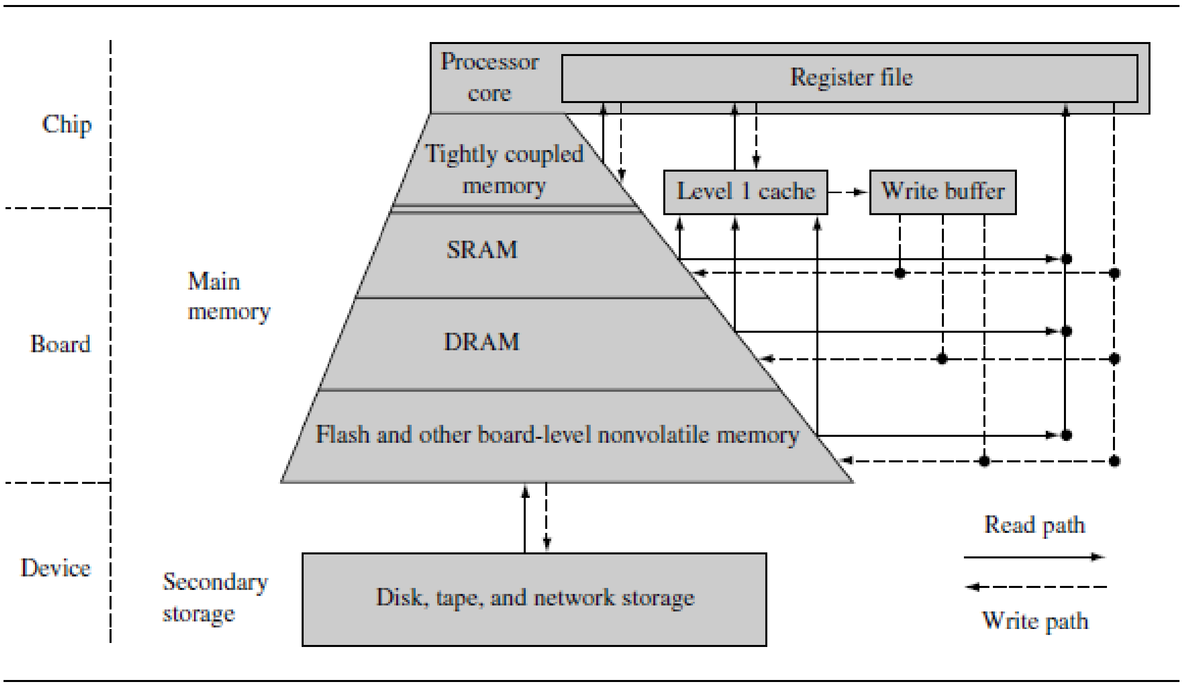
- A *cache* is a small, fast array of memory placed between the processor core and main memory that stores portions of recently referenced main memory.
- The goal of a cache is to reduce the memory access bottleneck imposed on the processor core by slow memory.
- Often used with a cache is a *write buffer*—a very small first-in-first-out (FIFO) memory placed between the processor core and main memory. The purpose of a write buffer is to free the processor core and cache memory from the slow write time associated with writing to main memory.
- Since cache memory only represents a very small portion of main memory, the cache fills quickly during program execution.
- Once full, the cache controller frequently evicts existing code or data from cache memory to make more room for the new code or data.
- This eviction process tends to occur randomly, leaving some data in cache and removing others.

