Chapter 4a: Fully Associative Cache

Gene Cooperman

Copyright ©2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

Contents

4a Fully Associative Cache **2**
4a.1 Motivation: Why Do We Need a Cache? . . . 2
4a.2 The Cache as a Key-Value Store . . . 3
4a.3 Fast Lookup: Parallel Comparators . . . 3
4a.4 Exploiting Temporal Locality . . . 5
4a.5 Exploiting Spatial Locality: Data Blocks . . . 6
4a.6 The Valid Bit and the Modified Bit . . . 9
4a.7 Eviction . . . 10
4a.8 Putting It All Together: A Worked Example . . . 11
4a.9 Looking Ahead: The Scalability Problem . . . 12
4a.10 Summary . . . 13
4a.11 Check Your Understanding . . . 14

Copyright © 2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

We will present a cache as a *key-value store*. A **cache** is a small, fast memory that keeps copies of recently used data from a larger, slower memory. For a CPU cache, the key is a memory address, and the value is the data stored at that address. A key-value store is normally considered a software architecture. It is often used in distributed systems, where multiple computer nodes need a shared key-value store. Consistent with the software focus of computer systems, we will view the hardware cache on a CPU chip as simply one additional type of key-value store.

The chapters ahead describe several hardware mechanisms that share some common, unifying elements:

1. Each one can be thought of, conceptually, as simply a **key-value store**.
2. Each one requires **hardware support** to be efficient, because a lookup happens on nearly every memory access and must take only a few nanoseconds.
3. In each one, the central data structure can be thought of as an array of entries, and each entry holds a fixed-size chunk of memory. That chunk is **atomic**: it is always copied as a whole between RAM and the cache, or between disk and RAM, and never in part. For example, a CPU cache typically copies 64 bytes at a time from RAM, never just some of those bytes.

This chapter and the next two describe three ways to build a key-value store in hardware: the *fully associative*, *direct-mapped*, and *set-associative* caches. All three appear in a real computer:

- **CPU caches:** The caches between the CPU and RAM are generally *set-associative caches*.
- **Page table:** In **virtual memory** (the subject of a fourth chapter), a page table records which memory pages of a process reside in RAM (and where in RAM), and which are still on disk. The page table works much like a *direct-mapped cache*.

- **TLB:** Consulting the page table (which is itself stored in RAM) on every memory access would be slow. So the CPU keeps a small TLB (Translation Lookaside Buffer), which holds the most recently used entries of the page table. The TLB is a *fully associative cache*.

We begin with the fully associative cache, because it is the simplest type of cache, and the concepts that it introduces apply to every cache.

4a Fully Associative Cache

4a.1 Motivation: Why Do We Need a Cache?

When a CPU executes a program, many of the instructions need to access data in RAM (main memory). In assembly language, these are the **load word** (`lw`) and **store word** (`sw`) instructions:

- `lw` reads a word (4 bytes) from RAM into a CPU register.
- `sw` writes a word from a CPU register into RAM.

Throughout these chapters, we assume 32-bit addresses. (Real 64-bit CPUs use longer addresses, but nothing in these chapters changes except the number of address bits.) For simplicity, we also assume that the CPU runs just a single process. The chapter on virtual memory will refine this picture, with a brief note about the cache when the CPU switches among many processes and the kernel.

The problem is that **RAM is much slower than the CPU**. A modern CPU core can execute an instruction in about 0.3 nanoseconds, but a RAM access might take 50-100 nanoseconds. If the CPU had to wait for RAM on every `lw` or `sw`, each wait would cost a few hundred CPU cycles.

The solution is to place a small, fast memory, a **cache**, directly on the CPU chip. The cache holds copies of the most recently used addresses and their corresponding data from recent `lw` and `sw` operations. Because the cache is physically on the same chip as the CPU, accessing the cache is extremely fast. For the fastest cache, called L1 (see below), an access often takes just 1 or 2 nanoseconds.

The goal is simple: if the CPU needs data it has accessed recently, it can find that data in the cache and avoid the slow trip to RAM.

4a.1.1 The Three Levels of CPU Cache

A real CPU has not one cache, but a hierarchy of three caches, called **L1**, **L2**, and **L3** (for "level 1", "level 2", and "level 3"). On a `lw` or `sw`, the CPU first checks L1. On an L1 miss, it checks L2, and then L3. Only if all three miss does it go to RAM. Each level is larger, but slower, than the one before it. Typical values for a current desktop or laptop CPU are:

Level	Typical size	Typical access time	Shared by cores?
L1	32-64 KB for data, plus a similar L1 cache for instructions	about 1 ns (4-5 CPU cycles)	No: one per core
L2	0.5-3 MB	about 3-5 ns	Usually one per core
L3	8-64 MB	about 10-20 ns	Yes: shared by all cores
RAM	many GB	50-100 ns	Yes

The L1 cache is split into two halves: an **L1 instruction cache**, used for fetching instructions, and an **L1 data cache**, used by `lw` and `sw`. In these chapters, "the cache" means the L1 data cache, unless we say otherwise.

