Memory Hierarchy: Caches and Virtual Memory

David Castro-Perez

November 10, 2025

Contents

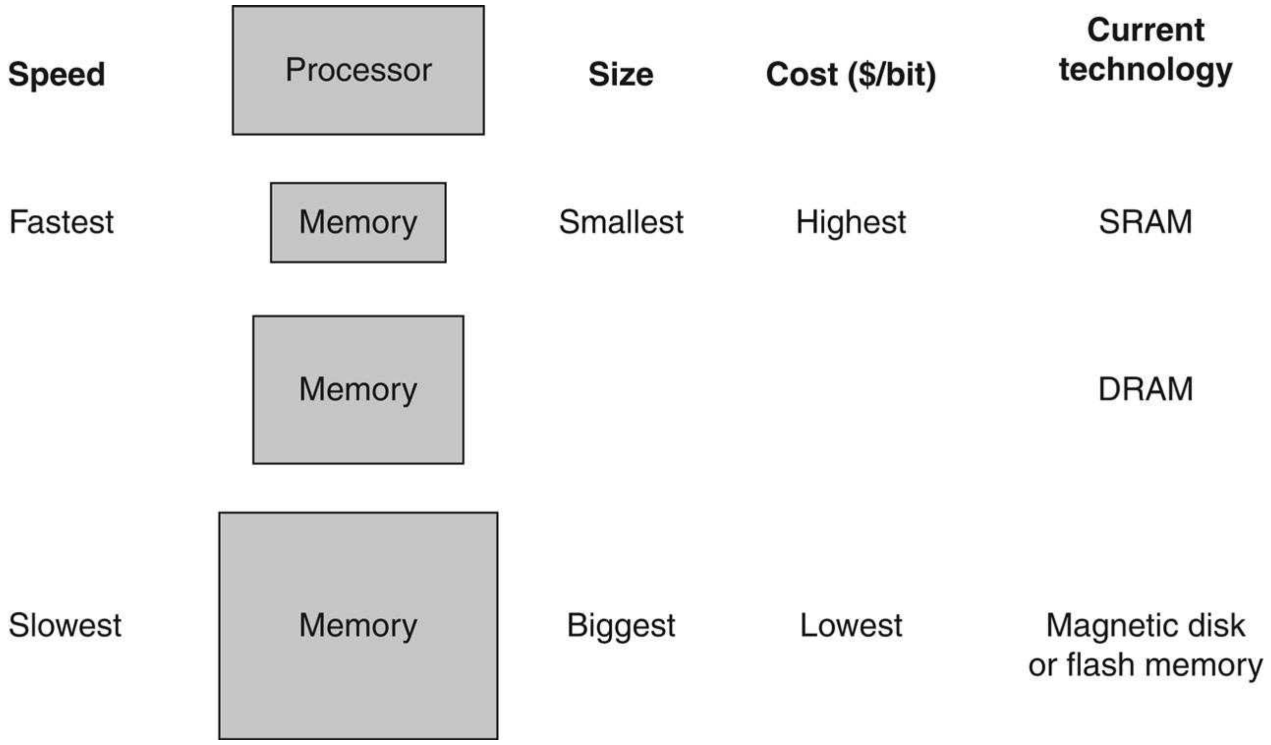


Figure 1: **The basic structure of a memory hierarchy**. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. (Fig 5.1 in [Patterson & Hennessy])

Based on (must-read): Patterson & Hennessy, *Computer Organization and Design, MIPS Edition*, Chapter 5: "Large and Fast: Exploiting Memory Hierarchy".

1 The Memory Hierarchy in Context

1.1 Definition and Motivation

A **memory hierarchy** is a structure that uses multiple levels of memory. As the distance from the processor increases, the size of each level increases, the access time also increases, and the cost per bit decreases. This arrangement allows the computer to appear as though it has both large capacity and high speed.

The performance gap. Since the 1980s, processor clock rates have improved by several orders of magnitude, while DRAM latency has improved only modestly. A modern CPU core can execute billions of operations per second, but each main-memory access may take hundreds of cycles. Without some intermediate storage, the processor would spend most of its time waiting for data.

An intuitive analogy. Imagine a chef (the CPU) preparing meals in a kitchen. If every ingredient were stored in a warehouse miles away (the main memory), the chef would spend most of the day walking back and forth rather than cooking. To stay productive, the chef keeps the most commonly used ingredients within arm's reach (registers and caches), less frequently used items in a nearby pantry (main memory), and rare ingredients in the distant warehouse (disk or SSD). The closer the storage, the faster it is-but the less it can hold.

Why a hierarchy works. Even though only a small fraction of the total data fits in the top levels, programs often reuse the same data and instructions repeatedly. If those active portions can

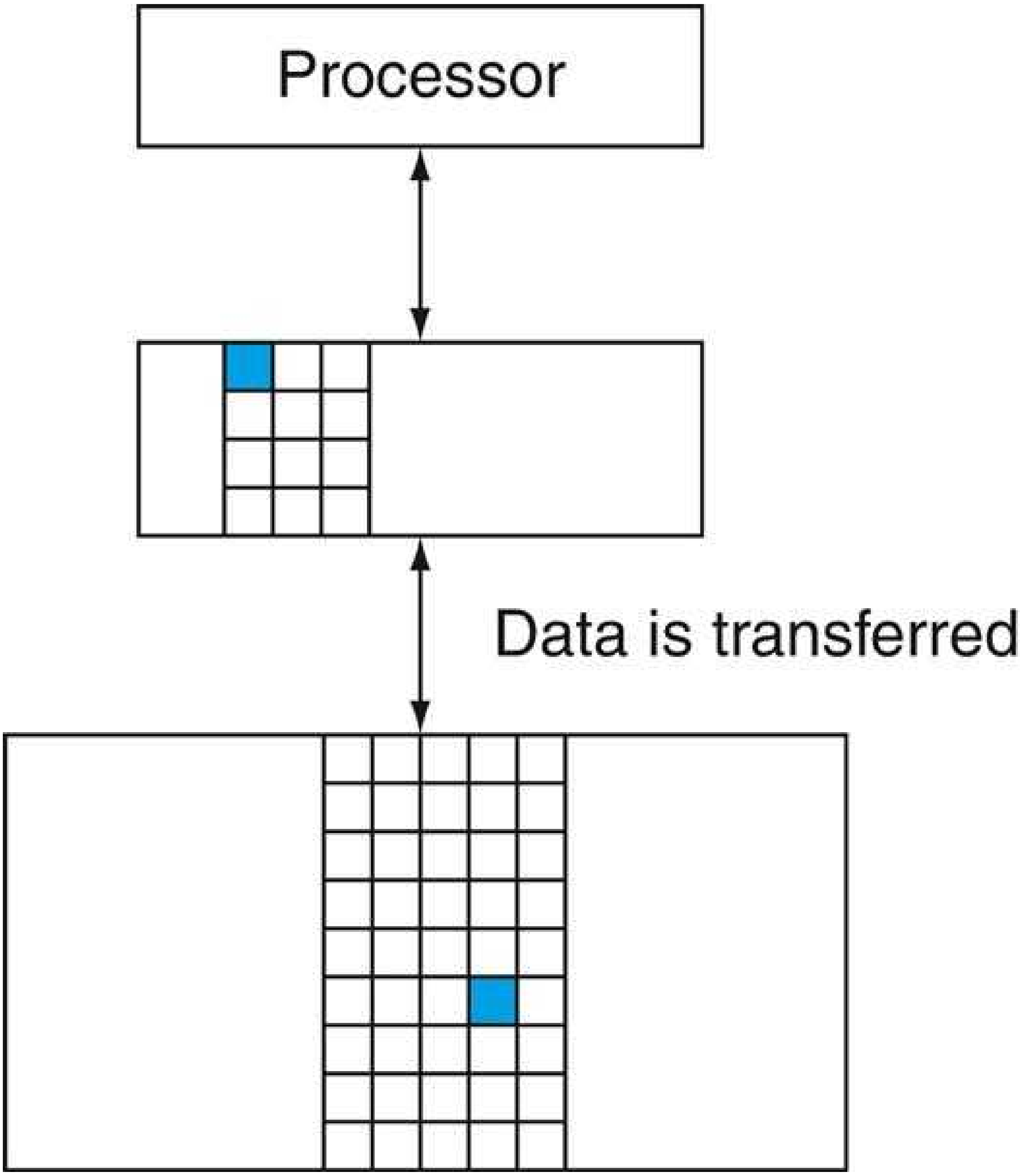


Figure 2: **Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level**. Within each level, the unit of information that is present or not is called a block or a line. Usually we transfer an entire block when we copy something between levels. (Fig 5.2 in [Patterson & Hennessy])

stay near the processor, the system will behave as if all memory were fast.

1.2 The Principle of Locality

The idea that makes the hierarchy work is the **principle of locality**. Empirically, programs do not access memory randomly; instead, they tend to use the same data and nearby data for extended periods.

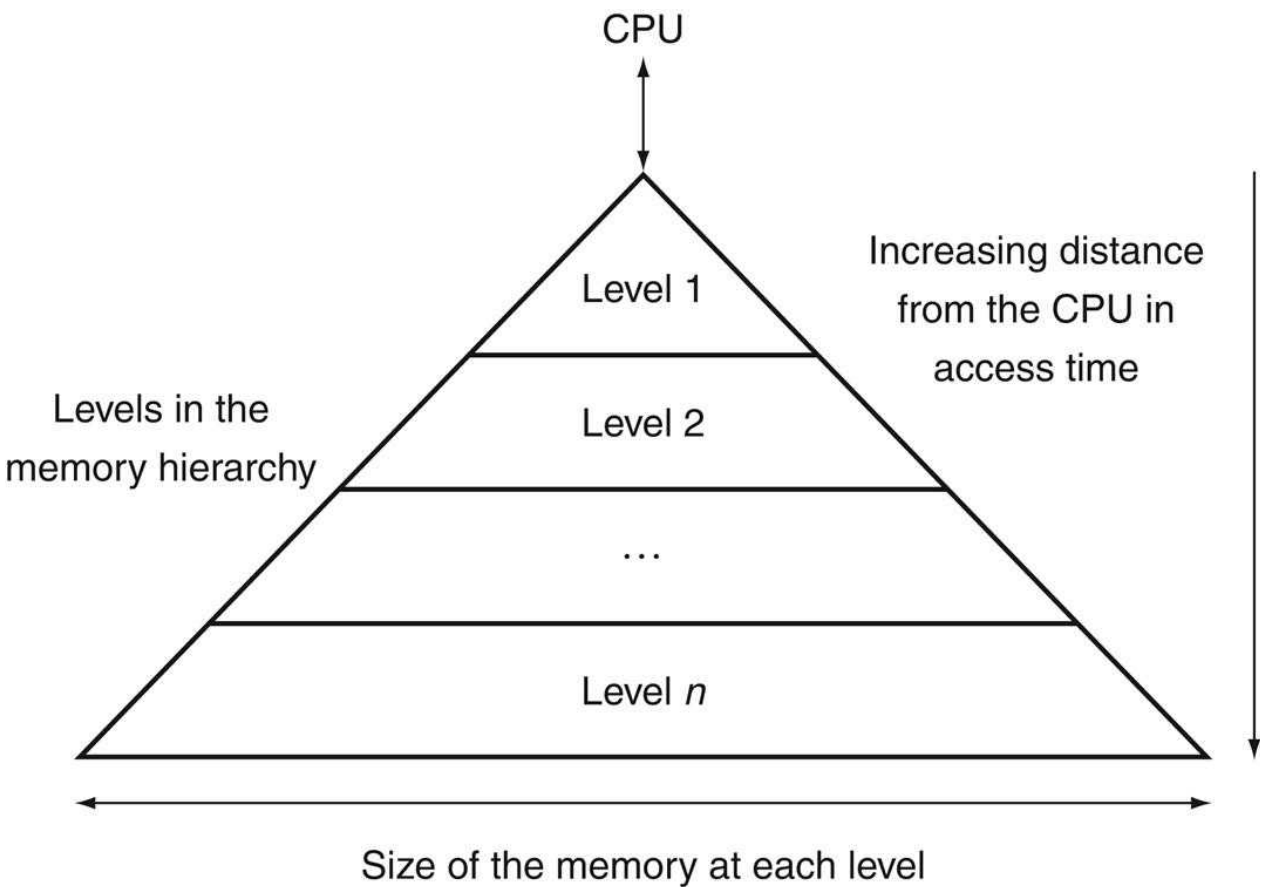

CPU
Increasing distance from the CPU in access time
Levels in the memory hierarchy
Level 1
Level 2
...
Level n
Size of the memory at each level

Figure 3: This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size.This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level n. Maintaining this illusion is the subject of Week 6 lectures (Fig 5.3 in [Patterson & Hennessy])

Temporal locality If a location is referenced at one time, it is likely to be referenced again soon. Typical examples include variables reused within a loop, recently called procedures, or elements of a stack frame that are touched repeatedly.

Spatial locality If a location is referenced, nearby locations are likely to be referenced soon. For instance, instructions are fetched sequentially, and arrays are traversed element by element.

Both forms of locality imply that recently and nearby used information is far more valuable than distant information. This is the key reason that small, fast memories can dramatically accelerate computation.

1.3 Structure of the Hierarchy

Each level of the hierarchy acts as a *cache* for the level below it. The processor always interacts with the highest level (registers and caches), and only if data is missing does it access the next level.

Levels.

- **Registers:** a few dozen to a few hundred bytes, accessed every clock cycle.
- **Caches:** small SRAM memories (tens of kilobytes to a few megabytes) that automatically keep frequently used data close to the CPU.

- **Main memory (DRAM):** gigabytes of capacity, slower but inexpensive per bit.
- **Secondary storage (SSD, HDD):** terabytes of capacity, but millions of times slower than registers.

