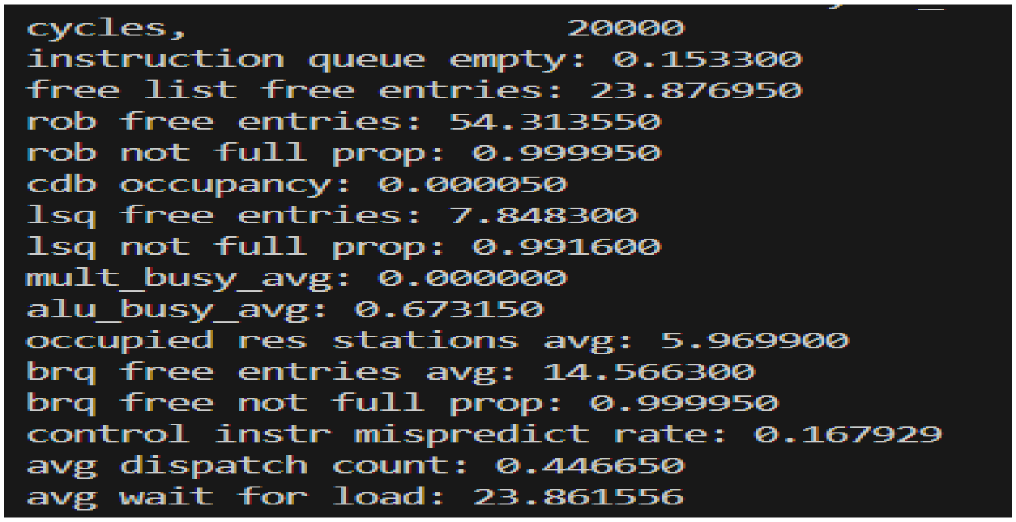
Aside from the load reservation station and the store queue, we also made changes to the memory arbiter to support the separated load/store structures and out-of-order load issue. Previously, with a combined FIFO LSQ, there was only one memory instruction that the memory arbiter could try to issue, which was the memory instruction at the head of the LSQ. However, with separate load/store structures and out-of-order loads, multiple loads and the store at the head of the store queue can all be ready to issue in the same cycle. With separated load/store structures, the memory arbiter does an associative search of the load reservation stations to try to find load reservation stations in the READY state. Because there are many load reservation stations (in our processor there are eight), starvation of a load instruction is possible if the associative search always begins at the same reservation station. To avoid the potential of starvation, the memory issue arbiter performs a round-robin search of the load reservation stations. For example, if the load reservation station with index 4 was the last load to be issued, then the memory issue arbiter will begin its search for the next load to be issued from the load reservation station with index 5. To check for store instructions ready to be issued, the memory arbiter checks whether there exists a store instruction in the store queue, and if so, whether the store instruction at the head of the store queue has valid source operands and whether this store instruction sits at the head of the ROB. If there exists a store instruction that is ready to be issued, then it is given priority over load instructions that are also ready to issue. Another change to the memory arbiter that we made was to handle mispredictions that occurred during the execution of a load instruction. Previously, the mispredict signal could never be raised during the execution of a memory instruction because during the execution of a memory instruction, the head of the ROB corresponded to a memory instruction, not a control instruction. However, because loads can now be executed out-of-order (and in particular can be executed when they are not at the head of the ROB), it is possible for the mispredict signal to be raised when a load is being executed. To address this, we modified the memory issue arbiter's state machine. In particular, the modified memory issue arbiter's state machine consisted of three states {MEM_IDLE, MEM_BUSY, MEM_INVALID}. If a misprediction occurs while a load instruction is being executed (and the memory issue arbiter is in the MEM_BUSY state), then the memory issue arbiter moves to the MEM_INVALID state to indicate that once the D-cache responds, the data read from the D-cache should be discarded and not written to the CDB. Additionally, we register various data in the memory arbiter, such as the address from which the instruction loads, to ensure that this data is still present within the memory issue arbiter even after the mispredict signal has been raised and the other structures have been flushed. This is necessary because the address from the memory issue arbiter is passed through the D-cache to the cacheline adaptor, where it is used to match up against the mem_raddr signal from the banked memory to determine which address corresponds to the data provided by banked memory. If the address is not preserved in the memory issue arbiter, then it is possible for an incorrect address to be passed to the cacheline adaptor, which causes the memory transaction to never complete and stalls the processor.

The following statistics concern the dna.elf test in the competition suite. Prior to implementing this scheme for memory disambiguation, the average number of cycles a load instruction waited in the LSQ before it was issued was 23.64 cycles, and the IPC of dna.elf was 0.389 at a frequency of 297 MHz, leading to a delay of 906579 ns. After implementing this scheme for memory disambiguation, the average number of cycles a load instruction waited in a load reservation station before it was issued was 2.03 cycles, and the IPC of dna.elf was 0.790 at a frequency of 345 MHz (after other optimizations that allowed for increased fmax), leading to a delay of 384662 ns. These statistics demonstrate how the separate load/store structures in addition to the ability to issue loads out-of-order helped decrease the time needed to complete load instructions, which led to substantial speedups on memory-intensive test cases like dna.elf.

In general, our implementation of separate load/store structures and out-of-order issue for loads would work well for workloads with many loads, and especially those where there are many loads whose operands are already ready/valid prior to the dispatch of the load. Our implementation would allow these loads to be issued almost immediately after dispatch, which would decrease the amount of time these loads spend waiting to be issued. On the other hand, our implementation of separate load/store structures and out-of-order issues for loads may perform poorly for workloads with many instances where stores are immediately followed by loads. In this workload and under our implementation, the load instructions would need to wait until the preceding store has committed, and if the preceding store is the instruction immediately preceding the load, then the load will essentially have to wait until it is at the head of the ROB before it can issue. In this case, our implementation would act very similarly to a unified store-load LSQ FIFO, which as we established above, oftentimes has poor performance.

