# Computer Organization and Architecture

## Module 4- Part 2

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

Department of Computer Science and Engineering

IIT Kharagpur

# Memory Hierarchy Design

## Introduction

- Programmers want unlimited amount of memory with very low latency.
- Fast memory technology is more expensive per bit than slower memory.
  - SRAM is more expensive than DRAM, DRAM is more expensive than disk.
- Possible solution?
  - Organize the memory system in several levels, called *memory hierarchy*.
  - Exploit temporal and spatial locality on computer programs.
  - Try to keep the commonly accessed segments of program / data in the faster memories.
  - Results in faster access times on the average.

## Quick Review of Memory Technology

- **Static RAM:**
  - Very fast but expensive memory technology (requires 6 transistors / bit).
  - Packing density is limited.
- **Dynamic RAM:**
  - Significantly slower than SRAM, but much less expensive (1 transistor / bit).
  - Requires periodic refreshing.
- **Flash memory:**
  - Non-volatile memory technology that uses floating-gate MOS transistors.
  - Slower than DRAM, but higher packing density, and lower cost per bit.

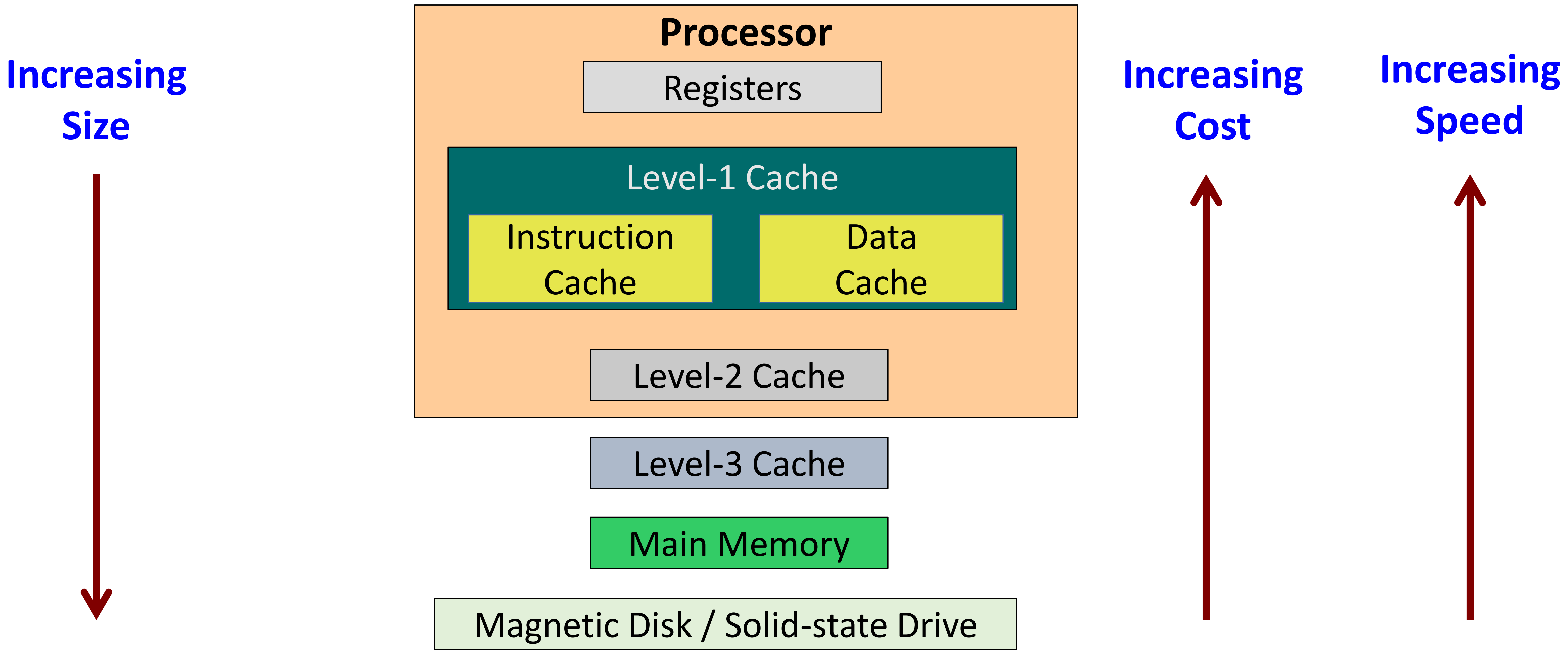
- **Magnetic disk:**
  - Provides large amount of storage, with very low cost per bit.
  - Much slower than DRAM, and also flash memory.
  - Requires mechanical moving parts, and uses magnetic recording technology.

## Memory Hierarchy

- The memory system is organized in several levels, using progressively faster technologies as we move towards the processor.
  - The entire addressable memory space is available in the largest (but slowest) memory (typically, magnetic disk or flash storage).
  - We incrementally add smaller (but faster) memories, each containing a *subset* of the data stored in the memory below it.
  - We proceed in steps towards the processor.

- Typical hierarchy (starting with closest to the processor):
  1. Processor registers
  2. Level-1 cache (typically divided into separate instruction and data cache)
  3. Level-2 cache
  4. Level-3 cache
  5. Main memory
  6. Secondary memory (magnetic disk / solid-state drive)

- As we move away from the processor:
  a) Size increases
  b) Cost decreases
  c) Speed decreases

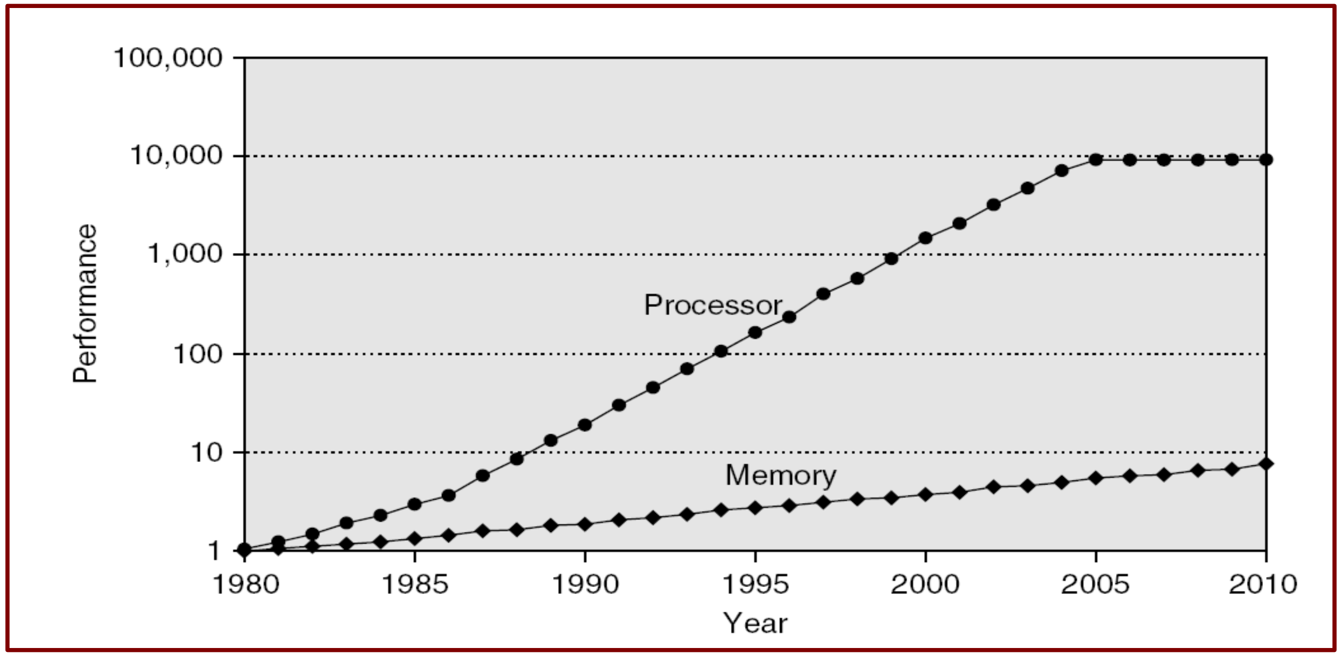

Increasing Size
Processor
Registers
Level-1 Cache
Instruction Cache
Data Cache
Level-2 Cache
Level-3 Cache
Main Memory
Magnetic Disk / Solid-state Drive
Increasing Cost
Increasing Speed

## A Comparison

| Level | Typical Access Time | Typical Capacity | Other Features |
|---|---|---|---|
| Register | 300-500 ps | 500-1000 B | On-chip |
| Level-1 cache | 1-2 ns | 16-64 KB | On-chip |
| Level-2 cache | 5-20 ns | 256 KB – 2 MB | On-chip |
| Level-3 cache | 20-50 ns | 1-32 MB | On or off chip |
| Main memory | 50-100 ns | 1-16 GB | |
| Magnetic disk | 5-50 ms | 100 GB – 16 TB | |

# Processor-Memory Performance Gap

- Processor is much faster than main memory.
  - Processor has to spend much of the time waiting while instructions and data are being fetched from main memory.
  - Memory speed cannot be increased beyond a certain point.



# Impact of Processor / Memory Performance Gap

| Year | CPU Clock | Clock Cycle | Memory Access | Minimum CPU Stall Cycles |
|---|---|---|---|---|
| 1986 | 8 MHz | 125 ns | 190 ns | 190 / 125 – 1 = 0.5 |
| 1989 | 33 MHz | 30 ns | 165 ns | 165 / 30 – 1 = 4.5 |
| 1992 | 60 MHz | 16.6 ns | 120 ns | 120 / 16.6 – 1 = 6.2 |
| 1996 | 200 MHz | 5 ns | 110 ns | 110 / 5 – 1 = 21.0 |
| 1998 | 300 MHz | 3.33 ns | 100 ns | 100 / 3.33 – 1 = 29.0 |
| 2000 | 1 GHz | 1 ns | 90 ns | 90 / 1 – 1 = 89.0 |
| 2002 | 2 GHz | 0.5 ns | 80 ns | 80 / 0.5 – 1 = 159.0 |
| 2004 | 3 GHz | 0.33 ns | 60 ns | 60 / 0.33 – 1 = 179.0 |

Ideal memory access time = 1 CPU cycle

Real memory access time >> 1 CPU cycle

- **Memory Latency Reduction Techniques:**
  - Faster DRAM cells (depends on VLSI technology)
  - Wider memory bus width (fewer memory accesses needed)
  - Multiple memory banks
  - Integration of memory controller with processor
  - New emerging RAM technologies
- **Memory Latency Hiding Techniques**
  - Memory hierarchy (using SRAM-based cache memories)
  - Pre-fetching instructions and/or data from memory before they are actually needed (used to hide long memory access latency)

## Locality of Reference

- Programs tend to reuse data and instructions they have used recently.
  - **Rule of thumb**: 90% of the total execution time of a program is spent in only 10% of the code (also called 90/10 rule).
  - **Reason**: nested loops in a program, few procedures calling each other repeatedly, arrays of data items being accessed sequentially, etc.
- Basic idea to exploit this rule:
  - Based on a program's recent past, we can predict with a reasonable accuracy what instructions and data will be accessed in the near future.

- The 90/10 rule has two dimensions:

  a) **Temporal Locality** (locality in time)
     - If an item is referenced in memory, it will tend to be referenced again soon.

  b) **Spatial locality** (locality in space)
     - If an item is referenced in memory, nearby items will tend to be referenced soon.

