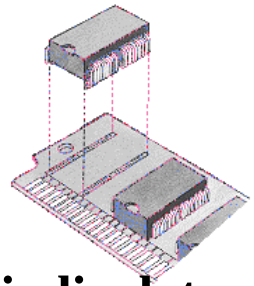
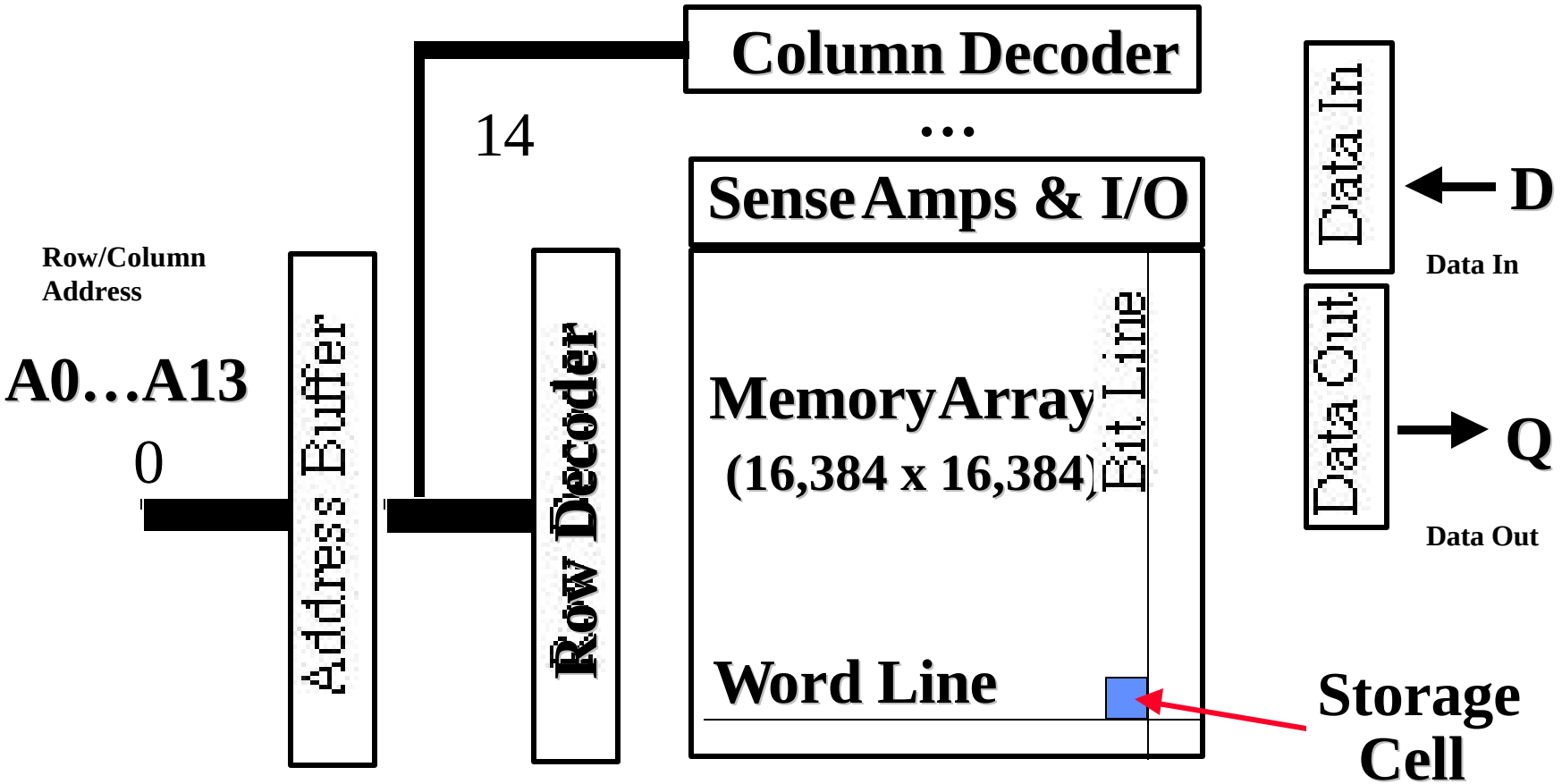
Main Memory



