Chapter 12
Caches and Caching

Comer, D. (2017). *Essentials of Computer Architecture* (2nd ed.). CRC Press.

Mano, M. M., Kim, C. R. & Martin, T. (2015). *Logic and Computer Design Fundamentals* (5th ed.). Pearson.

Contents

Contents

1. Definition of Caching

- *Caching* refers to an important optimization technique used to improve the performance of any hardware or software system that **retrieves information**.
- A cache acts as an intermediary.
  - That is, a cache is placed on the path between a mechanism that makes requests and a mechanism that answers requests, and the cache is configured to intercept and handle all requests.
- The central idea in caching is high-speed, temporary storage: the cache keeps a local copy of selected data, and answers requests from the local copy whenever possible. Performance improvement arises because the cache is designed to return answers faster than the mechanism that normally fulfills requests.

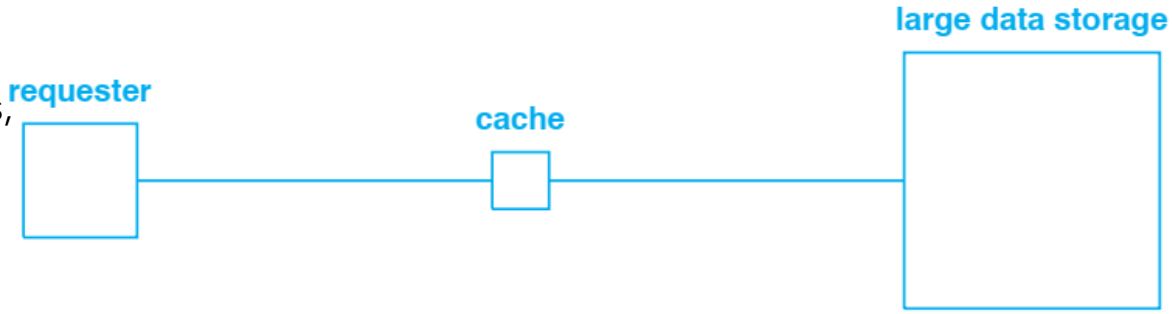


Figure 12.1 Conceptual organization of a cache, which is positioned on the path between a mechanism that makes requests and a storage mechanism that answers requests.

2. Characteristics of a Cache

- General characteristics of a cache:
  - Small
  - Active
  - Transparent
  - Automatic

2. Characteristics of a Cache

- Small
  - To keep economic cost low.
  - The amount of storage associated with a cache is much smaller than the amount of storage. (< 10%)
  - One of the central design issues revolves around the selection of data items to keep in the cache.
- Active
  - An active mechanism examines each request and decides how to respond.
  - Activities include: checking to see if a requested item is available in the cache, retrieving a copy of an item from the data store if the item is not available locally, and deciding which items to keep in the cache.

2. Characteristics of a Cache

- Transparent
  - A cache can be inserted without making changes to the requester or data store.
    - That is, the interface the cache presents to the requester is exactly the same as the interface a data store presents, and the interface the cache presents to the data store is exactly the same as the interface a requester presents.
- Automatic
  - A cache implements an algorithm that examines the sequence of requests, and uses the requests to determine how to manage the cache.

3. Cache Terminology

- *Cache hit* (*hit*)
  - A request that can be satisfied by the cache without access to the underlying data store.
- *Cache miss* (*miss*)
  - A request that cannot be satisfied by the cache.
- A sequence of references exhibits *high locality of reference* if the sequence contains repetitions of the same requests; otherwise, we say that the sequence has *low locality of reference*.
  - High locality of reference leads to higher performance.
  - Locality refers to items in the cache.

4. Best and Worst Case Cache Performance

- If a data item is stored in the cache, the cache mechanism can return the item faster than the data store.
- C_h is the cost if an item is found in the cache (i.e., a hit), and C_m is the cost if an item is not found in the cache (i.e., a miss).

