Module -5

Cache in ARM Embedded Systems

Cache: A small, fast array of memory placed between the processor core and main memory. It stores portions of recently referenced main memory to increase system performance by reducing the memory access bottleneck imposed on the processor core by slower main memory.

Write Buffer: A small FIFO memory placed between the processor core and main memory. It frees the processor core and cache from the slow write times associated with writing to main memory.

Functionality

- **Cache Memory**: It automatically handles the movement of code and data between the processor and main memory. Knowing the details of a processor's cache design can help create faster-running programs on a specific ARM core.
- **Write Buffer**: Helps manage the slow write times by queuing write operations, allowing the processor to continue executing without waiting for the write to complete.

Benefits of Using Cache

- **Increased Performance**: By storing frequently accessed data, the cache reduces the need for slower main memory access, thus speeding up the overall system performance.
- **Transparency**: Cache and write buffer hardware are designed to be invisible to software, meaning no code modifications are needed for existing software to run on a cached core.

Drawbacks of Using Cache

- **Execution Time Variability**: The main drawback is the difficulty in determining the exact execution time of a program. This is due to the random eviction process of cache memory, which may or may not contain the needed data at any given time, leading to variable execution times.
- **Cache Eviction**: Since cache memory is limited in size, it quickly fills up during program execution. The cache controller then frequently evicts existing code or data to make room for new data, which can lead to inconsistent performance.

Understanding the role and functioning of cache memory and write buffers in ARM systems is crucial for optimizing software performance. Despite the challenges in determining precise execution times due to the dynamic nature of cache memory, the benefits of significantly improved system performance make the use of caches a vital component in modern computing systems.

The Memory Hierarchy and Cache Memory

Memory Hierarchy

- **Memory Hierarchy**: Caches fit into the standard memory hierarchy between the processor and main memory. They utilize the principle of locality of reference, which states that programs tend to access the same set of memory locations repetitively over a short period. A structured arrangement of memory types in a computer system designed to balance speed and cost.
- **Register File**: The innermost level of memory, tightly coupled to the processor core, providing the fastest possible memory access.

Primary Level Memory Components

- **Tightly Coupled Memory (TCM)**: Memory closely connected to the processor core, providing fast access.
- **Level 1 (L1) Cache**: An array of high-speed, on-chip memory that temporarily holds frequently accessed code and data.
- **Write Buffer**: A small FIFO buffer that supports writes to main memory from the cache, freeing up the cache for other tasks.
- **Main Memory**: Includes volatile memory (SRAM, DRAM) and non-volatile memory (flash). It holds programs and data currently in use by the processor.

Secondary Storage

- **Secondary Storage**: Comprises large, slow, and inexpensive storage devices like disk drives and removable memory. It stores unused portions of large programs and data from peripheral devices.

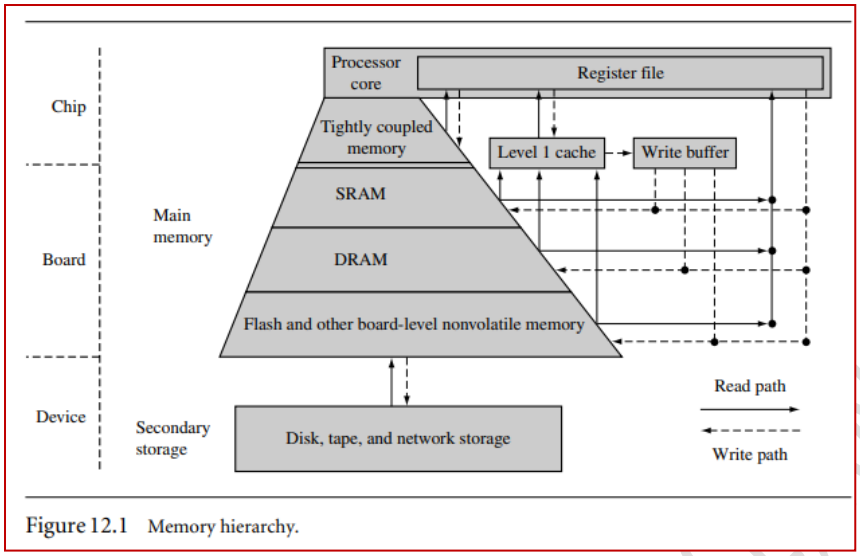


Figure 12.1 Memory hierarchy.

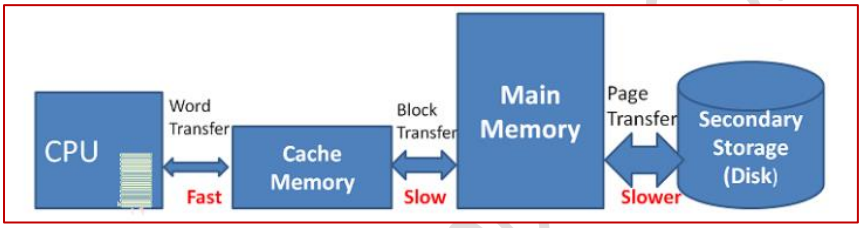


Figure: Memory hierarchy.

- **Cache Controller**: Manages the movement of data between the cache and main memory, including the eviction process. A cache temporarily moves data from a slower level of the hierarchy to a faster one to reduce access time.
- **Cache Lines**: The basic unit of data storage in a cache. When the processor requests data, an entire line (containing the requested data and its neighboring data) is loaded into the cache. Small blocks of data transferred between the slower main memory and the faster cache memory.
- **Level 2 (L2) Cache**: Located between the L1 cache and main memory, serving as secondary cache to further speed up data access.

Relationship Between Cache and Memory

- **Without Cache**: The processor core accesses main memory directly, resulting in slower data access.
- **With Cache**: The processor core accesses the faster cache memory, which holds copies of frequently accessed data from the slower main memory, resulting in quicker data retrieval.

