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 Motivation: Why Do We Need a Cache? 1

4a Fully Associative Cache 2
4a.1 The Cache as a Key-Value Store . . . 2
4a.2 Hardware Optimization 1: Parallel Comparison with Comparators . . . 3
4a.3 Hardware Optimization 2: Data Blocks . . . 4
4a.4 The Valid Bit and the Modified Bit . . . 6
4a.5 Looking Ahead: The Scalability Problem . . . 7
4a.6 Summary . . . 8

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.

THIS IS STILL A WORK IN PROGRESS.

TO FILL IN: Start with a brief overview of the three major CPU caches. Also, explain where fully associative, direct-mapped and set associative caches are used.

This chapter and the next three will describe three types of CPU caches (fully associative, direct-mapped, and set associative), and then how a page table works to support virtual memory. These four chapters have some common, unifying elements:

1. The three CPU caches and the page table for virtual memory can all be thought of, conceptually, as simply a **key-value store**.
2. All four mechanisms require **hardware support** to be efficient.
3. In each of the four cases, the central data structure can be thought of as an array of cache or page table entries. The data in each entry (i.e., the data block of a cache line or the page in a virtual memory system) must be **atomic**. This means that the data block or page must be copied as a whole. There is no such thing as a data block or page with some bytes left over as garbage.

We will present a cache as a *key-value store*. 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.

4a 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.

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

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 it is extremely fast – often 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 Fully Associative Cache

4a.1 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 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

4a.1.1 Cache Hit vs. Cache Miss

When the CPU executes `lw` or `sw`, the hardware looks up the address in the cache:

- **Cache HIT:** The address is found in the cache. The data can be read (or written) immediately from the cache. This is fast.
- **Cache MISS:** The address is *not* found in the cache. The CPU must go to RAM to fetch the data. This is slow.

4a.1.2 Example: Two Loads from the Same Address

Consider a program that executes `lw` twice on the same address:

1. **First `lw`:** The address is not yet in the cache – **cache MISS**. The hardware fetches the word from RAM and stores a copy in the cache.
2. **Second `lw`:** The address is now in the cache – **cache HIT**. The hardware returns the word directly from the cache, without going to RAM.

The second access is dramatically faster. This is the fundamental benefit of caching: **exploit the fact that programs tend to re-use the same addresses** (a property known as *temporal locality*).

4a.2 Hardware Optimization 1: Parallel Comparison with Comparators

4a.2.1 The Problem with a Software Lookup

If we implemented our key-value hardware cache naively by loading the stored address for each cache entry using `lw`, then we would have to **compare the requested address against each stored address one at a time**, looping through the entries sequentially. For a cache with N entries, this could take up to N comparison steps. Even for a small cache of just N=64 entries, this would be almost as slow as a direct lookup in RAM.

4a.2.2 The Hardware Solution: Comparators in Parallel

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

The building block is a digital circuit called a **comparator**. A comparator takes two inputs – two addresses – 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:

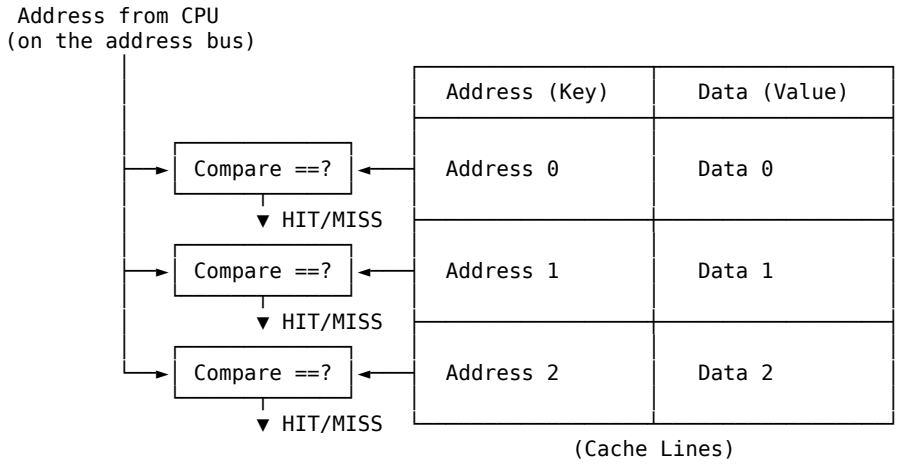
1. The **address from the CPU** (the address requested by `lw` or `sw`).
2. The **address stored in that cache line**.