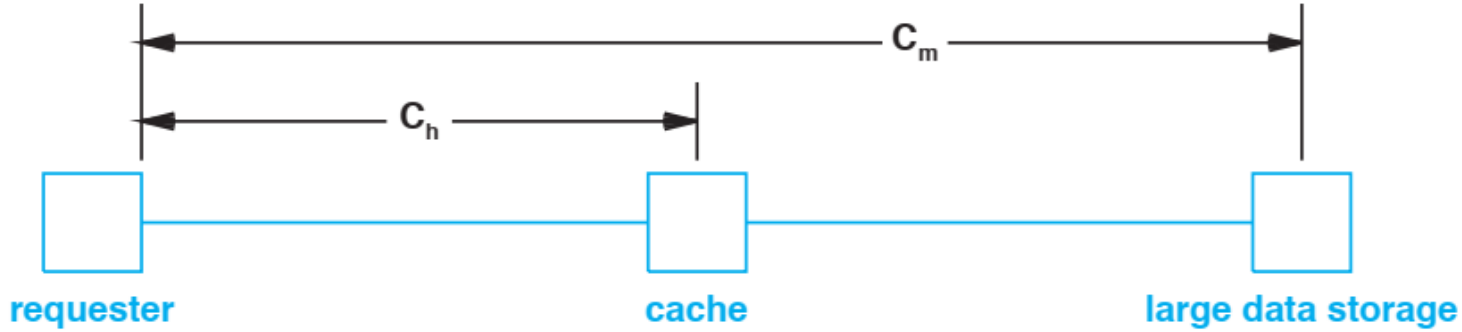


Figure 12.2 Illustration of access costs when using a cache. Costs are measured with respect to the requester.

4. Best and Worst Case Cache Performance

- Examine the performance on a sequence of *N* requests.
  - The worst case:
    - Each request references a new item.
    - $C_{worst} = NC_m$
    - The average cost per request is C_m.
  - The best case:
    - All requests in the sequence specify the same data item.
    - $C_{best} = C_m + (N-1)C_h$
    - The average cost per request is $C_{per\_request} = \frac{C_m+(N-1)C_h}{N} = \frac{C_m}{N} - \frac{C_h}{N} + C_h$
    - As $N \rightarrow \infty$, the first two terms approach zero, which means that the cost per request in the best case becomes C_h.

5. Cache Performance on a Typical Sequence

- Hit ratio *r*
  - $r = \frac{number\ of\ request\ that\ are\ hits}{total\ number\ of\ requests}$
- The cost of access
  - $Cost = rC_h + (1 - r)C_m$
  - C_m is fixed. Thus, there are two ways a cache designer can improve the performance of a cache: **increase the hit ratio or decrease the cost of a hit.**

6. Cache Replacement Policy

- To increase the hit ratio
  - Increase the cache size
  - Improve the replacement policy
- *Increase the cache size*
  - When it begins, a cache keeps a copy of each response. Once the cache storage is full, an item must be removed from the cache before a new item can be added. A larger cache can store more items.
- *Improve the replacement policy*
  - A cache uses a replacement policy to decide which item to remove when a new item is encountered and the cache is full.

LRU Replacement

- What replacement policy should be used? There are two issues.
  - First, to increase the hit ratio, the replacement policy should retain those items that will be referenced most frequently.
  - Second, the replacement policy should be inexpensive to implement, especially for a memory cache.
- Least Recently Used (LRU), the policy
  - Replacing the item that was referenced the longest time in the past.
  - When an item is referenced, the item moves to the front of the list; when replacement is needed, the item at the back of the list is removed.

7. Multilevel Cache Hierarchy

- The cost to access the new cache is lower than the cost to access the original cache.
  - A set of Web caches provides an example of a multilevel hierarchy. The path between a browser running on a user's computer can pass through a cache at the user's ISP as well as the local cache mechanism used by the browser.
  - The cost equation

$$Cost = r_1 C_{h1} + r_2 C_{h2} + (1 - r_1 - r_2) C_m$$

Figure 12.3 The organization of a system with an additional cache inserted.

7. Multilevel Cache Hierarchy

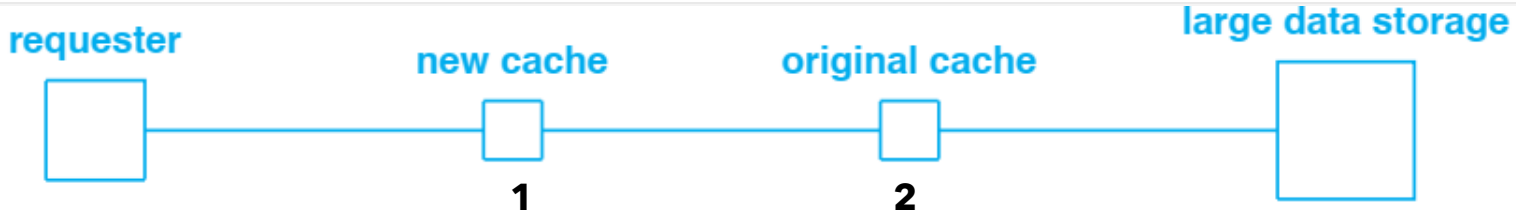


Figure 12.3 The organization of a system with an additional cache inserted.