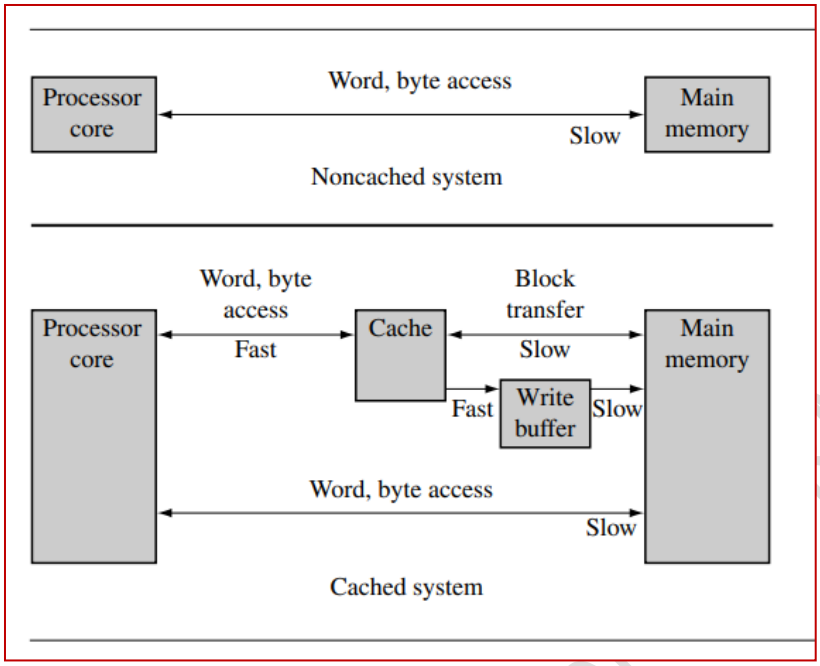
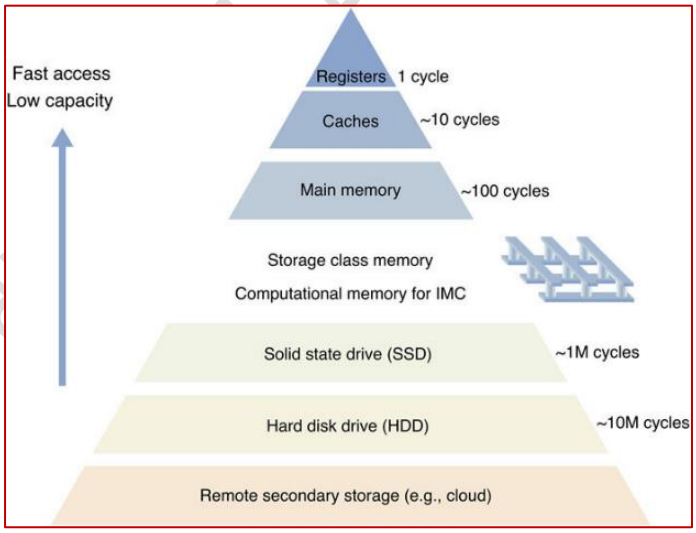


Figure: Relationship that a cache has between the processor core and main memory.

- **Noncached System**: Processor accesses slow main memory directly.
- **Cached System**: Processor accesses fast cache memory, which in turn accesses slow main memory.



Caches and Memory Management Units (MMUs)

- **Logical Cache (Virtual Cache)**: Located between the processor and the MMU, storing data in virtual address space. The processor accesses data without MMU translation.

- **Physical Cache**: Located between the MMU and main memory, storing data in physical address space. The MMU translates virtual addresses to physical addresses before accessing cache memory.
- **ARM Cached Cores**: ARM7 through ARM10 use logical caches; ARM11 uses physical caches.

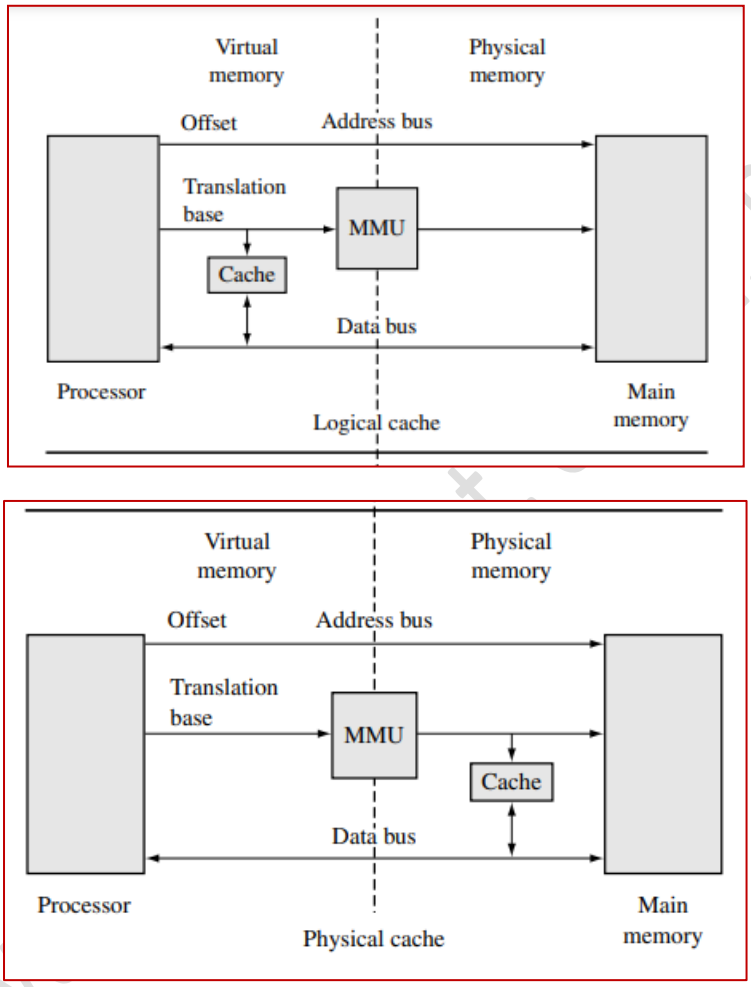


Figure: Logical and physical caches.

