Memory System Design

Agnik Saha
Lecturer
Dept. of CSE,
RPSU.

Memory System Design

Design problem – We want a memory unit that:

- Can keep up with the CPU's processing speed
- Has enough capacity for programs and data
- Is inexpensive, reliable, and energy-efficient

Topics in This Part	
Chapter 1	Main Memory Concepts
Chapter 2	Cache Memory Organization
Chapter 3	Mass Memory Concepts
Chapter 4	Virtual Memory and Paging

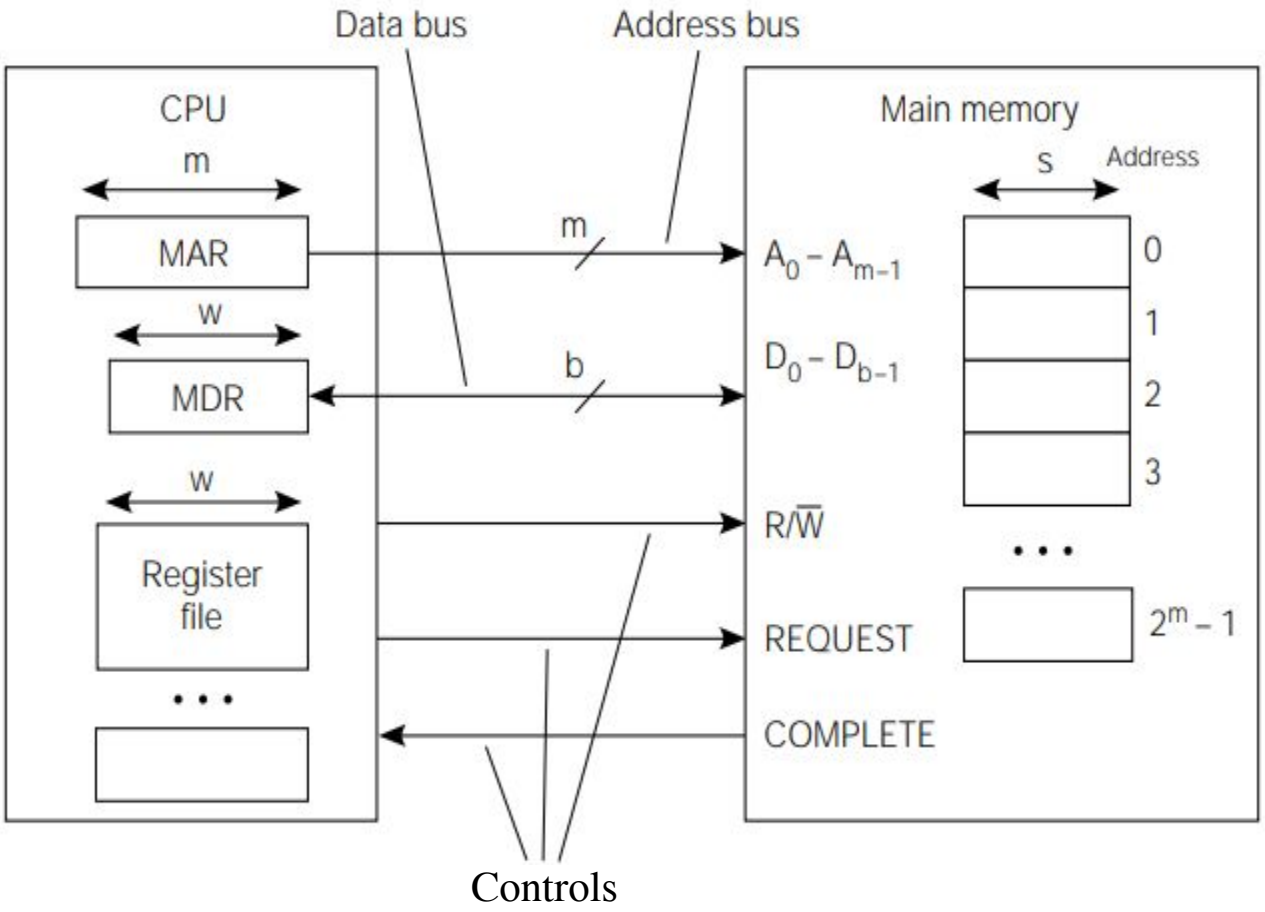
Main Memory Concepts

Technologies & organizations for computer's main memory

- SRAM (cache), DRAM (main), and flash (nonvolatile)

Topics in This Lecture	
1	Memory Structure and SRAM
2	DRAM and Refresh Cycles
3	Nonvolatile Memory
4	The Need for a Memory Hierarchy

The CPU–Main Memory Interface



The CPU–Main Memory Interface

Sequence of events:
Read:
1. CPU loads MAR, issues Read, and REQUEST
2. Main Memory transmits words to MDR
3. Main Memory asserts COMPLETE.

Write:
1. CPU loads MAR and MDR, asserts Write, and REQUEST
2. Value in MDR is written into address in MAR.
3. Main Memory asserts COMPLETE.

The CPU–Main Memory Interface

Additional points:

- If b<w, Main Memory must make w/b b-bit transfers.
- Some CPUs allow reading and writing of word sizes <w.
- Example: Intel 8088: m=20, w=16,s=b=8.
  8- and 16-bit values can be read and written
- If memory is sufficiently fast, or if its response is predictable, then COMPLETE may be omitted.
- Some systems use separate R and W lines, and omit REQUEST.

Some memory Properties

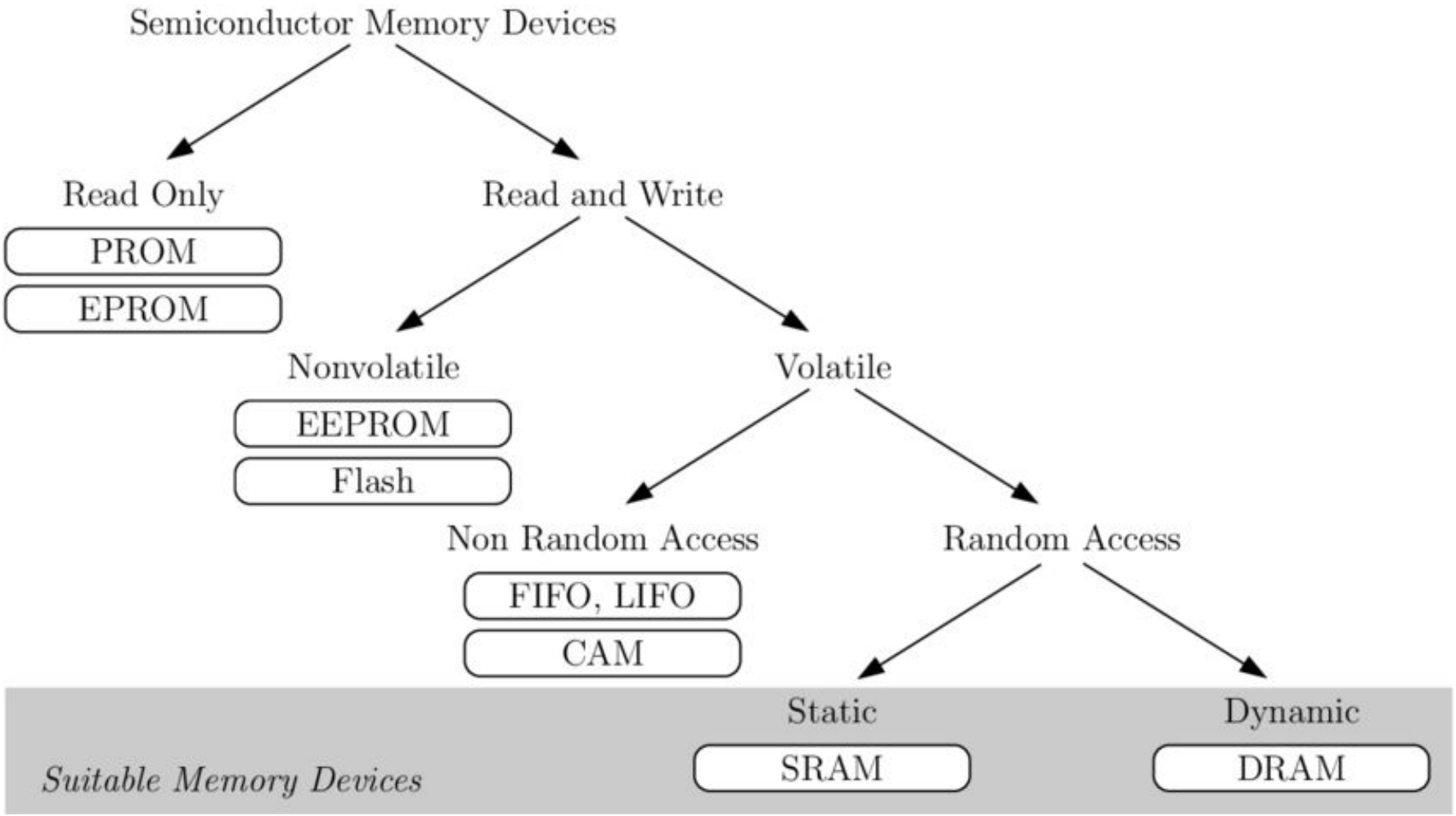
Symbol	Definition	Intel 8088	Intel 8086	IBM/Moto. 601
w	CPU Word Size	16bits	16bits	64 bits
m	Bits in a logical memory address	20 bits	20 bits	32 bits
s	Bits in smallest addressable unit	8	8	8
b	Data Bus size	8	16	64
2^m	Memory wd capacity, s-sized wds	2^{20}	2^{20}	2^{32}
2^mxs	Memory bit capacity	2^{20}x8	2^{20}x8	2^{32}x8

Memory Performance Parameter

Symbol	Definition	Units	Meaning
t_a	Access time	time	Time to access a memory word
t_c	Cycle time	time	Time from start of access to start of next access
k	Block size	words	Number of words per block
b	Bandwidth	words/time	Word transmission rate
t_l	Latency	time	Time to access first word of a sequence of words
$t_{bl} = t_l + k/b$	Block access time	time	Time to access an entire block of words

(Information is often stored and moved in blocks at the cache and disk level.)

Semiconductor Memory Classification



Semiconductor Memory Classification

- Size:
  - Bits, Bytes, Words
- Timing Parameters:
  - Read access, write access, cycle time
- Function:
  - Read Only (ROM) — non-volatile
  - Read-Write (RWM) — volatile
  - NVRWM - Non-volatile Read
- Access Pattern:
  - Random Access, FIFO, LIFO, Shift Register, CAM
- I/0 Architecture:
  - Single Port, Multi-port
- Application:
  - Embedded, External, Secondary.

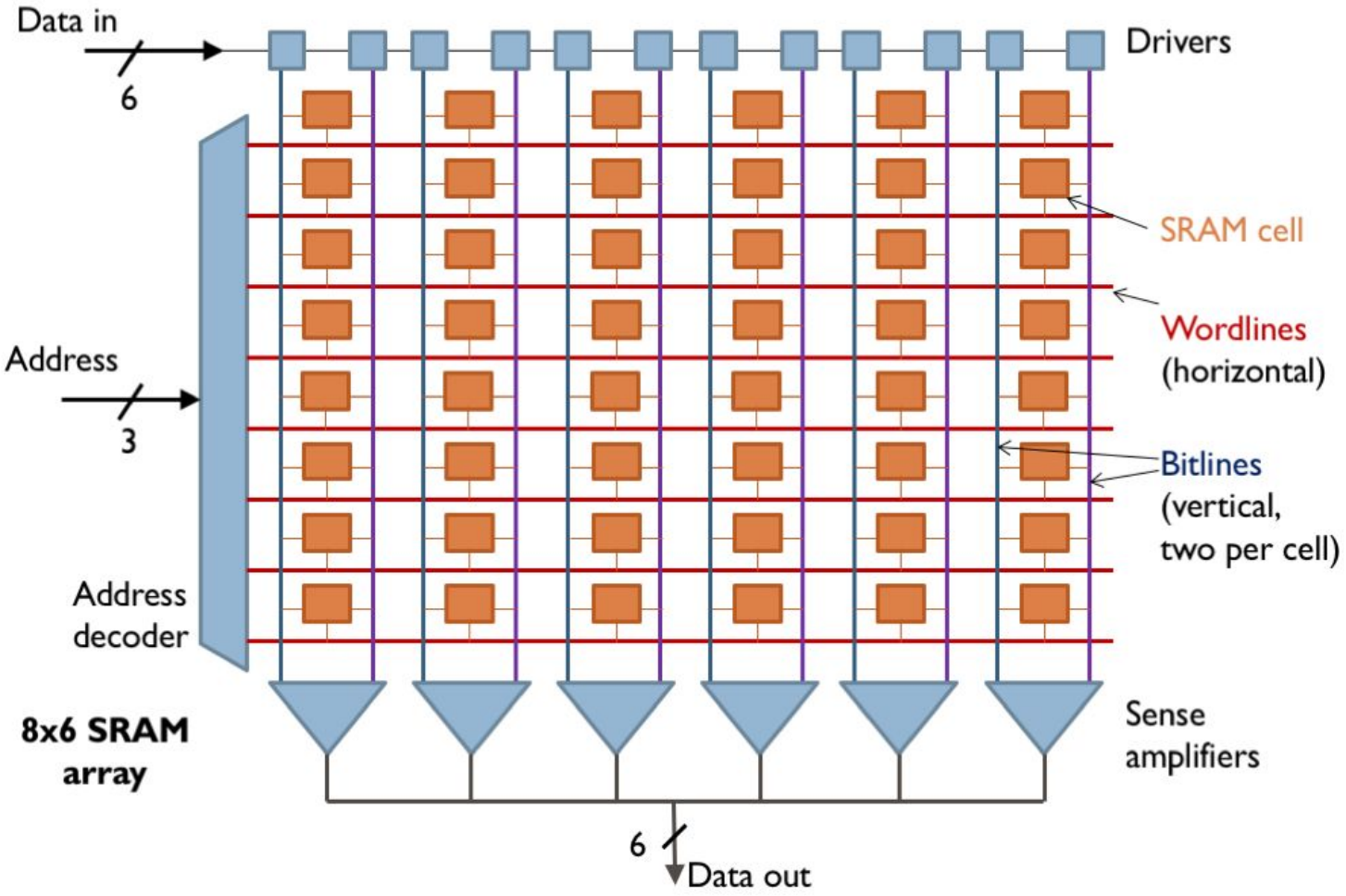
Let's make a 1 MB memory

- 1 MB=2^{20} words X 8 bits=2^{23} bits, each word in a separate row
- A decoder would reduce the number of access pins from 2^{20} access pins to 20 address lines.
- We'd fit the pitch of the decoder to the word cells, so we'd have Word Lines with no area overhead.
- The output lines (=bit lines) would be extremely long, as would the delay of the huge decoder.
- The array's height is about 128,000 times larger than its width ($2^{20}/2^{3}$).

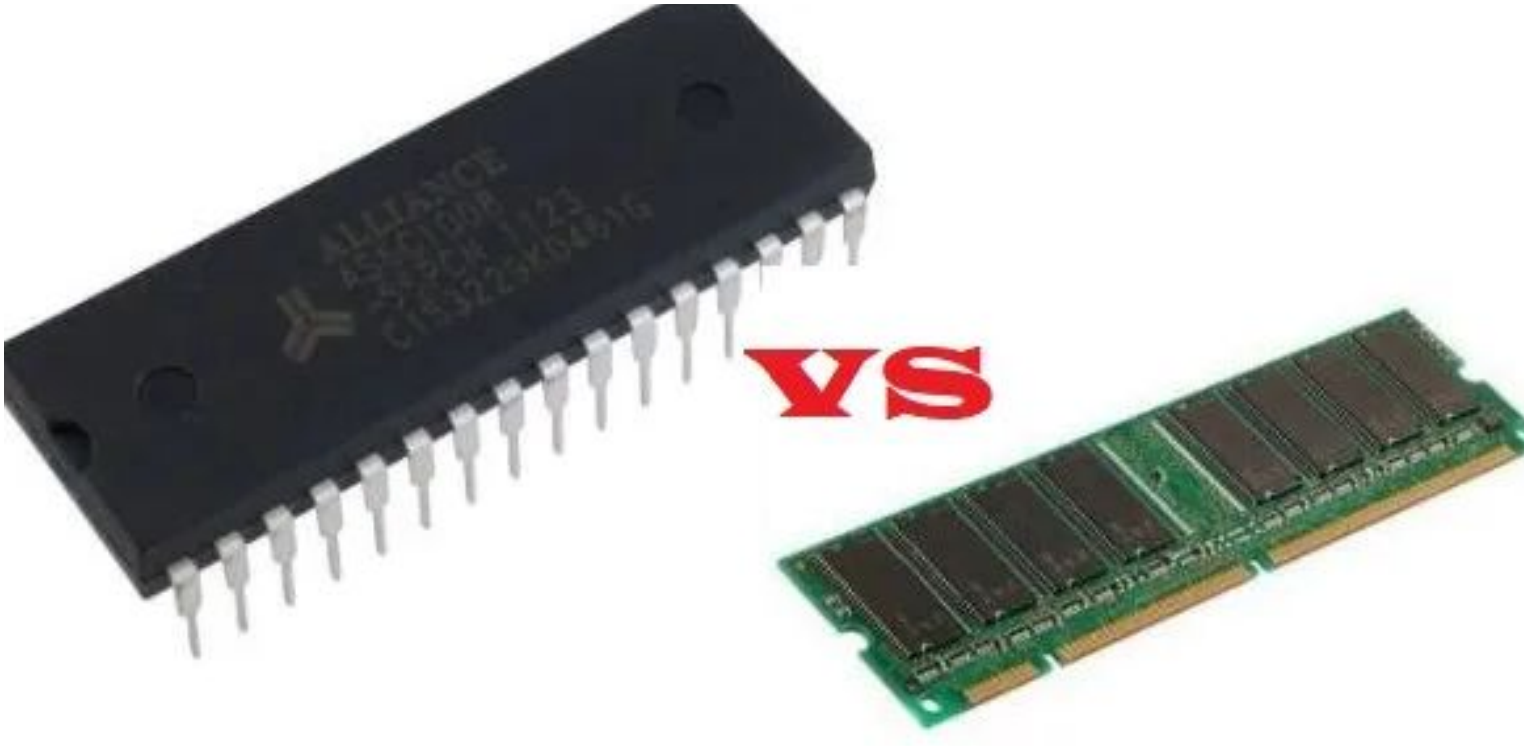
SRAM

- Instead, let's make the array square:
- 1 MB=2^{23} bits=2^{12} rows X 2^{11} columns.
- There are 4000 rows, so we need a 12-bit row address decoder (to select a single row)
- There are 2000 columns, representing 256 8-bit words.
- We need to select only one of the 256 words through a column address decoder (or multiplexer)
- We call the row lines "Word Lines" and the column lines "Bit Lines".

SRAM


Data in
6
Drivers
SRAM cell
Wordlines
(horizontal)
Address
3
Bitlines
(vertical,
two per cell)
Address
decoder
8x6 SRAM
array
Sense
amplifiers
6
Data out

DRAM vs SRAM



DRAM vs SRAM

SRAM	DRAM
It can store data as long as electricity is available.	It saves data for as long as the power is on or for a few moments if the power is turned off.
Because capacitors aren't utilized, there's no need to refresh.	The contents of the capacitor must be updated on a regular basis in order to store information for a longer amount of time.
SRAM has a storage capacity of 1 MB to 16 MB in most cases.	DRAM, which is often found in tablets and smartphones, has a capacity of 1 GB to 2 GB.
The storage capacity of SRAM is low.	The storage capacity of DRAM is higher than SRAM.
SRAM is more expensive than DRAM.	DRAM is less expensive than SRAM.
It is comparatively faster.	It is comparatively slower.
The power consumption is minimal, and the access speed is quick.	The cost of production is low, and the memory capacity is higher.
SRAM is used in cache memories.	DRAM is used in main memories.

MEMORY CHIPS AND THEIR CAPACITY

How many chips are necessary to implement a 4 MBytes memory:

1) using 64 Kbit SRAM;

2) using 1Mbit DRAM;

3) 64 KBytes using 64 Kbit SRAM and the rest using 1Mbit DRAM.

MEMORY CHIPS AND THEIR CAPACITY

How many chips are necessary to implement a 4 MBytes memory:

1) using 64 Kbit SRAM;

2) using 1Mbit DRAM;

3) 64 KBytes using 64 Kbit SRAM and the rest using 1Mbit DRAM.

$$\frac{\text{Memory capacity (expressed in bytes)}}{\text{Chip capacity (expressed in bytes)}}$$

1)$n1 = 2^{22}/2^{13} = 512$ chips.(using the second formula)

2)$n2 = 2^{22}/2^{17} = 32$ chips.

3)$n3 = 2^{16}/2^{13} + \text{floor}((2^{22}-2^{16})/2^{17}) = 8 + 32(\text{SRAM} + \text{DRAM})$

Principle of Locality

In running a program the memory is accessed for two reasons:

- read instructions;
- read/write data.

Memory is not uniformly accessed; addresses in some region are accessed more often than others, and some addresses are accessed again shortly after the current access. In other words programs tend to favor parts of the address space at any moment of time.

- temporal locality: an access to a certain address tend to be repeated shortly thereafter;
- spatial locality: an access to a certain address tends to be followed by accesses to nearby addresses.

