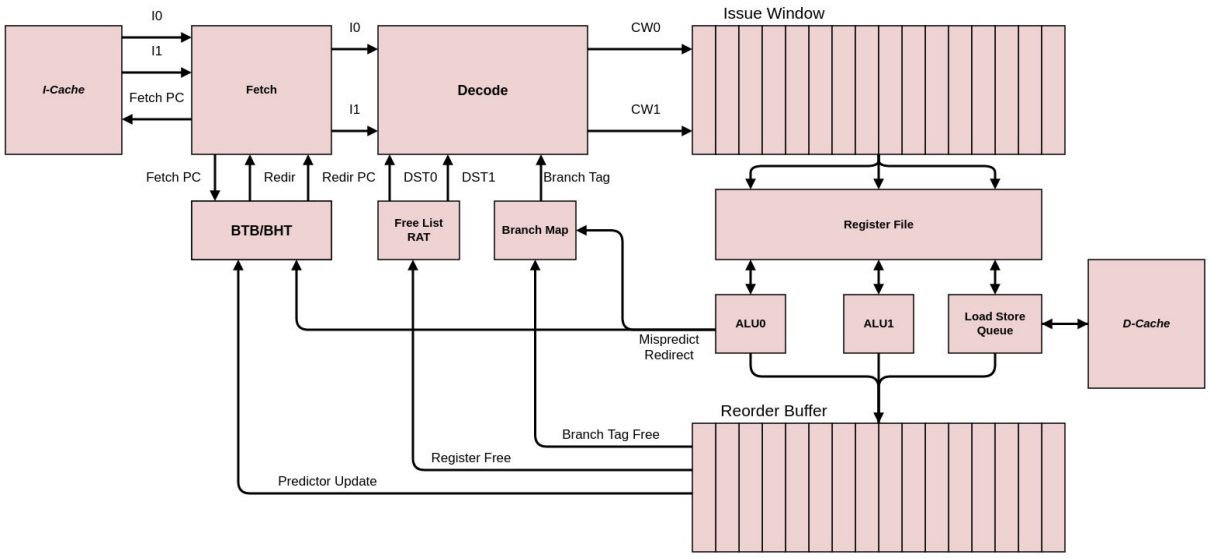
QuantumTsunami

Out-Of-Order Processor inspired by the Berkeley-Out-Of-Order Machine (BOOM)



Adam Auten, Doug MacNerland, Yuan Chih Wu

ECE 411
Spring 2017

Introduction

For our final project for ECE 411, we designed and implemented an out-of-order superscalar LC3b processor in SystemVerilog. Our design, the Quantum Tsunami processor (QT), contains several advanced features designed to increase IPC. These include a 2-instruction wide pipeline, gShare branch predictor and 32-entry BTB, dual port load-store queue with a banked d-cache, and multi-level branch speculation. All caches used in the design are 4-way set associative and use a pseudo-LRU replacement policy.

We designed the core over the course of 6 weeks, with the work divided into four checkpoints. Each checkpoint marks a milestone in the design process, and incorporates additions and refinements to the core design. An overview of each checkpoint is described below, and our overall results of the project are summarized.

Project Overview

Our intent was to challenge devising hardware to support the best performance for execution of programs written in LC3b. We decided to create an out-of-order processor in an attempt to leverage these instances of speedup and performance increase due to instruction level parallelism, as well as our interest in the compelling and complex challenges it posed. With out-of-order execution and superscalar architecture as a goal, we aimed to also provide a flexible memory hierarchy that can support multiple read/write ports to the pipeline, and to maximize the use of the memory bandwidth.

Work was divided around the stages of the pipeline and between the pipeline and memory. These are places in the design that operate with reasonable interfaces to the rest of the design, so modules here can be created without interfering with the rest of the modules. The authors work together to integrate individual modules together, and general debugging was performed by all members. For each weekly report, all three members contribute on what they did for the past week.

Milestones

I. Supporting Out-of-order Arithmetic Instructions

We first focused on creating the basic infrastructure to handle arithmetic instructions, as arithmetic instructions provide single-cycle, processor-contained results that is more easily debuggable. The structures created for this checkpoint are the following: fetch, decode, register rename, re-order buffer, issue window and issue select logic, register file, and execution unit (at this point a single ALU). All of the aforementioned structures are parameterized for easier implementation as well as tuning:

- NUM_ENTRY = 16 for the sizes of ROB and issue window
- NUM_DISPATCH = 2 for the up to two arithmetic instructions allowed to dispatch from issue
- NUM_ISSUE = 2 for dual-issue

Fetch: Fetches two instructions from icache at a time. The Branch Unit is also responsible for translating a branch mispredict into a flush mask.
Decode: Translates the instruction to the control word
Rename: Removes false dependencies by assigning the instruction's ISA register to one of the free physical registers
Reorder Buffer: Circular buffer that keeps track of all in-flight instructions in-order, and commits instructions as they become the oldest instructions and thus non-speculative
Issue Window: Pseudo-queue of all in-flight instructions which marks an instruction ready for dispatch when its source operands are ready
Issue Select Logic: Selects NUM_DISPATCH ready instructions from issue window to dispatch
Register File: parameterized file with enough read and write ports to support all dispatch instruction types