Additional Observations, Reflections, the Competition, and Next Steps

During development, we learned a lot about how our processor behaves, specifically what tasks the processor is good at and where our processor suffers. Towards the end of the deadline, we used this knowledge along with other performance metrics to help with parameter tuning. As previously mentioned, most of our advanced features targeted IPC improvements, so we now turned our focus to area, frequency, and power. Our original ROB had 64 entries for example, but we noticed that on average, around 50 ROB entries (kept track of in HVL) were not being used, so we decided to halve the number of ROB entries to 32. Similar observations and changes were made for the store queue, reservation stations, and functional units. We also observed that the register file took up over 100,000 µm^2 in our area report, primarily as a result of the large number of ports to the functional units, so we reduced the number of ports to decrease combinational area. These optimizations allowed us to trim the area from 350,000 µm^2 to a final 220,000 µm^2, which was still high, but the nature of having superscalar logic meant it was difficult to continue reducing the area without hurting performance. We were also able to bump up the clock frequency to 345 MHz (2900 ps delay) from the 297 MHz (3366 ps) initial frequency after implementing superscalar, giving us a 16% improvement in clock speed. We identified that our final critical path was in the issue stage, but there weren't many edits we could make without drastically changing the functionality and structure of our CPU.



```
cycles,                    20000
instruction queue empty: 0.153300
free list free entries: 23.876950
rob free entries: 54.313550
rob not full prop: 0.999950
cdb occupancy: 0.000050
lsq free entries: 7.848300
lsq not full prop: 0.991600
mult_busy_avg: 0.000000
alu_busy_avg: 0.673150
occupied res stations avg: 5.969900
brq free entries avg: 14.566300
brq free not full prop: 0.999950
control instr mispredict rate: 0.167929
avg dispatch count: 0.446650
avg wait for load: 23.861556
```

Figure 8: Some metrics from coremark to determine bottlenecks and unnecessary structures.

Although we were generally satisfied with where we got with our processor we acknowledge that there is still room for improvement. One feature that definitely hurt us in the competition was the lack of compatibility with the RISC-V division instructions. For the rsa and physics benchmarks, our delay was significantly worse than other processors since we had to perform sequential subtraction rather than having a division functional unit. Support for division instructions would reduce the number of instructions we have to execute, but this would only give a performance benefit for programs that can be reduced with division while increasing area. Another improvement we could've made was a nextline prefetcher in the cache which would reduce the instruction memory access time on cache misses. Rather than fetching the next instruction cacheline from memory when needed on a cache miss, we can fetch the current instruction cacheline as well as the subsequent cacheline to potentially cut the number of cache misses in half. However, this feature would not help in branch/jump heavy programs since the prefetched instruction line will be useless after a branch. Finally, a more complex advanced feature that will likely have very little tradeoffs is early branch recovery (EBR). EBR would detect mispredictions before commit, reducing the amount of time we are executing invalid instructions and increasing IPC. The degree of performance improvement would range from program to program (depending on prediction accuracy/control instruction count), but all mispredictions will benefit from being detected early. This feature also becomes less necessary if we improve our branch predictor, and would not benefit a hypothetical processor that always predicts accurately. It is also important to note that these features would only improve IPC, so we would still have to make our processor more lightweight by potentially scaling down superscalar capabilities to increase clock speed.

We also had some takeaways on the features that we did implement. We noticed that in general, superscalar capabilities were not key to success in performance. Most of the time, a combination of dependent instructions, waiting for memory accesses, and branch mispredictions meant that increasing the superscalar factor didn't mean that we could actually take advantage of committing multiple instructions. In an attempt to compensate for this, we implemented asymmetric superscalar so the amount fetched and the amount committed in each clock cycle were different. This provided mixed results, and we determined that a 2-fetch/2-commit was still optimal. If we were to do the project again and wanted to focus solely on competition performance, we would likely attempt EBR due to the overhead required for superscalar functionality. Specifically, the dispatch stage had to perform a lot of minimum calculations. The rest of the features were less controversial and generally provided solid gains in the tasks they were targeted to improve, so we were able to include all of our features in the competition.

Final Performance Metrics

Test	Delay (ns)	IPC	Power (mW)	PDA
coremark_im	792449	0.5217	24.6073	180.97
compression	530993	0.4933	24.4267	54.04
dna	384662	0.7895	24.9496	20.99
fft	729147	0.5301	24.4265	139.94
graph	669451	0.4982	24.2651	107.59
mergesort	472593	0.5585	24.8707	38.79
physics	1395112	0.4704	24.7128	991.68
rsa	1451400	0.3166	24.8490	1122.77

sudoku	364005	0.6049	24.7325	17.63
raytracing	1054607	0.5895	24.7796	429.53

Conclusion

In this project, we successfully designed an explicit register renaming based out-of-order CPU that has a parameterizable superscalar pipeline, two-bit saturating counter branch predictor, a combinational shift register based multiplier, pipelined instruction cache, and separated load-store issue structures. This CPU is compatible with the RISC-V 32-bit integer instruction set as well as portions of the multiplication (M) extension and was tested rigorously with various benchmarks and targeted programs. Through this project, we gained valuable insight into CPU architecture and the design process, exercised and sharpened our debugging skills, and practiced our organizational and planning skills. This was one of the most involved and large-scale projects we've done in our curriculum here at UIUC, and we are excited to use all the knowledge and experience we've gained in future projects.

Figure 9: Chig 🅱ungus hanging out with everyone's favorite professor (the next exam is easy, 20 minutes max)