But the ideas apply to all three levels.

4a.2 The Cache as a Key-Value Store

The simplest way to think about a cache is as a **key-value store** (like a dictionary or hash map in software):

- The **key** is the memory address used by `lw` or `sw`.
- The **value** is the data (the word) at that address.

Each row in the cache stores one such key-value pair. We call each row, or entry, a **cache line** (more on this term later). Here is a simple picture of a cache with 4 entries:

Address (Key)	Data (Value)
0x0000_1A00	0xDEADBEEF
0x0000_3F04	0x12345678
0x0000_2C08	0x00000042
0x0000_7B0C	0xCAFEBABE

Just as in a software key-value store, a key can be stored anywhere: **any address can be stored in any cache line**. A cache with this property is called a **fully associative cache**. The word *associative* means that data is found by its key (its address), not by its position in the cache. Software uses the same idea in an *associative array*, another name for a dictionary or hash map. The word *fully* means that a key may be in *any* of the cache lines. So a lookup must be prepared to check all of the cache lines.

4a.2.1 Cache Hit vs. Cache Miss

When the CPU executes `lw` or `sw`, the hardware looks up the address in the cache. If the address is found, the access is a **cache hit**, and the data is read or written directly in the cache. If the address is not found, the access is a **cache miss**. The CPU must then fetch the data from RAM, and the cache keeps a copy. For example, suppose that a program executes `lw` twice on the same address. The first `lw` is a MISS, and the second `lw` is a HIT. The second access is dramatically faster. This is the fundamental benefit of caching: **programs often re-use the same addresses** (a property called *temporal locality*, which we will discuss below).

4a.3 Fast Lookup: Parallel Comparators

4a.3.1 Why a Naive Lookup Is Slow

Because the cache is on the CPU chip, loading one cache line into the CPU is fast: about 1 or 2 nanoseconds, or roughly 25 to 100 times faster than loading from RAM. But suppose we built a naive hardware cache that works like a loop in software. On every `lw` or `sw`, the CPU would:

1. Load the first cache line into the CPU.
2. Compare the address stored in that cache line with the address requested by `lw` or `sw`.
3. If the two addresses match, it is a HIT. Otherwise, load the next cache line and repeat.

On a cache miss, the CPU would have to load and check **all N cache lines** before it could even conclude that the address is not in the cache. Even for a small cache of just N=64 cache lines, that is 64 loads at 1 or 2 nanoseconds each, or about 64 to 128 nanoseconds. That is just as slow as going directly to RAM, which defeats the whole purpose of a cache.

The problem is not that each individual check is slow. The problem is that the checks happen **one at a time**. For this reason, the hardware provides a **comparator** for each cache line, so that all of the cache lines are checked in parallel.

4a.3.2 One Comparator per Cache Line

In digital hardware, we can do something that a loop cannot do: **perform all N comparisons simultaneously, in parallel.**

The building block is a digital circuit called a **comparator**. A comparator takes two addresses as its inputs, and produces a single-bit output:

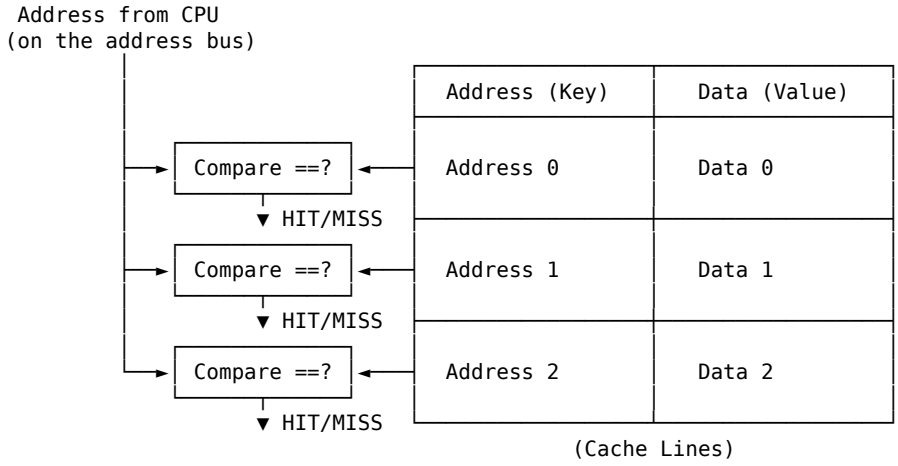
- Output = **1** (HIT) if the two addresses are equal.
- Output = **0** (MISS) if the two addresses are different.