The Need for a Memory Hierarchy

The widening speed gap between CPU and main memory

Processor operations take of the order of 1 ns

Memory access requires 10s or even 100s of ns

Memory bandwidth limits the instruction execution rate

Each instruction executed involves at least one memory access

Hence, a few to 100s of MIPS is the best that can be achieved

A fast buffer memory can help bridge the CPU-memory gap

The fastest memories are expensive and thus not very large

A second (third?) intermediate cache level is thus often used

Typical Levels in a Hierarchical Memory

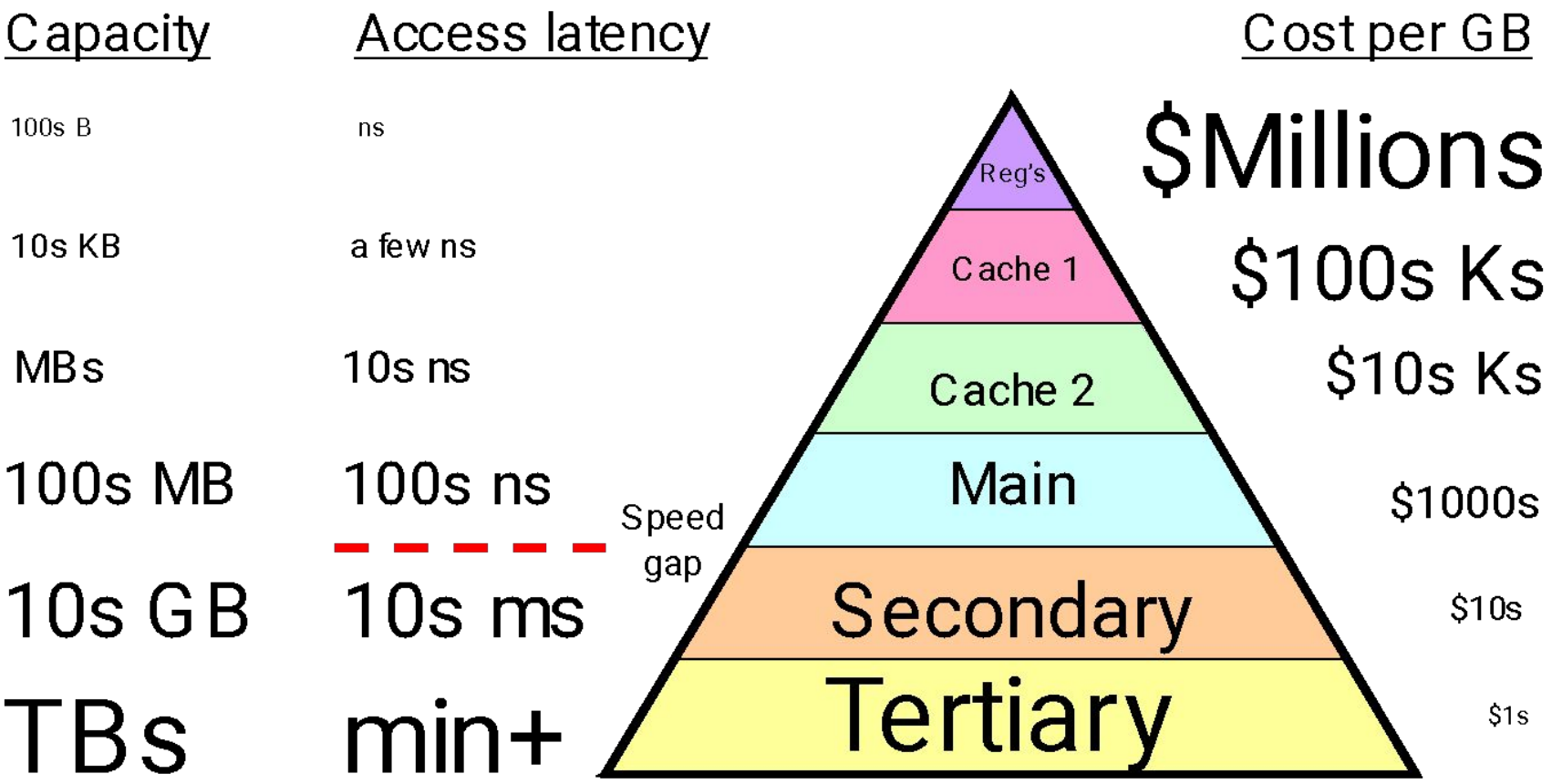
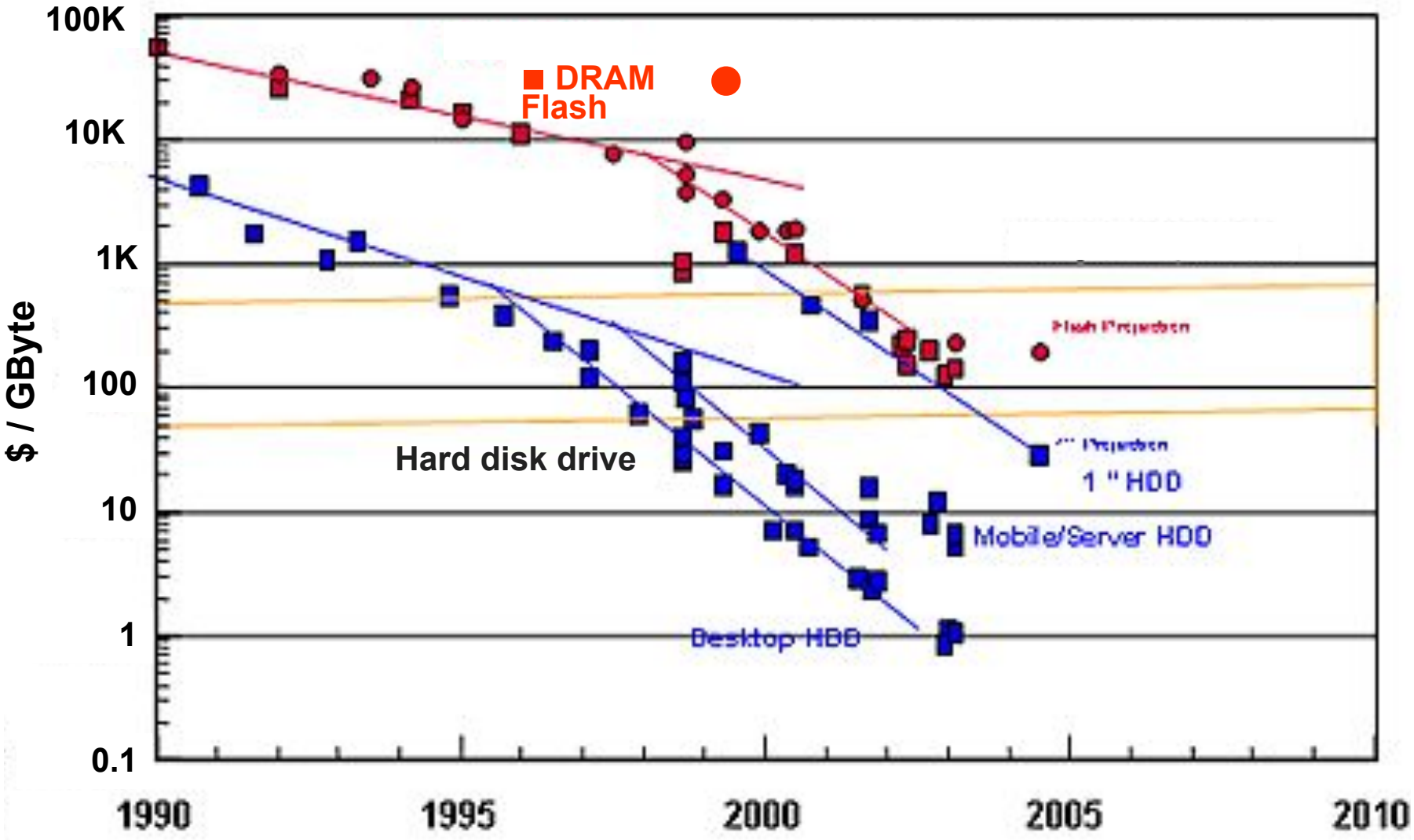


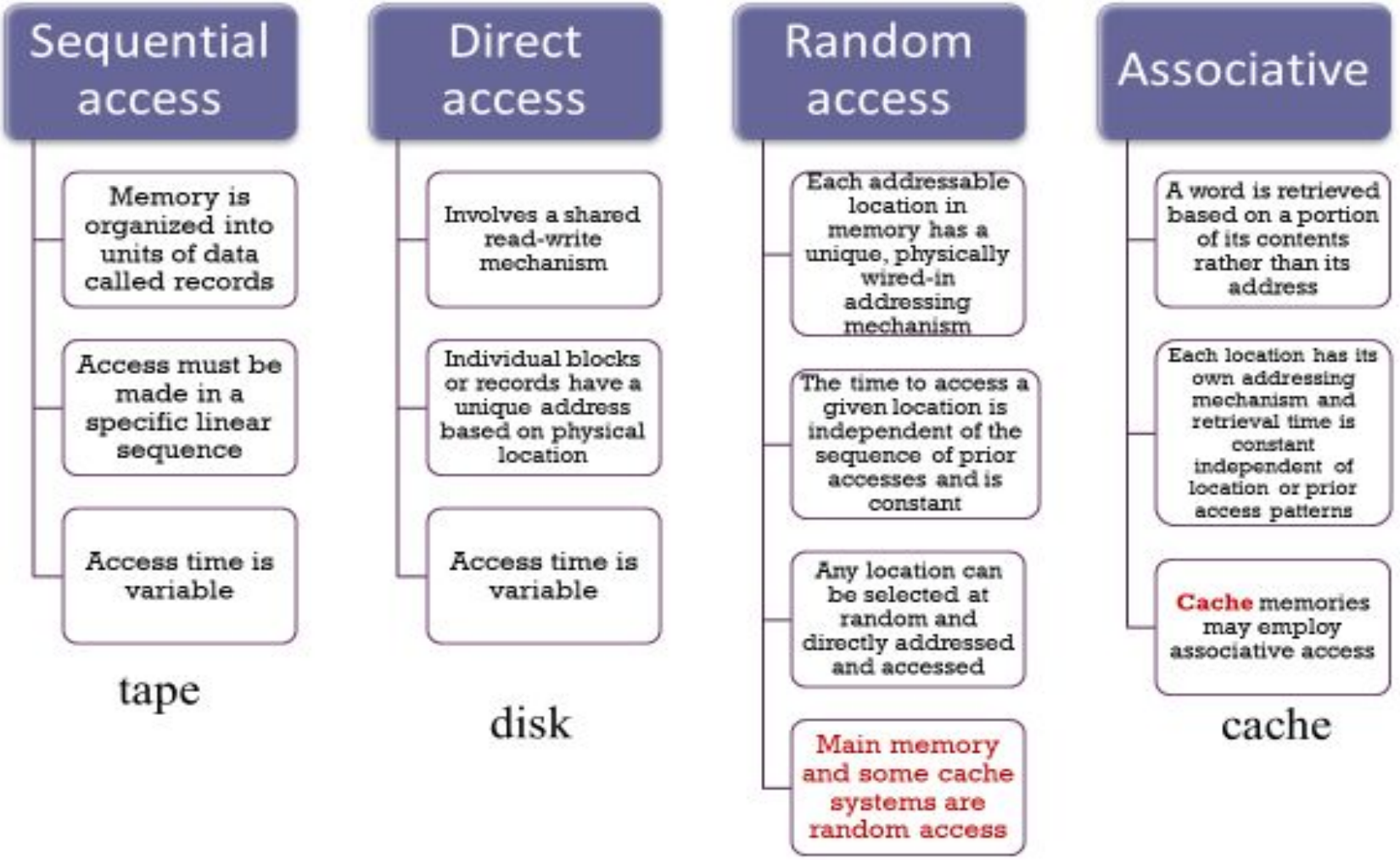
Fig. 17.14 Names and key characteristics of levels in a memory hierarchy.

Memory Price Trends



Source: https://www1.hitachigst.com/hdd/technolo/overview/chart03.html

Method of Accessing Units of Data



Cache Memory Organization

Processor speed is improving at a faster rate than memory's

- Processor-memory speed gap has been widening
- Cache is to main as desk drawer is to file cabinet

Topics in This Lecture	
1	The Need for a Cache
2	What Makes a Cache Work?
3	Direct-Mapped Cache
4	Set-Associative Cache
5	Cache and Main Memory
6	Improving Cache Performance

Many Types of Cache

Examples

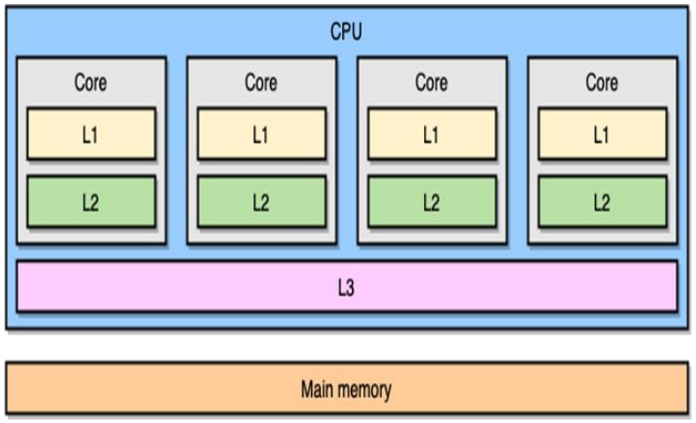
- Hardware: L1 and L2 CPU caches, …
- Software: virtual memory, web browser caches, …

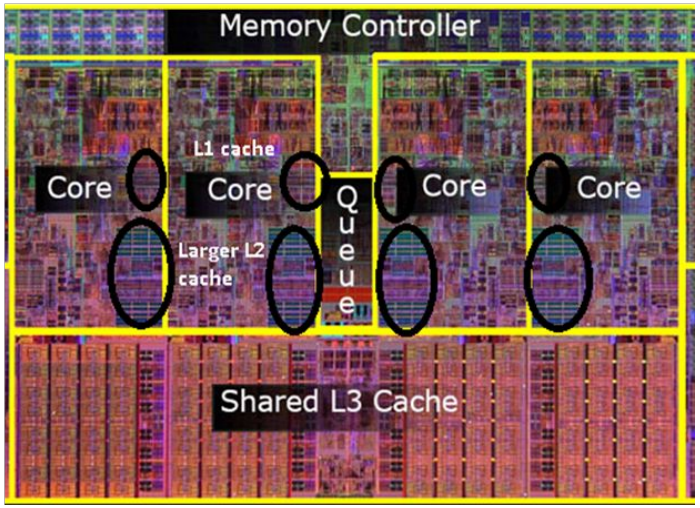
Each processor core sports two levels of cache:

- 2 to 64 KB Level 1 (L1) cache very high speed cache
- ~256 KB Level 2 (L2) cache medium speed cache
- All cores also share a Level 3 (L3) cache. The L3 cache tends to be around 8 MB.

Performance difference between L1, L2 and L3 caches

- L1 cache access latency: 4 cycles
- L2 cache access latency: 11 cycles
- L3 cache access latency: 39 cycles
- Main memory access latency: 107 cycles





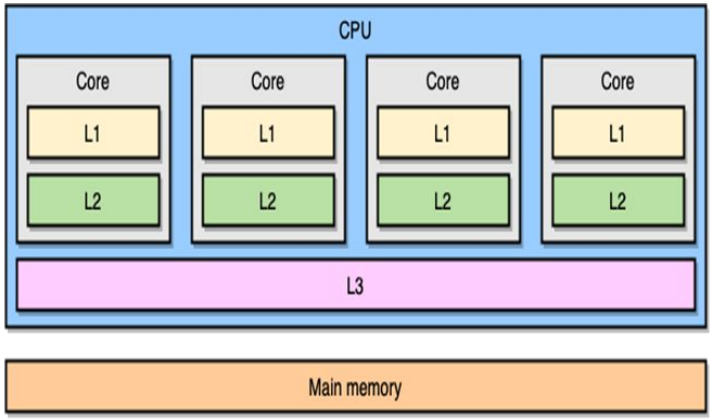
Many Types of Cache

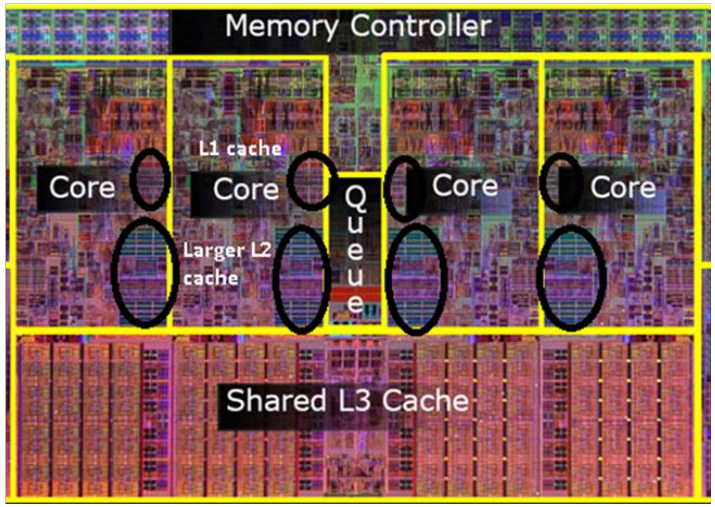
Many common design issues

- each cached item has a “tag” (an ID) plus contents
- need a mechanism to efficiently determine whether given item is cached
  - combinations of indices and constraints on valid locations
- on a miss, usually need to pick something to replace with the new item
  - called a “replacement policy”
- on writes, need to either propagate change or mark item as “dirty”
  - write-through vs. write-back

Different solutions for different caches

- Lets talk about CPU caches as a concrete example…





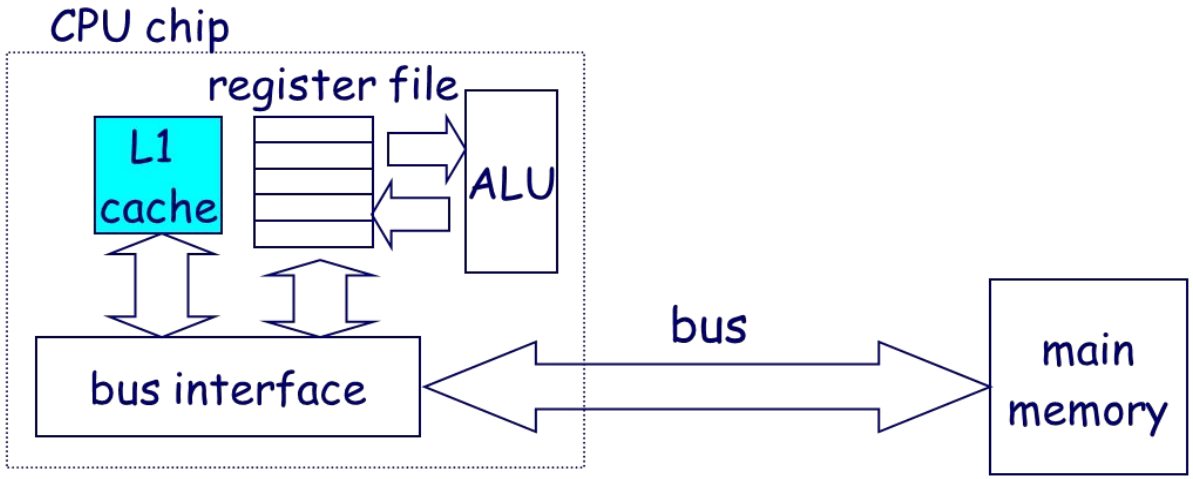
Hardware cache memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware

- Hold frequently accessed blocks of main memory

CPU looks first for data in L1, then in main memory

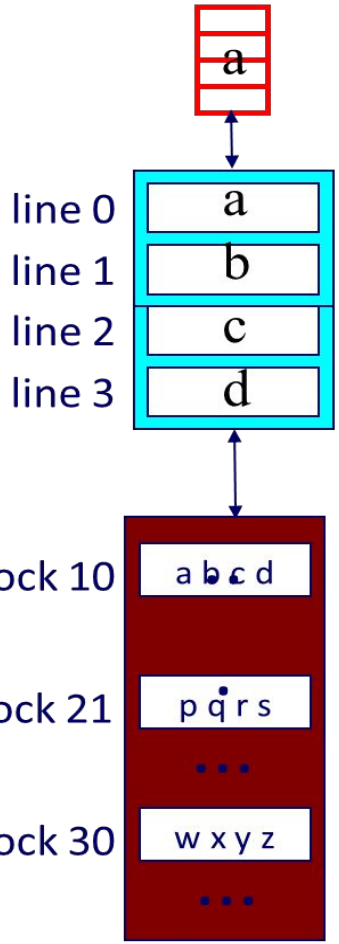
Typical system structure:



Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is **a 4-byte word**

The transfer unit between the cache and main memory is a 4-word block (16 bytes)