The Memory Hierarchy and Cache Memory

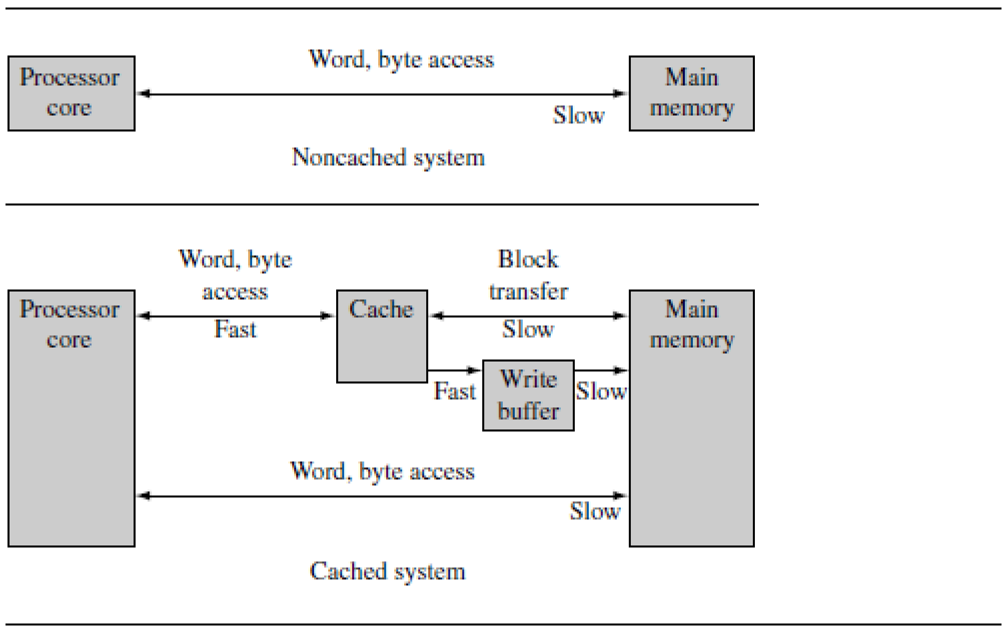


Memory Hierarchy

- Figure reviews some of this information to show where a cache and write buffer fit in the hierarchy.

- The innermost level of the hierarchy is at the processor core.
- This memory is so tightly coupled to the processor that in many ways it is difficult to think of it as separate from the processor. This memory is known as a *register file*.
- Also at the primary level is main memory. It includes volatile components like SRAM and DRAM, and non-volatile components like flash memory
- The next level is secondary storage—large, slow, relatively inexpensive mass storage devices such as disk drives or removable memory.
- Also included in this level is data derived from peripheral devices, which are characterized by their extremely long access times.
- A cache may be incorporated between any level in the hierarchy where there is a significant access time difference between memory components.
- The L1 cache is an array of high-speed, on-chip memory that temporarily holds code and data from a slower level.
- The write buffer is a very small FIFO buffer that supports writes to main memory from the cache.

- An L2 cache is located between the L1 cache and slower memory. The L1 and L2 caches are also known as the *primary* and *secondary* caches.

