N-way R10k-Style Out-of-Order Processor

EECS 470 Final Project
University of Michigan
Electrical Engineering and Computer Science
Team A99

Zhenyan Zhu
`uniqname:lukezhuz`
`lukezhuz@umich.edu`

Parin Senta
`uniqname:psenta`
`psenta@umich.edu`

Lai Wang
`uniqname:laiwang`
`laiwang@umich.edu`

Hithesh Prabhakar Reddy
`uniqname:hitheshp`
`hitheshp@umich.edu`

Noah Kaplan
`uniqname:kaplannp`
`kaplannp@umich.edu`

Vivianna Mahtab
`uniqname:vmahtab`
`vmahtab@umich.edu`

Abstract

This report outlines the project report for our final project for EECS 470. Our design introduces an R10k-Style N-way Out-of-Order Superscalar Processor emphasizing smart design choices for better performance. Key features include Early Tag Broadcast, Conservative Store-to-Load Forwarding, Victim Cache, and a K-way Non-blocking D-Cache. The multi-banked ROB, for high scalability and a practical FIFO Queue for Freelist Management, enhances resource use. Each functional unit (ALU, MULT, MEM) has its dedicated reservation station for efficient execution. A standout is our impressive branch prediction, with a Gshare predictor, RAS, and BTB achieving an 87.79% hit rate. The GUI Debugger proves invaluable during debugging, ensuring easy bug detection and resolution. Early Tag Broadcast streamlines instruction completion at the Execute-Complete boundary, preventing stalls. The Memory System optimizes memory access with an N-ported I-Cache and K-way set associative D-Cache. The Memory Stage, managing load and store instructions, integrates a Store Queue for effective handling. In summary, our design is a commitment to efficiency, innovation, and a high-performing N-way Superscalar Out-of-Order Processor.

Contents

1 Design Overview

Our Design framework centers around implementing an R10K N-way Out-of-Order Superscalar Processor. Our processor supports 32 bits RV32IM ISA, without fences, division, CSR operations, and system calls.

Our processor's architecture is shaped by strategic design decisions to optimize performance and efficiency by implementing features such as Early Tag Broadcast, Conservative Store-to-Load Forwarding, Victim Cache, and Non-blocking D-Cache. Along with these key units and features, we have implemented the RAT (aka Map Table), RRAT (aka the Architectural Map Table), and PRF (Physical Register File).

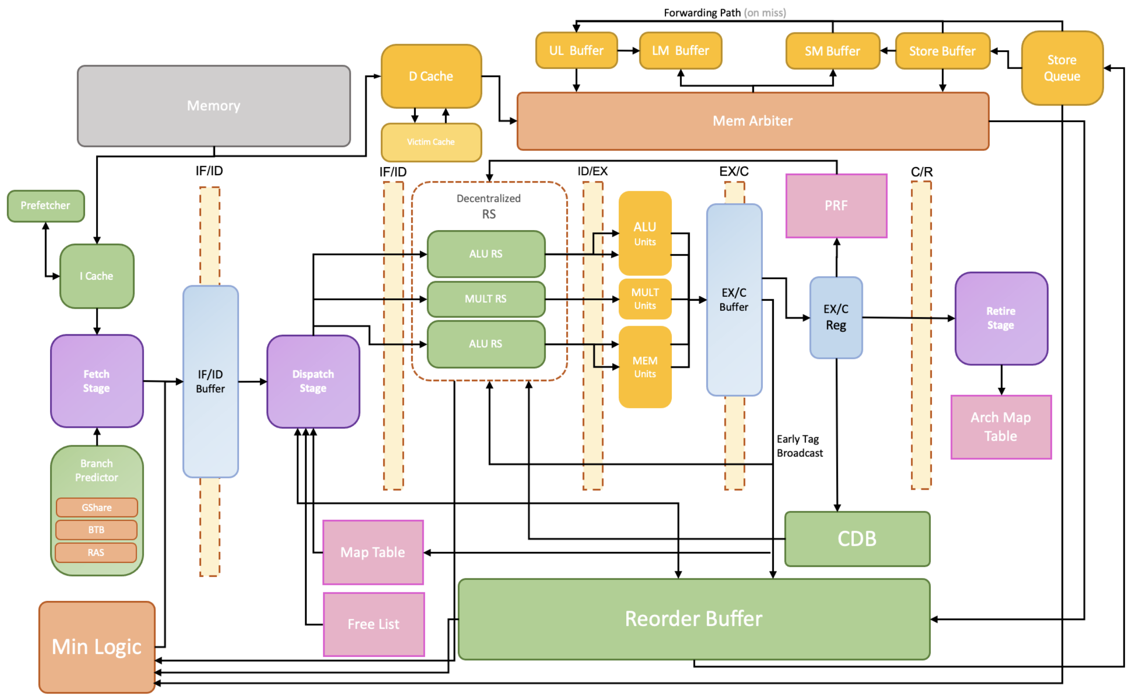


Figure 1: A99 Processor Overview Diagram

We implemented a multi-banked ROB for high scalability. We used a FIFO Queue for Freelist Management (allocating and deallocating Physical Registers (PRs)). We also added a dedicated Distributed Reservation Station for each type of functional unit (ALU, MULT, MEM). For efficient handling of branches, our processor employs a Gshare branch predictor, a Return Address Stack (RAS), and a Branch Target Buffer (BTB) with enhanced branch prediction accuracy. In the memory stage, we included a Store Buffer, and multiple other buffers (store miss buffer, load miss buffer, unanswered load buffer, etc) were implemented to support Conservative store-to-load forwarding.

An integral part of our design, the GUI Debugger was an invaluable tool during the debugging phase (it came to our rescue right when we needed it the most). Its user-friendly interface facilitated efficient bug detection and resolution, granting our team more time for optimization and fine-tuning.

The core features of our design are summarized in the following listed Table 1 below:

Advanced Features	Integrated in Final
N-Way Superscalar	Yes
Early Tag Broadcast	Yes
Victim Cache	Yes
Conservative Store to Load Forwarding	Yes
RAS	Yes
Gshare	Yes
BTB	Yes
Store Buffer	Yes
Non-blocking Dcache	Yes
K-Way Associative DCache	Yes
LRU Replacement Policy for DCache	Yes
GUI Debugger "Vivituber"	Yes
Icache Prefetcher	Yes
N-ported Icache	Yes
Multi-banked RoB	Yes
Distributed RS	Yes
Early Prefetch and Stall on Branch Complete	Yes

Table 1: Status Summary of Advanced Features

For the final submission, we chose the following parameters:

Parameter	Value
N	4
RS Size	16
ROB Size	32
Num ALU FUs	4
Num MEM FUs	3
Num MUL FUs	2
Dcache Associativity	4
Gshare History Length	6
Gshare PC Length	6
BTB Tag Length	8
BTB Target Length	18
BTB Index Length	6
Prefetch Length	6
Store Buffer Size	8
Store Miss Buffer Size	8
Load Buffer Size	32
Load Miss Buffer Size	32
RAS SIZE	32
Free List SIZE	32

Table 2: Final Parameter Values

We achieved an average CPI of **1.10** and a clock period of **20 ns**. Consequently, we achieved an average time per instruction of **22 ns**.