## (a) Temporal Locality

- Recently executed instructions are likely to be executed again very soon.
- Example: computing factorial of a number.

```
fact = 1;
for  k = 1 to N
  fact = fact * k;
```

```
      ADDI  $t1,$zero,1
      ADDI  $t2,$zero,N
      ADDI  $t3,$zero,1
Loop: MUL   $t1,$t1,$t3
      ADDI  $t3,$t3,1
      SGT   $t4,$t3,$t2
      BNEZ  $t4,Loop
```

- The four instructions in the loop are executed more frequently than the others.

## (b) Spatial Locality

- Instructions residing close to a recently executing instruction are likely to be executed soon.
- Example: accessing elements of an array.

```
sum = 0;
for  k = 1 to N
   sum = sum + A[k];
```

```
      SUB   $t1,$t1,$t1
      ADDI  $t2,$zero,N
      ADDI  $t3,$zero,1
      ADDI  $t5,$zero,A
Loop: LW    $t8,0($t5)
      ADD   $t1,$t1,$t8
      ADDI  $t3,$t3,1
      SGT   $t4,$t3,$t2
      BNEZ  $t4,Loop
```

- Performance can be improved by copying the array into cache memory.

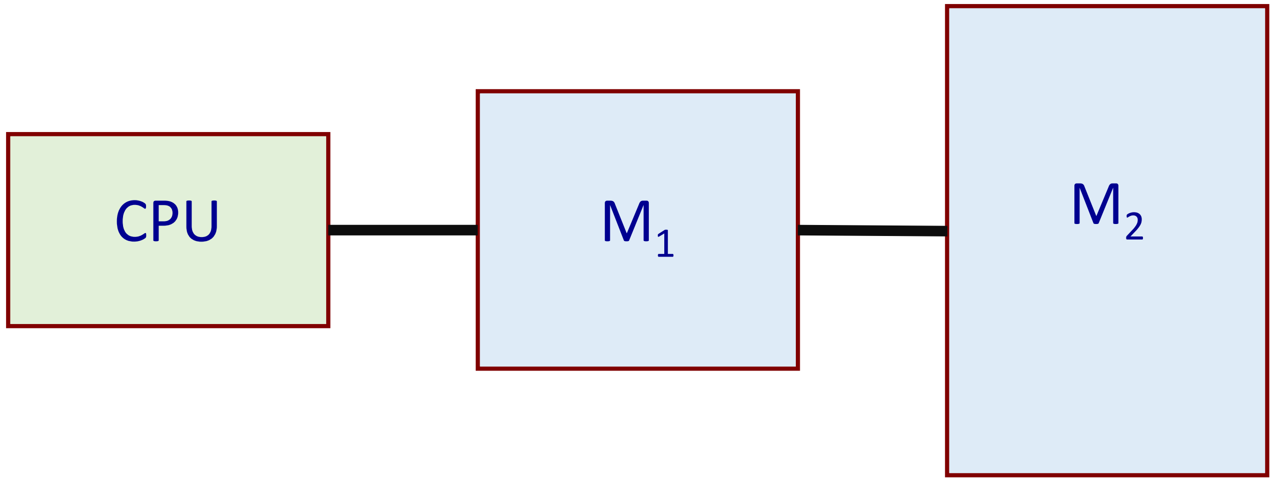
- An example:
    - Accessing a 2-D array row-wise or column-wise.
    - Assume that the array elements are stored row-wise in memory.
    - One row of data can be brought into cache memory at any given time.

```
for (i=0; i<128; i++)
  for (j=0; j<128; j++)
    A[i][j] = A[i][j] + 1;
```

```
for (j=0; j<128; j++)
  for (i=0; i<128; i++)
    A[i][j] = A[i][j] + 1;
```

# Performance of Memory Hierarchy

- We first consider a 2-level hierarchy consisting of two levels of memory, say, $M_1$ and $M_2$.



## a) Cost:

- Let $c_i$ denote the cost per bit of memory $M_i$, and $S_i$ denote the storage capacity in bits of $M_i$.
- The average cost per bit of the memory hierarchy is given by:

$$\text{Cost } c = \frac{c_1S_1 + c_2S_2}{S_1 + S_2}$$

## b) Hit Ratio / Hit Rate:

- The hit ratio H is defined as the probability that a logical address generated by the CPU refers to information stored in $M_1$.
- We can determine H experimentally as follows:
  - A set of representative programs is executed or simulated.
  - The number of references to $M_1$ and $M_2$, denoted by $N_1$ and $N_2$ respectively, are recorded.

$$H = \frac{N_1}{N_1 + N_2}$$

- The quantity (1 – H) is called the *miss ratio*.

## c) Access Time:

- Let $t_{A1}$ and $t_{A2}$ denote the access times of $M_1$ and $M_2$ respectively, relative to the CPU.
- The average time required by the CPU to access a word in memory can be expressed as:

$$t_A = H.t_{A1} + (1 - H).t_{MISS}$$

where $t_{MISS}$ denotes the time required to handle the miss, called *miss penalty*.

- The miss penalty $t_{MISS}$ can be estimated in various ways:
  a) The simplest approach is to set $T_{MISS} = t_{A2}$ , that is, when there is a miss the data is accessed directly from $M_2$.
  b) A request for a word not in $M_1$ typically causes a block containing the requested word to be transferred from $M_2$ to $M_1$. After completion of the block transfer, the word can be accessed in $M_1$.

     If $t_B$ denotes the block transfer time, we can write

     $t_{MISS} = t_B + t_{A1}$ [since $t_B >> t_{A1}$ , $t_{A2} \approx t_B$]

     Thus, $t_A = H.t_{A1} + (1 – H).(t_B + t_{A1})$
  c) If $t_{HIT}$ denotes the time required to check whether there is a hit, we can write

     $t_{MISS} = t_{HIT} + t_B + t_{A1}$

## d) Efficiency:

- Let $r = t_{A2} / t_{A1}$ denote the access time ratio of the two levels of memory.
- We define the access efficiency as $e = t_{A1} / t_A$ , which is the factor by which $t_A$ differs from its minimum possible value.

$$\text{Efficiency } e = \frac{t_{A1}}{H.t_{A1} + (1 - H).t_{A2}} = \frac{1}{H + (1 - H).r}$$

## e) Speedup:

- The speedup gained by using the memory hierarchy is defined as $S = t_{A2} / t_A$ .
- We can write:

$$S = \frac{t_{A2}}{H.t_{A1} + (1 - H).t_{A2}} = \frac{1}{(1 - H) + H / r}$$

- The same result follows from *Amdahl's law*.

## Some Common Terminologies Used

- **Block**: The smallest unit of information transferred between two levels.
- **Hit Rate**: The fraction of memory accesses found in the upper level.
- **Hit Time**: Time to access the upper level
  - Upper level access time + Time to determine hit/miss
- **Miss**: Data item needs to be retrieved from a block in the lower level.
- **Miss Rate**: The fraction of memory accesses not found in the upper level.
- **Miss Penalty**: Overhead whenever a miss occurs.
  - Time to replace a block in the upper level + Time to transfer the missed block

## Example 1

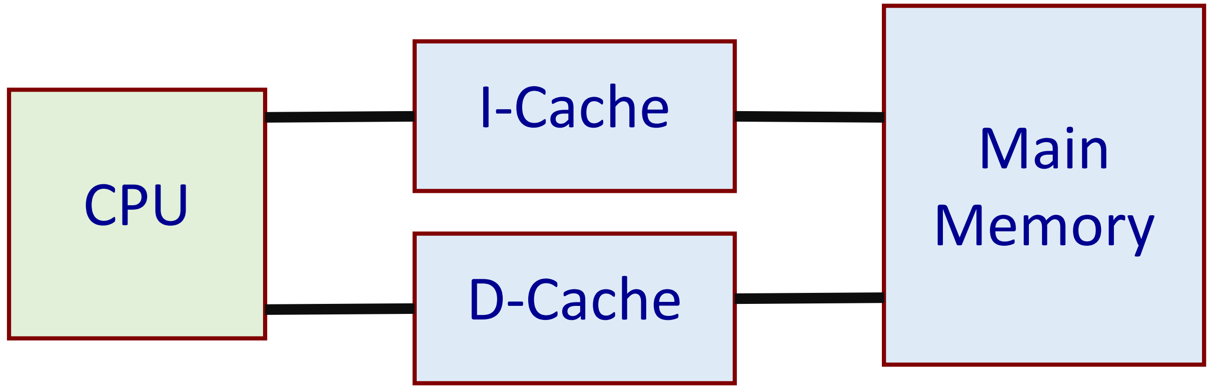
- Consider a 2-level memory hierarchy consisting of a cache memory $M_1$ and the main memory $M_2$. Suppose that the cache is 6 times faster than the main memory, and the cache can be used 90% of the time. How much speedup do we gain by using the cache?

Here, $r = 6$ and $H = 0.90$

Thus, $S = 1 / [H / r + (1 - H)] = 1 / (0.90 / 6 + 0.10) = 1 / 0.25 = 4$

## Example 2

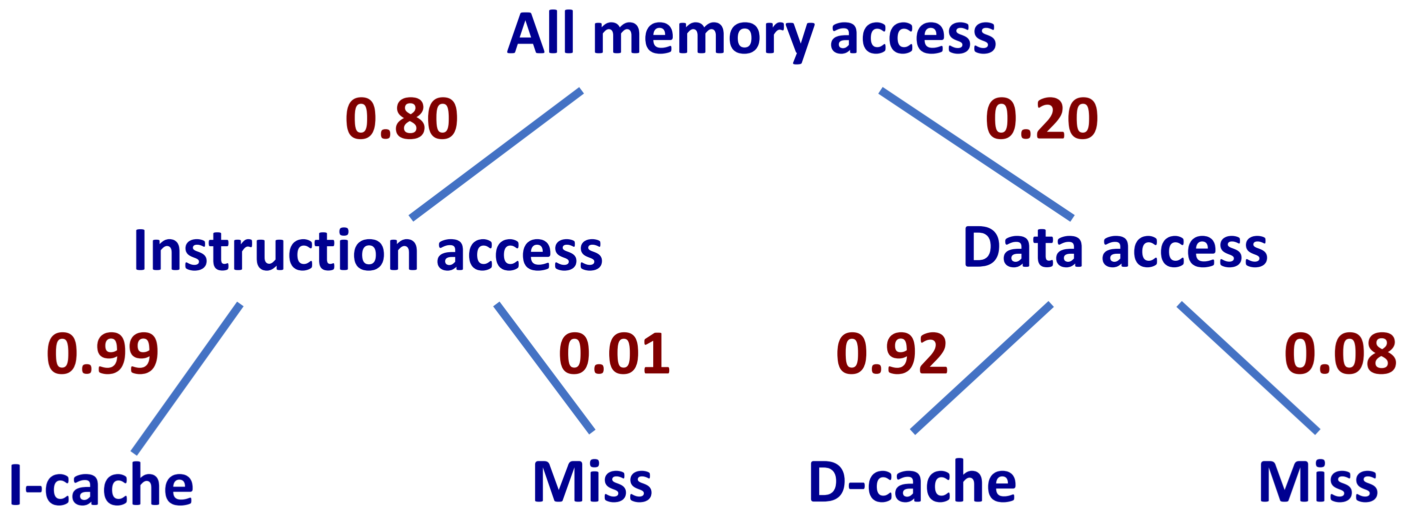
- Consider a 2-level memory hierarchy with separate instruction and data caches in level 1, and main memory in level 2.