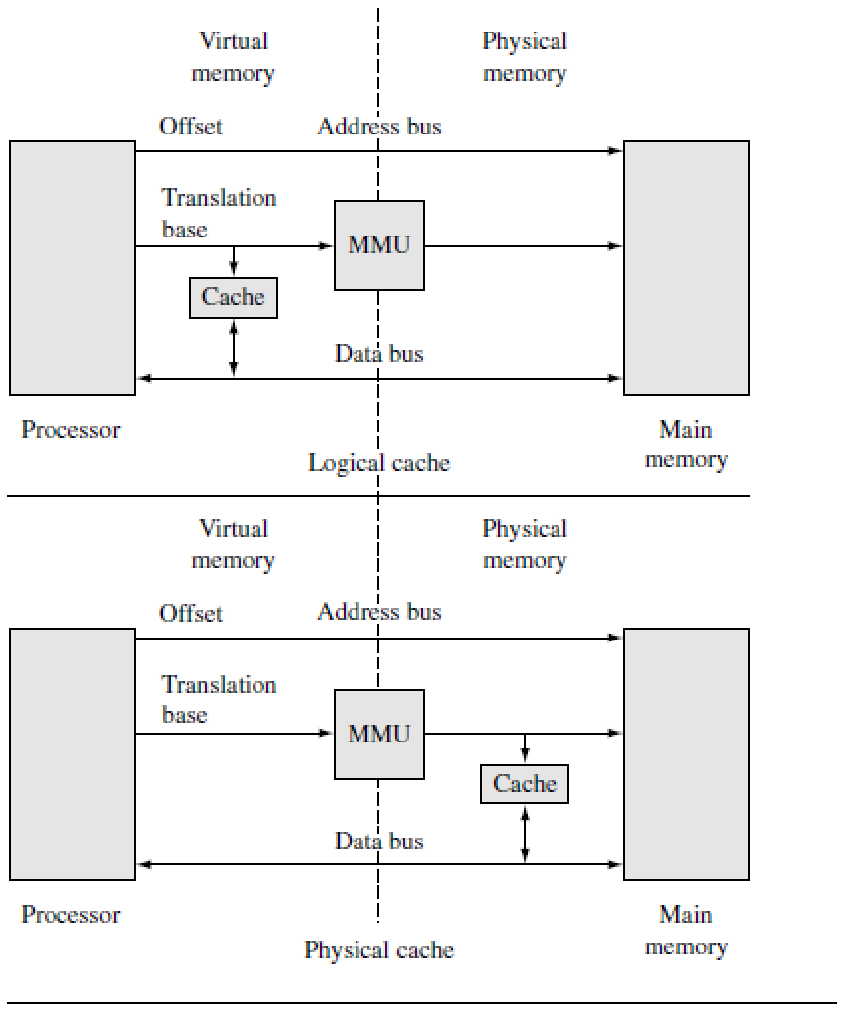


Relationship that a cache has between the processor core and main memory.

- Figure shows the relationship that a cache has with main memory system and the processor core.
- Upper part is without Cache and lower one is With cache.

Caches and Memory Management Units

- If a cached core supports virtual memory, it can be located between the core and the memory management unit (MMU), or between the MMU and physical memory.
- Placement of the cache before or after the MMU determines the addressing realm the cache operates in and how a programmer views the cache memory system.



Logical and physical caches.

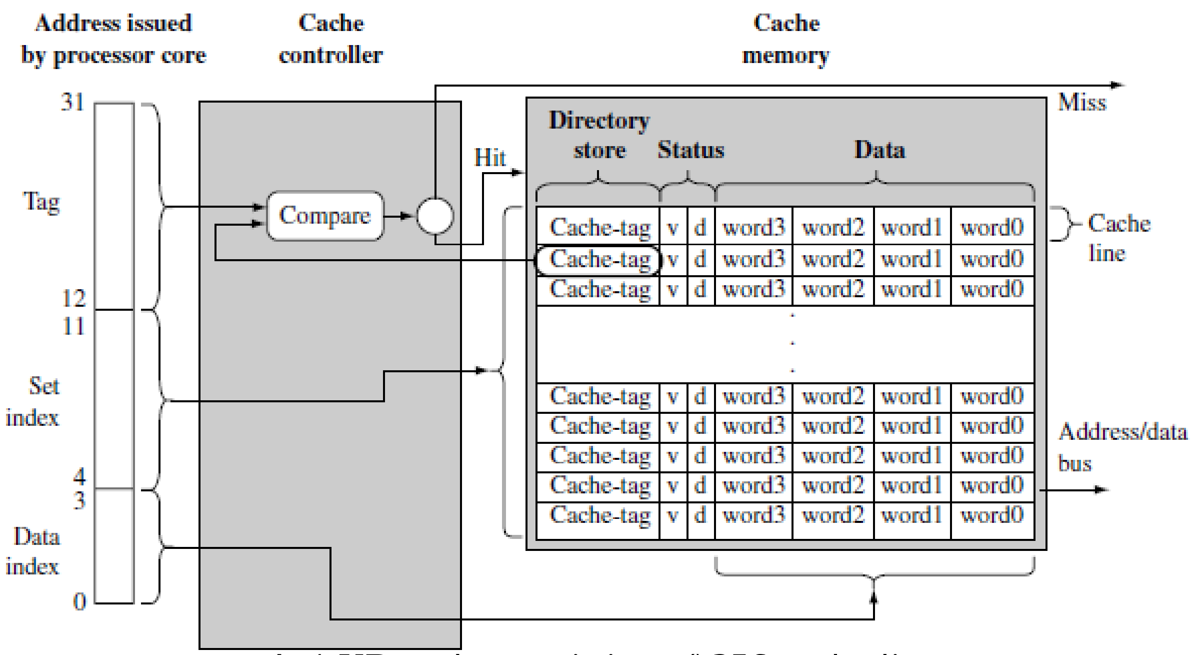
- A logical cache is located between the processor and the MMU.
- The processor can access data from a logical cache directly without going through the MMU.
- A logical cache is also known as a *virtual cache*.
- A *logical cache* stores data in a virtual address space.
- physical cache is located between the MMU and main memory. For the processor to access memory, the MMU must first translate the virtual address to a physical address before the cache memory can provide data to the core.
- A *physical cache* stores memory using physical addresses.

Cache Architecture

- ARM uses two bus architectures in its cached cores, the Von Neumann and the Harvard.
- A different cache design is used to support the two architectures.
- In processor cores using the Von Neumann architecture, there is a single cache used for instruction and data. This type of cache is known as a *unified cache*.

