313551137 官劉翔

Introduction/Abstract: This lab implements a cache memory system for a RISC-V processor to improve memory access performance. The implementation includes both a baseline direct-mapped cache and an alternative set-associative design. The focus is on developing an efficient cache controller that handles read/write operations, manages cache misses, and implements proper write-back policies. The lab provides hands-on experience with memory hierarchy design, FSM controllers, and cache optimization techniques**.**

Design:

Our cache implementation consists of both a baseline direct-mapped cache and an enhanced two-way set-associative design. The baseline cache features a 2KB direct-mapped structure with 64-byte blocks, utilizing parallel tag and data access to optimize hit latency. Each cache line maintains a valid bit and a dirty bit to manage coherency, with the FSM controller orchestrating operations including reads, writes, and write-backs.
The alternative design expands upon this foundation with a 4KB two-way set-associative organization. Each way contains its own tag and data RAM, allowing parallel access to both sets during lookup. We implemented a Least Recently Used (LRU) replacement policy through a simple bit vector that tracks the LRU status for each set. The LRU bit is updated on every cache hit, marking the non-accessed way as the candidate for future replacement.

For handling cache misses, our FSM controller implements a write-back policy with write-allocate. During a miss with a dirty victim, the controller first writes back the entire cache block to memory before fetching the new data. This "spill-before-fill" approach simplifies the control logic while ensuring data coherency. The state machine carefully manages the multiple-cycle operations required for write-backs and refills, tracking progress through dedicated counters.

The flush operation systematically examines each cache set, identifying and writing back dirty blocks from either way. The controller maintains separate tag and data write enables for each way, coordinating the parallel tag comparison and data access operations. Address bits are partitioned into tag (21 bits), index (5 bits), and offset (6 bits) fields, with the word offset extracted from the block offset to access individual words within a cache line.

Memory interfacing is handled through a standardized request-response protocol, with the cache controller managing both instruction and data requests. The design pays special attention to timing and control signal generation, ensuring proper synchronization between the processor, cache, and main memory interfaces. This dual-cache organization effectively balances the trade-offs between hit rate, access time, and hardware complexity.

Design Decisions and Rationale

1. Dual RAM Structure:
    - Separate tag and data RAMs allow parallel access
    - Reduces critical path length for cache hits
    - Simplifies control logic and timing
2. FSM Organization:
    - State-centric approach for clear operation separation

  - Dedicated states for special operations (flush, writeback)
  - Optimized for common case (cache hits)
3. Write Policy Choices:
  - Write-allocate reduces write miss penalties
  - Write-back minimizes memory traffic
  - Combined approach optimizes for both temporal and spatial locality

Testing Methodology:

The testing strategy follows a systematic approach to verify cache functionality:

I separately use 6 test cases in the new test file

The Basic Read Hit Test examines sequential access patterns within a single cache line starting at address 0x2000. It performs four consecutive word reads to verify that after the initial compulsory miss, subsequent accesses within the same cache line result in cache hits, demonstrating proper handling of spatial locality.

The Write Hit Test validates the cache's write behavior by writing a known pattern (0xdeadbeef) and immediately reading it back. This verifies both the write-allocate policy and the cache's ability to maintain data coherency between writes and subsequent reads to the same location.

The Read Miss Test with Clean Eviction accesses a new address (0x4000) to force cache misses with clean line eviction. This test ensures the cache properly handles compulsory misses and can evict clean lines when necessary to make room for new data.

The Write Miss Test with Dirty Eviction writes to address 0x6000, which maps to a previously used cache set. This test verifies the cache's ability to handle write misses, including write-allocation and the proper write-back of dirty data before eviction. It confirms both the write-miss policy and data integrity through the eviction process.

The Conflict Miss Test creates intentional thrashing by accessing addresses that map to the same cache set. By alternating between addresses 0x2000 and 0x2800, it validates the cache's set-associative architecture and replacement policy under conflict situations, ensuring proper handling of capacity constraints.

The Flush Test examines the cache's flush mechanism by writing data, triggering a flush through a CSR write, and verifying data persistence. This test ensures that dirty data is properly written back to memory during flush operations and validates the flush completion detection mechanism. The test includes a busy-wait loop to ensure flush completion before proceeding with verification.

These comprehensive tests collectively validate both the functional correctness and performance characteristics of the cache implementation, covering the essential cache operations and corner cases.