- **Main memory generally utilizes Dynamic RAM (DRAM), which use a single transistor to store a bit, but require a periodic data refresh by reading every row (~every 8 msec).**
- **Static RAM may be used for main memory if the added expense, low density, high power consumption, and complexity is feasible (e.g. Cray Vector Supercomputers).**
- **Main memory performance is affected by:**
  - **Memory latency: Affects cache miss penalty. Measured by:**
    - **Access time: The time it takes between a memory access request is issued to main memory and the time the requested information is available to cache/CPU.**
    - **Cycle time: The minimum time between requests to memory (greater than access time in DRAM to allow address lines to be stable)**
  - **Memory bandwidth: The maximum sustained data transfer rate between main memory and cache/CPU.**

Logical DRAM Organization (16 Mbit)



Control Signals:
Row Access Strobe (RAS): Low to latch row address
Column Address Strobe (CAS): Low to latch column address
Write Enable (WE)
Output Enable (OE)

Key DRAM Speed Parameters

- **Row Access Strobe (RAS)Time:**
  - **Minimum time from RAS (Row Access Strobe) line falling to the first valid data output.**
  - **A major component of memory latency and access time.**
  - **Only improves 5% every year.**

- **Column Access Strobe (CAS) Time/data transfer time:**
  - **The minimum time required to read additional data by changing column address while keeping the same row address.**
  - **Along with memory bus width, determines peak memory bandwidth.**

Memory Hierarchy: Motivation

- The gap between CPU performance and realistic (non-ideal) main memory speed has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions.
- The memory hierarchy is organized into several levels of memory or storage with the smaller, more expensive, and faster levels closer to the CPU: registers, then primary or Level 1 (L_1) SRAM Cache, then possibly one or more secondary cache levels (L_2, L_3…), then DRAM main memory, then mass storage (virtual memory).
- Each level of the hierarchy is a subset of the level below: data found in a level is also found in the level below but at a lower speed.
- Each level maps addresses from a larger physical memory/storage level to a smaller level above it.
- This concept is greatly aided by the principal of locality both temporal and spatial which indicates that programs tend to reuse data and instructions that they have used recently or those stored in their vicinity leading to working set of a program.