All comparators operate at almost the same time, because they are all connected to the same electrical signal, the address bus from the CPU.

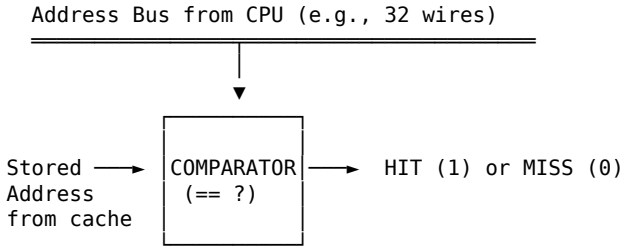
4a.2.3 How the Bus Splits

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 electrical signal to each comparator. This is the same principle as household electrical wiring: 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 signal 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:



4a.2.4 From HIT/MISS Signals to Data

Each comparator produces its own HIT/MISS signal. Since an address can be stored in at most one cache line, at most one comparator can output 1. 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 is what makes a fully associative cache fast – and it is something only hardware can do efficiently. The term **"fully associative"** means that any address can be stored in any cache line, and we check all lines simultaneously.

4a.3 Hardware Optimization 2: Data Blocks

4a.3.1 From Words to Blocks

So far, each cache line stores a single word (4 bytes) of data. But we can do better. Instead of storing just one word per cache line, we can store a larger chunk of contiguous memory – for example, **32 bytes** (8 words). We call this larger chunk a **data block**.

Why does this help? Programs exhibit a property called *spatial locality*: if a program accesses address `X`, it is very likely to access nearby addresses (`X+4`, `X+8`, etc.) soon after. By loading an entire block of 32 bytes into the cache at once, we preload the nearby data that the program is likely to need next.

4a.3.2 The Key Insight: A Data Block is Atomic

The critical idea behind data blocks is:

> **A data block is atomic.**

This means the entire 32-byte block is transferred as a single unit between RAM and the cache (but transferred as a *burst* of 4 back-to-back transfers on a 64-bit bus). This burst allows all 32 bytes to be transferred in a single burst: 64 electric wires (64 bits * 4-cycle burst = 32 bytes). (In fact, we use 32 bytes for ease of examples, but a DDR channel of RAM uses an 8-way burst of 64 bits.)

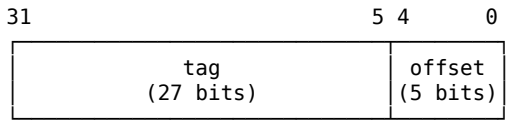
4a.3.3 Block-Aligned Addresses

When we use 32-byte data blocks, the addresses stored in the cache must always be **aligned** to 32-byte boundaries. That is, every address in the cache is divisible by 32.

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

1. Compute the **block-aligned address**: `BLOCK_ADDR = ADDR - (ADDR % 32)`. This rounds `ADDR` down to the nearest multiple of 32.
2. Fetch the **32 bytes** starting at `BLOCK_ADDR` from RAM.
3. Store `BLOCK_ADDR` as the address (key) in the new cache line, and store the 32 bytes as the data block (value).

In hardware, no arithmetic is needed for step 1. Since $32 = 2^5$, the value `ADDR % 32` is simply the low 5 bits of `ADDR`, and `BLOCK_ADDR` is just `ADDR` with those low 5 bits set to zero. So the hardware views a 32-bit address as two fields:



- The **offset** (low 5 bits) says which byte *within* the data block we want.
- The **tag** (high 27 bits) identifies *which* data block we want.

Every block-aligned address has its low 5 bits equal to zero, so there is no point in storing or comparing them. The cache stores only the 27-bit **tag** as the key of each cache line, and each comparator compares only 27 bits, not 32. From now on, when we say "the address stored in a cache line," remember that the hardware actually stores only the tag.

For example, if `ADDR = 0x0000_0044` (which is 68 in decimal): - `68 % 32 = 4`, so `BLOCK_ADDR = 68 - 4 = 64 = 0x0000_0040`. - The cache fetches bytes 64 through 95 from RAM (32 bytes total). - In binary, the low 5 bits of `0x44` are `00100`, so the offset is 4. The tag is `0x44 >> 5 = 2`. - 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, …, 92, every one of them is a cache HIT. Those words were already brought into the cache, as part of the same data block.