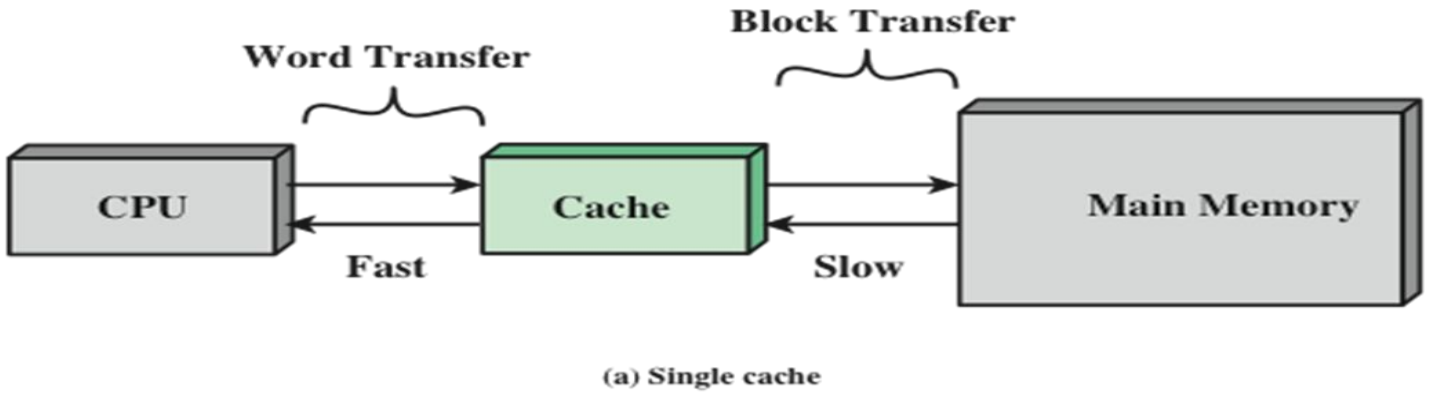


The tiny, very fast CPU register file has room for one word (4-byte)

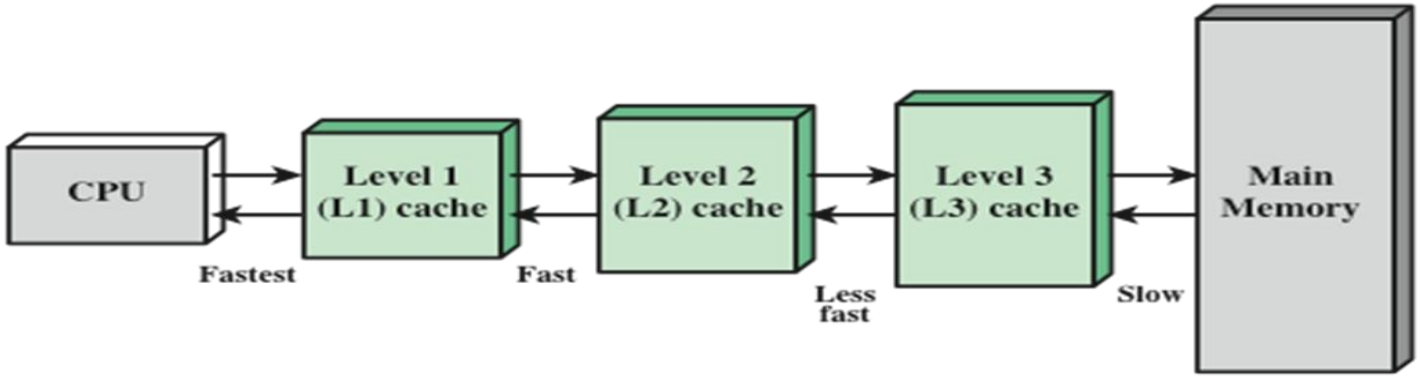
The small fast **L1 cache** has room for one block

- The big slow main memory has room for many blocks
- Each block contains 4-word
- (namely a, b, c, d)
- Size of 1 word (ex: a) = 4 bytes

Cache and Main Memory

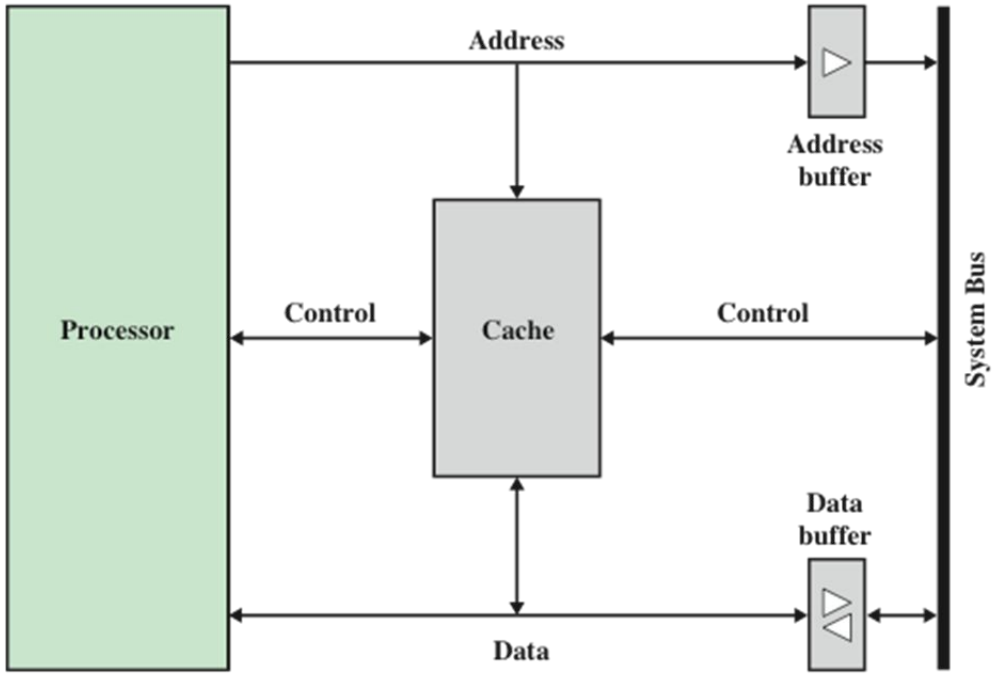


(a) Single cache



(b) Three-level cache organization

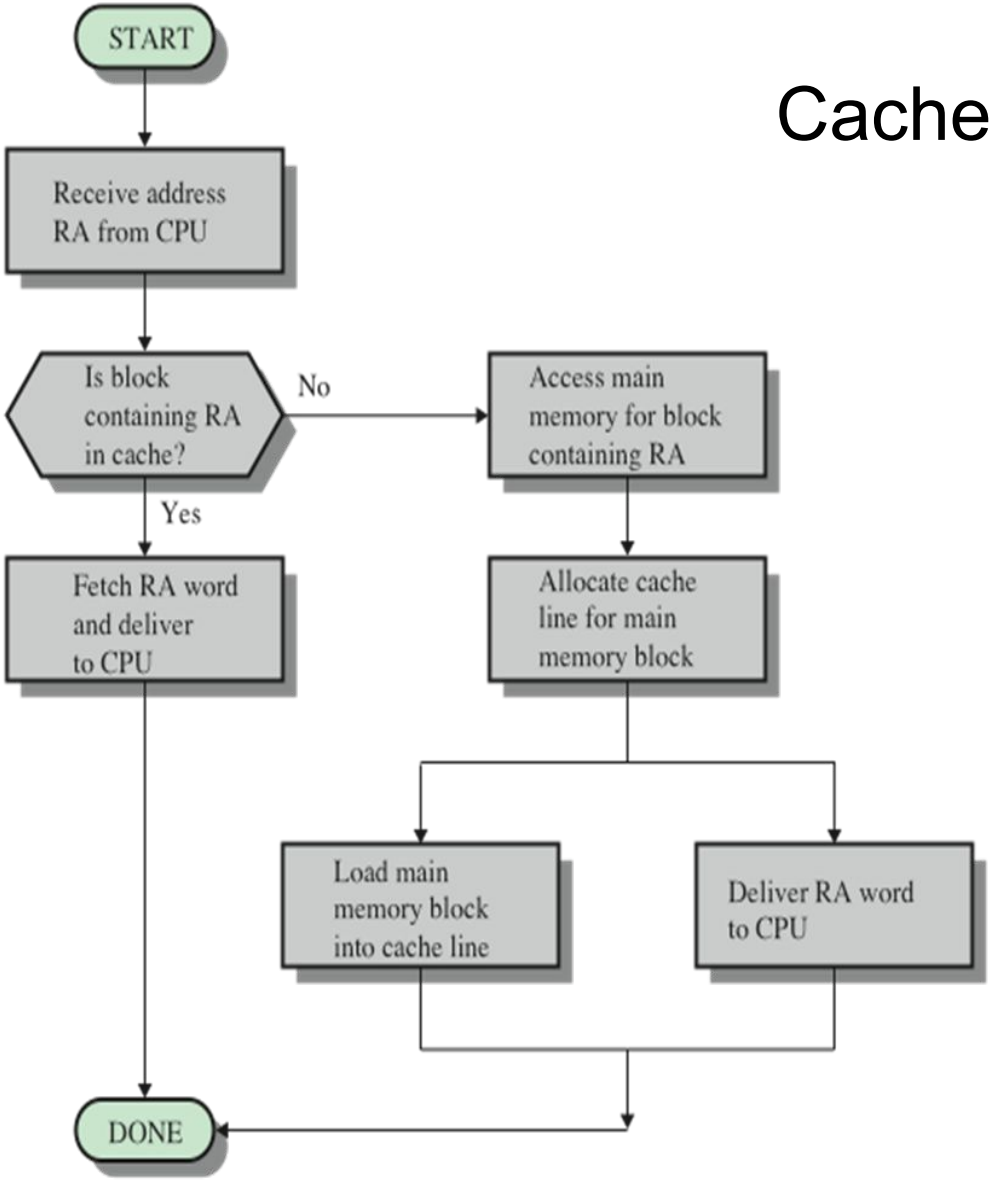
Typical Cache Organization



When a cache hit occurs, the data and address buffers are disabled and **communication is only between processor and cache**, with no system bus traffic

for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

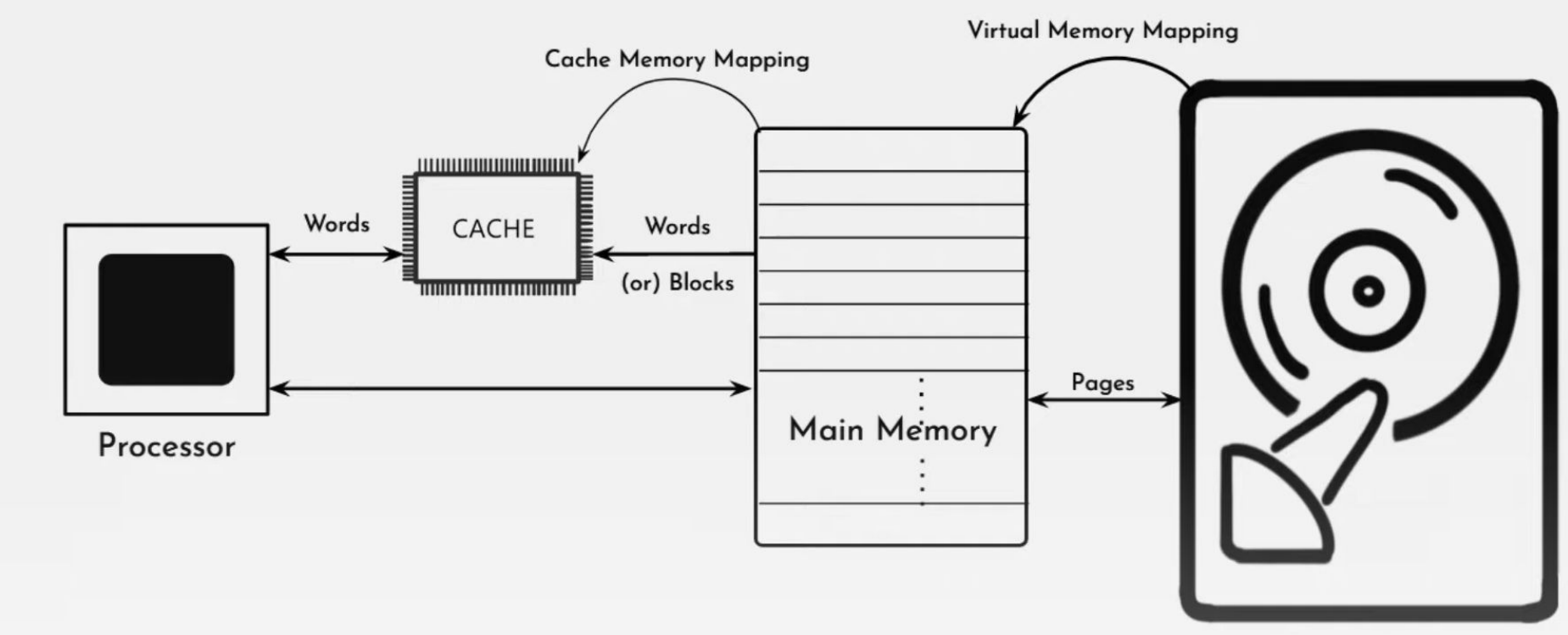
Cache Read Operation



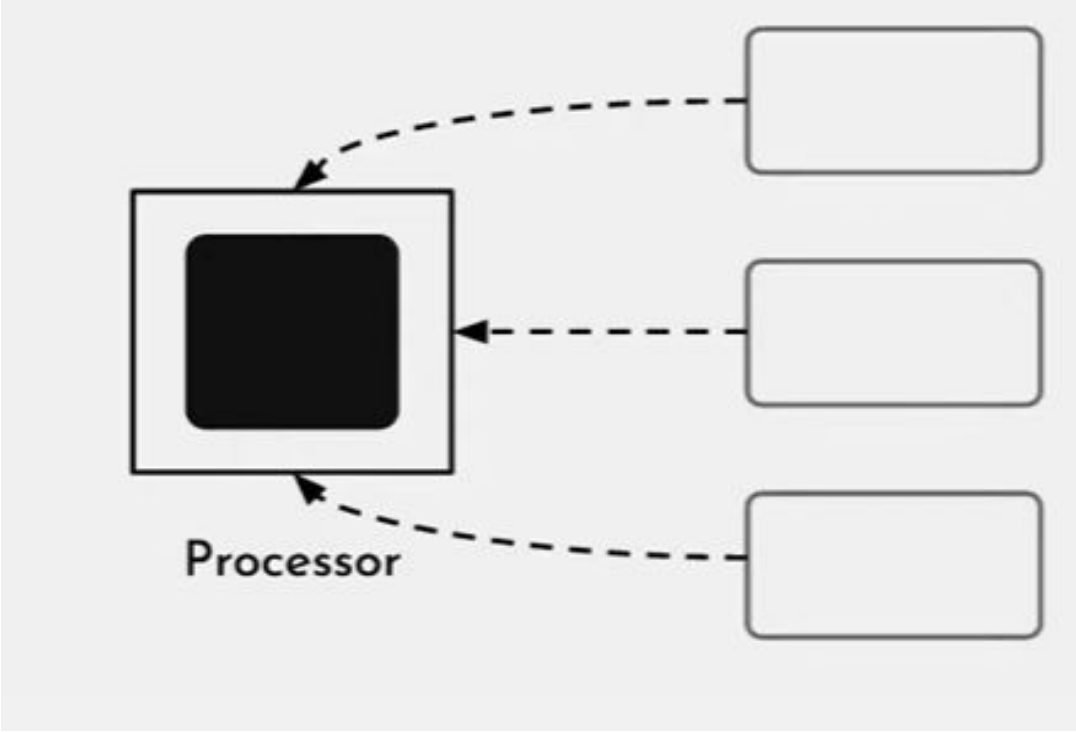
The processor generates the read address (RA) of a word to be read

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

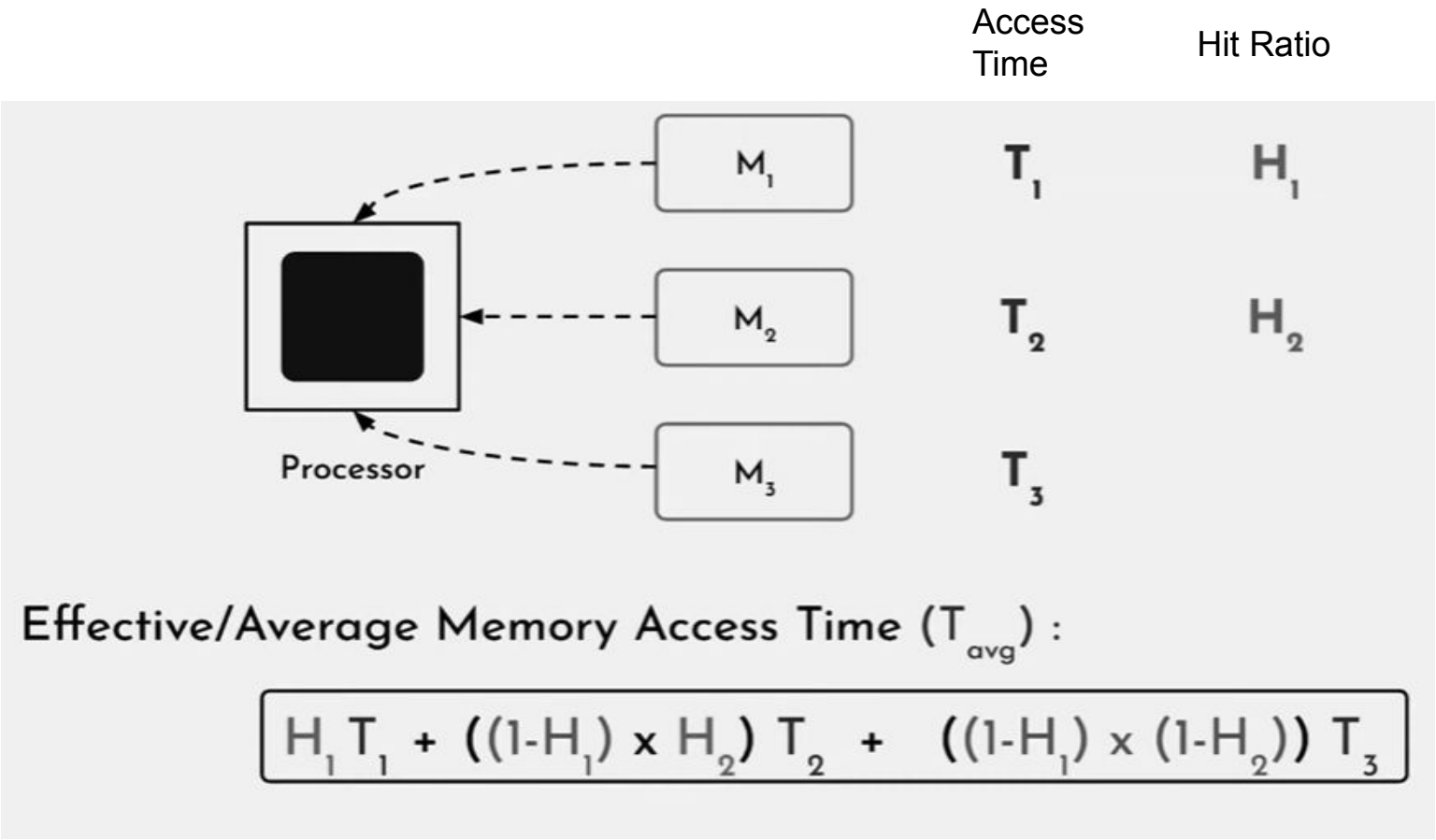
The Big Picture



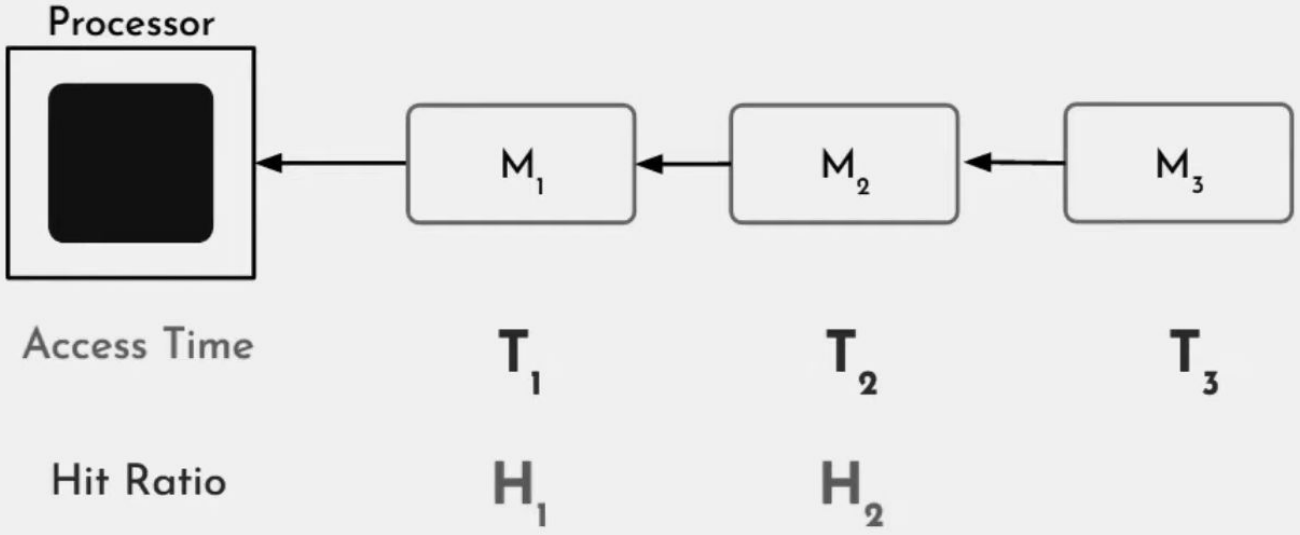
Memory Interfacing



Memory Interfacing



Memory Interfacing



Effective/Average Memory Access Time (T_{avg}) :

