Chapter 4b: Direct-Mapped Cache

Gene Cooperman

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.

1 Lecture Notes: The Direct-Mapped Cache

1.1 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.

1.2 Splitting the Address into Three Fields

1.2.1 The Setup

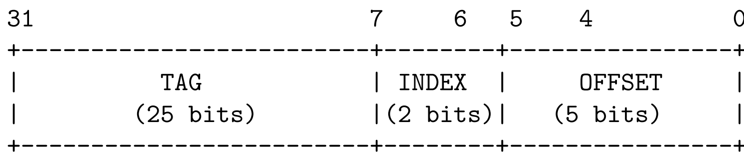
Suppose our cache has **4 lines** (we will generalize shortly), each holding a **32-byte data block**, and addresses are **32 bits** wide.

Because each data block is 32 bytes (= 2ˆ5 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 (= 2ˆ2 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):



1.2.2 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)

1.3 HARDWARE OPTIMIZATION: One Comparator Instead of N

1.3.1 How the Index Selects a Line

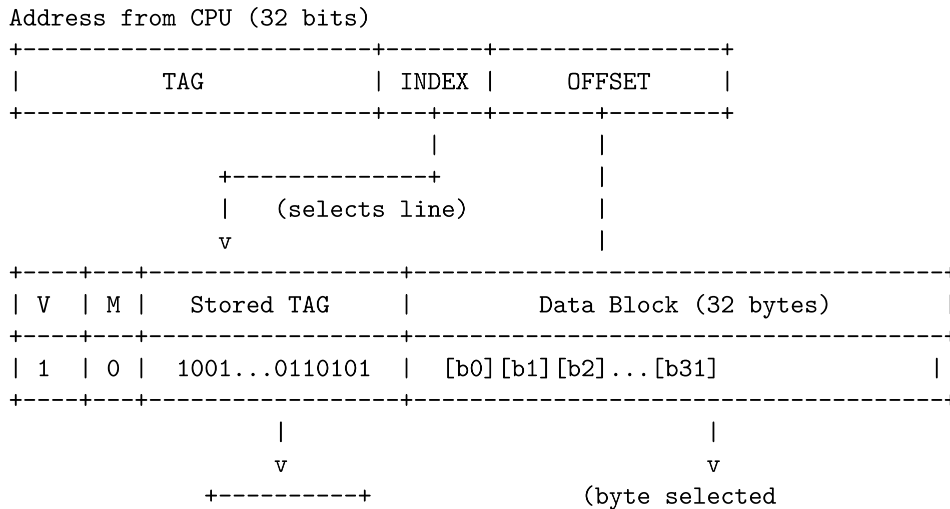
When the CPU executes `lw` or `sw` with some address `ADDR`, the hardware does the following:

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. **Compare only the tag field** of `ADDR` against the tag stored in that one cache line.

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** – the single comparator attached to the selected line. This is a dramatic reduction from the N comparators of the fully associative cache.

1.3.2 Diagram: Direct-Mapped Cache Lookup



```
                 |          |                by OFFSET bits)
CPU TAG  ---> |COMPARATOR| ---> HIT (1) or MISS (0)
                 | (== ?)   |
                 +----------+
```

The CPU's tag field is sent to the comparator. The comparator compares it to the tag stored in the line that the index selected. If they match and V=1, it is a hit; otherwise it is a miss.

A fuller picture, showing the full cache array:

```
                             INDEX bits
                             select row
                                 |
Cache Line Array:                v
+-------+----+---+-----------+------------------------------------+
| Index | V  | M |  Stored   |          Data Block                |
|       |    |   |    TAG    |          (32 bytes)                |
+-------+----+---+-----------+------------------------------------+
|  00   | 1  | 0 | 110...01  |  [b0][b1]...[b31]                  |
+-------+----+---+-----------+------------------------------------+
|  01   | 1  | 1 | 010...10  |  [b0][b1]...[b31]   <-- selected |
+-------+----+---+-----------+------------------------------------+
|  10   | 0  | 0 | (empty)   |  (empty)                           |
+-------+----+---+-----------+------------------------------------+
|  11   | 1  | 0 | 001...11  |  [b0][b1]...[b31]                  |
+-------+----+---+-----------+------------------------------------+
                           |
                 ONE comparator fires here:
                 stored TAG vs. CPU TAG
```

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.