Here is a diagram of the cache with data blocks:

Address (Key) (block-aligned)	Data Block (Value) (32 bytes)
0x0000_0040	[byte 0] [byte 1] ... [byte 31]
0x0000_3F00	[byte 0] [byte 1] ... [byte 31]
0x0000_2C60	[byte 0] [byte 1] ... [byte 31]
0x0000_7B20	[byte 0] [byte 1] ... [byte 31]

Notice that in every address in the "Address" column, the last two hex digits are `00`, `20`, `40`, `60`, `80`, `A0`, `C0`, or `E0`. These are all multiples of 32 (0x20). Equivalently, the low 5 bits of each address are zero. This is what "32-byte aligned" means.

4a.3.4 Cache Hits with Data Blocks

With data blocks, we must refine what a cache HIT means. A HIT no longer means that the exact address `ADDR` is stored in the cache. A HIT means that the **data block containing `ADDR`** is in the cache. The hardware takes the tag of `ADDR` (equivalently, its block-aligned address) and compares that against the tags stored in the cache lines.

For example, suppose `lw` on address `0x44` caused the block at `0x40` to be loaded. A later `lw` on address `0x58` is a HIT, even though the program never accessed `0x58` before: `0x58` rounds down to the block address `0x40`, which is already in the cache.

4a.4 The Valid Bit and the Modified Bit

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 32 bytes).

V	M	Address (Key)	Data Block (32 bytes)
1	0	0x0000_0040	[byte 0] ... [byte 31]
1	1	0x0000_3F00	[byte 0] ... [byte 31]
0	0	(ignored)	(ignored)
1	0	0x0000_7B20	[byte 0] ... [byte 31]

4a.4.1 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`.

Eventually, the cache fills up – all lines have `V=1`. At that point, if a cache miss occurs and we need to load new data, we must **evict** an existing cache line to make room. A common eviction policy is **LRU (Least Recently Used)**: the cache line that has not been accessed for the longest time is the one chosen for eviction. This policy is effective because it keeps the most recently used data in the cache, which is exactly the data the program is most likely to need again. (Actually, vendors implement a proprietary pseudo-LRU cache.)

4a.4.2 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. This policy of loading the data block on a store miss is called **write-allocate**. Why must we load the whole data block from RAM, just to overwrite 4 bytes of it? Because the data block is atomic. The cache cannot hold a data block in which 4 bytes are valid and the other 28 bytes are garbage.
- **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.4.3 Eviction and Write-Back

When a cache line must be evicted to make room for new data, the cache checks the 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.5 Looking Ahead: The Scalability Problem

The fully associative cache is elegant: any address can live in any cache line. And yet, parallel comparators make the cache lookup fast. However, it 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. In the more complex caches to be studied next, circuit delay can be up to 4 CPU clock cycles.

For small caches (e.g., 16-64 lines), the fully associative design works well and is sometimes used in practice. The most important example is the **TLB** (translation lookaside buffer): a small, fully associative cache of page table entries, which we will see in the chapter on virtual memory. But for **large caches** (thousands of lines), the cost of N comparators becomes prohibitive.

In the next chapter, we will study a modification called the **direct-mapped cache**. The key idea is to use the address itself to determine *which* cache line to check, so that only **one comparator** is needed instead of N. This dramatically reduces both energy consumption and chip area, at the cost of somewhat less flexibility in where data can be stored. That loss of flexibility has a price: two frequently used data blocks may be forced into the same cache line, so that each one keeps evicting the other. These extra misses are called **conflict misses**. Later, we will see that real CPU caches use a compromise between the two designs, called a **set-associative cache**, in which each data block can be placed in any one of a small number of cache lines (typically 4 to 16).

4a.6 Summary

Concept	Description
Cache	Small, fast memory on the CPU chip that stores recently accessed data
Cache line	One entry: address + data block + V bit + M bit
Data block	The data portion of a cache line (e.g., 32 bytes)
Temporal locality	Programs tend to re-use the same addresses repeatedly (see the section on key-value store)
Spatial locality	Programs tend to access addresses near recently used ones (see the section on data blocks)
Cache HIT	Requested address found in cache (fast)
Cache MISS	Requested address not in 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
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)
LRU eviction policy	Evict the least recently used cache line when cache is full

Concept	Description
Write-back	Dirty data blocks are written to RAM only upon eviction