EECS 470 Computer Architecture
MDX3080 Out-of-Order Processor Design Report

Eric Ding, Shuangyu Lei, Weixin Yu, Runze Xue, and Yue Huang

January 9, 2025

Abstract

The report describes the design and implementation of an out-of-order N-way superscalar processor, employing an R10k style register renaming scheme. Our processor supports the basic features of 32-bit RISC-V ISA. The advanced features include a load/store queue with data forwarding, tournament branch predictor, prefetcher, and non-blocking set-associative cache. This is the final project of EECS 470 Computer Architecture at the University of Michigan.

1 Introduction

The report describes the design and implementation of our final project in EECS 470 Computer Architecture, and also all group members' Capstone Design project. The goal of the project is to create an out-of-order processor in System Verilog with reasonable performance. We implement one of the most classic register renaming scheme, R10K, as the main part of our out-of-order architecture. Aiming for high performance, we implemented various advanced features that are intellectually challenging. Besides correctness, our goal is to minimize execution time per instruction, which will be calculated by $CPI \times ClockPeriod$.

The main structure of our components are designed to achieve a lower CPI, while optimizations for clock period are made by lower-level design decisions. The latter will be elaborated later in each individual component in this report, while the features targeting the former are outlined below. Overall, we achieve a average CPI of 2.342 over the entire set of test cases.

1.1 Exploiting Instruction Level Parallelism

Related advanced features include:

- Superscalar Machine
- Tournament Branch Predictor
- Issue Memory Accesses Out-Of-Order (Load-Store Queue)

1.2 Exploiting Locality of Memory Accesses

The other big part of our effort points towards reducing memory latency by exploiting locality of memory accesses. Related advanced features include

- Instruction Prefetching
- Associative Cache
- Non-blocking ICache and DCache
- Data Forwarding from Stores to Loads

In this report, we will describe our design, implementation, and performance analysis for our out-of-order processor. The remainder of the report is organized as follows. We begin by providing an overview of our high-level design. Section 3 describes the design decisions and implementation details of the basic out-of-order features of our processor. The advanced features we introduced to our design are described in Section 4. In section 5, we describe our testing methodologies and analyze the performance implications of some of the advanced features. Lastly, we discussed some ideas and features that did not work out well in section 6.

2 Design Overview

The design diagram of our out-of-order system is shown in Figure 1. The top part of the diagram includes prefetcher, ICache, and DCache, which serves as the processor's interface to memory. In the middle of the diagram, the four yellow rectangles represent our state registers separating each pipelined stage. In addition, reservation station, re-order buffer, multiple functional units, physical register file, register alias table and retirement register alias table are instantiated to record information needed for R10K-based out-of-order instruction execution. Lastly, a tournament branch predictor is instantiated inside the fetch state to further exploit instruction level parallelism.

Table 1: Module Design (†data are parameterized and adjustable)

Module	Parameter	Module	Parameter
Super Scalar width (N)	3† ways	Re-Order Buffer	24† entries
Branch Target Buffer	32† entries	Branch History Buffer	8† history, 16† entries
Reservation Station	16† entries	Physical Register	56†
ICache	32† entries, 4† sets	IMSHR	3† entries
ALU Unit	4† Unit	MULT	2† Unit
Load Buffer	3† entries	Store Queue	6† entries
DCache	64† entries $\times$ word, 16† sets	DMHSR	4† entries

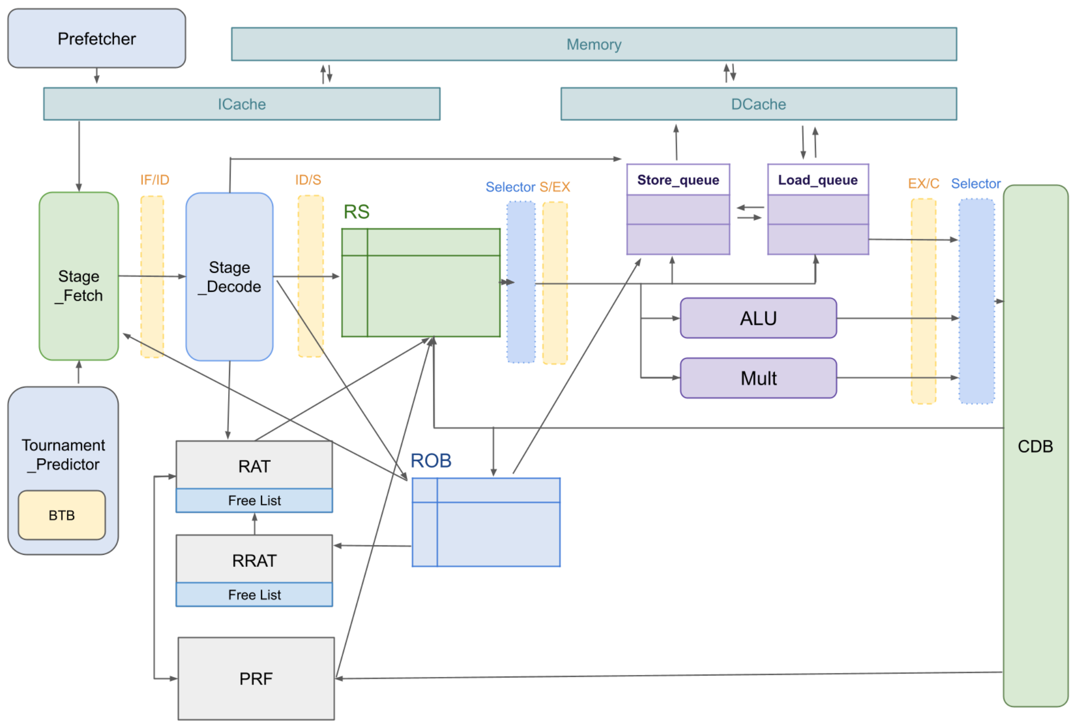


Figure 1: The Design Overview

3 Basic Out-of-order Features

3.1 Fetch Stage

Our fetching stage module includes the ICache and branch predictor sub-modules. The fetching stage module has a PC counter that records the PC of the first instruction in the next stage. Normally, the fetching stage module first queries the branch predictor to get the addresses of N instructions and then passes the addresses to a set-associative non-blocking ICache. If the ICache hits, it will forward the instruction and a hit signal immediately. The PC counter for the next stage is also updated based on the branch predictor result if all instructions are hit. If the ICache misses, the ICache will return a cache miss signal, and the fetching stage module to set the PC counter to be the first missed PC.