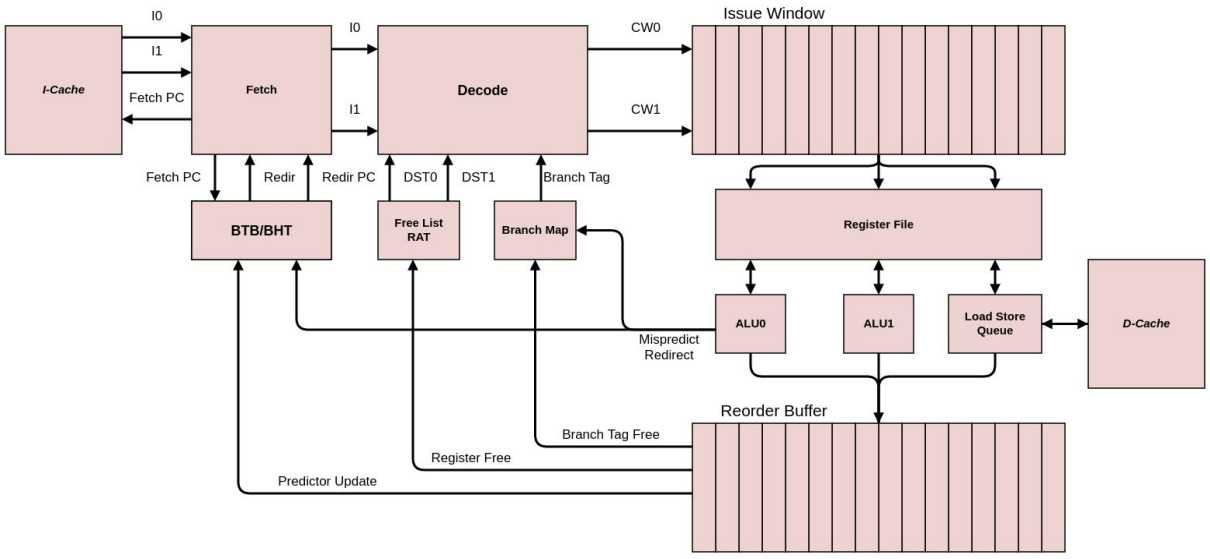


Fig. 1: Diagram of the QuantumTsunami Out-of-Order Processor

II. Supporting the Remainder of LC3-b ISA

In the second checkpoint, we added support for control instructions and memory instructions, and increased NUM_DISPATCH to three: 2 arithmetic instructions, where the lower instruction can be supplanted by a control instruction, and 1 memory instruction.

Implementation of Control Flow Instructions

After we had correct superscalar execution of arithmetic instructions working, we moved on the implement the control flow instructions: jumps, subroutine calls, branches, and returns. We chose to allow speculation across 3 branches, which allows us to keep the issue window full in branch-heavy code. The high-level features of our implementation are as follows:

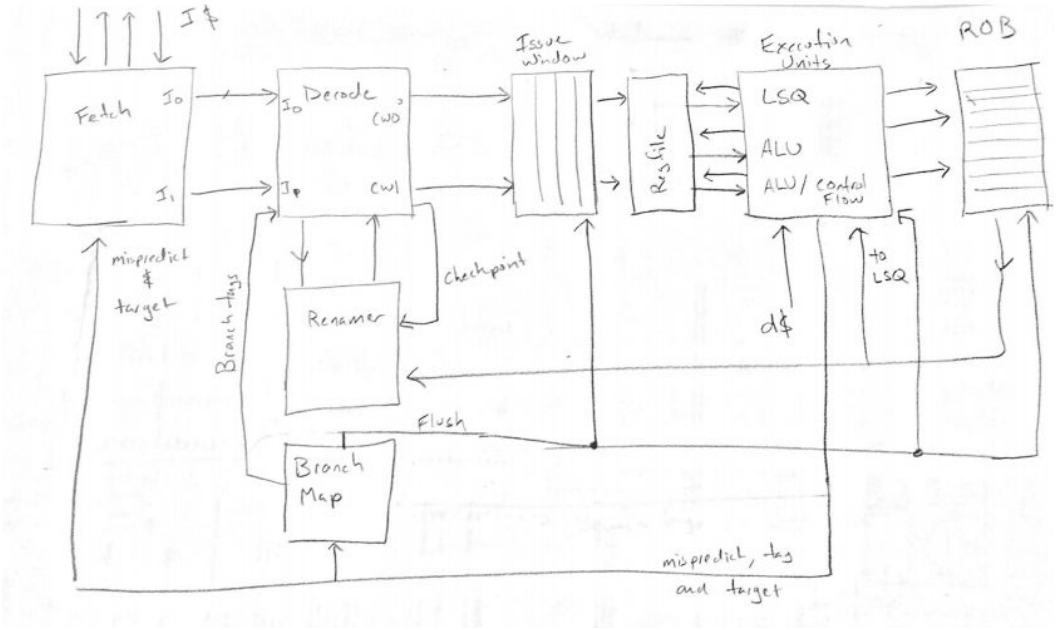


Figure 2: Block Diagram of Control Flow Signals

- Speculation across 3 in-flight control-flow instructions
- Target speculation supported for all instructions, allowing future integration of a branch target buffer (BTB)
- Mispredictions handled using branch tags and checkpointing

Our design features a 16-entry reorder buffer, meaning we will often be processing 16 in-flight instructions at a time. It is important to be able to speculatively execute across branches to keep the issue window filled with instructions. This allows us to continue fetching, decoding, completing instructions before we are certain they belong to the true instruction stream.

Branch Tagging

Each instruction's control word includes a branch tag, which is a one-hot signal marking which basic block an instruction belongs to. The branch tag stored in the issue window and the ROB, and is passed along with an instruction through the execute stage. During decode/rename, each instruction is assigned a branch tag. Branch tags are allocated by the branch map unit. Branch tags are reclaimed if an older tag mispredicts or if the branch that owns the tag commits.

Branch Resolution

Branches are resolved once they are issued in ALU0. We have two ALU's in our design to process arithmetic instructions. For simplicity, only ALU0 supports branch resolution. Because of this, branch instructions are given priority in the wakeup logic. ALU0 computes the target of the branch and compares it with the predicted target. If there's no difference, nothing special happens and it is marked as completed in the ROB. If the target mismatches the predicted target, a mispredict signal is broadcast, and the instruction's branch tag is sent to the branch-map unit. The branch map unit determines which tags are younger, and broadcasts a flush mask to all units on the following cycle.