Principle of Locality of Reference

- **Locality of Reference**: There are two types:

1. **Temporal Locality**: Recently accessed data is likely to be accessed again soon. The tendency for programs to access the same memory locations repeatedly within short time intervals.
2. **Spatial Locality**: Data located close to recently accessed data is likely to be accessed soon. The tendency for programs to access memory locations that are close to each other within short intervals.

Performance Improvement: The principle of locality of reference explains why cache memory improves performance, as repeated access to cached data is much faster than accessing main memory.

- **Register File**: Integral to the processor core, providing the fastest memory access.
- **TCM vs. SRAM**: Same technology but differ in architectural placement—TCM is on-chip, while SRAM is on-board.
- **Cache Eviction**: The process of removing old data from the cache to make room for new data, which can affect execution time predictability.

Cache Architecture

1. Von Neumann vs. Harvard Architecture

- **Von Neumann Architecture:**
  - **Unified Cache:** A single cache used for both instructions and data.
  - **Memory Access:** Both instruction and data paths share the same cache memory, which simplifies the design but can lead to contention for cache access.

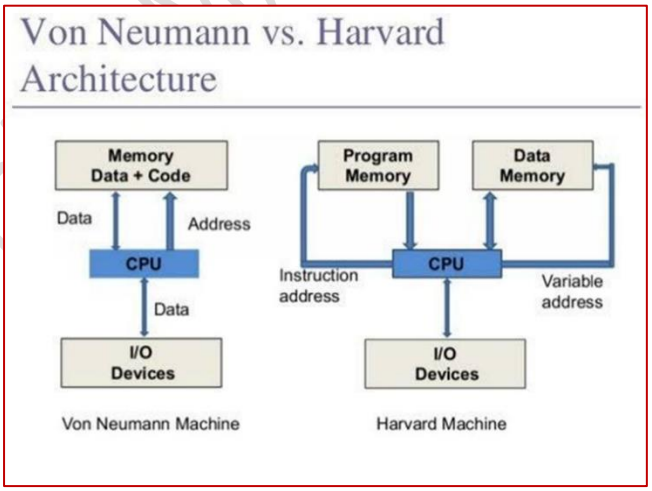


Figure: Von Neumann vs. Harvard Architecture.

- **Harvard Architecture:**
  - **Split Cache:** Separate caches for instructions (I-cache) and data (D-cache).

- **Memory Access:** Separate buses for instructions and data improve performance by allowing simultaneous access, but this requires more complex cache management.

A. Basic Architecture of Cache Memory

- **Main Components:**
  - **Cache Controller:** Manages the operation of the cache memory, including fetching data, checking cache hits/misses, and updating cache contents.
  - **Cache Memory:** Dedicated array accessed in units called cache lines.
- **Cache Memory Parts:**
  - **Directory Store (Cache-Tag):** Holds the address of the data's location in main memory.
  - **Data Section:** Stores the actual data read from main memory.
  - **Status Information:** Includes bits like the valid bit and dirty bit to manage the state of the cache lines.
- **Cache Lines:**
  - Each cache line consists of a cache-tag, status bits, and data words.
  - **Valid Bit:** Indicates if the cache line contains current, valid data.
  - **Dirty Bit:** Shows if the cache line data has been modified from its original value in main memory.

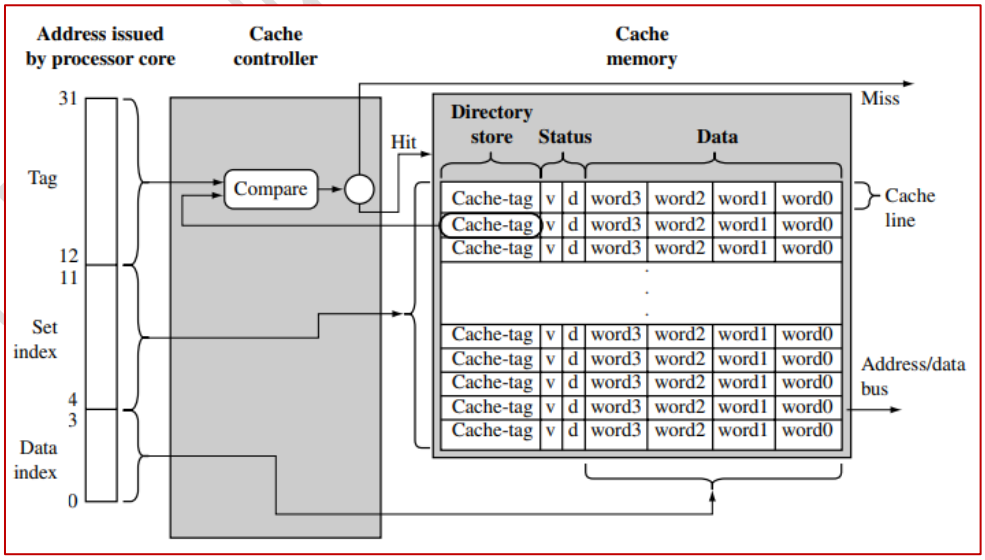


Figure: A 4 KB cache consisting of 256 cache lines of four 32-bit words.

3. Cache Memory Operations

- **Cache Hit:** When the requested data is found in the cache, leading to faster data access.
- **Cache Miss:** When the requested data is not in the cache, requiring a fetch from slower main memory.