We place **one comparator next to each cache line**. Every comparator receives two inputs:

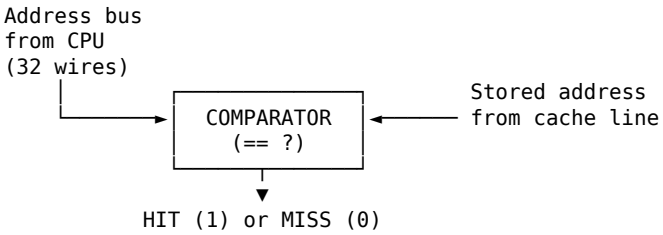
1. The **address from the CPU** (the address requested by `lw` or `sw`).
2. The **address stored in that cache line**.

All comparators operate at almost the same time, because they all receive the same electrical signal. The address from the CPU travels on a **bus**, a set of parallel electric wires (one wire per bit of the address). This bus physically splits, sending a copy of the signal to each comparator. Household electrical wiring follows the same principle. Electricity enters a house through a single line, and the wires then branch off to deliver power to every room simultaneously. No room has to "wait its turn." Similarly, every comparator receives the address at nearly the same instant.

Here is a diagram of the cache with comparators:



A more detailed view of a single cache line and its comparator:



For now, each comparator compares a full 32-bit address. So each comparator is fed by all 32 wires of the address bus. Later in this chapter, each cache line will hold a 64-byte data block, and the low 6 bits of every block-aligned address will always be zero (see the section Block-Aligned Addresses). So there is no need to store or compare those 6 bits. The hardware will store and compare only the remaining high 26 bits, called the **tag**. Nothing else about the comparators changes: they simply compare 26 bits instead of 32.

4a.3.3 From HIT/MISS Signals to Data

Each comparator produces its own HIT/MISS signal. The cache copies an address from RAM only on a MISS. So it never stores the same address in two cache lines, and at most one comparator can output 1. (In a later section, The Valid Bit and the Modified Bit, we will see that a HIT output of a comparator will be a HIT only if a valid bit for the cache line has also been set.) Two things then happen:

1. The N individual HIT/MISS signals are combined by an OR gate into a single overall HIT signal for the CPU. If no comparator outputs 1, the result is a MISS, and the cache must go to RAM.
2. Each comparator's output also acts as an *enable* signal for its own cache line. On a HIT, the one cache line whose comparator fired places its data onto the data bus going to the CPU. All other cache lines stay silent.

Key insight: Because all comparators run in parallel, the lookup time is nearly the same whether the cache has 4 entries or 64 entries. This parallel lookup is what makes a fully associative cache fast. Moreover, only hardware can do such a lookup efficiently. Recall that in a **fully associative** cache, any address can be stored in any cache line. The comparators are what make it possible to check all of those cache lines simultaneously.

4a.4 Exploiting Temporal Locality

The comparators make each lookup fast. However, a fast lookup helps only if the lookup usually finds what it is looking for. In other words, the cache is efficient only if most `lw` and `sw` instructions are HITs. Two properties of programs make that likely: *temporal locality* (this section) and *spatial locality* (the next section). For each property, we will see what goes wrong when a program lacks it, and how the cache takes advantage of it. For temporal locality, the cache itself is the hardware that takes advantage of it: by keeping a copy of the data after the first use of an address, the cache turns later uses of the same address into HITs. For spatial locality, the cache needs one more piece of hardware: larger data blocks.

4a.4.1 Temporal Locality

A program has **temporal locality** (locality in time) if, after it uses an address, it is likely to use the *same* address again soon. Most often, the reason is a loop, which uses the same counter and accumulator on every iteration. Each re-use of an address that is still in the cache is a HIT.

Consider the loop below. Assume that `total` is a global variable that the compiler keeps in RAM (not in a register). Then each iteration executes a `lw` and a `sw` on `total`, as well as a `lw` on `a[i]`.

```
int a[1000000];
int total = 0;
...
for (int i = 0; i < 1000000; i++)
    total = total + a[i];
```

The loop has two kinds of data, `total` and `a[i]`. But only one of them has temporal locality.

