Chapter 4b: Direct-Mapped 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

4b Lecture Notes: The Direct-Mapped Cache **1**
4b.1 Motivation: The Scalability Problem of Fully Associative . . . 1
4b.2 Splitting the Address into Three Fields . . . 2
4b.3 Hardware Optimization: One Comparator Instead of N . . . 2
4b.4 Walking Through a Full Example . . . 4
4b.5 The Cost of Direct Mapping: Conflict Misses . . . 5
4b.6 The Valid Bit and Modified Bit . . . 5
4b.7 Direct-Mapped vs. Fully Associative: A Comparison . . . 6
4b.8 Summary . . . 6

Copyright (c) 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.

4b Lecture Notes: The Direct-Mapped Cache

4b.1 Motivation: The Scalability Problem of Fully Associative

Recall from Chapter 4a that the fully associative cache uses **N comparators** – one per cache line – all firing in parallel on every memory access. For small caches this is fine, but for large caches with thousands of lines, the cost in chip area and energy becomes prohibitive.

The root cause of this problem is *flexibility*: in a fully associative cache, any address can live in any cache line, so there is no way to know *which* line to check without checking *all* of them.

The **direct-mapped cache** attacks this problem head-on with a simple insight:

> **Use the address itself to decide exactly which cache line to check.**

If we can determine the correct line from the address alone – before doing any comparison – then only **one comparator** is ever needed, no matter how large the cache. This is the defining idea of the direct-mapped cache.

4b.2 Splitting the Address into Three Fields

4b.2.1 The Setup

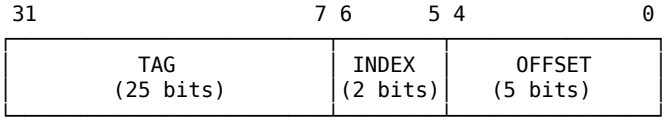
Suppose our cache has **4 lines** (we will generalize shortly), each holding a **32-byte data block**, and addresses are **32 bits** wide.

Because each data block is 32 bytes (= 2^5 bytes), we need **5 bits** to index into the block and select a specific byte. We call these the **offset** bits.

Because the cache has 4 lines (= 2^2 lines), we need **2 bits** to select one of the 4 lines. We call these the **index** bits.

The remaining **32 - 5 - 2 = 25 bits** are called the **tag**.

We split every address as follows (from most significant to least significant bit):



Notice that the index bits are *not* stored in the cache. A block sitting in line 2 must have index `10`, because line 2 is the only place it is allowed to go. The line's position supplies the index, the stored tag supplies the high-order bits, and the offset selects a byte within the block – together they reconstruct the full address. So each line needs to store only a 25-bit tag. By comparison, a fully associative cache with 32-byte blocks must store a 27-bit tag (every bit except the offset), since a block could be in any line.

4b.2.2 Generalizing

In general, for a direct-mapped cache with **2^k lines** and **2^b bytes** per block, and **A-bit** addresses:

- **Offset:** b bits (selects a byte within the data block)
- **Index:** k bits (selects a cache line)
- **Tag:** A - k - b bits (the remaining bits, stored in the cache for comparison)

4b.3 Hardware Optimization: One Comparator Instead of N

4b.3.1 How the Index Selects a Line

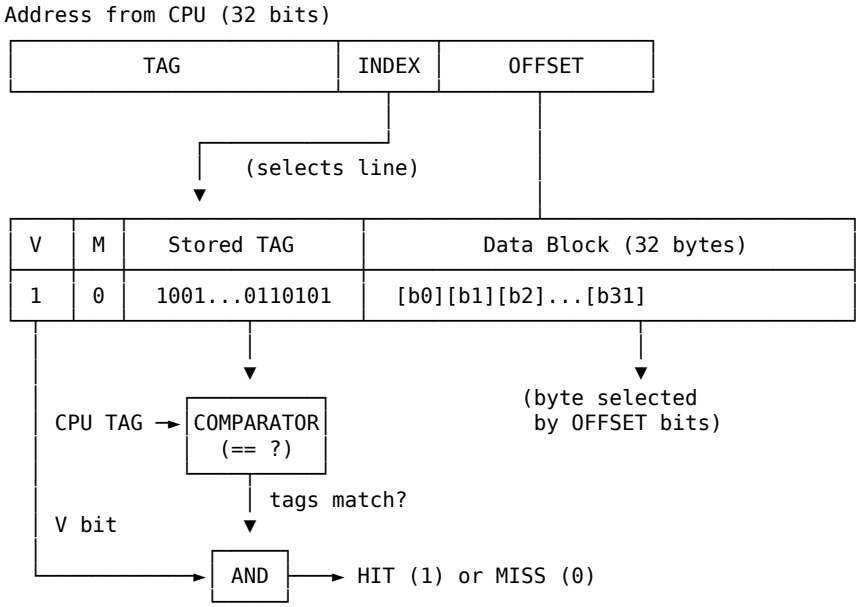
When the CPU executes `lw` or `sw` with some address `ADDR`, the hardware does the following:

1. **Extract the index field** from `ADDR` (bits k+b-1 down to b).
2. **Go directly to that cache line** – no search, no loop, no parallel firing of multiple comparators.
3. **Check the valid bit and compare only the tag field** of `ADDR` against the tag stored in that one cache line. It is a hit only if V=1 *and* the tags match.

