314513064-lab5-report

1. Introduction

In this lab, I integrated a memory system into the RISC-V pipelined processor. I implemented a **Baseline Instruction Cache (I-Cache)** using a direct-mapped architecture and an **Alternative Data Cache (D-Cache)** featuring a 2-way set-associative design with an LRU replacement policy. Additionally, a **Victim Cache** was incorporated into the D-Cache to mitigate conflict misses. The design was verified using both directed assembly tests and random simulations, and performance was evaluated by comparing cycle counts across different cache configurations (None, I-Cache only, D-Cache only, and All) using standard benchmarks.

2. Design

2.1 Baseline Design (I-Cache)

I implemented a **2KB Direct-Mapped Cache** with 64-byte blocks using vc_RAM_1w1r_pf for storage arrays, strictly avoiding register-based implementation. The controller employs a **Write-Allocate** policy, utilizing an FSM that transitions between IDLE, READ_BLOCK (memory refill), and WRITE_BLOCK to manage instruction fetches and evictions.

2.2 Alternative Design (D-Cache)

The D-Cache is a **4KB 2-way Set-Associative** design using vc_RAM_1w2r_pf to allow simultaneous way access.

- **Architecture:** The 32 sets are indexed via 5 bits, with way selection managed by manipulating the address MSB (e.g., {way_bit, index}).
- **Logic:** It features an **LRU replacement policy** tracked by old bits and correctly handles sub-word accesses (sb, sh) using read-modify-write masking during S_IDLE and S_RESPONSE states.

2.3 Victim Cache

To reduce conflict misses, I integrated a **2-entry Fully Associative Victim**

Cache.

- **Mechanism:** On a D-Cache miss, the controller inspects the Victim Cache. A **hit** triggers a swap between the D-Cache LRU block and the victim block (S_VICTIM_FILL), bypassing main memory. A **miss** results in the evicted D-Cache line being stored in the Victim Cache, pushing its own LRU entry to memory if full.

3. Testing Methodology

I adopted a multi-layered testing strategy, progressing from unit verification to system-level stress testing.

3.1 Custom Directed Testing

To cover microarchitectural corner cases missed by standard tests, I developed a custom assembly suite (riscv-cache-test.S) targeting specific scenarios:

- **Sub-word Access:** Verified the read-modify-write and bit-masking logic for sb and sh instructions to ensure partial writes do not corrupt the cache line.
- **Associativity & LRU:** I used memory addresses with a 2048-byte stride (0x800) to force index collisions. This verified that the 2-way associative design utilizes both ways and that the LRU policy correctly evicts the least recently used block while preserving the MRU block .
- **Victim Cache & Eviction:** I simulated set thrashing by accessing 4 distinct blocks mapping to the same set. This confirmed that the Victim Cache correctly captures evicted dirty lines and performs swaps upon victim hits, effectively reducing miss penalties .

3.2 Standard & Randomized Testing

I utilized the provided test harness for broader validation:

- **Functional Verification (make check-asm):** Verified basic Instruction Cache fetch and Data Cache load/store operations.
- **Stress Testing (make check-asm-rand):** Injected random delays (0-50 cycles) into memory signals. This validated the robustness of the FSM

state transitions (particularly S_WAIT_MEMRESP) and handshake protocols (val/rdy) under variable latency conditions.

3.3 Strategy

The testing followed an incremental approach: first verifying the Baseline I-Cache, then the Alternative D-Cache (fixing masking logic), and finally validating the Victim Cache integration to ensure seamless cooperation between the hierarchy levels.

4. Evaluation

Benchmark	None	Icache	Dcache	All
bin-search-rand	1,101,042	28,381	145,576	82,026
cmplx-mult-rand	10,114,220	92,022	173,268	80,934
masked-filter-rand	13,450,246	194,634	482,762	184,336
vvadd-rand	3,121,351	27,194	57,588	40,154

Benchmark	None (IPC)	Icache (IPC)	Dcache (IPC)	All (IPC)
bin-search-rand	0.00094	0.03629	0.00708	0.01256
cmplx-mult-rand	0.00017	0.01872	0.00994	0.02129
masked-filter-rand	0.00035	0.02448	0.00987	0.02584
vvadd-rand	0.00015	0.01677	0.00787	0.01136

5. Discussion

5.1 Performance Analysis & Trade-offs

The evaluation confirms that **memory locality is the primary driver of performance**.

- **I-Cache Dominance:** The **I-Cache** configuration yielded the most significant performance gain. Since instruction fetches occur every cycle, caching them removes the 50-cycle memory penalty for nearly every instruction. In contrast, the **D-Cache** only benefits load/store instructions (approx. 20-30% of operations), making its isolated impact smaller.

- **The "All" Configuration Anomaly:** Surprisingly, the All configuration (Base+Alt) underperformed compared to Icache alone in benchmarks with poor locality (bin-search and vvadd). This is due to **write-allocation overhead**. When the D-Cache hit rate is low, the controller spends cycles fetching entire blocks from memory. If the spatial locality is poor (data isn't reused), this "fill" latency exceeds the cost of simply bypassing the cache, causing a net performance loss compared to the I-Cache-only setup.

