EECS 470 Final Project Report

Group 8

1. Overview

Our processor meets all the specified minimum requirements for this project - instruction cache, data cache, multiple functional units with varying latencies, out-of-order implementation, and branch prediction with address prediction. For advanced features, we targeted arbitrary superscalar execution and early branch resolution and were able to implement it with full functional support. Our processor works flawlessly with all values of N (given that we have tested it from N=1 up to N=8 with varying memory latencies and compiler optimization options). It passes all the provided .s and .c test cases in simulation as well as post-synthesis.

We implemented multiple additional features that gave a major boost to our processor's performance. For the fetch stage, we implemented hardware prefetching and a non-blocking multi-ported L1 instruction cache, which boosts the front-end performance of our processor. To improve the accuracy of our branch predictor we implemented a parameterized set-associative cache based branch target buffer (BTB). For the load/store handling, we implemented a split load buffer and store queue (LSQ) component that gracefully handles the in-order processing of memory instructions. The LSQ interfaces with an advanced non-blocking, write allocate, write through L1 data cache. We also implemented a visual debugger that prints the state of all relevant components of our processor for a specified range of clock cycles, which made debugging much less of a hassle while working with the complete pipeline. We leave it to the instructors to decide if our debugger counts as an additional feature.

2. What we did

2.1 Core

The core of our advanced out-of-order computer processor (shown in Figure 1) consists of multiple stages. These stages are fetch, decode, dispatch-1, dispatch-2, execute, and commit. In this breakdown, we lumped data memory into the execute stage. Our processor supports N-way superscalar execution and early branch resolution. These advanced features permeate every aspect of our processor.

The fetch stage occurs in the top-level fetch module, which instantiates the I-cache controller and the I-cache. The I-cache controller instantiates the bimodal branch direction predictor and the BTB. We have a 128 byte non-blocking, direct-mapped I-cache. The BTB is a 16-line, 8-way set associative cache with pseudo LRU. The branch predictor and BTB are updated when a branch instruction commits. Our processor prefetches instructions into a fetched instruction queue with 16 entries, and our processor prefetches past branch instructions that are predicted to be taken. We examine the fetch stage in greater detail later in the report. The fetch stage sends a maximum of N instructions from the head of the fetched instruction queue to the decode stage. The decode stage consists of N decoders and a decoded instruction queue with 16 entries. The outputs from these decoders and the inputs from the fetch stage are stored in the decoded instruction queue. The decode stage sends a maximum of N instructions from the head of the decoded instruction queue to the dispatch-1 stage.

The dispatch-1 stage occurs in the top-level dispatch module, which instantiates the register alias table (RAT), the branch register alias tables (BRAT), and the physical register file (PRF). Our processor maintains 8 BRATs and 128 physical registers. Instructions that arrive from the decode stage first pass through the RAT. For each of these N instructions, the RAT renames the source operand registers and the destination register to physical registers. The RAT maintains a free list and a mapping of architected registers to physical registers. During normal branch resolution, the RAT restores its free list and mapping from the retirement register alias table (RRAT). During early branch resolution, the RAT restores its free list and mapping from a BRAT.

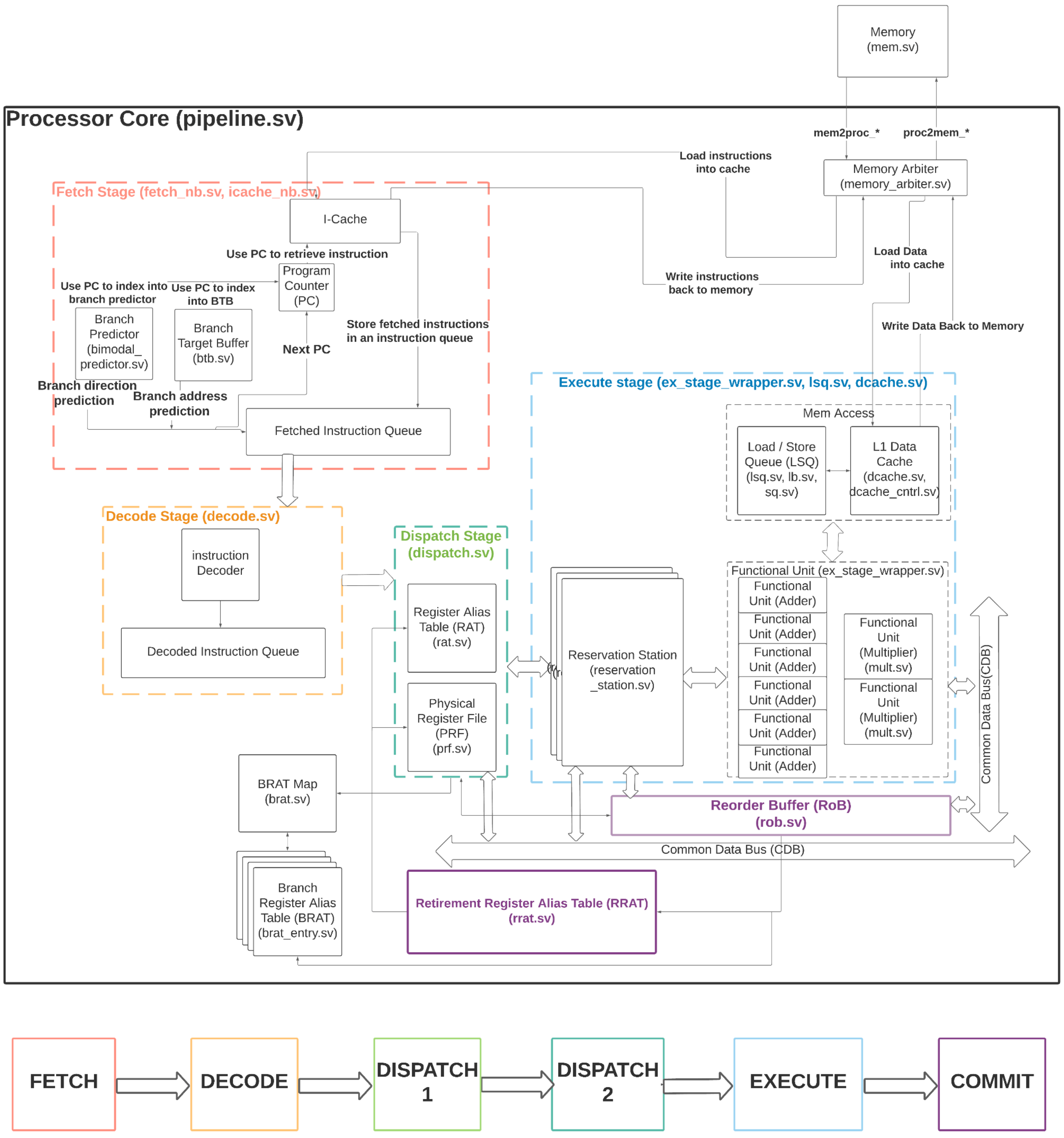

Memory (mem.sv)
Processor Core (pipeline.sv)
mem2proc_*
proc2mem_*
Load instructions into cache
Memory Arbiter (memory_arbiter.sv)
Fetch Stage (fetch_nb.sv, icache_nb.sv)
I-Cache
Use PC to retrieve instruction
Program Counter (PC)
Use PC to index into branch predictor
Use PC to index into BTB
Branch Predictor (bimodal_ predictor.sv)
Branch Target Buffer (btb.sv)
Next PC
Store fetched instructions in an instruction queue
Write instructions back to memory
Load Data into cache
Write Data Back to Memory
Branch direction prediction
Branch address prediction
Fetched Instruction Queue
Execute stage (ex_stage_wrapper.sv, lsq.sv, dcache.sv)
Mem Access
Load / Store Queue (LSQ) (lsq.sv, lb.sv, sq.sv)
L1 Data Cache (dcache.sv, dcache_cntrl.sv)
Decode Stage (decode.sv)
instruction Decoder
Decoded Instruction Queue
Dispatch Stage (dispatch.sv)
Register Alias Table (RAT) (rat.sv)
Physical Register File (PRF) (prf.sv)
Reservation Station (reservation _station.sv)
Functional Unit (ex_stage_wrapper.sv)
Functional Unit (Adder)
Functional Unit (Multiplier) (mult.sv)
Common Data Bus(CDB)
BRAT Map (brat.sv)
Reorder Buffer (RoB) (rob.sv)
Common Data Bus (CDB)
Branch Register Alias Table (BRAT) (brat_entry.sv)
Retirement Register Alias Table (RRAT) (rrat.sv)
FETCH
DECODE
DISPATCH 1
DISPATCH 2
EXECUTE
COMMIT