Step 2 is the key: the index bits act like an array subscript. Just as `array[i]` retrieves the i-th element in one step, the index bits retrieve exactly one cache line in one step.

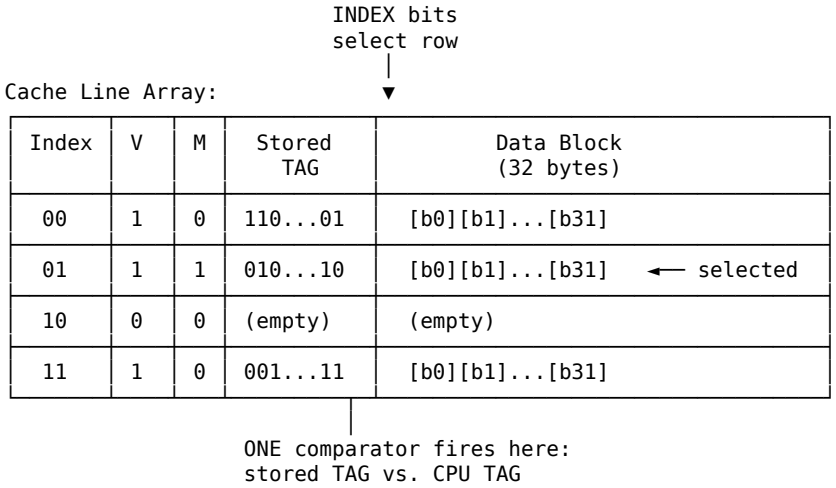
Step 3 then requires only **one comparator**. That comparator does not belong to any particular line; it is shared by all of them. The index bits drive a selector circuit (a *multiplexer*) that routes the stored tag of the chosen line – and only that line – to the comparator's input. This is a dramatic reduction from the N comparators of the fully associative cache.

4b.3.2 Diagram: Direct-Mapped Cache Lookup



The CPU's tag field is sent to the comparator, which compares it to the tag stored in the line that the index selected. The comparator's output and the line's V bit feed an AND gate: the result is a hit only if the tags match *and* V=1; otherwise it is a miss.

A fuller picture, showing the full cache array:



Key insight: The number of comparators does not grow with the cache size. A direct-mapped cache with 1024 lines still uses only **one comparator**, not 1024.

A second benefit: speed. Because there is only one candidate line, the hardware can start reading that line's data block *at the same time* as the tag comparison. If the access turns out to be a hit, the data is already on its way to the CPU. A fully associative cache must first find out which of its N comparators matched before it can select the corresponding data block, which adds delay to every hit.

4b.4 Walking Through a Full Example

Let us trace through a concrete example. Assume: - 32-bit addresses - Cache has **4 lines** (index = 2 bits) - **32-byte** data blocks (offset = 5 bits) - Tag = 25 bits

4b.4.1 Starting State

The cache is empty: all V bits are 0.

Index	V	M	TAG	Data Block
00	0	0	(empty)	(empty)
01	0	0	(empty)	(empty)
10	0	0	(empty)	(empty)
11	0	0	(empty)	(empty)

4b.4.2 Access 1: `lw` from address `0x0000_0044`

Address in binary: `0000 0000 0000 0000 0000 0000 0100 0100`

Split the fields: - **Tag** (bits 31-7): `0000 0000 0000 0000 0000 0000 0` = `0x0000000` - **Index** (bits 6-5): `10` = 2 - **Offset** (bits 4-0): `00100` = 4

The hardware goes to line **2** (index `10`). V=0 there, so this is a **MISS** regardless of the tag.

The hardware: 1. Computes the block-aligned address: offset bits become 0, giving `0x0000_0040`. 2. Fetches 32 bytes (addresses `0x40` through `0x5F`) from RAM. 3. Stores the tag `0x0000000` and the 32-byte block in line 2, sets V=1, M=0. 4. Returns the 4-byte word beginning at byte offset 4 within the block (bytes `0x44` through `0x47`).