The following parameters are given:

- The clock cycle time is 2 ns.
- The main mem. access time is 15 clock cycles (for both read and write).
- 1 % of instructions are not found in I-cache.
- 8 % of data references are not found in D-cache.
- 20 % of the total memory accesses are for data.
- Cache access time (including hit detection) is 1 clock cycle.

Determine the overall average access time.



$t_{MISS}$ = 1 + 15 = 16 cycles

Average number of cycles per access:

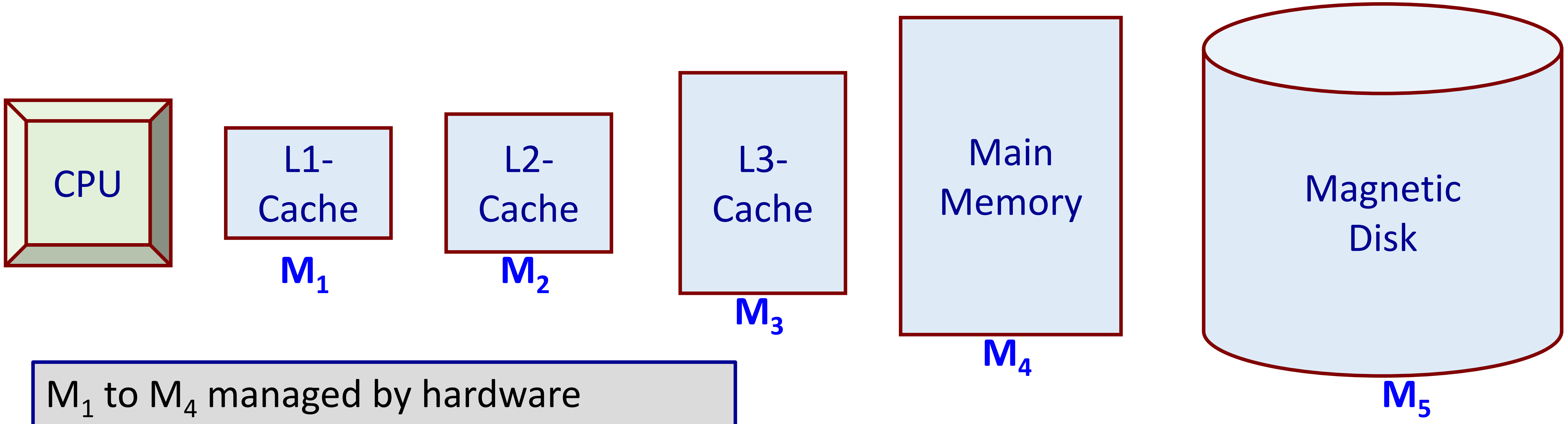
0.80 x (0.99 x 1 + 0.01 x 16) + 0.20 x (0.92 x 1 + 0.08 x 16)

= 0.92 + 0.44 = 1.36

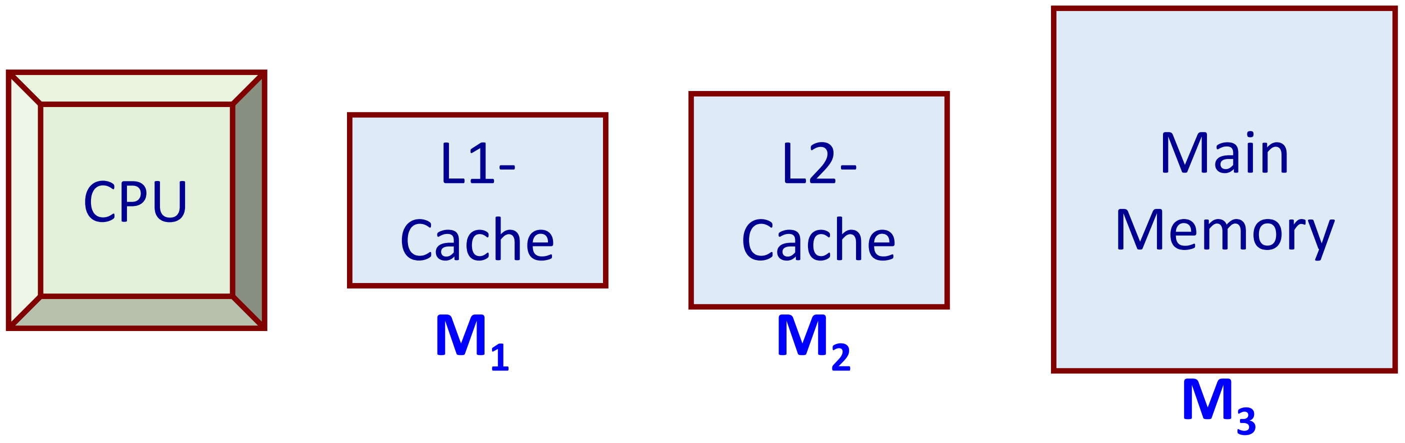
Thus, average access time $t_A$ = 1.36 x 2 ns = 2.72 ns

## Performance Calculation for Multi-Level Hierarchy

- Most of the practical memory systems use more than 2 levels of hierarchy.



$M_1$ to $M_4$ managed by hardware
$M_4$ to $M_5$ managed by operating system



$t_{L1}$ : access time of $M_1$

$t_{L2}$ : access time of $M_2$

$H_{L1}$ : hit ratio of $M_1$

$H_{L2}$ : hit ratio of $M_2$ with respect to the residual accesses that try to access $M_2$

- Consider a 3-level hierarchy consisting of L1-cache, L2-cache and main memory.
- Whenever there is a miss in L1, we go to L2.
- Average access time can be calculated as:

$$t_A = H_{L1}.t_{L1} + (1 - H_{L1}) . [H_{L2}.t_{L2} + (1 - H_{L2}).t_{MISS2}]$$

- Here, $t_{MISS2}$ is the miss penalty when the requested data is found neither in $M_1$ nor in $M_2$.

## Implications of a Memory Hierarchy to the CPU

- Processors designed without memory hierarchy are simpler because all memory accesses take the same amount of time.
  - Misses in a memory hierarchy implies *variable memory access times* as seen by the CPU.
- Some mechanism is required to determine whether or not the requested information is present in the top level of the memory hierarchy.
  - Check happens on every memory access and affects hit time.
  - Implemented in hardware to provide acceptable performance.

- Some mechanism is required to transfer blocks between consecutive levels.
  - If the block transfer requires 10's of clock cycles (like in cache / main memory hierarchy), it is controlled by *hardware*.
  - If the block transfer requires 1000's of clock cycles (like in main memory / secondary memory hierarchy), it can be controlled by *software*.
- Four main questions:
  1. *Block Placement*: Where to place a block in the upper level?
  2. *Block Identification*: How is a block found if present in the upper level?
  3. *Block Replacement*: Which block is to be replaced on a miss?
  4. *Write Strategy*: What happens on a write?

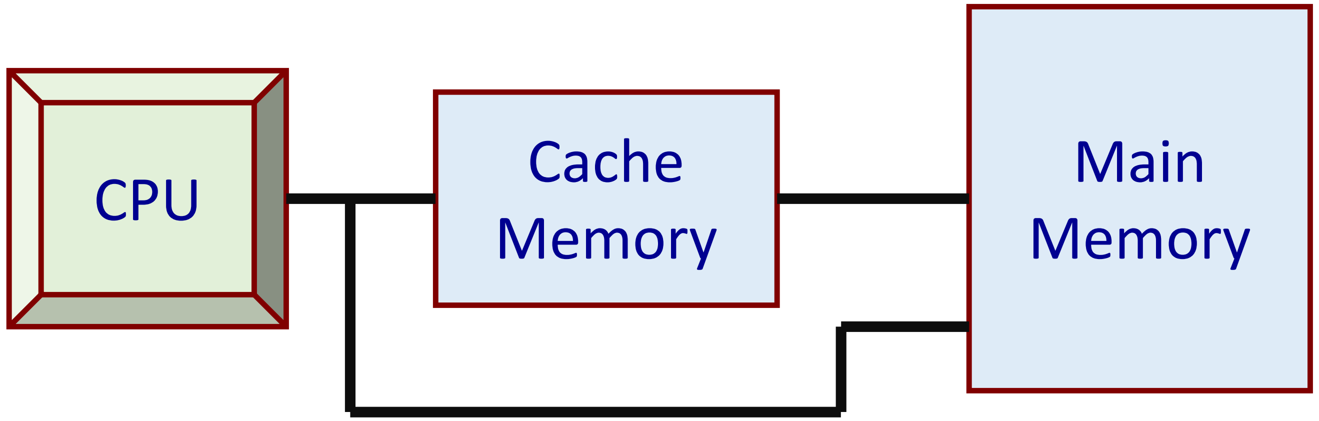
## Common Memory Hierarchies

- In a typical computer system, the memory system is managed as two different hierarchies.
  - The *Cache / Main Memory hierarchy*, which consists of 2 to 4 levels and is managed by hardware.
    - Main objective: provide fast average memory access.
  - The *Main Memory / Secondary Memory hierarchy*, which consists of 2 levels and is managed by software (operating system).
    - Main objective: provide large memory space for users (virtual memory).

# Cache Memory

## Introduction

- Let us consider a single-level cache, and that part of the memory hierarchy consisting of cache memory and main memory.



- Cache memory is logically divided into *blocks* or *lines*, where every block (line) typically contains 8 to 256 bytes.

- When the CPU wants to access a word in memory, a special hardware first checks whether it is present in cache memory.
  - If so (called *cache hit*), the word is directly accessed from the cache memory.
  - If not, the block containing the requested word is brought from main memory to cache.
  - For writes, sometimes the CPU can also directly write to main memory.

- Objective is to keep the commonly used blocks in the cache memory.
  - Will result in significantly improved performance due to the property of *locality of reference*.

## Q1. Where can a block be placed in the cache?

- This is determined by some *mapping algorithms*.
  - Specifies which main memory blocks can reside in which cache memory blocks.
  - At any given time, only a small subset of the main memory blocks can be held in main memory.