Mispredict Recovery

Checkpointing of the RAT and register free list is used to recover from mispredicts. Our core supports speculation across up to 3 branches, but this is parameterized for easy tuning. In our current implementation, the RAT and free list are duplicated 4 times (one for each branch tag). The current branch tag is used to index into the renaming structures in decode.

Whenever a control instruction is decoded, a new branch tag is retrieved from the branch-map unit. The current and next branch tag are sent to the RAT and free-list, as well as a checkpoint signal that is asserted. On the following cycle, the contents of the renaming structures are copied from the old index to the new index. Subsequent decodes will use the new RAT and free list.

When the branch map receives the mispredict signal and tag from the ALU, it broadcasts a flush signal, flush mask, and flush tag. Also, the current branch tag is restored to that of the mispredicting branch. The RAT and freelist are restored quickly by the act of resetting the current branch tag. The flush signal and flush mask are broadcast to the entire pipeline: decode, issue window, execution units, and ROB. Any branch tag that matches the mask is invalidated.

Fetch Redirection

The fetch unit is redirected on the cycle following the assertion of the mispredict by the execution units. The execution units send the fetch unit the correct branch target, and fetch is resumed on the following cycle.

Load Store Unit

The main goal of the load store unit is to support correctness, i.e. not writing to regfile upon LD instruction invalidate and not writing to memory until ST instruction is the oldest instruction in ROB (and thus non-speculative). Moreover, we want to support forwarding as much as possible, in order to minimize the number of potentially expensive memory accesses. Forwarding is made even more important since LDs are prioritized over STs, and therefore it is quite likely that the data the LD is looking for resides in the store queue rather than in memory.

Hardware Structures

Load Queue, Store Queue
Circular buffers that contain arrays to store all usable information for each memory instructions, including its control word, ROB ID (marker for its age), address, data (for stores), and others.

Dependent Store Generator
Generates a bit vector of older store instructions (by comparing ROB IDs of active stores in the store queue, the dispatch ROB ID, and ROB's head and tail pointers) for the dispatch instruction that is loaded with its address and/or data into the queues.

Memory Request Arbiter
Picks a request to grant from the request vector.

Memory Datapath
At high-level, this resembles the MAR/MDR registers and muxes of the basic LC3-b datapath. It additionally holds other logic, such as combinatorial mechanisms for obtaining the store data as forwarded data, constructing a second bit vector for older stores for the second stage of LDI, and latching all relevant signals.

Memory Control
At high-level, this resembles the control FSM of the basic LC3-b control. It holds a decreased amount of states to only handle memory instructions, and also includes new states to handle forwarding.

Load Store Instruction Flow

Instruction Allocation
When instructions are allocated space in the ROB, they are also allocated space in the load queue or store queue. This guarantees that only instructions that have been granted space in the ROB can be dispatched to execution, which allows us simpler logic for sending acks back to the issuewindow (in other words, every memory instruction dispatched from the issue window definitely exists in the queues and can be immediately handled/acknowledged). This necessarily requires both queues to each be the same size as the ROB (for the very unlikely scenario that the ROB is completely filled with either loads or stores; alternatively, we could have made the queues smaller while sending acks to potentially block new instructions from allocating, and thus decrease the critical path in this section).

Instruction Dispatch
When an instruction is selected by issue logic and dispatched to memory execution unit, its target address is calculated and stored with its entry in the appropriate queue entry. For stores, the data from the appropriate register is also stored in the queue. We also create a bit vector of all older active stores in the store queue, as later candidates to forward from.

Memory Request
LDs and STs have different memory request characteristics, and thus they are split into two queues. LDs can issue memory requests as soon as their addresses are available, but STs can only issue memory requests when they additionally are the oldest instruction in ROB, since memory writes cannot be undone. STIs and STBs furthermore cannot allow younger LDs to forward from them until they are fully resolved, i.e. STIs have their 2nd target address and STBs have their entire correct word.

Memory Request Arbitration
The arbiter prioritizes LD requests, since faster resolution of LDs makes subsequent instructions that depend on their memory read values ready for execution sooner.

Memory Request Grant
Once an instruction has been granted, it broadcasts the granted instruction so that the queues can mark the appropriate entry invalid, and so that ROB can mark it as non-busy and commit the instruction if that instruction were a ST. Upon LD grant, its value can either be forwarded from another ST instruction in the store queue, or be read from memory. Upon ST grant, it will issue a memory request.

Possible Load/Store Hazards

During the implementation of load-store execution at this checkpoint, we realized several hazards that needed to be taken care of, as listed below.

- A possible inconsistency issue can occur where the actual ST instruction to forward from has not been dispatched yet, but the LD and an older ST that both map to the same target address have. The LD will forward from that older value incorrectly.
    - Resolution: LDs must wait until all valid, older STs have their target addresses resolved (TSO - total store ordering.)
- Implement STI target resolution - it must either block all younger loads, or resolve its final address from other older store queue entries
- If there's a flush in a middle of a long memory access, LD can overwrite a register that has since been reassigned.
    - Resolution: Do another check of the load queue entry's valid bit after memory access before writing to the regfile.

III. Caching, Memory Hierarchy, Branch Prediction, Load Store Optimizations

Our third checkpoint included adding a two level cache, branch prediction logic, and further optimizations to the load store queues. By the end of this checkpoint, our core was functional on the CP2 testcode.

Cache Design and Implementation

By the third checkpoint, our design included a functional memory hierarchy between the pipeline and the main memory. The caches were four way set associative and were parameterized for size and width, the icache was generally large, with a banked dcache, large L2 and arbiter to control access to the L2. At this point the L2 cache was directly interfacing with the main memory. All of the caches were instances of a four way cache module. The banked cache was constructed from two cache modules. Critical and logic heavy components were subject to stringent testing; the plru module and four way cache module each had their own test bench in which all possible cases were tested to ensure that they would integrate into the overall design properly.