- The Harvard architecture has separate instruction and data buses to improve overall system performance, but supporting the two buses requires two caches.
- In processor cores using the Harvard architecture, there are two caches: an instruction cache (I-cache) and a data cache (D-cache). This type of cache is known as a *split cache*.

Basic Architecture of a Cache Memory



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

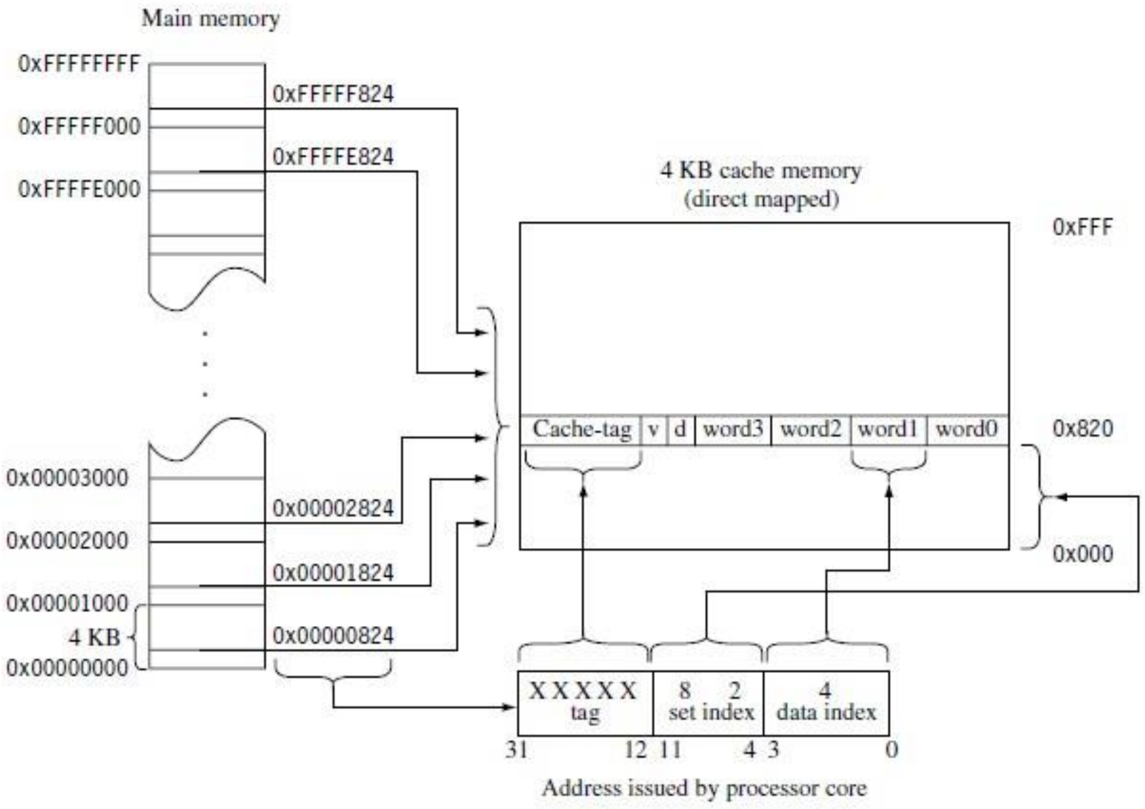
- A simple cache memory is shown on the right side of Figure
- It has three main parts: a directory store, a data section, and status information. All three parts of the cache memory are present for each cache line.
- The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address identifying where the cache line was copied from main memory. The directory entry is known as a *cache-tag*.
- A cache memory must also store the data read from main memory. This information is held in the data section
- The size of a cache is defined as the actual code or data the cache can store from main memory.
- There are also status bits in cache memory to maintain state information. Two common status bits are the valid bit and dirty bit.

- A *valid* bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the processor core on demand.
- A *dirty* bit defines whether or not a cache line contains data that is different from the value it represents in main memory.

Basic Operation of a Cache Controller

- The *cache controller* is hardware that copies code or data from main memory to cache memory automatically.
- First, the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and status bits, which the controller uses to determine the actual data stored there.
- The controller then checks the valid bit to determine if the cache line is active, and compares the cache-tag to the tag field of the requested address. If both the status check and comparison succeed, it is a cache *hit*. If either the status check or comparison fails, it is a cache *miss*.