The fetching stage module receives the status of the instructions committed in this cycle from the ROB module. It will update the BTB data correspondingly. If squashed, it will not fetch any instructions in this cycle, and the PC counter for the next cycle will be updated based on the resolved branch target received from memory.

If the fetching stage stalls due to a structural hazard, the PC counter for the next cycle will be the same as this cycle.

3.2 Decode Stage

Our decoding stage module is mainly based on the decoder provided in Project 3. It's responsible for dispatching data to ROB, RS, and RAT modules in the Out-of-Order system. It's purely combinational logic.

3.3 Reservation Station

We implemented one Reservation Station (RS) for all kinds of operations. Our RS module does not support internal forwarding, which means that one entry can't be dispatched and selected in the same cycle, and one entry can't be broadcast by CDB and selected in the same stage. However, one entry could be dispatched and broadcast by CDB in the same cycle.

We use priority_selector to select free entries to dispatch, and ready entries to be selected by FU. The `ALU/ MULT/ STORE` instructions will not wake up until the registers they depend on are ready. The `LOAD` instruction will not wake up until the previous STORE instructions are committed.

The RS module outputs `almost_full` signal when there are less than N available slots. The RS module does not accept dispatched instructions when `almost_full`.

3.4 Re-order Buffer

Our Re-order Buffer (ROB) module design is mainly based on a FIFO queue. It receives dispatched instruction from the decoding stage module, which will be pushed to the queue. ROB outputs the instructions committed and squash signal. The commit and squash information will be forwarded to the RRAT and fetching stage modules.

Besides, the ROB sends the number of store instructions at the head of the ROB to the store queue module. The store queue module responds to the ROB number of instructions with addresses calculated, which implies that they are ready to commit. The ROB updates the `executed` bit and then decides which instructions to commit.

The ROB module outputs `almost_full` signal to inform the previous stages of structural hazard. It does not accept any instruction dispatched if it has less than N available slots.

3.5 Functional Units

Our functional unit (FU) consists of four sub-units: arithmetic logic unit (ALU), multiplication unit (MU), load buffer (LB), and store queue (SQ). The LB and SQ interact with each other to form a load-store queue (LSQ). See Section 4 for details.

The ALU is loosely based on the one adopted in the five-stage micro-architecture from Project 3. To lower the clock period, we separated the multiplication functionality into a 4-stage, fully pipelined MU. Furthermore, while the target address for load/store instructions is calculated in ALU in Project 3, we migrate it into LB and SQ respectively to avoid over-complicated interactions between the ALU and the LSQ. The conditional branch is still handled by ALU, and the package sent from the FU to the ROB signals whether a conditional branch is calculated to be taken.

The FU does not dispatch instructions to these four units. Instead, the selection of sub-units is done in the instruction decoding stage and is reflected in four input packages of FU sent to each sub-unit. The FU still collects the output from these sub-units to signal RS if there will be a stall caused by the structural hazards, and to give ROB a summarized output including whether an instruction inside ROB is executed, a branch is taken, and where a branch instruction branches into.

3.6 Common Data Bus

Our common data bus (CDB) collects the result of the FU from the ROB and selects N of them to broadcast. In pursuance of hardware efficiency, we used the priority selector provided by the staff team and implemented a tailored multiplexer to utilize the one-hot signal from the selector. The one-hot multiplexer intensively used wire-ORs to reduce the length of the critical line. Our CDB maintains a state, keeping the selector logic and the FU calculations into separate cycles. This is to avoid putting the high-fanout CDB selector and data transmission logic on the critical path.

3.7 Physical Register File

Our physical register file (PRF) consists of registers of a total number matching the sum of architectural registers (32) and the number of ROB entries. Each entry of the PRF holds a 32-bit word as the content, and a `valid` bit to indicate whether the register is in use. The PRF can handle $2N$ read requests and N write requests simultaneously, in accordance with the number of sources and destination registers of RISC-V instructions.

Upon initialization, the first 32 entries will be initialized as valid, holding a value of zero. This state matches the initial state of the RAT and RRAT, and the specification that all architectural registers shall be initialized to zero.

The PRF automatically marks the corresponding entry as valid when receiving a write request, and takes a separate input to invalidate certain entries. Especially, the entry with index zero is always mapped to the `x0` register and hence is always valid and holds a constant value of zero, regardless of the input. Consequently, write requests with a zero index are treated as invalid, and will be neglected. This design allows us to avoid using an extra bit to indicate whether a write request is unused.

The PRF forwards the content of write requests to read requests to enable the data of CDB to be broadcasted immediately to the RS. We therefore do not need additional data buses to forward the data from CDB to RS.

3.8 Register Alias Table and Retirement Register Alias Table

Our Register Alias Table (RAT) and Retirement Register Alias Table (RRAT) are implemented to perform register renaming in R10k style. Both modules have a free_list module for PRF.

The RRAT module receives the committed instructions information from the ROB module and updates the table. If squashed, the RRAT module will forward the squash signal, the RRAT table, and the RRAT free_list data to the RAT module.

The RAT module receives the dispatched instructions information from the decoding stage module and updates the table. It outputs the Physical Register Number for ROB and RS. If it receives a squash signal from the RRAT module, it will copy the alias table and the free list from RRAT.

4 Advanced High-performance Features

4.1 Superscalar

We implemented a parametrized N-way superscalar machine. Related designs include data forwarding for multiple modules, including reservation station and physical register files. The final version uses a 3 way superscalar to balance between CPI and clock period.

4.2 More Sophisticated Branch Predictors

We implemented a tournament-selector style branch predictor with a 2-bit counter. It selects between a gshare predictor and a local history predictor with a 1-bit counter. For the gshare predictor, we take 5 bits of the program counter, xor the higher 4 bits of global history as the index of the pattern history table. The local history predictor takes 8 bits of local history into account.

4.3 Instruction Prefetching

We implement a simple prefetching logic using an FSM. The prefetcher is placed inside the fetching stage and interfaces with ICache alongside normal instruction load requests. There are three states inside prefetcher: `reset`, `on`, `off`. The prefetcher fetches one cache line at most per cycle, in case the prefetched instructions block the limited memory bandwidth too much. If the number of consecutive prefetched cache lines count exceeds a threshold, or there is a cache hit for the prefetching request, the state is transitioned to `off`. The state will only transition back to `on` when the last prefetched instruction address is lower than the current instruction address. We make sure the prefetched address is always higher than the highest instruction address in the fetching stage in the current cycle.