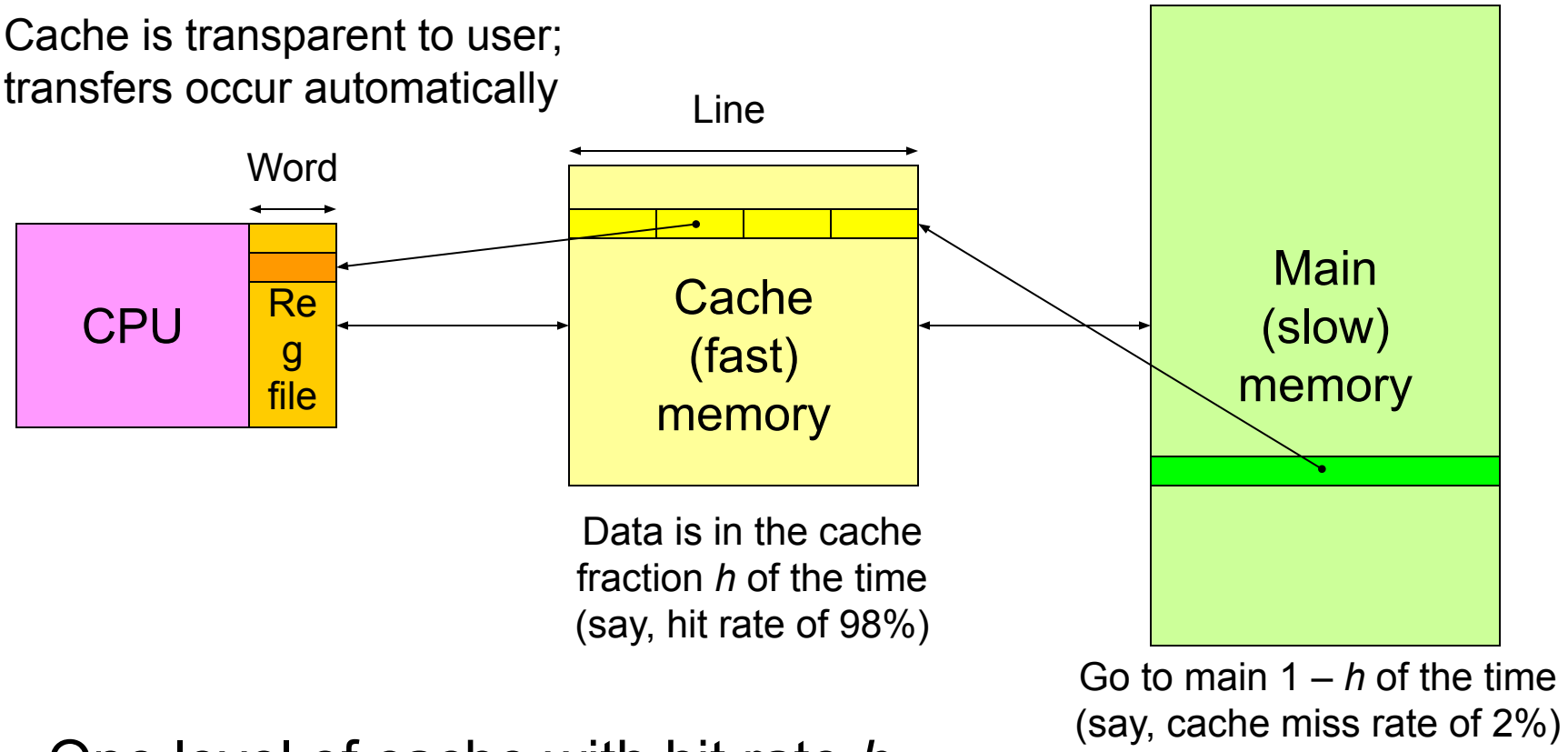
$$H_1 T_1 + ((1-H_1) \times H_2)(T_1+T_2) + ((1-H_1) \times (1-H_2))(T_1+T_2+T_3)$$

Problems

A cache memory needs an access time of 30 ns and main memory 150 ns, what is the average access time of CPU (assume hit ratio = 80%)?

Assume that for a certain processor, a read request takes 50 nanoseconds on a cache miss and 5 nanoseconds on a cache hit. Suppose while running a program, it was observed that 80% of the processor's read requests result in a cache hit. The average read access time in nanoseconds is

Cache, Hit/Miss Rate, and Effective Access Time



One level of cache with hit rate h

$$C_{eff} = hC_{fast} + (1 - h)(C_{slow} + C_{fast}) = C_{fast} + (1 - h)C_{slow}$$

Multiple Cache Levels

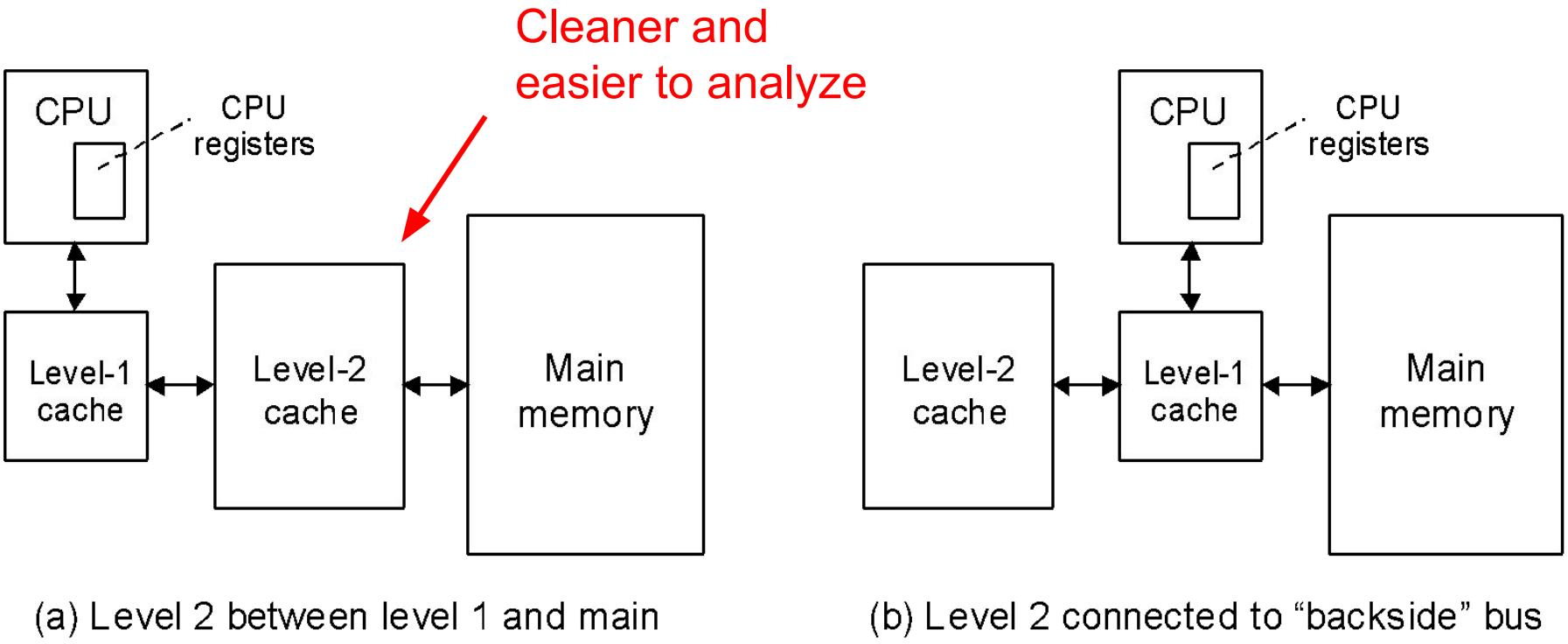


Fig. 18.1 Cache memories act as intermediaries between the superfast processor and the much slower main memory.

Cache Memory Design Parameters

Cache size (in bytes or words). A larger cache can hold more of the program's useful data but is more costly and likely to be slower.

Block or *cache-line size* (unit of data transfer between cache and main). With a larger cache line, more data is brought in cache with each miss. This can improve the hit rate but also may bring low-utility data in.

Placement policy. Determining where an incoming cache line is stored. More flexible policies imply higher hardware cost and may or may not have performance benefits (due to more complex data location).

Replacement policy. Determining which of several existing cache blocks (into which a new cache line can be mapped) should be overwritten. Typical policies: choosing a random or the least recently used block.

Write policy. Determining if updates to cache words are immediately forwarded to main (*write-through*) or modified blocks are copied back to main if and when they must be replaced (*write-back* or *copy-back*).

What Makes a Cache Work?

Temporal locality
Spatial locality

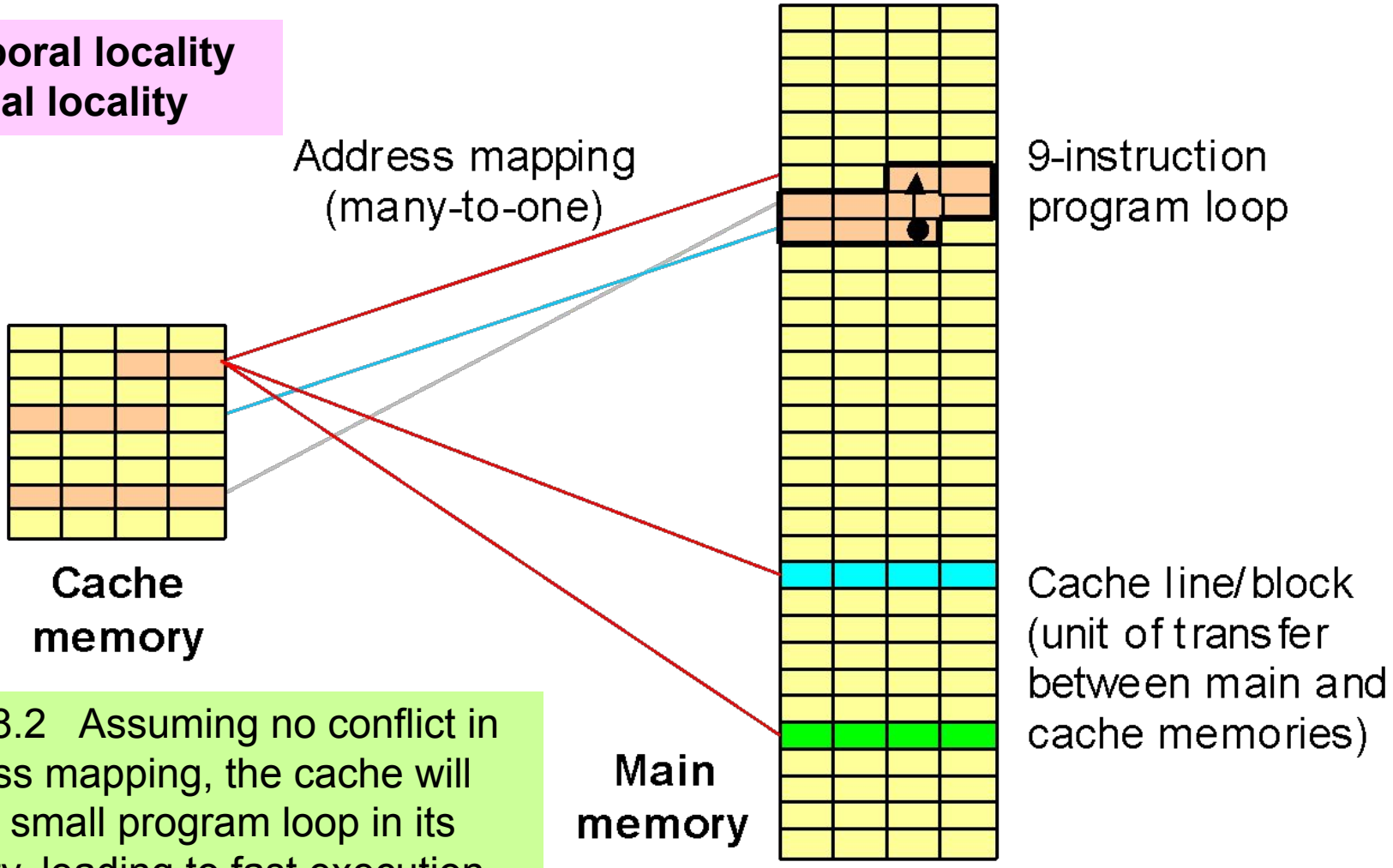


Fig. 18.2 Assuming no conflict in address mapping, the cache will hold a small program loop in its entirety, leading to fast execution.

Desktop, Drawer, and File Cabinet Analogy

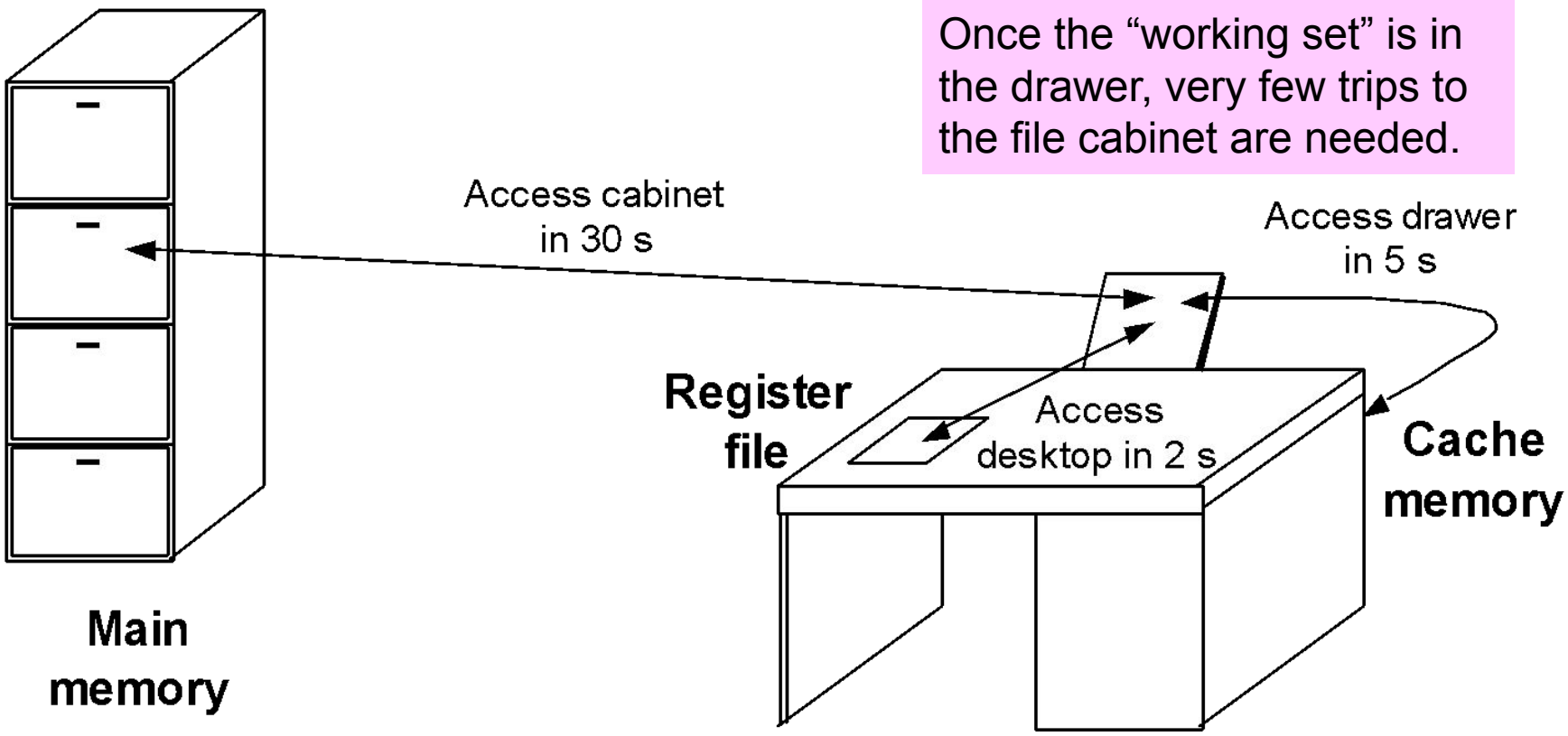


Fig. 18.3 Items on a desktop (register) or in a drawer (cache) are more readily accessible than those in a file cabinet (main memory).

Compulsory, Capacity, and Conflict Misses

Compulsory misses: With *on-demand fetching*, first access to any item is a miss. Some "compulsory" misses can be avoided by prefetching.

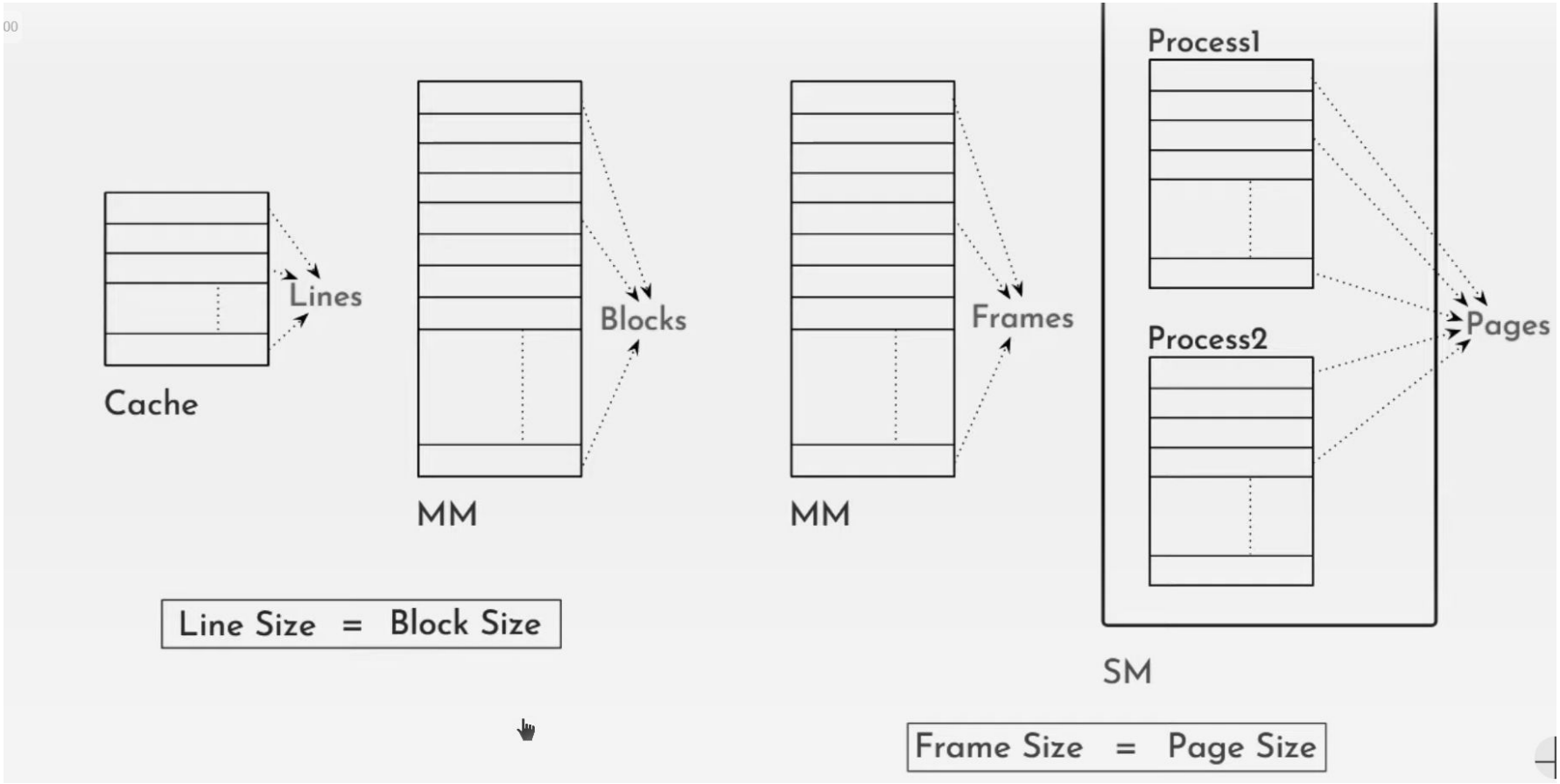
Capacity misses: We have to oust some items to make room for others. This leads to misses that are not incurred with an infinitely large cache.

Conflict misses: Occasionally, there is free room, or space occupied by useless data, but the mapping/placement scheme forces us to displace useful items to bring in other items. This may lead to misses in future.

Given a fixed-size cache, dictated, e.g., by cost factors or availability of space on the processor chip, compulsory and capacity misses are pretty much fixed. Conflict misses, on the other hand, are influenced by the data mapping scheme which is under our control.

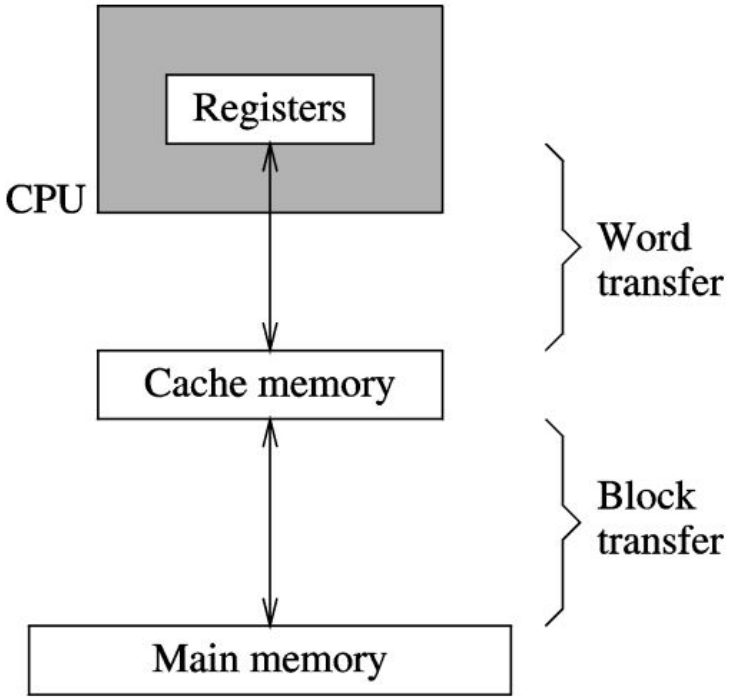
We study two popular mapping schemes: direct and set-associative.