The Relationship between Cache and Main Memory



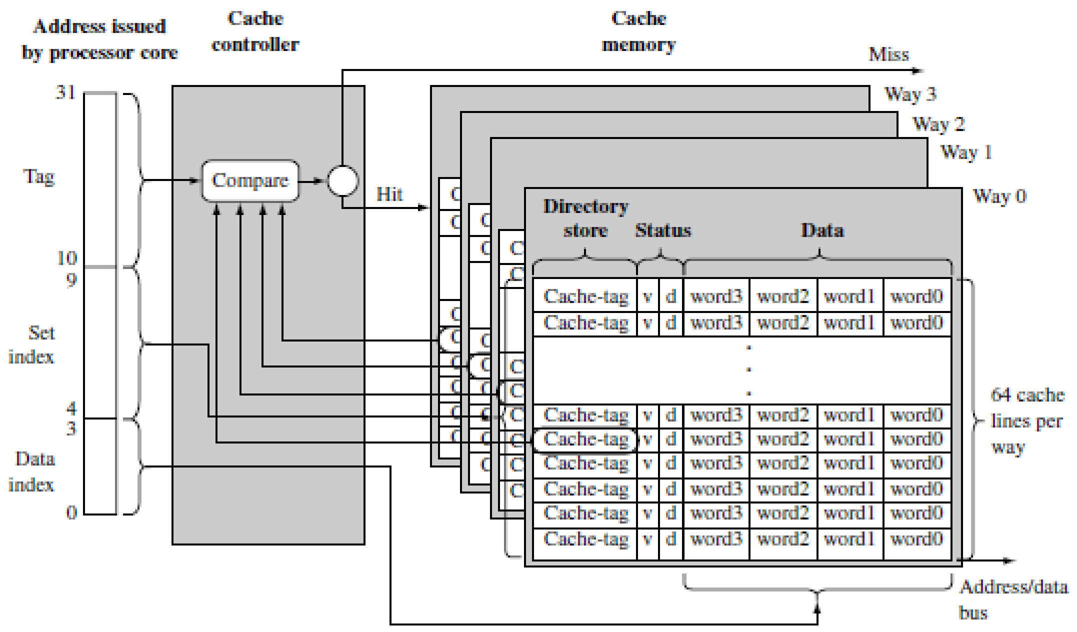
How main memory maps to a direct-mapped cache.

- The figure represents the simplest form of cache, known as a *direct-mapped* cache.

- In a direct-mapped cache each addressed location in main memory maps to a single location in cache memory.
- Since main memory is much larger than cache memory, there are many addresses in main memory that map to the same single location in cache memory.
- The figure shows this relationship for the class of addresses ending in 0x824.
- The set index selects the one location in cache where all values in memory with an ending address of
    - 0x824 are stored.
- The tag field is the portion of the address that is compared to the cache-tag value found in the directory store.
- The comparison of the tag with the cache-tag determines whether the requested data is in cache or represents another of the million locations in main memory with an ending address of 0x824.
- During a cache line fill the cache controller may forward the loading data to the core at the same time it is copying it to cache; this is known as *data streaming*.
- If valid data exists in this cache line but represents another address block in main memory, the entire cache line is evicted and replaced by the cache line containing the requested address. This process of removing an existing cache line as part of servicing a cache miss is known as *eviction*
- A direct-mapped cache is a simple solution, but there is a design cost inherent in having a single location available to store a value from main memory.
- Direct-mapped caches are subject to high levels of *thrashing*—a software battle for the same location in cache memory.

Set Associativity

- This structural design feature is a change that divides the cache memory into smaller equal units, called *ways*.