1. **`total` has temporal locality.** Its address is used on every iteration. The first `lw` on `total` is a MISS, but after that, every `lw` and `sw` on `total` is a HIT.
2. **`a[i]` has no temporal locality.** Each element of `a` is loaded exactly once, and never again. So every `lw` on `a[i]` is a MISS: one million trips to RAM. The cache stores a copy of each `a[i]`, but never gets a second chance to use it.

This is what goes wrong without temporal locality: the cache keeps copies of data that the program never asks for again. (Fortunately, the hardware can still speed up the loads of `a[i]` by using spatial locality, as we will see next.)

4a.5 Exploiting Spatial Locality: Data Blocks

4a.5.1 Spatial Locality

A program has **spatial locality** (locality in space) if, after it uses an address, it is likely to use *nearby* addresses soon. Most often, the reason is a loop that walks through an array in order. For example, the loop over `a[i]` above loads `a[i]` at some address `X`, and then loads `a[i+1]` at address `X+4`.

Whether a program has spatial locality often depends on the order in which the program visits its data. That order is the programmer's choice. In C, a two-dimensional array is stored row by row: `m[i][j]` is immediately followed in memory by `m[i][j+1]`. Here are two loops that compute the same sum, but visit the array in different orders:

```
int m[1024][1024];     /* 4 MB in total; each row is 4096 bytes */

/* Loop 1: row by row (good spatial locality) */
for (int i = 0; i < 1024; i++)
    for (int j = 0; j < 1024; j++)
        total = total + m[i][j];    /* next address: 4 bytes later */

/* Loop 2: column by column (poor spatial locality) */
for (int j = 0; j < 1024; j++)
    for (int i = 0; i < 1024; i++)
        total = total + m[i][j];    /* next address: 4096 bytes later */
```

The two loops visit the same data. But only one of them has good spatial locality.

1. **Loop 1 has good spatial locality.** It walks along each row in order. Each access is just 4 bytes after the previous one. So the program keeps using addresses near the one that it just used.
2. **Loop 2 has poor spatial locality.** It walks down each column. Each access is 4096 bytes away from the previous one. So the program never uses an address near the one that it just used.

On a real CPU, Loop 2 is often several times slower than Loop 1, even though both loops execute the same number of `lw` instructions. The reason is the hardware solution described next.

4a.5.2 Larger Data Blocks: 64 Bytes per Cache Line

So far, each cache line holds a single word (4 bytes). With one-word cache lines, the cache cannot take advantage of spatial locality: in Loop 1, loading `m[i][j]` into the cache does nothing to help with `m[i][j+1]`.

The hardware solution is to make each cache line hold a larger **data block**: a chunk of contiguous memory. On a MISS, the cache loads the entire data block that contains the requested address. So the neighboring addresses come along for free.

How large should a data block be? Larger is not always better. A larger block brings in more neighbors on each MISS, which helps a program with spatial locality. But a larger block also takes longer to transfer on each MISS. In addition, the size of the cache is fixed. So larger blocks mean fewer blocks in the cache. With fewer blocks, the cache can hold fewer separate regions of memory, which hurts temporal locality. Finally, when the program does not use the neighbors (as in Loop 2), the extra bytes are wasted.

Most current Intel, AMD, and ARM processors settle on **64-byte data blocks**, and we will use 64 bytes in the rest of this chapter. A 64-byte data block holds 16 `int`s. There is also a specific reason for the number 64: RAM delivers data in bursts of exactly 64 bytes (see the next subsection). So fetching fewer than 64 bytes would hardly make a MISS any faster.

Now compare the two loops:

- **Loop 1:** A load of `m[i][0]` is a MISS, and it brings in the data block holding `m[i][0]` through `m[i][15]`. The next 15 loads are all HITs. Only 1 load in 16 goes to RAM. (The same is true for `a[i]` in the earlier loop.)
- **Loop 2:** Each load is in a different data block. The cache loads 64 bytes, but the program uses only 4 of them before jumping 4096 bytes away. The rest of that data block would be useful for column `j+1`, but the inner loop first touches 1024 different data blocks (64 KB). That is more than a typical L1 data cache holds (often 32 KB or 48 KB). So by the time the program comes back, the cache has already had to evict the original data block to make room. Nearly every load is a MISS, and 60 of every 64 bytes fetched from RAM are wasted.

4a.5.3 A Data Block Is Atomic

The critical idea behind data blocks is:

> **A data block is atomic.**

This means the entire 64-byte block is transferred as a single unit between RAM and the cache. A data block is either entirely present in a cache line, or not present at all. Why insist on this? There are two reasons.