Figure 1: The processor architecture diagram

Instructions that arrive from the decode stage also pass through the BRATs. A top-level BRAT wrapper module maintains 8 BRATs. Our processor attempts to allocate a BRAT for each branch instruction that arrives from the decode stage. If all BRATs are in use, any branch instruction that arrives from the decode stage will simply not receive a BRAT. In this fashion, BRATs can never create a structural hazard. We examine BRATs and early branch resolution in greater detail later in the report.

The outputs from the RAT, outputs from the top-level BRAT wrapper module, and inputs from the decode stage are stored in a circular buffer in the dispatch-2 stage of the top-level dispatch module. This circular buffer has 16 entries. The dispatch-2 stage sends a maximum of N instructions from the head of this circular buffer to the reservation stations, reorder buffer (ROB), and load-store queue (LSQ). However, the dispatch-2 stage will only send a maximum of one load instruction or one store instruction per clock cycle. Along the way, these N instructions pass through the PRF and receive either a value or a physical register number for each of their source operands.

A top-level reservation station wrapper module maintains 32 reservation stations. Each instruction that arrives from the dispatch stage occupies a reservation station. Each reservation station listens to all N common data buses (CDBs) for its source operands. Once a reservation station has its source operands, it asserts one of three wakeup signals. The wakeup_mult signal indicates to the execution stage that the reservation station holds a multiply instruction. The wakeup_mem signal indicates that the reservation station holds a load instruction or a store instruction. The wakeup_add signal indicates that the reservation station holds some other instruction. If the execution stage selects a reservation station for execution, then that reservation station sends off its data.

A top-level execution stage module instantiates two pipelined 4-stage multipliers, 2N adders, and N branch condition checkers. The top-level execution stage module receives all three wakeup signals (wakeup_add, wakeup_mem, and wakeup_mult) from each reservation station. Our processor selects a maximum of N reservation stations asserting the wakeup_add signal for execution. These reservation stations use N of the adders (general-purpose adders) and the N branch condition checkers. Our processor also selects a maximum of N reservation stations asserting the wakeup_mem signal for execution. These reservation stations use the other N adders (memory adders). Finally, our processor selects a maximum of two reservation stations asserting the wakeup_mult signal for execution, which use the two multipliers. The top-level execution stage module also drives N CDBs numbered from 0 through N-1. During CDB arbitration, priority is given to the multipliers, then load instructions from the top-level LSQ module, and then general-purpose adders. The two multipliers contend for CDBs N-1 and N-2 with a preference for higher-numbered CDBs. Load instructions from the top-level LSQ module contend for CDB N-3. The N general-purpose adders contend for all N CDBs with a preference for lower-numbered CDBs. We implemented CDB arbitration in this way to greatly reduce the complexity of our arbitration logic. Memory adders do not contend for CDBs and instead send their results directly to the top-level LSQ module. If a reservation station would not receive a CDB, then that reservation station is not selected for execution.

The top-level LSQ module instantiates a load buffer and a store queue having 16 entries each. The top-level LSQ module sends one memory request from the load buffer or store queue to the top-level D-cache module per clock cycle. The top-level D-cache module instantiates the D-cache controller and the D-cache, which is a 128 byte non-blocking, write allocate, write-through, direct mapped cache. The D-cache controller maintains 16 miss status holding registers (MSHRs) and a write allocate station. Once a load instruction receives its data from the D-cache controller, the top-level LSQ module broadcasts the result of that load instruction on a CDB. We examine the LSQ and the D-cache in greater detail later in the report.

The reorder buffer (ROB) is a circular buffer that has 64 entries. Each instruction that arrives from the dispatch stage occupies an entry in the reorder buffer. A ROB entry that corresponds to a branch instruction or a store instruction is marked as executed when that instruction computes its address. In all other cases, a ROB entry is marked as executed when the corresponding instruction broadcasts its result on a CDB. Our processor keeps track of every branch instruction in the ROB. If one or more branch instructions are mispredicted, then the ROB reports the mispredicted branch instruction that is closest to the head of the ROB. If this mispredicted branch instruction has a BRAT allocated, then the ROB initiates early branch resolution and broadcasts which BRAT corresponds to the mispredicted branch instruction. If this mispredicted branch instruction does not have a BRAT allocated and is at the head of the ROB, then the ROB initiates normal branch resolution. If one or more branch instructions are correctly predicted, then the ROB broadcasts which BRATs correspond to the correctly predicted branch instructions. Our processor sends a maximum of N instructions from the head of the ROB to the RRAT to be committed. However, our processor will only commit a maximum of one store instruction per clock cycle.

Instructions that arrive from the ROB are committed at the RRAT. The RRAT maintains a free list and a mapping of architected registers to physical registers. When an instruction with a valid destination register is committed, the RRAT updates its free list, updates its mapping, and also reports which physical register is freed.

2.2 Fetch

The top-level fetch module instantiates the fetch controller and the direct mapped I-cache. The fetch controller instantiates the bimodal branch direction predictor and BTB in addition to interfacing with the I-cache, making it the centerpiece of the entire fetch stage. Instead of having two separate circular queues (i.e., fetch queue storing instructions ready to be dispatched to the decode stage, and MSHR queue storing PCs that are missed in cache until they are serviced by the memory, we merged the backend of our fetch stage with the frontend of the I-cache controller. As a result, we had a fetch controller consisting of a single 16 entry fetch/MSHR circular queue that fundamentally worked on four pointers. The 'head' and 'tail' pointers track the entries allocated in the fetched instruction queue, the 'send pointer' tracks the cache misses to be sent to the memory, and the miss pointer keeps track of the fetch queue entry receiving its data back from the memory. The fetch controller integrated with a multi-ported direct mapped instruction cache allowed us to implement effective prefetching limited only by the availability of empty fetch queue entries and number of I-cache read ports.

On every clock, we check the number of instructions accepted by the decode stage, update the number of free fetch queue entries, and issue new requests to I-cache based on the free fetch queue entries (limited by the number of I-cache read ports). On a cache hit, the instruction gets stored in the fetch queue entry and marked as data_valid whereas, on a cache miss the instruction is marked as issue_to_mem. The fetched instructions are also sent to the branch predictor and the BTB; if an instruction turns out to be a taken branch based on the address and direction prediction, we discard the instructions that follow the branch instruction in that batch. This allows squashing of instructions based on address and direction prediction at the tail of the fetch queue, adding minimal overhead as opposed to any other implementation scheme. The send pointer works in parallel and issues the fetch queue entry closest to the head pointer having issue_to_mem set to the memory. The miss pointer on the other hand, keeps track of the data coming back from memory and latches onto that data if it matches with the mem_tag stored in the miss pointer's index's fetch queue entry, marking that entry's data as valid. Based on the head pointer and miss pointer, we generate the next set of instructions to be sent to the decode stage. The logic decouples loading of new instructions to the fetch queue from the unloading of instructions getting accepted by the decode stage, providing effective prefetching. The branch direction predictor and the BTB are updated independently based on the committed instructions coming to it from the ROB.