The banked cache interface has two ports to interface into the pipeline, but share a line into the L2 cache. Two caches are in the banked cache, each with a reduced addressability, with one cache addressing onto all odd words, and one cache addressing onto all even words. Either port can address into either cache, with the second bit acting as a request into either cache. The lower port has priority on the lower cache, and the upper port has priority on the upper cache, so any conflicts will probably wash out in the end. This is implemented through a controller that internally grants requests to a port, and uses a mux-demux configuration to route the request onto the proper cache. If there is no conflict in the cache request between the ports, then they can be granted simultaneously. Access to the L2 cache is given to either cache that needs it, with priority being given to the lower cache request. The control and data signals are roued to the grant. If the same word is being accessed across both banks, then the return line will be shared. Write logic in the L2 cache was changed to allow for reconstruction of lines being written from the banked dcache.

The icache has one port that returns two consecutive words, addressing into the icache is effectively reduced by one. The Arbiter works to move the control signals between each cache and looks to resolve simultaneous requests and accesses. Because each of the cache's functionality largely mirrored that of the four way cache module, a single testbench was created that tested the entire memory hierarchy. This hierarchy was stored in a single module with interface signals for the pipeline and memory; this was done so that it could easily be switched out with later memory hierarchies for comparison or fallback purposes.

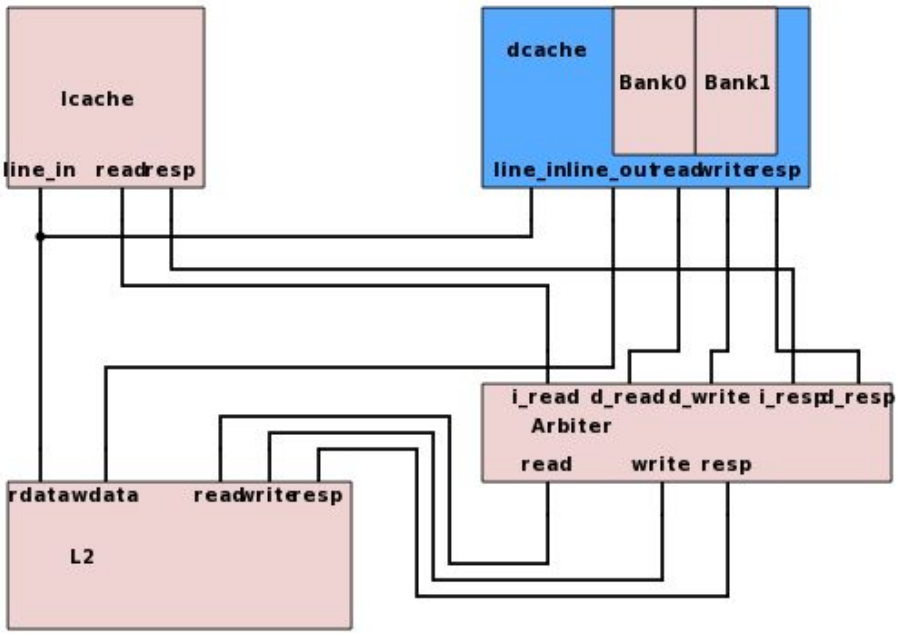


Figure 3: The memory system internal organization

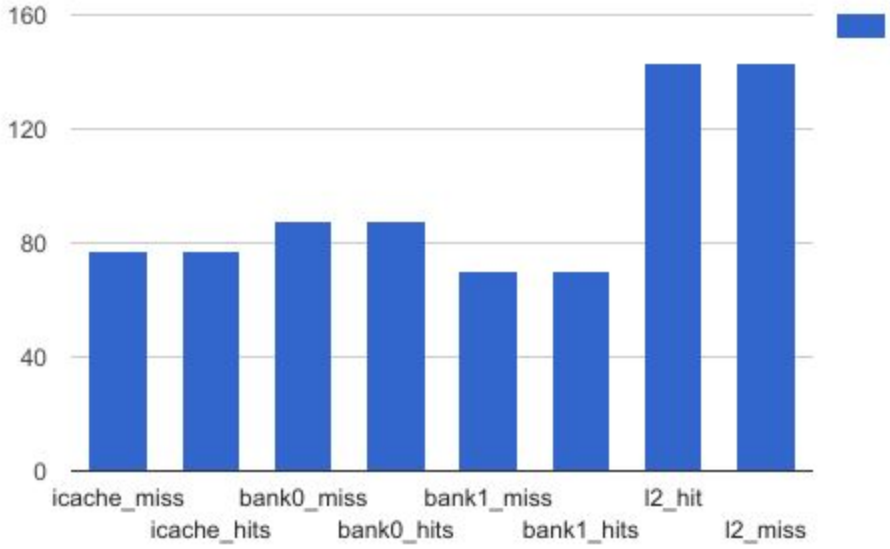


Figure 4: Cache statistics from the mp3-final testcode

Design and Implementation of a Branch Target and Direction Predictor

Because our core issues twice as many instructions per cycle as the standard pipelined LC3b design, branch mispredictions are especially costly. To reduce the prediction penalty, we added a 32-entry fully associative BTB and a 256 entry gshare branch predictor to help reduce the misprediction penalty.