4. Cache Structure

- A 4 KB cache with 256 cache lines, each containing four 32-bit words.
- **Addressing:** The processor issues addresses, which are split into tags and indices by the cache controller to locate the corresponding cache lines.
- **Address/Data Bus:** Connects the processor core with the cache memory.
- **Set Index and Data Index:** Used to locate the specific cache line and word within the cache.

B. Basic Operation of a Cache Controller

The cache controller is a hardware component that automatically manages the transfer of code and data between main memory and cache memory. This automation makes cache operations transparent to the software, allowing the same applications to run on systems with or without a cache.

1. **Intercepting Memory Requests:**
   - The cache controller intercepts memory read and write requests before they reach the memory controller.
   - It divides the address of each request into three fields: *the tag field, the set index field, and the data index field.*
2. **Address Breakdown:**
   - **Tag Field:** Identifies the specific block of memory.
   - **Set Index Field:** Locates the set or cache line within the cache memory.
   - **Data Index Field:** Points to the exact data within the cache line.

   Example: 1 million possible main memory locations map to one cache location.
3. **Locating the Cache Line:**
   - The controller uses the set index to find the corresponding cache line that might hold the requested data or instruction.
   - Each cache line includes a cache-tag and status bits.

4. **Cache Hit or Miss Determination:**
    - **Cache Hit:**
        - The controller checks the valid bit to ensure the cache line is active.
        - It then compares the cache-tag with the tag field of the requested address.
        - If both checks succeed, it confirms a cache hit.
    - **Cache Miss:**
        - If either check fails, it is a cache miss.
5. **Handling a Cache Miss:**
    - On a cache miss, the controller performs a **cache line fill** by copying an entire cache line from main memory to cache memory.
    - The requested data or code is then provided to the processor from the newly filled cache line.
6. **Handling a Cache Hit:**
    - On a cache hit, the controller uses the data index to select the specific code or data within the cache line.
    - It then supplies this data directly from the cache memory to the processor.

C. The Relationship Between Cache and Main Memory

- **Cache memory** is a smaller, faster type of volatile memory that provides high-speed data access to a processor.
- **Cache controller** manages the storage and retrieval of data in the cache.
- **Direct-mapped cache** is the simplest form of cache where each location in main memory maps to a single location in cache memory.

Direct-Mapped Cache

- **Main Memory and Cache Memory Relationship**:
    - Main memory is much larger than cache memory.
    - Multiple addresses in main memory map to the same single location in cache memory.
    - Example: Addresses ending in 0x824 map to the same cache location.

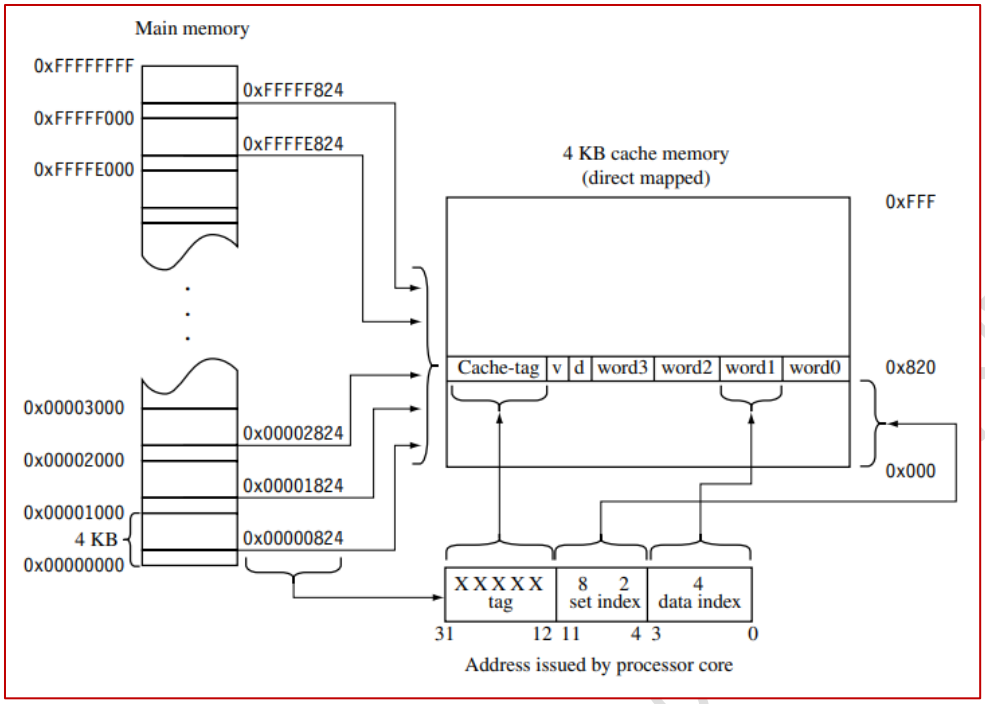


Figure: Illustrates how portions of main memory map to a direct-mapped cache.

Cache Line Fill and Data Streaming

- During a cache line fill, the cache controller may forward data to the core while copying it to the cache.
- This process is known as **data streaming**, allowing the processor to continue execution.

Eviction and Cache Miss

- **Eviction**: When valid data in a cache line represents a different address in main memory, it is replaced by new data.
- This process involves removing an existing cache line to make room for new data, which is crucial for servicing a cache miss.

Thrashing in Direct-Mapped Cache

- **Thrashing**: Occurs when two routines in a program map to the same cache line, leading to continuous eviction and loading.