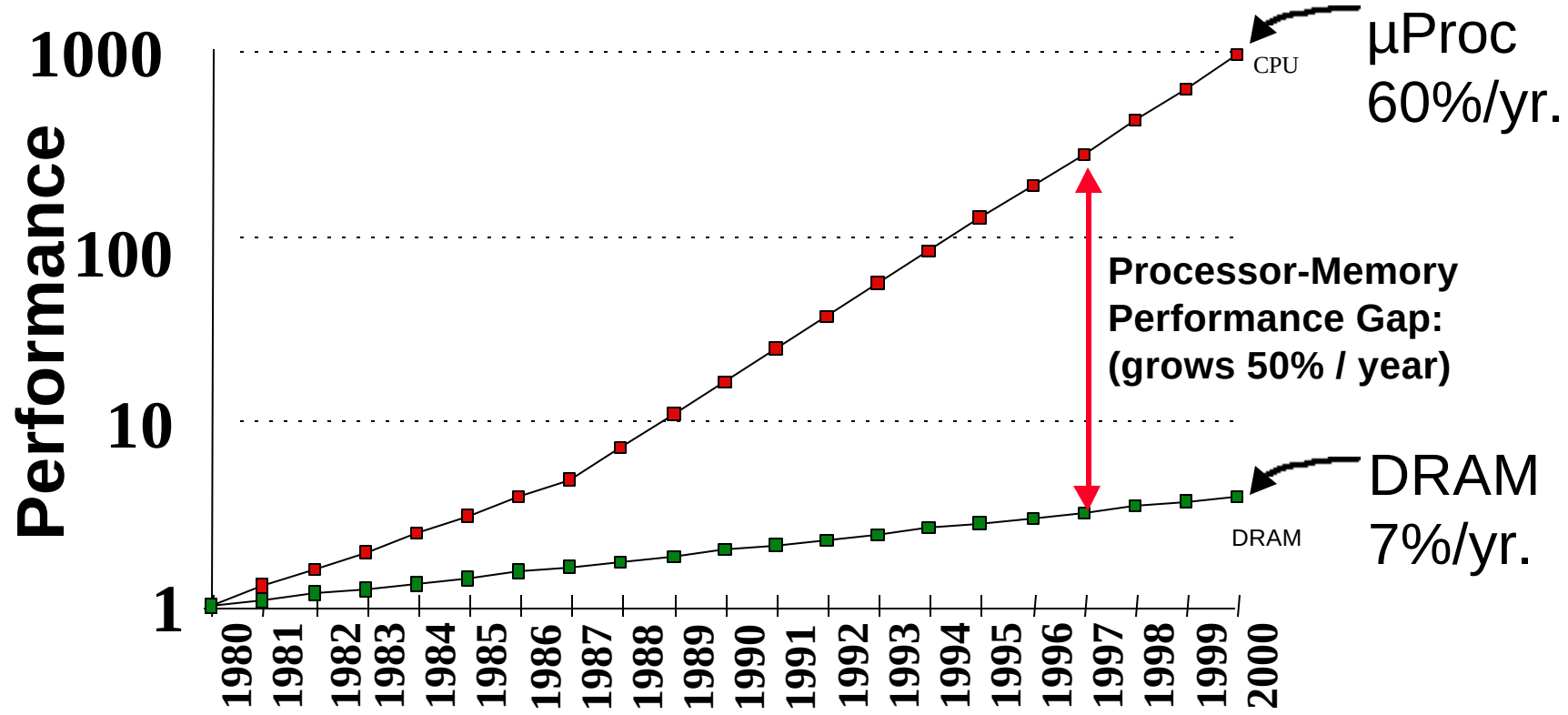
From Technology Trends

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

DRAM Generations

Year	Size	RAS (ns)	CAS (ns)	Cycle Time
1980	64 Kb	150-180	75	250 ns
1983	256 Kb	120-150	50	220 ns
1986	1 Mb	100-120	25	190 ns
1989	4 Mb	80-100	20	165 ns
1992	16 Mb	60-80	15	120 ns
1996	64 Mb	50-70	12	110 ns
1998	128 Mb	50-70	10	100 ns
2000	256 Mb	45-65	7	90 ns
2002	512 Mb	40-60	5	80 ns
	8000:1 (Capacity)		15:1 (~bandwidth)	3:1 (Latency)

Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap



Processor-DRAM Performance Gap Impact

- To illustrate the performance impact, assume a single-issue pipelined RISC CPU with ideal CPI = 1 using non-ideal memory.
- Ignoring other factors, the minimum cost of a full memory access in terms of number of wasted CPU cycles:

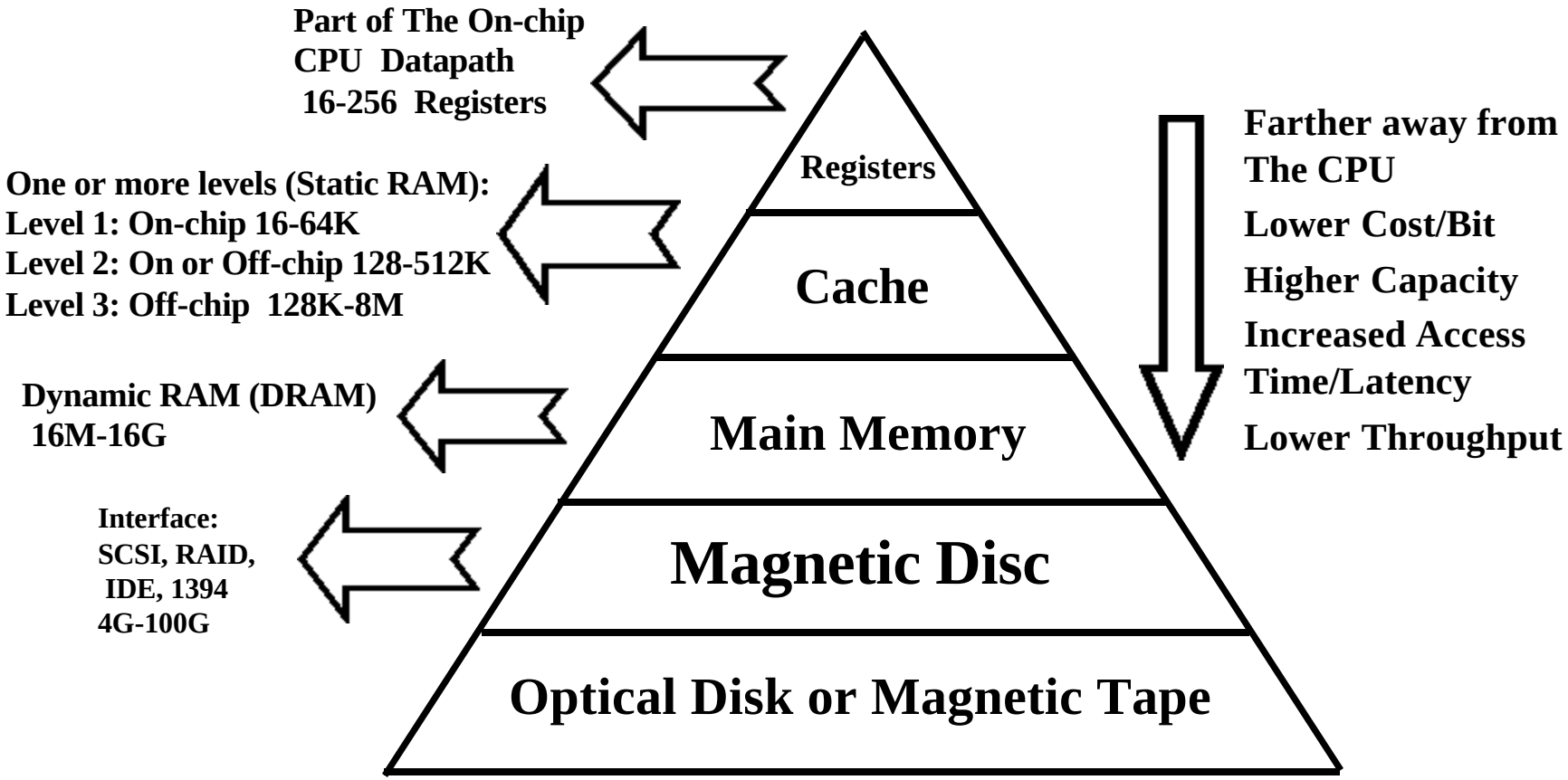
Year	CPU speed MHZ	CPU cycle ns	Memory Access ns	Minimum CPU cycles or instructions wasted
1986:	8	125	190	190/125 - 1 = 0.5
1989:	33	30	165	165/30 -1 = 4.5
1992:	60	16.6	120	120/16.6 -1 = 6.2
1996:	200	5	110	110/5 -1 = 21
1998:	300	3.33	100	100/3.33 -1 = 29
2000:	1000	1	90	90/1 - 1 = 89
2002:	2000	.5	80	80/.5 - 1 = 159