- r_1 denotes the fraction of hits for the new cache.
- r_2 denotes the fraction of hits for the original cache.
- C_{h1} denotes the cost of accessing the new cache
- C_{h2} denotes the cost of accessing the original cache.

$$Cost = r_1 C_{h1} + r_2 C_{h2} + (1 - r_1 - r_2) C_m$$

8. Preloading Caches

- Cache designers observe that although many cache systems perform well in the steady state (i.e., after the system has run for awhile)
- The system exhibits higher cost during startup.
  - That is, the initial hit ratio is extremely low because the cache must fetch items from the data store.
  - In some cases, the startup costs can be lowered by *preloading* the cache. That is, values are loaded into the cache before execution begins.

9. Physical Memory Cache

- When the cache encounters a ***fetch*** request
  - The cache hardware performs two tasks in parallel: the cache simultaneously passes the request to the physical memory and searches for an answer locally.
  - If it finds an answer locally, the cache must cancel the memory operation.
  - If it does not find an answer locally, the cache must wait for the underlying memory operation to complete.
  - Furthermore, when an answer does arrive from memory, the cache uses parallelism again by simultaneously saving a local copy of the answer and transferring the answer back to the processor.
- Parallel activities make the hardware complex.

10. Write Through and Write Back

- Write-through cache
  - To handle *write* operations
    - The cache kept a copy and forwarded the *write* operation to the underlying memory.
- Write-back cache
  - A write-back cache keeps an extra bit with each item that is known as the *dirty bit*.
    - In a physical memory cache, a dirty bit is associated with each block in the cache.
  - When an item is **fetched** and a copy is placed in the cache, the dirty bit is initialized to **zero**.
  - When the processor **modifies** the item (i.e., performs a **write**), the dirty bit is set to **one**.
  - When it needs to eject a block from the cache, the hardware first examines the dirty bit associated with the block. If the dirty bit is one, a copy of the block is written to memory.
- A write-back cache achieves higher performance than a write-through cache.

11. Cache Coherence

- Memory caches are especially complex in a system with multiple processors (e.g., a multicore CPU).
- Consider what happens if the two caches use a write-back approach.
  - When processor 1 writes to a memory location X, cache 1 holds the value for X. Eventually, when it needs space, cache 1 writes the value to the underlying physical memory.
  - Similarly, whenever processor 2 writes to a memory location, the value will be placed in cache 2 until space is needed.
  - The problem should be obvious: without an additional mechanism, incorrect results will occur if both processors simultaneously issue read and write operations for a given address.

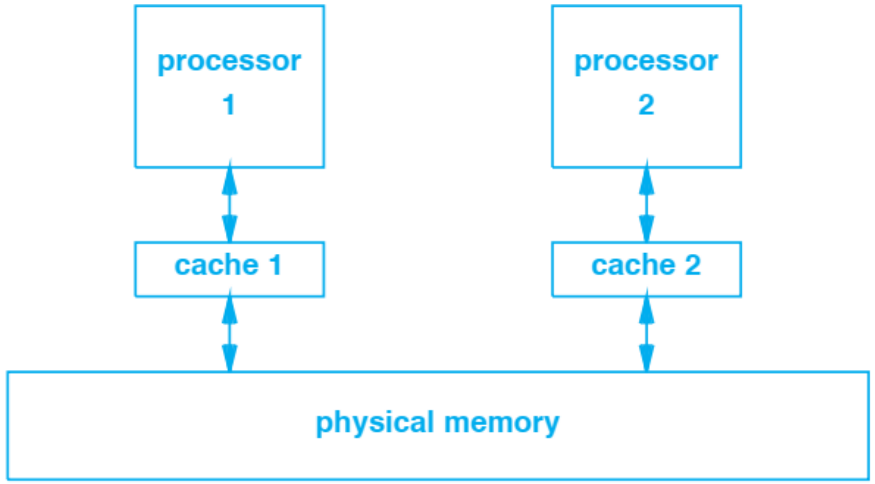


Figure 12.4 Illustration of two processors sharing an underlying memory. Because each processor has a separate cache, conflicts can occur if both processors reference the same memory address.