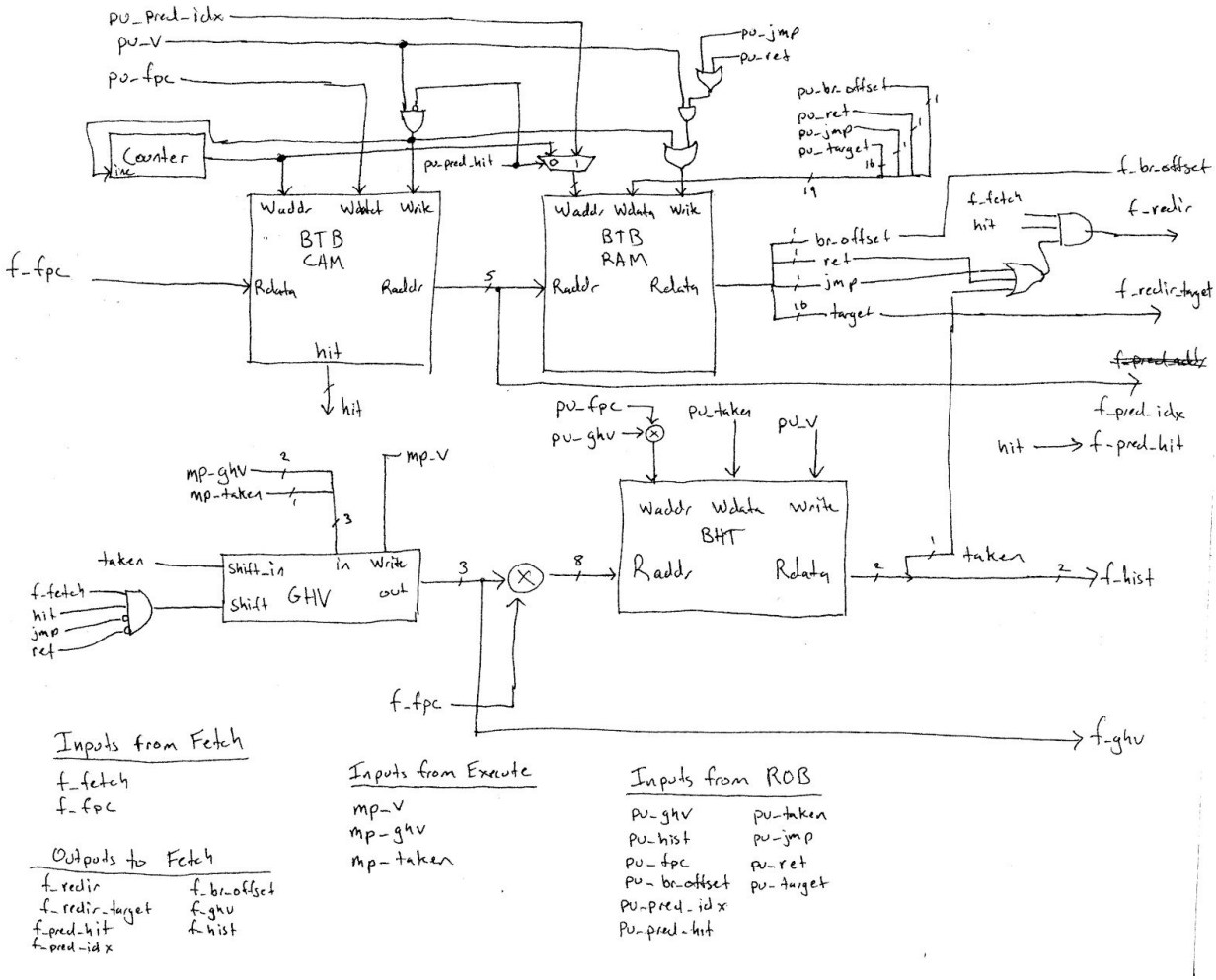


Figure 5: BTB and BHB Block Diagram

BTB Architecture

The BTB is made of a CAM that stores the branch target, decoded branch type, and branch offset within the fetch packet. The offset field is used to identify which instruction in the fetch packet is the mispredicted branch, so that subsequent instructions can be squashed. For all conditional branches, the target is the target of the taken branch. There are two bits for branch type identification: RET and JMP. JMP signals that the jump is unconditional and the gshare predictor should not be used. RET signifies that the instruction is a return. This is designed to be used with an RAS which has yet to be implemented.

Gshare Predictor

The gshare predictor uses an N-bit global history vector to help the Branch History Table (BHT) make predictions based on branch context as well as branch address. The Gshare predictor is implemented using a shift register, where for each conditional branch a 1 is shifted in if the branch is taken, and a 0 if not taken. The GHV is updated in the fetch stage in order to keep the GHV in sync with the current speculative fetch state. Without a snapshotting method of restoring the GHV, mispredicted branches would pollute the GHV and reduce the accuracy of the predictor. To solve this, we store the GHV in the control word, and send it back to the fetch unit on a branch misprediction, whereupon it is restored.

Predictor Update Mechanism

Since our core supports speculative execution, we must be careful when we update the BTB and BHT upon a branch resolution. We support multiple levels of branch speculation within our core, so younger branches often resolve before older branches. If predictor updates were done upon completion, this would pollute the predictor. To mitigate this, we hold off all predictor updates until commit. This has two negative consequences. The first is that there is an 1+n cycle delay added to the predictor updates, which can reduce prediction accuracy on tight loops. The second is that additional branch prediction state must be stored in the ROB.

STI/STB Hazards

While LDs obviously need to block until all older STs have been resolved, that in fact holds true for STI/STBs as well.

STB

A case of the hazard is shown below.

>> STR $x, STB $x, LDR $x

Simply forwarding from the youngest of the oldest matching STs will cause the LDR to load a half-correct, half-incorrect word. Thus, STBs need to forward data from the next oldest (a) STR/STR; or (b) STB that writes to the other byte, and overwrite that forwarded data with its own byte; then it is allowed to assert memory request. This is so that LDs that may forward from it get the correct full word.
If the STB forwards, it has essentially degenerated to an STR. This is indicated by a bit indicator so that the memory control can take the correct STR state transition. Else, the STB goes through the normal STB operation and does not allow younger LDs to proceed until it is completed/committed.
Alternatively, STBs can simply block all younger LDs from proceeding, which seems to be sub-optimal and was therefore not designed for.