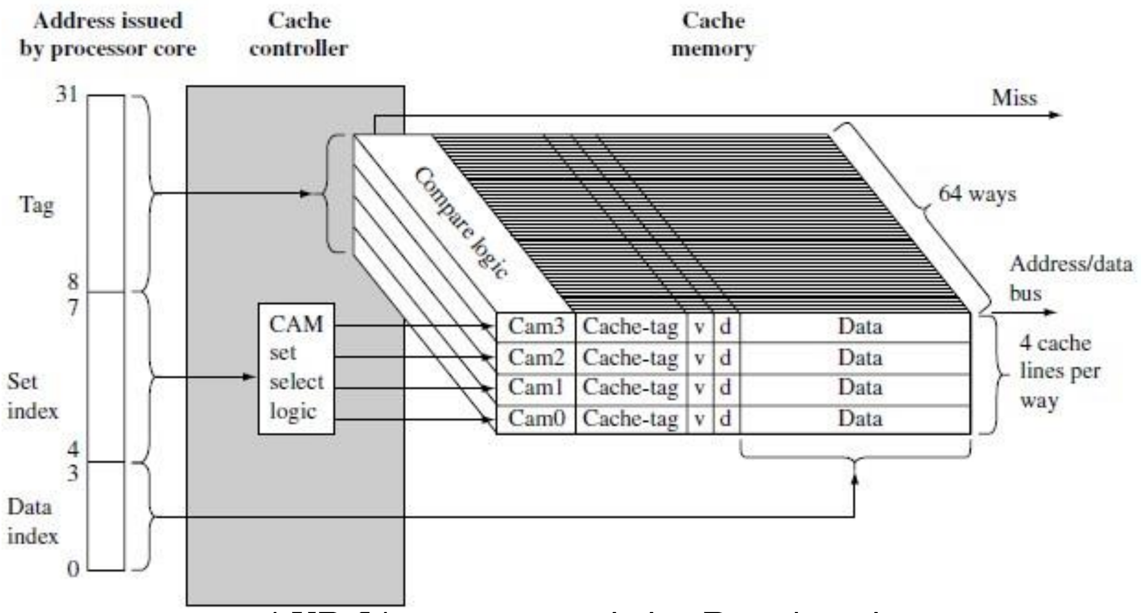
A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words

- the set index now addresses more than one cache line—it points to one cache line in each way. Instead of one way of 256 lines, the cache has four ways of 64 lines.
- The four cache lines with the same set index are said to be in the same *set*, which is the origin of the name "set index."
- A data or code block from main memory can be allocated to any of the four ways in a set without affecting program behaviour; in other words the storing of data in cache lines within a set does not affect program execution.
- The important thing to note is that the data or code blocks from a specific location in main memory can be stored in any cache line that is a member of a set.
- The bit field for the tag is now two bits larger, and the set index bit field is two bits smaller. This means four million main memory addresses now map to one set of four cache lines, instead of one million addresses mapping to one location.

Increasing Set Associativity

- The ideal goal would be to maximize the set associativity of a cache by designing it so any main memory location maps to any cache line.

- However, as the associativity increases, so does the complexity of the hardware that supports it. One method used by hardware designers to increase the set associativity of a cache includes a *content addressable memory* (CAM).
- CAM uses a set of comparators to compare the input tag address with a cache-tag stored in each valid cache line.
- Using a CAM allows many more cache-tags to be compared simultaneously, thereby increasing the number of cache lines that can be included in a set.



4 KB 64-way set associative D-cache using a CAM.

- The tag portion of the requested address is used as an input to the four CAMs that simultaneously compare the input tag with all cache-tags stored in the 64 ways.
- If there is a match, cache data is provided by the cache memory. If no match occurs, a miss signal is generated by the memory controller.
- The controller enables one of four CAMs using the set index bits.

Write Buffers

- A write buffer is a very small, fast FIFO memory buffer that temporarily holds data that the processor would normally write to main memory.
- In a system with a write buffer, data is written at high speed to the FIFO and then emptied to slower main memory.
- The write buffer reduces the processor time taken to write small blocks

of sequential data to main memory. The FIFO memory of the write buffer is at the same level in the memory hierarchy as the L1 cache and is shown in Figure

- The efficiency of the write buffer depends on the ratio of main memory writes to the number of instructions executed.
- A write buffer also improves cache performance; the improvement occurs during cache line evictions. If the cache controller evicts a dirty cache line, it writes the cache line to the write buffer instead of main memory.
- Data written to the write buffer is not available for reading until it has exited the write buffer to main memory.

Measuring Cache Efficiency

- The *hit rate* is the number of cache hits divided by the total number of memory requests over a given
  - *time interval. The value is expressed as a percentage:*

$$\textit{hit rate} = \left(\frac{\textit{cache hits}}{\textit{memory requests}}\right) \times 100$$

- The *miss rate* is similar in form: the total cache misses divided by the total number of memory requests expressed as a percentage over a time interval. Note that the miss rate also equals 100 minus the hit rate.
- *hit time*—the time it takes to access a memory location in the cache
- *miss penalty*—the time it takes to load a cache line from main memory into cache.