Cache Memory



Cache Memory

- Transfer between main memory and cache
  - In units of blocks
  - Implements spatial locality
- Transfer between main memory and cache
  - In units of words
- Need policies for
  - Block placement
  - Mapping function
  - Block replacement
  - Write policies



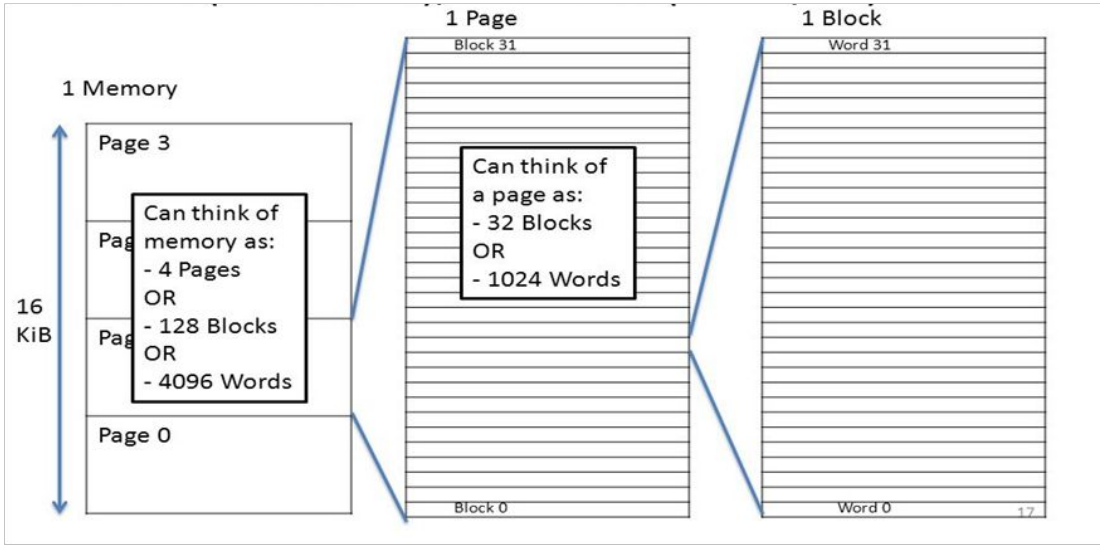
Now let us see the terms

Memory Capacity

- **Word** - an addressable unit

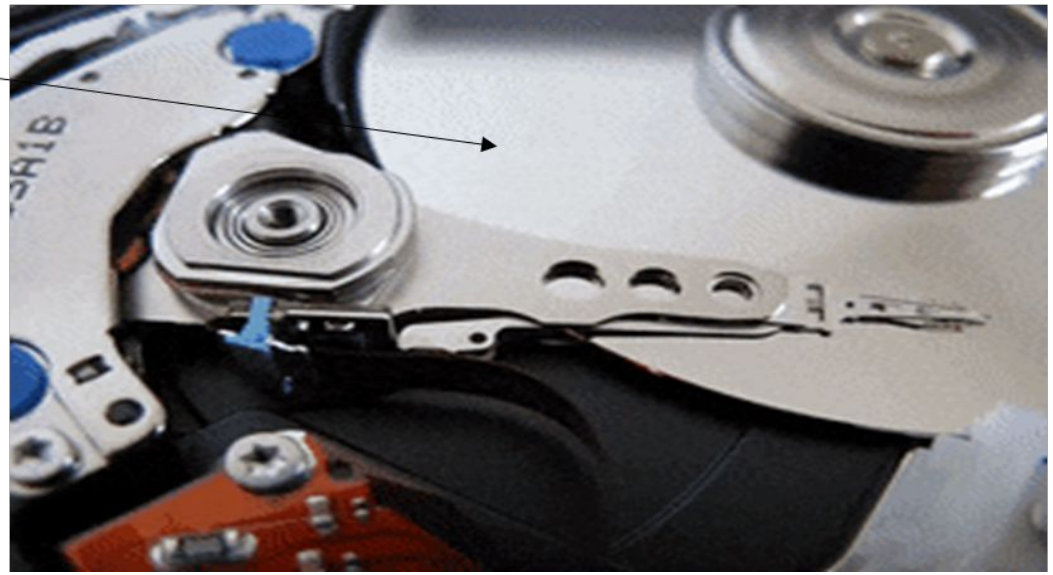
Unit of Transfer

- Word
- Block- Main memory words can be divided into blocks



Accessing Method

- Sequential Access
- Non- sequential access (RAM)

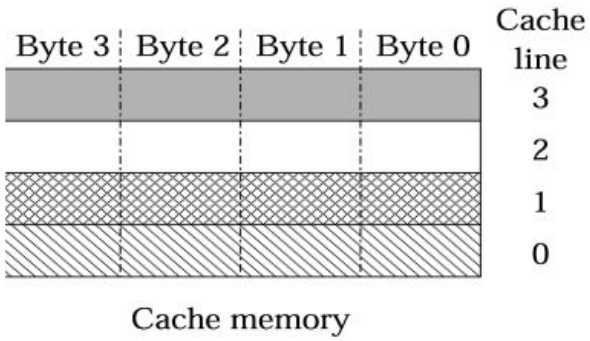


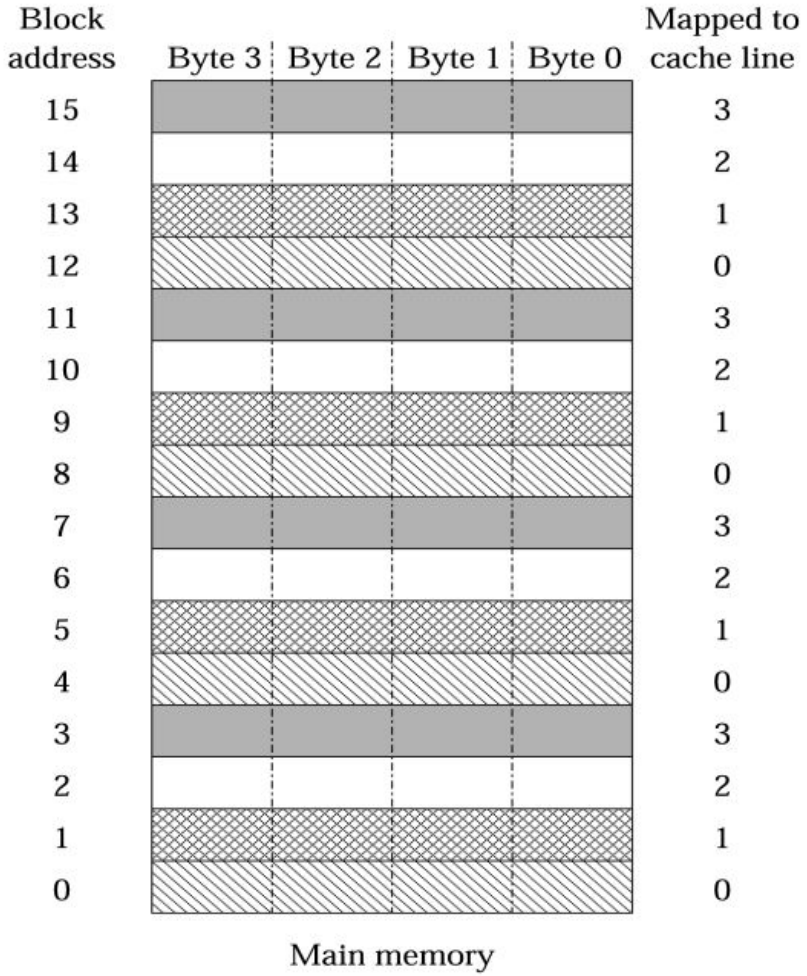
Mapping Functions

- Determines how memory blocks are mapped to cache lines
- Three types
  - * Direct mapping
    - » Specifies a single cache line for each memory block
  - * Set-associative mapping
    - » Specifies a set of cache lines for each memory block
  - * Associative mapping
    - » No restrictions
      - – Any cache line can be used for any memory block

Mapping Functions (Contd..)

Direct mapping example





Direct Memory Mapping

Main Memory Size : 64 words i.e. (0, 1, . . . ,63)

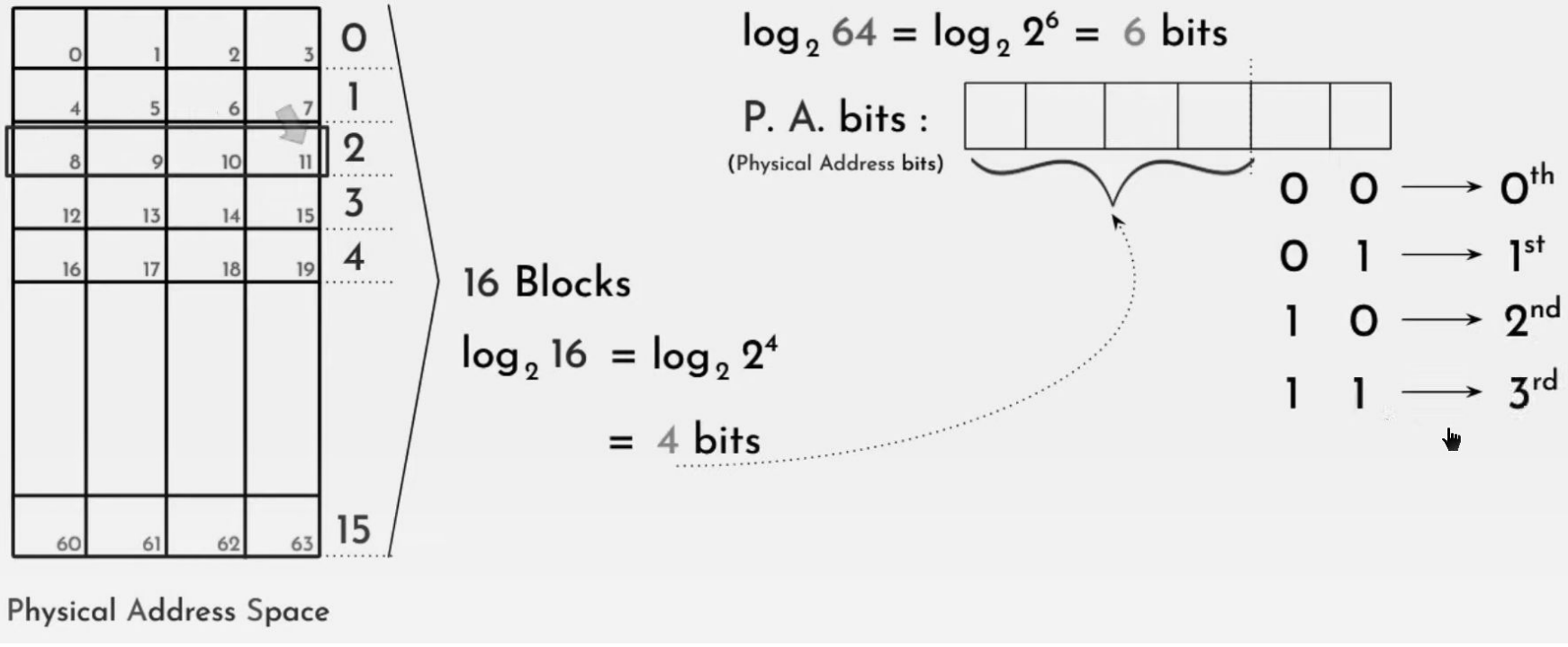
Block Size : 4 words

No. of Blocks in MM : 64 / 4 = 16

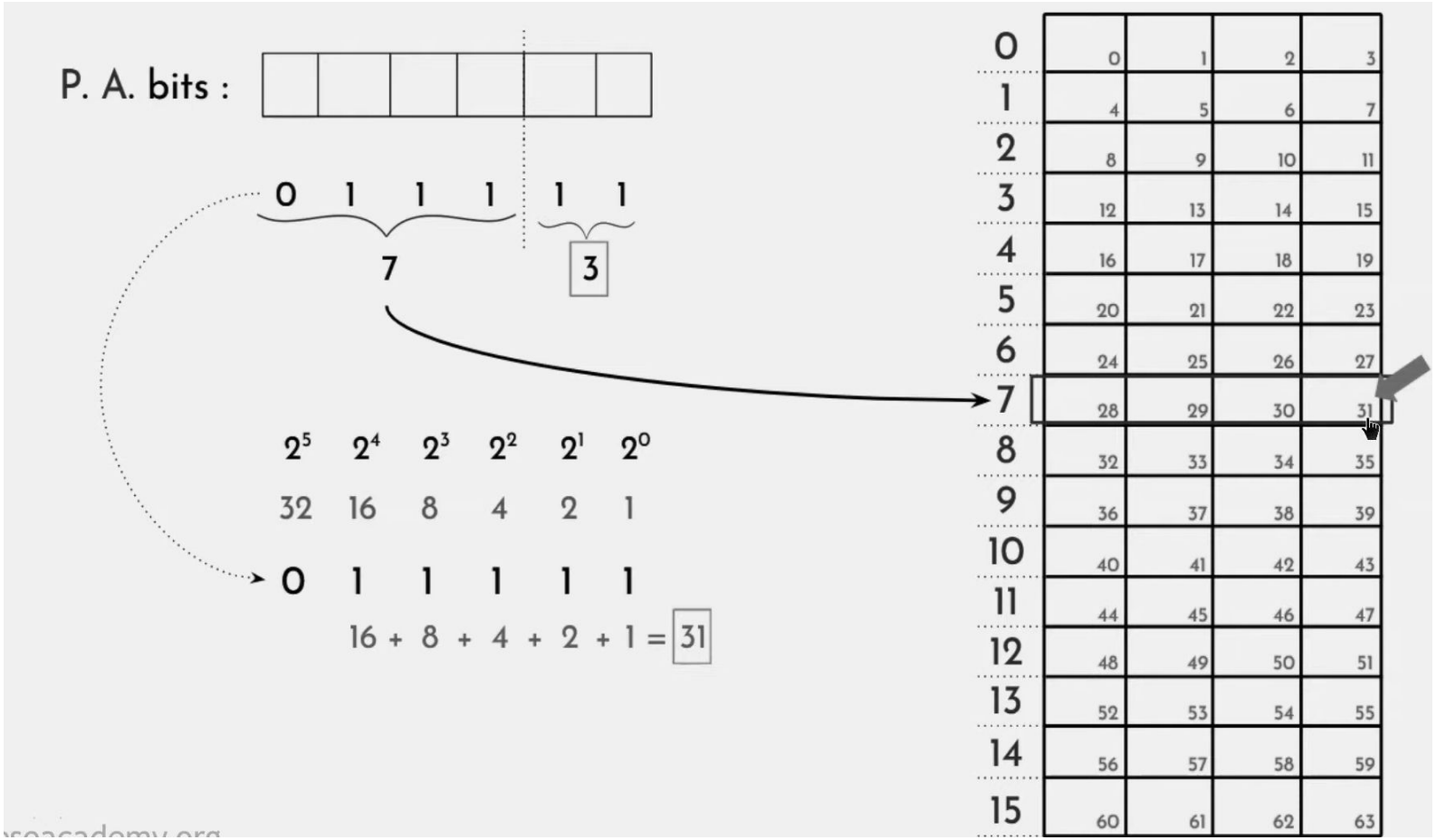
0 0 0	→	m_0	64 words	
0 0 1	→	m_1	$\log_2 64 = \log_2 2^6$	
0 1 0	→	m_2	= 6 bits	
0 1 1	→	m_3		
1 0 0	→	m_4		
1 0 1	→	m_5		
1 1 0	→	m_6		
1 1 1	→	m_7		

0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23
6	24	25	26	27
7	28	29	30	31
8	32	33	34	35
9	36	37	38	39
10	40	41	42	43
11	44	45	46	47
12	48	49	50	51
13	52	53	54	55
14	56	57	58	59
15	60	61	62	63