STI

STI needs to only assert ready when both its addresses have been resolved. The simplest way of implementing this is to indiscriminately block STIs and wait for it to be the oldest instruction and thus safely execute/commit; the result is that if there are any STIs in the store queue, all younger LDs will block until the STI is cleared. While this is safe behavior, it causes undesirable latency, which may

moreover be unnecessary latency if the second STI address can in fact be resolved by an earlier ST in the store queue. Thus, we attempt to utilize that optimization: we search for matching addresses once all older STs have been resolved; if found, we set high a bit indicator and forward the youngest data as the new address, and mark the ST entry as ready (thus allowing younger LDs to proceed). Otherwise, it must wait until it is the head instruction, and all younger LDs are necessarily blocked. Upon memory issue of the STI, if the bit indicator is high, directly write the data to that address (a.k.a STR); else, go through the entire STI indirect process.

This optimization yields two-fold speed-up if taken, allowing LDs to assert memory requests sooner and having a fewer-cycle memory operation for that STI.

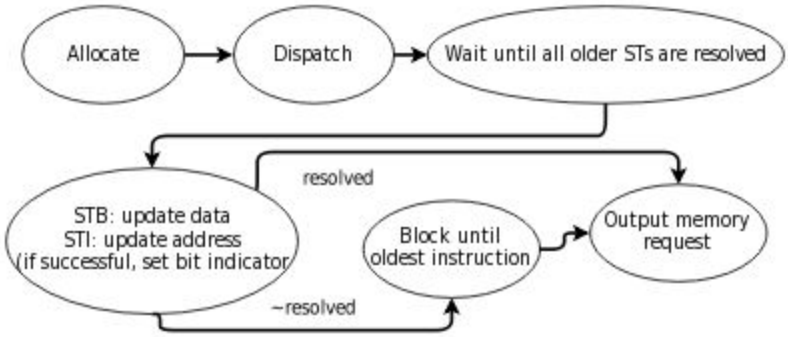


Fig. 6: Control flow for STI/STB instructions in LoadStore execution unit

While this method reduces the number of memory accesses and instruction blockage, it also severely impacted our clock frequency. Initial timing analysis of the resulting layout showed the STB data forwarding to be the processor's critical path, bringing it down to 80 MHz. Improvements could obtained by either decreasing the size of the store and load queues, and/or implementing a dynamic bit vector that populates as instructions mark themselves resolved, rather than being a bundle of combinational logic that checks each store queue entry against every other store queue entry in each cycle.

IV. Testing & Tuning

Verification Process and Custom Test-o-matic Script

At this stage in the design process, our design was beginning to resemble a finished LC3b microarchitecture. Our next task was to verify the design against a known good model of the LC3b core. The ECE 411 course staff provided us with a Test-o-matic script which compares register file transactions against a core model and a software simulated core, allowing one to identify erroneous register transfers during the execution of a test program. Because this script assumes a physical register file with only the 8 architectural registers specified in LC3b, and moreover assumes that all register transfers are performed in program order, this script is not compatible with our design.

In order to verify our design, we wrote a custom testbench and script to achieve the same functionality as Test-o-matic on our out of order superscalar core. To do this, we first attach synchronous processes in the top level testbench that monitors commit outputs from the ROB. Once an instruction commits, a log message is generated to standard out containing the disassembly of the instruction that was committed, the virtual and physical register numbers, and the source and destination register contents.

After a simulation is run, the output of the testbench is filtered for commit messages. The commit messages are run through a simple perl script that re-interprets the architected register state from the destination register values and the virtual destination register. This state is then reconstructed and printed to a log file.

Finally, the assembly program is run on the software core model provided by the course staff, and the register file transactions are saved. To verify the correctness of the design, we simply made sure that the register transactions matched.

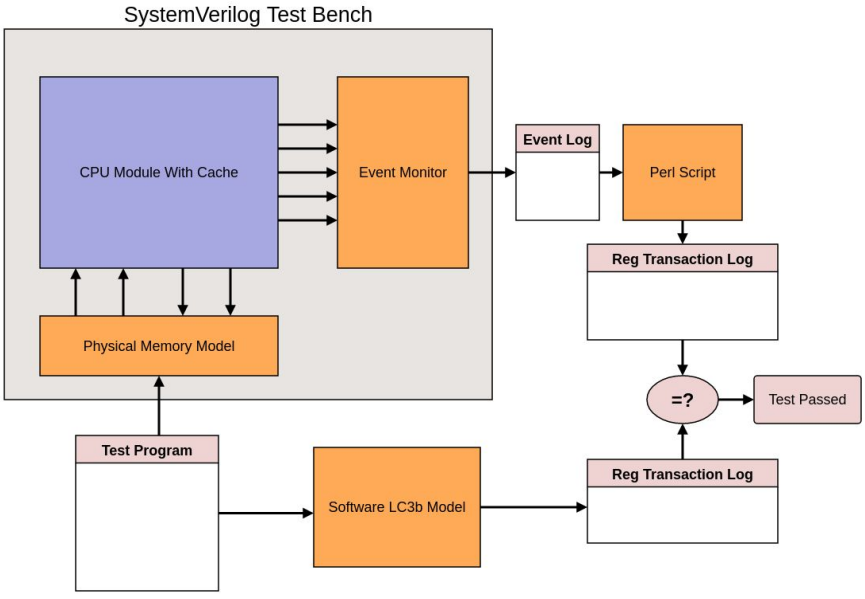


Figure 7: Diagram of Verification Setup

LoadStore Improvements

Testing the processor showed that our critical paths were in the load store execution unit, which brought the frequency down due to the heavy combinatorial logic in the constant checking of relative age and forwarding data within the module. Several techniques were considered and a couple were implemented with the time that we had, which are listed below.

Dual Memory Operations

To take advantage of the dual-port cache, the processor was updated to be able to issue up to two simultaneous memory operations. This involves making two copies each of the dependent store generator, datapath, and control, one for each dispatch instruction. Since STs are always one-hot and the oldest instructions, we can safely prioritize STs if the simultaneous memory operations are ST + LD.