Cache Policy

There are three policies that determine the operation of a cache: the write policy, the replacement policy, and the allocation policy. The cache write policy determines where data is stored during processor write operations. The replacement policy selects the cache line in a set that is used for the next line fill during a cache miss. The allocation policy determines when the cache controller allocates a cache line.

Write Policy—Writeback or Writethrough

When the processor core writes to memory, the cache controller has two alternatives for its write policy. The controller can write to both the cache and main memory, updating the values in both locations; this approach is known as writethrough. Alternatively, the cache controller can write to cache memory and not update main memory, this is known as writeback or copyback.

Writethrough

When the cache controller uses a writethrough policy, it writes to both cache and main memory when there is a cache hit on write, ensuring that the cache and main memory stay coherent at all times. Under this policy, the cache controller performs a write to main memory for each write to cache memory. Because of the write to main memory, a writethrough policy is slower than a writeback policy.

Writeback

When a cache controller uses a writeback policy, it writes to valid cache data memory and not to main memory. Consequently, valid cache lines and main memory may contain different data. The cache line holds the most recent data, and main memory contains older data, which has not been updated.

Caches configured as writeback caches must use one or more of the dirty bits in the cache line status information block. When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true. If the core accesses the cache line at a later time, it knows by the state of the dirty bit that the cache line contains data not in main memory. If the cache controller evicts a dirty cache line, it is automatically written out to main memory.

The controller does this to prevent the loss of vital information held in cache memory and not in main memory. One performance advantage a writeback cache has over a writethrough cache is in the frequent use of temporary local variables by a subroutine. These variables are transient in nature and never really need to be written to main memory

transient variables is a local variable that overflows onto a cached stack because there are not enough registers in the register file to hold the variable.

Cache Line Replacement Policies

On a cache miss, the cache controller must select a cache line from the available set in cache memory to store the new information from main memory. The cache line selected for replacement is known as a victim. If the victim contains valid, dirty data, the controller must write the dirty data from the cache memory to main memory before it copies new data into the victim cache line. The process of selecting and replacing a victim cache line is known as eviction.

The strategy implemented in a cache controller to select the next victim is called its replacement policy. The replacement policy selects a cache line from the available associative member set; that is, it selects the way to use in the next cache line replacement. To summarize the overall process, the set index selects the set of cache lines available in the ways, and the replacement policy selects the specific cache line from the set to replace.

ARM cached cores support two replacement policies, either pseudorandom or round-robin.

Round-robin or cyclic replacement simply selects the next cache line in a set to replace. The selection algorithm uses a sequential, incrementing victim counter that increments each time the cache controller allocates a cache line. When the victim counter reaches a maximum value, it is reset to a defined base value.

Pseudorandom replacement randomly selects the next cache line in a set to replace. The selection algorithm uses a nonsequential incrementing victim counter. In a pseudoran dom replacement algorithm the controller increments the victim counter by randomly selecting an increment value and adding this value to the victim counter. When the victim counter reaches a maximum value, it is reset to a defined base value

Another common replacement policy is least recently used (LRU). This policy keeps track of cache line use and selects the cache line that has been unused for the longest time as the next victim.

Allocation Policy on a Cache Miss

There are two strategies ARM caches may use to allocate a cache line after a the occurrence of a cache miss. The first strategy is known asread-allocate, and the second strategy is known as read-write-allocate.

A read allocate on cache miss policy allocates a cache line only during a read from main memory. If the victim cache line contains valid data, then it is written to main memory before the cache line is filled with new data.

A read-write allocate on cache miss policy allocates a cache line for either a read or write to memory. Any load or store operation made to main memory, which is not in cache memory, allocates a cache line. On memory reads the controller uses a read-allocate policy.

Coprocessor 15 and Caches

There are several coprocessor 15 registers used to specifically configure and control ARM cached cores. Table 12.2 lists the coprocessor 15 registers that control cache configuration. Primary CP15 registers c7 and c9 control the setup and operation of cache. Secondary CP15:c7 registers are write only and clean and flush cache. The CP15:c9 register defines the

Table 12.2 Coprocessor 15 registers that configure and control cache operation.

Function	Primary register	Secondary registers	Opcode 2
Clean and flush cache	*c7*	*c5, c6, c7, c10, c13, c14*	0, 1, 2
Drain write buffer	*c7*	*c10*	4
Cache lockdown	*c9*	*c0*	0, 1
Round-robin replacement	*c15*	*c0*	0

victim pointer base address, which determines the number of lines of code or data that are locked in cache.