First, it costs almost nothing. The bus to RAM has 64 data wires. So a block is sent as a *burst* of 8 back-to-back transfers: 8 transfers × 64 bits = 64 bytes. This burst of 8 transfers is exactly what a memory channel for DDR4 RAM uses. (DDR5 RAM uses narrower 32-bit channels with bursts of 16 transfers, which again comes to 64 bytes.)

Second, it keeps the bookkeeping simple. Each cache line needs just one valid bit (described below), meaning "this whole data block is good."

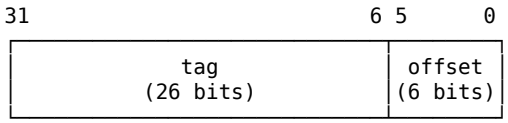
4a.5.4 Block-Aligned Addresses

When we use 64-byte data blocks, the addresses stored in the cache must always be **aligned** to 64-byte boundaries. That is, the address of the first byte in a data block is divisible by 64.

Here is how alignment works. Suppose the CPU executes `lw` with address `ADDR`, and the access is a cache miss. The cache will:

1. Compute the **block-aligned address**: `BLOCK_ADDR = ADDR - (ADDR % 64)`. This rounds `ADDR` down to the nearest multiple of 64.
2. Fetch the **64 bytes** starting at `BLOCK_ADDR` from RAM.
3. Store `BLOCK_ADDR` as the address (key) in the new cache line, and store the 64 bytes as the data block (value).

In hardware, no arithmetic is needed for step 1. Since $64 = 2^6$, the value `ADDR % 64` is simply the low 6 bits of `ADDR`, and `BLOCK_ADDR` is just `ADDR` with those low 6 bits set to zero. So the hardware views a 32-bit address as two fields:



- The **offset** (low 6 bits) says which byte *within* the data block we want.
- The **tag** (high 26 bits) identifies *which* data block we want.

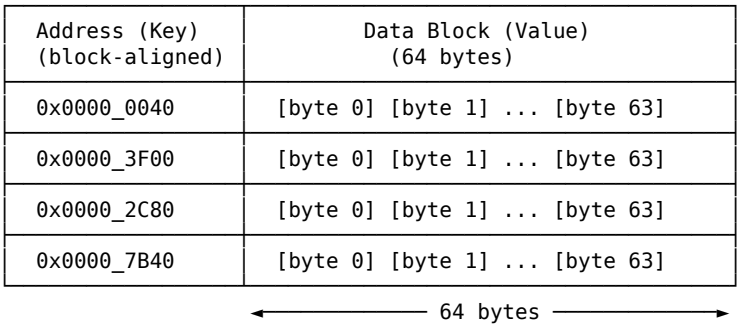
Since the low 6 bits of a block-aligned address are always zero, the cache stores and compares only the 26-bit **tag**, as promised in the section One Comparator per Cache Line. From now on, "the address stored in a cache line" means its tag.

For example, if `ADDR = 0x0000_0044` (which is 68 in decimal):

- `68 % 64 = 4`. Therefore, `BLOCK_ADDR = 68 - 4 = 64 = 0x0000_0040`.
- The cache fetches bytes 64 through 127 from RAM (64 bytes total).
- In binary, the low 6 bits of `0x44` are `000100`. So the offset is 4. The tag is `0x44 >> 6 = 1`.
- The word at address 68 occupies bytes 4 through 7 of the data block. The hardware selects those 4 bytes using the offset.

Now the payoff of spatial locality: if the program next executes `lw` on addresses 72, 76, ..., 124, every one of them is a cache hit. Those words were already brought into the cache, as part of the same data block. Notice that HIT means that the **data block containing `ADDR`** is in the cache. The hardware checks this by comparing the tag of the `ADDR` requested by the CPU against the tag stored in the cache line that contains that data block.

Here is a diagram of the cache with data blocks:



Notice that in every address in the "Address" column, the last two hex digits are `00`, `40`, `80`, or `C0`. These are all multiples of 64 (0x40). Equivalently, the low 6 bits of each address are zero. This is what "64-byte aligned" means.

> **NOTE:** Alignment matters for individual words, too. A C program can create a misaligned `int`, for example:

```
char *buf = malloc(128);
int *p = (int *)(buf + 62);   // address is not divisible by 4
int x = *p;                   // misaligned load
```

The C standard says that dereferencing a misaligned `int *` is *undefined behavior*. Even so, many CPUs (including most ARM and x86 CPUs) will carry out the misaligned load in hardware. On other CPUs (including many RISC-V CPUs), the load traps, and the operating system emulates it in software. Either way, a misaligned access is costly. An *aligned* 4-byte word always lies inside a single 64-byte data block. But a *misaligned* word can straddle two data blocks. For example, if `buf` happens to be 64-byte aligned, then `*p` consists of bytes 62 through 65, which lie in two different data blocks. The cache must then do two lookups, and possibly handle two cache misses, for a single `lw`.