- Three common block mapping techniques are used:
  a) **Direct Mapping**
  b) **Associative Mapping**
  c) **(N-way) Set Associative Mapping**

# An Example Scenario: A 2-level memory hierarchy

- Consider a 2-level cache memory / main memory hierarchy.
  - The cache memory consists of 256 blocks (lines) of 32 words each.
    - Total cache size is 8192 (8K) words.
  - Main memory is addressable by a 24-bit address.
    - Main memory is word addressable.
    - Total size of the main memory is $2^{24}$ = 16 M words.
    - Number of 32-word blocks in main memory = 16 M / 32 = 512K

## (a) Direct Mapping

- Each main memory block can be placed in only one block in the cache.
- The mapping function is:

  Cache Block = (Main Memory Block) % (Number of cache blocks)

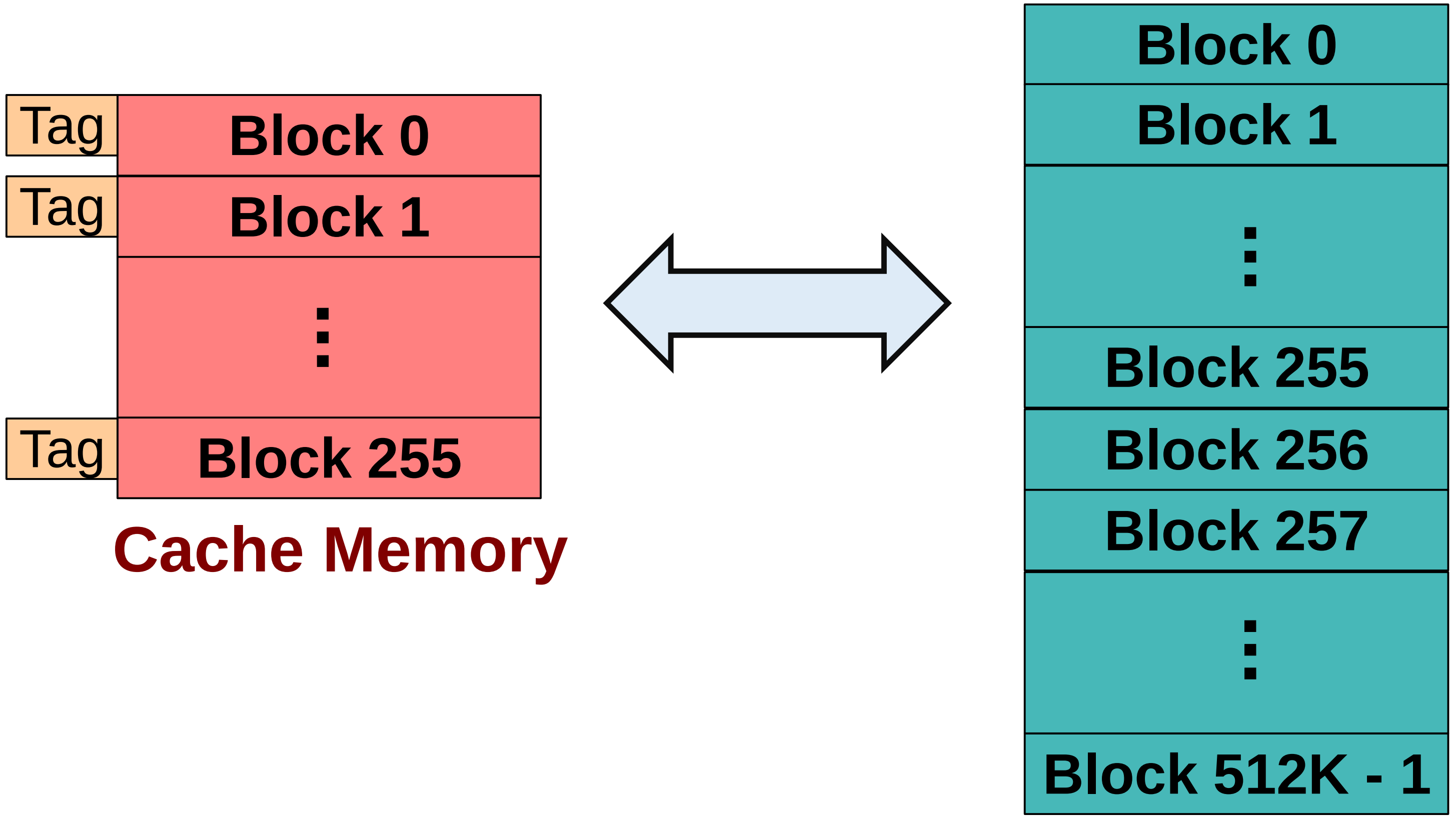
- For the example,

  Cache Block = (Main Memory Block) % 256

  - Some example mappings:

    0 → 0, 1 → 1, 255 → 255, 256 → 0, 257 → 1, 512 → 0, 512 → 1, etc.

## Direct Mapping



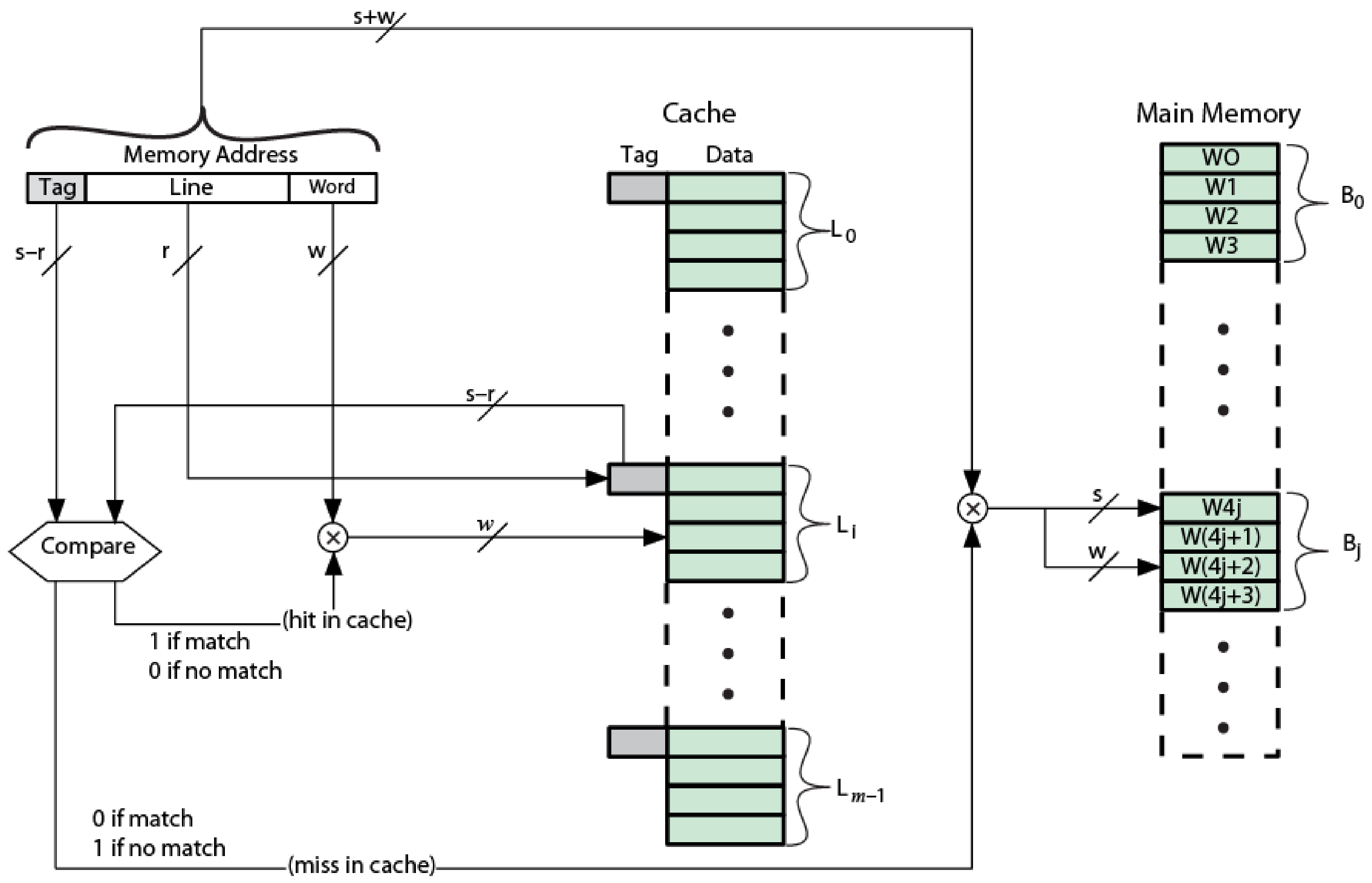
Cache Memory

Main Memory

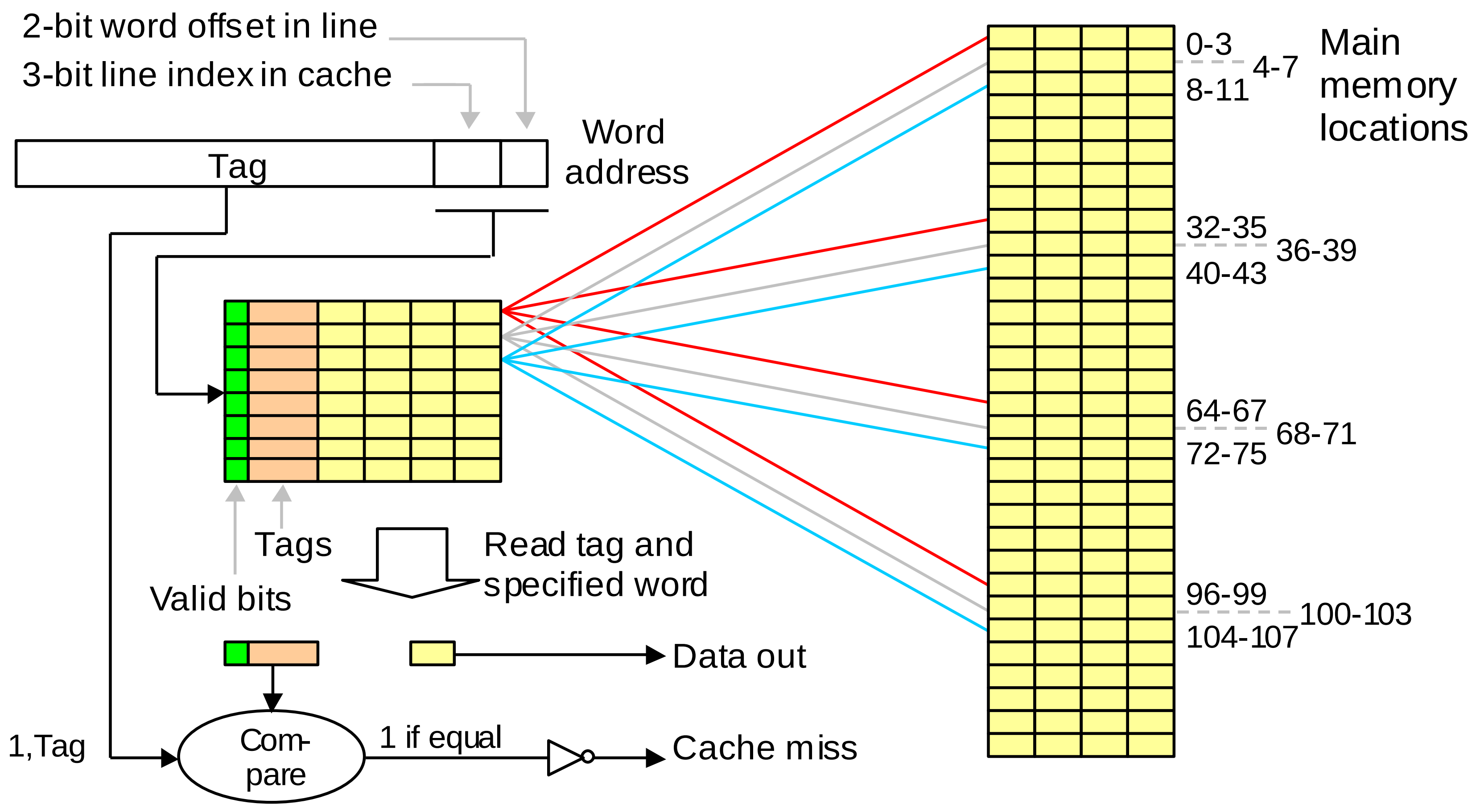
| TAG | BLOCK | WORD |
|---|---|---|
| 11 | 8 | 5 |

Memory Address

# Direct Mapped Cache



# Direct-Mapped Cache



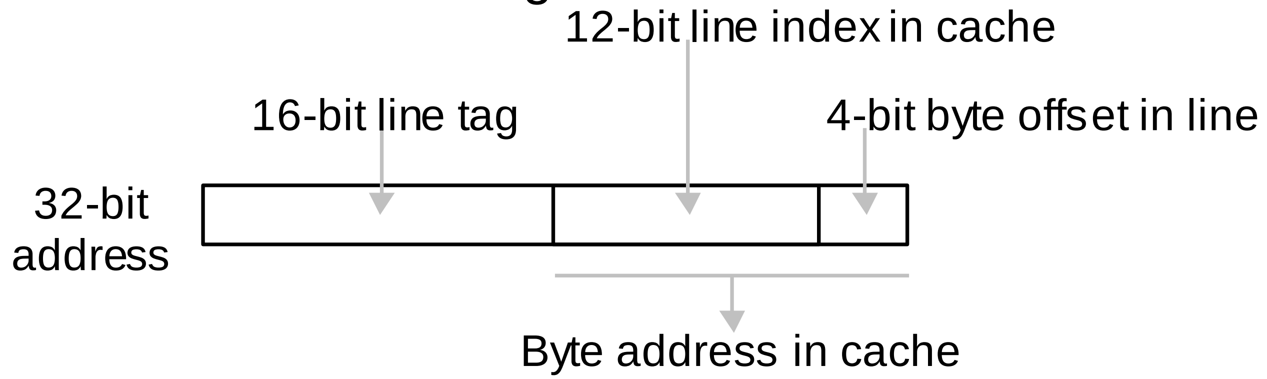
Direct-mapped cache holding 32 words within eight 4-word lines. Each line is associated with a tag and a valid bit.