Each level exploits locality by storing blocks of data that are likely to be reused soon.

Blocks (or cache lines). Information moves between adjacent levels in fixed-size chunks called **blocks** (or **cache lines**). A block is typically 16 – 128 bytes. When the CPU requests one byte or word, the entire block containing it is fetched into the higher level. If the program later accesses another word from that same block, the request will be satisfied quickly—this is spatial locality in action.

Hits and misses. If the requested data is already present in the current level, we have a **hit**; otherwise, we have a **miss**. The hit rate is the fraction of accesses served by that level, usually 90 – 99 %. The miss penalty is the time required to fetch the data from the next level.

Control of data movement. Movement between levels is automatic and invisible to the programmer. Hardware caches manage blocks using address tags; operating systems manage pages using virtual-memory tables. In both cases, recently used information is retained while less useful data is displaced.

1.4 Quantitative View: Average Memory Access Time (AMAT)

To evaluate how effective a hierarchy is, architects use the **Average Memory Access Time (AMAT)**:

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}.$$

- *Hit time* — the time to access data on a hit (includes tag check).
- *Miss rate* — the fraction of accesses that miss in this level.
- *Miss penalty* — the extra time to retrieve the block from the next level.

For multi-level hierarchies, AMAT is computed recursively: the average access time of one level becomes the miss penalty of the level above.

Worked Example 1.1 — Quantifying the Effect of a Cache

Suppose main memory access takes 100 ns. We add an L1 cache that can serve a hit in 2 ns and has a 95 % hit rate. Then:

$$\text{AMAT} = 2 + 0.05 \times 100 = 7 \text{ ns}.$$

The cache reduces average access time by a factor of about 14.

Two-level AMAT derivation. Let L1 have hit time t_1 and miss rate m_1. Let L2 have hit time t_2 and miss rate m_2 (fraction of L1 misses), and main memory time t_M.

$$\text{AMAT} = t_1 + m_1\Big(t_2 + m_2 \cdot t_M\Big).$$

Example: $t_1 = 1$, $m_1 = 0.05$, $t_2 = 10$, $m_2 = 0.10$, $t_M = 100 \Rightarrow \text{AMAT} = 2.0$ cycles.

Worked Example 1.2 — Adding a Second Level

Now add an L2 cache that handles 90 % of the L1 misses in 10 ns. Only 10 % of L1 misses go to main memory (100 ns):

$$\text{AMAT} = 2 + 0.05(10 + 0.10 \times 100) = 2 + 0.05 \times 20 = 3 \text{ ns}.$$

Adding L2 more than halves the average access time again.

1.5 The Impact on CPU Performance

Every cache miss introduces extra cycles in the processor pipeline. If each instruction on average performs one memory access, we can express the total **cycles per instruction (CPI)** as:

$$\text{CPI}_{\text{total}} = \text{CPI}_{\text{base}} + (\text{misses/instruction}) \times \text{miss penalty}.$$

Example. Base CPI = 1, L1 miss rate = 2 %, miss penalty = 50 cycles:

$$\text{CPI}_{\text{total}} = 1 + 0.02 \times 50 = 2.$$

Even a tiny miss rate doubles total execution time. This is why architectural effort (multi-level caches, prefetching) and software effort (data locality) are both critical.

1.6 Trade-offs in the Hierarchy

Each level balances five competing factors:

- **Latency:** how fast each access is.
- **Bandwidth:** how much data per second can be transferred.
- **Capacity:** total storage size.
- **Cost per bit:** economic efficiency of the technology.
- **Complexity:** hardware or software management overhead.

Fast memories (SRAM) are small and expensive; slow memories (DRAM, SSD) are large and cheap. By combining them, the system behaves like a single large, fast, affordable memory—provided locality holds.

Additional Contents (non-examinable): Why Programmers Should Care

This short section is not examinable. It shows the practical relevance of locality for anyone writing programs.

Even high-level languages are affected by the memory hierarchy. Consider the following Python loop:

```
import time
from array import array

N = 10000
stride = 10000
data = array('d', (float(i) for i in range(stride * N)))
s = 0.0

t0 = time.perf_counter()
```

```
for i in range(N):                  # visit consecutive elements
    s += data[i]
t1 = time.perf_counter()

print("Sequential time (good spatial locality):", t1 - t0, "seconds")

t0 = time.perf_counter()
for i in range(0, stride * N, stride):   # jump by 'stride' each time
    s += data[i]
t1 = time.perf_counter()

print("Strided time (bad spatial locality):", t1 - t0, "seconds")
```

When `stride = 1`, elements are accessed consecutively; each cache line fetched from memory contains many useful values. When `stride = 64`, each access touches a different cache line, forcing many misses. The program may slow down by an order of magnitude even though the algorithm is identical. This is a simple, visible effect of spatial locality on performance.

1.7 Summary

- The **memory hierarchy** combines small, fast, costly memories near the processor with large, slow, cheap memories farther away.
- The hierarchy's effectiveness depends on the **principle of locality**.
- Information moves between levels in fixed-size **blocks** (cache lines).
- Performance is measured by the **Average Memory Access Time (AMAT)**.
- Even small miss rates can significantly slow execution.
- (Non-examinable) Awareness of locality helps programmers write faster, more cache-friendly code.

2 Caches: Organization, Operation, and Performance

2.1 From Concept to Hardware

A **cache** is a small, fast memory built from static RAM (SRAM) that stores copies of recently used portions of main memory (DRAM). It sits between the CPU and main memory and acts as a buffer to exploit the principle of locality:

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}.$$

When the processor issues a load or store:

1. The cache checks if the requested address is already present (a *hit*).
2. If so, data are returned immediately.
3. If not (a *miss*), the entire *block* of memory containing that address is fetched from the next lower level and stored in the cache.

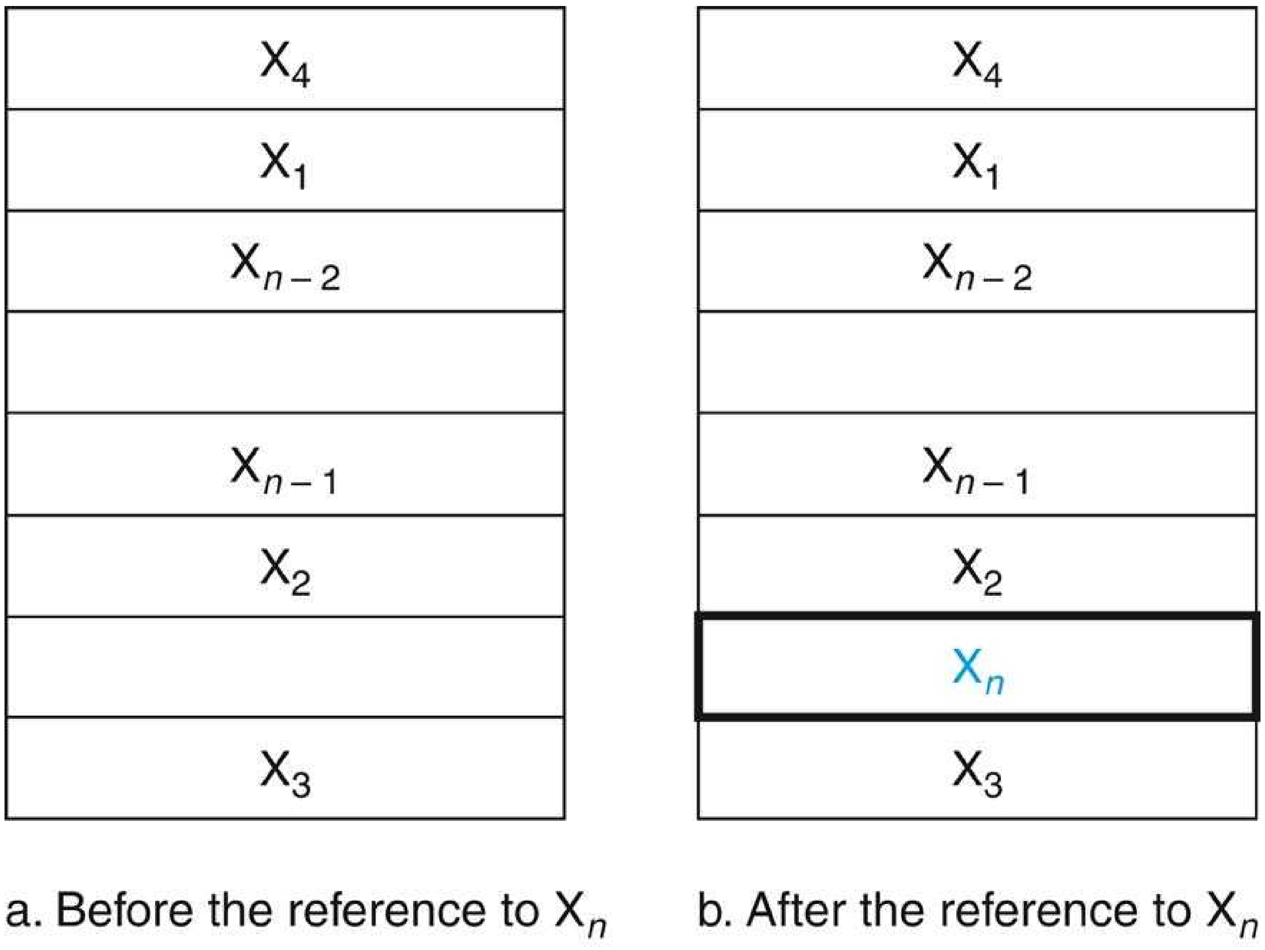


Figure 4: Cache contents immediately before (left) and after (right) an access to a word x_n in memory that is not initially in the cache. The reference causes a *miss* that forces the cache to fetch x_n from memory and insert it into the cache (Fig 5.7 [Patterson & Hennesy]).

EXTRA (NON-EXAMINABLE) SRAM stands for Static RAM, and these are have faster access times (usually 1 cycle). DRAM stands for Dynamic RAM, and these memories have slower access times. SRAMs use 8 transistors per bit of information, whereas DRAMs use capacitors to store a cell values, and one transistor per bit to read or write the value stored by a capacitor. Because DRAMs only use one transistor per bit, they are denser and cheaper than SRAMs. However, since DRAMs contents store data in capacitors, and these eventually lose their charge, they need to be refreshed periodically (i.e. read and written back).

2.2 Cache Structure and Address Fields

A cache holds data in fixed-size **blocks** (or **lines**), each containing B bytes. Each block corresponds to a contiguous region of main memory.

Let

$$C = \text{cache capacity (bytes)}, \qquad B = \text{block size (bytes)}, \qquad A = \text{associativity (ways)}.$$

Then the number of sets is

$$S = \frac{C}{B \times A}.$$

Each cache entry stores:

- a **data block** (B bytes),

- a **tag** (the high-order address bits),
- a **valid bit**,
- optionally a **dirty bit** (for write-back caches).

Address breakdown. Given an m-bit address, the fields are:

$$\underbrace{\text{Tag}}_{\text{identifies memory block}} \mid \underbrace{\text{Index}}_{\text{selects set}} \mid \underbrace{\text{Block offset}}_{\text{locates byte within block}}.$$

$$\text{Offset bits } = \log_2(B), \quad \text{Index bits } = \log_2(S), \quad \text{Tag bits } = m - (\text{offset} + \text{index}).$$

Worked Example 2.1 — Address Breakdown

A 4 KB cache, 16 B blocks, direct-mapped, 32-bit addresses:

$$S = \frac{4096}{16} = 256 = 2^8 \Rightarrow \text{offset} = 4,\ \text{index} = 8,\ \text{tag} = 20.$$

Address 0x12AB34CD → lowest 4 bits = offset, next 8 bits = index, top 20 bits = tag.

We can expand the example to see precisely how the address fields are divided.

Given a 32-bit address `0x12AB34CD`:

$$\texttt{0x12AB34CD} = \texttt{0001 0010 1010 1011 0011 0100 1100 1101}_2$$

We divide the 32-bit address accordingly (from least to most significant bits):

Field	Bit positions	Binary	Hex	Decimal	Meaning
Offset	[3..0]	1101	D	13	Byte within 16 B block
Index	[11..4]	0100 1100	4C	76	Cache line number
Tag	[31..12]	0001 0010 1010 1011 0011	12AB3	—	Identifies memory block

Notes:

- The index crosses a byte boundary: bits [11..8] are the *low* nibble of `0x34` (= `0x4`); bits [7..4] are the *high* nibble of `0xCD` (= `0xC`). Together they form `0x4C`.
- Block base address = `0x12AB34CD` & `0xFFFFFFF0` = `0x12AB34C0`.
- Cache line index = `0x4C` = 76.
- Tag stored in that line = `0x12AB3`.

$$\boxed{\text{Tag} = \texttt{0x12AB3}, \quad \text{Index} = \texttt{0x4C}, \quad \text{Offset} = \texttt{0xD}.}$$

This explicit division will be used repeatedly when analyzing cache hits and misses in Section 3.

2.3 Mapping Strategies: Where Does Each Block Go?

The *mapping* policy determines which cache location(s) can store a given memory block.