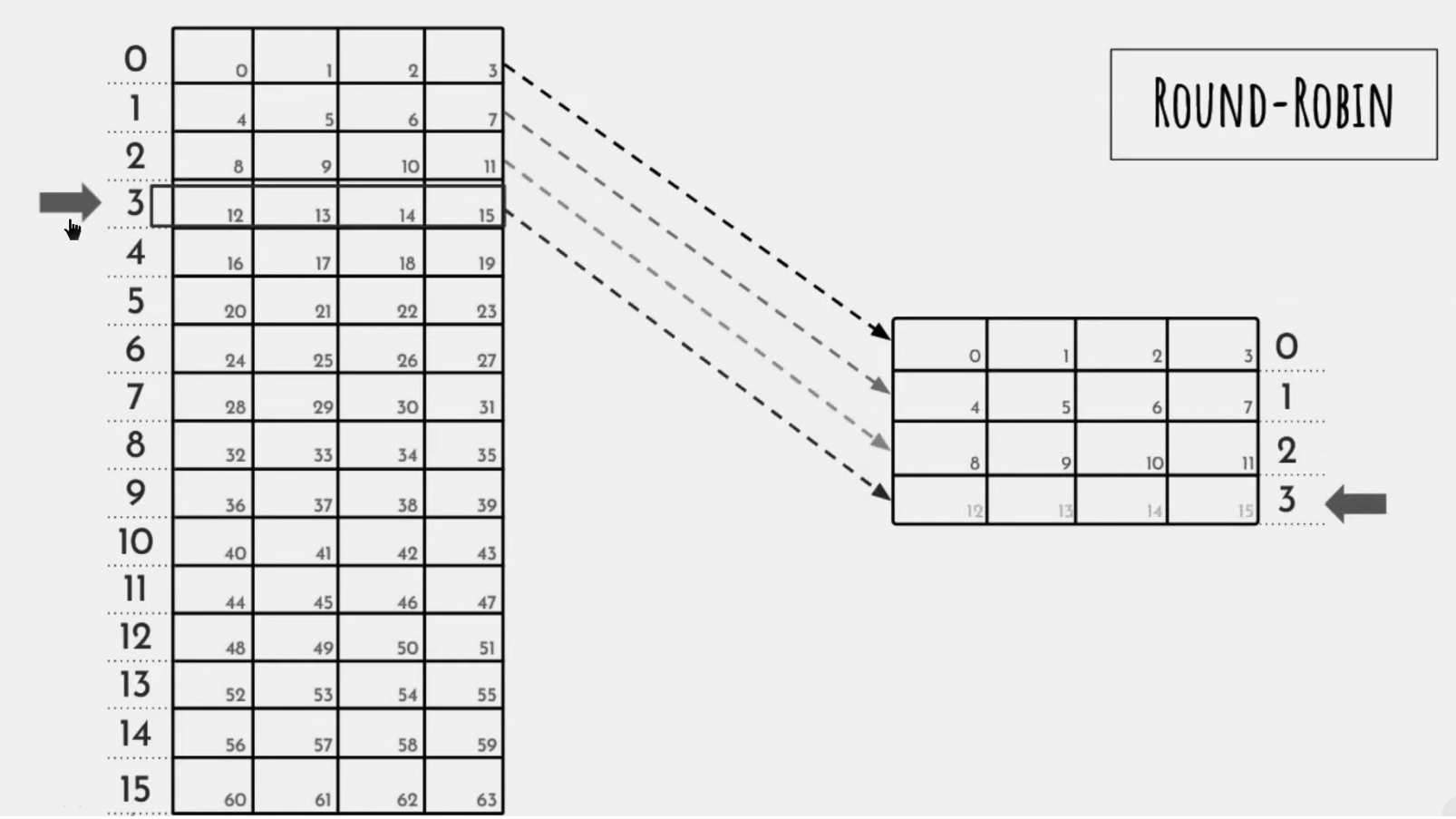
Direct Memory Mapping



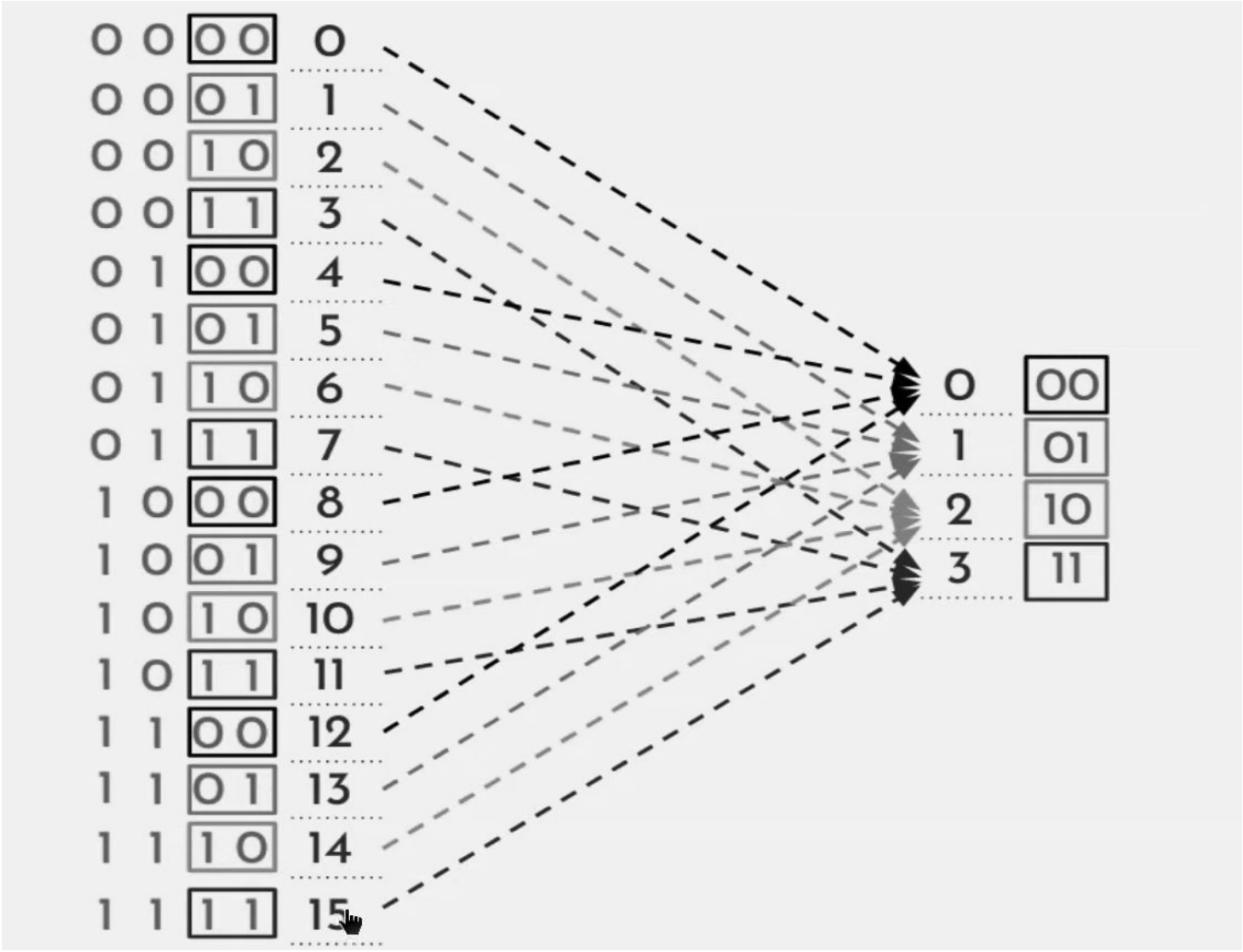
Direct Memory Mapping



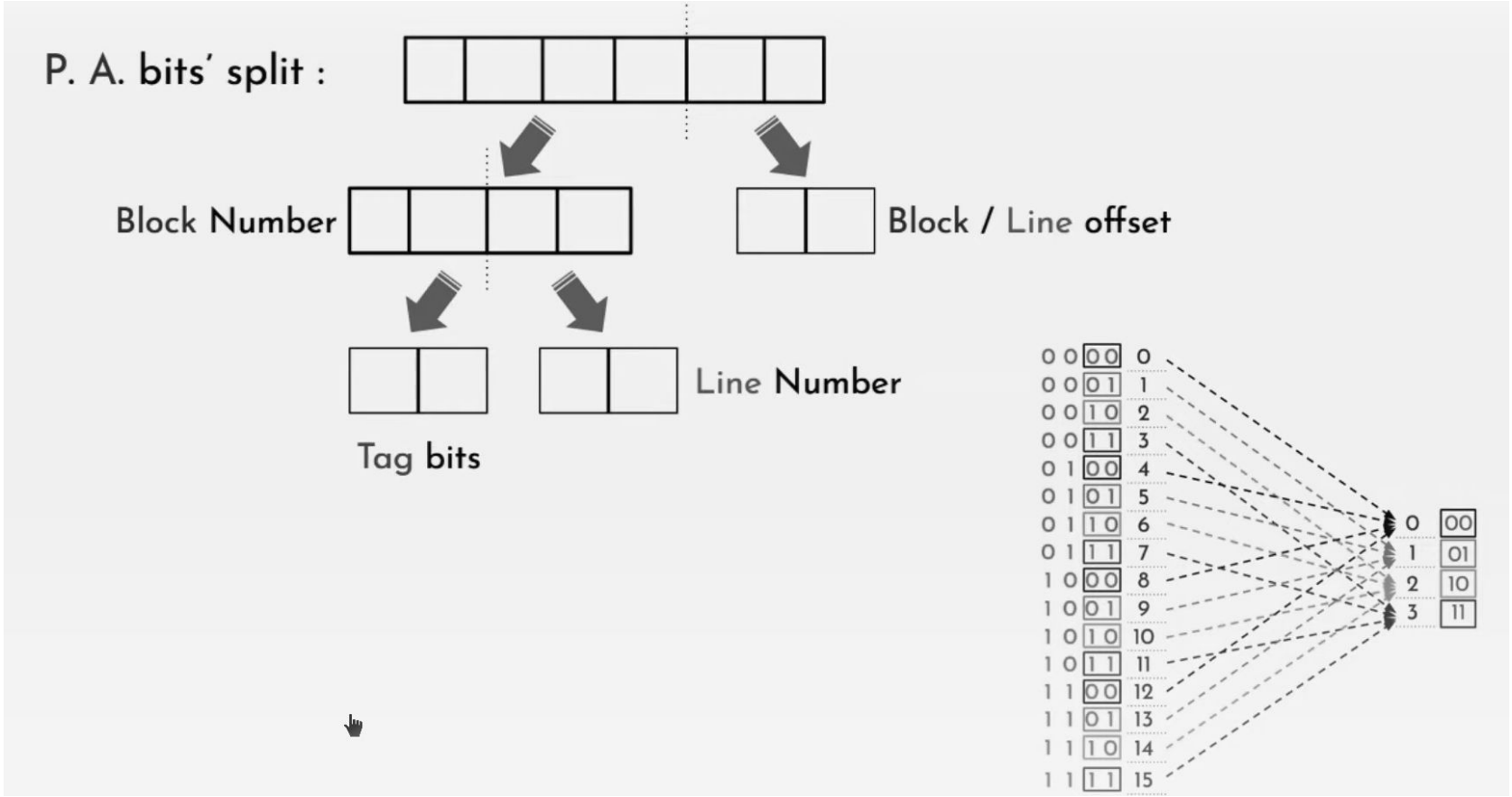
Direct Memory Mapping



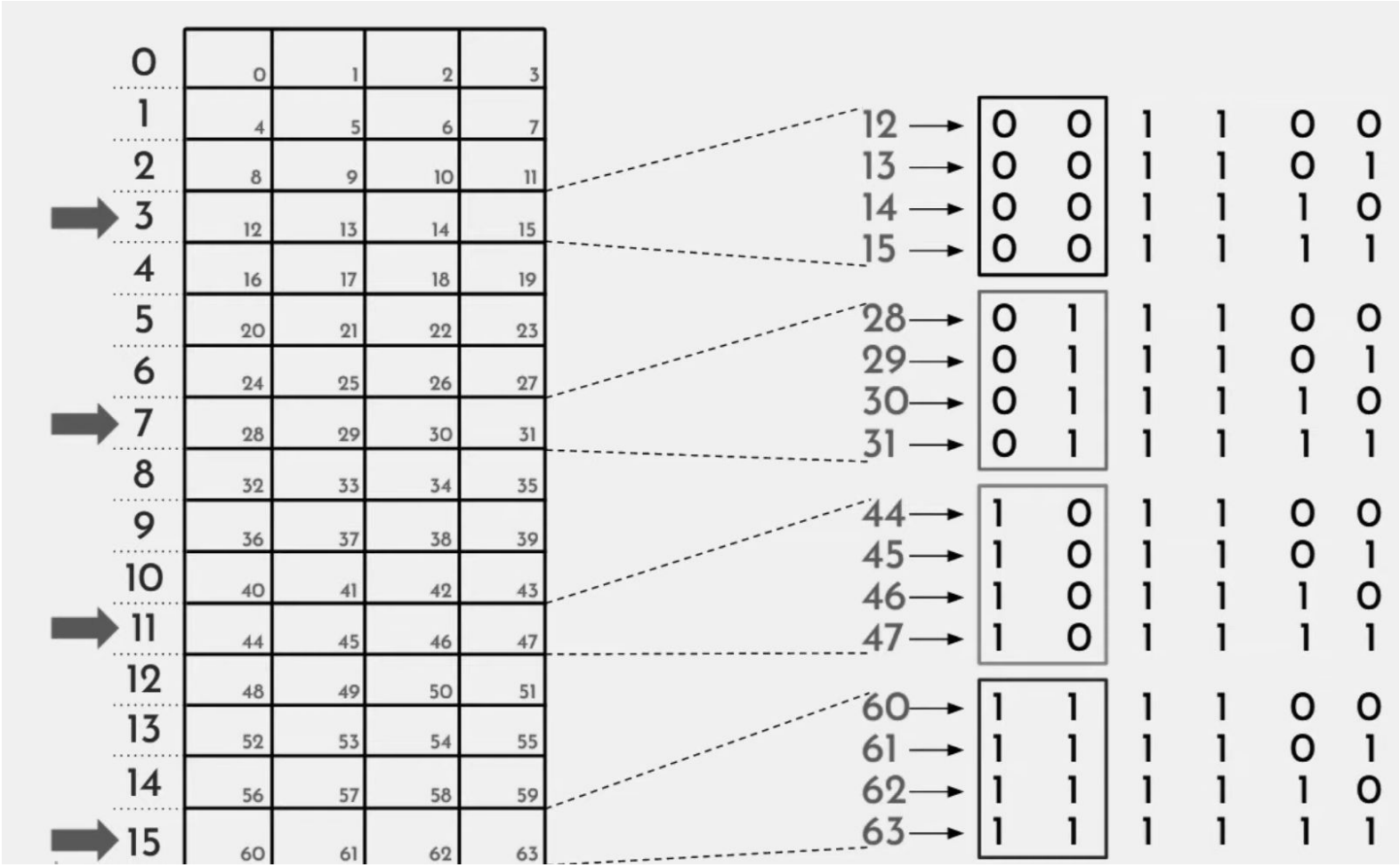
Direct Memory Mapping



Direct Memory Mapping



Direct Memory Mapping



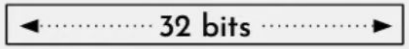
Direct Memory Mapping

Ex 1: MM Size: 4 GB
Cache Size: 1 MB
Block Size: 4 KB

1. P.A. bits' split?
2. Tag directory size?

Sol. MM Size = 4 GB = $2^2 \times 2^{30}$ B = $2^{(2+30)}$ B = 2^{32} B

∴ No. of P.A. bits = $\log_2 2^{32}$ = 32

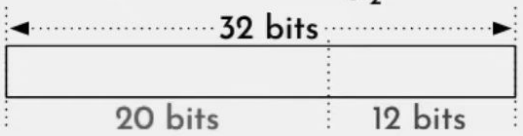


Block Size = 4 KB = $2^2 \times 2^{10}$ B = 2^{12} B

No. of Blocks in MM = $2^{32} / 2^{12}$ = 2^{20}

∴ Block number bits = $\log_2 2^{20}$ = 20

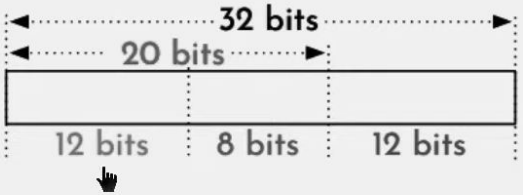
∴ Block offset = $\log_2 2^{12}$ = 12



Cache Size = 1 MB = 1 x 2^{20} B = 2^{20} B

No. of Lines in Cache = $2^{20} / 2^{12}$ = 2^8

∴ Line number bits = $\log_2 2^8$ = 8



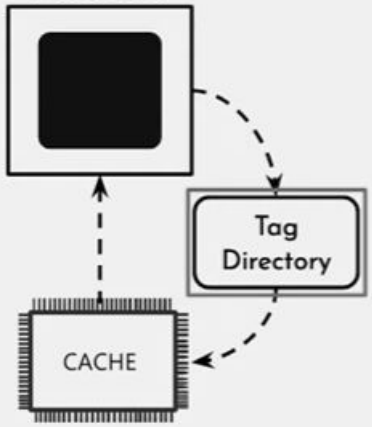
No. of Tag bits : P.A. bits - (Line no. bits + offset) = 32 - (8+12) = 12

Direct Memory Mapping

Ex 1: MM Size: 4 GB
Cache Size: 1 MB
Block Size: 4 KB

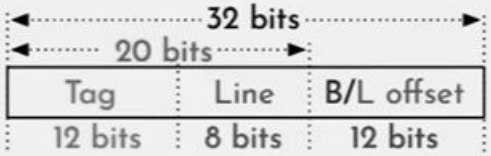
1. P.A. bits' split?
2. Tag directory size?

Sol.



Tag directory:

- Keeps primarily the record of Tag bits, cache line-wise.
- No. of entries = No. of Cache lines.



No. of Lines in Cache = 2^8

No. of Tag bits = 12

Tag directory size = 2^8 x 12 bits = 3072 bits

Mapping Functions

- Implementing direct mapping
  - * Easier than the other two
  - * Maintains three pieces of information
    - » Cache data
      - – Actual data
    - » Cache tag
      - – Problem: More memory blocks than cache lines
        - ✈Several memory blocks are mapped to a cache line
      - – Tag stores the address of memory block in cache line
    - » Valid bit
      - – Indicates if cache line contains a valid block

Direct-Mapped Cache

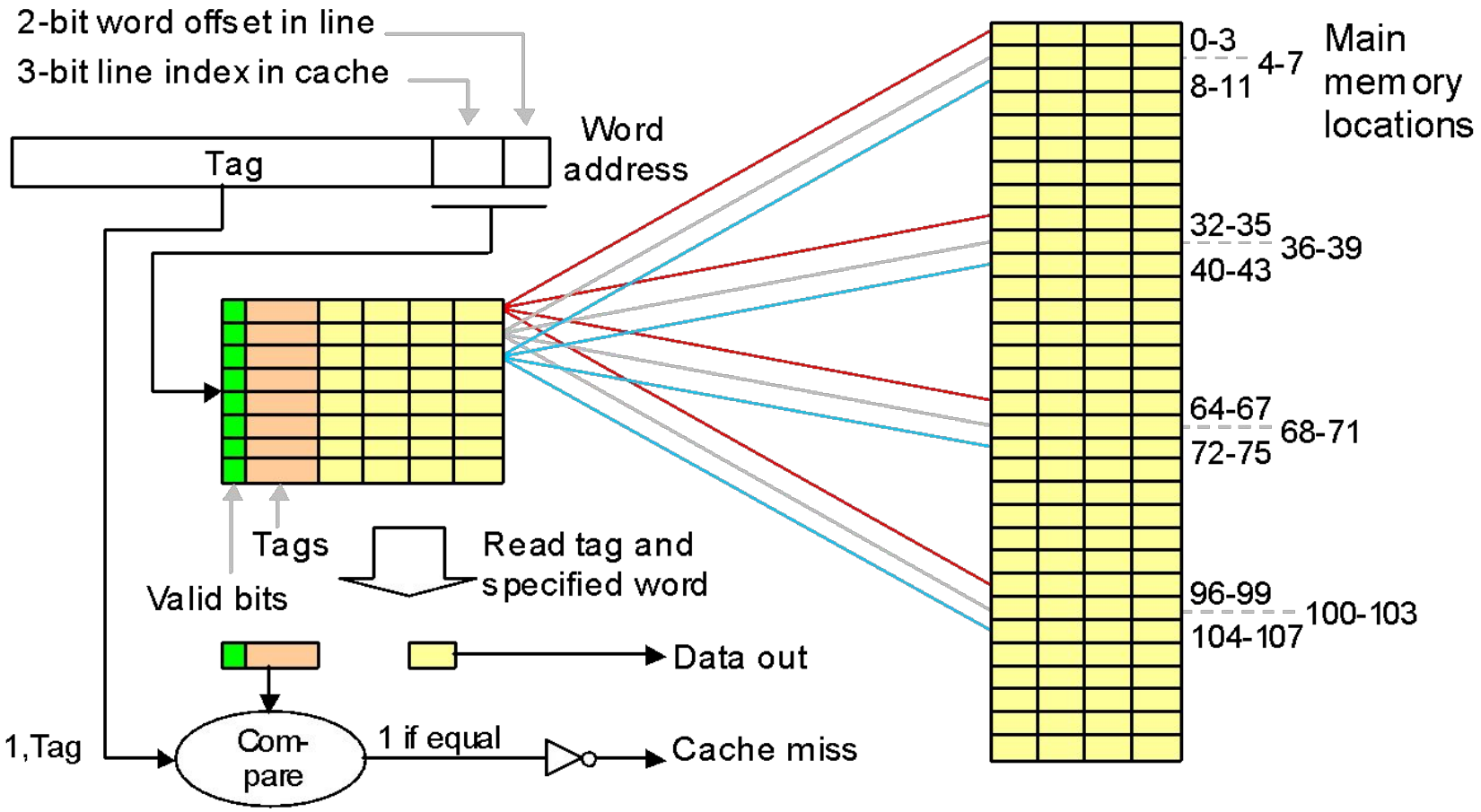


Fig. 18.4 Direct-mapped cache holding 32 words within eight 4-word lines. Each line is associated with a tag and a valid bit.

Accessing a Direct-Mapped Cache

Example

Show cache addressing for a byte-addressable memory with 32-bit addresses. Cache line W = 16 B. Cache size L = 4096 lines (64 KB).

Solution

Byte offset in line is $\log_2 16 = 4$ b. Cache line index is $\log_2 4096 = 12$ b. This leaves 32 – 12 – 4 = 16 b for the tag.

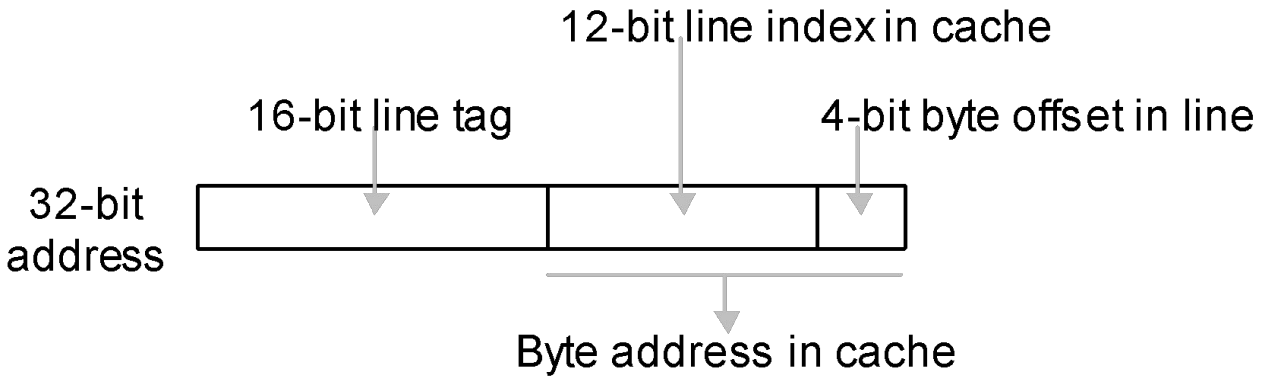


Fig. Components of the 32-bit address in an example direct-mapped cache with byte addressing.

Practice Problems

1. Consider a direct mapped cache of size 16 KB with block size 256 bytes. The size of main memory is 128 KB. Find-

   Number of bits in tag

   Tag directory size
2. Consider a direct mapped cache of size 512 KB with block size 1 KB. There are 7 bits in the tag. Find-

   Size of main memory

   Tag directory size
3. Consider a direct mapped cache with block size 4 KB. The size of main memory is 16 GB and there are 10 bits in the tag. Find-

   Size of cache memory

   Tag directory size

Practice Problems

4. Consider a direct mapped cache of size 32 KB with block size 32 bytes. The CPU generates 32 bit addresses. The number of bits needed for cache indexing and the number of tag bits are respectively __ and __ ?

5. Consider a machine with a byte addressable main memory of 232 bytes divided into blocks of size 32 bytes. Assume that a direct mapped cache having 512 cache lines is used with this machine. The size of the tag field in bits is _____.