# Example: Accessing a Direct-Mapped Cache

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.

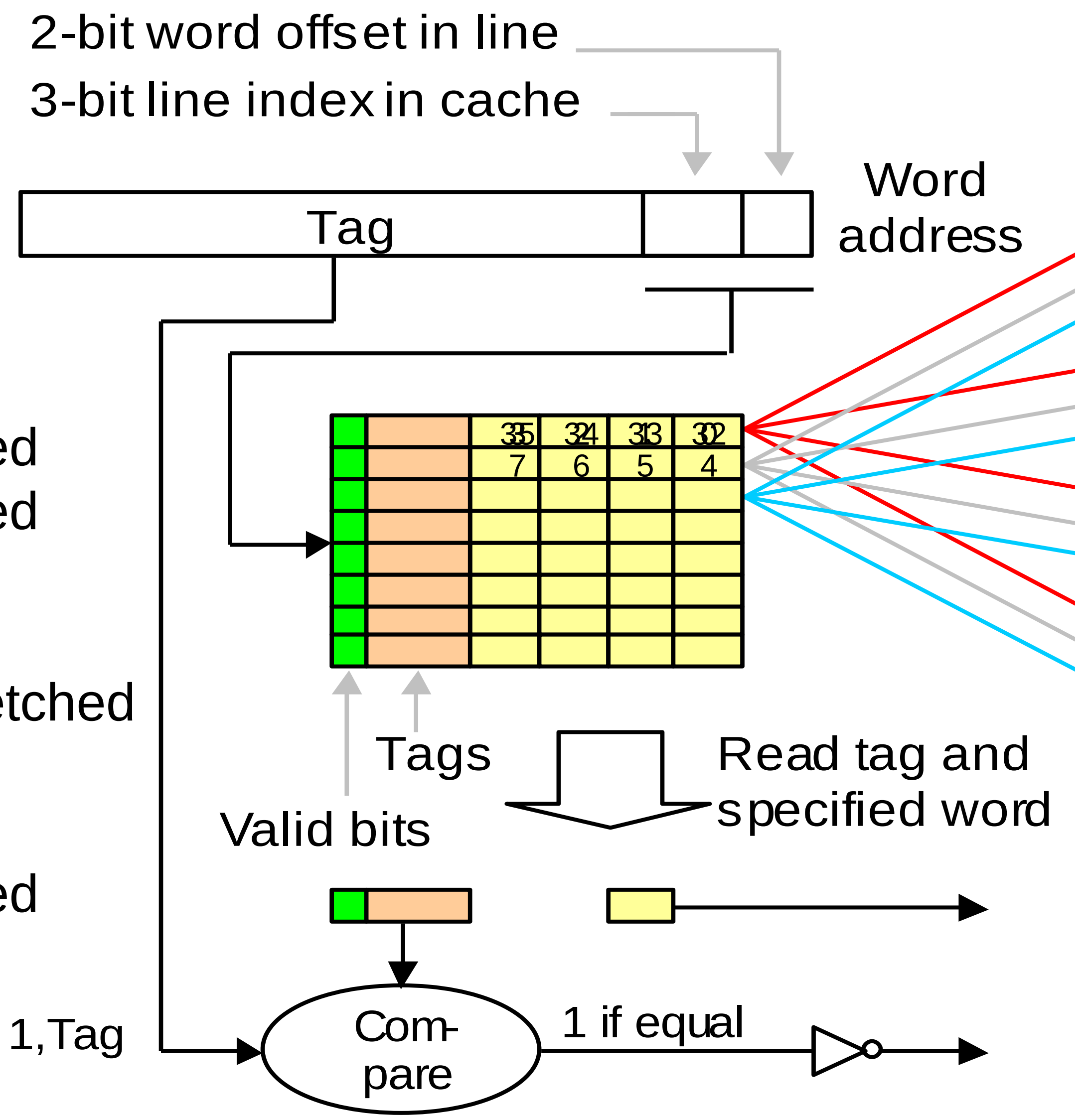


Components of the 32-bit address in an example direct-mapped cache with byte addressing.

## Direct-Mapped Cache Behavior

Address trace:
1, 7, 6, 5, 32, 33, 1, 2, . . .

  1: miss, line 3, 2, 1, 0 fetched
  7: miss, line 7, 6, 5, 4 fetched
  6: hit
  5: hit
32: miss, line 35, 34, 33, 32 fetched
      (replaces 3, 2, 1, 0)
33: hit
  1: miss, line 3, 2, 1, 0 fetched
      (replaces 35, 34, 33, 32)
  2: hit
 ... and so on


2-bit word offset in line
3-bit line index in cache
Tag
Word address
35 34 33 32
3 2 1 0
7 6 5 4
Tags
Valid bits
Read tag and specified word
1,Tag
Com-pare
1 if equal

- Block replacement algorithm is trivial, as there is no choice.
- More than one MM block is mapped onto the same cache block.
  - May lead to contention even if the cache is not full.
  - New block will replace the old block.
  - May lead to poor performance if both the blocks are frequently used.
- The MM address is divided into three fields: *TAG*, *BLOCK* and *WORD*.
  - When a new block is loaded into the cache, the 8-bit *BLOCK* field determines the cache block where it is to be stored.
  - The high-order 11 bits are stored in a *TAG* register associated with the cache block.
  - When accessing a memory word, the corresponding TAG field is compared.
    - Match implies HIT.

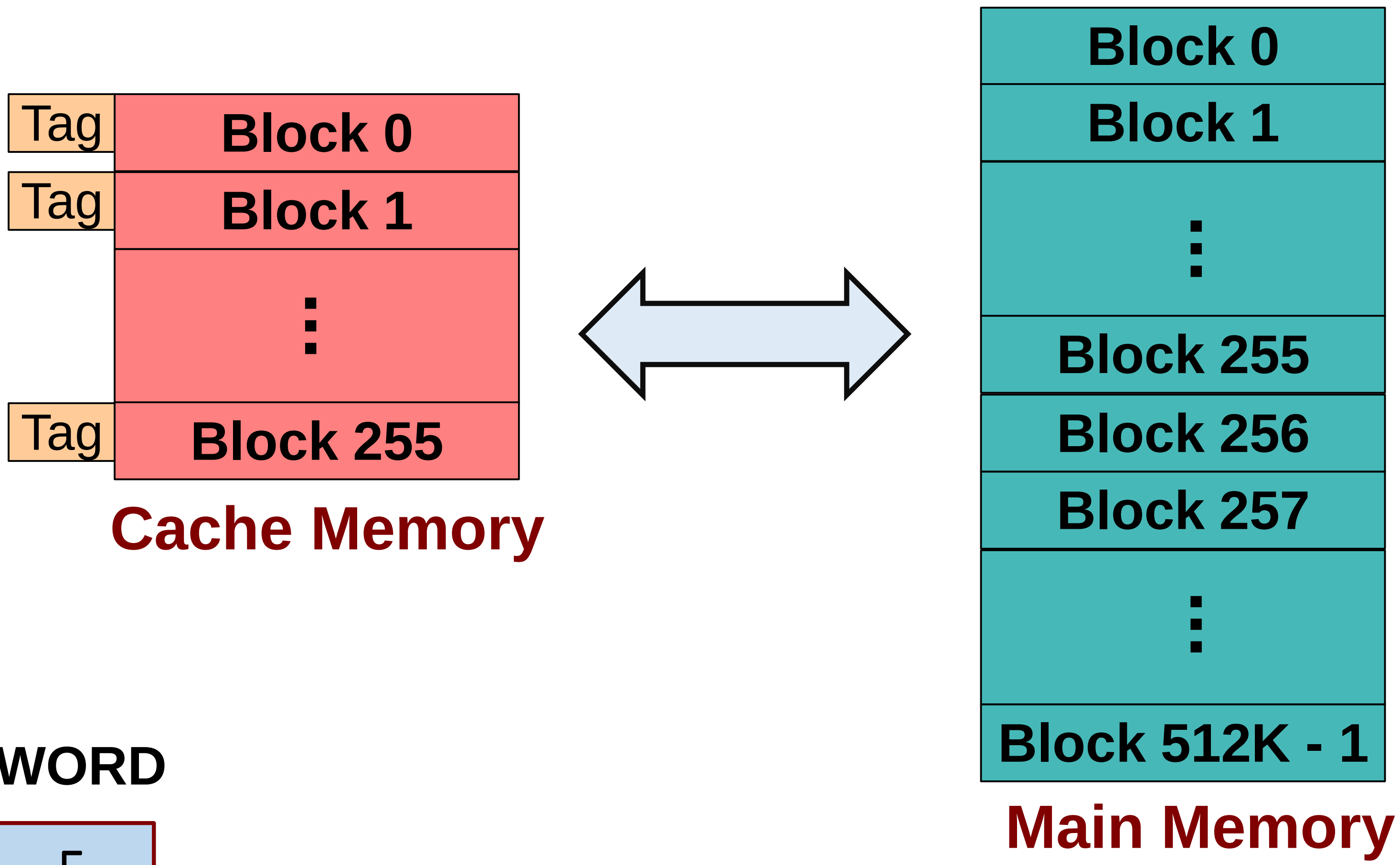
## (b) Associative Mapping

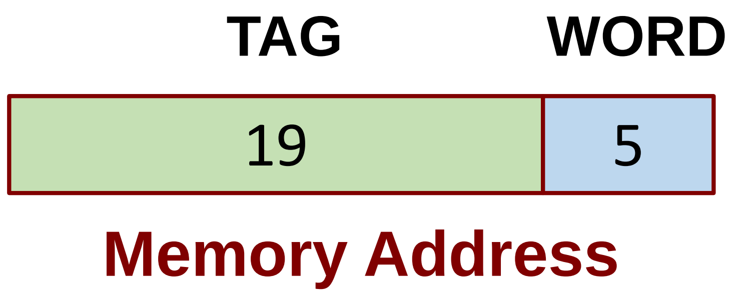
- Here, a MM block can potentially reside in *any cache block position*.
- The memory address is divided into two fields: *TAG* and *WORD*.
  - When a block is loaded into the cache from MM, the higher order 19 bits of the address are stored into the TAG register corresponding to the cache block.
  - When accessing memory, the 19-bit TAG field of the address is compared with all the TAG registers corresponding to all the cache blocks.
- Requires associative memory for storing the TAG values.
  - High cost / lack of scalability.
- Because of complete freedom in block positioning, a wide range of replacement algorithms is possible.