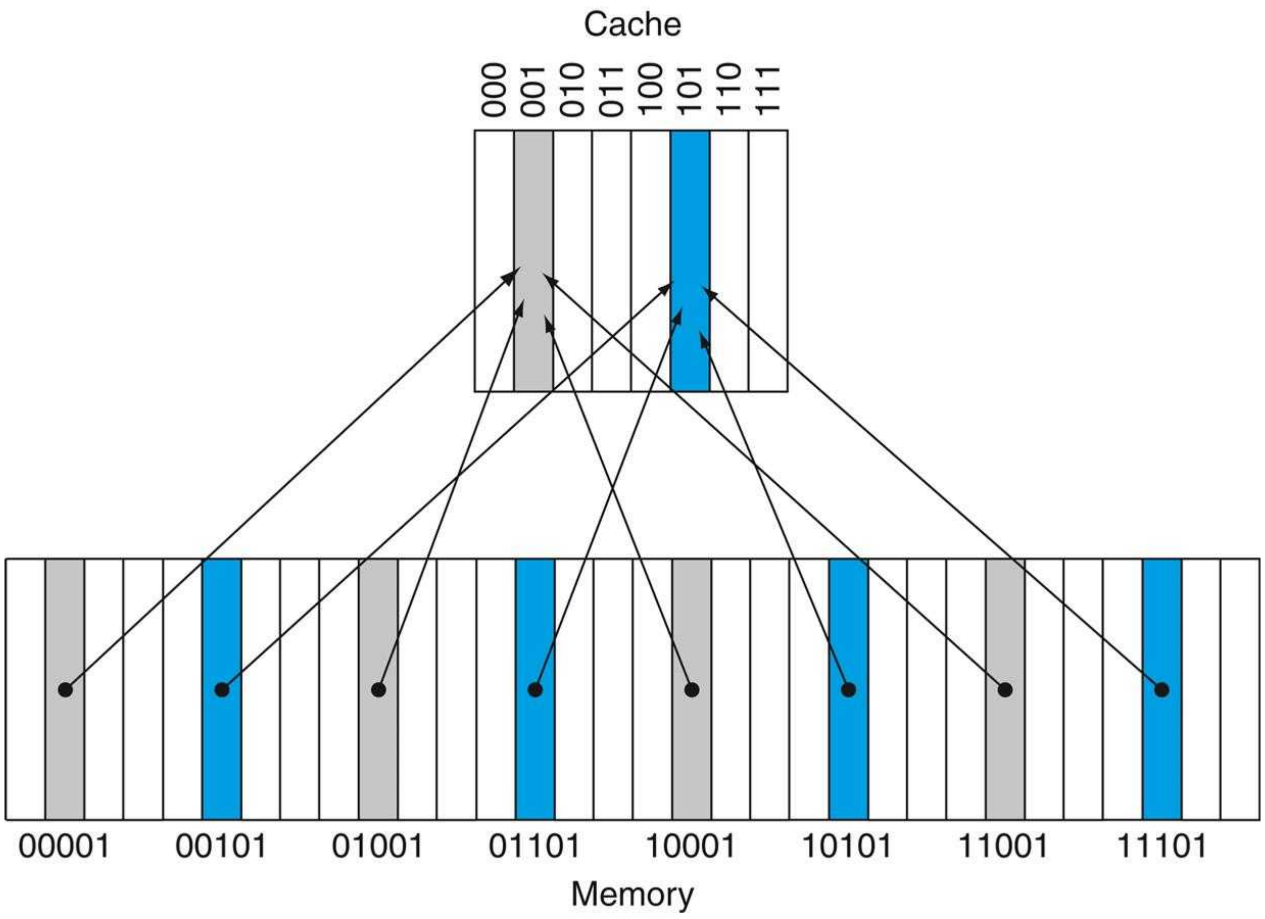


Figure 5: A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word $X\%8$. That is, the low-order $\text{LOG}_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001two, 0b01001, 0b10001, and 0b11001 all map to entry 0b001 of the cache, while addresses 0b00101, 0b01101, 0b10101, and 0b11101 all map to entry 0b101 of the cache. (Fig 5.8 of [Patterson & Hennessy]).

1. Direct-Mapped. Each memory block maps to exactly one cache line:

$$\text{line index} = (\text{block address}) \bmod N,$$

where N = number of lines.

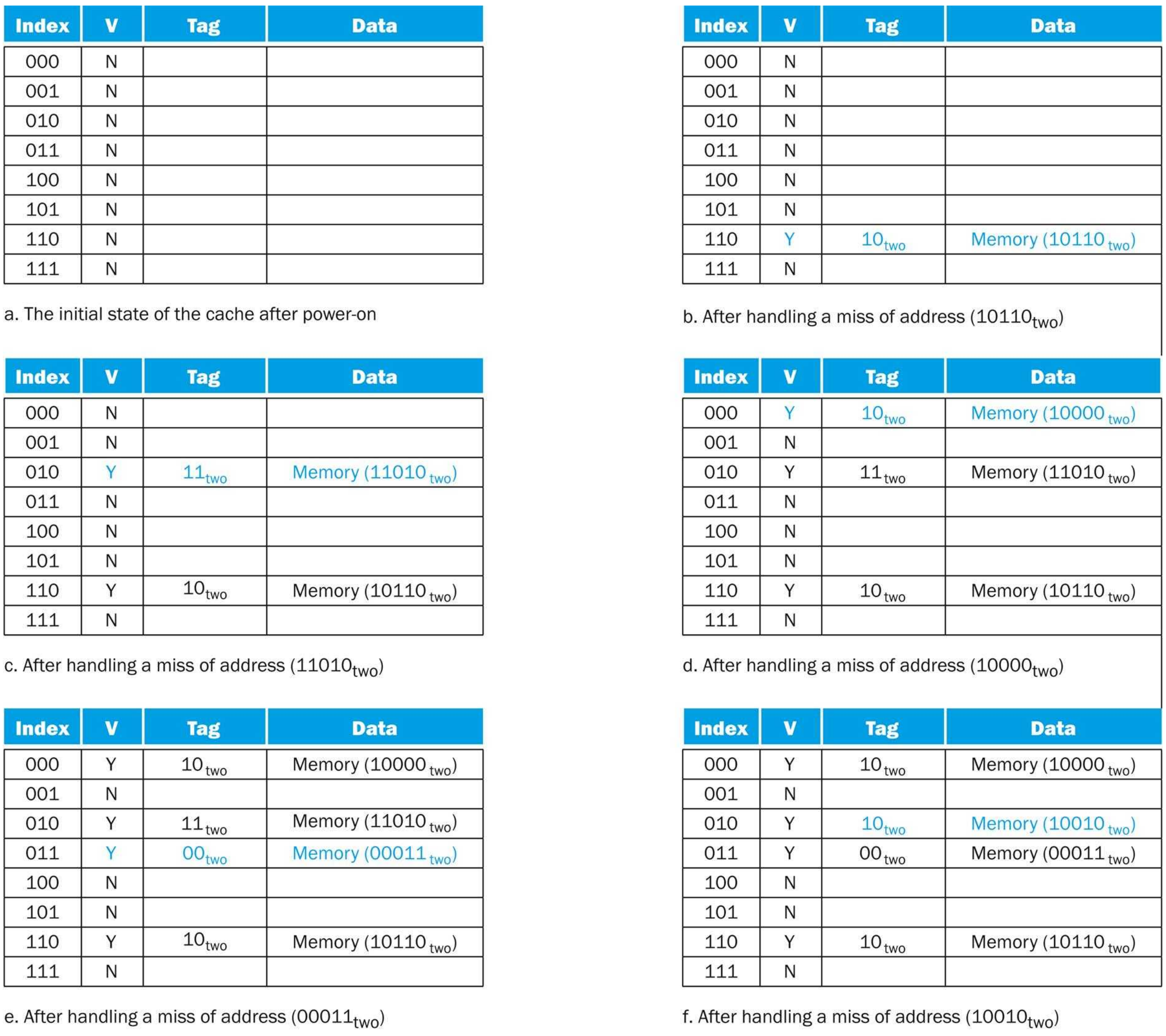
Hardware:

1. Extract index from address → select line.
2. Compare stored tag with address tag.
3. If equal and valid → hit; else → miss → fetch block → overwrite line.

This is fast (one comparator) but vulnerable to **conflict misses**.

As an example of how a direct-mapped cache works, consider a cache with 8 lines (indexes 0b000 to 0b111) and a block size of one word. Each memory reference is given as a word address, so the cache index is simply (address mod 8). The table below traces a short sequence of addresses and shows for each one:

1. the decimal and binary forms of the reference,
2. whether the reference is a hit or miss,



Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

Figure 6: Example of cache contents after each reference request. (Fig 5.9 in Patterson & Hennesy).

3. and the cache line (block) that it maps to.

Each line of the table corresponds to one snapshot in the illustration (Fig. 6a – f). Notice that addresses separated by multiples of 8 map to the same cache block, causing *conflict misses* in a direct-mapped cache. This simple example captures the essence of cache replacement and locality in direct mapping.

Decimal address of reference	**Binary address of reference**	**Hit or miss in cache**	**Assigned cache block (where found or placed)**
22	0b10110	miss (Fig. 6b)	$(0b10110 \bmod 8) = 0b110$
26	0b11010	miss (Fig. 6c)	$(0b11010 \bmod 8) = 0b010$
22	0b10110	hit	$(0b10110 \bmod 8) = 0b110$
26	0b11010	hit	$(0b11010 \bmod 8) = 0b010$
16	0b10000	miss (Fig. 6d)	$(0b10000 \bmod 8) = 0b000$
3	0b00011	miss (Fig. 6e)	$(0b00011 \bmod 8) = 0b011$
16	0b10000	hit	$(0b10000 \bmod 8) = 0b000$
18	0b10010	miss (Fig. 6f)	$(0b10010 \bmod 8) = 0b010$
16	0b10000	hit	$(0b10000 \bmod 8) = 0b000$

2. Set-Associative. Each set contains multiple **ways**. A memory block maps to exactly one set but may occupy any of its ways.

If A=number of ways, then

$$S = \frac{C}{B \times A}.$$

The index chooses one of the S sets; within that set, A entries exist.

Internal structure (the hardware "table"). Each set is a small table of A entries. Each entry stores:

- its **tag** (high-order bits of the address),
- a **valid** and **dirty** bit,
- the **data block**.

On every access:

1. The *index* bits select one set.
2. All A tags in that set are read simultaneously and compared (using A comparators).
3. If one matches and valid $= 1 \rightarrow$ hit.
4. Otherwise $\rightarrow$ miss $\rightarrow$ fetch block from lower level. If all ways are full, choose one line to evict.

Hence, each set physically contains a tiny parallel table mapping *tag* → *data*. Associativity means there are multiple such possible entries per index.

Example — 2-Way Cache Lookup. 4 KB cache, 16 B blocks, 2-way associative: $S = 4096/(16{\times}2) = 128$ sets. Each set holds 2 entries:

$$[\text{Tag}_0, \text{Valid}_0, \text{Data}_0], [\text{Tag}_1, \text{Valid}_1, \text{Data}_1].$$

If the index = 37, both tags of set 37 are compared with the requested tag. If neither matches, one entry is evicted and replaced.

Associativity reduces conflict misses because several competing blocks with the same index can coexist within a set.

3. Fully Associative. This is a special case with $S = 1$. Every block can occupy any line; all tags (hundreds) are compared in parallel. Hardware cost grows with number of comparators, so this is used only for small caches such as the TLB.

2.4 Replacement Policies

When a set is full and a miss occurs, one of its A entries must be evicted.

Least Recently Used (LRU): replace the block not accessed for the longest time. Perfect for small A (2 – 4). For 2-way caches, a single "recently used" bit per set suffices.

Random: choose any way at random; simple and almost as effective for high A.

Pseudo-LRU: approximate LRU using a binary tree of bits (used in 8-way L2 caches).

Example — 2-Way LRU. Each set has one "use" bit: when way 0 is used, mark 0 → recent; when way 1 is used, mark 1 → recent. On a miss, evict the least-recently used way.

2.5 Write Policies

How and when writes propagate to main memory:

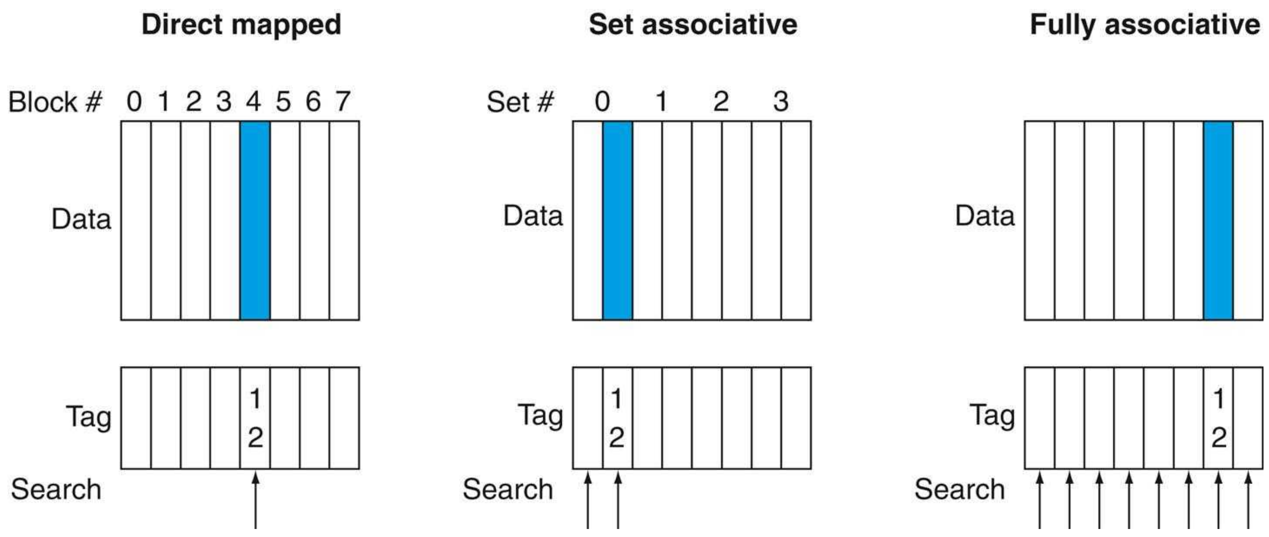


Figure 7: **Direct mapped, set associative, vs fully associative.** The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement.In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks. (Fig. 5.14 in [Patterson & Hennessy]).