2 Branch Predictor

The branch predictor consists of a Branch Target Buffer, a G-share Direction Predictor, and a Return Address Stack. Branches are predicted at the Fetch (IF) stage, and only one branch is allowed to exist in the fetched instructions at each cycle. In cases where instructions are fetched in the fetched instruction packet after a branch instruction, these instructions are invalidated and treated as if they were never fetched. If the branch instruction is not a function return and the G-share predicts it as a taken branch, the `BTB`'s hit and valid bits are checked. If there's a matching and valid output from the `BTB`, we treat that PC value as the next fetch's start point and send it to the Fetch stage. On the other hand, if the branch is predicted as not taken, the PC will increase based on the number of valid instructions in the fetch packet. If the instruction is a function return, the RAS's head will be popped and sent to the fetch stage as the next PC. Every branch instruction is resolved in the Complete stage, and so are the updates of the branch predictor. An average hit rate of **87.79%** is finally achieved.

2.1 Branch Target Buffer

The Branch Target Buffer takes the current PC as the input, and it outputs a hit bit, a valid bit, and the buffer's row corresponding to the input PC. The hit bit is different from the valid bit: whenever the index and the tag bits match for an input, the hit bit is on. However, only when that BTB row contains some value other than the initial values the valid bit will be on. When updating, both the source and the target PC address should be provided by the Complete stage. The input PC is sliced into index bits and tag bits, and only part of the target PC is stored in the buffer to save space since most branches' targets never exceed too many lines. The number of index, tag, and stored bits are adjustable parameters.

2.2 G-share Direction Predictor

G-share, first proposed by McFarling, is a high-performance branch predictor with a global history and hashing approach to address the locality (McFarling, 1993). The G-share will always output its prediction and the row index for that prediction for any PC input. Instead of using speculative update where the global history register and the branch history table are updated according to the prediction and corrected based on information from the Complete stage, a relatively simple conservative approach is used. The row index for each position is passed along the pipeline. In the Complete stage, after a branch is confirmed, the result and the index are passed back to the G-share for the update. The history length is an adjustable parameter, and a 2-bit saturation counter is used for the branch history table. Note that the initial value is "Strongly Taken" for all entries.

2.3 Return Address Stack

The Return Address Stack stores the PC+4 of a function call in a first-in-last-out (FILO) manner. Specifically, if a `jal` instruction is fetched, that instruction's PC will be sent to the RAS, and the PC+4 is stored in the stack. When a `jalr` instruction is fetched, a pop signal will be sent to the RAS, and the stack's top entry will be popped off the stack, and the PC value in that entry is the predicted return address. When an exception happens in the pipeline, the RAS is also flushed. When the RAS is full, and there's still an incoming push request, it will begin overriding the oldest entries in the stack and reset its head pointer.

3 Reorder Buffer

The ROB is designed with N queues, each having a size of `rows_per_queue`. Each queue functions as a complete 1-wide `rob`. The full `rob` combines these queues with a global head pointer that indicates the next queue to retire. This design simplifies the implementation of a single queue and facilitates testing. However, it introduces complexity when adding parameters, as code changes are required in two modules, and data must be passed between them.

The decision to implement a multi-queue `rob` was made when the team had limited System Verilog experience. Despite thorough debates, the decision was not well thought out. The team is now curious about whether any performance benefits were gained from the multi-queue `rob`, as there is no single-queue version for comparison. It is evident that the multi-queue version made code modifications more challenging, and any potential performance benefits likely did not justify the additional effort. Also, since the queues' retiring order is not fixed every time, there's a row-shifting logic for the output from the rob; this operation is costly in terms of wires and in terms of time. All the complex logic inside the ROB potentially contributes to our critical path and slows down our entire design.

In addition to the regular tags stored in the `rob`, many elements are maintained in a `rob` entry for use in resolving branches instead of simply using branch masks. For example, we store the PC, predicted PC, actual PC, predicted taken, and actual taken in the ROB. This takes an extra 66 bits for each row, and it takes up nearly half of the spaces in the ROB wastefully.

When a mispredicted branch reaches the head of the ROB, it generates an exception bit in the retirement output, which is then propagated as a squash signal throughout the pipeline.

4 Execute-Complete Boundary

The CDB and register writeback (complete stage) are each N wide, but the outputs of stage `EX` may be wider than this. Let W be the width of `EX`. We then forward the first N valid instructions from the W instructions that finished executing into the N wide `ic_reg`, prioritizing `loads`, `muls`, `alu`, and then `stores`. The remaining instructions are put into a buffer, the `ex_ic_buffer`, and will be sent to `ic_reg` in the next cycle with priority over instructions from execute.

To avoid stalls, the `ex_ic_buffer` is sized according to the `ROB_SIZE`.

4.1 Early Tag Broadcast

At any stage, the instructions in the `ic_reg` are the instructions to complete in the next cycle. We therefore broadcast their tags just before they enter the `ic_reg`. This is possible because the valid bits, and the tags for executing instructions are ready at the start of the next cycle, as are instructions in the `ex_ic_buffer`.

4.2 Early Branch Dispatch Stall

If any of the W instructions from execute are mispredicted branches, we send a stall signal to dispatch to prevent incorrect instructions from polluting the bandwidth of the pipe. In particular, we found that since the RS does not issue instructions in a FIFO ordering, new instructions could stop valid instructions from being issued. We also send the branch target to the `icache` to prefetch that value from memory.

The misprediction bit and target are not computed until the end of the cycle, so we store them in a register before forwarding them to fetch and dispatch.

5 Memory System

5.1 I-Cache

The N-ported, direct-mapped `ICache` module plays an instrumental role in optimizing memory access and minimizing cache misses within the system. When provided with a Program Counter (PC) address, the `ICache` is capable of returning N consecutive instructions, starting from the specified PC address, in the event of a cache hit. In scenarios where a cache miss occurs amidst this sequence, the `ICache` efficiently handles this by returning the instructions already present in the cache and simultaneously initiating the fetching of new instructions starting from the point of the miss.