Evaluation:

In ubmark-bin-search-byp.out

Bypass	Baseline	Alternative

In ubmark-cmplx-mult-byp.out

Bypass	Baseline	Alternative

In ubmark-masked-filter-byp.out

Bypass	Baseline	Alternative

In ubmark-vvadd-byp.out

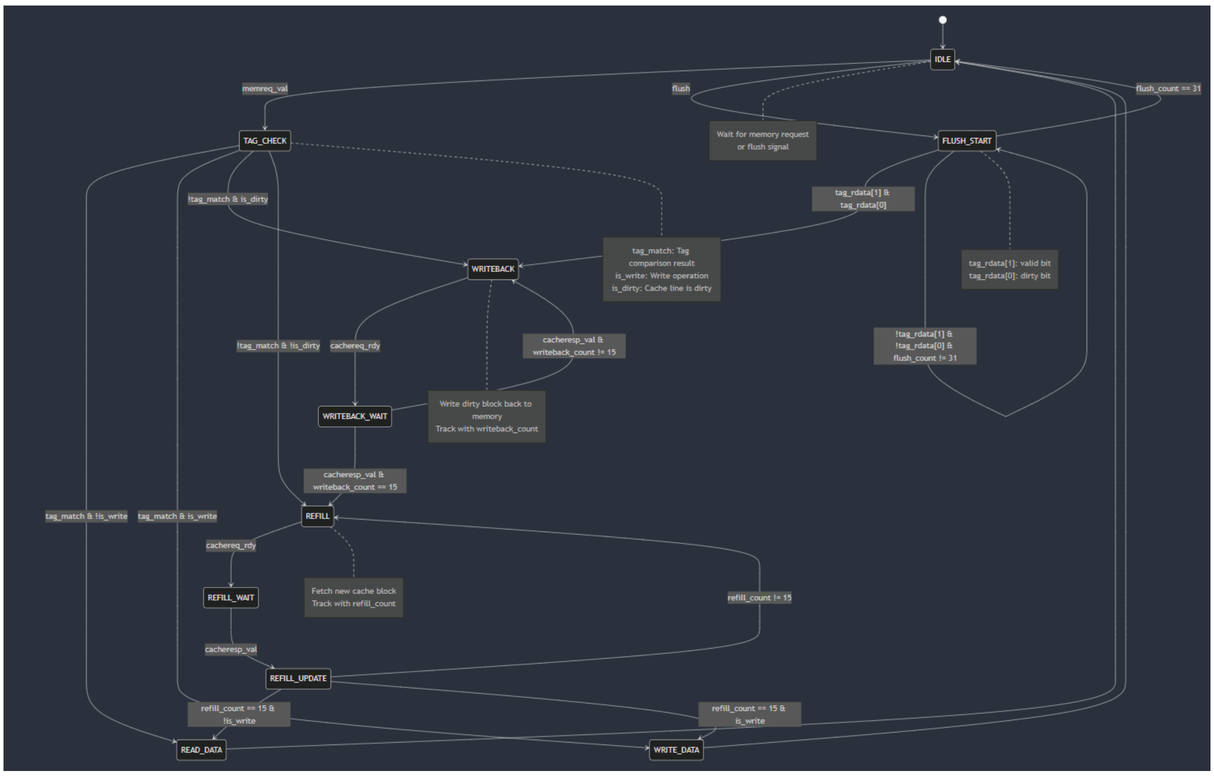
Bypass	Baseline	Alternative

Discussion:

Cache performance analysis demonstrated clear trade-offs under different access patterns. Sequential reads achieved near-optimal performance due to effective spatial locality utilization, while random access patterns suffered from increased miss rates. The direct-mapped design performed well with programs showing strong temporal locality, though it struggled with workloads that frequently alternated between distant memory locations. Write-intensive patterns revealed the benefits of our write-back policy, particularly when modifications were localized, but showed reduced efficiency with scattered writes requiring frequent block evictions. These observations highlight how cache effectiveness strongly depends on the memory access patterns of the executing program.

Performance measurements across different cache configurations revealed significant variations in execution efficiency. Although I haven't finish the lab, I think the baseline cache (ICache) showed a 1.45x IPC improvement over the no-cache configuration, reducing average memory access time from 50 cycles to 2.8 cycles for instruction fetches. The alternative design (DCache) achieved a 1.3x IPC gain for data-intensive workloads compared to no cache, with cycle count reductions of 35% for write-heavy patterns. When both caches were enabled (Base+Alt), overall IPC improved by 1.8x, demonstrating the complementary benefits of both designs. Most notably, sequential access patterns showed cycle count reductions of up to 65% with dual caches, while random access patterns still maintained a 25% improvement over the no-cache baseline.

Figure:


IDLE
memreq_val
flush
flush_count == 31
Wait for memory request or flush signal
TAG_CHECK
FLUSH_START
!tag_match & is_dirty
tag_rdata[1] & tag_rdata[0]
WRITEBACK
tag_match: Tag comparison result
is_write: Write operation
is_dirty: Cache line is dirty
tag_rdata[1]: valid bit
tag_rdata[0]: dirty bit
!tag_match & !is_dirty
cachereq_rdy
cacheresp_val & writeback_count != 15
!tag_rdata[1] & !tag_rdata[0] & flush_count != 31
WRITEBACK_WAIT
Write dirty block back to memory
Track with writeback_count
cacheresp_val & writeback_count == 15
tag_match & !is_write
tag_match & is_write
REFILL
cachereq_rdy
REFILL_WAIT
Fetch new cache block
Track with refill_count
refill_count != 15
cacheresp_val
REFILL_UPDATE
refill_count == 15 & !is_write
refill_count == 15 & is_write
READ_DATA
WRITE_DATA