Write-through. Every write updates both cache and next level. +: simpler consistency. −: more memory traffic.

Write-back. Write only modifies the cache copy; mark entry’ s **dirty bit**. When evicted, the whole block is written to lower memory. +: fewer writes to memory. −: requires dirty-bit tracking and write-back on replacement.

Write-allocate vs. No-write-allocate. On a write miss:

- **Write-allocate:** fetch block into cache, then write → used with write-back.
- **No-write-allocate:** write directly to memory, skip cache → used with write-through.

2.6 Types of Misses (3C Model)

All cache misses fall into one of three classes (Fig. 5.10):

Compulsory: first reference to a block.

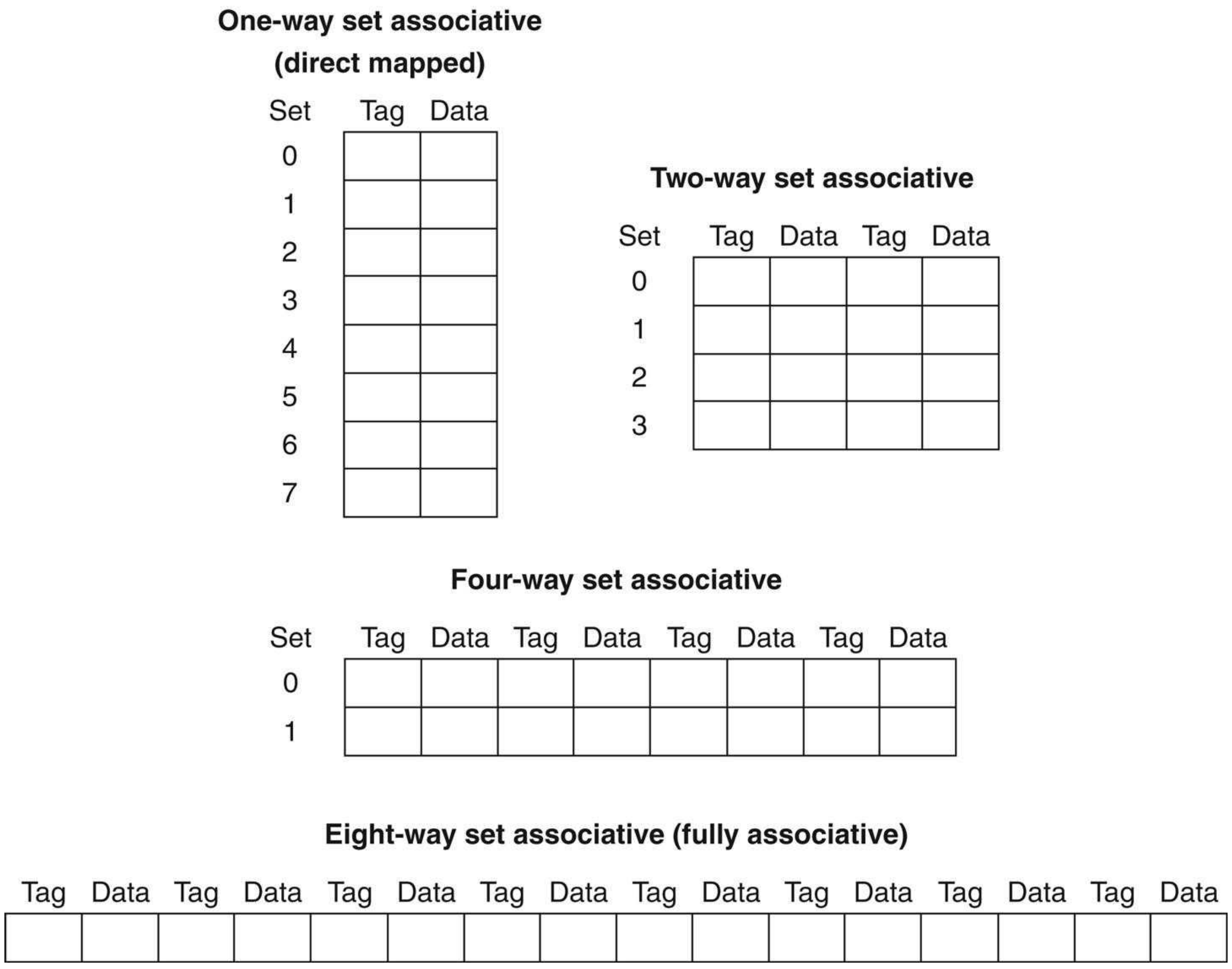
Capacity: working set larger than cache capacity.

Conflict: multiple blocks mapping to same set.

Increasing block size reduces compulsory misses (spatial locality); increasing total capacity reduces capacity misses; increasing associativity reduces conflict misses.

2.7 Quantitative Analysis: Average Memory Access Time

$$\text{AMAT} = t_{\text{hit}} + r_{\text{miss}} \times t_{\text{penalty}}.$$



One-way set associative (direct mapped)
Set
Tag
Data
0
1
2
3
4
5
6
7
Two-way set associative
Four-way set associative
Eight-way set associative (fully associative)

Figure 8: An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. (Fig. 5.15 in Patterson & Hennessy).

Example 1 – Single Level. Hit = 1 cycle, miss rate = 5 %, penalty = 50 cycles ⇒ AMAT = 1 + 0.05×50 = 3.5 cycles.

Example 2 – Two Levels. L1: 1 cycle, 5 % miss; L2: 10 cycles, 10 % of L1 misses; Memory: 100 cycles:

$$\text{AMAT} = 1 + 0.05(10 + 0.1 \times 100) = 2.0 \text{ cycles.}$$

Example 3 – Associativity Trade-off. Direct-mapped: hit = 1 cycle, miss rate = 5 %; 2-way: hit = 2 cycles, miss rate = 3 %; penalty = 50 cycles.

$$\text{AMAT}_{\text{DM}} = 1 + 0.05 \times 50 = 3.5, \quad \text{AMAT}_{\text{2-way}} = 2 + 0.03 \times 50 = 3.5.$$

Here associativity removes conflicts but increases hit time; overall effect can balance.

2.8 Pipeline Interaction: Instruction vs. Data Cache

In pipelined CPUs, both instruction fetch (IF) and data memory (MEM) stages access memory each cycle. To avoid conflicts, most designs split L1 into:

- an **Instruction cache (I-cache)** – read-only for fetching instructions,
- a **Data cache (D-cache)** – read/write for load-store data.

Lower levels (L2/L3) are unified. Thus, IF and MEM can proceed simultaneously without contention.

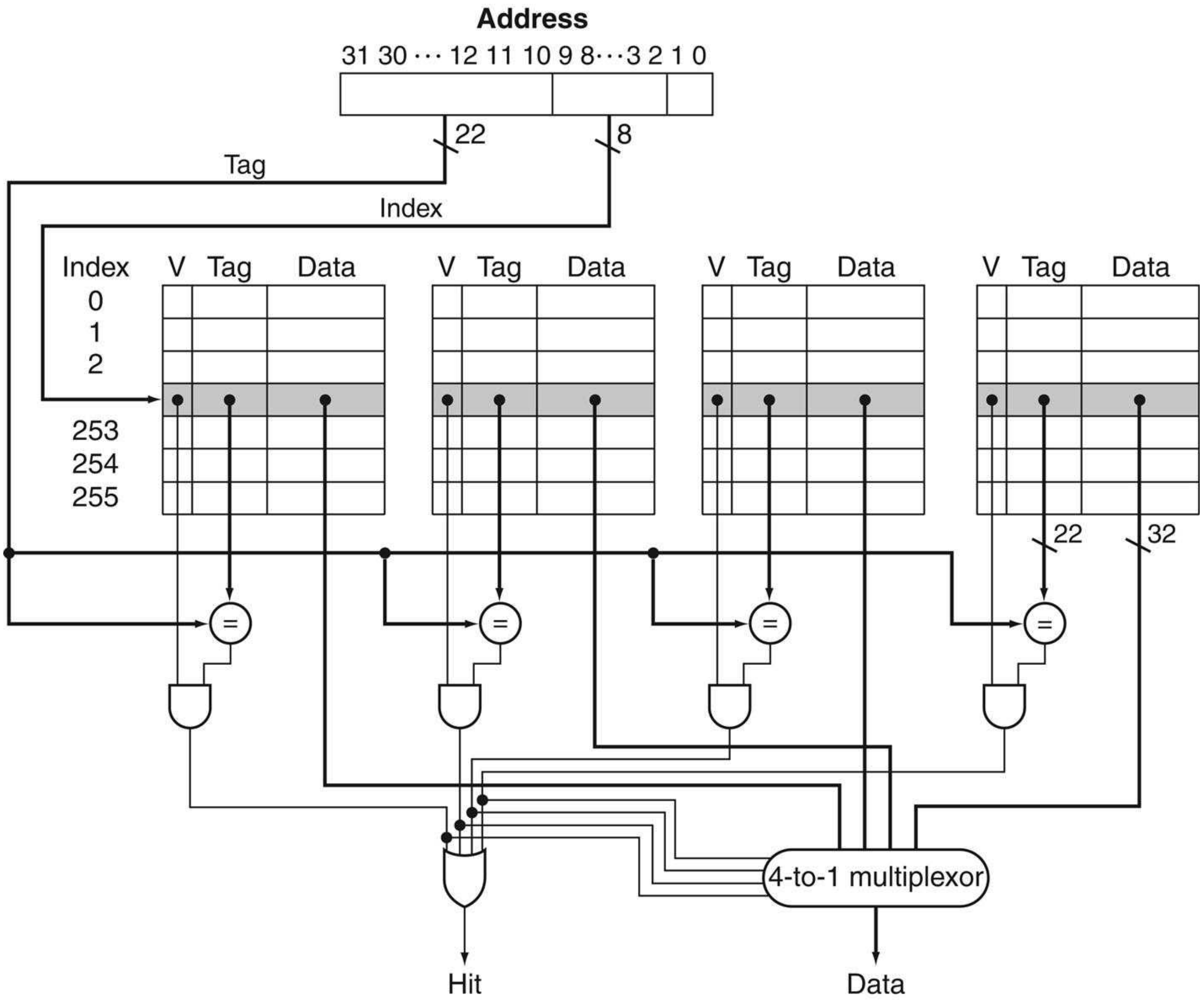


Figure 9: **Illustrative only!** Building a 4-way set associative cache.

Additional Contents (non-examinable): Locality and Data Layout in High-Level Languages

Cache principles apply to all programming languages through memory layout.

Row-major vs. Column-major order.

- **Row-major (C, C++, Python/NumPy):** consecutive elements of each row are contiguous.
- **Column-major (Fortran, MATLAB):** consecutive elements of each column are contiguous.

Traversing memory in storage order improves spatial locality.

Arrays of arrays. Python lists of lists store each row as a separate object, possibly far apart in memory. NumPy arrays store one contiguous buffer, allowing efficient cache use.

Example – Matrix Multiplication Order.

```
for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i][j] += A[i][k] * B[k][j]
```

This version scans B by column → poor locality in row-major storage. Reordering:

```
for i in range(N):
    for k in range(N):
        aik = A[i][k]
        for j in range(N):
            C[i][j] += aik * B[k][j]
```

now scans B row-wise → better cache reuse.

Key ideas.

- Matching loop order to array layout improves spatial locality.
- Small contiguous "working sets" exploit temporal locality.
- Understanding these patterns helps design faster software even in high-level languages.

Additional (non-examinable): Typical Parameters in Real Systems

Level	Typical size	Associativity	Hit ∼ cycles	Notes
L1 I/D	32 – 64 KiB	4 – 8	3 – 5	Split I/D, often VIPT
L2	256 KiB – 2 MiB	4 – 8	10 – 20	Unified
L3	8 – 64 MiB	8 – 16	30 – 50+	Shared in multicore
TLB L1	32 – 128 entries	4 – 8	≈ 1	Separate I/D common
DRAM	—	—	50 – 200	Depends on freq/DDR gen

Illustrative only; not examinable.

2.9 Summary of Section 2

- Cache = small, fast SRAM memory that stores blocks of main memory.
- Each address divides into tag / index / offset.
- Mapping strategies:
  - Direct-mapped – one line per set (simple, high conflict risk).
  - Set-associative – multiple ways per set (balanced).
  - Fully associative – any line (flexible, expensive).
- Each set physically stores a small table of entries (tag, valid, data). On access, all tags in that set are compared in parallel.
- Replacement policies: LRU, Random, Pseudo-LRU.
- Write policies: write-through / write-back, write-allocate / no-write-allocate.
- AMAT quantitatively measures performance; associativity and block size affect both hit time and miss rate.
- Split I/D caches prevent pipeline conflicts.
- (Non-examinable) Locality awareness in software leads to cache-efficient algorithms.

Concept check (Section 2)

1. If we double block size, which 3C miss typically decreases first? *Compulsory (spatial locality).*
2. What parameter primarily reduces conflict misses? *Higher associativity.*
3. In a direct-mapped cache, what determines the line index? *Block number mod #lines.*

Common pitfalls

- Do not mix hex nibbles with bit fields; rewrite addresses in binary when counting bits.
- Keep units straight: cycles vs ns; restate the frequency when converting.
- For set-associative caches, tags are per way in the selected set; compare all ways in parallel.

Prefetching (non-examinable). Hardware prefetchers fetch the next line (or detect short strides) to reduce compulsory misses; software may issue prefetch hints. Prefetching helps when access patterns are predictable.