5.1.1 Prefetching

To effectively mask the latency inherent in memory accesses, the `ICache` implements a strategic prefetching mechanism. This mechanism is activated following a memory request triggered by a cache miss. The number of prefetching operations is parameterized, thereby significantly reducing the overall latency of the `ICache`.

5.1.2 Other Optimizations

To further optimize the performance of the `ICache`, the status of each cache line is monitored. This status information, which indicates whether a cache line is valid, in the process of being loaded, or invalid, plays a pivotal role in preventing the duplication of load requests to the same memory address. The status information is shown in the below diagram.

When the I-Cache encounters a cache line marked as invalid, it identifies this as a cache miss. In response, the I-Cache changes the status of the line from `invalid` to `loaded` and initiates a memory load. Simultaneously, it sends an invalid signal to the processor to induce a blocking state. If subsequent requests target a cache line that is already loaded, the I-Cache still sends an invalid signal to the processor, but it will stop sending additional memory requests. Conversely, if a `valid` cache line with a corresponding tag match is requested, the I-Cache sends a valid signal and returns the requested instructions. In cases of a tag mismatch on a valid cache line, the I-Cache shifts the line status from `valid` to `loaded` and requests data from the memory.

5.2 D-Cache

The D-Cache is designed to be K-way set associative, non-blocking, write-through, and write-allocate with LRU replacement policy. When receiving a memory address as input, the D-Cache operates in two distinct

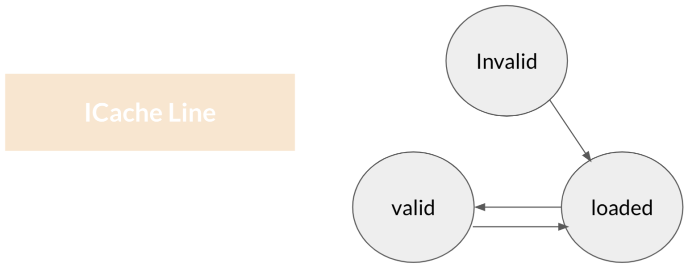


Figure 2: I-Cache line status

modes based on cache status. In the event of a cache hit, it directly returns the value currently stored at the specified memory address. Conversely, during a cache miss, the D-Cache issues a ticket, which signifies the order of the memory access request instead of simply returning an invalid bit. This mechanism between the processor and the D-Cache is mirrored from the interface between the D-Cache and the Memory, providing the fundamentals for non-blocking memory access. The interaction among Processor, D-Cache, and Memory is illustrated in Figure 2:

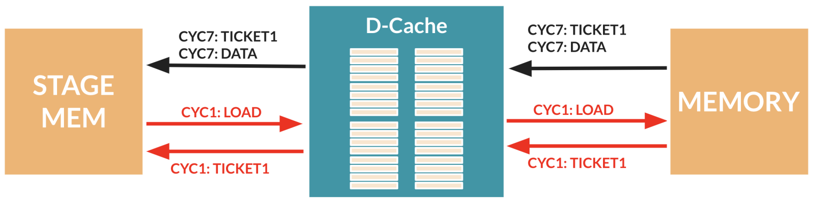


Figure 3: D-Cache workflow

5.2.1 Victim Cache

In an effort to decrease the miss rate of the D-Cache, a fully associative Victim Cache has been integrated within it. This Victim Cache is comprised of four cache lines, each holding 32 bytes. When a line is evicted from the D-Cache due to memory data, it is transferred to the Victim Cache. When a load or store operation accesses any line within the Victim Cache, that line is then promoted to the D-Cache, specifically to the position of the lowest priority as determined by the Least Recently Used (LRU) Replacement Policy. This approach enhances cache efficiency by retaining recently accessed data in a readily accessible location.

5.2.2 LRU Replacement Policy

The LRU Policy is encoded with 10 bits for each D-Cache cache line and Victim Cache cache line. Whenever a memory request hits one cache line, LRU encoding of other cache lines in the corresponding set will be increased by one. The cache line with the highest LRU encoding will be evicted or replaced.

6 Memory Stage

The memory stage is responsible for handling load and store instructions. It can handle multiple memory instructions (parameterized) simultaneously. It is fully pipelined and consists of three stages - Address Computation stage, Store-Queue and Store-Buffers Look-up stage, and D-Cache Look-up stage. Since the memory stage does not explicitly have an MSHR module, the various store and load buffers are responsible for making the D-Cache non-blocking. Even though the D-Cache only allows one memory request per cycle, having separate store-queue, store buffer look-up, and D-Cache look-up stages enables the memory stage to fulfill multiple memory requests per cycle. This is a key reason for better performance on programs having a lot of memory operations.

In addition to the store queue, the memory stage consists of 4 FIFO Queues - Store Buffer, Store Miss Buffer, Unscheduled Load Buffer, and Load Miss Buffer.

6.1 Store Queue

The store queue is a FIFO queue that supports conservative store-to-load forwarding. Multiple loads can look up the store queue in parallel. To prevent the head-tail wrapping around edge cases, one entry is left empty. So, a store queue of size K is full, when it has K-1 entries in it.

6.2 Store Instruction Flow

- When a store instruction is dispatched, it is allocated an entry in the RS and the store queue.
- Once the operands of the store instruction are ready, it is issued from the RS, and it goes to the Address Computation stage. Once the address is computed, it's written in the store's corresponding position in the store queue.
- When the store retires, it is popped from the store queue and inserted into the store buffer.
- When this store instruction becomes the head of the store buffer, it requests the D-Cache memory arbiter. If the request is granted, the store instruction is popped from the store buffer.
- If the store request to the D-Cache is a hit, the data is written to the appropriate cache line. However, if the request was a miss, the store is given a ticket by the D-Cache, and it is inserted into the store miss buffer.
- Every cycle, the D-Cache outputs the tag of the data brought in by the memory. Since the memory requests are fulfilled in order, this tag is only compared with the head entry of the store miss buffer. So when the store reaches the head of the store miss buffer, and its ticket matches with the tag, it is popped, and its data is written to the D-Cache. All of this happens in the D-Cache Look-up stage.