On cache misses, our controller is equipped with the logic to aggressively send a request off to memory in that very clock cycle; this caused the fetch stage to be the critical clock path during synthesis. Although, aggressive memory access gave a 15% boost in the overall processor's performance, therefore we decided to optimize the critical path by removing conditional requests to the BTB; on every clock, the BTB, branch predictor, and the I-cache, work in parallel making predictions and returning data for the maximum number of instructions that can be fetched instead of the actual number of instructions. We scrap the data for instructions which end up being invalid which costs much less on the synthesis time rather than conditionally fetching data for every instruction. The number of I-cache read ports controls the efficacy of our N-way superscalar feature, therefore the number of ports are directly dependent on the value of N_WAY. In ideal conditions, our fetch stage is capable of sending N instructions to the decode stage in every clock cycle and also fetching that many instructions from the cache in a single clock cycle.

2.3 Load-Store Queue and Data Cache

The top-level load-store queue module instantiates a load buffer and a store queue with 16 entries each. When a store instruction is dispatched, that store instruction is pushed to the tail of the circular queue. Store instructions receive their addresses and data from the execution stage. When a store instruction that has completed execution reaches the head of the ROB, the ROB notifies the store queue. Upon receiving a notification from the ROB, the store queue attempts to send that store instruction to memory. When a load instruction is dispatched, that load instruction occupies an empty load buffer entry. Each load buffer entry maintains a bit vector that indicates which store instructions that load instruction is dependent on. This bit vector is initialized when the load instruction is dispatched. This bit vector is updated when the load instruction receives its address, when a store instruction receives its address, and when a store instruction is sent to memory. Load instructions in the load buffer receive their addresses from the execution stage. When a load instruction knows its address and is not dependent on any store instructions in the store queue, the load buffer attempts to send a load instruction to memory. When a load instruction has its data, the load buffer attempts to broadcast the result of a load instruction on the CDB. The top-level LSQ module arbitrates between the load buffer and the store queue and forwards a single memory request to the top-level D-cache module. If both the store queue and the load buffer request access to memory at the same time, then the top-level LSQ module gives priority to the store queue.

The top-level D-cache module passes the request from the top-level LSQ module to the D-cache controller and the D-cache. The D-cache controller maintains 16 MSHRs and a write allocate station. If a load instruction hits in the cache, then the D-cache controller immediately returns the cache data to the top-level LSQ module. MSHR 0 holds load instructions that have missed in the cache but have not yet been sent to memory. Once a load instruction in MSHR 0 has been sent to memory and has received a response, that load instruction is evicted from MSHR 0 and instead occupies the MSHR corresponding to the response from memory (e.g., if the response from memory is 2, then that load instruction now occupies MSHR 2). In this fashion, MSHRs 1 through 15 hold load instructions that have been sent to memory but have not yet received their data. When memory broadcasts data along with a tag, the MSHR corresponding to the tag grabs that data (e.g., if memory broadcasts data along with a tag of 4, then MSHR 4 grabs that data). Once a load instruction has received its data, then that load instruction is evicted from its MSHR, and its data is sent to the top-level LSQ module.

The write allocate station holds store instructions that have either missed in the cache or have not yet received a response from memory. Due to restrictions placed on the interface with memory, our processor is only capable of writing to memory in 64 bit blocks. However, the RISC-V instruction set architecture only supports storing words, half words, or bytes. Therefore, in order to write to memory, a store instruction must know the data being stored and the 64 bit block being written to. If a store instruction misses in the cache, then that store instruction does not know the 64 bit block being written to. Thus, the store instruction waits in the write allocate station until it is able to load the correct 64 bit block from memory. Once the store instruction knows the 64 bit block being written to, the store instruction simultaneously writes to the cache and memory. If the store instruction does not receive a response from memory anywhere in this two step process (load from memory, write to memory), then the store instruction waits in the write allocate station and tries again on the next clock cycle.

The D-cache is a 128 byte direct mapped cache. Because of the MSHRs and the write allocate station, the D-cache is non-blocking, write allocate, and write through. The D-cache has two write ports and one read port. One write port is used by store instructions, which simultaneously write to the cache and memory. The other write port is used when an MSHR or the write allocate station receives data from memory. The read port is used when the top-level LSQ module forwards a request to the top-level D-cache module.

2.4 Branch Register Alias Tables and Early Branch Resolution

The BRAT wrapper maintains 8 BRATs in an unordered set. When a branch instruction is dispatched, our processor attempts to allocate a BRAT for that branch instruction. If there are no BRATs available, then that branch instruction simply does not receive a BRAT. A BRAT records the architected state of the processor at the point when the corresponding branch instruction is dispatched. Specifically, a BRAT records the mapping of architected registers to physical registers, the free list, and the tail pointer of the store queue. When a BRAT is allocated, the BRAT wrapper initializes a bit vector that tracks the BRAT's dependencies on other BRATs. In other words, the BRAT wrapper maintains a bit vector for each BRAT that indicates which BRATs came before that BRAT in program order.

Every instruction that gets dispatched receives two bit vectors from the BRAT wrapper -- the first to indicate which BRATs that instruction is dependent on, and the second to indicate which BRAT that instruction was allocated. For non-branch instructions, this second bit vector is always zero. If the ROB reports a mispredicted branch instruction that does not have a BRAT allocated, then the ROB initiates normal branch resolution. If the ROB reports a mispredicted branch instruction that has a BRAT allocated, then the ROB initiates early branch resolution and broadcasts which BRAT corresponds to the mispredicted branch instruction (i.e., which BRAT is getting nuked). All instructions in the processor that are dependent on the BRAT getting nuked are discarded. Additionally, all BRATs that are dependent on the BRAT getting nuked are also nuked.

During early branch resolution, we nuke the fetched instruction queue, decoded instruction queue, and dispatch buffer because all instructions in these buffers are guaranteed to come after the mispredicted branch instruction in program order. We also halt the front end of the processor by freezing the reservation stations. When we freeze the reservation stations, each reservation station checks its data. If a reservation station holds an instruction that is dependent on the BRAT getting nuked, then that reservation station discards its data. Otherwise, that reservation station holds onto its data. Although we permit signals from the reservation stations to propagate through the execution stage, we tell the ROB and PRF to ignore incoming data from the execution stage during early branch resolution. In this fashion, the front end of our processor makes no forward progress during early branch resolution. However, we do not halt the back end of the processor. Each load buffer entry, store queue entry, and MSHR checks its incoming data. If the incoming data is dependent on the BRAT getting nuked, then that data is discarded. Otherwise, that data is accepted. Thus, memory operations proceed as normal during early branch resolution. The only exception is load instructions, which are automatically denied access to a CDB during early branch resolution. During early branch resolution, we also allow instructions to commit as normal.