Multicore note (non-examinable). Real systems enforce coherence so cores see a consistent view of memory. Our single-core cache model remains valid for understanding locality, mapping, and AMAT; coherence adds additional traffic and latency not covered here.

3 Exercises: Caches

3.1 Purpose and Overview

This section consolidates the principles of cache design through quantitative practice. The exercises emphasize direct-mapped caches—because they are the simplest to analyze—and Average Memory Access Time (AMAT) evaluation under varying parameters. Worked examples are written step by step; the remaining exercises are intended for individual practice. A final subsection offers a few optional, more advanced problems on associative caches and replacement policies.

3.2 Worked Example 3.1 — Direct-Mapped Cache Indexing and Miss Pattern

Problem. A processor uses a 32-bit address space and a 2 KB direct-mapped data cache with 32-byte blocks. The following byte addresses are accessed in order:

`0x0000, 0x0020, 0x0040, 0x0060, 0x0000, 0x0020.`

Determine for each access whether it is a hit or a miss, and explain why.

Step 1 – Compute parameters.

$$\frac{2048 \text{ B cache}}{32 \text{ B per block}} = 64 \text{ blocks (cache lines)} = 2^6.$$

Therefore:

$$\text{Offset bits} = \log_2(32) = 5, \quad \text{Index bits} = \log_2(64) = 6, \quad \text{Tag bits} = 32 - 6 - 5 = 21.$$

Each address can be divided as:

$$\underbrace{\text{tag}_{[31:11]}}_{21 \text{ bits}} \mid \underbrace{\text{index}_{[10:5]}}_{6 \text{ bits}} \mid \underbrace{\text{block offset}_{[4:0]}}_{5 \text{ bits}}.$$

Step 2 – Determine block numbers. A block is 32 bytes, so each consecutive block starts 32 bytes (0x20 in hexadecimal) after the previous one. We can find the *block number* by dividing the address by 32.

$$\begin{aligned}
\texttt{0x0000}/32 = 0 &\quad\Rightarrow\quad \text{block number 0,}\\
\texttt{0x0020}/32 = 1 &\quad\Rightarrow\quad \text{block number 1,}\\
\texttt{0x0040}/32 = 2 &\quad\Rightarrow\quad \text{block number 2,}\\
\texttt{0x0060}/32 = 3 &\quad\Rightarrow\quad \text{block number 3.}
\end{aligned}$$

Step 3 – Compute cache line indices. Because the cache is *direct-mapped*, each block can only go in one line, found by:

$$\text{Cache line index} = (\text{block number}) \bmod (\text{number of lines}).$$

There are 64 lines, so $\bmod 64$ selects the lower 6 bits of the block number.

$$\begin{aligned}
\text{Block } 0 &\quad\Rightarrow\quad 0 \bmod 64 = \text{line } 0,\\
\text{Block } 1 &\quad\Rightarrow\quad 1 \bmod 64 = \text{line } 1,\\
\text{Block } 2 &\quad\Rightarrow\quad 2 \bmod 64 = \text{line } 2,\\
\text{Block } 3 &\quad\Rightarrow\quad 3 \bmod 64 = \text{line } 3.
\end{aligned}$$

Therefore, `0x0020` maps to cache line 1 because it belongs to block 1, and

$$1 \bmod 64 = 1.$$

Each successive block occupies the next cache line in order.

Step 4 – Trace the sequence.

- `0x0000`: miss → fill line 0 (tag 0)
- `0x0020`: miss → fill line 1 (tag 0)
- `0x0040`: miss → fill line 2 (tag 0)
- `0x0060`: miss → fill line 3 (tag 0)
- `0x0000`: hit (tag 0 still in line 0)
- `0x0020`: hit (tag 0 still in line 1)

Step 5 – Summary. The first access to each block causes a compulsory miss (first reference to that block). Subsequent accesses to the same block hit because the corresponding cache line still contains that data.

3.3 Worked Example 3.2 — Conflict Misses in a Direct-Mapped Cache

Problem. Same configuration (2 KB direct-mapped, 32 B blocks). Access sequence:

`0x0000, 0x0800, 0x1000, 0x0000, 0x1000`

Step 1 – Mapping conflict. Cache has 64 lines → index = 6 bits. Blocks differing by multiples of $64 \times 32 = 2048$ B map to the same line.

Step 2 – Pattern. Addresses differ by 0x0800 = 2048 B → same index, different tags.

Step 3 – Trace.

- 0x0000: miss (tag 0)
- 0x0800: miss (tag 1, replaces 0)
- 0x1000: miss (tag 2, replaces 1)
- 0x0000: miss again (tag $0 \neq 2$)
- `0x1000` (final access): hit (still resident from previous step).

Conclusion. Each access evicts the previous one—pure **conflict misses**. Such pathologies motivate set-associative caches or address interleaving in software.

3.4 Worked Example 3.3 — AMAT Under Different Parameters

Problem. Compare the effect of different cache parameters on performance.

Case A: L1 hit = 1 cycle, miss rate = 5 %, penalty = 50 cycles. Case B: same cache but miss rate = 2 % (larger cache). Case C: same miss rate as A but penalty = 30 cycles (faster memory).

Step 1 – Formula.

$$\text{AMAT} = t_{\text{hit}} + r_{\text{miss}} \times t_{\text{penalty}}.$$

Step 2 – Compute.

$$\begin{aligned} \text{A:} &\quad 1 + 0.05 \times 50 = 3.5, \\ \text{B:} &\quad 1 + 0.02 \times 50 = 2.0, \\ \text{C:} &\quad 1 + 0.05 \times 30 = 2.5. \end{aligned}$$

Interpretation. Reducing miss rate or penalty both improve AMAT, but their relative benefit depends on which term dominates. Hardware (memory speed) and architecture (cache size) thus interact directly in total CPI.

3.5 Worked Example 3.4 — Address Fields in a Direct-Mapped Cache

Problem. A 1 KiB direct-mapped cache with 16 B blocks receives address `0xABCD`. Find the tag, index, and offset fields assuming 16-bit addresses.

Parameters. Cache size = 2^{10} B, block size = 2^4 B. Number of lines = $2^{10}/2^4 = 2^6$. Hence: offset = 4 bits, index = 6 bits, tag = $16 - 6 - 4 = 6$ bits.

Binary breakdown. `0xABCD` = $\texttt{1010 1011 1100 1101}_2$.

$$\text{Tag}[15:10] = \texttt{101010}\ (0x2A), \quad \text{Index}[9:4] = \texttt{111100}\ (0x3C), \quad \text{Offset}[3:0] = \texttt{1101}\ (0xD).$$

Interpretation. Index = $0x3C = 60$ selects cache line 60, tag = $0x2A$ must match for a hit, and the byte is at offset 13 within its 16 B block.

3.6 Exercises

The following problems build intuition for direct-mapped cache behavior and AMAT estimation. Work them carefully by hand before using any simulator.

3.6. **Basic Tag/Index/Offset Computation.** 32-bit address, 4 KB direct-mapped cache, 16 B blocks. Find tag, index, and offset sizes. *Hint:* $4096/16 = 256$ lines $\rightarrow$ 8 index bits.

3.6. **Address Tracing 1.** Same cache; access sequence `0x00`, `0x10`, `0x20`, `0x00`, `0x30`. Determine hits/misses and classify the miss types. *Hint:* Each block = 16 B; draw a small table of line $\leftrightarrow$ tag.

3.6. **Address Tracing 2.** 2 KB cache, 32 B blocks, direct-mapped. Accesses `0x0000`, `0x0800`, `0x1000`, `0x1800`. Predict the pattern. *Hint:* Stride = cache size $\rightarrow$ conflict every time.

3.6. **Block Size Effect on AMAT.** Hit = 1 cycle, miss rate = 8 % with 16 B blocks; 32 B blocks reduce miss rate to 5 % but raise hit time to 2 cycles. Compute both AMATs. *Hint:* $1 + 0.08M$ vs. $2 + 0.05M$ with same penalty M.

3.6. **Impact of Memory Speed.** Hit = 1 cycle, miss rate = 4 %, penalty = 100 cycles. If faster DRAM halves penalty, what is the percentage speed-up? *Hint:* Compare old/new AMAT.

3.6. **Nested AMAT.** L1: hit 1 cycle, miss 5 %; L2: hit 10 cycles, miss 10 % of L1 misses; Memory = 100 cycles. Compute overall AMAT. *Hint:* $1 + 0.05(10 + 0.1 \times 100)$.

3.6. **Performance Scaling.** For a CPU at 3 GHz, compute nanoseconds per access for AMAT = 2 cycles. *Hint:* 1 cycle = 1/3 ns.

Optional (Advanced): Associativity and Replacement Policies

The following problems extend the analysis to associative caches and realistic policies. They are recommended for deeper understanding but are not examinable.

O3.6. **2-Way Set-Associative Mapping.** 8 KB cache, 16 B blocks, 2-way associative. Compute tag/index/offset bits and explain how associativity changes miss patterns compared to direct-mapped.

O3.6. **Replacement Policy Trace.** 2-way cache, LRU replacement. Access sequence: A, B, A, C, A, B. Which blocks are evicted at each step? *Hint:* Track recency of each set.

O3.6. **Write-Back vs. Write-Through Timing.** Suppose 25 % of accesses are writes; memory write takes 40 cycles. Compare total memory traffic for write-through and write-back policies assuming 5 % miss rate. *Hint:* Only dirty blocks cause writes in write-back.

O3.6. **Effect of Associativity on AMAT.** L1 direct-mapped: miss rate = 5 %; L1 2-way: miss rate = 3 %, hit time +1 cycle. Compute which is faster given 50-cycle penalty. *Hint:* Compare AMATs quantitatively.

3.7 Concluding Remarks

These problems illustrate how cache performance arises from quantitative interactions among block size, cache size, associativity, and memory speed. Always begin with a clear understanding of the address breakdown and the hierarchy' s timing model. Mastery of direct-mapped caches forms the foundation for the virtual-memory mechanisms studied next.

4 Virtual Memory and Address Translation

Before we start: the tiny translator in the CPU

When a program uses memory, it asks for a location by giving a number. This number is the program's own way of naming a location; it is not yet a real slot in the computer's memory chips.

Inside the processor there is a small hardware translator that turns the program's number into a real location in memory. This translator is called the *Memory Management Unit* (MMU).

At a high level:

- The program provides a number we will call a *virtual address*.
- The MMU converts it into a *physical address*, which is where the data actually live in DRAM.
- To know how to convert, the MMU consults a table prepared by the operating system (we will define this table in the next pages).
- To go fast, the MMU also keeps a few recent conversions in a tiny hardware cache; we will give this cache a name later in this section.

After this brief overview, we will now define these terms carefully (virtual address, physical address, pages, the table, and the small cache) and work through step-by-step examples.

4.1 Motivation

Up to this point, we have worked with *physical addresses*—the real locations of data in memory. Modern computer systems introduce an additional layer called **virtual memory**, which allows each running program to behave as though it has a large, private, and contiguous memory space.

This layer serves three essential purposes:

1. **Isolation:** each process runs in its own protected address space.
2. **Protection:** one program cannot accidentally overwrite another's data.
3. **Flexibility:** programs can use large contiguous virtual spaces even when physical memory is fragmented or partly stored on disk.

Virtual memory is implemented through cooperation between the processor hardware and a software component of the system called the **operating system (OS)**. The hardware part performs fast address translation; the OS manages which parts of each program are currently loaded in memory.

$$\text{CPU (virtual address)} \rightarrow \text{MMU (hardware translation)} \rightarrow \text{Physical Memory}$$

4.2 Pages and Frames

Both virtual and physical memories are divided into fixed-size blocks:

- A **page** is a contiguous block of virtual memory.
- A **frame** (or *physical page*) is an equally sized block in physical memory.