6.3 Load Instruction Flow

- When a load instruction is dispatched, it is allocated an entry in the RS, and its position in the store queue is noted.
- To avoid forwarding of values to load from multiple places due to instructions such as store byte and store half, the processor only issues load when all the stores in the store queue, store buffer, and store miss buffer are store words. Hence, when all the operands are ready, all stores before the load's position in the store queue have their addresses, and when all the stores are store words, the load goes to the Address Computation stage.
- After the load's address is computed, it looks up the store queue, store buffer, and store miss buffer in parallel. If there's an address match, the value is forwarded, and the load goes straight to the complete stage.
- If there's no address match, the load is inserted into the unscheduled load buffer.
- When this load instruction is at the head of the unscheduled load buffer, it requests the D-Cache memory arbiter. If the request is granted, the load instruction is popped.
- If the load request to the D-Cache is a hit, the data is loaded from the appropriate cache line, and the load goes to the complete stage. However, if the request was a miss, the load is given a ticket by the D-cache, and it is inserted into the load miss buffer.
- Similar to the store miss buffer, the tag from the D-Cache is only compared with the head of the load miss buffer. So when the load becomes the head of the load miss buffer and its ticket matches with the tag, it gets its data from the D-Cache. All of this happens in the D-Cache Look-up stage.

7 Testing & Debugging

7.1 Test Strategy

In the beginning, team members are split into different groups to make some initial components. Thus testing is critical before merging everything into the pipeline. For each individual work or group work, a module-specific testbench is included. Both simulation and synthesis tests are included, making sure every component is working correctly individually.

As modules are merged into the pipeline, many more test benches are written collectively by the group members to make each stage and critical function work correctly. For example, the Fetch Stage, Issue Stage, Execute Stage, and Complete Stage are tested before assembling the entire pipeline. This testing approach makes debugging easier for later overall integration.

The integrated pipeline is tested with `$display` statements and the visual debugger. All public programs provided are tested. A shell script is made to do the test and verify the performance for all test cases automatically.

7.2 Testbench

There are three types of testbench developed for the project: the module testbench, the stage testbench, and the pipeline driver testbench. The module testbench is only for a component in the pipeline, and it will only examine whether the input and output are correct based on some provided values. It will also provide some basic visual helper to print out component's values. The stage testbench is designed to test different pipeline stages. It prints the logical values for each stage, and group members can then check whether each stage is working as expected. The pipeline drivers are special testbenches that contain the function of loading memory files, creating the write-back file, formatting the output file, and collecting information to calculate the CPI. There's also a special testbench for the visual debugger to extract all necessary debug outputs from different components in the pipeline.

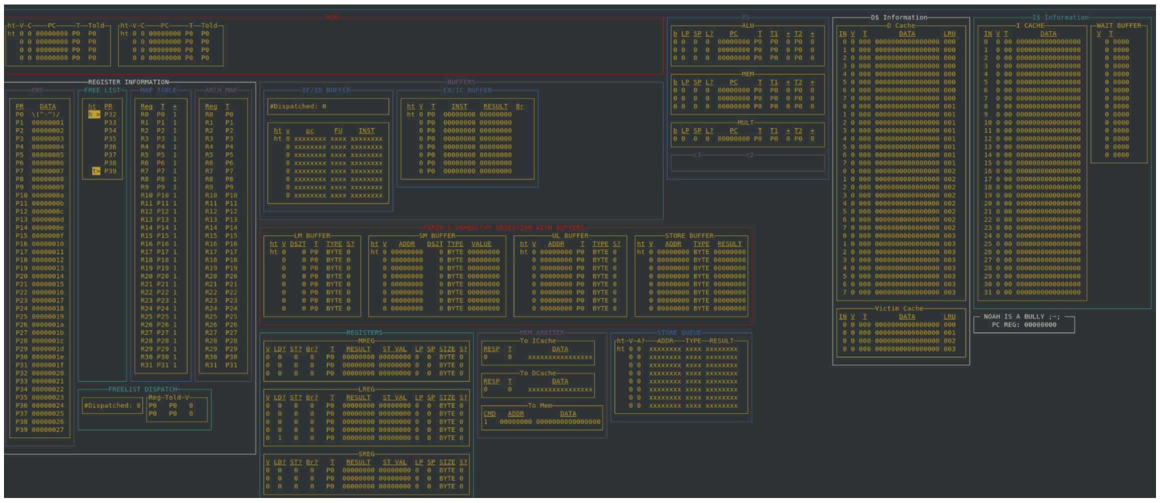
7.3 The Shell Script

The tests are organized in a shell script so that each time group members can perform the task with a single command.

The shell script runs the driver testbench on all available test programs and compares the `.wb` and `.out` files with the ground truth output from `p3-original`. Test/fail for writeback and memory values' check will be printed out into the console. The script also saves the difference between the current test and the ground truth to a log file.