Ideally, we would have liked to avoid halting the front end of the processor. If every module could have checked its incoming data and chosen to discard or accept that data (in a manner similar to the load buffer, store queue, and MSHRs), then we could have allowed the front end of the processor to proceed as normal. However, the ROB and the PRF do not receive the BRAT dependencies of incoming data from the execution stage. As a result, the ROB and PRF cannot easily check the validity of incoming data during early branch resolution. To address this problem, we tell the ROB and PRF to assume that all incoming data is invalid during early branch resolution. Knowing that the ROB and PRF will ignore incoming data from the execution stage during early branch resolution, we freeze the reservation stations. The reservation stations try to send data again on the next clock cycle.

One alternative to freezing the reservation stations would have been to send BRAT dependencies to the ROB and PRF. However, we anticipated that this decision would result in massive fanout and hurt our clock period. Another alternative would have been to allow early branch resolution to suppress the wakeup signals from the reservation stations. However, when we previously attempted this solution, allowing branch resolution signals to propagate into our combinational logic proved to be disastrous for our clock period. While other schemes may have allowed us to avoid halting the front end of our processor, we ultimately settled on freezing the reservation stations due to the simplicity of the solution. We also suspect that freezing the reservation stations during early branch resolution has a minimal impact on performance. Because we allow memory operations to proceed and instructions to commit during early branch resolution, we suspect that freezing the reservation stations frequently goes unnoticed. Additionally, because we nuke the fetched instruction queue, decoded instruction queue, and dispatch buffer during early branch resolution, our processor takes several clock cycles before there is any demand for reservation stations.

When branch instructions are correctly predicted, the ROB checks whether or not those branch instructions had BRATs allocated. If those branch instructions had BRATs allocated, then the ROB broadcasts which BRATs correspond to the correctly predicted branch instructions (i.e., which BRATs are getting freed). All instructions in the processor that are dependent on the BRATs getting freed clear their dependencies on those BRATs. Additionally, all BRATs that are dependent on the BRATs getting freed also clear their dependencies on those BRATs. In this fashion, our processor strives to free and reallocate BRATs as soon as possible.

To facilitate freeing and reallocating BRATs as soon as possible, we decided to maintain our BRATs in an unordered set rather than a stack or a queue. In a stack implementation, a BRAT cannot be reallocated until it becomes the top of the stack. Similarly, in a queue implementation, a BRAT cannot be reallocated until it becomes the tail of the queue. Under nonideal conditions in which all BRATs are in use and branch instructions corresponding to those BRATs are resolved out of order, a stack implementation or a queue implementation can produce BRATs in the middle of the stack or queue that have been freed but cannot be reallocated. These circumstances may prevent a branch instruction from receiving a BRAT even though some BRATs are currently doing no work. Although these nonideal conditions are rare, we observed a few .c test cases (especially the sorting algorithms) in which all 8 BRATs were in use simultaneously. To avoid having BRATs doing no work, we settled on an unordered set implementation that allows us to reallocate a BRAT as soon as it is freed, regardless of the circumstances. We believed that the overhead in complexity was worth the potential performance gain.

2.5 Branch Predictor and Branch Target Buffer

Our processor utilizes a bimodal branch direction predictor and the set associative cache branch target buffer (BTB). The set associative cache utilizes pseudo LRU replacement policy updated on both writes and reads, avoiding duplicate entries in the structure. There were a couple of factors we adjusted to squeeze out the best performance from our processor: the associativity (2-way, 4-way, and 8-way) and number of cache lines (64, 32, and 16). After thorough testing of the impact on overall CPI, performance, and synthesis timing, of all the permutations, we integrated an 8-way set associative 16 line BTB.

The direction and address predictors are updated based on instructions committed from the head of the ROB, which prevents squashed branch instructions from updating the predictor. With regards to making predictions for the fetched instructions, we evaluated three positions -

- Prediction in decode stage - The primary advantage of making a prediction in the decode stage is that our processor makes the prediction closer to the execute stage. By delaying the prediction, the processor allows the branch predictor to receive more updates from committing instructions. The opportunity to receive additional updates should increase the accuracy of the predictor. However, this position greatly suffers from late flushing of fetch stage and prefetcher in case of branch taken, eventually keeping the pipeline empty by squashing more instructions.
- Prediction at the head of fetch stage - In this scenario prediction happens just before sending instructions to the decode stage. This approach also suffers from delayed flushing of the fetch stage and prefetcher but provides more recent predictions. Both the first and second approach can be perceived as making 2 predictions, (i) when instructions are put into the fetch queue wherein we are predicting - always not taken; (ii) when instructions are at the head of fetch queue or in the decode stage wherein we use the branch and address predictions.
- Prediction at the tail of fetch stage - In this case we make predictions on the instructions being fetched in the current clock cycle; if any fetched instruction is a predicted taken branch, we squash the instructions following it in that cycle and start fetching from the target PC address in the next clock cycle. The number of instructions squashed based on predictions decreases tremendously, although, we are getting slightly stale address and direction predictions as the distance between the predictions and ROB commit increases. This allows more instructions to enter the pipeline before we update our predictors. But this has a minimal overhead as the predictors get trained with fixed latency though maintaining overall throughput.

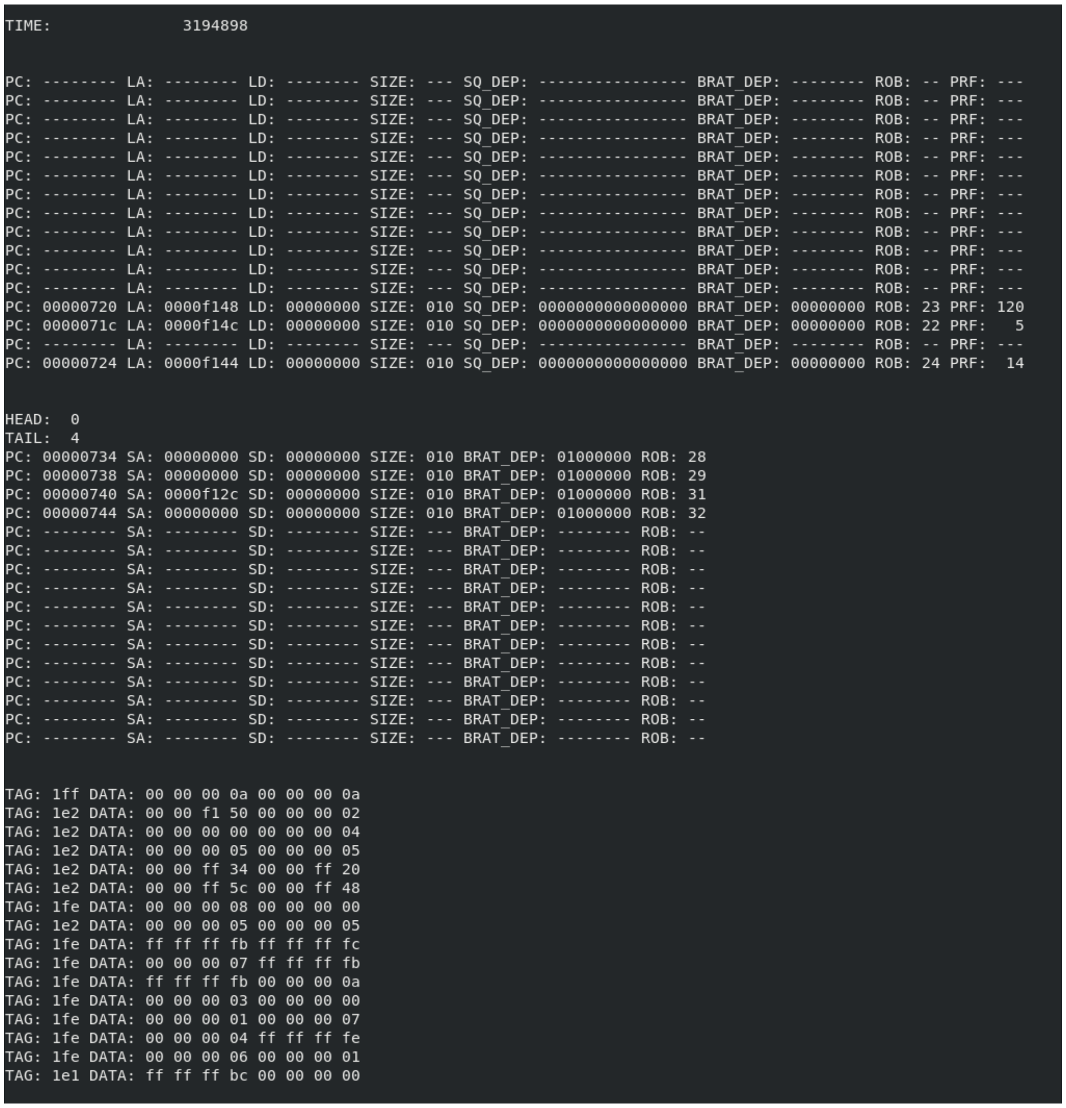
Therefore, our processor makes predictions at the tail of the fetch stage. Even though it unraveled lots of corner cases when combined with early branch resolution, the CPI improvement from this structure compensates for the effort.