4a.6 The Valid Bit and the Modified Bit

Any CPU cache is small compared to the size of RAM. So the cache soon fills up, and from then on, each MISS must **evict** some cache line to make room for the new data block. To manage this, each cache line carries two additional single-bit flags beyond the address and data block:

- **V (Valid bit):** Indicates whether this cache line holds meaningful data.
- **M (Modified bit):** Indicates whether the data block has been changed relative to what is stored in RAM.

When we include these bits, we refine our terminology:

- **Cache line** refers to the full entry: the address, the data block, the V bit, and the M bit.
- **Data block** refers only to the data portion of the cache line (the 64 bytes).

In practice, you will often see "cache line" used for just the data block, as in "a CPU with 64-byte cache lines."

V	M	Address (Key)	Data Block (64 bytes)
1	0	0x0000_0040	[byte 0] ... [byte 63]
1	1	0x0000_3F00	[byte 0] ... [byte 63]
0	0	(ignored)	(ignored)
1	0	0x0000_7B40	[byte 0] ... [byte 63]

4a.6.1 The Valid Bit (V)

When the machine is powered on, the cache starts out **empty**. Every cache line has `V=0`, meaning "this line does not contain valid data; ignore it." As the program runs and accesses memory, cache lines are filled and their valid bits are set to `V=1`.

Why is the V bit needed? When the power comes on, the cache's memory cells hold random bits. Without the V bit, a random stored tag could happen to equal the tag of a real address. The cache would then report a HIT, and return garbage data. So a HIT requires *both* a matching tag *and* `V=1`.

While some cache line still has `V=0`, a MISS simply fills that cache line, and no eviction is needed. Once every cache line has `V=1`, each new MISS must **evict** a cache line to make room for the new data block. The eviction unit chooses which cache line to evict (see the section Eviction).

4a.6.2 The Modified Bit (M)

The modified bit tracks whether the data block in the cache still matches the corresponding data in RAM.

Case 1: The program only uses `lw` (loads). Every data block in the cache is simply a copy of what is in RAM. The cache and RAM are always in agreement. So `M=0` for all cache lines.

Case 2: The program uses `sw` (stores). When `sw` writes data to an address, there are two sub-cases:

- **Cache miss on `sw`:** The data block is first loaded from RAM into the cache. Then the word written by `sw` is placed into the appropriate position within the data block. The modified bit is set to `M=1`, because the data block now differs from RAM.
- **Cache hit on `sw`:** The data block is already in the cache. The word from `sw` is written directly into the data block. Again, `M=1` is set, because the cache now holds data that RAM does not.

In both cases, after a `sw`, the cache has newer data than RAM. We say the cache line is **dirty** (modified).

4a.7 Eviction

When the cache is full, each MISS requires two decisions. First, which cache line should be evicted? Second, what must be done with the data block in that cache line before it is overwritten?

4a.7.1 The Eviction Unit: Approximate LRU

The CPU contains a hardware **eviction unit** that chooses which cache line to evict, on every MISS when the cache is full. How should it choose? Temporal locality suggests the answer. A cache line that was used recently is likely to be used again soon. So such a cache line should stay. A cache line that has not been used for a long time is the best candidate to evict. This policy is called **LRU (Least Recently Used)**: evict the cache line whose most recent use is furthest in the past.

Why not simply evict the *oldest* cache line, meaning the one that was loaded longest ago? (This policy is called **FIFO**, for First In, First Out.) The answer is that the time when a data block was loaded says nothing about whether the program is still using it. Recall the loop `total = total + a[i]` from the section Temporal Locality. In that loop, `total` is loaded before any element of `a`, and yet `total` is used on every iteration. FIFO would evict `total` each time the cache fills up again, causing a needless MISS. In contrast, LRU never evicts `total`, because `total` is always among the most recently used cache lines. Instead, LRU evicts old elements of `a`, which the program will never use again anyway. Temporal locality is about recent *use*, not about when the data arrived. So LRU is the policy that follows from temporal locality.

Exact LRU is expensive to build in hardware. The eviction unit would have to keep all N cache lines sorted by the time of their last use, and update that order on every `lw` and `sw`. So real CPUs use an approximation, called **pseudo-LRU**. Each cache line keeps a few extra bits that record roughly how recently the line was used. The eviction unit then picks a cache line that is *approximately* the least recently used. The exact scheme is proprietary and differs between CPU vendors, but the goal is always the same: avoid evicting a cache line that was used recently.