7.4 The Visual Debugger: an Ode to the Vivituber

To help debug, Vivianna developed a visual debugger, named the Vivituber, based on the vtuber from Project 3. The debugger automatically scales according to `N` and allows visualization of useful program structures including, but not limited to, ROB, RS, Dcache, Icache, Store/Load Buffers, and data to and from memory. The debugger can step or jump through clock cycles, and was invaluable in debugging simulation bugs.



```
ROB
ht-V-C----PC-------T--Told      ht-V-C----PC-------T--Told
ht 0 0 00000000 P0   P0         ht 0 0 00000000 P0   P0
   0 0 00000000 P0   P0            0 0 00000000 P0   P0
   0 0 00000000 P0   P0            0 0 00000000 P0   P0
   0 0 00000000 P0   P0            0 0 00000000 P0   P0
```

```
RS
ALU
b LP SP L?    PC      T   T1  +  T2  +
0 0  0  0  00000000 P0   P0  0 P0   0
0 0  0  0  00000000 P0   P0  0 P0   0
0 0  0  0  00000000 P0   P0  0 P0   0
MEM
b LP SP L?    PC      T   T1  +  T2  +
0 0  0  0  00000000 P0   P0  0 P0   0
0 0  0  0  00000000 P0   P0  0 P0   0
0 0  0  0  00000000 P0   P0  0 P0   0
MULT
b LP SP L?    PC      T   T1  +  T2  +
0 0  0  0  00000000 P0   P0  0 P0   0
c1          c2
```

```
REGISTER INFORMATION
PRF                 FREE LIST    MAP TABLE       ARCH MAP
PR   DATA           ht   PR      Reg  T   +      Reg   T
P0   \(^-^)/        h >  P32     R0   P0  1      R0    P0
P1   00000001            P33     R1   P1  1      R1    P1
P2   00000002            P34     R2   P2  1      R2    P2
P3   00000003            P35     R3   P3  1      R3    P3
P4   00000004            P36     R4   P4  1      R4    P4
P5   00000005            P37     R5   P5  1      R5    P5
P6   00000006            P38     R6   P6  1      R6    P6
P7   00000007        t>  P39     R7   P7  1      R7    P7
P8   00000008                    R8   P8  1      R8    P8
P9   00000009                    R9   P9  1      R9    P9
P10  0000000a                    R10  P10 1      R10   P10
P11  0000000b                    R11  P11 1      R11   P11
P12  0000000c                    R12  P12 1      R12   P12
P13  0000000d                    R13  P13 1      R13   P13
P14  0000000e                    R14  P14 1      R14   P14
P15  0000000f                    R15  P15 1      R15   P15
P16  00000010                    R16  P16 1      R16   P16
P17  00000011                    R17  P17 1      R17   P17
P18  00000012                    R18  P18 1      R18   P18
P19  00000013                    R19  P19 1      R19   P19
P20  00000014                    R20  P20 1      R20   P20
P21  00000015                    R21  P21 1      R21   P21
P22  00000016                    R22  P22 1      R22   P22
P23  00000017                    R23  P23 1      R23   P23
P24  00000018                    R24  P24 1      R24   P24
P25  00000019                    R25  P25 1      R25   P25
P26  0000001a                    R26  P26 1      R26   P26
P27  0000001b                    R27  P27 1      R27   P27
P28  0000001c                    R28  P28 1      R28   P28
P29  0000001d                    R29  P29 1      R29   P29
P30  0000001e                    R30  P30 1      R30   P30
P31  0000001f                    R31  P31 1      R31   P31
P32  00000020
P33  00000021       FREELIST DISPATCH
P34  00000022                          Reg-Told-V
P35  00000023       #Dispatched: 0     P0   P0   0
P36  00000024                          P0   P0   0
P37  00000025
P38  00000026
P39  00000027
```

```
BUFFERS
IF/ID BUFFER                              EX/IC BUFFER
#Dispatched: 0                            ht V  T    INST     RESULT   Br
                                          ht 0 P0  00000000 00000000
ht v    pc       FU     INST                 0 P0  00000000 00000000
ht 0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx                  0 P0  00000000 00000000
   0 xxxxxxxx xxxx xxxxxxxx
   0 xxxxxxxx xxxx xxxxxxxx
```

```
PARIN'S UNHEALTHY OBSESSION WITH BUFFERS
LM BUFFER                 SM BUFFER                             UL BUFFER                      STORE BUFFER
ht V D$2T  T  TYPE S?     ht V   ADDR    D$2T TYPE  VALUE       ht V   ADDR    T   TYPE S?     ht V   ADDR    TYPE  RESULT
ht 0    0 P0  BYTE 0      ht 0 00000000     0 BYTE 00000000     ht 0 00000000 P0   BYTE 0      ht 0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
   0    0 P0  BYTE 0         0 00000000     0 BYTE 00000000        0 00000000 P0   BYTE 0         0 00000000 BYTE 00000000
```

```
REGISTERS
MREG
V LD? ST? Br?  T   RESULT   ST VAL   LP SP SIZE S?
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
LREG
V LD? ST? Br?  T   RESULT   ST VAL   LP SP SIZE S?
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  1   0   0   P0 00000000 00000000 0  0  BYTE 0
SREG
V LD? ST? Br?  T   RESULT   ST VAL   LP SP SIZE S?
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
0  0   0   0   P0 00000000 00000000 0  0  BYTE 0
```

```
MEM ARBITER
To ICache
RESP  T          DATA
0     0    xxxxxxxxxxxxxxxx
To DCache
RESP  T          DATA
0     0    xxxxxxxxxxxxxxxx
To Mem
CMD   ADDR          DATA
1   00000000 0000000000000000
```

```
STORE QUEUE
ht-V-A?---ADDR----TYPE--RESULT
ht 0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
   0 0  xxxxxxxx xxxx xxxxxxxx
```

```
D$ Information
D Cache
IN V  T         DATA          LRU
 0 0 000 0000000000000000 000
 1 0 000 0000000000000000 000
 2 0 000 0000000000000000 000
 3 0 000 0000000000000000 000
 4 0 000 0000000000000000 000
 5 0 000 0000000000000000 000
 6 0 000 0000000000000000 000
 7 0 000 0000000000000000 000
 0 0 000 0000000000000000 001
 1 0 000 0000000000000000 001
 2 0 000 0000000000000000 001
 3 0 000 0000000000000000 001
 4 0 000 0000000000000000 001
 5 0 000 0000000000000000 001
 6 0 000 0000000000000000 001
 7 0 000 0000000000000000 001
 0 0 000 0000000000000000 002
 1 0 000 0000000000000000 002
 2 0 000 0000000000000000 002
 3 0 000 0000000000000000 002
 4 0 000 0000000000000000 002
 5 0 000 0000000000000000 002
 6 0 000 0000000000000000 002
 7 0 000 0000000000000000 002
 0 0 000 0000000000000000 003
 1 0 000 0000000000000000 003
 2 0 000 0000000000000000 003
 3 0 000 0000000000000000 003
 4 0 000 0000000000000000 003
 5 0 000 0000000000000000 003
 6 0 000 0000000000000000 003
 7 0 000 0000000000000000 003
Victim Cache
IN V  T         DATA          LRU
 0 0 000 0000000000000000 000
 0 0 000 0000000000000000 001
 0 0 000 0000000000000000 002
 0 0 000 0000000000000000 003
```

```
I$ Information
I CACHE                              WAIT BUFFER
IN V T         DATA                  V  T
0   0 00 0000000000000000            0 0000
1   0 00 0000000000000000            0 0000
2   0 00 0000000000000000            0 0000
3   0 00 0000000000000000            0 0000
4   0 00 0000000000000000            0 0000
5   0 00 0000000000000000            0 0000
6   0 00 0000000000000000            0 0000
7   0 00 0000000000000000            0 0000
8   0 00 0000000000000000            0 0000
9   0 00 0000000000000000            0 0000
10  0 00 0000000000000000            0 0000
11  0 00 0000000000000000            0 0000
12  0 00 0000000000000000            0 0000
13  0 00 0000000000000000            0 0000
14  0 00 0000000000000000            0 0000
15  0 00 0000000000000000
16  0 00 0000000000000000
17  0 00 0000000000000000
18  0 00 0000000000000000
19  0 00 0000000000000000
20  0 00 0000000000000000
21  0 00 0000000000000000
22  0 00 0000000000000000
23  0 00 0000000000000000
24  0 00 0000000000000000
25  0 00 0000000000000000
26  0 00 0000000000000000
27  0 00 0000000000000000
28  0 00 0000000000000000
29  0 00 0000000000000000
30  0 00 0000000000000000
31  0 00 0000000000000000

NOAH IS A BULLY ;~;
  PC REG: 00000000
```

Figure 4: Screenshot of Vivituber running

7.5 Test Results

For simulation, all test programs (listed in Appendix A) passed for $N = 2, 3, 4, 5, 8$.