4.4 Associative Cache

We implemented a write-allocate, write-back set-associative DCache, and a set-associative ICache. The ICache has a block size of 8B to exploit more spatial locality of instruction fetching, while the DCache has a block size of 4B to exploit more temporal locality of loads and stores. The number of sets and ways are parameterized and can be adjusted conveniently. The DCache and ICache are multi-ported (allowing N access per cycle).

Every cache line stores `state` (valid or invalid), `tag`, `dirty`, as well as the cache data. We use pseudo-LRU policy to perform dirty cache line eviction. A dedicated module implements pseudo-LRU logic, and is instantiated for each cache set. A newly allocated cache line (from memory) will be placed at the line indicated by the pseudo-LRU pointer. If the replaced cache line is dirty, it will be evicted to memory with the highest priority over `mem_load` of data and instructions. For every cache access, the pseudo-LRU pointer will be updated. If there are multiple cache accesses per cycle in the same set, only the last access will be used to update the LRU state.

As our processor is N-ways superscalar, we want the cache interface to be N-ways as well, and the cache will return access valid bit to each cache access to indicate cache hit or miss. For ICache, each request from the fetch stage (load) needs to constantly drive the interface until a cache access valid bit is returned. If there is a missed instruction load, subsequent instructions will not be issued to maintain the program order in the issue stage.

For DCache, each request (load and store from load store queue) doesn't need to constantly drive the port, unless there is a structural hazard. If the cache hits, the data will be returned immediately for load. If the cache misses, DCache will record the associated load queue index, which will be returned to the load queue when the data is ready.

4.5 Non-blocking ICache and DCache

As the memory access latency is roughly 100ns, we use a non-blocking Cache and implement MSHR to pipeline the memory accessing process and increase the overall throughput. In terms of ICache, we require loads from the fetch stage to constantly drive the port, so that we do not need to maintain the metadata for each load. For each cache miss, a new MSHR entry will be allocated if the address of the requesting load does not match the ones of the existing entries. Then, a priority selector selects a valid and ready entry, and forwards the request to the memory interface, if DCache isn't accessing the memory in this cycle. The MSHR entry records the transaction tag returned from memory and wakes up when the memory provides the matching data tag and data.

Our implementation of MSHR for DCache is largely similar to Icache, except that we maintain the state for each outstanding request to distinguish loads from stores and queue up the requests to the same address. This is achieved through creating a queue for each active MSHR entry, as shown in Figure 2. Each missed load and store is pushed to a queue associated with an MSHR entry with a matching address. If there are structural hazards in the target queue or MSHR, the request will not be accepted in the issuing cycle, and the issuer (load queue or store queue) needs to make another request in the subsequent cycles. The queue is implemented using a circular buffer. When the data is ready, the entire queue is flushed to DCache, and the loads and stores in the queue are executed in order. For loads, the data and its associated load queue index will be returned to the load queue. When there is a squash event, ICache (including MSHR) will be cleared, whereas Dcache and its MSHR won't. However, the loads in the MSHR queue for Dcache will be cleared due to their speculative nature.

4.6 Issue Memory Accesses Out-Of-Order Using a Load-Store Queue

To parallelize non-dependent load and store instructions, we implemented a store queue with a load buffer. Store instructions are allocated in order as they enter the out-of-order system to ensure correctness. The load buffer is unordered, and load instructions do not allocate their space in the load unit until they are issued out of the reservation station. Our final version has a store queue size of $2 \times N$ and a load unit size of 3.

To avoid excessive address comparators on store-load data forwarding, we guarantee that all stores preceding the loads will have their address calculated before the entry of load instructions into the load buffer, and the reservation station is tailored accordingly to activate the load instructions at our desired time point.

For both the load buffer and store queue, we separated the address calculation from the ALU to avoid over-complicating the logic, as aforementioned.

The overall design diagram for our load-store queue is shown in Figure 3.

4.6.1 Load Buffer

The load buffer is pipelined with three stages: address calculation, data forwarding from the store queue, and querying the DCache for the data to be loaded. Such pipelining avoids the load buffer to extend the critical path. The mechanism of data forwarding will be detailed with our store queue in the next subsection. We utilize a priority selector to select among the load instructions to send to DCache.