4a.7.2 Eviction and Write-Back

Once the eviction unit has chosen a cache line, the cache checks that line's modified bit:

- **If `M=0`:** The data block is identical to RAM. It is safe to simply discard the old data block.

- **If `M=1`:** The data block has been modified and RAM has stale data. We must first **write the data block back to RAM** so that RAM is brought up to date.

In either case, the new data block is then copied into the cache line, overwriting the old one, and `V=1`. The new `M` bit depends on what caused the MISS. If the MISS was caused by `lw`, then `M=0`. If the MISS was caused by `sw`, then `M=1`, since the word from `sw` is written into the new data block.

This policy is called **write-back**: modified data is written to RAM only when the cache line is evicted, not on every `sw`. The alternative policy is called **write-through**: every `sw` writes to both the cache and RAM. So RAM is never stale and no M bit is needed. Write-back is often more efficient, because it avoids redundant writes to RAM. Programs often write to the same address multiple times, and also write to several different words of the same data block, before moving on. With write-back, all of those writes cost just one transfer of the data block to RAM, at eviction time.

4a.8 Putting It All Together: A Worked Example

Let us trace a short sequence of `lw` and `sw` instructions through a small fully associative cache with **4 cache lines** and **64-byte data blocks**. The cache starts out empty: every cache line has V=0.

For each access, the "Block" column gives the block-aligned address (the address with its low 6 bits set to zero). As before, the hardware actually stores only the tag (for example, `0x1000 >> 6 = 0x40`), but block addresses are easier to read. To show LRU clearly, we track exactly when each cache line was last used. (Real hardware tracks this only approximately, using pseudo-LRU.)

Step	Access	Block	Result	What the cache does
1	`lw` from `0x0000_1004`	`0x1000`	MISS	Line 0 is empty. Load the block from RAM into line 0. V=1, M=0.
2	`sw` to `0x0000_2008`	`0x2000`	MISS	Load the block into line 1, then write the word into it. V=1, M=1.
3	`lw` from `0x0000_1010`	`0x1000`	HIT	Same data block as step 1. No RAM access.
4	`lw` from `0x0000_3000`	`0x3000`	MISS	Load the block into line 2. V=1, M=0.
5	`lw` from `0x0000_4020`	`0x4000`	MISS	Load the block into line 3. V=1, M=0. The cache is now full.
6	`lw` from `0x0000_5000`	`0x5000`	MISS	Evict line 1 (the LRU line). It has M=1. So first write its block back to RAM. Then load the new block into line 1. M=0.
7	`sw` to `0x0000_1014`	`0x1000`	HIT	Write the word into line 0's data block. M=1.
8	`lw` from `0x0000_2008`	`0x2000`	MISS	Evict line 2 (the LRU line). It has M=0. So simply discard it. Load the block into line 2. M=0.

Here is the cache just before step 6. All four cache lines are valid. So step 6 must evict one of them. (The data blocks are not shown.)

Line	V	M	Address (Key)	Last used	
0	1	0	0x0000_1000	step 3	
1	1	1	0x0000_2000	step 2	← LRU
2	1	0	0x0000_3000	step 4	
3	1	0	0x0000_4000	step 5	

A few steps deserve a closer look:

- **Step 3** is a HIT, even though the program never accessed address `0x1010` before. That address is in the same 64-byte data block as `0x1004`, which was loaded in step 1. This is spatial locality at work.
- **Step 6** shows LRU at work. Line 0 holds the oldest data block (loaded in step 1). However, the program used that block again in step 3. So line 0 is not the least recently used. Line 1 was last used in step 2. Therefore, line 1 is the one evicted. (A FIFO policy would have evicted line 0 instead, and the `sw` in step 7 would then have been a MISS.) Because the `sw` in step 2 set M=1, the cache must write that data block back to RAM before reusing line 1.
- **Step 7** is a HIT on a `sw`. The word is written only into the cache, and M becomes 1. RAM now holds a stale copy of block `0x1000`, which is fine: the cache will write the block back when line 0 is eventually evicted.
- **Step 8** needs block `0x2000` again. That block was evicted in step 6. So this access is a MISS. Because the block was written back in step 6, RAM holds the word stored by the `sw` in step 2. That word is the value that this `lw` receives. Without the write-back, that `sw` would have been lost.

The cache after step 8:

Line	V	M	Address (Key)	Last used
0	1	1	0x0000_1000	step 7
1	1	0	0x0000_5000	step 6
2	1	0	0x0000_2000	step 8
3	1	0	0x0000_4000	step 5

4a.9 Looking Ahead: The Scalability Problem

The fully associative cache is elegant: any address can live in any cache line, and parallel comparators make the lookup fast. However, the fully associative cache has a scalability problem.