The resulting waveforms of some test codes showed satisfying zipper and parallel effects:

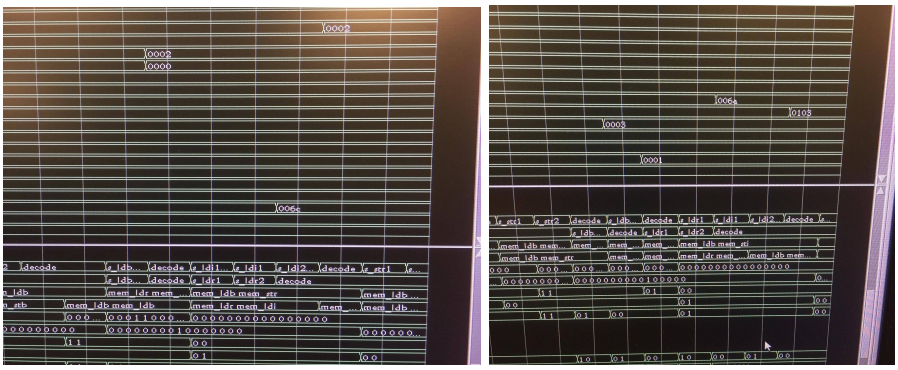


Figure 8: Zippered and parallel dual-memory operations

Testing of this implementation showed performance improvement on the MP3-final code: when we ran it on magic memory, the single-memory issue version completed at 623806ns, and the dual-memory issue version completed at 585736ns. This is an IPC improvement of ~6.4%.

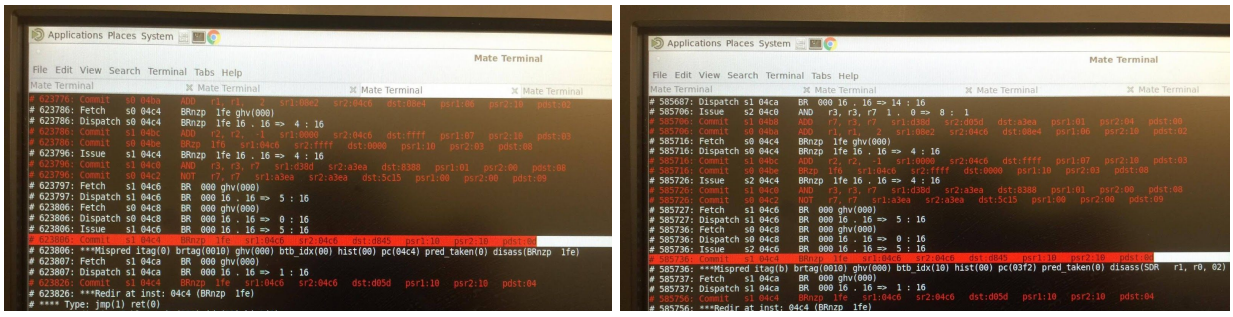


Figure 9: IPC improvement of 6.4% going from single-memory issue to dual-memory issue

Breaking LoadStore Critical Paths

After implementing dual-memory issue, timing analysis showed that the critical paths existed in the combinational logic of the LoadStore execution unit. We resolved them in sequential order, and report the improvements in clock frequency that resulted. We started off with clock frequency of 80.65 MHz.

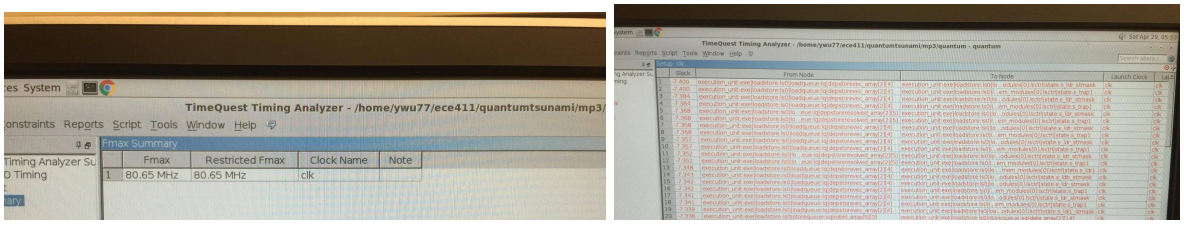


Figure 10: 80.65MHz critical path in load store

1) We first observe that the critical path extends all the way for the dependent store vectors to the memory control states. To break it up, we pipeline the load store execution unit into four stages: Dispatch/Allocate, Queues, Request Arbiter, and Datapath/Control. Individually, the queues and datapath/control are the most combinatorially-heavy modules, and lie on the critical path.

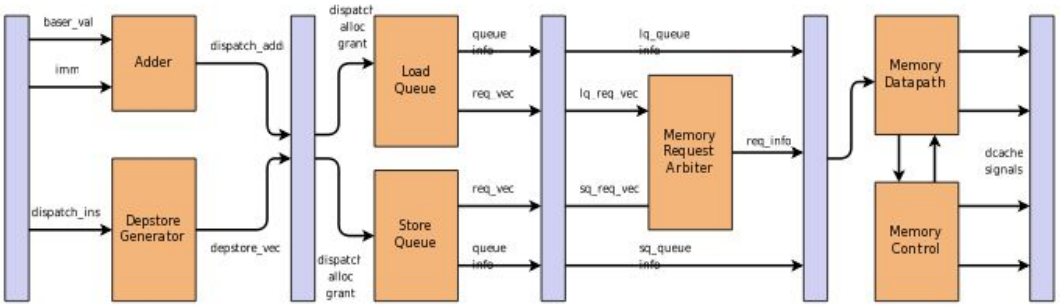


Figure. 11: Simplified diagram of the pipelined load store execution unit

This garnered us about 3 MHz of improvement, and an infrastructure on which to make more improvements.