11. Cache Coherence

- To avoid conflicts, all caches that access a given memory must follow a ***cache coherence protocol*** that coordinates the values.
  - When processor 2 reads from an address, A, the coherence protocol requires cache 2 to inform cache 1.
  - If cache 1 currently holds address A, cache 1 writes A to the physical memory so cache 2 can obtain the most recent value. That is, a ***read*** operation on any processor triggers a **write-back** in any cache that currently holds a cached copy of the address.
  - Similarly, if any processor issues a ***write*** operation for an address, A, all other caches must be informed to discard cached values of A.
  - Cache coherency introduces **additional delay**.

12. L1, L2, and L3 Caches

- Traditional terminology for multilevel cache hierarchy
  - *Level 1 cache (L1 cache):* to refer to the cache onboard the processor chip.
  - *Level 2 cache (L2 cache):* to refer to an external cache.
  - *Level 3 cache (L3 cache):* to refer to a cache built into the physical memory.
- That is, an L1 cache was originally on-chip and an L2 or L3 cache was off-chip.

12. L1, L2, and L3 Caches

- Recent terminology
  - *L1 cache:* To describe a cache that is associated with one particular core.
  - *L2 cache:* To describe an on-chip cache that may be shared.
  - *L3 cache:* To describe an on-chip cache that is shared by multiple cores.
- Typically, all cores share an L3 cache. Thus, the distinction between on-chip and off-chip has faded.

12. L1, L2, and L3 Caches

- Sizes of L1, L2, and L3 caches
  - The cache at the top of the hierarchy is the fastest, but also the smallest.
  - The L1 cache may be divided into separate instruction and data caches

Cache	Size	Notes
L1	64 KB to 96 KB	Per core
L2	256 KB to 2 MB	May be per core
L3	8 MB to 24 MB	Shared among all cores

Figure 12.5 Example cache sizes in 2016. Although absolute sizes continue to change; readers should focus on the amount of cache relative to RAM that is 4 GB to 32 GB.

13. Modified Harvard Architecture

- When both data and instructions are placed in the same cache, data references tend to push instructions out of the cache, lowering performance.
  - Adding a separate instruction cache will improve performance.
- If a cache becomes sufficiently large, intermixing instructions and data references will work fine.
- Separating instruction and data caches is trivial in a Harvard Architecture because an *I-cache* (*instruction cache*) is associated with the instruction memory and a *D-cache* (*data cache*) is associated with the data memory.
- Should architects abandon the Von Neumann Architecture?

13. Modified Harvard Architecture

- *Modified Harvard Architecture*
  - A computer has separate instruction and data caches, but the caches lead to a single memory.