Memory Hierarchy: Motivation

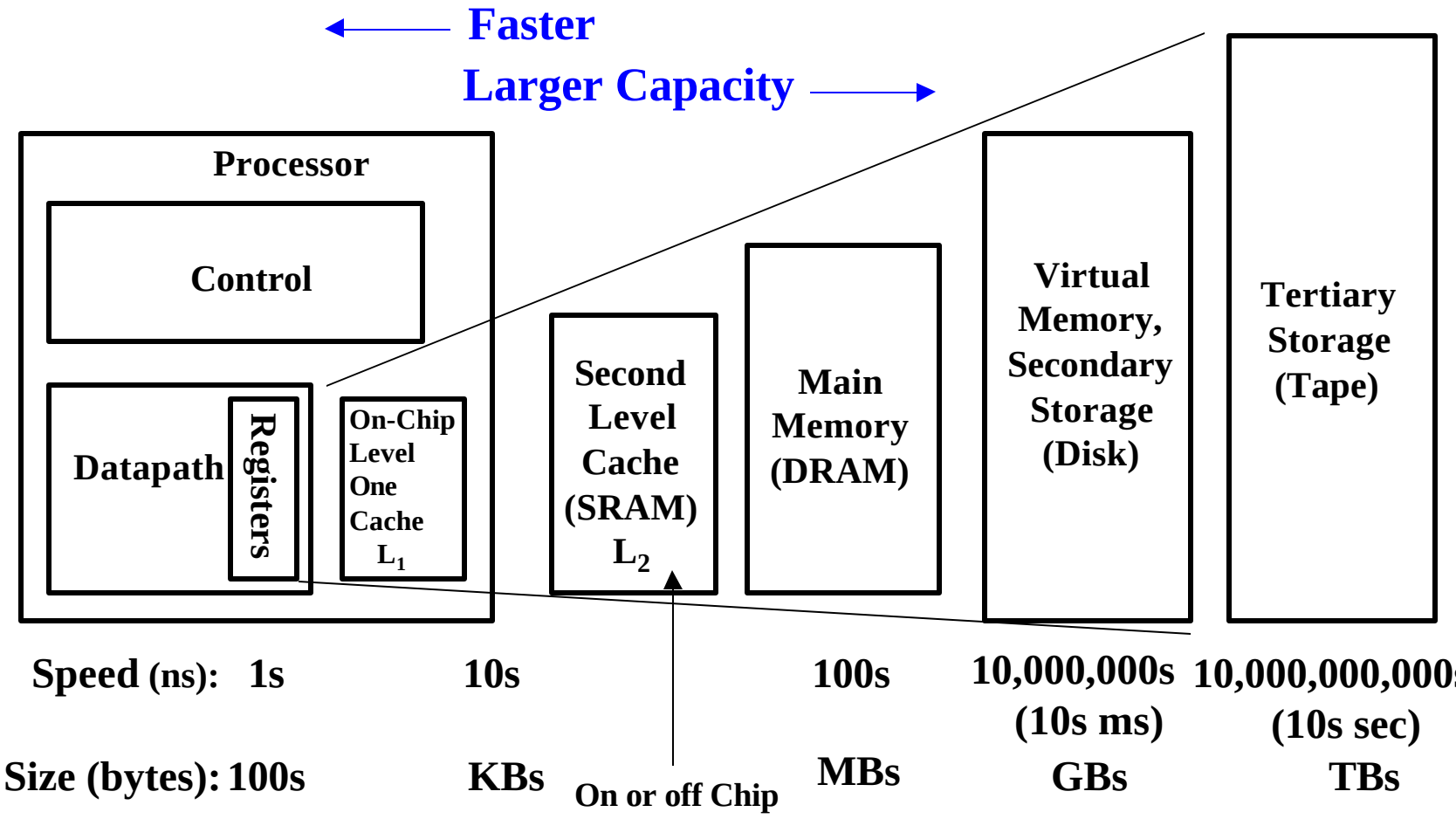
The Principle Of Locality

- **Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).**
- **Two Types of locality:**
  - **Temporal Locality: If an item is referenced, it will tend to be *referenced again soon*.**
  - **Spatial locality: If an item is referenced, items whose *addresses are close* will tend to be referenced soon.**
- **The presence of locality in program behavior, makes it possible to satisfy a large percentage of program access needs using memory levels with *much less capacity* than program address space.**