For synthesis, all test programs passed for $N = 2, 3, 4$ with a clock period of 20ns.

8 Performance Metric

8.1 Average CPI

The CPI (Clock cycles Per Instruction) of a program is defined as:

$$\textbf{CPI} = \frac{\textbf{\#Clock Cycle}}{\textbf{\#Completed Instructions}}$$

Average CPI is calculated by dividing the total clock cycles required by all the programs by the total number of instructions in all the programs.

8.2 Overall Performance

The "performance" used to evaluate the efficiency of the processor is the average time in nanoseconds that an instruction spends between being fetched and retired. The formal calculation could be derived from the CPI:

$$\textbf{Performance} = \textbf{Average CPI} \times \textbf{Clock Period} = \frac{\textbf{\#Total Clock Cycles}}{\textbf{\#Total Completed Instructions}} \times \textbf{Clock Period}$$

9 Experiments & Analysis

9.1 CPI vs Clock Period

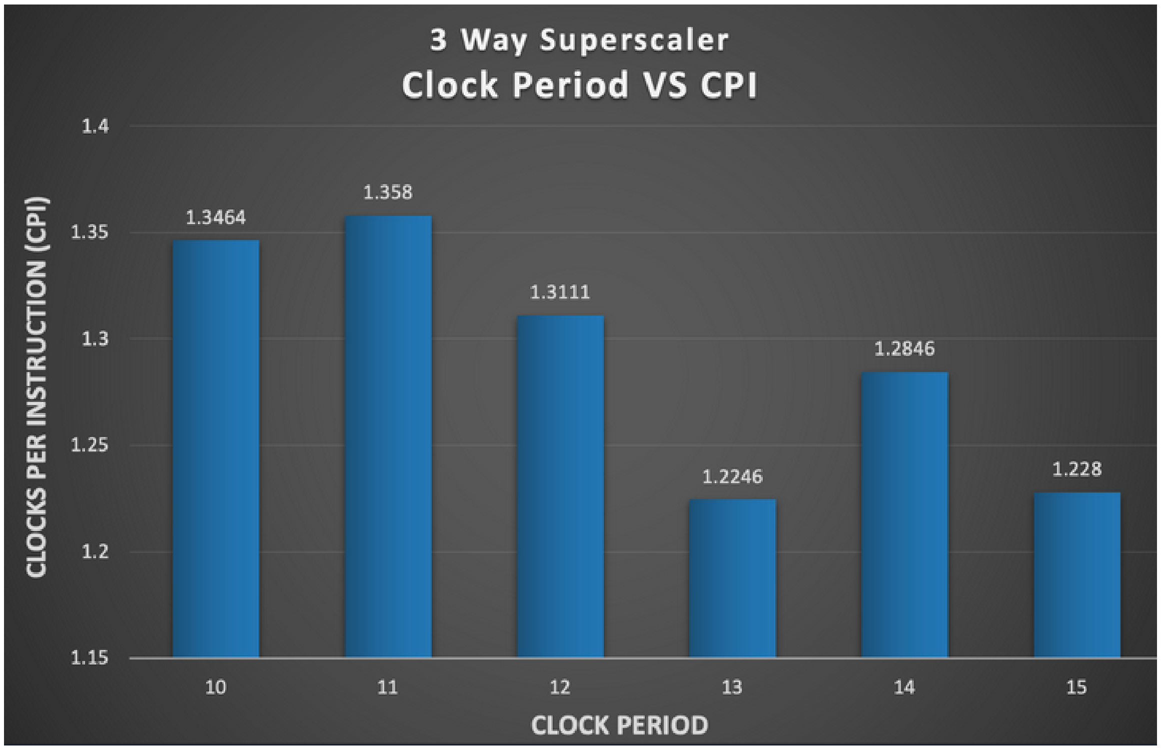


Figure 5: CPI for different clock periods with N=3

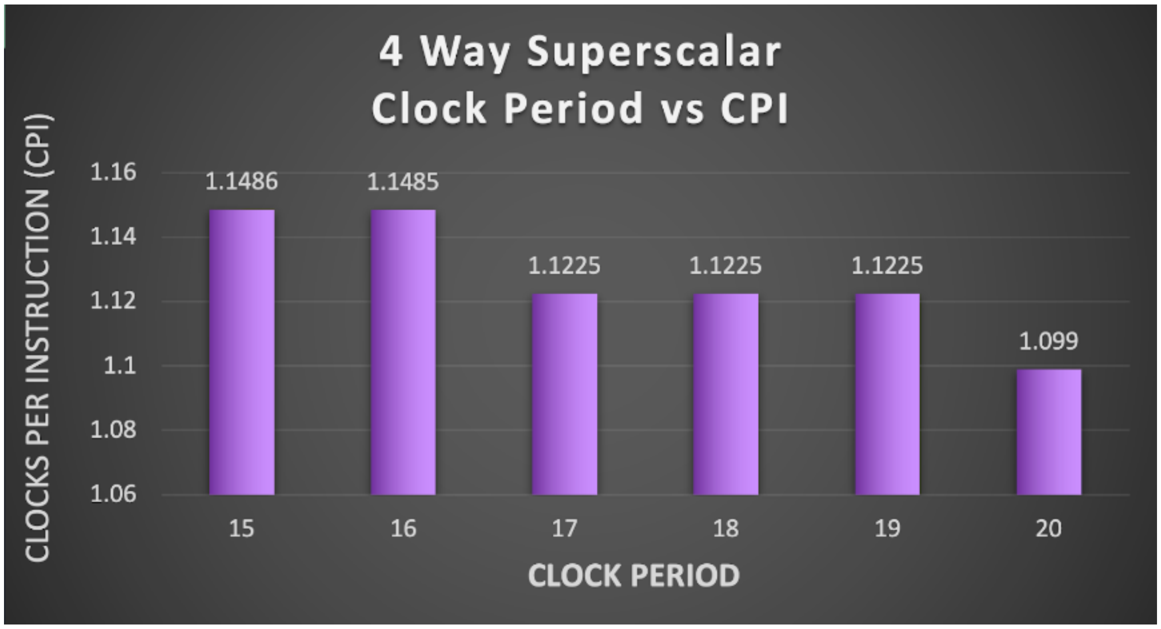


Figure 6: CPI for different clock periods with N=4

Because a lower clock period will increase the number of cycles to access memory, a lower clock period can negatively affect the CPI. To ensure that we maximize the Clock-CPI product while ensuring that the processor passes post-synthesis simulations, we did two scans over the clock period for N=3 and N=4. The results are shown in figures 5, and 6. We found that `N`=4 produces a better Clock-CPI product, and so we chose this `N` for our final design. Consequently, we get an average performance of **22 ns** per instruction.

