Lab 5 Report

313553054 陳冠霖

1. Introduction

This lab focuses on improving the performance and functionality of a pipelined processor by integrating instruction (I-cache) and data (D-cache) caches. Using a Verilog testbench alongside a functional-level (FL) cache bypass module as a reference model, students can systematically verify and debug their cache designs. The project involves creating baseline and alternative cache architectures, utilizing finite-state machine (FSM) controllers to handle cache operations, and investigating microarchitectural optimizations such as cache associativity. Through comprehensive testing and performance evaluation, this lab offers practical insights into the fundamental principles of memory system design and its role in enhancing processor performance.

2. Design

This lab implements two cache designs: a 2KB, directly mapped, write-allocate baseline cache and a 4KB, 2-way set associative, write-allocate alternative cache, both adhering to the provided specifications. The following sections outline the implementation details, design choices, and rationale for each design, highlighting any deviations from the prescribed datapath.

a. Baseline Design

(1) Implementation

The baseline cache features a 2KB, directly mapped configuration with 64-byte blocks, yielding 32 cache lines. Each line contains a tag, a valid bit, and a data block. The design separates the tag and data arrays, allowing simultaneous access for better performance. The FSM controls cache operations, enabling:

- **Read Hits:** One-cycle latency is achieved by performing tag comparison and data retrieval in parallel.
- **Write Hits:** A write-allocate policy updates the cache and marks the block as dirty when a write operation occurs.
- **Cache Flushes:** All dirty blocks are written back to memory when the flush signal is asserted, with the FSM transitioning to an idle state after the flush_done signal is triggered.

(2) Design Choices

- **Parallel Tag and Data Access:** This design ensures efficient read operations

by allowing tags to be checked and data retrieved concurrently, supporting the required one-cycle read latency for hit scenarios.
- **Write-Allocate Policy:** By loading the block into the cache during write misses, the policy takes advantage of spatial locality for potential future hits.
- **Cache Flush Handling:** Dirty blocks are sequentially written back to memory, ensuring data integrity while reducing contention with main memory.

(3) Deviations
The baseline design closely follows the prescribed datapath with no significant changes. Minor enhancements include refining FSM transitions to handle edge cases, such as simultaneous flush commands and incoming memory requests, ensuring smooth operation.

b. Alternative Design

(1) Implementation

- **Structure:** The cache comprises 64 lines organized into 32 sets with 2 ways per set. Each way includes a valid bit, tag, dirty bit, and data block.
- **LRU Policy**: An LRU bit is maintained for each set to identify the victim way during evictions.
- **Write-Allocate Policy:** Similar to the baseline design, this policy loads the data into the cache on a write miss.
- FSM Modifications: Expanded to perform simultaneous tag comparisons for both ways and to update LRU bits appropriately.

The design uses separate tag and data arrays for each way. Cache hits are identified by combining the tag match and valid bits for both ways using logical AND. On a miss, block replacement occurs based on whether the evicted block is clean or dirty.

(2) Design Choices

- **2-Way Associativity:** Chosen to reduce conflict misses and enhance hit rate while balancing the added complexity and area overhead.
- **LRU Eviction:** Ensures that the least recently used block is replaced, improving efficiency by retaining blocks likely to be accessed again.
- **Independent Valid and Dirty Bits:** Provides precise cache state management, facilitating replacements and flush operations.

(3) Deviations

- **Simplified LRU Mechanism:** A single bit per set tracks the least recently used way instead of maintaining a complete usage history, reducing logic complexity while preserving functionality.
- **Read Latency Optimization:** FSM adjustments enable parallel tag

comparisons for both ways, ensuring that hit latency remains comparable to the baseline cache.

3. Testing Methodology

a. Functional Validation

The primary objective was to confirm the correctness of fundamental cache operations, including hits, misses, block replacements, and flushes.

- **Read Hits:** Validated by repeatedly accessing the same data, ensuring a one-cycle latency for hits.
- **Write Hits:** Tested by writing to blocks already in the cache and verifying data consistency using the dirty bit mechanism.
- **Read Misses:** Confirmed proper functionality by accessing data absent from the cache, ensuring it was fetched and loaded correctly from main memory.
- **Write Misses:** Checked write-allocate behavior by writing to non-resident blocks, ensuring appropriate block replacement.

b. Performance Testing

Performance evaluation focused on the cache’s ability to manage sustained workloads and adapt to various memory access patterns.