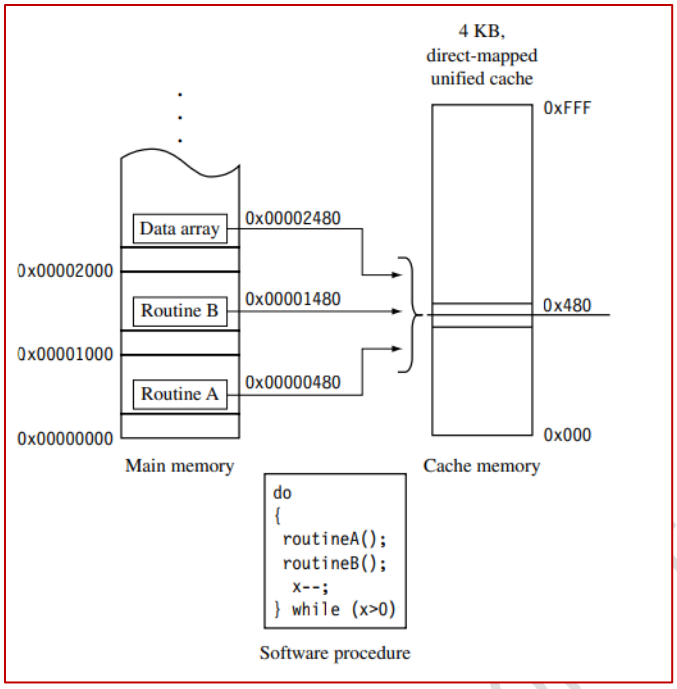


Figure Thrashing: two functions replacing each other in a direct-mapped cache.

- Example:
  - Routine A and Routine B have the same set index.
  - As Routine A executes, it loads into the cache.
  - When Routine B is called, it evicts Routine A, and vice versa, leading to repeated cache misses and evictions.

D. Set Associativity

Set Associativity: A cache design feature that reduces thrashing by dividing the cache into smaller equal units called ways.

- Example: A 4 KB cache is divided into four ways, each with 64 lines, making the cache a four-way set associative cache.

Cache Structure in Set Associativity

- **Set Index**: Addresses multiple cache lines, one in each way, instead of a single cache line.
- **Ways**: The cache is divided into several ways, each containing an equal number of cache lines.
  - Example: Four ways with 64 lines each in a 4 KB cache.

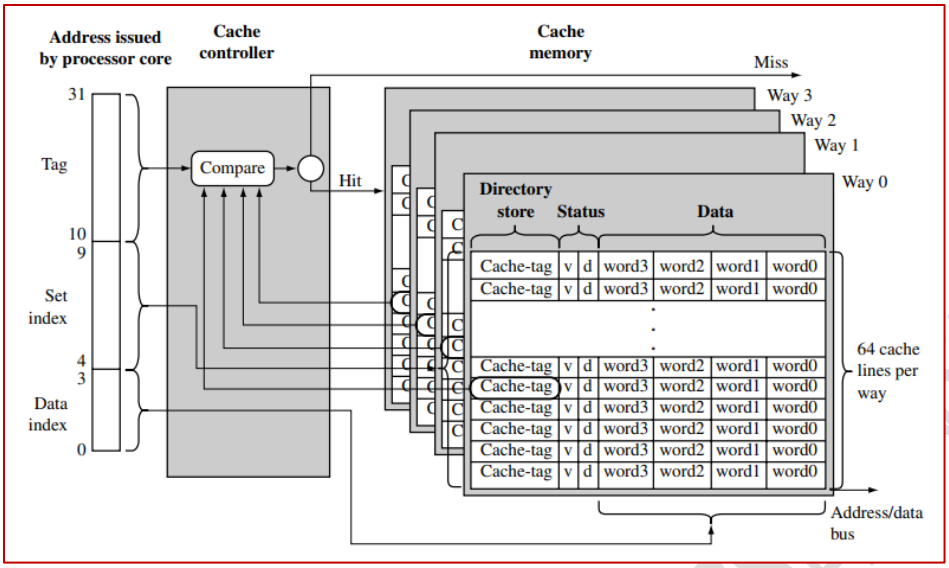


Figure: A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words.

Mapping in Set Associative Cache

- **Address Fields**:
    - **Tag**: Identifies the specific block of memory stored in the cache.
    - **Set Index**: Points to a set containing multiple cache lines.
    - **Data Index**: Specifies the exact byte/word in the cache line.
- A single memory location can map to any of the cache lines in a set, allowing more flexibility and reducing thrashing.