9.2 CPI Improvement with Branch Predictor

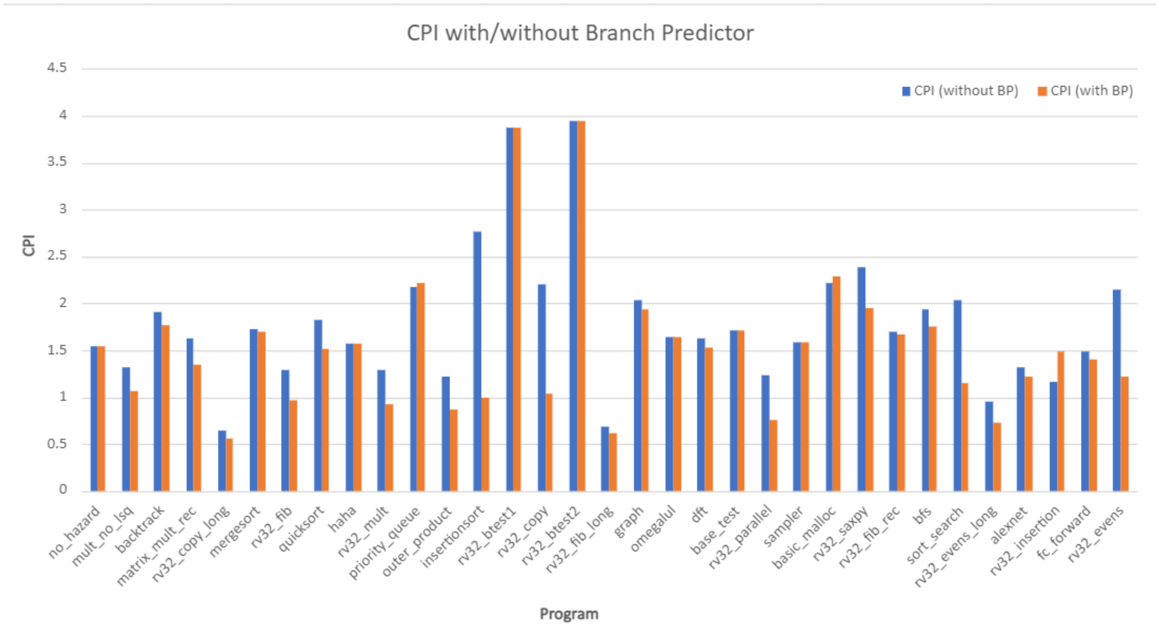


Figure 7: CPI on testbench programs for with and without branch predictor

Without branch predictor, RAS, or BTB, predicting not taken at every branch, we get an average CPI of 1.61, compared to 1.10 with branch prediction. A full breakdown across programs in the testbench is shown in figure 7

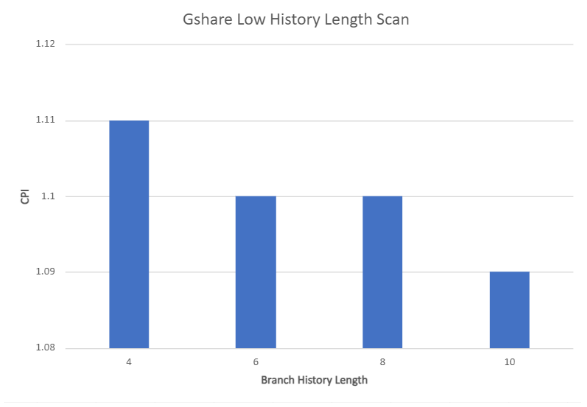


Figure 8: Gshare low history values: This scan evaluates cases where the global branch history length is the same as the number of bits used to index the PC. From left to right, this scan evaluates history lengths of 4, 6, 8, and 10. The other parameters were held fixed with `N`=4, `NUM_ROB_ROWS_PER_QUEUE`=8, `BTB_INDEX_WIDTH`=6, `BTB_TAG_WIDTH`=8, `BTB_TARGET_WIDTH`=15, `RAS_SIZE`=32, and `SIZE_RS`=8. Note that this scan was taken before some minor bug fixes that raise CPI, but we expect the general trends to be reliable.

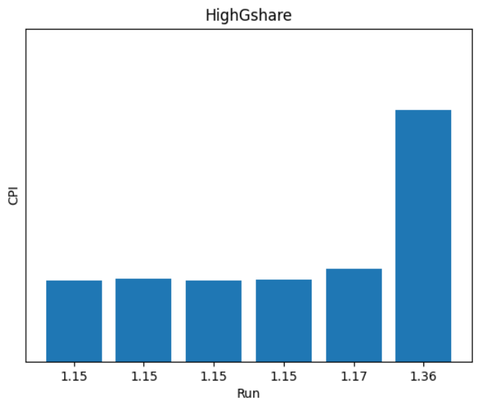


Figure 9: Gshare high history values: This scan evaluates cases where the global branch history length is the same as the number of bits used to index the PC. From left to right, this scan evaluates history lengths of 12, 14, 16, 18, 20, 22, 24, and 30. The other parameters were held fixed with `N`=4, `NUM_ROB_ROWS_PER_QUEUE`=8, `BTB_INDEX_WIDTH`=14, `BTB_TAG_WIDTH`=2, `BTB_TARGET_WIDTH`=16, `RAS_SIZE`=64, and `SIZE_RS`=8.

Branch predictor performance is sensitive to the number of bits used for the gshare history, and the number of PC bits used in the XOR. A larger history length will enable the branch predictor to remember more complicated branch sequences and increase the size of the table so branches are less likely to be remapped to the same spot. However, a branch predictor with a larger history will take longer to warm up.

To limit the search space, we only explore combinations where gshare history is the same length as the PC. We initially experimented with low values of gshare history, shown in figure 8. This figure was generated before all of the bugs in our pipeline were fixed, so the CPIs are not accurate, but we expect the trends to be consistent. Our initial scan showed that CPI improves all the way up to a history length of 10, so we conducted a second study over higher history lengths, figure 12. These results indicate that CPI steadily increases at lengths longer than 12. This is likely because at length 10 we are able to accurately predict most patterns that appear in the testbench. Any longer history only increases warmup time.