## What is an Associative Memory?

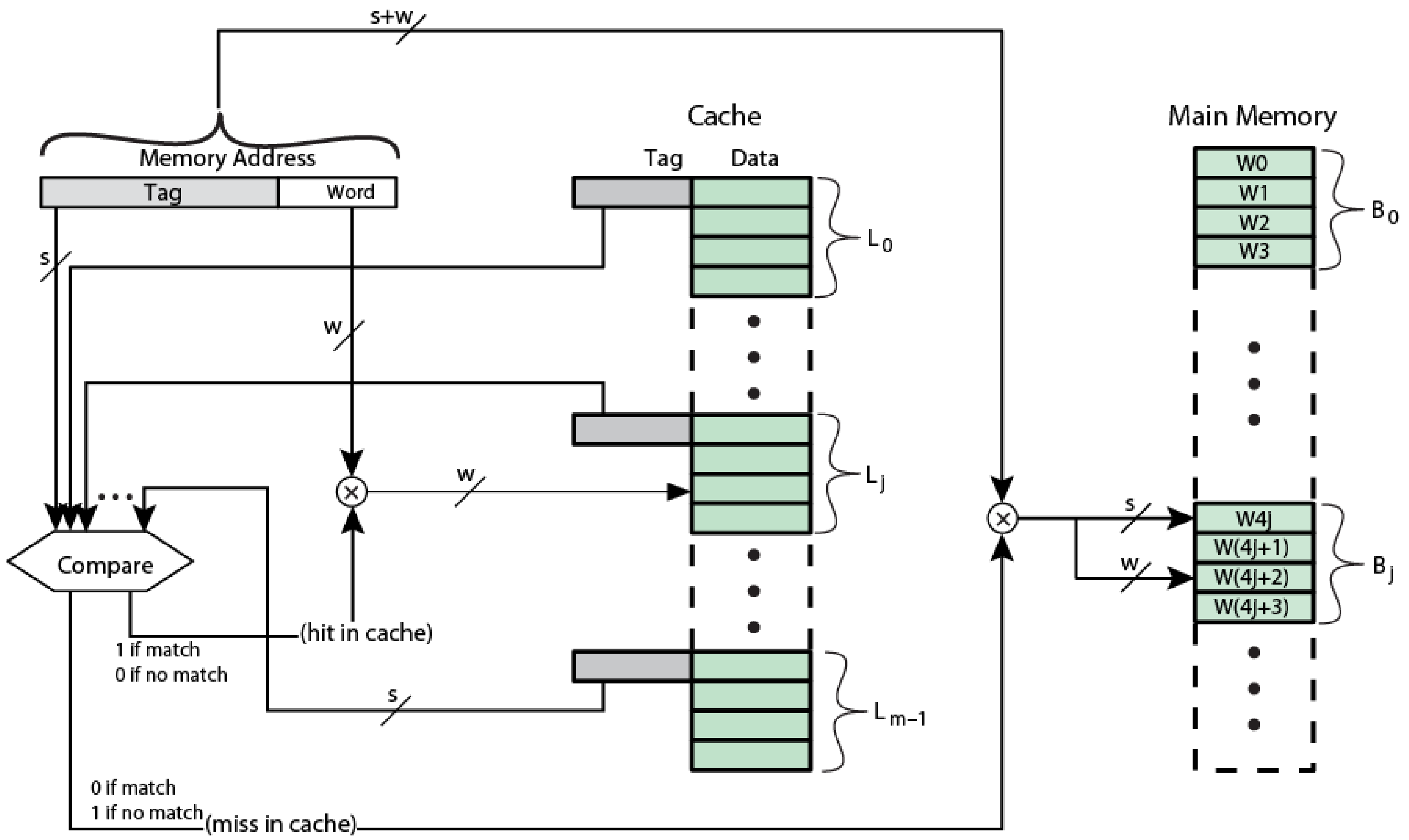
- It is a memory where data is accessed by contents rather than address.
  - There is circuitry to compare the applied data value with all the stored values in memory in parallel.
  - Wherever there is a match, the memory system will be returning the information.
  - Very expensive to build – only small capacity units are feasible.

## Fully Associative Mapping

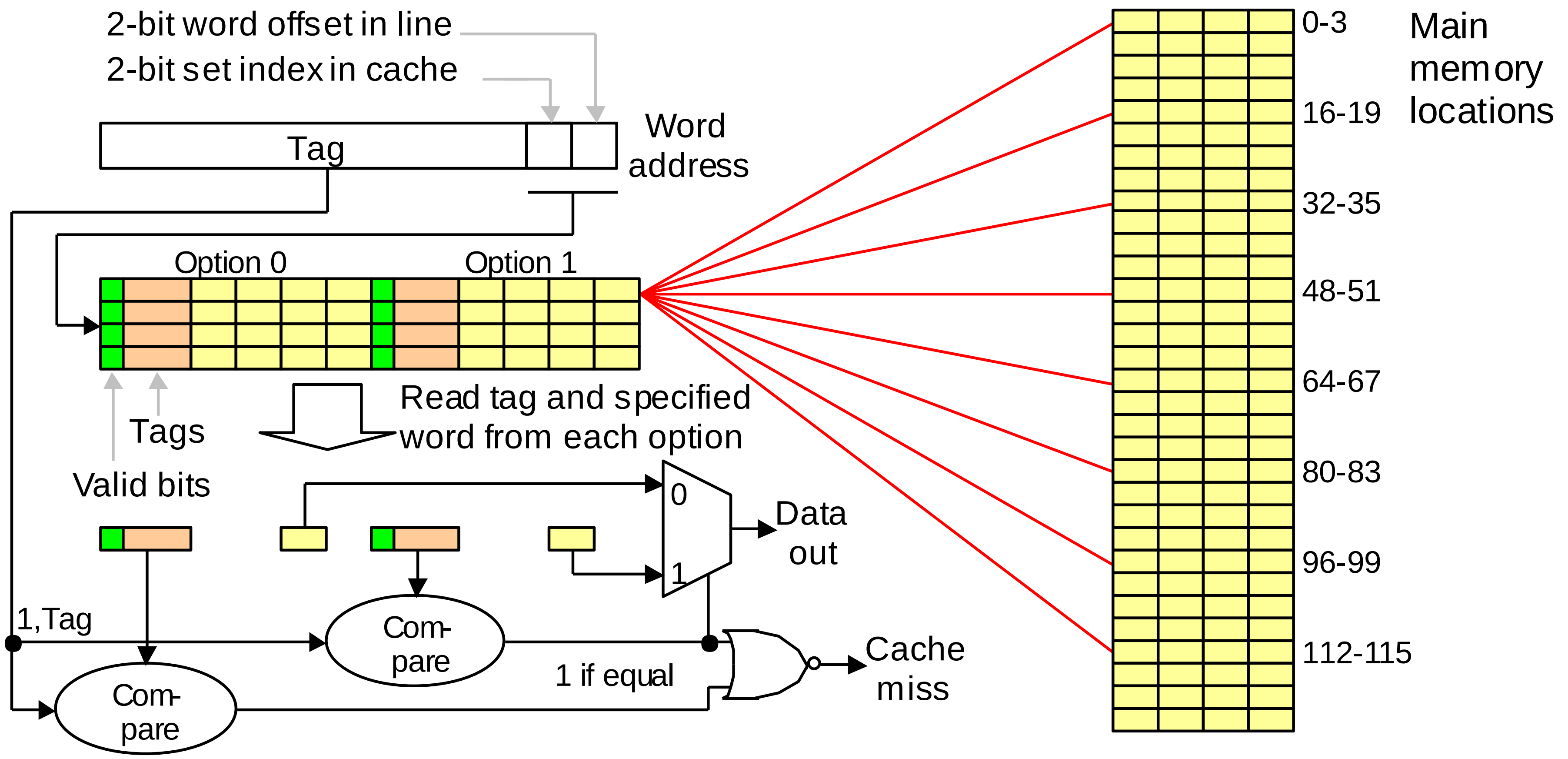




# Fully Associative Cache Design



# Set-Associative Cache



Two-way set-associative cache holding 32 words of data within 4-word lines and 2-line sets.

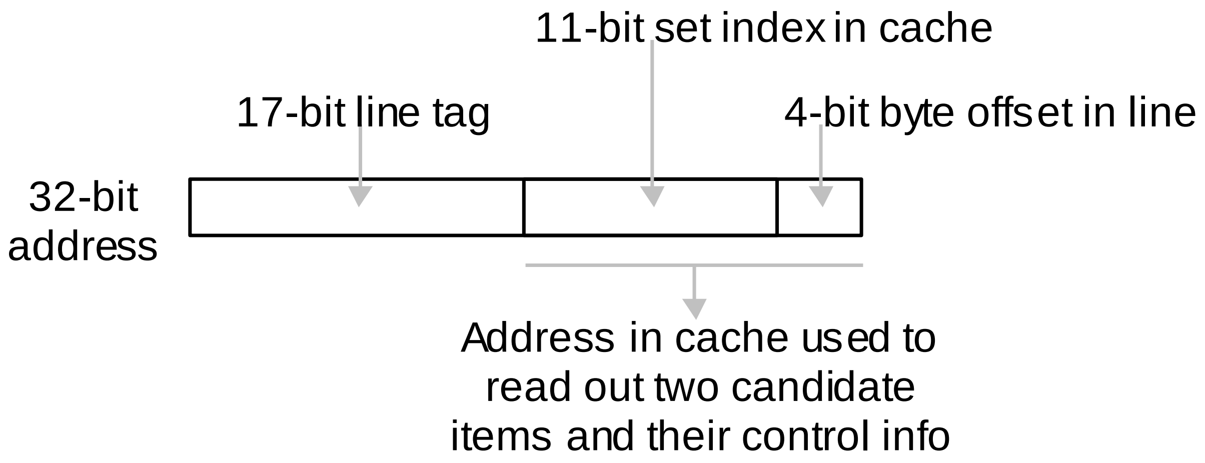
## Example: Accessing a Set-Associative Cache

Show cache addressing scheme for a byte-addressable memory with 32-bit addresses. Cache line width $2^W = 16$ B. Set size $2^S = 2$ lines. Cache size $2^L = 4096$ lines (64 KB).