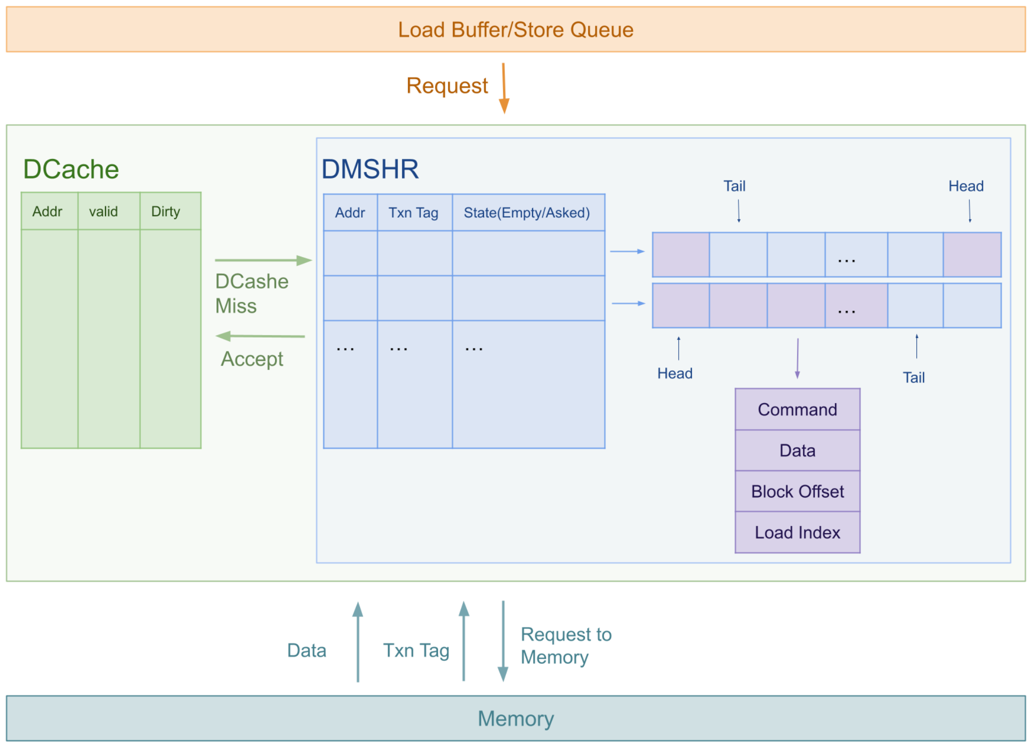


Figure 2: The layout of DCache

4.6.2 Store Queue

Similarly, the store queue is separated into two stages: calculating the address, and waiting for execution. They are also pipelined. While waiting for execution, there are two status bits indicating `committed` and `accepted`. `Committed` is set after the store instruction has reached the head of the ROB and the address has been calculated. Once `committed` is asserted for a given store instruction, the corresponding store entry acts as an extension of the MSHR entries of the non-blocking DCache. In other words, once the program halts on a `wfi` instruction, all `committed` store instructions within the store queue should be written back to memory. We made this design decision to separate the store queue's DCache access with the ROB commitment logic into different cycles. The DCache access is the critical path both before and after the separation.

The state `accepted` is to remedy the situation where DCache takes non-consecutive store requests. In that case, already `accepted` store instructions are not sent to DCache again, avoiding unnecessary structural hazards in the MSHR entries.

4.7 Data forwarding from Stores to Loads

By our LSQ design, load instructions only need to check data dependency once on entering the load buffer. The data forwarding part is structured as purely combinational logic in both units. Load unit asks the store queue with the address, memory granularity, and tail of store queue indexes it may depend on (shown as "Dependent Tail" in Figure 3), and receives the forwarded data (if available). Any data at the same word-level address will be forwarded, with an extra 4-bit vector indicating which bits are valid. The load unit will ask DCache afterward if and only if not all of the requested bytes are forwarded.

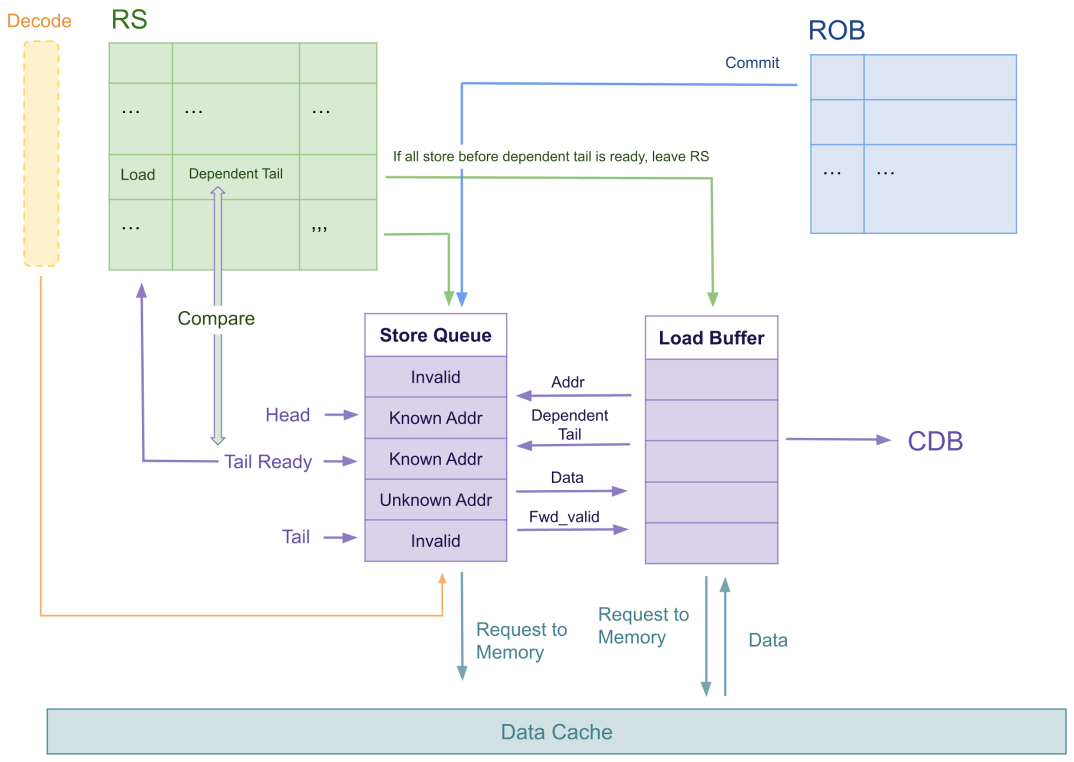


Figure 3: The layout of Load Store Queue

5 Evaluation

5.1 Testing

5.1.1 Module Testing

To shrink the search space once bugs occur, we approach testing incrementally. After we implement individual modules in sub-groups, we write Verilog testbenches that target the basic functionalities of each of them. Simulation and synthesis are run to ensure both correctness and reasonable performance in terms of clock period.

5.1.2 Integration Testing

After that, we simulate and synthesize the entire processor without support for memory operations and any caches. We ensure its correctness with relatively sophisticated test benches like `mult_no_lsq.s`. We finally integrate all parts of the processor and run all the testbenches mentioned in Appendix A. For integration testing, we simply print related register states and control signals from our processor, with proper labeling of the number of cycles and number of instructions committed. One testing strategy we found particularly useful is to terminate the program once the problematic instruction is executed(in `cpu_test.sv`). This avoids unnecessarily big debugging output files.

5.1.3 Testing Scripts

To automate the testing and evaluation on multiple test programs, we wrote shell scripts to automatically compare with correct writeback and memory output files, and calculate CPI and predictor/cache hit rates.

5.1.4 Correctness Results

Our completed version of the processor produces correct results under all of the test cases mentioned in Appendix A. All writeback output files and memory output lines starting with "@@@" match exactly with the provided Project 3 in-order pipeline output.

5.2 CPI and Clock Period

The final version of our processor achieves an average CPI of 2.342, with a $20ns$ clock period. The CPI for the testcase where every instruction is not dependent on each other within a loop is 0.503, and the CPI for the testcase where there are dependencies between subsequent instructions is 2.250.

5.3 Number of Ways

As our superscalar width N is parametrized, we run experiments with various widths to test the impact of superscalar on the performance of our processor. We will analyze its effect on the CPI and clock period below.

5.3.1 Effect on CPI

As shown in Figure 5, as the number of superscalar ways increases from 1 to 3, the average CPI drops considerably. This meets our expectations. However, as superscalar width increases more than 3, the average CPI stays relatively constant. This might be because we have exploited all of the instruction level parallelism within a reasonable window, or because higher N results in more structural hazards in other components.