Levels of The Memory Hierarchy



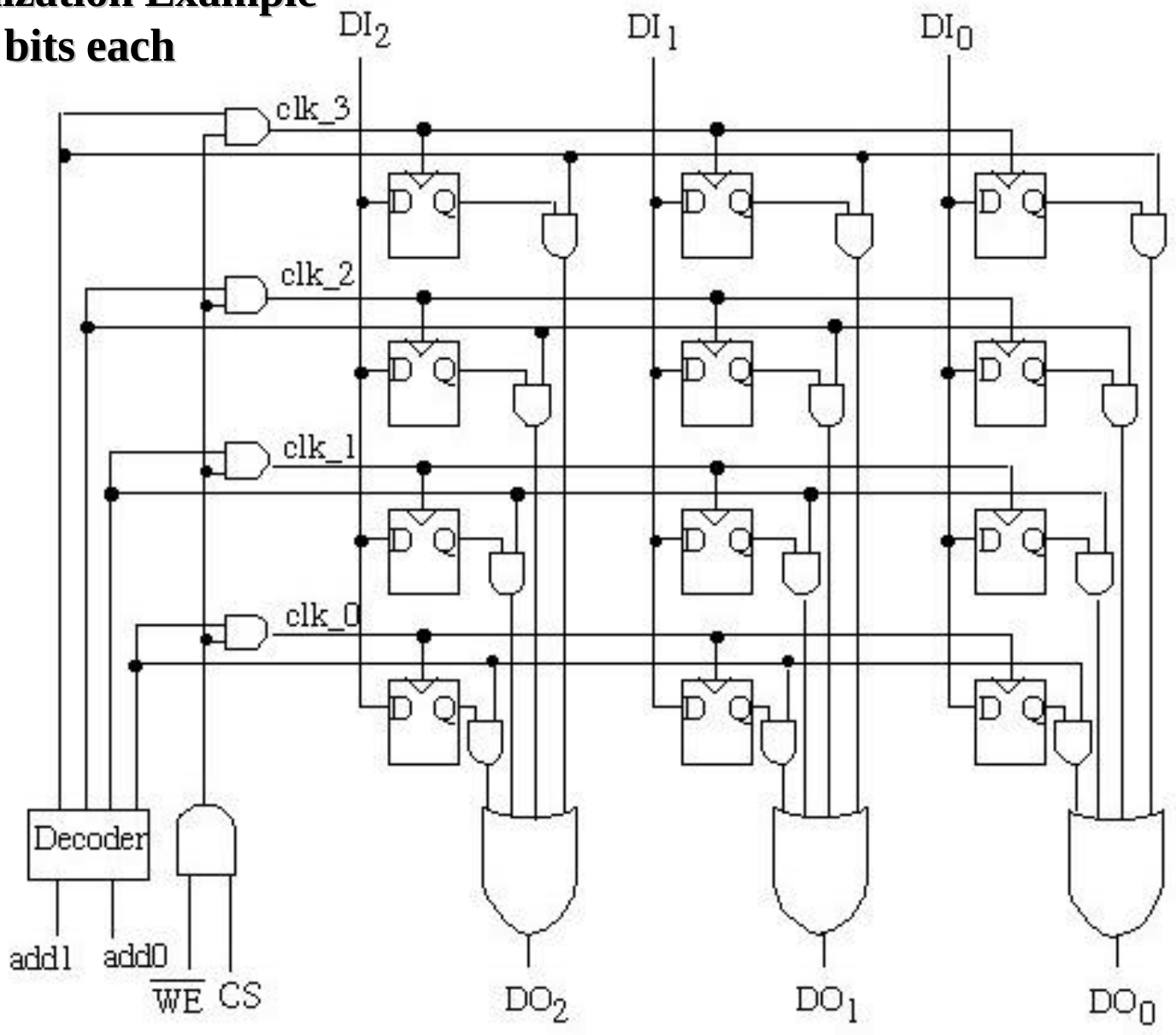
A Typical Memory Hierarchy
(With Two Levels of Cache)



Levels of The Memory Hierarchy

Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 4 MB	<4 GB	> 1 GB
Implementation technology	Custom memory with multiple ports, CMOS or BiCMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	2–5	3–10	80–400	5,000,000
Bandwidth (in MB/sec)	4000–32,000	800–5000	400–2000	4–32
Managed by	Compiler	Hardware	Operating system	Operating system/user
Backed by	Cache	Main memory	Disk	Tape