5.2 Victim Cache Impact

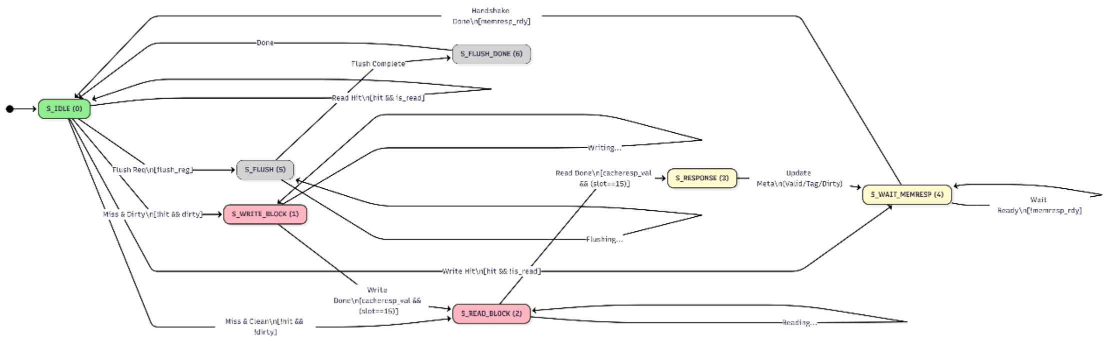
The **2-entry Victim Cache** proved essential for mitigating **conflict misses** inherent in the 2-way set-associative design .

- **Mechanism:** In scenarios where multiple active blocks map to the same set (e.g., set thrashing), the Victim Cache captures evicted dirty lines.
- **Benefit:** Instead of incurring a full main memory penalty (50+ cycles) to retrieve a recently evicted block, the processor retrieves it from the Victim Cache in just 1-2 cycles (Victim Hit). This effectively acts as a fully-associative "L1.5" buffer, smoothing out performance dips caused by "ping-pong" evictions in the main D-Cache.

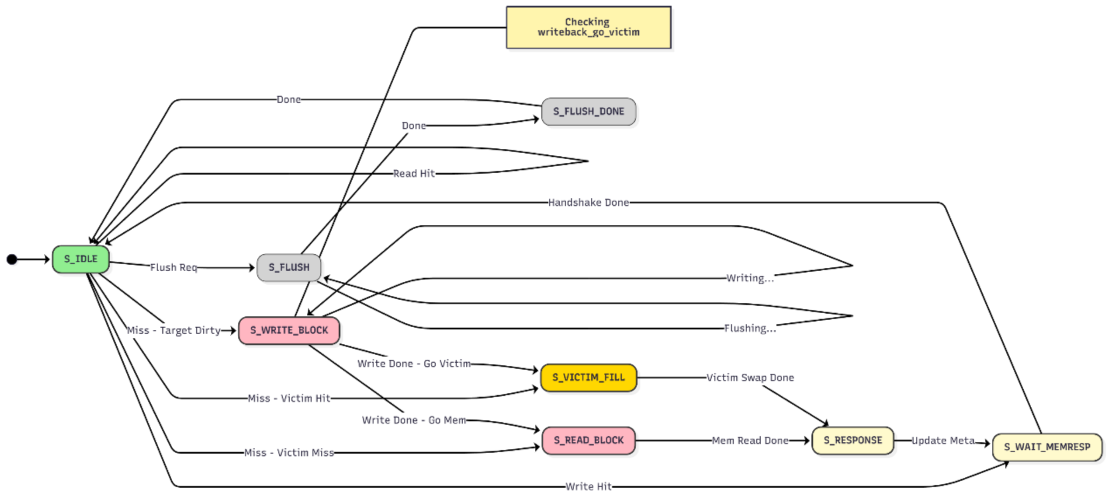
5.3 Architectural Trade-offs

- **Direct-Mapped (Base) vs. Set-Associative (Alt):** The Direct-Mapped I-Cache is hardware-efficient but suffers from conflict misses. The 2-way D-Cache reduces these conflicts but introduces complexity (LRU logic, way muxing).
- **Capacity vs. Latency:** While larger caches increase hit rates, the management overhead (tag checks, LRU updates) can become a bottleneck if the workload lacks sufficient locality to justify the allocation cost.

6. FIGURES


Handshake Done\n[memresp_rdy]
Done
S_FLUSH_DONE (6)
Flush Complete
Read Hit\n[hit && is_read]
S_IDLE (0)
Writing...
Flush Req\n[flush_req]
S_FLUSH (5)
Read Done\n[cacheresp_val && (slot==15)]
S_RESPONSE (3)
Update Meta\n(Valid/Tag/Dirty)
S_WAIT_MEMRESP (4)
Wait Ready\n[!memresp_rdy]
Miss & Dirty\n[!hit && dirty]
S_WRITE_BLOCK (1)
Flushing...
Write Hit\n[hit && !is_read]
Write Done\n[cacheresp_val && (slot==15)]
S_READ_BLOCK (2)
Miss & Clean\n[!hit && !dirty]
Reading...

Baseline FSM (I-Cache)



Alternative Design FSM (D-Cache with Victim Cache)