3. Testing

For all the modules (including reservation station, register alias table, and physical register file) we developed for Milestone 1 of this project, we also developed randomized unit level testbenches which were self checking in nature. We also incorporated SystemVerilog assertions for each of these modules separately to catch unintended RTL behavior including any error injection scenarios. These were particularly helpful even during later stages of this project when we progressed toward pipeline testing.

For the second milestone, in addition to developing individual module testbenches and SV assertions, we also added support for running a few custom test cases. These were designed specifically to stress test larger dependency chains, and correctness upon resolution of multiple branches in a row. These test cases helped us visualize our processor's performance during maximized utilization of resources such as ROB, BRATs, and RS entries.

For the third milestone and beyond, in addition to the aforementioned testing mechanism, our main focus was ensuring correctness and analysing and enhancing the performance of our processor. We conducted regression testing of our processor with the .s and .c test cases provided in the starter code repository, by taking the project 3 writeback.out and program.out as golden reference against which we used to check our own processor outputs. We automated the correctness checking process by creating shell scripts for both pre and post synthesis regression testing. For debugging the pipeline, we also created and extensively used a simulation logger/debugger as shown in Figure 2 inside our individual and core level testbenches to display useful information such as the contents of the split LSQ, the I-Cache/D-Cache, the ROB, the MSHR utilization, and the fetch stage at each timestamp.



```
TIME:              3194898


PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: 00000720 LA: 0000f148 LD: 00000000 SIZE: 010 SQ_DEP: 0000000000000000 BRAT_DEP: 00000000 ROB: 23 PRF: 120
PC: 0000071c LA: 0000f14c LD: 00000000 SIZE: 010 SQ_DEP: 0000000000000000 BRAT_DEP: 00000000 ROB: 22 PRF:   5
PC: -------- LA: -------- LD: -------- SIZE: --- SQ_DEP: ---------------- BRAT_DEP: -------- ROB: -- PRF: ---
PC: 00000724 LA: 0000f144 LD: 00000000 SIZE: 010 SQ_DEP: 0000000000000000 BRAT_DEP: 00000000 ROB: 24 PRF:  14


HEAD:  0
TAIL:  4
PC: 00000734 SA: 00000000 SD: 00000000 SIZE: 010 BRAT_DEP: 01000000 ROB: 28
PC: 00000738 SA: 00000000 SD: 00000000 SIZE: 010 BRAT_DEP: 01000000 ROB: 29
PC: 00000740 SA: 0000f12c SD: 00000000 SIZE: 010 BRAT_DEP: 01000000 ROB: 31
PC: 00000744 SA: 00000000 SD: 00000000 SIZE: 010 BRAT_DEP: 01000000 ROB: 32
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --
PC: -------- SA: -------- SD: -------- SIZE: --- BRAT_DEP: -------- ROB: --


TAG: 1ff DATA: 00 00 00 0a 00 00 00 0a
TAG: 1e2 DATA: 00 00 f1 50 00 00 00 02
TAG: 1e2 DATA: 00 00 00 00 00 00 00 04
TAG: 1e2 DATA: 00 00 00 05 00 00 00 05
TAG: 1e2 DATA: 00 00 ff 34 00 00 ff 20
TAG: 1e2 DATA: 00 00 ff 5c 00 00 ff 48
TAG: 1fe DATA: 00 00 00 08 00 00 00 00
TAG: 1e2 DATA: 00 00 00 05 00 00 00 05
TAG: 1fe DATA: ff ff ff fb ff ff ff fc
TAG: 1fe DATA: 00 00 00 07 ff ff ff fb
TAG: 1fe DATA: ff ff ff fb 00 00 00 0a
TAG: 1fe DATA: 00 00 00 03 00 00 00 00
TAG: 1fe DATA: 00 00 00 01 00 00 00 07
TAG: 1fe DATA: 00 00 00 04 ff ff ff fe
TAG: 1fe DATA: 00 00 00 06 00 00 00 01
TAG: 1e1 DATA: ff ff ff bc 00 00 00 00
```

Figure 2: The simulation logger/debugger dumping the load buffer, store queue, and D-cache

As part of our performance analysis, we ran the .s and .c test cases provided to us and analyzed their CPI numbers. If our CPI was large, we root-caused it to the test case having large dependency chains, several branches, dependent multiply instructions, high memory latency for load/store operations, or some combination of these four. We also analyzed these numbers by varying several processor parameters, such as BTB cache parameters, N for superscalar execution, fetch queue size, etc. We also varied the memory latency and compiler optimization levels to further stress test our processor under random scenarios.

4. Analysis

4.1 N-Way Superscalar Execution analysis

N-way superscalar execution is a distinguishing feature of our processor. Under ideal circumstances, our processor is capable of fetching, dispatching, executing, and committing N instructions per clock cycle. To gauge the impact of N-way superscalar execution on performance, we ran every .c test case on our processor. For each .c test case, we recorded the number of clock cycles in which our processor committed zero instructions, one instruction, two instructions, and so on, up to N. Using this data, we created the graph shown in Figure 3 for a 4-way superscalar processor which lists the .c test cases along the vertical axis. For each .c test case, a horizontal stacked bar indicates the proportion of clock cycles in which our processor committed zero instructions, one instruction, two instructions, and so on, up to N.