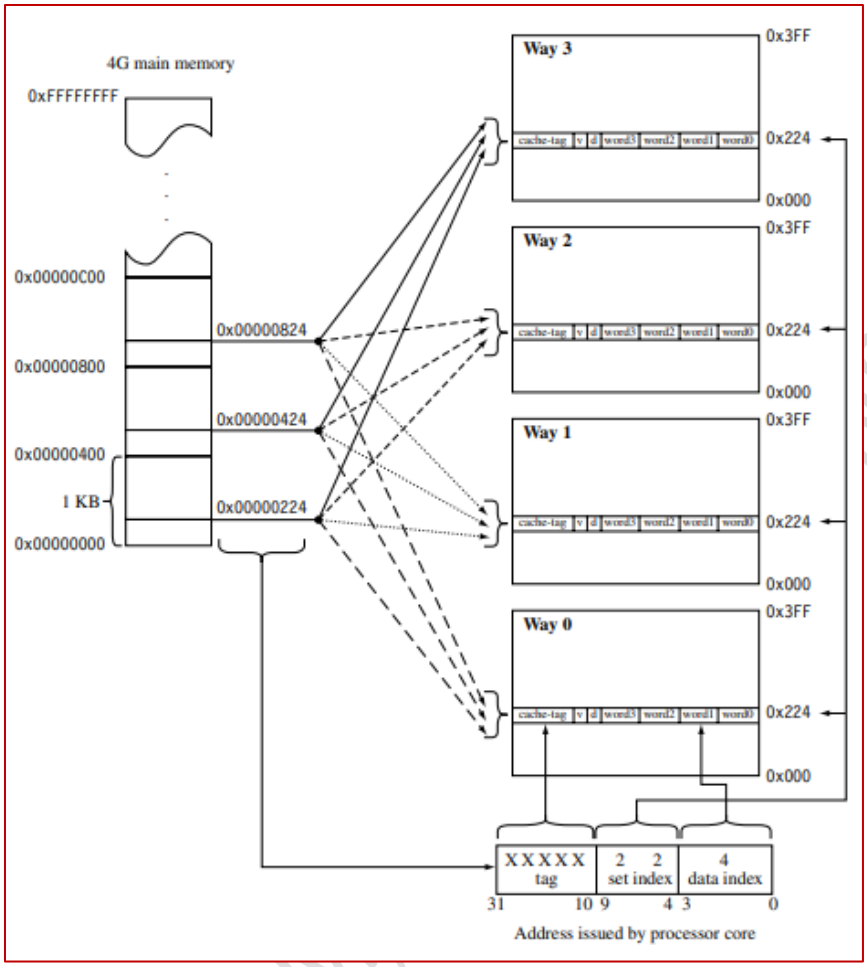


Figure: Main memory mapping to a four-way set associative cache

Benefits of Set Associativity

- Reduces thrashing by providing multiple locations in the cache for a single memory address.
- Example: Routine A, routine B, and a data array can coexist in different ways within the same set, reducing conflicts.

Changes in Address Mapping

- **Bit Fields**:
  - The tag field becomes larger, and the set index field becomes smaller.
  - More main memory addresses map to the same set of cache lines.

- **Memory Mapping**:
    - Any single location in main memory now maps to four different locations in the cache.

Increasing Set Associativity

- **Higher Associativity**: Decreases the probability of thrashing but increases hardware complexity.
- **Fully Associative Cache**: Any main memory location can map to any cache line, but requires complex hardware.

Content Addressable Memory (CAM)

- **CAM**: Used to increase set associativity by allowing simultaneous comparison of many cache-tags.
- **CAM Functionality**:
    - Takes an address tag as input.
    - Produces an address if the input tag matches any stored cache-tag.
- **Example in ARM Processors**: ARM920T and ARM940T use 64-way set associative caches with CAMs to manage cache-tags.

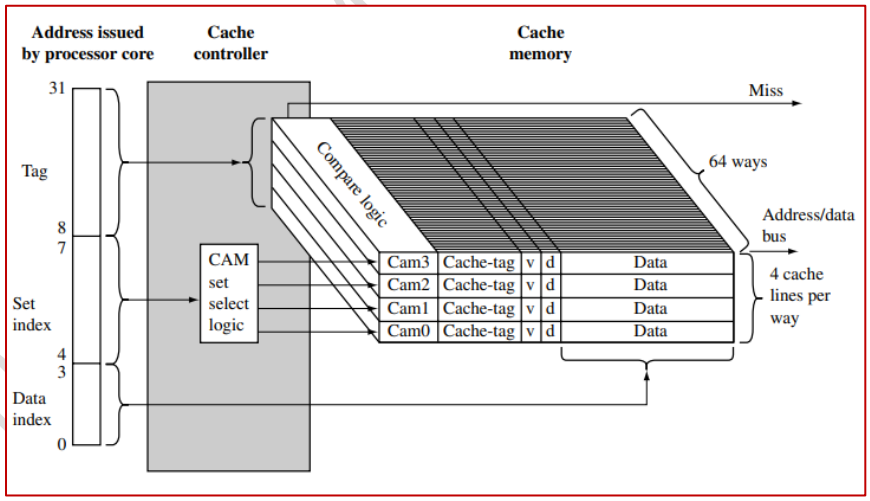


Figure: ARM940T—4 KB 64-way set associative D-cache using a CAM.

- **Structure**:
    - 64-way set associative cache.
    - Four CAMs used to compare input tags with stored cache-tags.

- Set index bits enable one of the CAMs, which then selects a cache line.
- Data index selects the specific byte/word within the cache line.

- **Function**:
  - Tag portion of the address is used as input to CAMs.
  - CAMs compare the input tag with cache-tags.
  - If a match is found, cache data is provided; otherwise, a miss signal is generated.

- Set associativity is a critical feature in cache design that balances performance and complexity.
- It reduces thrashing by allowing multiple cache lines for a single memory address.
- Higher set associativity, while reducing thrashing, requires more complex hardware, often implemented using CAM.

Write Buffers

A small, fast FIFO (First In, First Out) memory buffer that temporarily holds data the processor would normally write to main memory.

- **Purpose**: To improve the efficiency of writing data to main memory by allowing the processor to continue executing instructions while write operations are handled in the background.

Functionality of Write Buffers

- **Without Write Buffer**: Processor writes data directly to main memory, which can be slow.
- **With Write Buffer**: Data is written quickly to the write buffer and then gradually transferred to slower main memory.
- **Memory Hierarchy**: Write buffers are at the same level as the L1 cache in the memory hierarchy.

Efficiency of Write Buffers

- **Performance Dependence**: The efficiency depends on the ratio of main memory writes to the number of instructions executed.
  - Low or spaced-out memory writes: Write buffer rarely fills, allowing the program to continue executing using cache and registers.

- High-frequency writes: Write buffer may fill up, potentially causing delays.

Cache Performance Improvement

- **During Cache Line Evictions**:
  - **Dirty Cache Line**: When evicted, it is written to the write buffer instead of directly to main memory.
  - **Benefit**: The new cache line data becomes available sooner, allowing the processor to continue operating from cache memory.

Data Availability

- Data in the write buffer is not available for reading until it has been written to main memory.
- **Evicted Cache Line**: Cannot be read while it is in the write buffer.