Common page sizes are 4 KB, 8 KB, or 2 MB. Each virtual address is divided into two parts:

$$\underbrace{\text{Virtual Page Number (VPN)}}_{\text{high bits}} \mid \underbrace{\text{Page offset}}_{\text{low bits}}.$$

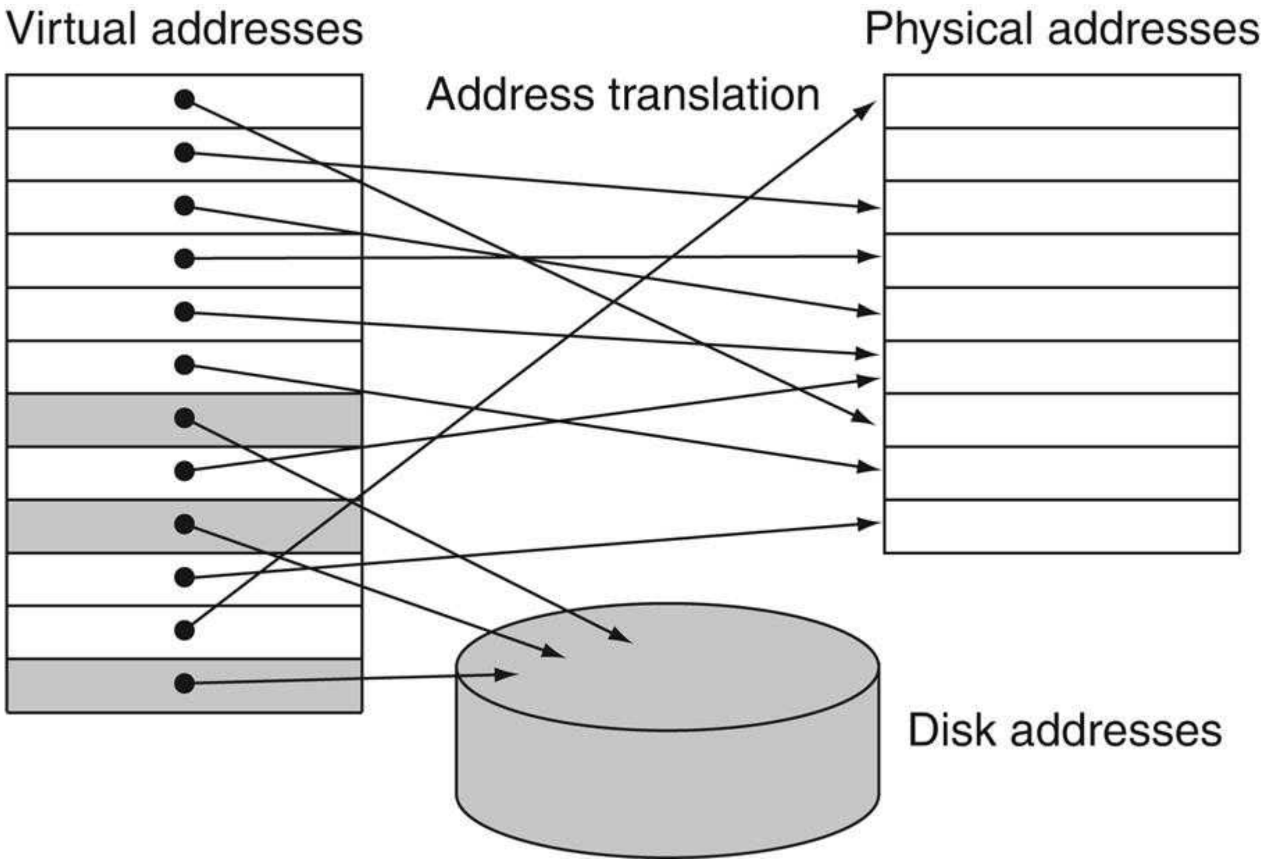


Figure 10: The concept of "Virtual Memory". In virtual memory, blocks of memory (called **pages**) are mapped to physical addresses. Both virtual and physical memory are split into pages (1 virtual page maps to 1 physical page). Processors generate virtual addresses, while memory is accessed using physical addresses. A hardware component called the MMU, or Memory Management Unit, is in charge of doing this translation (Fig 5.24 Patterson & Hennessy).

If the virtual address has m bits and each page has 2^p bytes, then:

$$\text{VPN bits} = m - p, \qquad \text{Offset bits} = p.$$

The offset selects a byte within a page (it is copied unchanged into the physical address). The VPN is used to index a data structure called the **page table**, which holds the corresponding physical frame number (PFN).

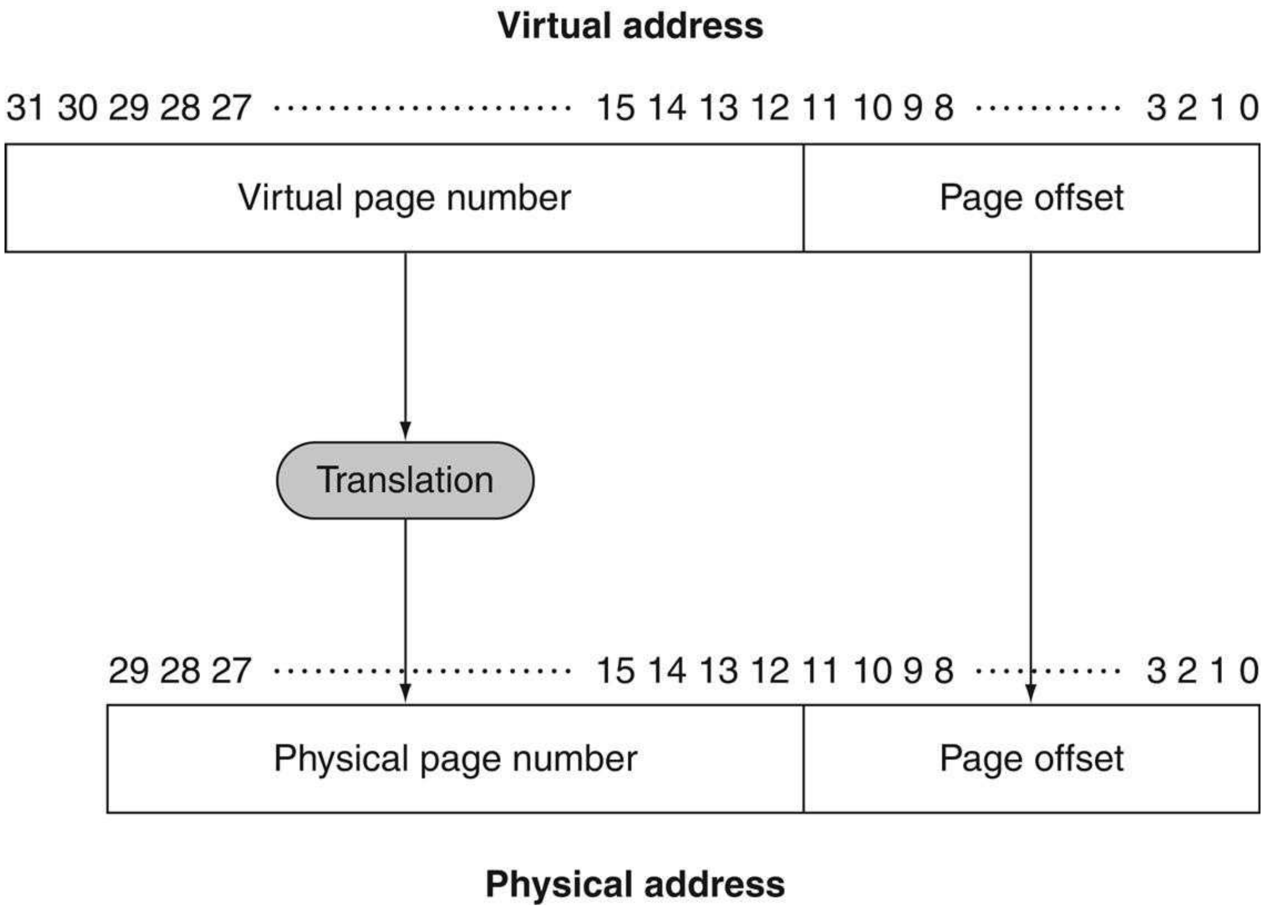


Figure 11: Translating a virtual address to a physical address. (Fig 5.25 Patterson & Hennessy)

Extra contents:
Once we define a *page* as a fixed-size unit of virtual memory (for example 4 KiB), several design choices arise:

- **Page size.** Pages should be large enough that the long access time to secondary storage (for example, disk) is amortized over a reasonably large transfer. Typical page sizes today range from 4 KiB to 16 KiB. New desktop and server systems are moving toward 32 KiB or even 64 KiB pages, while embedded systems often use smaller pages such as 1 KiB to save space.
- **Reducing page-fault rate.** It is desirable to minimize page faults because they involve extremely high latency. One effective organization is to allow *fully associative* placement of pages in physical memory—any virtual page can be placed in any physical frame—so that no page is forced out due to simple placement conflicts.
- **Handling page faults.** Because servicing a page fault already involves a very slow disk access, it is efficient to let *software* handle page faults and replacement decisions. The overhead of software management is negligible compared to the disk latency, and it allows the use of sophisticated algorithms that can further reduce the miss rate.
- **Write policy.** A write-through policy is impractical for virtual memory, since writing to disk on every store would be far too slow. Therefore, virtual memory systems always use a *write-back* policy: modified pages are marked as "dirty" and written to disk only when they are replaced.

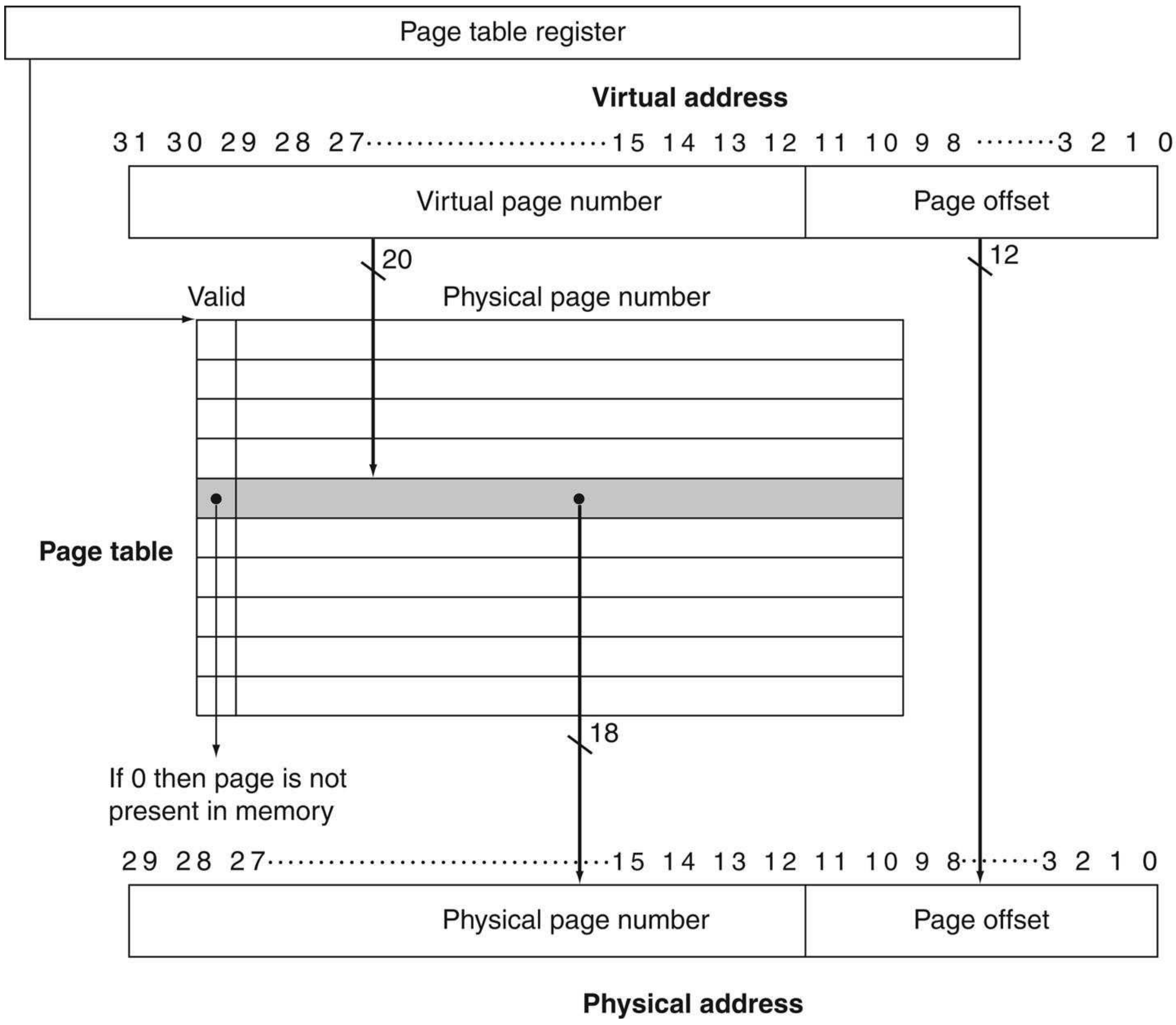


Figure 12: The Page Table. (Fig 5.26 Patterson & Hennessy).

Worked Example 4.1 – Basic Translation

Consider a 16-bit virtual address and 1 KB pages (2^{10} bytes). Then:

$$\text{VPN bits} = 6, \qquad \text{Offset bits} = 10.$$

Suppose that virtual page 5 maps to physical frame 12. For address `0x15A3`:

$$\texttt{0x15A3} = \underbrace{000101}_{\text{VPN=5}}\ \underbrace{10100011}_{\text{offset=}0xA3}.$$

Replacing the VPN with the PFN:

$$\text{PA} = \text{PFN} \,||\, \text{offset} = 00001100 \,||\, 10100011 = \texttt{0xCA3}.$$

Thus, virtual address `0x15A3` corresponds to physical address `0xCA3`.

4.3 The Page Table

The page table is a data structure maintained by the operating system but used by hardware during memory access. Each process has its own page table.

Each **Page Table Entry (PTE)** contains:

- The physical frame number (PFN),
- A valid bit (1 if the page is currently loaded in memory),
- Protection bits (read/write/execute),
- Optional flags such as dirty and reference bits for the OS.

When a program accesses memory:

1. The *Memory Management Unit*(MMU) extracts the virtual page number.
2. It looks up the corresponding page table entry.
3. If valid = 1, the page is in memory and translation succeeds.
4. If valid = 0, a **page fault** occurs, and the OS loads the page from disk.

Connection to caches. The page table plays a similar role to a cache directory: both map logical addresses (VPNs or block numbers) to physical locations.

Worked Example 4.2 – Page Table Lookup

Assume a 32-bit virtual address, 4 KB pages (2^{12} bytes). VPN = upper 20 bits, offset = lower 12 bits. If VPN = `0x00020` maps to PFN = `0x00007`, then:

$$\text{VA} = \texttt{0x00020ABC} \rightarrow \text{PA} = \texttt{0x00007ABC}.$$

Only the upper bits change; the offset is preserved.

EXTRA:
A single-level page table can be very large. For example, a 64-bit address space with 4 KB pages would require 2^{52} entries. To manage this, systems use **multi-level page tables**.
Each level acts like an index into a smaller table describing a portion of the address space. This structure drastically reduces the total size, because only the levels corresponding to used portions of memory need to exist.