If the cache has **N** cache lines, then it needs **N comparators**. Every time the CPU executes `lw` or `sw`, all N comparators fire simultaneously. This has two major costs and one smaller cost:

1. **Energy:** Each comparator consumes power every time the cache is accessed. With hundreds or thousands of cache lines, the energy cost becomes significant.
2. **Chip real estate:** Each comparator takes up physical space on the silicon die. More comparators means less room for other components (or a larger, more expensive chip).
3. **Circuit delay:** The delay for a comparator and other electronics in a fully associative cache is on the order of 1 or 2 CPU clock cycles, and this delay grows as N grows.

For a small cache, these costs are acceptable. But for a **large cache** with thousands of cache lines, the cost of N comparators becomes prohibitive. That is the reason for the **direct-mapped cache** of Chapter 4b. A direct-

mapped cache uses the address itself to choose the one cache line to check. So a direct-mapped cache needs only **one comparator**, no matter how large the cache is. Chapter 4c then describes the **set-associative cache**, which combines the best features of the fully associative and direct-mapped caches.

4a.10 Summary

Concept	Description
Cache	Small, fast memory on the CPU chip that stores recently accessed data
L1, L2, L3	The three levels of CPU cache; each is larger but slower than the one before
Cache line	One entry: address + data block + V bit + M bit
Data block	The data portion of a cache line (typically 64 bytes)
Temporal locality	Programs tend to re-use the same addresses repeatedly (see the section Temporal Locality)
Spatial locality	Programs tend to access addresses near recently used ones (see the section Spatial Locality)
Cache hit	The data block containing the requested address is in the cache (fast)
Cache miss	The data block containing the requested address is not in the cache; must go to RAM (slow)
Comparator	Hardware circuit that checks if two addresses are equal
Fully associative	Any address can go in any cache line; N comparators check in parallel
Block alignment	Data block addresses are always multiples of the block size
Tag	The high bits of an address (26 bits for 64-byte blocks); identifies which data block. The cache stores and compares only the tag
Offset	The low bits of an address (6 bits for 64-byte blocks); selects a byte within the data block
Valid bit (V)	1 = cache line holds valid data; 0 = empty/invalid
Modified bit (M)	1 = data block differs from RAM (dirty); 0 = matches RAM (clean)
Dirty	Informal synonym for "modified": a dirty cache line has M=1, and so its data block is newer than the copy in RAM
Eviction unit	Hardware that chooses which cache line to evict when the cache is full, using pseudo-LRU (an approximation of Least Recently Used)
LRU	Least Recently Used: evict the cache line whose most recent use is furthest in the past
Write-back	Dirty data blocks are written to RAM only upon eviction
Write-through	The alternative to write-back: every `sw` writes to both the cache and RAM

4a.11 Check Your Understanding

1. Why does a cache need the valid bit (V)? What could go wrong right after the machine is powered on, if there were no V bit?
2. Why can no two cache lines ever hold the same tag?
3. Suppose that a cache uses 128-byte data blocks, and addresses are 32 bits. How many bits are in the offset? How many bits are in the tag?
4. On a `sw` that is a cache miss, the cache first loads the entire 64-byte data block from RAM, even though `sw` writes only 4 bytes. Why?
5. In the worked example, suppose that there were a step 9: `lw` from `0x0000_3000`. Is it a HIT or a MISS? If it is a MISS, which cache line is evicted, and is a write-back needed?
6. In the worked example, suppose that the cache used FIFO instead of LRU. Which cache line would step 6 evict? What would then happen at step 7?
7. Why does write-back usually cause fewer writes to RAM than write-through?
8. In Loop 2 of the section Spatial Locality, how many different data blocks does the inner loop touch? Why does this cause nearly every load to be a MISS, on a CPU whose L1 data cache holds 32 KB?
9. Trace the sequence in the table below through a fully associative cache with 4 cache lines, 64-byte data blocks, and LRU eviction. The cache starts out empty. Use a table like the one in the worked example. For each access, give the block address, HIT or MISS, and what the cache does (including any write-back).

Step	Access
1	`lw` from `0x0000_0100`
2	`lw` from `0x0000_0140`
3	`sw` to `0x0000_0104`
4	`lw` from `0x0000_0200`
5	`lw` from `0x0000_0300`
6	`lw` from `0x0000_0144`
7	`lw` from `0x0000_0400`
8	`lw` from `0x0000_0108`