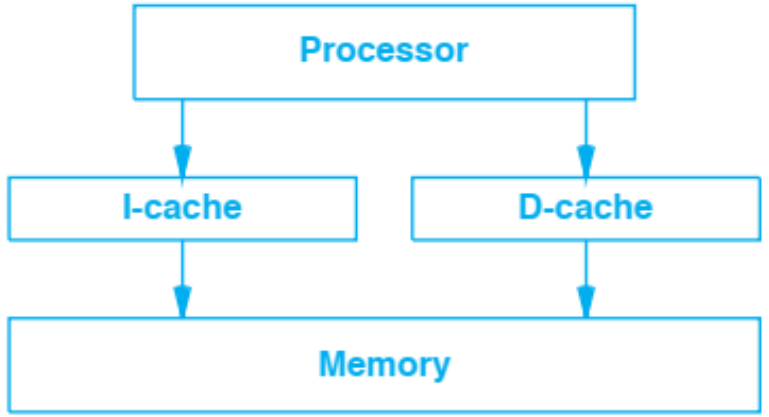


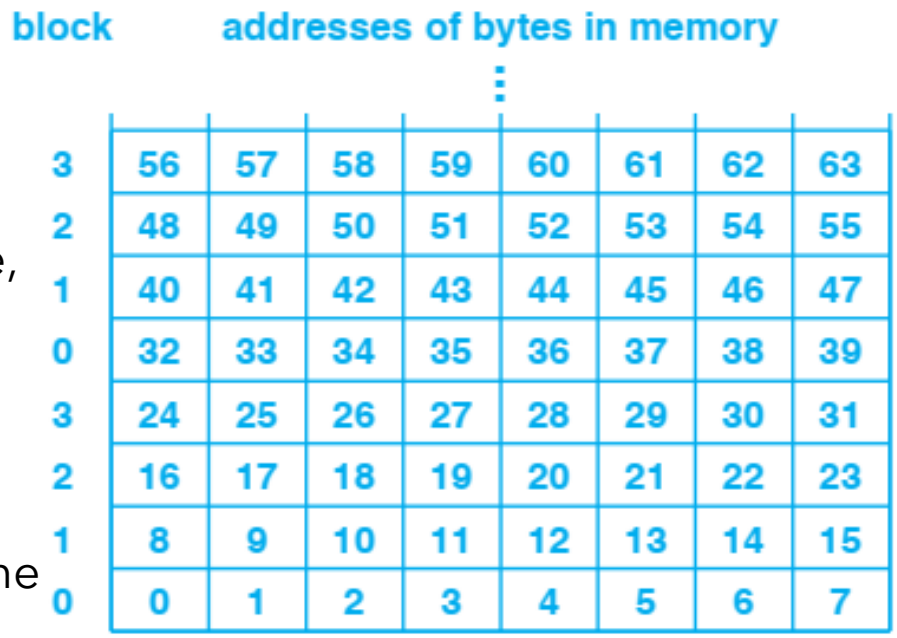
Figure 12.6 A Modified Harvard Architecture with separate instruction and data caches leading to the same underlying memory.

14. Implementation of Memory Caching

- Each entry in a memory cache contains two values:
  - a memory address
  - the value of the byte found at that address
- In practice, storing a complete address with each entry is inefficient. Therefore, memory caches use clever techniques to reduce the amount of space needed.
- The two most important cache optimization techniques are known as:
  - Direct mapped memory cache
  - Set associative memory cache

Direct Mapped Memory Cache

- Memory caches are used with byte-addressable memories, a cache does not record individual bytes.
- A cache divides both the memory and the cache into a set of fixed-size blocks, where the block size, *B* (measured in bytes), is chosen to be a power of two.
- The hardware places an entire block in the cache whenever a byte in the block is referenced.
- We refer to a block in the cache as a ***cache line***; the size of a direct mapped memory cache is often specified by giving the number of cache lines times the size of a cache line.
  - For example, the size might be specified as 4K lines with 8 bytes per line.

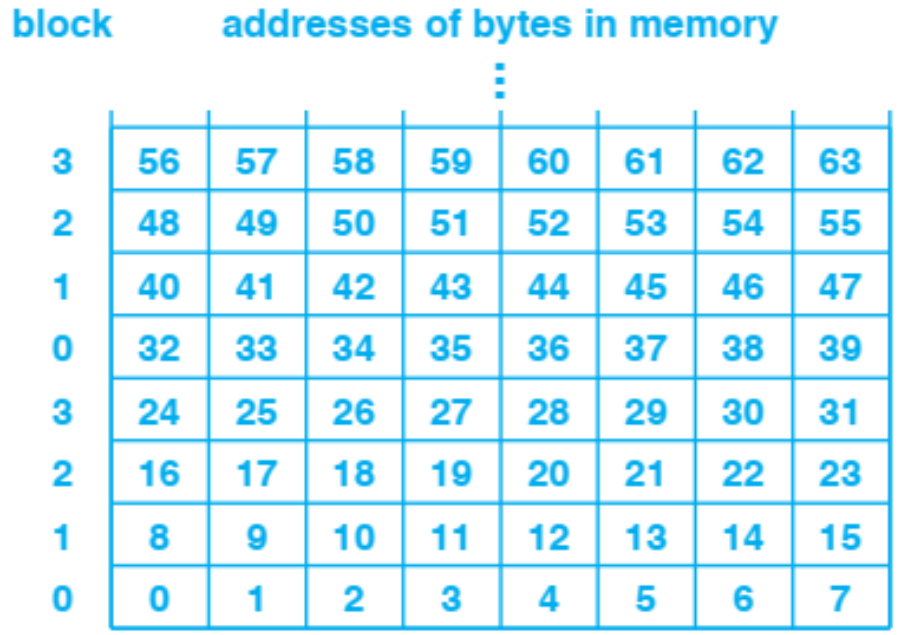


In this example, the cache has four lines with 8 bytes per line.

Figure 12.7 An example assignment of block numbers to memory locations for a cache of four blocks with eight bytes per block.