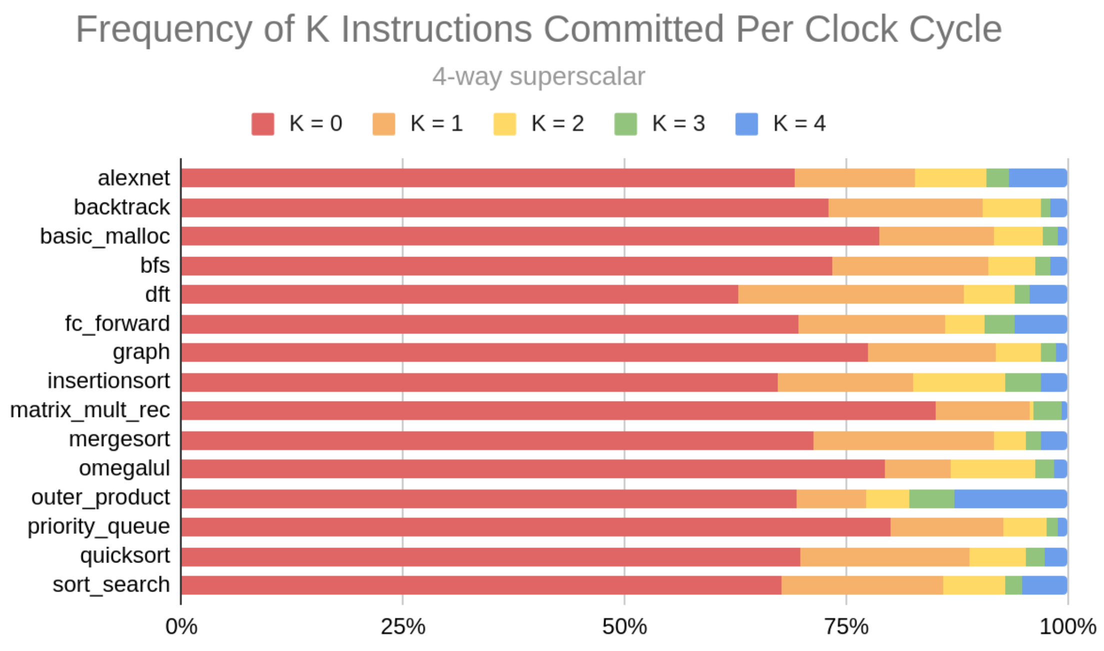
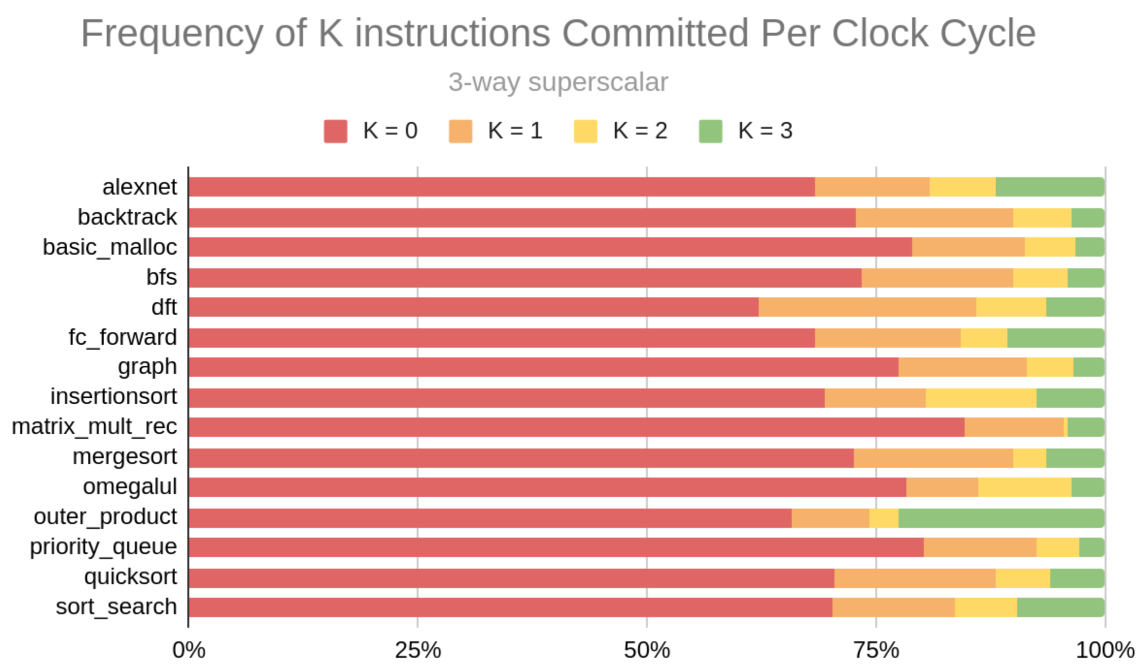


Figure 3: Frequency of K instructions committed per clock cycle for a 4-way superscalar processor

According to the data, our processor spends the majority of clock cycles committing no instructions. In fact, our 4-way superscalar processor rarely commits four instructions per clock cycle. In general, this trend is true for all values of N. We did not find this result particularly surprising. There are a number of reasons why our processor might not commit N instructions per clock cycle. Some of these reasons pertain to the design and limitations of our processor. While our processor can fetch N instructions from the I-cache in a single clock cycle, our processor can only fetch two instructions from memory in a single access. Additionally, the memory latency in clock cycles is large. Thus, our processor is unlikely to commit N instructions per clock cycle while the I-cache is warming up. Our processor also has several potential bottlenecks when it comes to load instructions and store instructions. Our processor dispatches a maximum of one load instruction or one store instruction per clock cycle. The top-level LSQ module sends a maximum of one load instruction or one store instruction to the D-cache controller per clock cycle. The D-cache controller can only retrieve the data for one load instruction or one store instruction from memory in a single access. Additionally, the memory latency in clock cycles is large. The D-cache controller sends data for a maximum of two load instructions to the top-level LSQ module per clock cycle. The top-level LSQ module can only broadcast the result of a single load instruction on a common data bus per clock cycle. Lastly, the ROB commits a maximum of one store instruction per clock cycle. Due to these bottlenecks, our processor is unlikely to commit N instructions per clock cycle in the presence of many load and store instructions. However, we believed that these bottlenecks were acceptable because we anticipated that load and store instructions would not be overwhelmingly common.

Other reasons why our processor might not commit N instructions per clock cycle pertain to the nature of the .c test cases and programs in general. In order for our processor to commit N instructions per clock cycle, there must be a batch of N consecutive instructions at the head of the ROB that have all completed execution. In a homogenous program where all instructions have similar latencies, the number of dependent instructions is small, and the number of mispredicted branch instructions is small, these batches of N consecutive instructions may be relatively common. Our test case nway_demo.s that we submitted is an example of one such homogenous program. If we run a homogenous program on our processor, we expect our processor to commit N instructions per clock cycle fairly frequently. However, real programs (including the .c test cases) are rarely homogenous. Real programs have instructions with varying latencies, many dependent instructions, and many mispredicted branch instructions. Under these circumstances, these batches of N consecutive instructions may be relatively rare. If we run a real program on our processor, we expect our processor to commit N instructions per clock cycle less frequently. Thus, the results in Figure 3 are unsurprising.

We would also expect that as the value of N increases, finding these ideal batches of N instructions would become increasingly difficult because more instructions means more chances for varying latencies, dependent instructions, and mispredicted branch instructions. To test our hypothesis, we collected data for a 3-way superscalar processor and a 5-way superscalar processor and created the graphs in Figure 4 which list the .c test cases along the vertical axes.