4.4 The Translation Lookaside Buffer (TLB)

Accessing the page table for every memory reference would be far too slow. To avoid this, the MMU includes a small, very fast cache of recent translations called the **Translation Lookaside Buffer (TLB)**.

Each TLB entry stores:

- The virtual page number (as the tag),
- The corresponding physical frame number,
- Access permissions and valid bit.

Operation.

1. On every memory access, the MMU checks the TLB for the VPN.
2. On a **TLB hit**, the PFN is returned immediately.
3. On a **TLB miss**, the hardware or OS must consult the page table, insert the translation into the TLB, and retry the access.

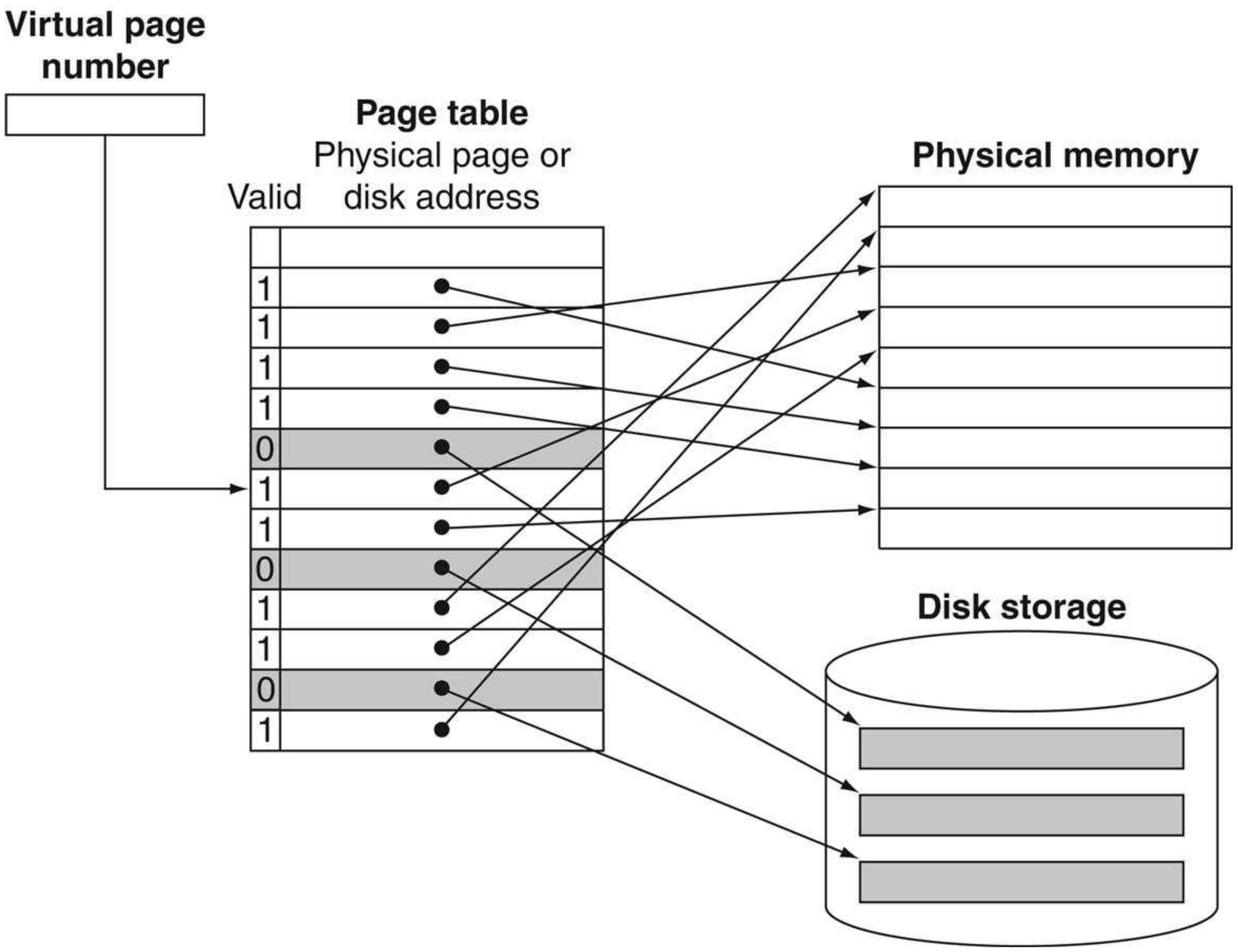


Figure 13: Virtual addresses are larger than physical addresses. That means we cannot store all virtual pages in physical pages. (Fig 5.27 Patterson & Hennessy).

Worked Example 4.3 – Effective Memory Access Time (EMAT)

Assume:

- TLB hit time = 1 ns,
- TLB hit rate = 99%,
- Memory access time = 100 ns.

$$\text{EMAT} = (0.99)(1 + 100) + (0.01)(1 + 200) = 102 \text{ ns}.$$

Even with a small miss rate, the extra translation time is noticeable.

4.5 Page Faults and the Role of the Operating System

When the page table indicates that a page is not in memory (valid = 0), the MMU raises a **page fault**. At this point, the operating system must intervene.

At a high level what happens is the following:

1. The processor temporarily stops the running program.
2. The Operating System locates the missing page on disk.
3. It copies the page into a free physical frame in RAM.

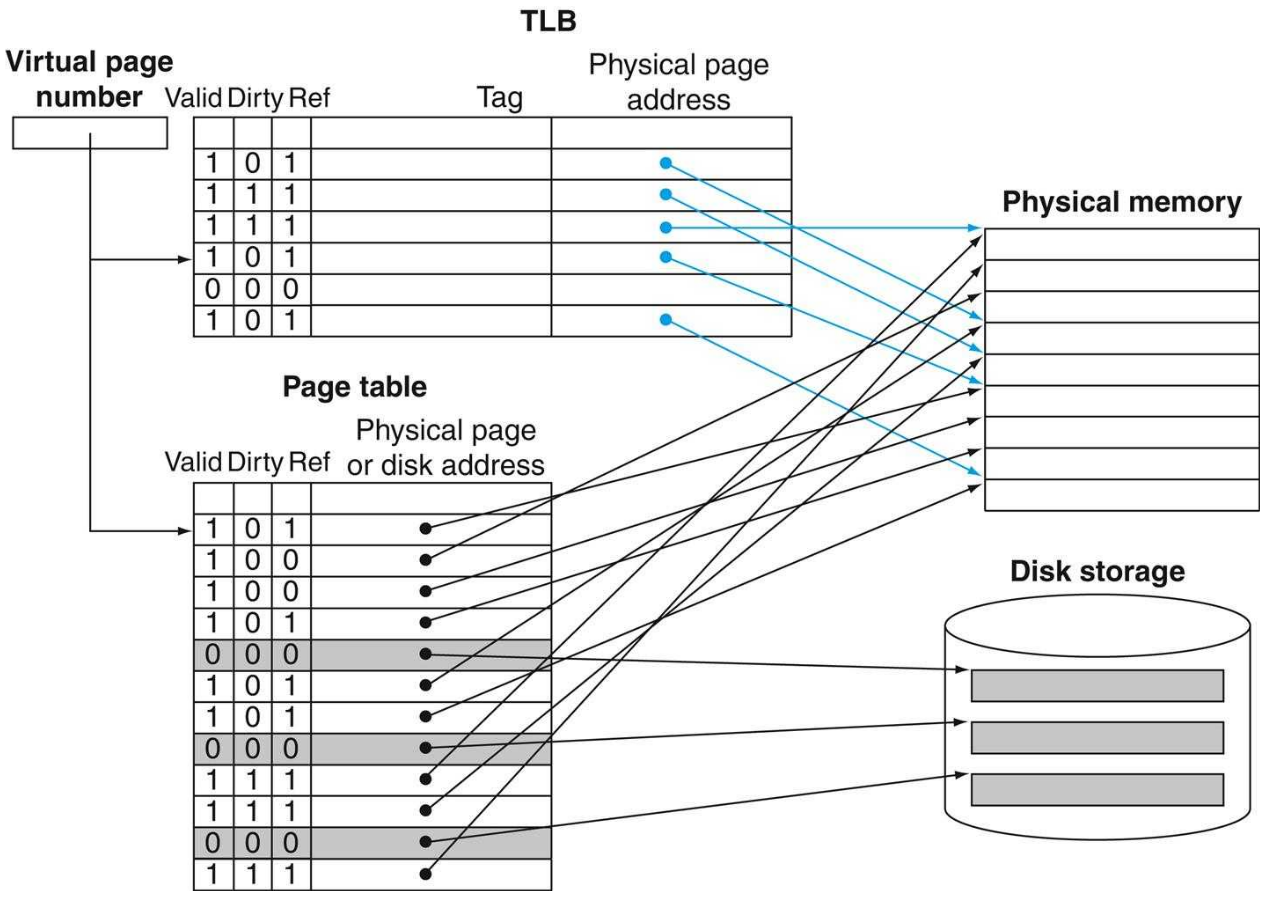


Figure 14: The TLB. (Fig 5.28 Patterson & Hennessy).

4. The OS updates the page table entry to make it valid.
5. The processor restarts the instruction that caused the fault.

From the program's point of view, this all happens automatically, but a page fault is millions of times slower than a normal memory access. This is why programs that reuse recently accessed data (good locality) perform much better.

4.6 Interaction with the Cache Hierarchy

Caches and virtual memory operate together:

- The MMU translates virtual to physical addresses before cache access.
- Caches store data using physical addresses to avoid confusion between processes.
- The TLB itself behaves as a tiny associative cache for page table entries.

In pipelined processors, address translation and cache lookup are performed in parallel to avoid slowing down memory access.

4.7 Summary of Section 4

- Virtual memory maps each process's virtual addresses to physical memory frames.
- The MMU performs this translation automatically on every memory access.
- The page table holds the mappings; the TLB caches the most recent ones.

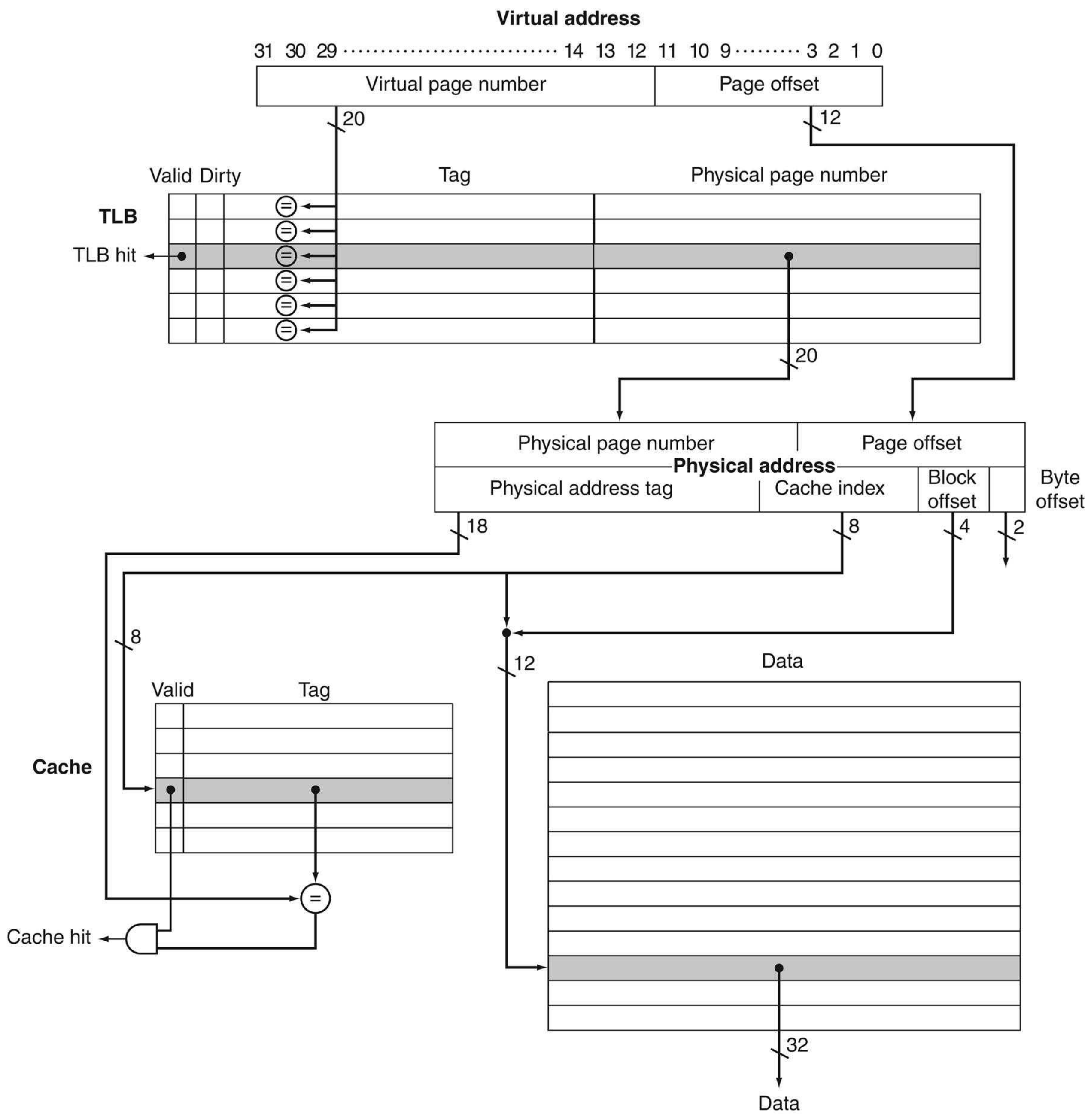


Figure 15: Virtual Memory and Cache working together to fetch a memory block. (Fig 5.29 Patterson & Hennessy).