Direct Mapped Memory Cache

- Only a memory block numbered *i* can be placed in cache **slot** *i*.
  - For example, the block with addresses 16 through 23 can be placed in slot 2, as can the block with addresses 48 through 55.
- The cache attaches a unique tag to each group of *C* blocks (lines).



In this example, the cache has four lines with 8 bytes per line.

Figure 12.7 An example assignment of block numbers to memory locations for a cache of four blocks with eight bytes per block.

Direct Mapped Memory Cache

- To identify the block currently in a slot of the cache, each cache entry contains a tag value.
- If slot zero in the cache contains tag *K*, the value in slot zero corresponds to block zero from the area of memory that has tag *K*.

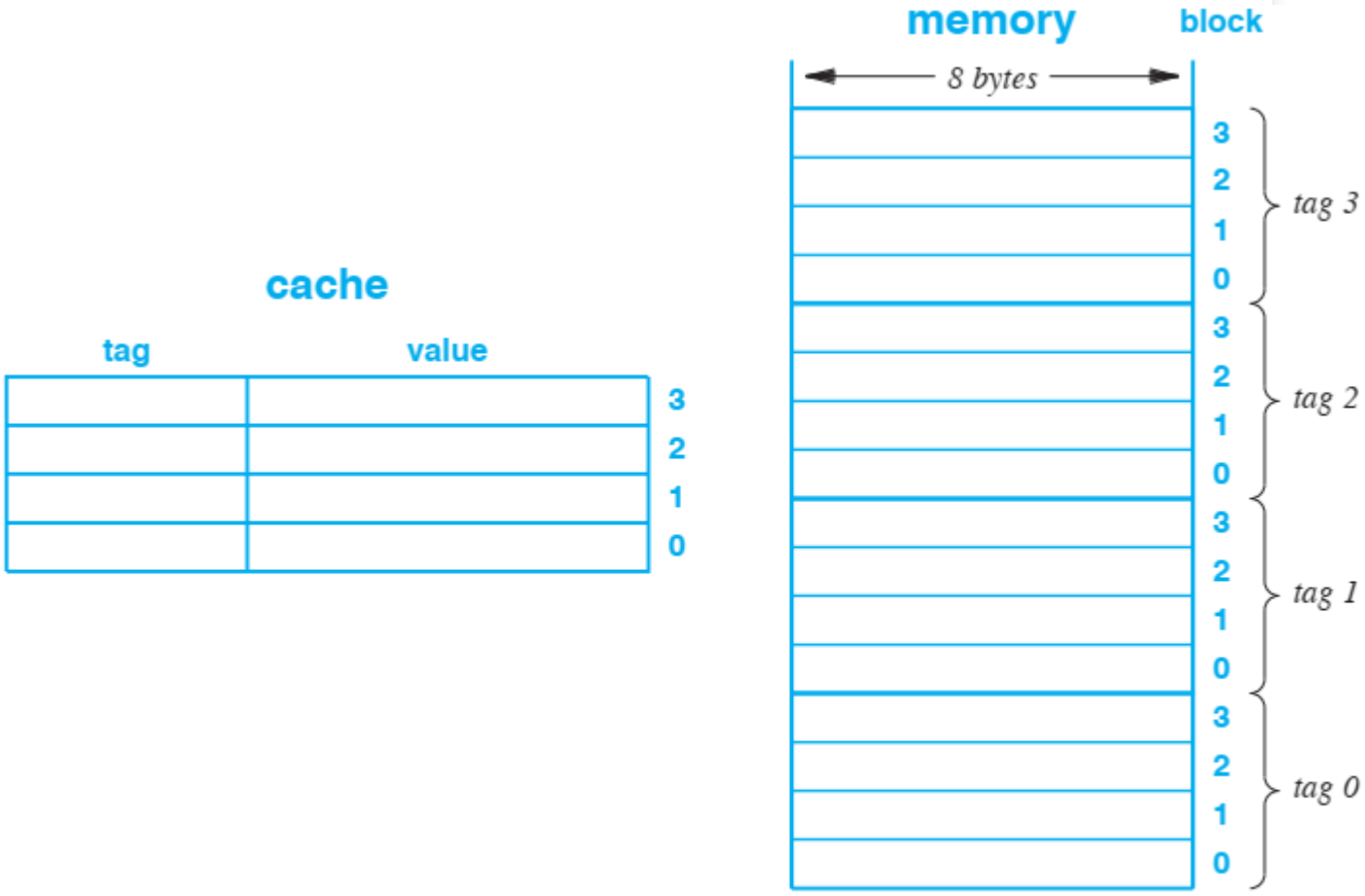


Figure 12.8 An example memory cache with space for four blocks and a memory divided into conceptual blocks of 8 bytes. Each group of four blocks in memory is assigned a unique tag.