Since the CPI drops more dramatically from $N = 2$ to $N = 3$ than from $N = 1$ to $N = 2$, we chose $N = 3$ in our final version.

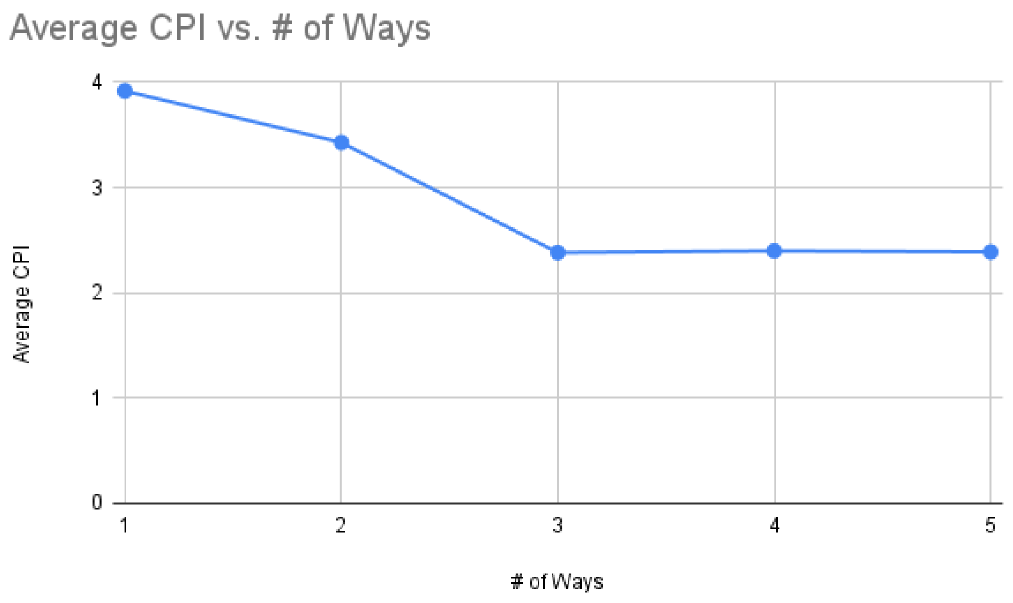


Figure 4: CPI of different superscalar width

5.3.2 Effect on Clock Period

As shown in Figure 5, the clock period grows a little faster than linear vs. superscalar width. This meets our expectations.

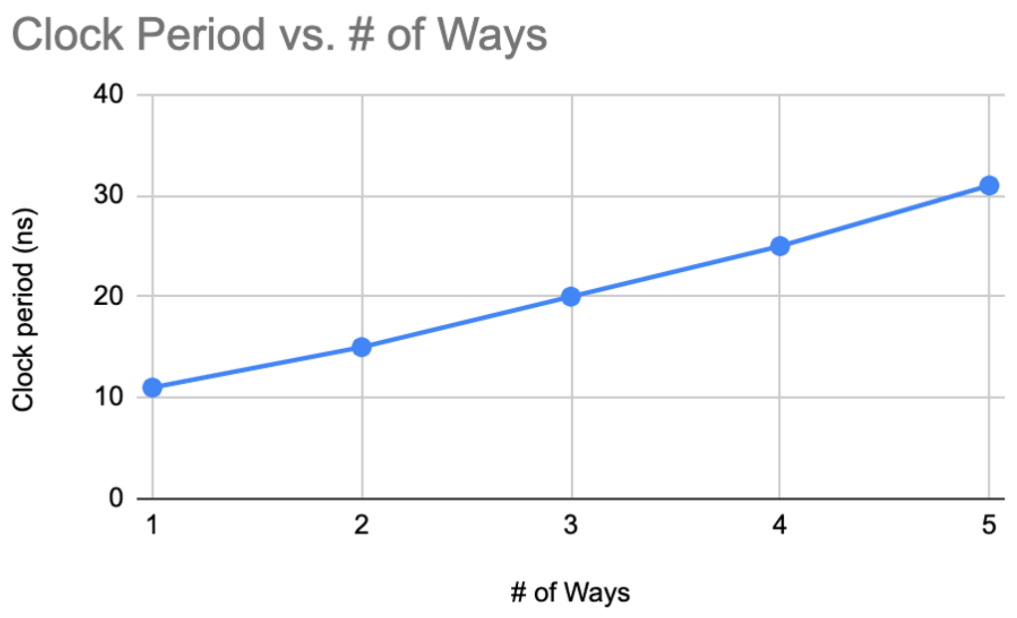


Figure 5: Clock Period of different superscalar width

5.4 Cache Associativity

The impact of cache associativity on CPI is not significant. Actually, the average CPI even goes a little higher when we decrease number of sets from 32 to 4. We do see a decrease in average CPI when there are only two sets, where the set is nearly fully associative. However, the variation in CPI is less than 0.05. This is probably because of rare occurrence of conflict misses in our test programs.

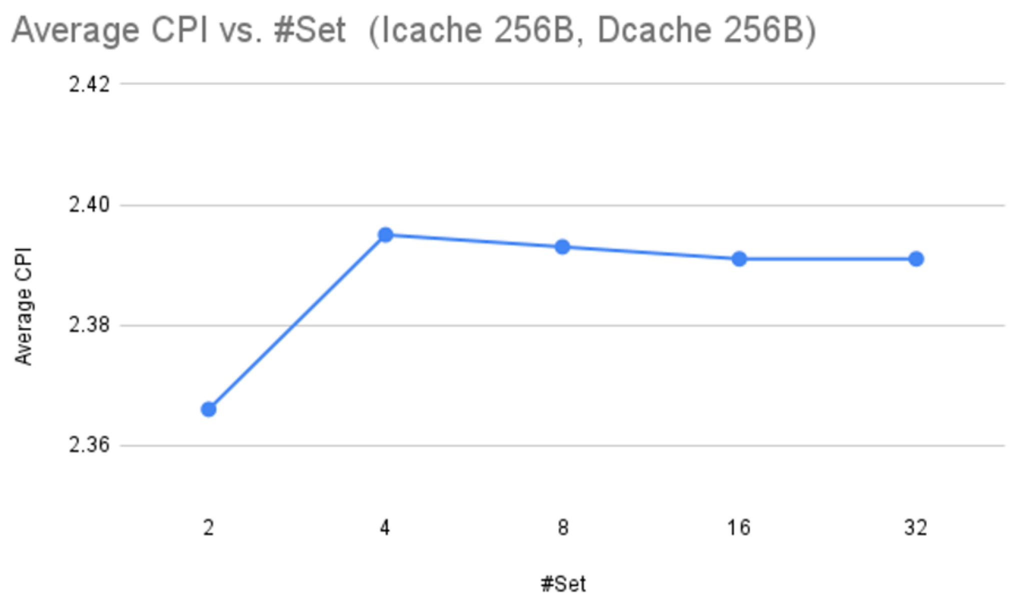


Figure 6: CPI of different number of sets

5.5 Prefetching

The maximum number of consecutive prefetched cache lines can be tuned. We adjusted the number and ploted the average CPI over our workloads, shown in Figure 7. The effect of prefetching isn't very significant, which we will discuss in section 6. We found that prefetching 8 cache lines consecutively results in the best performance.