9.3 BTB Parameter Variation

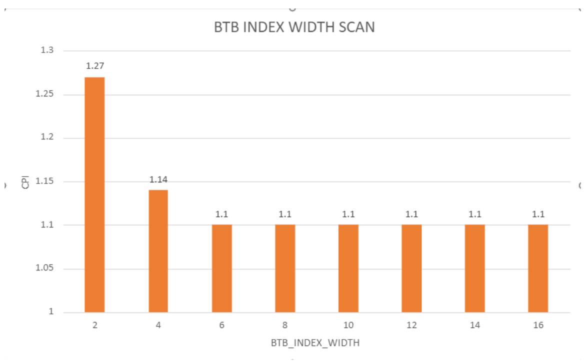


Figure 10: BTB index width scan, varying index from 2 to 16.

The chief tunable parameter for the BTB is the index width. A small index width will result in a smaller, faster table, but a larger index width will produce fewer collisions between branch instructions. We did a scan over the index width, figure 10, and found that at an index width of around 6, we stop seeing performance benefits from increasing the size of the table. At this point, branch instructions are probably no longer colliding in the table.

9.4 ICache Prefetch Depth Variation

It's observed that `Prefetch Depth` = 10 gives us the best performance. It might caused by the reason that the average length of basic blocks in the benchmarks is close to 10. This long prefetch depth is able to reduce cache miss without polluting the ICache.

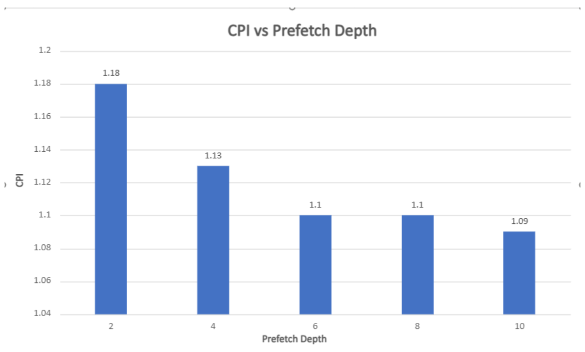


Figure 11: Variations of Prefetch parameters. The other parameters are kept the same `N`=4, `NUM_ROB_ROWS_PER_QUEUE`=8, `GSHARE_HISTORY_LENGTH`=6, `GSHARE_PC_N`=6, `RAS_SIZE`=32, and `SIZE_RS`=16.

9.5 D-Cache K-way Set Associativity Variation

Figure 8. compares the cache hit rates at different levels of cache associativity: direct-mapped (1-way), 2-way, 4-way, and 8-way. As associativity increases to 2-way, 4-way, and 8-way, the cache can accommodate more data items that would otherwise conflict in a direct-mapped cache. This added flexibility reduces the number of conflict misses and can better take advantage of the locality properties of application access patterns, resulting in a higher cache hit rate.

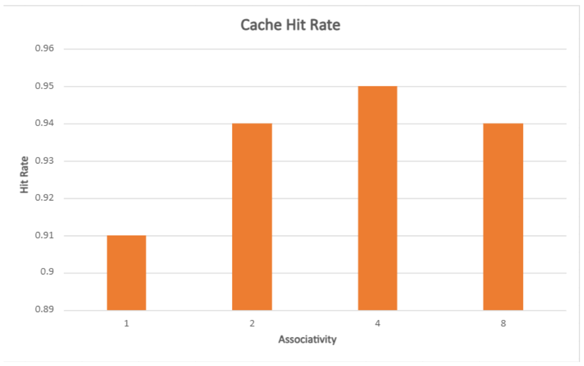


Figure 12: Variations of DCache set associativity parameters. The other parameters are kept the same `N`=4, `NUM_ROB_ROWS_PER_QUEUE`=8, `GSHARE_HISTORY_LENGTH`=6, `GSHARE_PC_N`=6, `RAS_SIZE`=32, and `SIZE_RS`=16.

10 Broader Impacts

#ifdef humor

We, the team of A99, feel that it is our duty to make the world a better place. Every night we worked at the dude, we would think to ourselves, "How can I direct the impacts of my work? How will this processor contribute to society?" That's why we decided to build this processor, to contribute to our vision of a brighter, more inclusive tomorrow. All it takes is a little more computing. We believe our processor will join the millions of products designed by engineers trying to indiscriminately solve problems by flinging more silicon at them.

#else

This was an educational project. Some team members developed skills in Verilog that they will use in their careers. Others developed skills in Verilog that they will never use again.

#endif

11 Acknowledgments

We would like to thank our instructor, Ron, and the IA/GSIs for their guidance over the course of this project.

12 Conclusion

Our processor gets a CPI of 1.10 with a clock rate of of 20ns for a decent set of parameters. Our analysis after submission indicates that with optimal parameters, we could do even better than this, but synthesis errors are exposed at lower clock frequencies, so we're stuck with 20ns. We expect that much of this performance came from early tag broadcast and a wide memory stage with opportunities for store load forwarding.

In hindsight, our multi-queue ROB was probably not worth the added complexity, and in later stages, we sometimes rushed into coding too early before specifying the module interfaces.

If we had additional time, we would work to debug our synthesis errors, and explore a wider than `N` fetch unit to ensure the buffer remains full.

References

Scott McFarling. 1993. Combining branch predictors. *Technical Report TN-36, Digital Western Research Laboratory*, 49.

A Appendix

A.1 Test Programs List

Every test program provided by the teaching team are used to test the pipeline. The `.c` and `.s` files are listed below:

1. `alexnet.c`
2. `backtrack.c`
3. `basic_malloc.c`
4. `bfs.c`
5. `dft.c`
6. `fc_forward.c`
7. `graph.c`
8. `insertionsort.c`
9. `matrix_mult_rec.c`
10. `mergesort.c`
11. `omegalul.c`
12. `outer_product.c`
13. `priority_queue.c`
14. `quicksort.c`
15. `sort_search.c`
16. `crt.s`
17. `haha.s`
18. `mult_no_lsq.s`
19. `no_hazard.s`
20. `rv32_btest1.s`
21. `rv32_btest2.s`
22. `rv32_copy_long.s`
23. `rv32_copy.s`
24. `rv_32_evens.s`
25. `rv32_evens_long.s`
26. `rv32_evens.s`
27. `rv32_fib_long.s`
28. `rv32_fix_rec.s`
29. `rv32_fib.s`
30. `rv32_halt.s`
31. `rv32_insertion.s`
32. `rv32_mult.s`
33. `rv32_parallel.s`
34. `rv32_saxpy.s`
35. `sampler.s`