Direct Mapped Memory Cache

- Hardware Implementation of a Direct Mapped Cache
  - The key point is that cache lookup can be performed quickly by combinatorial circuits.

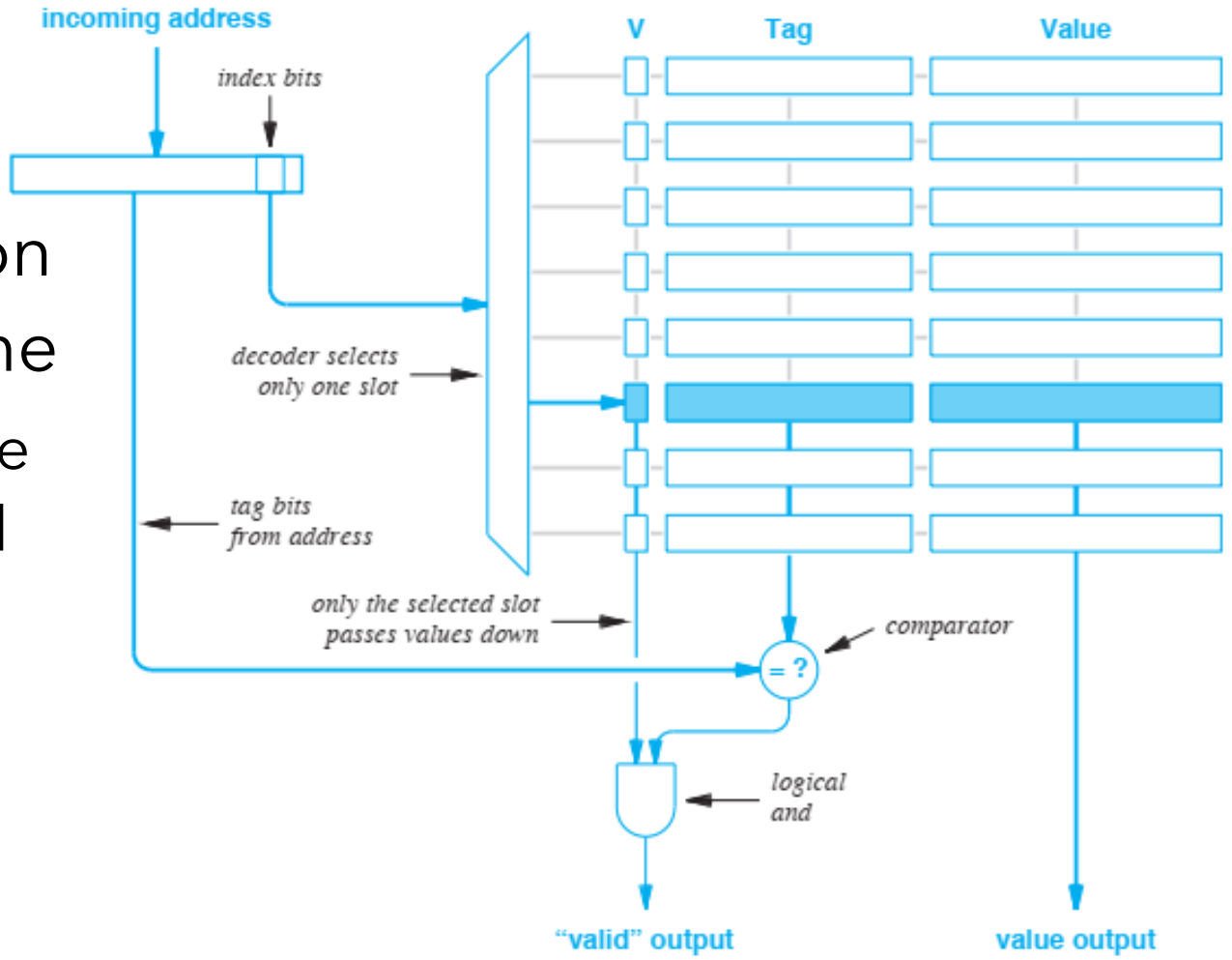


Figure 12.10 Block diagram of the hardware used to implement lookup in a memory cache.