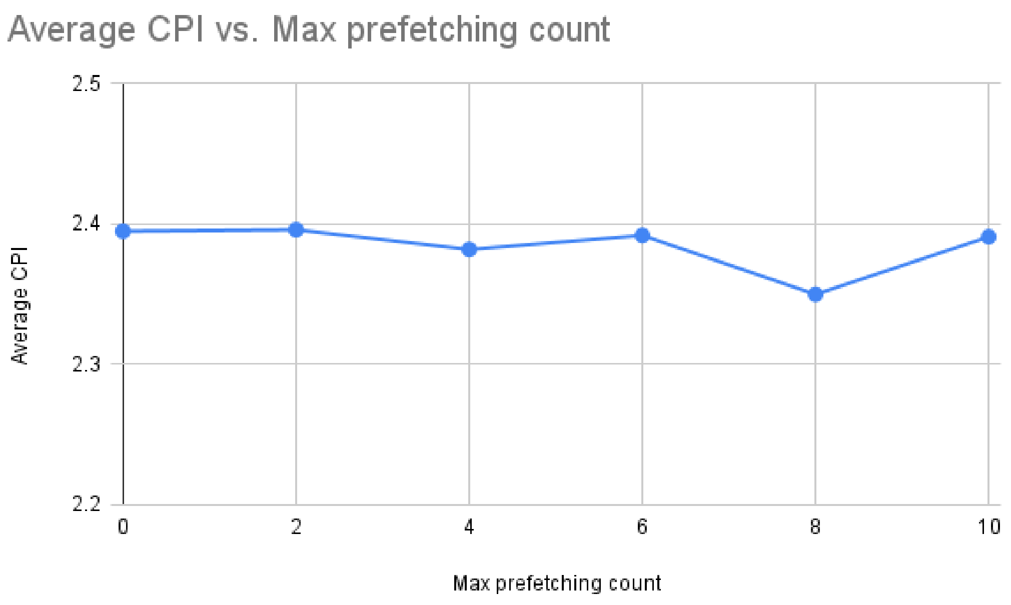
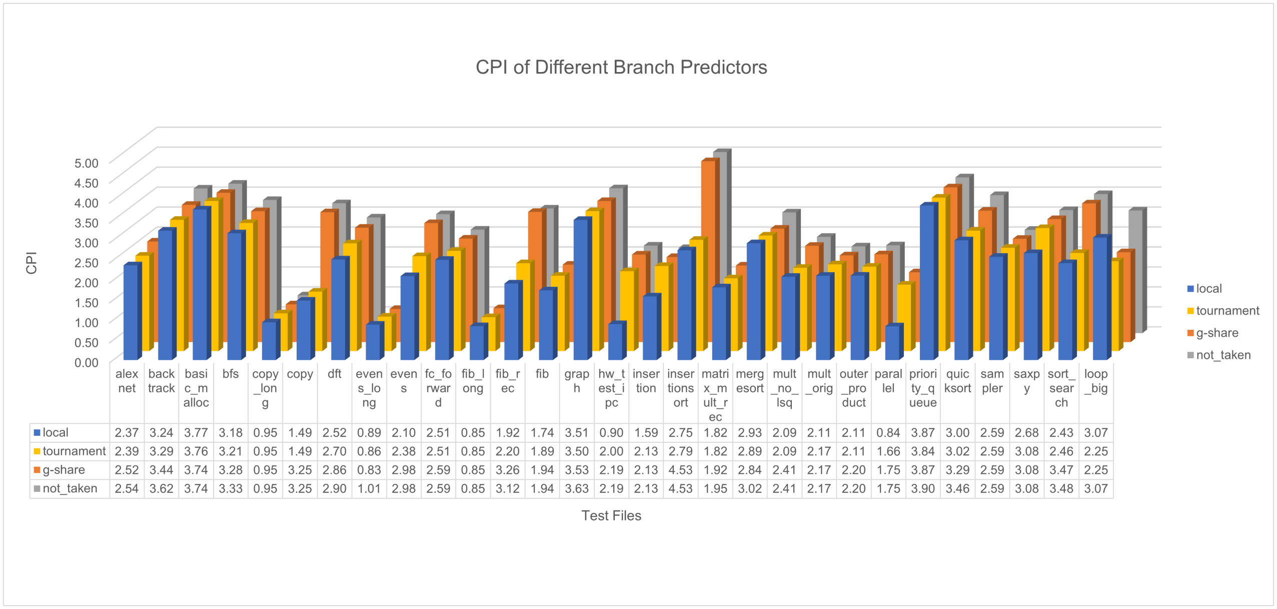


Figure 7: CPI of different prefetch thresholds

5.6 Branch Predictors

The final iteration of our processor employs a tournament predictor, which selects between a local history predictor and a GShare predictor. For analysis purposes, we take into account four distinct branch predictors: the always-not-taken predictor, the local predictor, the GShare predictor, and the tournament predictor itself. In an effort to optimize the hit rate and CPI, we have adjusted the XOR bit of the GShare predictor. The subsequent diagram illustrates their respective CPI and prediction hit rates. The local predictor is the best for some of the test cases, and the tournament predictor is best for some. We decide to employ a tournament predictor for a wider range of branch patterns. The CPI performance for the four branch predictors across all of our testbenches is shown in Figure 8.