- Page faults occur when data must be fetched from disk; the OS manages this.
- Locality is just as important here as in caching: good locality means fewer TLB misses and page faults.

Concept check (Section 4)

1. Which bits are preserved from VA to PA? *Page offset.*
2. What does a TLB store? *Mappings from VPN to PFN with permissions.*
3. What triggers a page fault? *PTE valid bit* $= 0$ *for the referenced VPN.*

5 Assembly Programming and the Integrated Memory View

5.1 Why Addressing Modes Matter

At the hardware level, every instruction must specify where its operands come from and where the result goes. The patterns by which instructions access data are called **addressing modes**.

These modes determine:

- whether the operand comes from a register or from memory,
- how the memory address is computed,
- how many memory accesses are required, and therefore
- how much time the instruction will take when executed on a pipelined processor with caches.

Efficient use of addressing modes is central to writing code that respects the memory hierarchy. Understanding them also helps interpret compiler output and reason about memory locality.

5.2 MIPS Addressing Modes

The MIPS architecture has a simple and regular design. Each instruction fits in 32 bits, and memory is accessed only by explicit load (‘lw‘, ‘lh‘, ‘lb‘) and store (‘sw‘, ‘sh‘, ‘sb‘) operations.

1. Immediate addressing. The operand is a constant encoded directly in the instruction.

```
addi $t0, $t1, 4     # $t0 = $t1 + 4
```

No memory access occurs here: the constant 4 is part of the instruction itself.

2. Register addressing. Both operands come from registers.

```
add $s0, $s1, $s2   # $s0 = $s1 + $s2
```

Again, this uses only the register file; no data cache access.

3. Base (displacement) addressing. A constant offset is added to a register to form a memory address.

```
lw $t0, 8($s1)       # load word from address ($s1 + 8) into $t0
sw $t0, 0($s2)       # store word from $t0 to address ($s2 + 0)
```

This is the standard way to access arrays or structure fields in MIPS. If $s1 holds the base address of an array, the offset selects an element.

4. PC-relative addressing. Used for branches and jumps: the new program counter (PC) is computed as the current PC plus a signed offset.

```
beq $t0, $t1, target   # if equal, PC = PC + offset to target
```

This mode improves code locality: short branches typically target nearby addresses, which remain in the instruction cache.

5. Pseudo-direct addressing. Used by the ‘j‘ and ‘jal‘ instructions. The target address is formed by combining high bits of the current PC with a 26-bit field in the instruction.

```
jal 0x00400000          # jump and link to fixed address
```

5.3 x86_64 Addressing Modes (for comparison)

While MIPS keeps addressing simple, x86_64 provides richer options. You are not required to memorize them for the exam, but learning their structure helps when reading compiler output.

In AT&T syntax (used by GNU assemblers), the general form of a memory operand is:

```
disp(base, index, scale)
```

which corresponds to the effective address:

```
EA = disp + base + index × scale.
```

Examples:

```
movq  $8(%rax), %rbx        # base + displacement
movq  (%rax,%rcx,4), %rdx   # base + index * scale
movq  $16(%rbp,%rcx,8), %rax
```

- ‘disp‘ is an immediate displacement (can be zero).
- ‘base‘ is a general-purpose register.
- ‘index‘ is an optional register multiplied by a scale factor (1, 2, 4, or 8).

These allow direct addressing of array elements or structure fields in memory. For instance, if ‘

Connection with MIPS. The x86_64 effective address formula combines the base and index computation that MIPS expresses using arithmetic instructions before a load. In both architectures, the final computed address passes through the same cache hierarchy and translation stages discussed earlier.

5.4 Efficiency and the Memory Hierarchy

From the memory-system perspective, instructions differ in cost:

- Register and immediate addressing: no memory access; fastest.
- Base/displacement addressing: one data cache access; slower.
- Indirect addressing (in x86): may cause multiple dependent memory accesses.

Programs that repeatedly use data already in registers or in nearby memory exhibit **temporal** and **spatial locality**. The cache system can then deliver data quickly, and few TLB or page-table lookups are needed.

Example: Loop locality.

```
Loop:
    lw  $t0, 0($s0)
    addi $s0, $s0, 4
    add  $t1, $t1, $t0
    bne  $s0, $s2, Loop
```

If $s0 and $s2 delimit consecutive memory locations, accesses are sequential, benefiting from spatial locality. If the loop instead followed pointers scattered across memory, cache and TLB misses would increase dramatically.

6 Exercises: Virtual Memory

Worked Example 6.1 – Virtual to Physical Address Translation

A system uses 16-bit virtual addresses and 1 KB pages (2^{10} bytes). Physical memory contains 8 frames.

1. Split the virtual address into its fields:

$$\text{VPN (6 bits)} \mid \text{Offset (10 bits)}.$$

2. Suppose the page table contains:

VPN	Valid	PFN
0	1	3
1	0	–
2	1	6
3	1	4
4	0	–
5	1	2

3. Translate virtual address `0x15A3` step by step:
   (a) Page size = 1 KB = 2^{10}. Offset = low 10 bits = `0x0A3`. VPN = bits [15:10] = 5.
   (b) In the page table: VPN 5 → PFN 2 (valid = 1).
   (c) Replace the VPN with the PFN, keep the same offset:

$$\text{PA} = (\text{PFN} \ll 10) + \text{offset} = (2 \times 1024) + 0x0A3 = 0x08A3.$$

Result: Virtual address `0x15A3` maps to physical `0x08A3`.

Worked Example 6.2 – TLB Lookup

Assume the same address space (16 bits, 1 KB pages). A 4-entry TLB stores recent translations:

Entry	VPN	PFN	Valid
0	0	3	1
1	2	6	1
2	5	2	1
3	7	4	1

Translate virtual address `0x15A3`:

1. Extract VPN = 5.
2. Search the TLB: entry 2 matches (hit).
3. PFN = 2 → Physical address = `0x08A3`.

If VPN 5 were not found, the MMU would check the page table in memory (a miss).

Worked Example 6.3 – Effective Memory Access Time (EMAT)

Parameters:

- Memory access = 100 ns
- TLB hit time = 1 ns
- TLB hit rate = 95 %
- Page-table lookup = one extra memory access on miss

$$\text{EMAT} = (0.95)(1 + 100) + (0.05)(1 + 200) = 106 \text{ ns}.$$

Even a small miss rate noticeably slows access.

6.1 Exercises

Each question below asks for a concrete operation: perform the translation, indicate hit/miss, or compute a value.

6.1. **Address Breakdown.** 32-bit address, 4 KB pages. (a) How many bits are used for the page offset? (b) How many bits remain for the VPN?

6.1. **Page Table Translation.** 16-bit virtual addresses, 1 KB pages. Given the page table:

VPN	Valid	PFN
0	1	5
1	1	3
2	0	–
3	1	1

Translate the following virtual addresses or state that a page fault occurs:

- (a) `0x03A2`
- (b) `0x0D10`
- (c) `0x07F5`

6.1. **TLB Lookup.** The TLB below contains recent entries. Determine for each virtual address whether the lookup hits or misses, and if it hits, give the physical address.

Entry	VPN	PFN	Valid
0	0x04	0x08	1
1	0x07	0x02	1
2	0x0A	0x05	1
3	0x0C	0x06	1

Virtual addresses (hex): (a) `0x0A14` (b) `0x0710` (c) `0x0408` (d) `0x0D30`

Assume page size = 1 KB.

6.1. **Compute EMAT.** Memory access = 120 ns, TLB access = 2 ns, TLB hit rate = 97%. On a TLB miss, the page-table lookup costs one extra memory access. (a) Write the exact EMAT formula you are using. (b) Compute EMAT. (c) If hit rate improves to 98.5%, what is the percent speed-up?

6.1. **Page Fault Recognition.** Given a page table with a valid bit array [1, 1, 0, 1, 0, 1], indicate for VPN = 2 whether access succeeds or triggers a page fault. What component (hardware or OS) handles it?

6.1. **Physical Memory Size.** Physical address = 18 bits, page size = 4 KB. (a) How many frames exist? (b) What is the total physical memory capacity?

(Optional) **Hierarchical Lookup.** A 32-bit virtual address is split into: 10 bits top-level index, 10 bits second-level index, 12 bits offset. What portion of the virtual address space does each top-level entry describe?

7 Integration and Summary

7.1 The Complete Memory Hierarchy

All modern systems organize memory in levels to balance speed and cost:

$$\text{CPU registers} \rightarrow \text{L1 cache} \rightarrow \text{L2/L3 cache} \rightarrow \text{Main memory (DRAM)} \rightarrow \text{Disk or SSD (storage).}$$

Each level:

- stores copies of data from lower levels,
- is smaller and faster than the next,
- exploits temporal and spatial locality.

7.2 Address Flow

When the CPU executes a load or store:

1. The instruction generates a *virtual address*.
2. The MMU translates it to a *physical address* (typically using a TLB).
3. The cache checks whether the block containing that physical address is present (hit or miss).
4. On a hit, data are returned; on a miss, the block is fetched from main memory.
5. If the page is not resident (valid bit = 0), a page fault occurs; the OS loads the page and updates the page table before the access can proceed.

Worked Example 7.1 – End-to-End Reference

A MIPS processor issues a load from virtual address `0x00402A10`. Assume 4 KB pages.

1. Split VA into VPN and offset: VPN = `0x00402`, offset = `0x0A10`.
2. TLB lookup for VPN `0x00402` hits with PFN `0x0017`.
3. Physical address = `0x0017A10`.
4. The data cache probes the set indexed by this physical address; if the tag matches and valid = 1, it is a cache hit and the load completes.
5. If the cache misses, the block containing `0x0017A10` is fetched from DRAM into the cache; the load then completes.

7.3 Design Trade-offs

- **Cache parameters:** block size, total size, associativity, and replacement policy influence hit rate and hit time (and therefore AMAT).
- **Write policies:** write-through vs. write-back and write-allocate vs. no-write-allocate affect bandwidth and latency.
- **Virtual memory parameters:** page size influences TLB coverage and page-fault frequency; TLB size/associativity affects hit rate and lookup time.

7.4 Summary of Section 7

- Each level acts as a cache for the next level down.
- Virtual memory extends the hierarchy to secondary storage and isolates processes.
- Address translation (MMU and TLB) precedes cache lookup in typical designs.
- Good locality improves both cache and TLB hit rates.
- The same core ideas recur at all levels: placement (mapping), identification (tags), replacement, and write/consistency policies.

A Reference Tables

A.1 MIPS vs x86_64 Addressing Summary

Concept	**MIPS**	**x86_64 (AT&T)**
Memory operand form	`offset($base)`	`disp(%base,%index,scale)`
Example	`lw $t0, 8($s1)`	`movq 8(%rsi), %rax`
Base + displacement	Yes	Yes
Index register	Computed in ALU	Built-in (%index)
Scale factor	1 only	1, 2, 4, 8
Load/store discipline	Load/store architecture	Memory operands in many ops
Typical array access	`lw $t0, 4($s1)`	`movl (%rdi,%rcx,4), %eax`

A.2 Glossary

AMAT Average Memory Access Time; hit time + miss rate $\times$ miss penalty.

Block (cache line) Fixed-size unit transferred between cache and memory.

Frame Fixed-size physical memory block for a virtual page.

Locality Temporal: reuse over time; Spatial: reuse of nearby addresses.

MMU Memory Management Unit; performs virtual-to-physical translation.

Page Fixed-size unit of virtual memory; size commonly 4 KB.

PTE Page Table Entry; maps VPN to PFN and stores permissions and valid/dirty bits.

TLB Translation Lookaside Buffer; small associative cache of recent VPN to PFN mappings.

VPN/PFN Virtual Page Number / Physical Frame Number.

Answer key (final numbers only; not examinable)

Section 3 (Caches)

3.1 Tag/index/offset for 4 KB DM, 16 B blocks, 32-bit: offset 4, index 8, tag 20.
3.2 With sequence 0x0000, 0x0800, 0x1000, 0x0000, 0x1000: miss, miss, miss, miss, **hit**.
3.3 2-way, parameters: sets $= 4096/(16 \cdot 2) = 128 \Rightarrow$ offset 4, index 7, tag 21; trace as written.
3.4 1 KiB DM, 16 B blocks, 16-bit: offset 4, index 6, tag 6; 0xABCD $\Rightarrow$ tag 0x2A, index 0x3C, offset 0xD.
AMAT example: $1 + 0.05 \cdot 50 = 3.5$ cycles; improved $1 + 0.02 \cdot 50 = 2.0$ cycles.

Section 6 (Virtual Memory)

6.1 VA 0x15A3 $\Rightarrow$ PA 0x08A3.
6.2 TLB hit for VPN 5 $\Rightarrow$ PFN 2 $\Rightarrow$ PA 0x08A3.
6.3 EMAT $= (0.95)(1 + 100) + (0.05)(1 + 200) = 106$ ns.
6.4 EMAT formula: EMAT $= h \cdot (t_{\text{TLB}} + t_{\text{mem}}) + (1 - h) \cdot (t_{\text{TLB}} + 2t_{\text{mem}})$.
Numbers: $h = 0.97$, $t_{\text{TLB}} = 2$ ns, $t_{\text{mem}} = 120$ ns $\Rightarrow$ EMAT $= 0.97 \cdot 122 + 0.03 \cdot 242 = 125.6$ ns. Speed-up to $h = 0.985$: new EMAT $= 0.985 \cdot 122 + 0.015 \cdot 242 = 123.8$ ns; speed-up $\approx (125.6 - 123.8)/125.6 \approx 1.4\%$. Frames for 18-bit PA, 4 KB pages: $2^{18}/2^{12} = 2^6 = 64$ frames; capacity $= 64 \cdot 4\,\text{KB} = 256\,\text{KB}$.