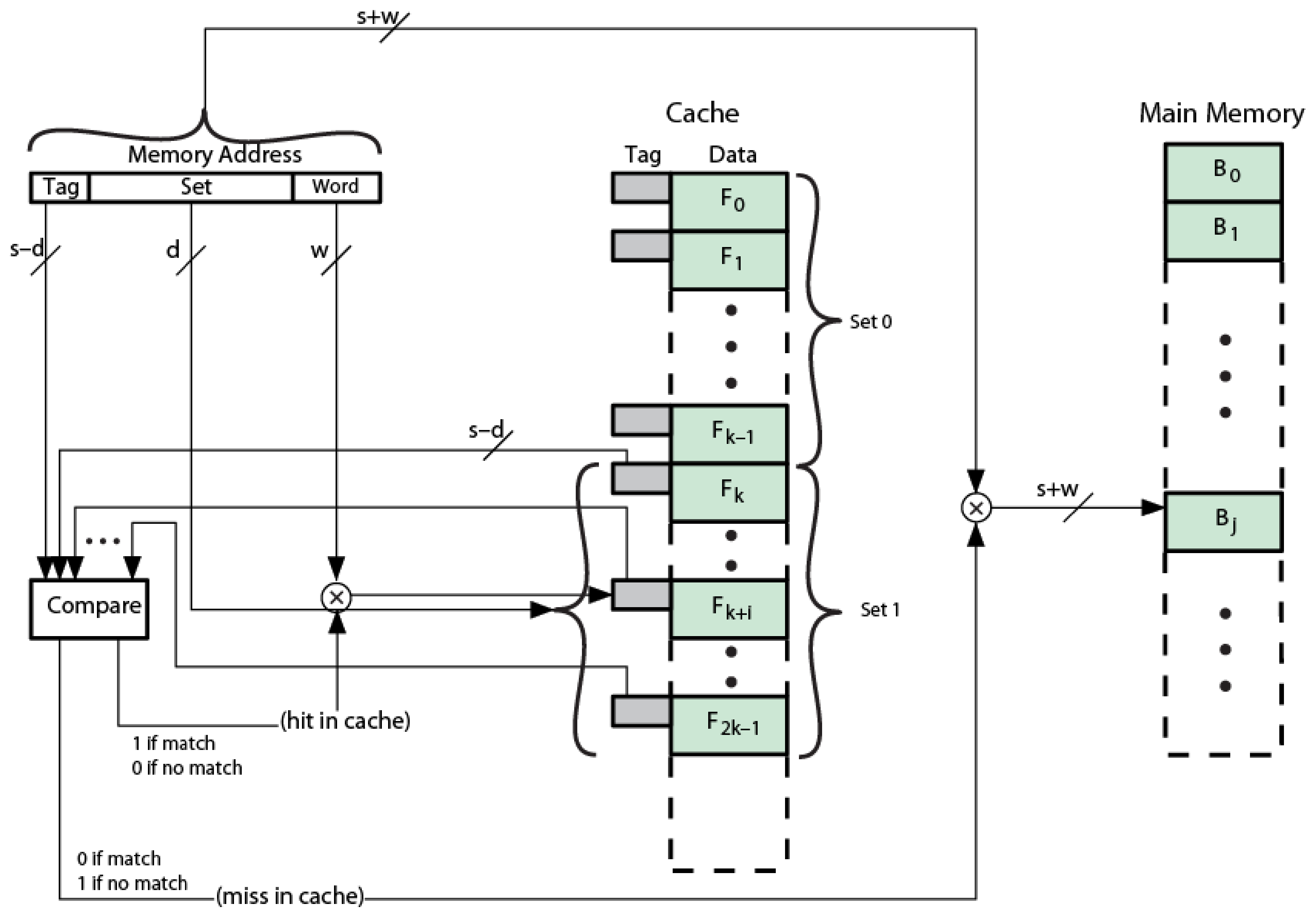
**Solution**

Byte offset in line is $\log_2 16 = 4$ b. Cache set index is $(\log_2 4096/2) = 11$ b. This leaves 32 – 11 – 4 = 17 b for the tag.

Components of the 32-bit address in an example two-way set-associative cache.


11-bit set index in cache
17-bit line tag
4-bit byte offset in line
32-bit address
Address in cache used to read out two candidate items and their control info

# K-way set-associative cache



# Quiz 2: Cache Address Mapping

A 64 KB four-way set-associative cache is byte-addressable and contains 32 B lines. Memory addresses are 32 b wide.

a. How wide are the tags in this cache?

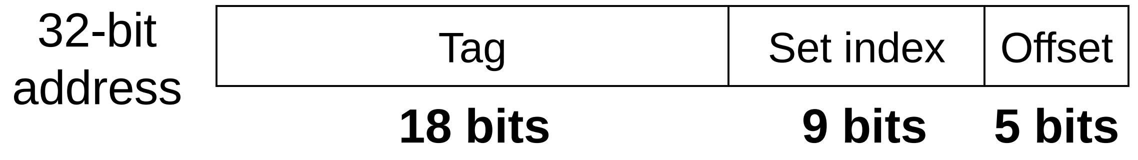
b. Which main memory addresses are mapped to set number 5?

**Solution**

The number of sets in the cache = 64KB/(4 x 32B) = 512.

a. Address (32 b) = 5 b byte offset + 9 b set index + 18 b tag

b. Addresses that have their 9-bit set index equal to 5. These are of the general form $2^{14}a + 2^5 \times 5 + b$; e.g., 160-191, 16 554-16 575, . . .



Tag width =
32 – 5 – 9 = 18

Set size = 4 $\times$ 32 B = 128 B
Number of sets = $2^{16}/2^7 = 2^9$

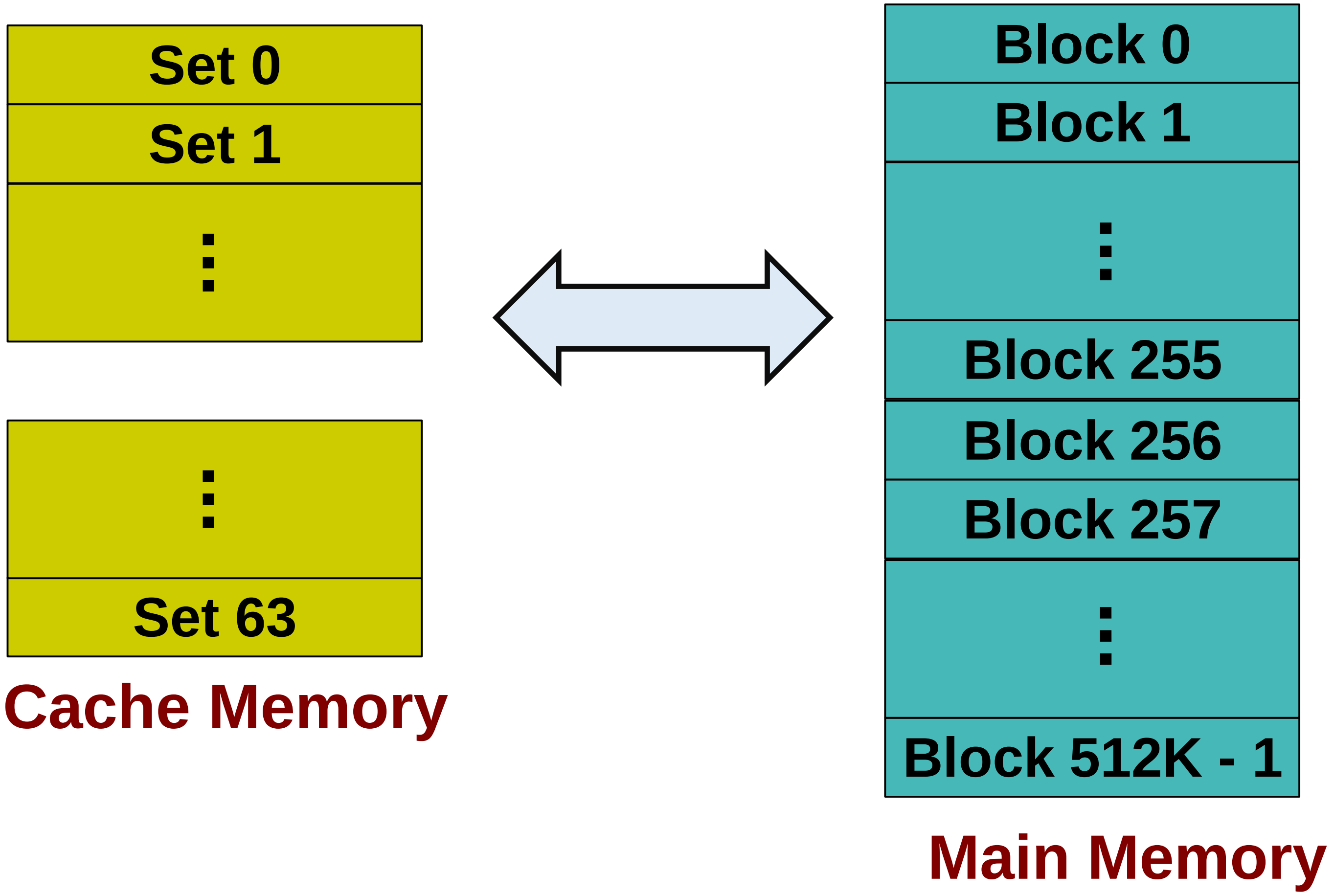
Line width =
32 B = $2^5$ B

## (c) N-way Set Associative Mapping

- A group of N consecutive blocks in the cache is called a *set*.
- This algorithm is a balance of direct mapping and associative mapping.
  - Like direct mapping, a MM block is mapped to a set.

    Set Number = (MM Block Number) % (Number of Sets in Cache)
  - The block can be placed anywhere within the set (there are N choices)
- The value of N is a design parameter:
  - N = 1 :: same as direct mapping.
  - N = number of cache blocks :: same as associative mapping.
  - Typical values of N used in practice are: 2, 4 or 8.

## 4-way Set Associative Mapping



| TAG | SET | WORD |
|---|---|---|
| 13 | 6 | 5 |

Memory Address

- Illustration for N = 4:
  - Number of sets in cache memory = 64.
  - Memory blocks are mapped to a set using modulo-64 operation.
  - Example: MM blocks 0, 64, 128, etc. all map to set 0, where they can occupy any of the four available positions.
- MM address is divided into three fields: *TAG*, *SET* and *WORD*.
  - The TAG field of the address must be associatively compared to the TAG fields of the 4 blocks of the selected set.
  - This instead of requiring a single large associative memory, we need a number of very small associative memories only one of which will be used at a time.

## Q2. How is a block found if present in cache?

- Caches include a TAG associated with each cache block.
  - The TAG of every cache block where the block being requested may be present needs to be compared with the TAG field of the MM address.
  - All the possible tags are compared in parallel, as speed is important.
- Mapping Algorithms?
  - Direct mapping requires a single comparison.
  - Associative mapping requires a full associative search over all the TAGs corresponding to all cache blocks.
  - Set associative mapping requires a limited associated search over the TAGs of only the selected set.

- Use of *valid bit*:
  - There must be a way to know whether a cache block contains valid or garbage information.
  - A valid bit can be added to the TAG, which indicates whether the block contains valid data.
  - If the valid bit is not set, there is no need to match the corresponding TAG.

## Q3. Which block should be replaced on a cache miss?

- With fully associative or set associative mapping, there can be several blocks to choose from for replacement when a miss occurs.
- Two primary strategies are used:
  a) *Random*: The candidate block is selected randomly for replacement. This simple strategy tends to spread allocation uniformly.
  b) *Least Recently Used* (LRU): The block replaced is the one that has not been used for the longest period of time.
     - Makes use of a corollary of temporal locality:

*“If recently used blocks are likely to be used again, then the best candidate for replacement is the least recently used block”*

- To implement the LRU algorithm, the cache controller must track the LRU block as the computation proceeds.

- Example: Consider a 4-way set associative cache.
    - For tracking the LRU block within a set, we use a 2-bit counter with every block.
    - When hit occurs:
        - Counter of the referenced block is reset to 0.
        - Counters with values originally lower than the referenced one are incremented by 1, and all others remain unchanged.
    - When miss occurs:
        - If the set is not full, the counter associated with the new block loaded is set to 0, and all other counters are incremented by 1.
        - If the set is full, the block with counter value 3 is removed, the new block put in its place, and the counter set to 0. The other three counters are incremented by 1.