Write Buffer Characteristics

- **FIFO Depth**: Usually quite small, holding only a few cache lines.
- **Write Coalescing**: Some write buffers, such as those in the ARM10 family, support coalescing (also known as write merging, write collapsing, or write combining).
  - **Coalescing**: Merging of multiple write operations into a single cache line if they represent the same data block in main memory.
  - **Benefit**: Reduces the number of write operations to main memory, improving efficiency.

Measuring Cache Efficiency

1. **Cache Hit Rate**:
   - The proportion of memory requests that are successfully found in the cache.

$$\text{Hit Rate} = \left(\frac{\text{Cache Misses}}{\text{Memory Requests}}\right) \times 100$$

2. **Cache Miss Rate**:
   - **Relation to Hit Rate**: $\text{Miss Rate} = 100 - \text{Hit Rate}$

Measuring Cache Efficiency

- **Memory Requests**: Total number of requests made to memory (includes both reads and writes).
- **Cache Hits**: The number of memory requests that were successfully retrieved from the cache.
- **Cache Misses**: The number of memory requests that were not found in the cache and had to be fetched from main memory.

Types of Hit and Miss Rates

- **Read Hit Rate**: Percentage of read requests that were found in the cache.
- **Write Hit Rate**: Percentage of write requests that were found in the cache.
- **Overall Hit and Miss Rates**: Can be calculated for both reads and writes combined.

Performance Metrics

1. **Hit Time**:
    - The time it takes to access a memory location in the cache.
    - **Impact**: A lower hit time means faster data retrieval from the cache.
2. **Miss Penalty**:
    - The time it takes to load a cache line from main memory into the cache when a cache miss occurs.
    - **Impact**: A higher miss penalty can significantly slow down overall system performance, as fetching data from main memory is slower than from the cache.

- **Cache Hit Rate** and **Cache Miss Rate** are primary metrics used to evaluate cache efficiency.
- These rates help determine how effectively the cache is being utilized by a program.
- **Hit Time** and **Miss Penalty** provide additional insights into the performance impact of cache hits and misses.
- Understanding these metrics is crucial for optimizing cache performance and overall system efficiency.

Cache Policy

Cache policies determine how a cache operates and manages data. The three main policies are:

1. **Write Policy**
2. **Replacement Policy**
3. **Allocation Policy**

Write Policy

The write policy determines how and where data is written during processor write operations. There are two main types of write policies: writethrough and writeback.

1. **Writethrough**
    - **Definition**: Data is written to both the cache and main memory simultaneously.
    - **Advantage**: Ensures cache and main memory coherence.
    - **Disadvantage**: Slower performance due to the need to write data to main memory for every cache write.
2. **Writeback**
    - **Definition**: Data is written only to the cache. Main memory is updated only when the cache line is evicted.
    - **Advantage**: Faster performance as writes to main memory are less frequent.
    - **Disadvantage**: Cache and main memory can become temporarily out of sync.
    - **Dirty Bit**: A status bit in the cache line that indicates if the cache line has been modified. If the cache line is evicted and the dirty bit is set, the cache line must be written to main memory.

Cache Line Replacement Policies

On a cache miss, the cache controller needs to select a cache line to replace. This process is called eviction, and the selected cache line is known as the victim. The strategy for selecting the victim cache line is determined by the replacement policy.

1. **Round-robin Replacement**
    - **Definition**: Selects the next cache line in a set sequentially for replacement.
    - **Mechanism**: Uses a victim counter that increments sequentially with each cache line allocation. When the counter reaches a maximum value, it resets to a base value.
    - **Advantage**: Predictable and simple to implement.

- **Disadvantage**: Performance can vary significantly with small changes in memory access patterns.

2. **Pseudorandom Replacement**
    - **Definition**: Selects the next cache line in a set randomly for replacement.
    - **Mechanism**: Uses a non-sequential incrementing victim counter that increments by a randomly selected value. When the counter reaches a maximum value, it resets to a base value.
    - **Advantage**: Less predictable and can handle varied access patterns better.
    - **Disadvantage**: More complex to implement compared to round-robin.

Allocation Policy

The allocation policy determines when a cache line is allocated.

Example: Measuring Cache Replacement Policies

This example demonstrates how the time to execute a software routine differs between round-robin and random cache replacement policies. The test uses the ARM940T processor and measures the execution time using the clock function from the C library.

Code Explanation

1. **Function cache_RRtest**
    - **Purpose**: Runs timing tests for both round-robin and random replacement policies.
    - **Process**:
        1. Enables the round-robin policy.
        2. Cleans and flushes the cache.
        3. Measures and prints the execution time for the test routine with round-robin policy.
        4. Enables the random policy.
        5. Cleans and flushes the cache.

6. Measures and prints the execution time for the test routine with random policy.

```
#include <stdio.h>
#include <time.h>

void cache_RRtest(int times, int numset)
{
clock_t count;
printf("Round Robin test size = %d\r\n", numset);

enableRoundRobin();
cleanFlushCache();

count = clock();
readSet(times, numset);
count = clock() - count;
printf("Round Robin enabled = %.2f seconds\r\n",
(float)count/CLOCKS_PER_SEC);

enableRandom();
cleanFlushCache();

count = clock();
readSet(times, numset);
count = clock() - count;
printf("Random enabled = %.2f seconds\r\n\r\n",
(float)count/CLOCKS_PER_SEC);
}
```

Function readSet

- **Purpose**: Fills a cache set with values.
- **Arguments**:
  - times: Number of times to run the test loop.
  - numset: Number of set values to read, determining the number of cache lines to load.
- **Process**:
  - Uses an assembly loop to read a value from memory and increment the address by 64 bytes (16 words) in each iteration.
  - If numset is 64, it fills all cache lines in a set on the ARM940T.

```
int readSet(int times, int numset)
{
    int setcount, value;
    volatile int *newstart;
    volatile int *start = (int *)0x20000;

    __asm {
        timesloop:
        MOV newstart, start
        MOV setcount, numset
        setloop:
        LDR value, [newstart, #0];
        ADD newstart, newstart, #0x40;
        SUBS setcount, setcount, #1;
        BNE setloop;
        SUBS times, times, #1;
        BNE timesloop;
    }
    return value;
}
```