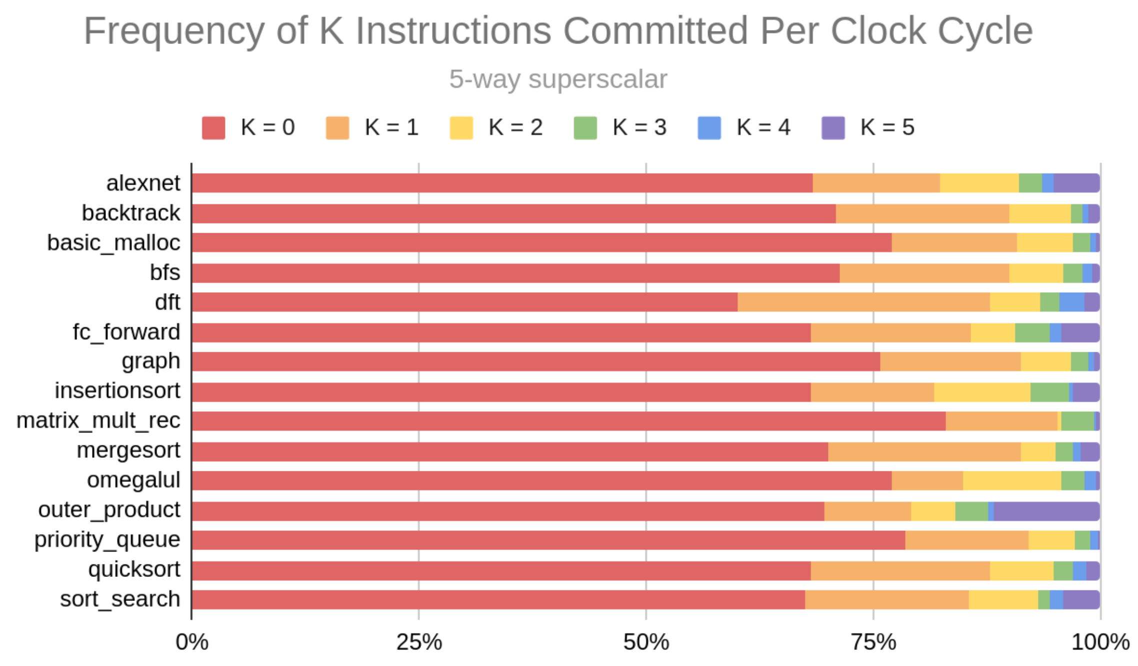


Figure 4: Frequency of K instructions committed per clock cycle for a 3-way superscalar processor (top) and a 5-way superscalar processor (bottom)

For each .c test case, a horizontal stacked bar indicates the proportion of clock cycles in which our processor committed zero instructions, one instruction, two instructions, and so on, up to N. Just as in Figure 3, we see that our N-way superscalar processor rarely commits N instructions per clock cycle. As the value of N increases from three to four to five, the proportion of clock cycles in which our processor commits N instructions decreases for each test case. These results are consistent with our expectations. For values of N between three and five inclusive, we also recorded the CPI for each .c test case and computed the average CPI across all .c test cases. We computed the average CPI across all .c test cases by dividing the total number of clock cycles across all .c test cases by the total number of instructions across all .c test cases. Additionally, we recorded the minimum clock period at that value of N within 0.1 ns granularity. These results are summarized in Table 1 and Table 2.

Table 1: CPI for each .c test case at different values of N for an N-way superscalar processor

Test case	**CPI (N=3)**	**CPI (N=4)**	**CPI (N=5)**	**Test case**	**CPI (N=3)**	**CPI (N=4)**	**CPI (N=5)**
alexnet	1.589038	1.559722	1.426727	*matrix_mult_rec*	4.163637	4.153626	3.629636
backtrack	2.428135	2.384391	2.177614	*mergesort*	2.276969	2.226198	2.035677
basic_malloc	3.032874	2.983033	2.711559	*omegalul*	2.589041	2.589041	2.273973
bfs	2.458226	2.426357	2.179443	*outer_product*	1.211890	1.192815	1.091000
dft	1.715276	1.685697	1.530406	*priority_queue*	3.305365	3.235213	2.955296
fc_forward	1.726316	1.678195	1.529323	*quicksort*	2.101724	2.062588	1.851409
graph	2.888809	2.854292	2.596944	*sort_search*	1.791506	1.701921	1.567909
insertionsort	1.730251	1.660233	1.549136				

Table 2: Average CPI, minimum clock period, and average nanoseconds/instruction across all .c test cases

Parameter	N=3	N=4	N=5
Average CPI	1.585636	1.549660	1.415189
Clock period	9.6	9.7	12.1
ns/instruction	15.222104	15.031707	17.123784

According to the data in Table 2, the average CPI decreases as the value of N increases. Intuitively, this result makes sense. As the value of N increases, our N-way superscalar processor is able to fetch, dispatch, execute, and commit a larger number of instructions per clock cycle. Thus, we would expect that as the value of N increases, the number of instructions per clock cycle increases and the number of clock cycles per instruction decreases. As the value of N increases, we also observe that the clock period increases. Intuitively, this result also makes sense. The amount of work it takes to fetch, dispatch, execute, and commit N instructions per clock cycle scales with the value of N. As the value of N increases, our N-way superscalar processor needs to do more work each clock cycle, resulting in a larger clock period.

Interestingly, transitioning from a 3-way superscalar processor to a 4-way superscalar processor only increases our clock period by 0.1 ns. However, transitioning from a 4-way superscalar processor to a 5-way superscalar processor increases our clock period by 2.4 ns. The reason for this discrepancy is that our critical paths change between N equal to four and N equal to five. For values of N less than or equal to four, the critical paths are in the execution stage. These critical paths are the multiplier stage and the wakeup, select, issue, and execute logic. For values of N greater than or equal to five, the critical path is in the fetch stage. This critical path is fetching N instructions into the fetched instruction queue, which entails N sequential accesses to the branch predictor and BTB to receive a branch prediction and update the pseudo LRU bits. Because the critical path changes, the clock period does not scale linearly with N for values of N between three and five inclusive.

Using the average CPI and the clock period, we computed the average number of nanoseconds per instruction for values of N between three and five inclusive. We consider nanoseconds per instruction to be the best measure of processor performance because nanoseconds per instruction directly correlates with program runtime. According to the data in Table 2, our 4-way superscalar processor achieved the lowest average number of nanoseconds per instruction at 15.031707 nanoseconds per instruction. Because the data indicates that a value of N equal to four yields the best processor performance, we asked to be graded relative to our peers with the value of N set equal to four when we submitted our processor.

4.2 Early Branch Resolution Analysis

The second distinguishing feature of our processor is early branch resolution (EBR). When a mispredicted branch instruction is resolved, every instruction after that mispredicted branch instruction in program order must be squashed. To uniquely count the number of instructions squashed by a mispredicted branch instruction, it is sufficient to count the number of instructions squashed in the fetched instruction queue, decoded instruction queue, dispatch buffer, and ROB. Although instructions in the reservation stations, LSQ, MSHRs, and multipliers are also squashed, these instructions are already accounted for in the ROB. Without early branch resolution, our processor would be forced to resolve every mispredicted branch instruction at the head of the ROB, which would entail squashing every instruction in the ROB. With early branch resolution, our processor can resolve a mispredicted branch instruction before that branch instruction hits the head of the ROB, which entails squashing only some of the instructions in the ROB. Therefore, early branch resolution should cause our processor to squash fewer instructions in the ROB and squash fewer instructions overall. To test our hypothesis, we built a switch into our processor that allows us to enable or disable early branch resolution. We then ran every .c test case on our processor and recorded the number of instructions squashed for each test case with early branch resolution enabled and with early branch resolution disabled. Using this data, we created the graph shown in Figure 5 which lists the .c test cases along the horizontal axis.