Cache after Access 1:

Index	V	M	TAG	Data Block
00	0	0	(empty)	(empty)
01	0	0	(empty)	(empty)
10	1	0	0000000	[0x40][0x41]...[0x5F]
11	0	0	(empty)	(empty)

4b.4.3 Access 2: `lw` from address `0x0000_0048`

- **Tag:** `0x0000000`
- **Index:** `10` = 2
- **Offset:** `01000` = 8

Go to line **2**. V=1, and the stored tag `0x0000000` matches the CPU tag `0x0000000` – this is a **HIT**. Return the 4-byte word beginning at byte offset 8 within the block (bytes `0x48` through `0x4B`). No RAM access needed.

This illustrates spatial locality: address `0x48` had not been accessed before, but it arrived for free inside the block fetched during Access 1.

4b.5 The Cost of Direct Mapping: Conflict Misses

4b.5.1 A New Kind of Miss

In a fully associative cache, any address can live in any line, so the cache is "full" only when all N lines are occupied. In a direct-mapped cache, two different addresses may have **the same index bits**, meaning they are forced to compete for the same single cache line.

When this happens, one address evicts the other – even if other cache lines are completely empty. This is called a **conflict miss** (sometimes called a **collision miss**). It is a miss caused not by the cache being full, but by two addresses mapping to the same line.

4b.5.2 A Concrete Example of Thrashing

Suppose our 4-line cache is used by a program that alternates between two addresses:

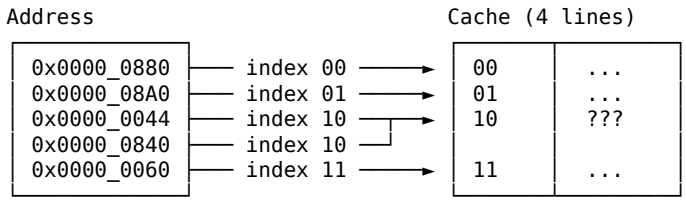
- Address A = `0x0000_0044`: Tag = `0x0000000`, Index = `10`
- Address B = `0x0000_0840`: Tag = `0x0000010`, Index = `10`

Both addresses have index `10`, so both map to **line 2**.

```
Access A:  line 2 is empty  ──► MISS; load A into line 2
Access B:  line 2 has A     ──► MISS; evict A, load B into line 2
Access A:  line 2 has B     ──► MISS; evict B, load A into line 2
Access B:  line 2 has A     ──► MISS; evict A, load B into line 2
...
```

Every single access is a miss! Lines 0, 1, and 3 sit completely empty the whole time, but the direct-mapped design cannot use them for this workload. This repeated eviction pattern is called **cache thrashing**. It is the principal weakness of the direct-mapped cache.

4b.5.3 Visualizing the Conflict




```
Addresses A (0x0000_0044) and B (0x0000_0840) fight over line 10;
lines 00, 01, 11 are unused in this example.
```

4b.6 The Valid Bit and Modified Bit

The valid bit and modified bit work exactly as in the fully associative cache (see Chapter 4a, Section 5), with one small difference in how eviction works.

In a fully associative cache, LRU eviction picks the *least recently used* line across all N lines. In a direct-mapped cache, there is no choice to make: a miss at index `i` **always** evicts whatever is currently in line `i`. The "eviction policy" is determined entirely by the address.

The write-back rule is unchanged:

- **If M=0:** The evicted block matches RAM; simply discard it and reuse the line.
- **If M=1:** The evicted block is dirty; write it back to RAM first, then reuse the line.

4b.7 Direct-Mapped vs. Fully Associative: A Comparison

Property	Fully Associative	Direct-Mapped
Comparators needed	N (one per line)	1
Energy per access	High (all N comparators fire)	Low (only 1 comparator fires)
Hit latency	Slower (must find which line matched, then select its data)	Faster (data is read in parallel with the single tag check)
Chip area for comparators	Proportional to N	Constant
Placement flexibility	Any address → any line	Address → exactly one line
Conflict misses possible?	No	Yes
Thrashing possible?	Yes, but only when a program cycles through more than N blocks	Yes, even when most of the cache is empty
Eviction policy	LRU (or similar)	Forced by index (no choice)
Best for	Small caches, where N comparators are affordable	Caches where fast hits and simple hardware matter most

---

4b.8 Summary

Concept	Description
Direct-mapped cache	Each address maps to exactly one cache line, determined by the index bits
Address fields	Every address is split into TAG | INDEX | OFFSET (high to low bits)
Offset bits	Select a byte within the data block; b bits for a 2^b-byte block
Index bits	Select which cache line to examine; k bits for a 2^k-line cache
Tag bits	The remaining high-order bits; stored in the cache and compared on access
One comparator	Only the single selected line's tag is compared – N comparators not needed
Conflict miss	Two addresses with the same index bits compete for the same line
Thrashing	Repeated conflict misses as two addresses alternate evicting each other

Concept	Description
Eviction in direct-mapped	Always evicts whatever is in the indexed line – no LRU choice
Write-back	Dirty (M=1) blocks are written to RAM upon eviction (same as fully associative)