Test Cases

- Runs the test routine for two scenarios:
    1. **numset = 64**: Fills all available cache lines in a set.
    2. **numset = 65**: Attempts to fill the set with 65 entries, causing an overflow.

```
unsigned int times = 0x10000;
unsigned int numset = 64;
cache_RRtest(times, numset);

numset = 65;
cache_RRtest(times, numset);
```

Results

- **Console Output**:

- Round-robin and random policies perform similarly when numset is 64.
- Significant difference in performance when numset is 65, showing the impact of the round-robin policy.

```
Round Robin test size = 64
Round Robin enabled = 0.51 seconds
Random enabled = 0.51 seconds
Round Robin test size = 65
Round Robin enabled = 2.56 seconds
Random enabled = 0.58 seconds
```

Allocation Policy on a Cache Miss

Allocation Policies

1. **Read-Allocate on Cache Miss**
   - **Description**: Allocates a cache line only during a read from main memory.
   - **Behavior**:
     - If the victim cache line contains valid data, it is written to main memory before the cache line is filled with new data.
     - Write operations update the cache if the data is already in the cache and may update main memory if using a writethrough policy.
     - If the data is not in the cache, the controller writes directly to main memory without updating the cache.
   - **Supported by**: ARM7, ARM9, ARM10 cores.
2. **Read-Write Allocate on Cache Miss**
   - **Description**: Allocates a cache line for both read and write operations.
   - **Behavior**:
     - On a read miss, it follows the read-allocate policy.
     - On a write miss, it allocates a cache line, writes back the valid victim cache line to main memory (if necessary), and fills the new cache line with data from main memory.
     - After filling, the controller writes the new data to the corresponding location in the cache line and updates main memory if using a writethrough policy.

- **Supported by**: Intel XScale (supports both policies).

Core	Write Policy	Replacement Policy	Allocation Policy
ARM720T	Writethrough	Random	Read-allocate
ARM740T	Writethrough	Random	Read-allocate
ARM920T	Writethrough, Writeback	Random, Round-robin	Read-allocate
ARM940T	Writethrough, Writeback	Random	Read-allocate
ARM926EJS	Writethrough, Writeback	Random, Round-robin	Read-allocate
ARM946E	Writethrough, Writeback	Random, Round-robin	Read-allocate
ARM10202E	Writethrough, Writeback	Random, Round-robin	Read-allocate
ARM1026EJS	Writethrough, Writeback	Random, Round-robin	Read-allocate
Intel StrongARM	Writeback	Round-robin	Read-allocate
Intel XScale	Writethrough, Writeback	Round-robin	Read-allocate, Write-allocate

Table: ARM cached core policies.

• Read-**Allocate**:

- Allocates on reads only.
- Writes update cache if data is present; otherwise, write directly to main memory.

• Read-**Write Allocate**:

- Allocates on both reads and writes.
- More flexible and efficient for write-heavy workloads.

• Cache **Line Allocation**:

- Ensures efficient data retrieval and storage, impacting overall system performance.

Coprocessor 15 and Caches

Coprocessor 15 Registers for Cache Control

Coprocessor 15 (CP15) registers are used to configure and control ARM cached cores. The primary registers involved in cache operations are c7 and c9. Here's a breakdown of their roles and functionalities:

Key CP15 Registers

1. **CP15**

(Primary Cache Control Register)

- **Role**: Controls the setup and operation of the cache.

- **Function**: Includes write-only commands to clean and flush the cache.

2. **CP15**

(Victim Pointer Register)

- **Role**: Defines the base address for the victim pointer.
- **Function**: Determines the number of lines of code or data that are locked in the cache.

Usage of CP15 Registers

- **Cache Configuration**: CP15 registers c7 and c9 are essential for configuring cache settings and operations.
- **Cleaning and Flushing Cache**: CP15

has specific commands to clean (remove data) and flush (clear the cache entirely) the cache, ensuring that all modified data is written back to main memory.

- **Locking Cache Lines**: CP15

is used to lock specific lines of code or data in the cache to optimize performance and ensure critical data remains in the cache.

Practical Applications

- **Example Routines**: The following sections provide example routines using CP15 registers to perform tasks such as cleaning and flushing caches and locking data in the cache. These routines are typically called by the control system as part of memory management activities.

Module -5: Questions

1. Define the term 'cache' in the context of ARM embedded systems. What are its primary functions?
2. Explain the concept of 'write buffer' and its role in ARM processors. How does it improve system performance?
3. List and describe the main benefits and drawbacks of using cache memory in ARM systems.
4. Differentiate between a 'logical cache' and a 'physical cache' in ARM architecture. Provide examples of ARM processors that use each type.
5. What is 'cache eviction'? Describe the process and its impact on program execution time.
6. Discuss the principle of locality of reference and its importance in the design of cache memory.
7. Explain the difference between Von Neumann and Harvard architecture in the context of cache memory.
8. Describe the structure and operation of a basic cache controller in an ARM processor. Include the concepts of cache hits and cache misses.
9. What is 'set associativity' in cache design? How does it help in reducing thrashing?
10. Explain the role and functionality of a write buffer in ARM processors. How does it handle high-frequency writes?
11. Describe the terms 'cache hit rate' and 'cache miss rate'. How are they calculated, and why are they important metrics for evaluating cache performance?
12. Discuss the write policies (writethrough and writeback) used in cache memory. Compare their advantages and disadvantages.
13. Explain the cache replacement policies (round-robin, pseudorandom, etc.). How do they influence the performance of a cached system?
14. Describe the concept of 'write coalescing' in the context of write buffers. How does it improve system performance?