	alexnet	backtrack	basic_malloc	bfs	copy_long	copy	dft	evens_long	evens	fc_forward	fib_long	fib_rec	fib	graph	hw_test_ipc
🟦 local	2.37	3.24	3.77	3.18	0.95	1.49	2.52	0.89	2.10	2.51	0.85	1.92	1.74	3.51	0.90
🟨 tournament	2.39	3.29	3.76	3.21	0.95	1.49	2.70	0.86	2.38	2.51	0.85	2.20	1.89	3.50	2.00
🟧 g-share	2.52	3.44	3.74	3.28	0.95	3.25	2.86	0.83	2.98	2.59	0.85	3.26	1.94	3.53	2.19
⬜ not_taken	2.54	3.62	3.74	3.33	0.95	3.25	2.90	1.01	2.98	2.59	0.85	3.12	1.94	3.63	2.19

	insertion	insertionsort	matrix_mult_rec	mergesort	mult_no_lsq	mult_orig	outer_product	parallel	priority_queue	quicksort	sampler	saxpy	sort_search	loop_big
🟦 local	1.59	2.75	1.82	2.93	2.09	2.11	2.11	0.84	3.87	3.00	2.59	2.68	2.43	3.07
🟨 tournament	2.13	2.79	1.82	2.89	2.09	2.17	2.11	1.66	3.84	3.02	2.59	3.08	2.46	2.25
🟧 g-share	2.13	4.53	1.92	2.84	2.41	2.17	2.20	1.75	3.87	3.29	2.59	3.08	3.47	2.25
⬜ not_taken	2.13	4.53	1.95	3.02	2.41	2.17	2.20	1.75	3.90	3.46	2.59	3.08	3.48	3.07

Figure 8: CPI of different branch predictors

As clearly depicted in Figure 9, all nontrivial types of branch predictors we have implemented outperformed the not_taken predictor. This meets our expectations.

The local history predictor works generally better than both the tournament predictor and the GShare predictor. It is reasonable that it works better than the GShare predictor because it maintains many more bits in its state. Moreover, the programs might have fewer branches dependent on each other. However, it is out of our expectation that the tournament selector between them works generally worse than the local history predictor. We suspect that this is because the GShare predictor works so much worse than the local history predictor that even a reasonable selector between them lowers the prediction accuracy in some cases.

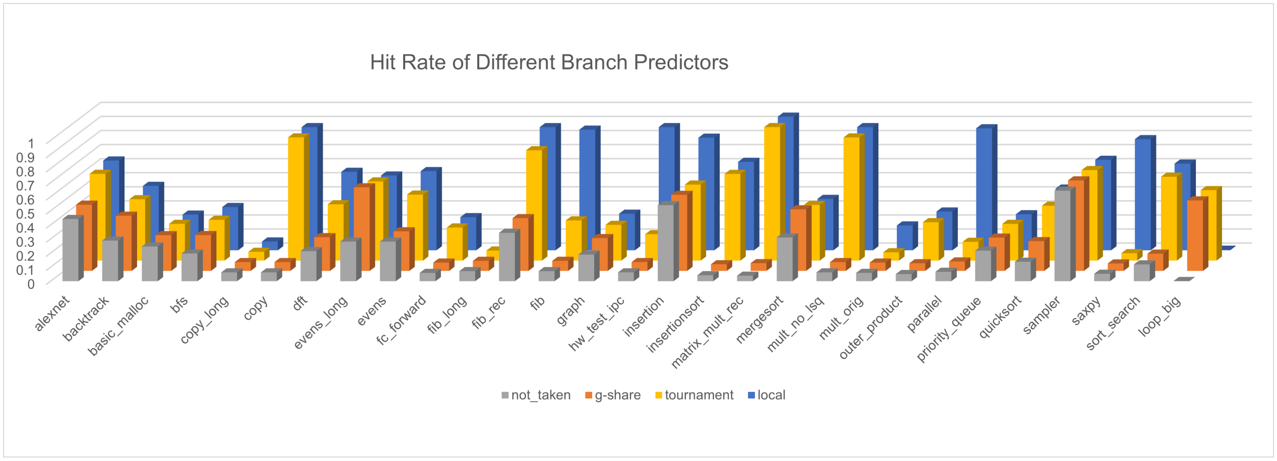


Figure 9: Hit Rate of different branch predictors

The parameters utilized for analyzing different branch predictors are listed in Table 2.

N	ROB Size	ICache Set	DCache Set
3	24	4	16

Table 2: Parameter in Branch Predictor Evaluation

6 What does (not) work

6.1 Branch Predictor

We have implemented four different branch predictors in our code base: the not-taken predictor, the local predictor, the g-share predictor, and the tournament predictor (selecting among local and g-share). Though the local predictor outnumbers the tournament by accuracy in some of the testbenches, we eventually adopt the tournament predictor, as we believe this will fit a wider range of branch patterns.

6.2 Return Address Stack

As a part of sophisticated branch prediction, we have implemented a prototype of the return address stack (RAS). However, in light of the potential technical difficulties in its integration into our superscalar processor, and the limited promotion in efficiency expected, we eventually abandoned it and adopted the branch predictor for conditional branches only. The major difficulty of integrating the RAS is that the instruction must be fetched and decoded in advance to allow the RAS to push the return address into the stack. In contrast, with the records in the branch target buffer (BTB), the branch predictor can identify branch instructions without actually querying the ICache and the memory, and therefore enable us to predict N addresses simultaneously.

6.3 Prefetcher

The prefetcher FSM only creates about 0.04 reduction at most in CPI compared to ICache without a prefetcher, which is contrary to what we originally thought. We think this can be attributed to contention. We implemented a multi-ported ICache, with at most N outstanding instruction load request per cycle. If another prefetching request is made, it may interfere with memory access of normal instruction fetching. We used the priority selector for scheduling, which may not select the oldest load request in MSHR.

6.4 Cache Size Omission

The submitted version of our processor uses a cache size of 32 bytes (for both Icache and Dcache). We later realized that the cache size can be further increased to 256 bytes, and we updated our processor, resulting a 0.8 reduction in average CPI.

7 Team Member Contributions

- Eric Ding (20%): Reservation Station (together), Register Alias Table and Retirement Register Alias Table, Re-order Buffer, Non-blocking caches, Prefetcher, Set-Associativity of ICache and DCache, and Debugging
- Shuangyu Lei (20%): Reservation Station (together), Register Alias Table and Retirement Register Alias Table, Re-order Buffer, Non-blocking caches, and Debugging
- Weixin Yu (20%): Reservation Station (together), Physical Register File, Functional Unit, Common Data Bus, Load-Store Queue, Data Forwarding, and Debugging
- Runze Xue (20%): Reservation Station (together), Physical Register File, Functional Unit, Common Data Bus, Load-Store Queue, Branch Predictors, Return Address Stack (Trial), and Debugging
- Yue Huang (20%): Reservation Station (together), Physical Register File, Functional Unit, Common Data Bus, Load-Store Queue, and Debugging

References

A Test Case List

Our benchmark is based on the following test cases.