Direct Mapping Cache Organization

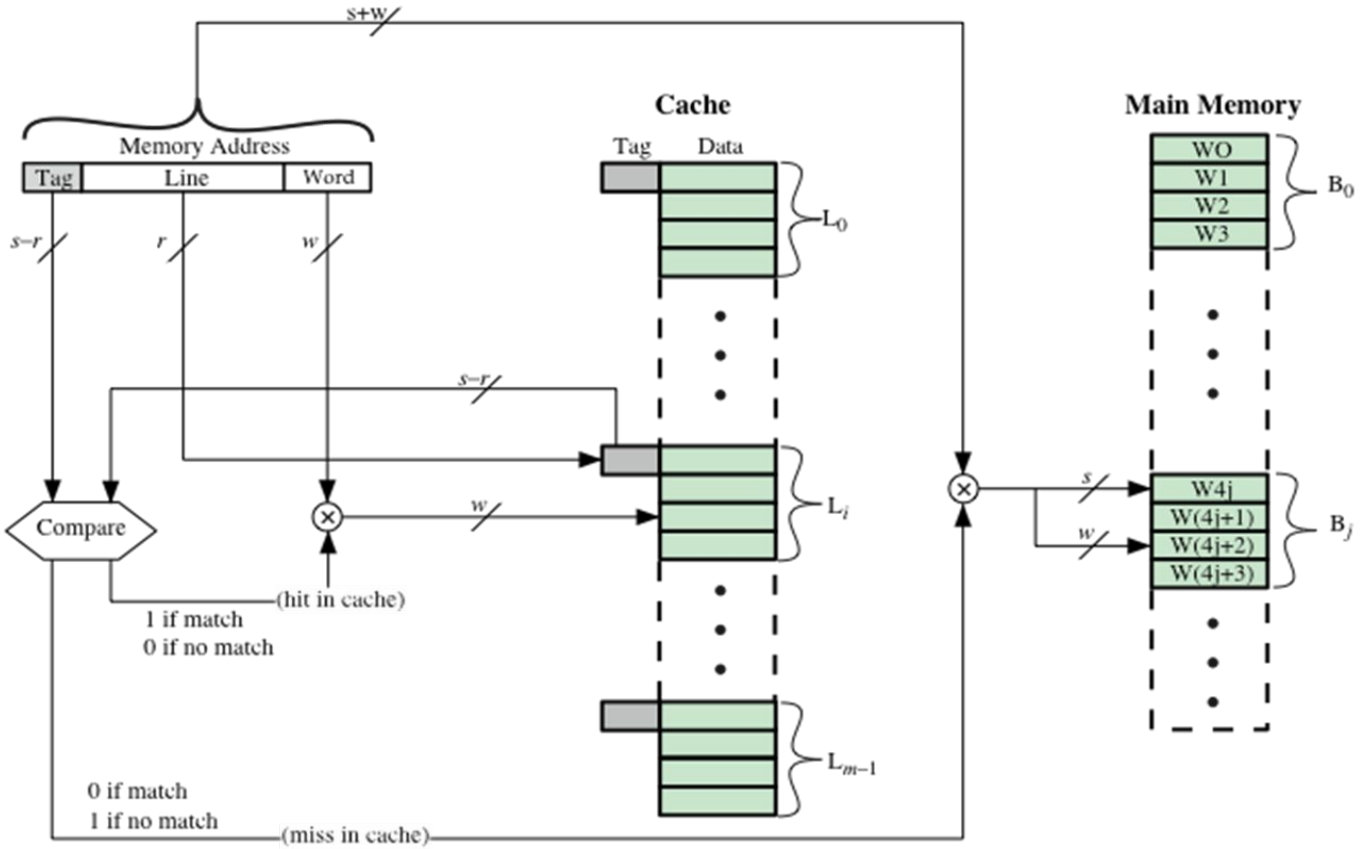
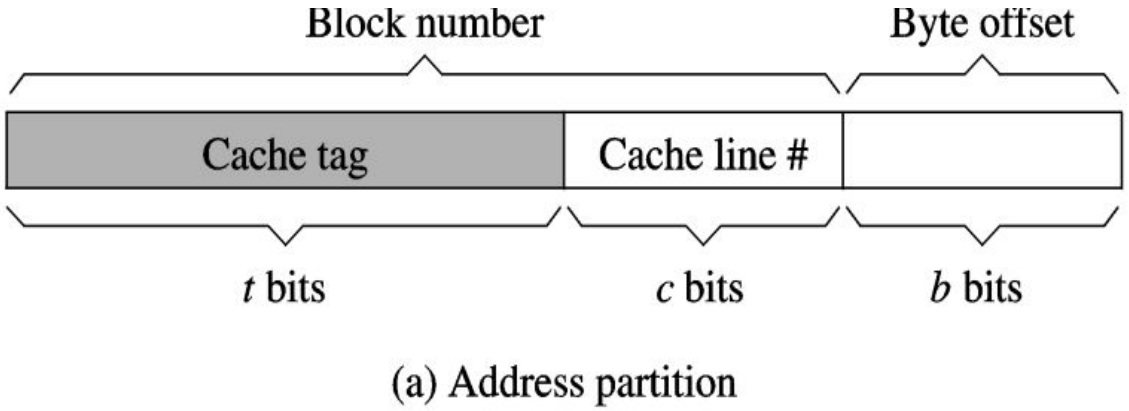


Figure 4.9 Direct-Mapping Cache Organization

Direct Mapping



(a) Address partition

Valid bit	Cache tag	Cache data	Cache line
0	???	??? ??? ??? ???	3
1	Valid tag	4-bytes of valid cache data	2
1	Valid tag	4-bytes of valid cache data	1
1	Valid tag	4-bytes of valid cache data	0

Direct Mapping

Q: Consider a machine with a byte addressable main memory of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system.

A 50 × 50 two-dimensional array of bytes is stored in the main memory starting from memory location 1100H. Assume that the data cache is initially empty. The complete array is accessed twice. Assume that the contents of the data cache do not change in between the two accesses.

Sol. MM Size = 2^{16} B ∴ P.A. bits = 16

No. of cache lines = 32

∴ Line no. bits = $\log_2 32$ = 5

Block Size = 64 B = 2^6 B ∴ Offset = 6

arr[0][0], arr[0][1], ... ,arr[0][15]
arr[1][0] , arr[1][1], ... ,arr[1][15]
. . . . . .
arr[49][0],arr[49][1], . . . ,arr[49][49]

Total no. of elements = 50 x 50 = 2500

Element Size = 1 B

∴ Array Size = 2500 x 1 B = 2500 B

No. of blocks to store the array

= 2500 B / 64 B = 39.0625

Direct Mapping

Q: Consider a machine with a byte addressable main memory of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system.

A 50 × 50 two-dimensional array of bytes is stored in the main memory starting from memory location 1100H. Assume that the data cache is initially empty. The complete array is accessed twice. Assume that the contents of the data cache do not change in between the two accesses.

Sol.

arr[0][0],arr[0][1], ... ,arr[0][15]
arr[1][0] , arr[1][1], ... ,arr[1][15]
. . . . . .
arr[49][0],arr[49][1], . . . ,arr[49][49]

No. of blocks to store the array ≈ 40

1100H → 0001 0001 0000 0000

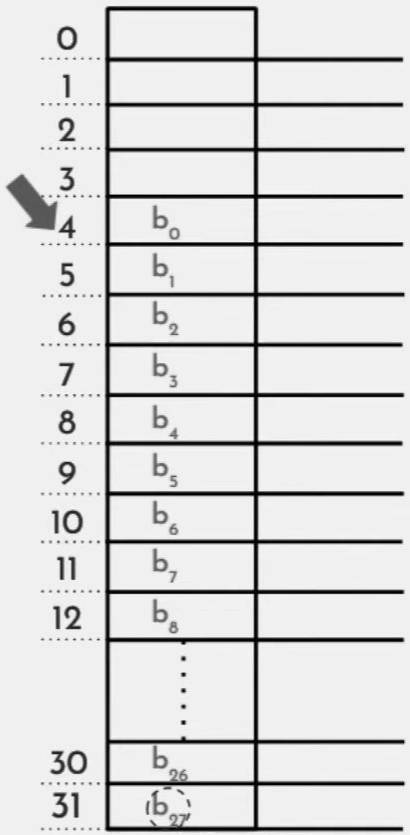
00010 00100 000000

Direct Mapping

Consider a machine with a byte addressable main memory of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system.

A 50 × 50 two-dimensional array of bytes is stored in the main memory starting from memory location 1100H. Assume that the data cache is initially empty. The complete array is accessed twice. Assume that the contents of the data cache do not change in between the two accesses.

Sol. 40 blocks i.e. $b_0, b_1, b_2, \ldots, b_{39}$

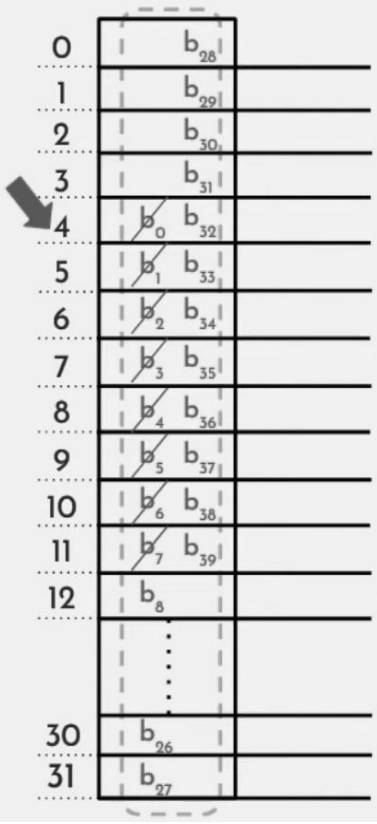


Direct Mapping

Q: Consider a machine with a byte addressable main memory of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system.

A 50 × 50 two-dimensional array of bytes is stored in the main memory starting from memory location 1100H. Assume that the data cache is initially empty. The complete array is accessed twice. Assume that the contents of the data cache do not change in between the two accesses.

Sol. 40 blocks i.e. $b_0, b_1, b_2, \ldots, b_{39}$



1st iteration:

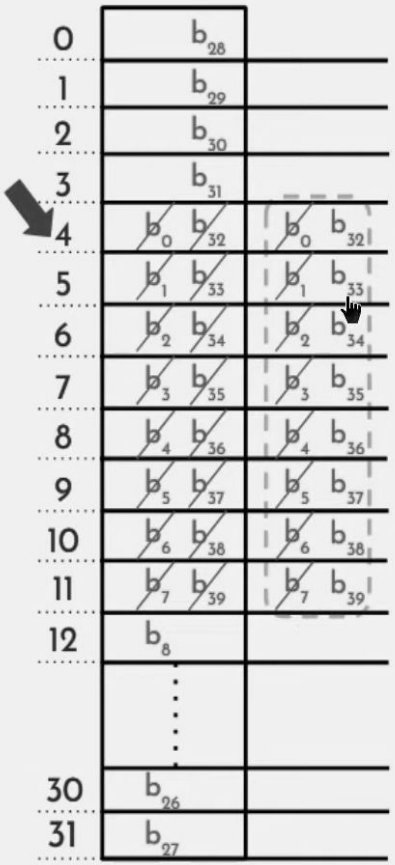
Cache miss = 40

Direct Mapping

Consider a machine with a byte addressable main memory of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system.

A 50 × 50 two-dimensional array of bytes is stored in the main memory starting from memory location 1100H. Assume that the data cache is initially empty. The complete array is accessed twice. Assume that the contents of the data cache do not change in between the two accesses.

Sol. 40 blocks i.e. $b_0, b_1, b_2, \ldots, b_{39}$



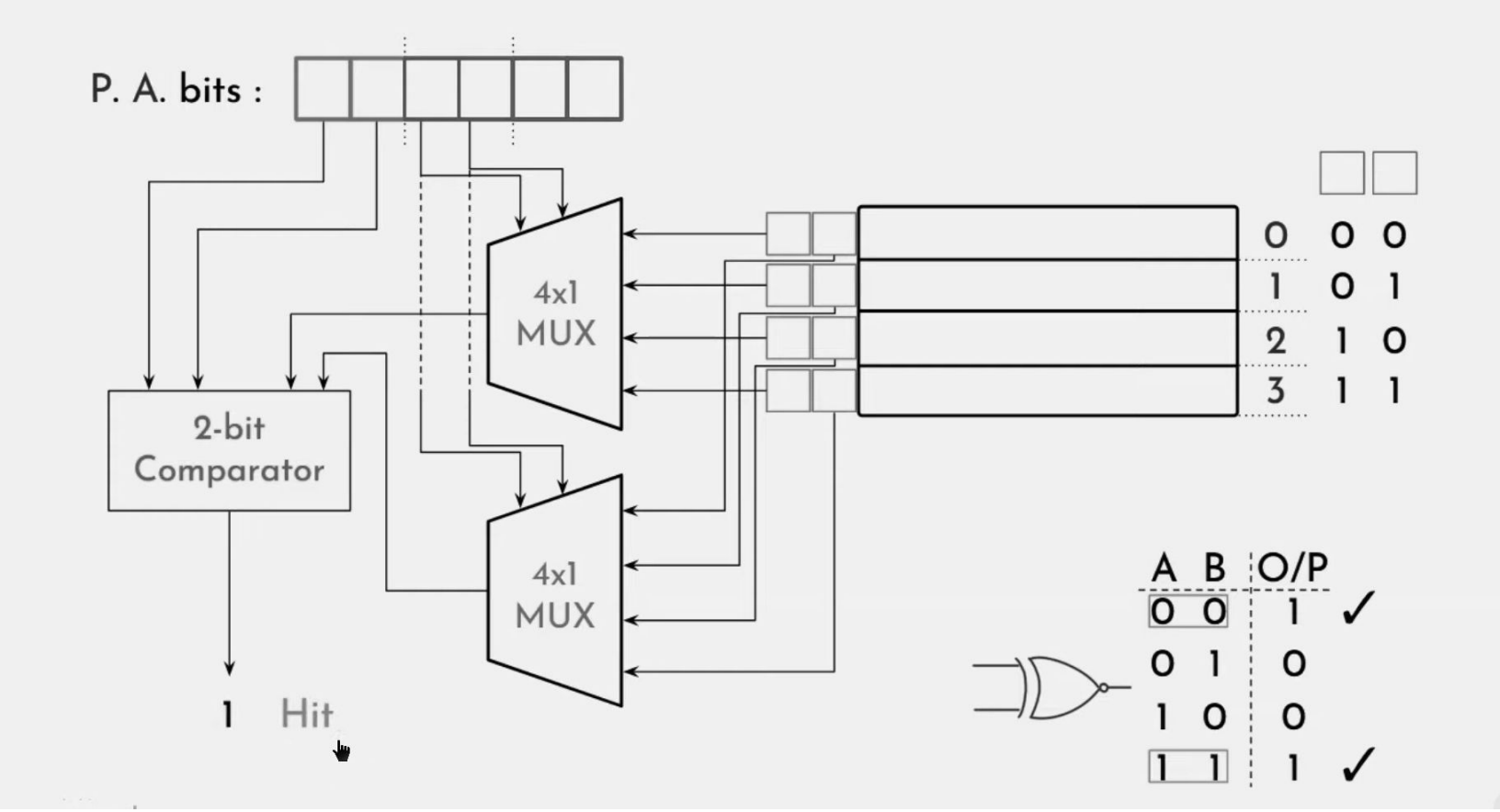
1st iteration:

Cache miss = 40

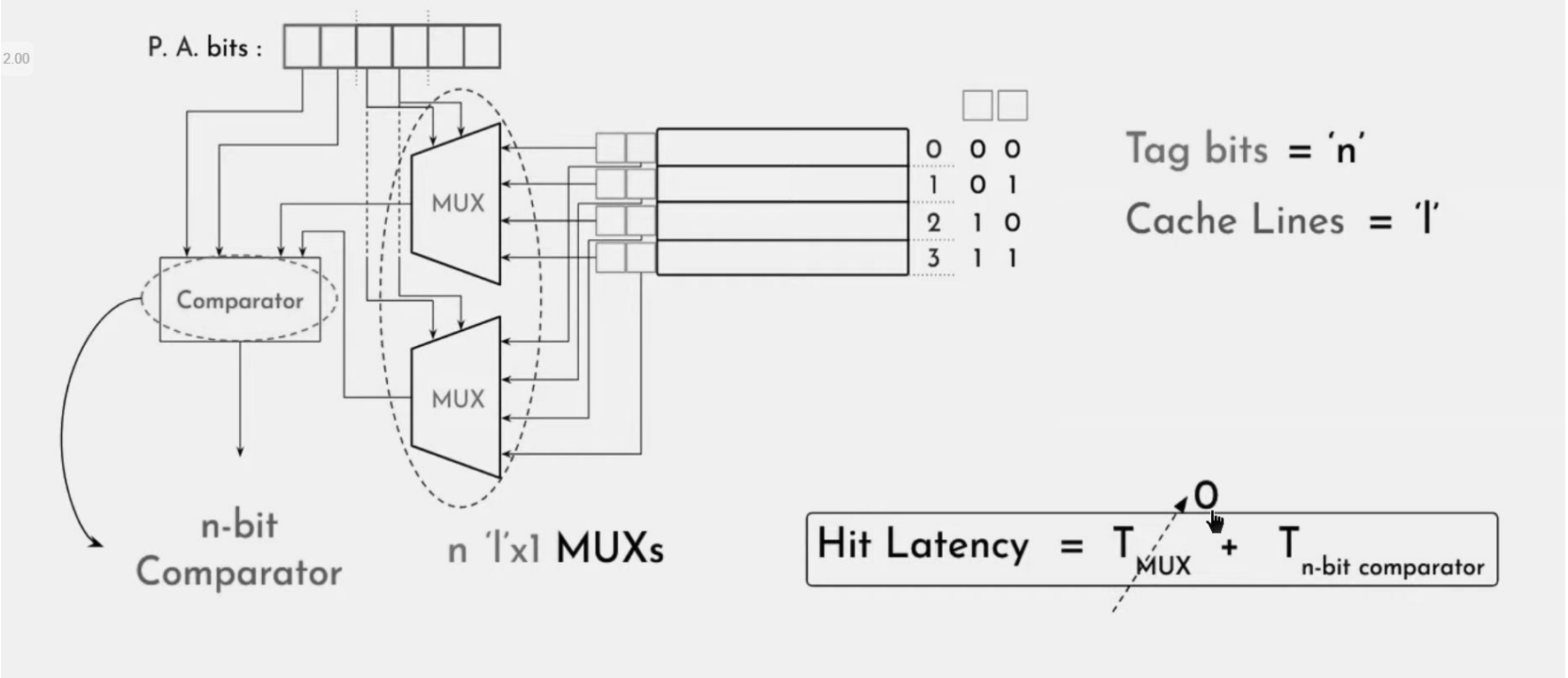
2nd iteration:

Cache miss = 16

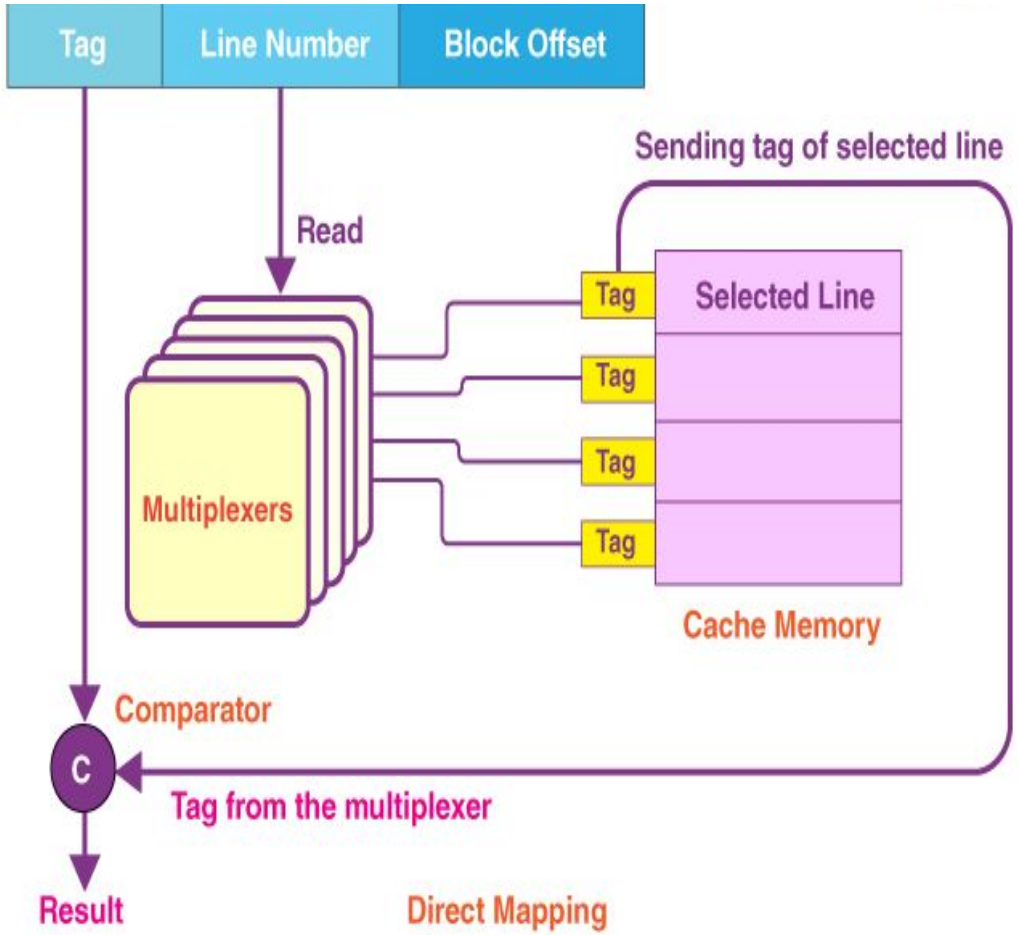
Direct Mapping Hardware implementation



Direct Mapping Hardware implementation



Direct Mapping Hardware implementation



Q: MM Size = 2 GB , Cache Size = 1 MB , Comparator Delay = 8n nsec
Hit latency = ?