- It may be verified that the counter values of occupied blocks are all distinct.
- An example:

| | Initial | Miss: Block 2 | Miss: Block 0 | Miss: Block 3 | Miss: Block 1 | Hit: Block 0 |
|---|---|---|---|---|---|---|
| Block 0 | x | x | 0 | 1 | 2 | 0 |
| Block 1 | x | x | x | x | 0 | 1 |
| Block 2 | x | 0 | 1 | 2 | 3 | 3 |
| Block 3 | x | x | x | 0 | 1 | 2 |

| | Miss: Block 2 | Hit: Block 3 | Hit: Block 2 | Hit: Block 0 | Miss: Block 1 | Hit: Block 1 |
|---|---|---|---|---|---|---|
| Block 0 | 1 | 2 | 2 | 0 | 1 | 1 |
| Block 1 | 2 | 3 | 3 | 3 | 0 | 0 |
| Block 2 | 0 | 1 | 0 | 1 | 2 | 2 |
| Block 3 | 3 | 0 | 1 | 2 | 3 | 3 |

## Types of Cache Misses

a) **Compulsory Miss**
   - On the first access to a block, the block must be brought into the cache.
   - Also known as cold start misses, or first reference misses.
   - Can be reduced by increasing cache block size or prefetching cache blocks.

b) **Capacity Miss**
   - Blocks may be replaced from cache because the cache cannot hold all the blocks needed by a program.
   - Can be reduced by increasing the total cache size.

## c) Conflict Miss

- In case of direct mapping or N-way set associative mapping, several blocks may be mapped to the same block or set in the cache.
- May result in block replacements and hence access misses, even though all the cache blocks may not be occupied..
- Can be reduced by increasing the value of N (cache associativity).

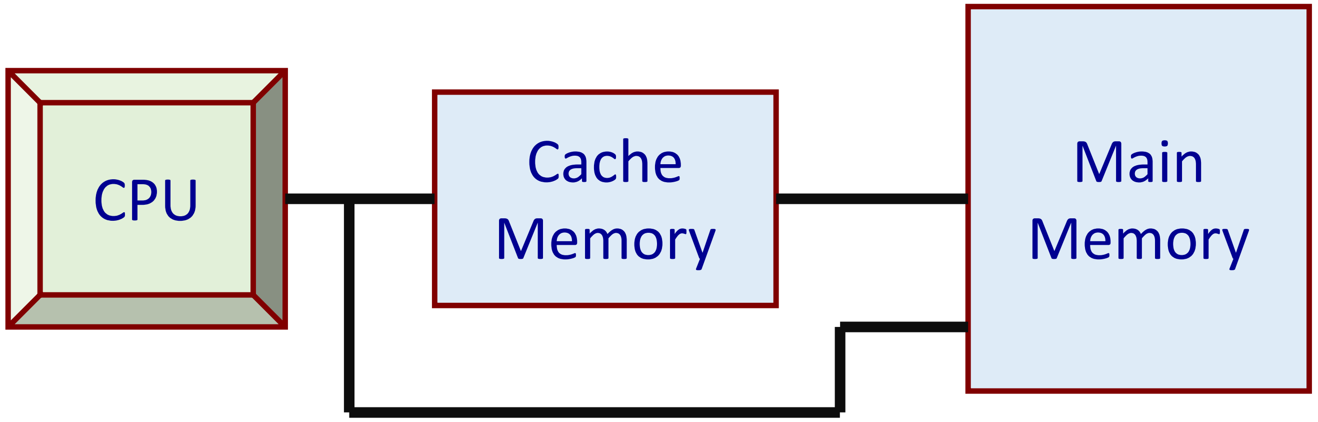
## Q4. What happens on a write?

- Statistical data suggests that read operations (including instruction fetches) dominate processor cache accesses.
  - All instruction fetch operations are read.
  - Most instructions do not write to memory.
- Making the common case fast:
  - Optimize cache accesses for reads.
  - But Amdahl’s law reminds that for high performance designs we cannot ignore the speed of write operations.

- The common case (read operations) is relatively easy to make faster.
    - A block(s) can be read at the same time while the TAG is being compared with the block address.
    - If the read is a HIT the data can be passed to the CPU; if it is a MISS ignore it.
- Problems with write operations:
    - The CPU specifies the size of the write (between 1 and 8 bytes), and only that portion of a block has to be changed.
        - Implies a read-modify-write sequence of operations on the block.
        - Also, the process of modifying the block cannot begin until the TAG is checked to see if it is a hit.
    - Thus, cache write operations take more time than cache read operations.

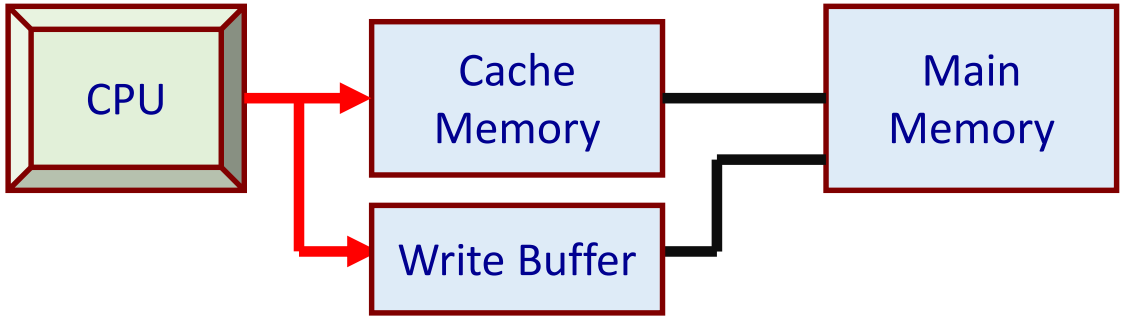
## Cache Write Strategies

- Cache designs can be classified based on the write and memory update strategy being used.
    1. Write Through / Store Through
    2. Write Back / Copy Back



## (a) Write Through Strategy

- Information is written to both the cache block and the main memory block.
- Features:
  - Easier to implement
  - Read misses do not result in writes to the lower level (i.e. MM).
  - The lower level (i.e. MM) has the most updated version of the data – important for I/O operations and multiprocessor systems.
  - A write buffer is often used to reduce CPU write stall time while data is written to main memory.



- **Perfect Write Buffer:**
  - All writes are handled by write buffer; no stalling for write operations.
  - For unified L1 cache.

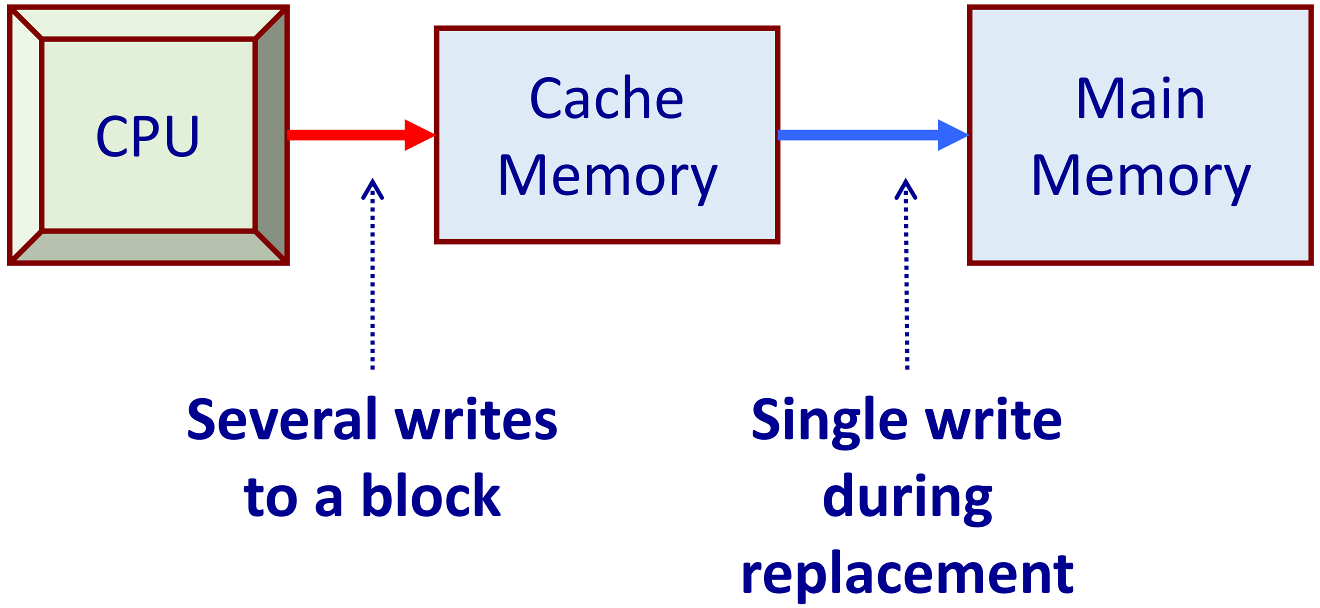
    Stall Cycles / Memory Access = % Reads x $(1 - H_{L1}) . t_{MM}$

- **Realistic Write Buffer:**
  - A percentage of write stalls are not eliminated when the write buffer is full.
  - For unified L1 cache,

    Stall Cycles / Memory Access = (% Reads x $(1 - H_{L1})$ + % write stalls not eliminated) x $t_{MM}$

## (b) Write Back Strategy

- Information is written only to the cache block.
- A modified cache block is written to MM only when it is replaced.
- Features:
  - Writes occur at the speed of cache memory.
  - Multiple writes to a cache block requires only one write to MM.
  - Uses less memory bandwidth, makes it attractive to multiprocessors.
- Write-back cache blocks can be *clean* or *dirty*.
  - A status bit called *dirty bit* or *modified bit* is associated with each cache block, which indicates whether the block was modified in the cache (0: clean, 1: dirty).
  - If the status is clean, the block is not written back to MM while being replaced.


CPU
Cache Memory
Main Memory
Several writes to a block
Single write during replacement

## Cache Write Miss Policy

- Since information is usually not needed immediately on a write miss, two options are possible on a cache write miss:

a) **Write Allocate**
   - The missed block is loaded into cache on a write miss, followed by write hit actions.
   - Requires a cache block to be *allocated* for the block to be written into.

b) **No-Write Allocate**
   - The block is modified only in the lower level (i.e. MM), and not loaded into cache.
   - Cache block is *not allocated* for the block to be written into.

- Typical usage:

  a) Write-back cache with write-allocate
     - In order to capture subsequent writes to the block in cache.

  b) Write-through cache with no-write-allocate
     - Since subsequent writes still have to to go to MM.

## Estimation of Miss Penalties

- **Write-Through Cache**
  - Write Hit Operation:
    - Without write buffer, miss penalty = $t_{MM}$
    - With perfect write buffer, miss penalty = 0
- **Write-Back Cache**
  - Write Hit Operation
    - Miss penalty = 0

- **Write-Back Cache (with Write Allocate)**
  - **Write Hit Operation**
    - Miss penalty = 0
  - **Read or Write Miss Operation**
    - If the replaced block is clean, miss penalty = $t_{MM}$
      - No need to write the block back to MM.
      - New block to be brought into MM ($t_{MM}$).
    - If the replaced block is dirty, miss penalty = 2 $t_{MM}$
      - Write the block to be replaced to MM ($t_{MM}$).
      - New block to be brought into MM ($t_{MM}$).