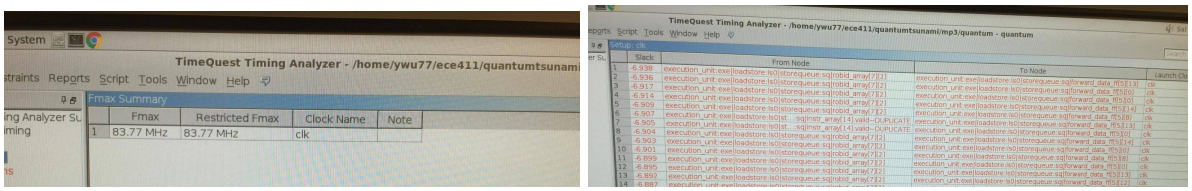


Figure 12: 83.77MHz critical path in load store

2) This next critical path lay in the data-forwarding logic within store queue for STBs to resolve correctly, which was entirely combinatorial. While this has potential resolutions mentioned earlier in the document, we implement a simple latch fix to break this critical path. This incurs a two-cycle delay: the chosen

forwarding index for each store queue entry is latched, which is then used to index into the store queue's data array to obtain the correct forwarding data, which is finally used to latch into the data array upon indication that it is allowed to forward.

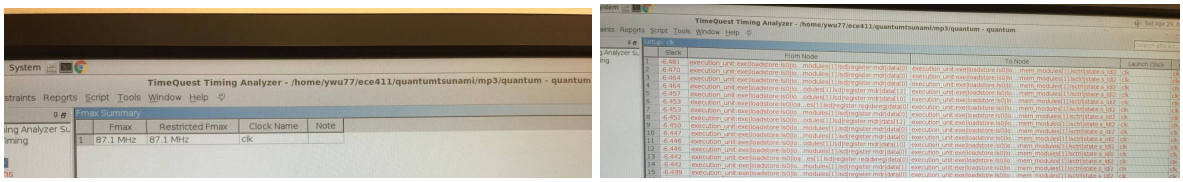


Figure 13: 87.1MHz critical path in load store

3) The resulting critical path lay between the memory datapath and memory control. Specifically, it was due to the use of the MDR's data, requested index, and the s_ldi2 state; we determine whether to go to the s_ldi2 state or s_ldi2_stmask state based on whether or not it is allowed to forward by constructing a bit vector of older store queue entries with matching addresses of the newly acquired 2nd target address in MDR. We break this critical path by latching that constructed vector, which finally brought us to the 100MHz threshold.

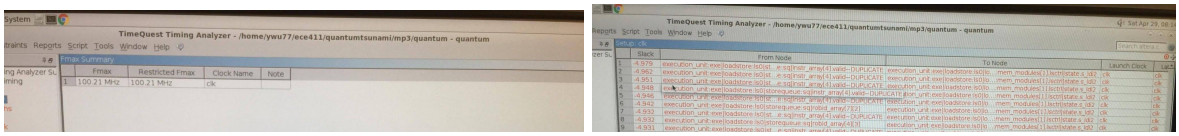


Figure 14: 100.21MHz critical path in load store

Future Work

There were several features that we had planned on implementing and testing, but ran out of time. We include our thoughts about them in this section.

Store Commit Buffer

Since we prioritize LD memory requests and only allow STs to issue memory requests when the ST is the oldest instruction in ROB, this causes the ROB to fill up quickly and only slowly drain when ST is at the head of the ROB.
To remove this bottleneck, when we mark the SQ entry as committed, we also mark the corresponding ST instruction in ROB has completed. This allows the ROB to free up the blocking ST entry.

- Memory requests is constructed from the Store Commit Buffer (oldest first)
- Other instructions prioritize forwarding from SQ (since it will always have younger instructions) before forwarding from SCB.
    - I.e. if no matching addresses in SQ, then check SCB.

STB Optimization

Currently, STB will stall all younger LDs if it did not successfully forward data from older entries in the store queue. This is unnecessary if younger LDs do not have the same target address; an additional bit indicator for each store queue entry can be added to mark whether it is resolved (i.e. safe to forward from).

- LDB: cares about STB that writes to the same byte, or STR/STI
- LDR/LDI: cares about all STR/STI/STB

Collapse Stores

Stores to the same address could be collapsed to the youngest of all the stores to minimize unnecessary memory writes. This feature would be quite amenable if the Store Commit Buffer was first implemented.

Victim Cache

Although the modules and logic exist to support this cache, there was not sufficient testing to allow it to be implemented into the memory system. A small, fully associative cache that was meant to be placed between the L2 and main memory, this cache helped reduce critical paths by placing the write-backs after reads. L2 functionality was also changed so that write-backs from L2 were never performed. Instead, the L2 and Victim Cache simply swapped lines, and allowed the Victim cache to perform the write backs

Collapse Buffer

We had originally wanted to implement the issue window as an collapsing buffer, because it would give priority to older instructions to get issued first, and therefore reduce the chance of executing younger, incorrect instructions over older, correct instructions. However, we were not able to get the logic correct in time for the first checkpoint, and thus this piece of logic was deprioritized. An existing collapse buffer implementation that was not integrated into our project functioned by taking in an array of valid bits corresponding to the valid bits of a buffer, and created a partial sum for each entry in the buffer of all previous valid entries. This partial sum corresponds to the next index for a given entry in the buffer.

Conclusion

In these six weeks we spent designing, implementing and testing an out-of-order superscalar processor along with a cache hierarchy in ECE 411, we have strongly developed both hardware design and verification skills. We've heightened our awareness of how dependencies manifest in unbounded ways when there is no enforced program order to the instruction processing, and managed to devise robust solutions to the ones that we thought of or happened across. We've designed tools to help us complete our (relatively) unorthodox project, become more proficient at tracing the source of hardware bugs, and spiritedly considered various optimizations inspired by both apparent design inefficiencies and brainstorming.

At the end of these six weeks, we successfully created a single-memory issue out-of-order processor that works with all MP3 code and competition codes with our cache hierarchy which runs at 80 MHz (with unoptimized compile); we have also created a dual-memory issue version which runs at 93 MHz (with unoptimized compile) that works with all codes with magic memory, and works with all MP3 code and competition-2.asm code with our cache hierarchy.