Set Associative Memory Cache

- Instead of maintaining a single cache, the set associative approach maintains **multiple underlying caches**, and provides hardware that can search all of them **simultaneously**.
  - A set associative memory cache can store more than one block that has the same number.

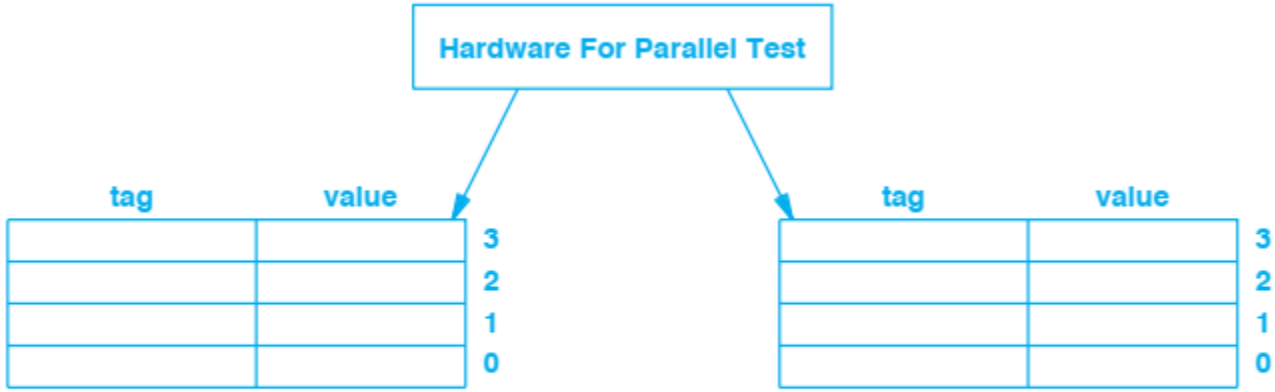


Figure 12.11 Illustration of a set associative memory cache with two copies of the underlying hardware. The cache includes hardware to search both copies in parallel.

Set Associative Memory Cache

- Consider a reference string in which a program alternately references two addresses, A1 and A2, that have different tags, but both have block number zero.
- In a direct mapped memory cache, the two addresses contend for a single slot in the cache.
  - A reference to A1 loads the value of A1 into slot 0 of the cache, and a reference to A2 overwrites the contents of slot 0 with the value of A2.
  - Thus, in an alternating sequence of references, every reference results in a cache miss.
- In a set associative memory cache, A1 can be placed in one of the two underlying caches, and A2 can be placed in the other. Thus, every reference results in a cache hit.

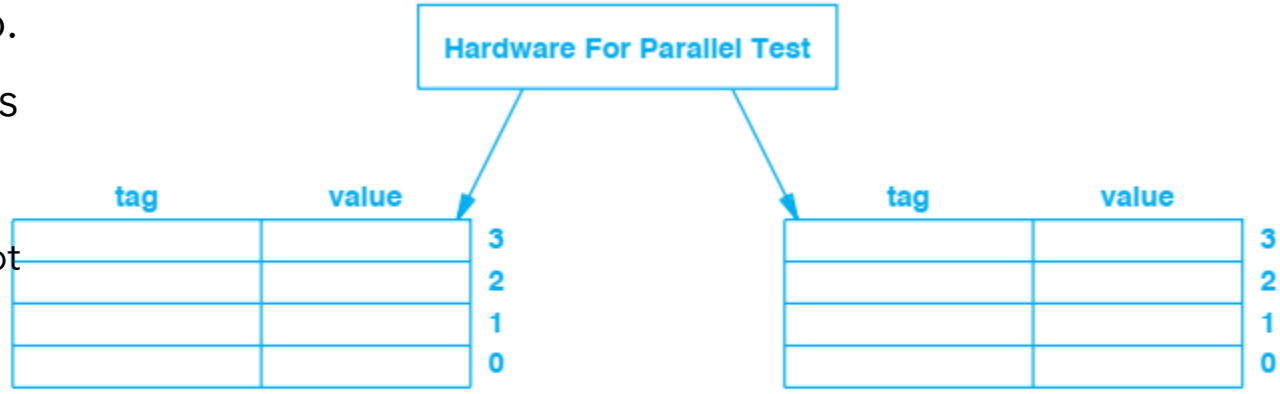


Figure 12.11 Illustration of a set associative memory cache with two copies of the underlying hardware. The cache includes hardware to search both copies in parallel.