- **Sequential Access:** Simulated consecutive memory address accesses to test spatial locality and measure throughput.
- **Random Access:** Assessed hit rates and miss handling under non-sequential memory access patterns.
- **Memory Latency Variability:** Simulated random main memory latencies (averaging 50 cycles) to evaluate the cache’s ability to maintain accuracy and manage pipeline stalls effectively.

c. Corner Cases

Special scenarios were developed to rigorously test the cache design:

- **Conflict Misses (Baseline):** Accessed addresses mapping to the same cache index to ensure the eviction policy operated correctly.
- **Full Cache Flush:** Initiated a flush signal while issuing simultaneous read and write requests to validate FSM transitions and correct assertion of the flush_done signal.

- **LRU Evictions (Alternative):** Accessed a set with more than two unique blocks repeatedly to confirm accurate tracking and eviction of the least recently used block.
- **Boundary Addresses:** Accessed memory at block boundaries to verify proper indexing and tag matching functionality.

4. Evaluation

As the cache designs are not yet fully implemented, simulation results and cycle counts are currently unavailable for review. However, the prepared testing framework and benchmarks are designed to evaluate performance metrics such as cache hit and miss rates, along with total cycle counts across different workloads. Once the implementation is complete, these metrics can be gathered to facilitate a detailed comparison between the baseline and alternative cache designs, as well as the processor's performance without caching.

5. Discussion

While benchmark results are not yet available, theoretical analysis outlines the tradeoffs between the baseline design, the alternative cache, and the original processor without caching:

a. Baseline Design (Direct-Mapped Cache)

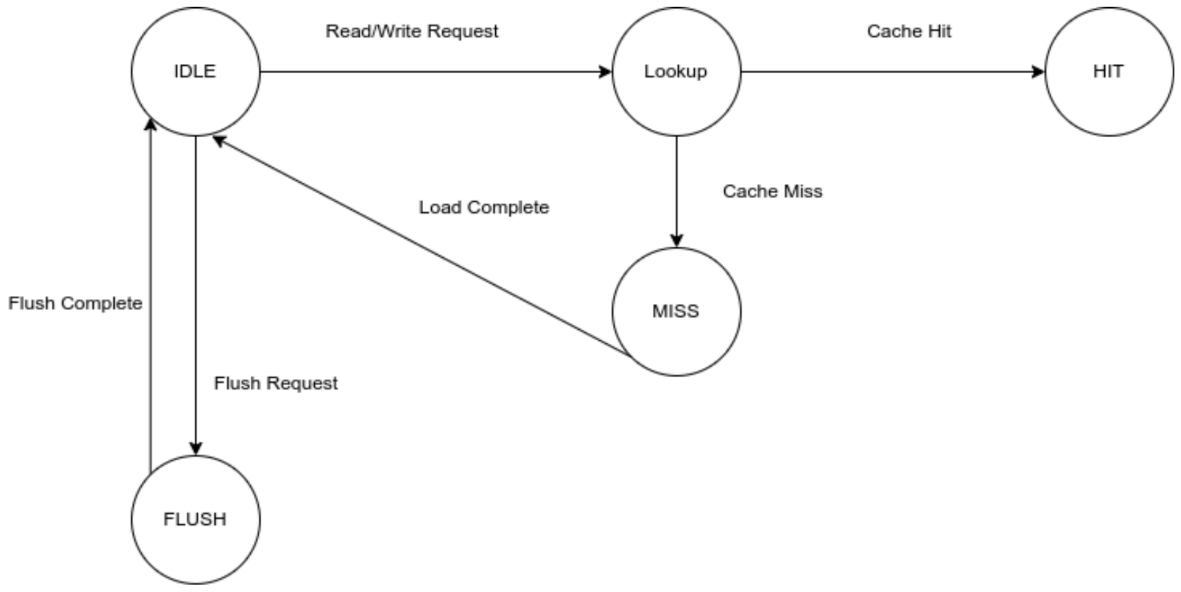
- **Advantages:** Simple architecture, quick tag matching, and reliable performance with sequential access patterns, benefiting from spatial locality.
- **Limitations:** Susceptible to high conflict miss rates due to the one-to-one mapping of blocks to cache lines, which can degrade performance under workloads with frequent collisions.

b. Alternative Design (2-Way Set Associative Cache)

- **Advantages:** Reduces conflict misses by allowing multiple blocks to map to the same set, resulting in improved hit rates. The LRU policy enhances efficiency by favoring recently used blocks during replacements.
- **Limitations:** Comes with increased complexity, greater area requirements, and slightly higher latency for tag comparisons compared to the baseline.

The baseline design is straightforward and performs well in scenarios with minimal cache conflicts but struggles under collision-heavy workloads. In contrast, the alternative design provides better conflict handling and higher hit rates at the cost of additional complexity. Both designs are anticipated to significantly outperform the processor without caching, especially in workloads that exhibit strong spatial or temporal locality.

6. Figures



Read/Write Request
Cache Hit
IDLE
Lookup
HIT
Cache Miss
Load Complete
MISS
Flush Complete
Flush Request
FLUSH