MM Size = $2^1 \times 2^{30} = 2^{31}$ B

Cache Size = 2^{20} B

No of Tag bits = $\log_2(2^{31}/2^{20}) = \log_2(2^{31-20}) = \log_2 2^{11} = 11$

Hit Latency = 8 x 11 = 88 nsec

Block j	Main Memory 64 Words			
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23
6	24	25	26	27
7	28	29	30	31
8	32	33	34	35
9	36	37	38	39
10	40	41	42	43
11	44	45	46	47
12	48	49	50	51
13	52	53	54	55
14	56	57	58	59
15	60	61	62	63

Direct Mapping Summary

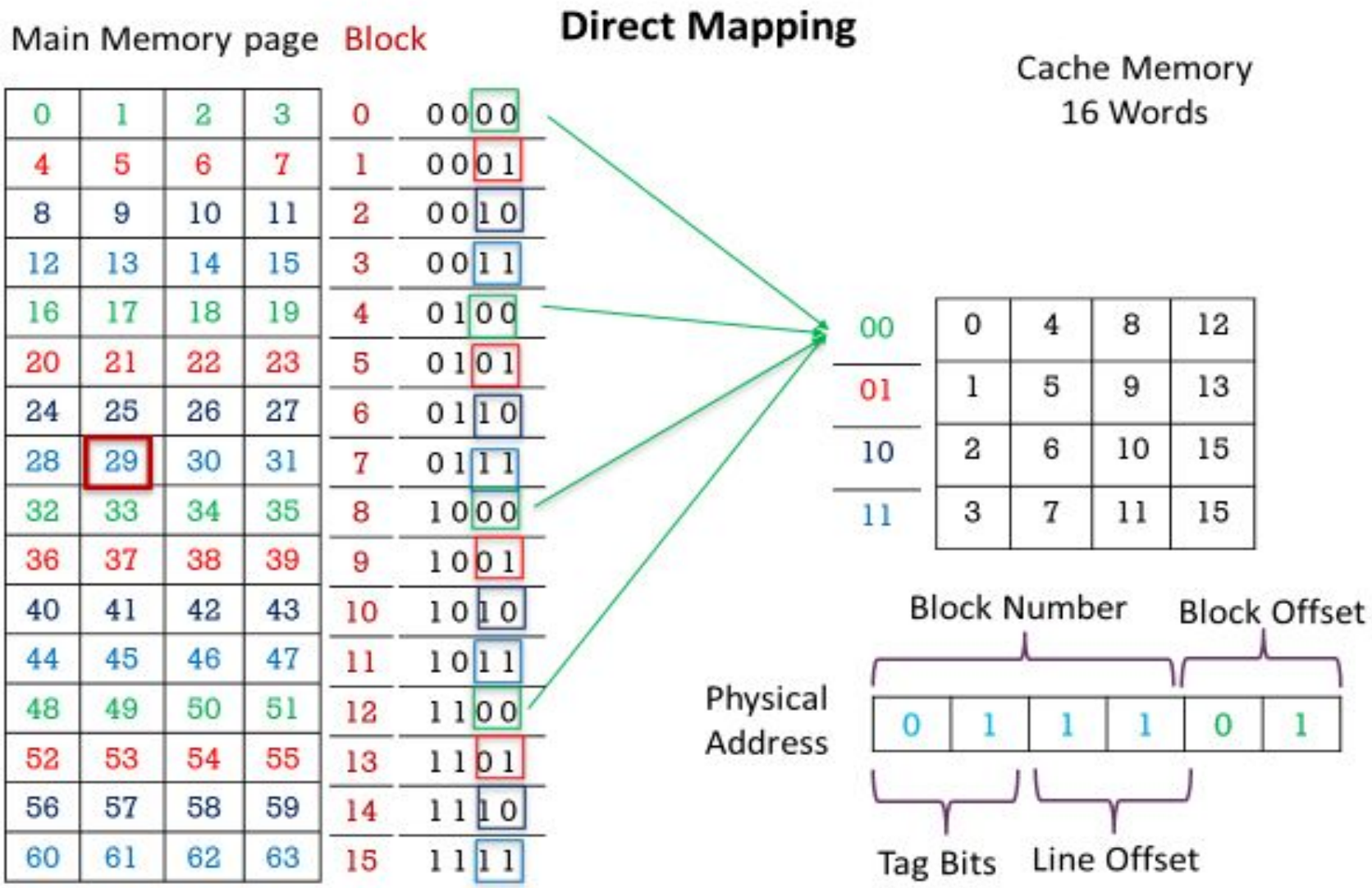
The mapping is expressed as

i = j modulo m
where
i = cache line number
j = main memory block number
m = number of lines in the cache:

				Line i
0	4	8	12	0
1	5	9	13	1
2	6	10	14	2
3	7	11	15	3

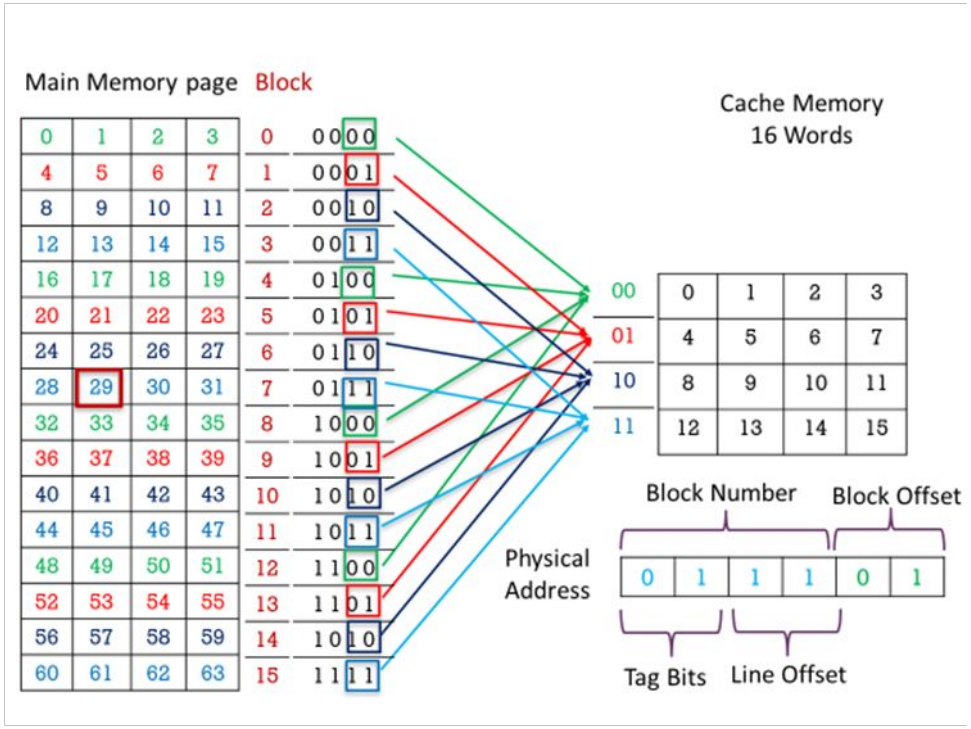
Cache Memory
16 Words

Direct Mapping



Direct Mapping pros & cons

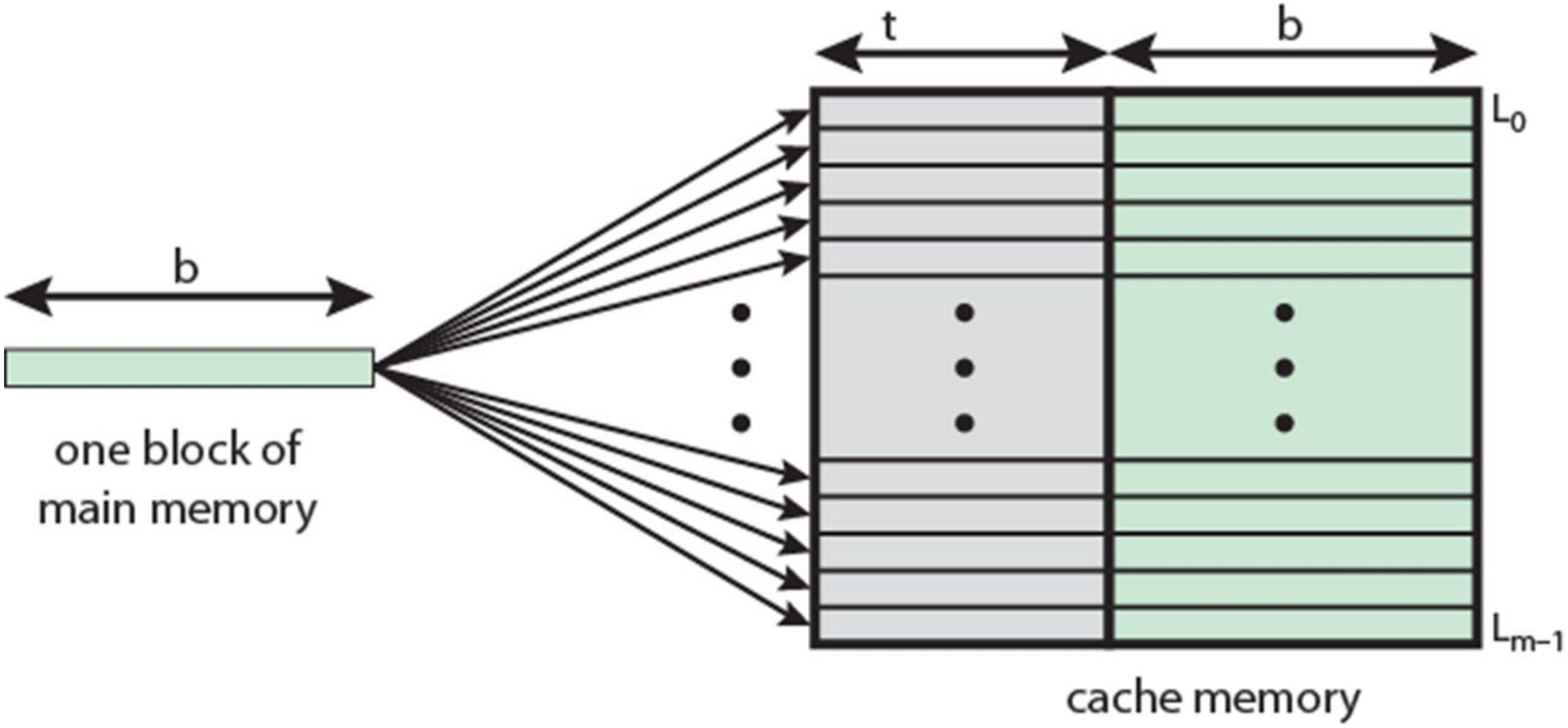
- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high



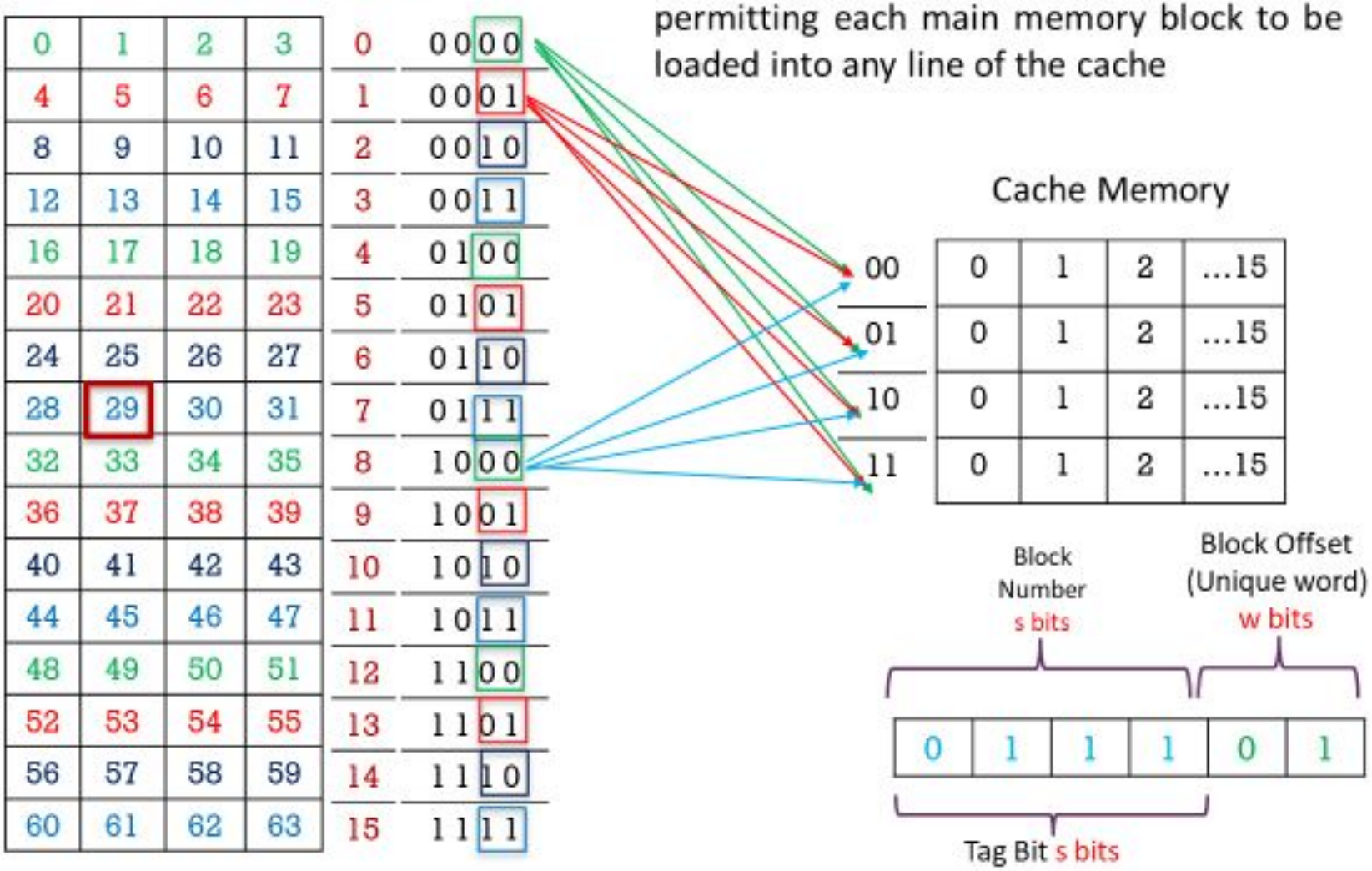
Associative Mapping

- A main memory block can load into **any line of cache**
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined for a match
- **Cache searching gets expensive**

Associative Mapping from Cache to Main Memory

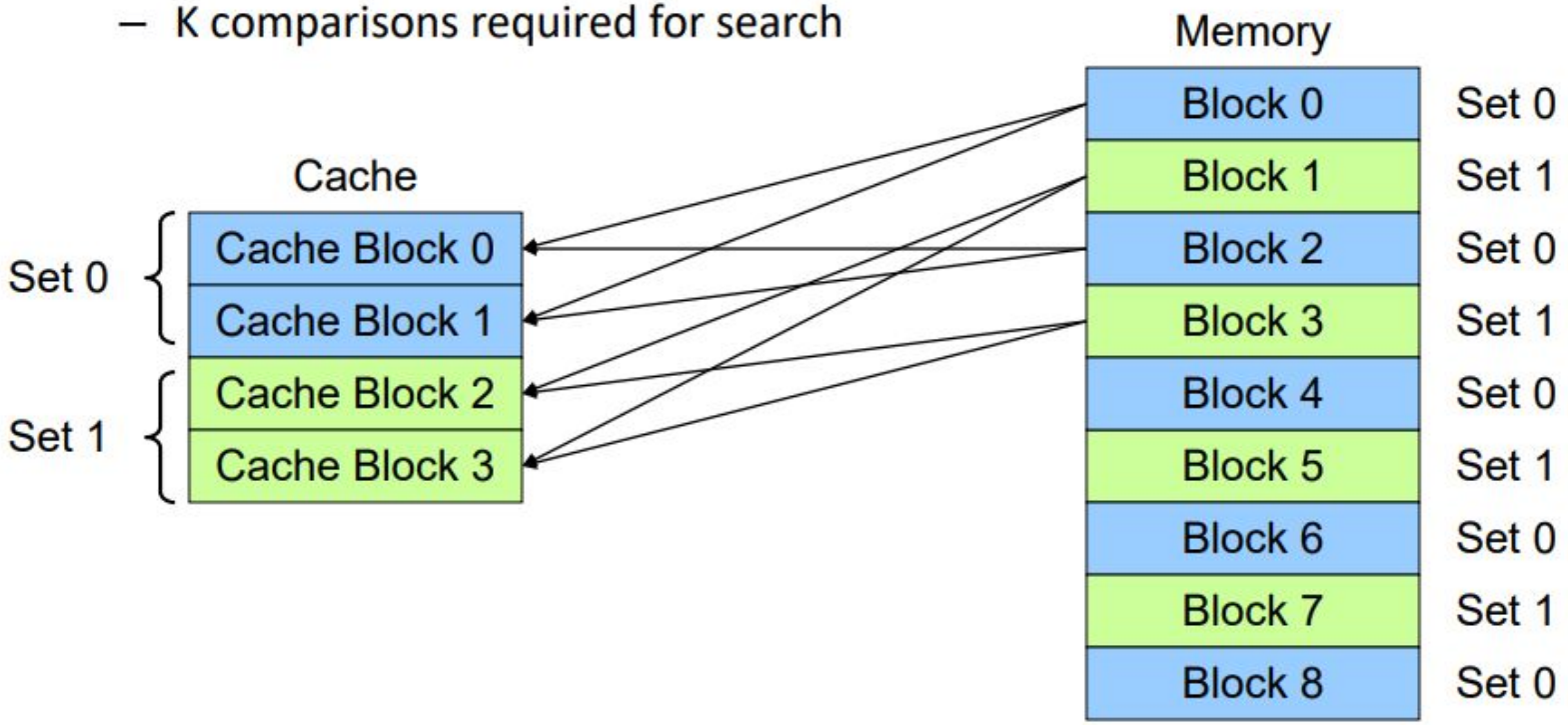


Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache



K-way Set-Associative Cache

- Given, S sets, block i of MM maps to set i mod s
- Within the set, block can be put anywhere
- Let k = number of cache blocks per set = n/s
  - K comparisons required for search



Cache and Main Memory

Split cache: separate instruction and data caches (L1)
Unified cache: holds instructions and data (L1, L2, L3)

Harvard architecture: separate instruction and data memories
von Neumann architecture: one memory for instructions and data

The writing problem:

Write-through slows down the cache to allow main to catch up

Write-back or copy-back is less problematic, but still hurts performance due to two main memory accesses in some cases.

Solution: Provide write buffers for the cache so that it does not have to wait for main memory to catch up.

Improving Cache Performance

For a given *cache size*, the following design issues and tradeoffs exist:

Line width (2^W). Too small a value for W causes a lot of main memory accesses; too large a value increases the miss penalty and may tie up cache space with low-utility items that are replaced before being used.

Set size or associativity (2^S). Direct mapping ($S = 0$) is simple and fast; greater associativity leads to more complexity, and thus slower access, but tends to reduce conflict misses. More on this later.

Line replacement policy. Usually LRU (least recently used) algorithm or some approximation thereof; not an issue for direct-mapped caches. Somewhat surprisingly, random selection works quite well in practice.

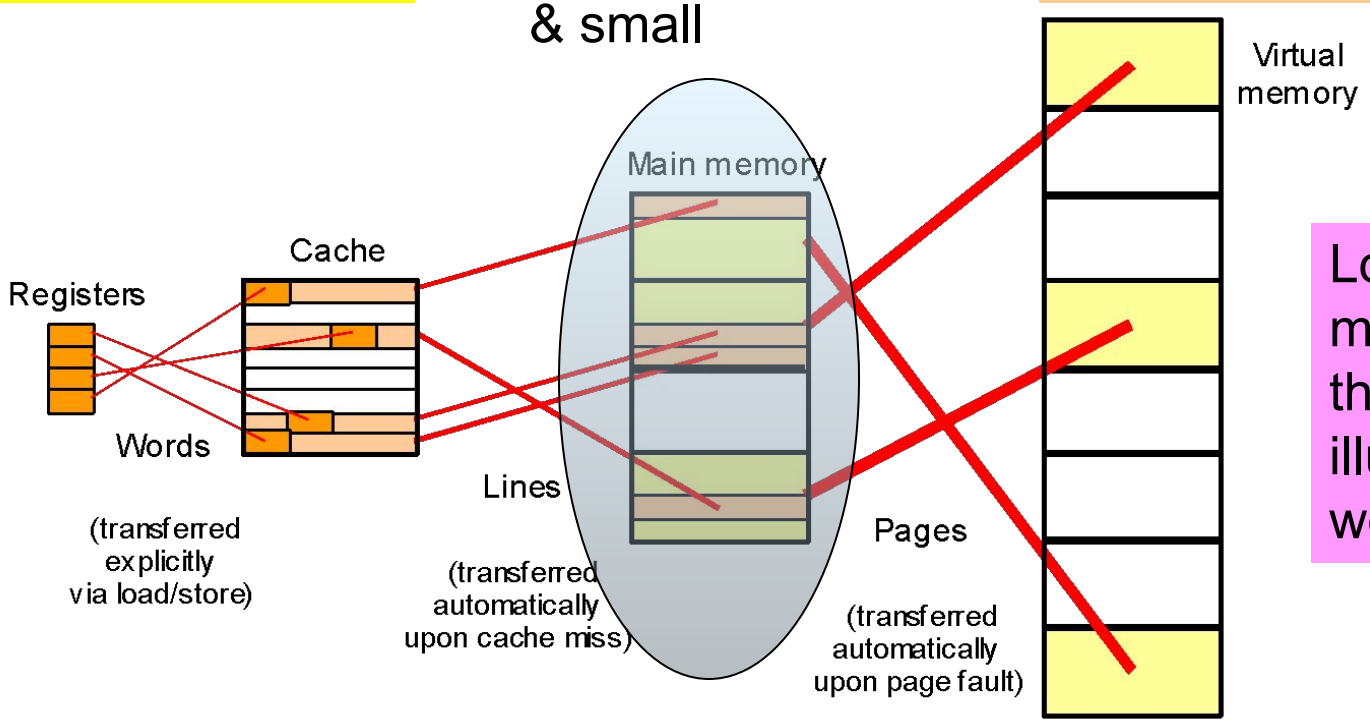
Write policy. Modern caches are very fast, so that *write-through* is seldom a good choice. We usually implement *write-back* or *copy-back*, using write buffers to soften the impact of main memory latency.

Summary of Memory Hierarchy

Cache memory: provides illusion of very high speed

Main memory: reasonable cost, but slow & small

Virtual memory: provides illusion of very large size



Locality makes the illusions work

Fig. 20.2 Data movement in a memory hierarchy.