- alexnet
- backtrack
- basic_malloc
- bfs
- copy_long
- copy
- dft
- evens_long
- evens
- fc_forward
- fib_long
- fib_rec
- fib
- graph
- haha
- hw_test_ipc
- insertion
- insertionsort
- matrix_mult_rec
- mergesort
- mult_no_lsq
- mult_orig
- no_hazard
- omegalul
- outer_product
- parallel
- priority_queue
- quicksort
- sampler
- saxpy
- sort_search
- loop_big

B Testing Scripts

```
#!/bin/bash
set -e

# Define a function to calculate CPI
calculate_cpi() {
    cycles=$1
    instrs=$2
    cpi=$(bc <<< "scale=6;␣$cycles␣/␣$instrs")
    echo "$cpi"
}

# List of filenames in an array
file_list=(
    "alexnet" \
    "backtrack" \
    "basic_malloc" \
    "bfs" \
    "copy_long" \
    "copy" \
    "dft" \
    "evens_long" \
    "evens" \
    "fc_forward" \
    "fib_long" \
    "fib_rec" \
    "fib" \
    "graph" \
    "haha" \
    "hw_test_ipc" \
    "insertion" \
    "insertionsort" \
    "load_simple" \
    "load_store_simple" \
    "matrix_mult_rec" \
    "mergesort" \
    "mult_no_lsq" \
    "mult_orig" \
    "no_hazard" \
    "omegalul" \
    "outer_product" \
    "parallel" \
    "priority_queue" \
    "quicksort" \
    "sampler" \
    "saxpy" \
    "sort_search" \
    "loop_big" \
)
# Initialize variables to hold total cycles and instructions
total_cycles=0
total_instrs=0

total_cache_access=0
total_cache_hit=0

working_path=$(pwd)
report_path="$working_path/report.txt"
printf "%-20s␣%-10s␣%-10s\n" "filename" "cpi" "hit" >> "$report_path"

# Iterate over the list of filenames
for filename in "${file_list[@]}"; do
    make "$filename.out" -j
    # Extract cycles, instructions, and CPI from the file
    if [ -e "$working_path/output/$filename.cpi" ]; then

        cycles=$(grep -oP '@@@␣␣\K[0-9]+(?=␣cycles)' "$working_path/output/$filename.cpi")
        instrs=$(grep -oP '@@@␣␣[0-9]+␣cycles␣/␣\K[0-9]+(?=␣instrs)' "$working_path/output/$filename.c
        cpi=$(calculate_cpi "$cycles" "$instrs")
        echo "@@@␣␣$cycles␣cycles␣/␣$instrs␣instrs␣=␣$cpi␣CPI"
        total_cycles=$((total_cycles + cycles))
        total_instrs=$((total_instrs + instrs))

        correct=$(grep -oP 'predictor␣hit␣rate:␣\K[0-9]+(?=␣correct)' "$working_path/output/$filename.
        branches=$(grep -oP 'predictor␣hit␣rate:␣[0-9]+␣correct␣/␣\K[0-9]+(?=␣branches)' "$working_pat
        branch_hit=$(calculate_cpi "$correct" "$branches")
        echo "@@@␣␣$correct␣correct␣/␣$branches␣branches␣=␣$hit␣hit"
        total_correct=$((total_correct + correct))
        total_branches=$((total_branches + branches))

        cache_hit=$(grep -oP 'cache␣hit␣rate:␣\K[0-9]+(?=␣hits)' "$working_path/output/$filename.cpi")
        cache_accesses=$(grep -oP 'cache␣hit␣rate:␣[0-9]+␣hits␣/␣\K[0-9]+(?=␣accesses)' "$working_path
        cache_hit=$(calculate_cpi "$correct" "$branches")
        echo "@@@␣␣$correct␣correct␣/␣$branches␣branches␣=␣$hit␣hit"
        total_cache_access=$((total_cache_access + cache_accesses))
        total_cache_hit=$((total_cache_hit + cache_hit))


        printf "%-20s␣%-10s␣%-10s␣%-10s\n" "$filename" "$cpi" "$branch_hit" "$total_cache_hit" >> "$re
    else
```

```
        echo "File␣not␣found:␣output/$filename.cpi"
    fi
done

# Calculate the average CPI
average_cpi=$(calculate_cpi "$total_cycles" "$total_instrs")
average_hit=$(calculate_cpi "$total_correct" "$total_branches")
average_cache_hit=$(calculate_cpi "$total_cache_hit" "$total_cache_access")

# Print the average CPI
echo "Average␣CPI:␣$average_cpi"
echo "Average␣hit:␣$average_hit"

# Print the report path
printf "%-20s␣%-10s␣%-10s␣%-10s\n" "Average" "$average_cpi" "$average_hit" "$average_cache_hit">> "$r
echo "Report␣saved␣to:␣$report_path"
```

C Test Case

This test case acts as part of our benchmark, revealing our processor's ability to predict large for loops and handling back-to-back dependent instructions.

loop_big.s:

```
    li x1, 1
    li x2, 1024
start:
    addi x1, x1, 1
    addi x3, x1, 1
    addi x4, x3, 1
    addi x5, x4, 1
    addi x6, x5, 1
    addi x7, x6, 1
    addi x8, x7, 1
    addi x9, x8, 1
    addi x10, x9, 1
    addi x11, x10, 1
    addi x12, x11, 1
    addi x13, x12, 1
    addi x14, x13, 1
    addi x15, x14, 1
    addi x3, x15, 1
    addi x4, x3, 1
    addi x5, x4, 1
    addi x6, x5, 1
    addi x7, x6, 1
    addi x8, x7, 1
    addi x9, x8, 1
    addi x10, x9, 1
    addi x11, x10, 1
    addi x12, x11, 1
    addi x13, x12, 1
    addi x14, x13, 1
    addi x15, x14, 1
    bne x1, x2, start
    wfi
```

Acknowledgement

The authors would like to thank Professor Mark Brehob, Bradley Schulz, Ian Wrzesinski, and Mustafa Miyaziwala for their help and advice on this project.