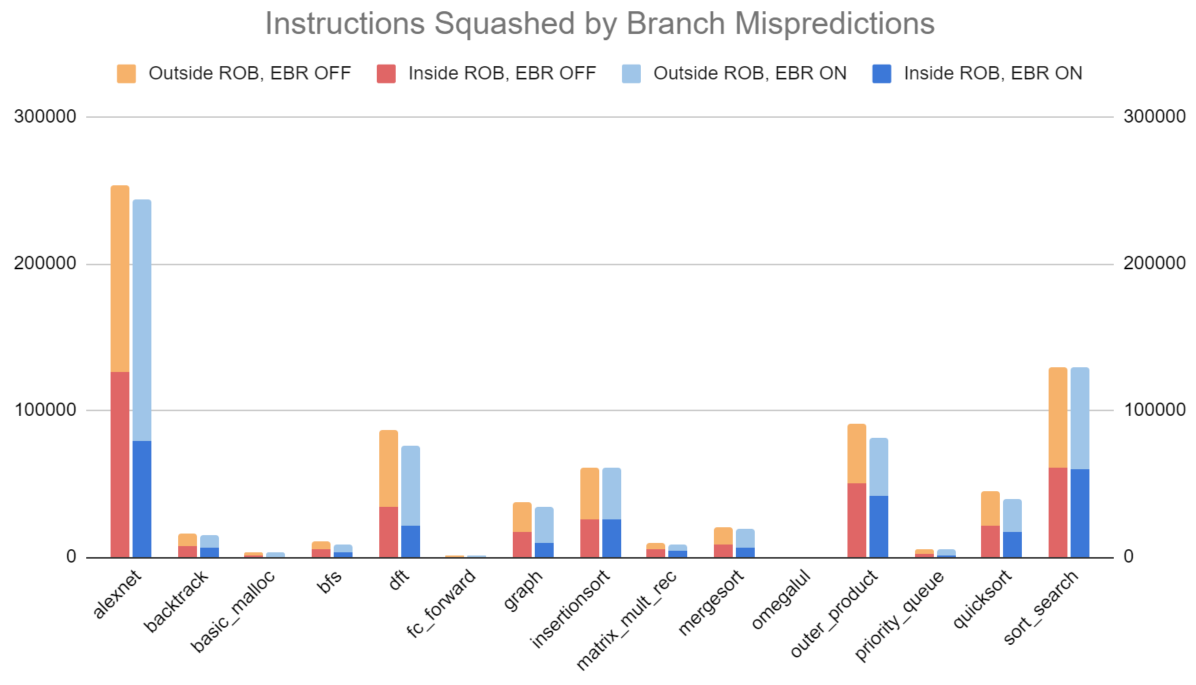


Figure 5: Number of instructions squashed by branch mispredictions

For each .c test case, there are two vertical stacked bars. The vertical stacked bar on the left indicates the number of instructions squashed with early branch resolution disabled. The red portion of this vertical stacked bar indicates the number of instructions squashed inside the ROB while the orange portion of this vertical stacked bar indicates the number of instructions squashed outside the ROB (i.e., in the fetched instruction queue, decoded instruction queue, and dispatch buffer). The vertical stacked bar on the right indicates the number of instructions squashed with early branch resolution enabled. The dark blue portion of this vertical stacked bar indicates the number of instructions squashed inside the ROB while the light blue portion of this vertical stacked bar indicates the number of instructions squashed outside the ROB.

As we expected, enabling early branch resolution reduces the number of instructions squashed inside the ROB and the number of instructions squashed overall. In some test cases like alexnet.c, the number of instructions squashed inside the ROB is reduced significantly. In other test cases like sort_search.c, the number of instructions squashed inside the ROB is barely reduced. In Figure 5, we also observe that the number of instructions squashed outside the ROB appears to increase when early branch resolution is enabled. This increase is most noticeable in alexnet.c but is present in all test cases except for fc_forward.c, matrix_mult_rec.c, outer_product.c, and quicksort.c. We investigated these trends and realized that enabling early branch resolution increases the number of mispredicted branch instructions in most .c test cases. The number of mispredicted branch instructions with early branch resolution enabled and with early branch resolution disabled for each .c test case is recorded in Table 3.

We believe that enabling early branch resolution increases the number of mispredicted branch instructions because early branch resolution encourages our processor to fetch instructions more aggressively. In the event of a mispredicted branch instruction with early branch resolution enabled, our processor begins fetching from the correct memory address before instructions ahead of the mispredicted branch instruction in the ROB have been committed. In the event of a mispredicted branch instruction with early branch resolution disabled, our processor only begins fetching from the correct memory address after instructions ahead of the mispredicted branch instruction in the ROB have been committed. Because we update our branch direction predictor and BTB when a branch instruction commits, our branch predictor has received fewer updates by the time we begin fetching from the correct memory address when early branch resolution is enabled. Therefore, we are not surprised that the number of mispredicted branch instructions increases in this case. This increase in the number of mispredicted branch instructions explains why the number of instructions squashed outside the ROB appears to increase when early branch resolution is enabled. This increase also explains why the impact of early branch resolution on the number of instructions squashed inside the ROB appears to be muted.

Thus far, we've established that early branch resolution reduces the number of instructions squashed in the ROB and the number of instructions squashed overall. Reducing the number of instructions squashed may reduce the power consumption of our processor. However, we are more interested in program runtime rather than power consumption for the purposes of this class. Ideally, early branch resolution should also reduce program runtime by enabling our processor to begin fetching from the correct memory address sooner in the event of a mispredicted branch instruction. To test our hypothesis, we recorded the CPI for each .c test case and computed the average CPI across all .c test cases with early branch resolution enabled and with early branch resolution disabled. These results are summarized in Table 3 and Table 4.

Table 3: CPI for each .c test case with EBR enabled and with EBR disabled

Test case	CPI (EBR OFF)	Mispredicts (EBR OFF)	CPI (EBR ON)	Mispredicts (EBR ON)
alexnet	1.760614	6321	1.559722	8595
backtrack	2.558117	340	2.384391	468
basic_malloc	3.311771	104	2.983033	130
bfs	2.630204	254	2.426357	317
dft	1.836109	2621	1.685697	2825
fc_forward	2.097744	23	1.678195	25
graph	3.124854	1005	2.854292	1337
insertionsort	1.660944	834	1.660233	838
matrix_mult_rec	4.254337	182	4.153626	197
mergesort	2.352016	496	2.226198	614
omegalul	2.890411	4	2.589041	4
outer_product	1.207771	1659	1.192815	1642
priority_queue	3.578404	160	3.235213	207
quicksort	2.168048	925	2.062588	1056
sort_search	1.702643	2127	1.701921	2151

Table 4: Average CPI, minimum clock period, and average nanoseconds/instruction across all .c test cases

Parameter	EBR OFF	EBR ON
Average CPI	1.623872	1.549660
Clock period	9.7	9.7
ns/instruction	15.751560	15.031707

According to the data in Table 4, the average CPI decreases with early branch resolution enabled. Intuitively, this result makes sense for the reasons described above. With early branch resolution enabled, our processor can begin fetching from the correct memory address sooner in the event of a branch misprediction, which should reduce the number of clock cycles for each .c test case. Additionally, enabling or disabling early branch resolution does not change our clock period because neither early branch resolution nor the branch register alias tables appear in our critical path. Using the average CPI and the clock period, we computed the average number of nanoseconds per instruction with early branch resolution enabled and with early branch resolution disabled. Enabling early branch resolution reduced the average number of nanoseconds per instruction from 15.751560 ns to 15.031707 ns, which is a notable performance improvement.

5. What's done and what's not done

Every component in our processor functions as intended. That being said, our processor may have benefitted from an advanced tournament branch predictor that we did implement. It included a local history branch predictor and a global history predictor, equipped to speculatively update the global history register while making the predictions and also unrolling the updates when a branch resolved and was found to be inconsistent with the prediction made initially. We ended up dropping that module given our processor's fetch stage is already nearly our critical path and with the tournament predictor we would have had to analyze and break that critical path to achieve efficient synthesis timing. We chose correctness over complexity for the branch predictor and implemented a bimodal branch predictor instead of that. The tournament branch predictor is not in our final_submit branch but can be found in the predictor_wip branch.