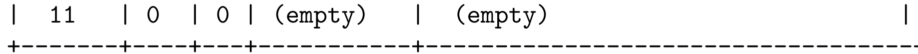
1.4 Walking Through a Full Example

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

1.4.1 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)                           |
+-------+----+---+-----------+------------------------------------+
```

1.4.2 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 010` = `0x0000002` - **Index** (bits 6-5): `00` = 0 - **Offset** (bits 4-0): `00100` = 4

The hardware goes to line **0**. 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 `0x0000002` and the 32-byte block in line 0, sets V=1, M=0. 4. Returns the word at offset 4 within the block (address `0x44`).

Cache after Access 1:

```
+-------+----+---+-----------+------------------------------------+
| Index | V  | M |    TAG    |            Data Block              |
+-------+----+---+-----------+------------------------------------+
|  00   | 1  | 0 | 0000002   |  [0x40][0x41]...[0x5F]             |
+-------+----+---+-----------+------------------------------------+
|  01   | 0  | 0 | (empty)   |  (empty)                           |
+-------+----+---+-----------+------------------------------------+
|  10   | 0  | 0 | (empty)   |  (empty)                           |
+-------+----+---+-----------+------------------------------------+
|  11   | 0  | 0 | (empty)   |  (empty)                           |
+-------+----+---+-----------+------------------------------------+
```

1.4.3 Access 2: `lw` from address `0x0000_0048`

- **Tag:** `0x0000002`
- **Index:** `00` = 0
- **Offset:** `01000` = 8

Go to line **0**. V=1, and the stored tag `0x0000002` matches the CPU tag `0x0000002` – this is a **HIT**. Return the word at offset 8 within the block (address `0x48`). No RAM access needed.

This illustrates spatial locality: address `0x48` was never explicitly loaded, but it arrived for free inside the block fetched during Access 1.

1.5 The Cost of Direct Mapping: Conflict Misses

1.5.1 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.

1.5.2 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 = `0x0000002`, Index = `00`
- Address B = `0x0000_0840`: Tag = `0x0000042`, Index = `00`

Both addresses have index `00`, so both map to **line 0**.

```
Access A:  line 0 is empty  --> MISS; load A into line 0
Access B:  line 0 has A     --> MISS; evict A, load B into line 0
Access A:  line 0 has B     --> MISS; evict B, load A into line 0
Access B:  line 0 has A     --> MISS; evict A, load B into line 0
...
```

Every single access is a miss! Lines 1, 2, 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.

1.5.3 Visualizing the Conflict

```
Address space                    Cache (4 lines)
+-----------+                    +-------+---------+
| 0x0000040 |--- index 00 --\--> | 00    |  ???    |
| 0x0000840 |--- index 00 --/    +-------+---------+
| 0x0000060 |--- index 01 -----> | 01    |  ...    |
| 0x0000880 |--- index 10 -----> | 10    |  ...    |
| 0x00008A0 |--- index 11 -----> | 11    |  ...    |
+-----------+                    +-------+---------+

Two addresses fighting over line 00;
lines 01, 10, 11 are unused in this example.
```

1.6 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.

1.7 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)
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?	No	Yes
Eviction policy	LRU (or similar)	Forced by index (no choice)
Best for	Small, high-value caches	Large caches where cost matters

---

1.8 Digging Deeper: Set-Associative Caches

The fully associative cache and the direct-mapped cache sit at opposite ends of a spectrum:

- **Fully associative:** Maximum flexibility, N comparators.
- **Direct-mapped:** Minimum comparators (1), but conflict misses.

There is a natural middle ground: the **set-associative cache**. In a set-associative cache, the index bits select not a single line but a small **set** of lines – say, 2 or 4. Within the selected set, the address can go into any line, and a small number of comparators (one per line in the set) check all lines in the set simultaneously.

A 2-way set-associative cache with 4 sets uses **2 comparators per access** (not 8, as a fully associative cache with 8 lines would), but almost entirely eliminates the thrashing problem. This is why nearly all real processors use set-associative caches: they are a practical compromise between the two extremes.

We will not discuss set associative caches in these notes.

1.9 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

Concept	Description
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
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)
Set-associative	Middle ground: index selects a *set* of lines; small number of comparators per access