SRAM Organization Example
4 words X 3 bits each



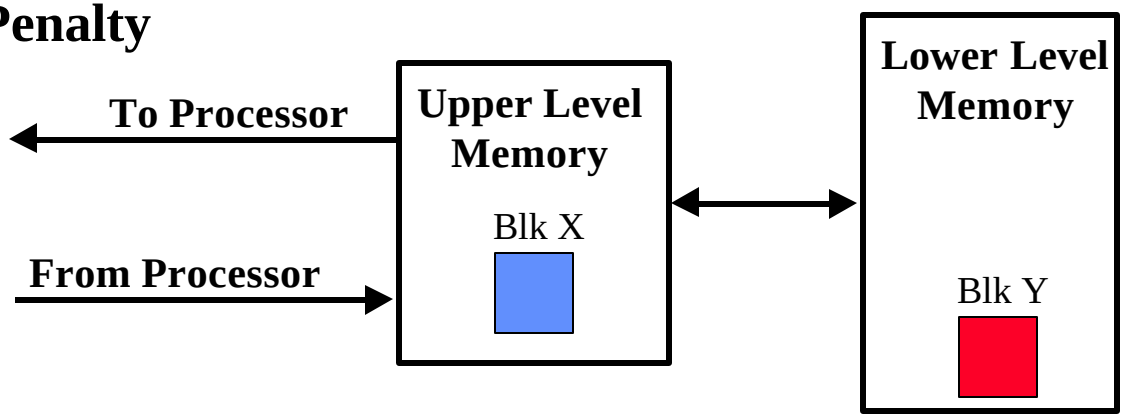
Memory Hierarchy Operation

If an instruction or operand is required by the CPU, the levels of the memory hierarchy are searched for the item starting with the level closest to the CPU (Level 1, L_1 cache):

- **If the item is found, it's delivered to the CPU resulting in *a cache hit* without searching lower levels.**
- **If the item is missing from an upper level, resulting in *a cache miss*, then the level just below is searched.**
- **For systems with several levels of cache, the search continues with cache level 2, 3 etc.**
- **If all levels of cache report a miss then main memory is accessed for the item.**
  - **CPU $\leftrightarrow$ cache $\leftrightarrow$ memory: Managed by hardware.**
- **If the item is not found in main memory resulting in a page fault, then disk (virtual memory) is accessed for the item.**
  - **Memory $\leftrightarrow$ disk: Mainly managed by the operating system with hardware support.**

Memory Hierarchy: Terminology

- **A Block:** The smallest unit of information transferred between two levels.
- **Hit:** Item is found in some block in the upper level (example: Block X)
  - **Hit Rate:** The fraction of memory access found in the upper level.
  - **Hit Time:** Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss
- **Miss:** Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level +
    Time to deliver the block the processor
- Hit Time << Miss Penalty



Cache Concepts

- **Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.**
- **If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.**
- **On a cache miss a block of data must be brought in from main memory to into a cache block frame to possibly *replace* an existing cache block.**
- **The allowed block addresses where blocks can be mapped into cache from main memory is determined by *cache placement strategy*.**
- **Locating a block of data in cache is handled by cache block identification mechanism.**
- **On a cache miss the cache block being removed is handled by the *block replacement strategy* in place.**
- **When a write to cache is requested, a number of main memory update strategies exist as part of *the cache write policy*.**

Cache Design & Operation Issues

Q1: Where can a block be placed cache?

(Block placement strategy & Cache organization)

- **Fully Associative, Set Associative, Direct Mapped.**

Q2: How is a block found if it is in cache?

(Block identification)

- **Tag/Block.**

Q3: Which block should be replaced on a miss?

(Block replacement policy)

- **Random, Least Recently Used (LRU).**

Q4: What happens on a write?

(Cache write policy)

- **Write through, write back.**

We will examine:

- **Cache Placement Strategies**
  - Cache Organization.
- **Locating A Data Block in Cache.**
- **Cache Replacement Policy.**
- **What happens on cache Reads/Writes.**
- **Cache write strategies.**
- **Cache write miss policies.**
- **Cache performance.**

Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frame addresses divide cache into three organizations:

1 **Direct mapped cache:** **A block can be placed in one location only, given by (mapping function):**

(Block address) MOD (Number of blocks in cache)

2 **Fully associative cache:** **A block can be placed anywhere in cache (no mapping function).**

3 **Set associative cache:** **A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:**

(Block address) MOD (Number of sets in cache)

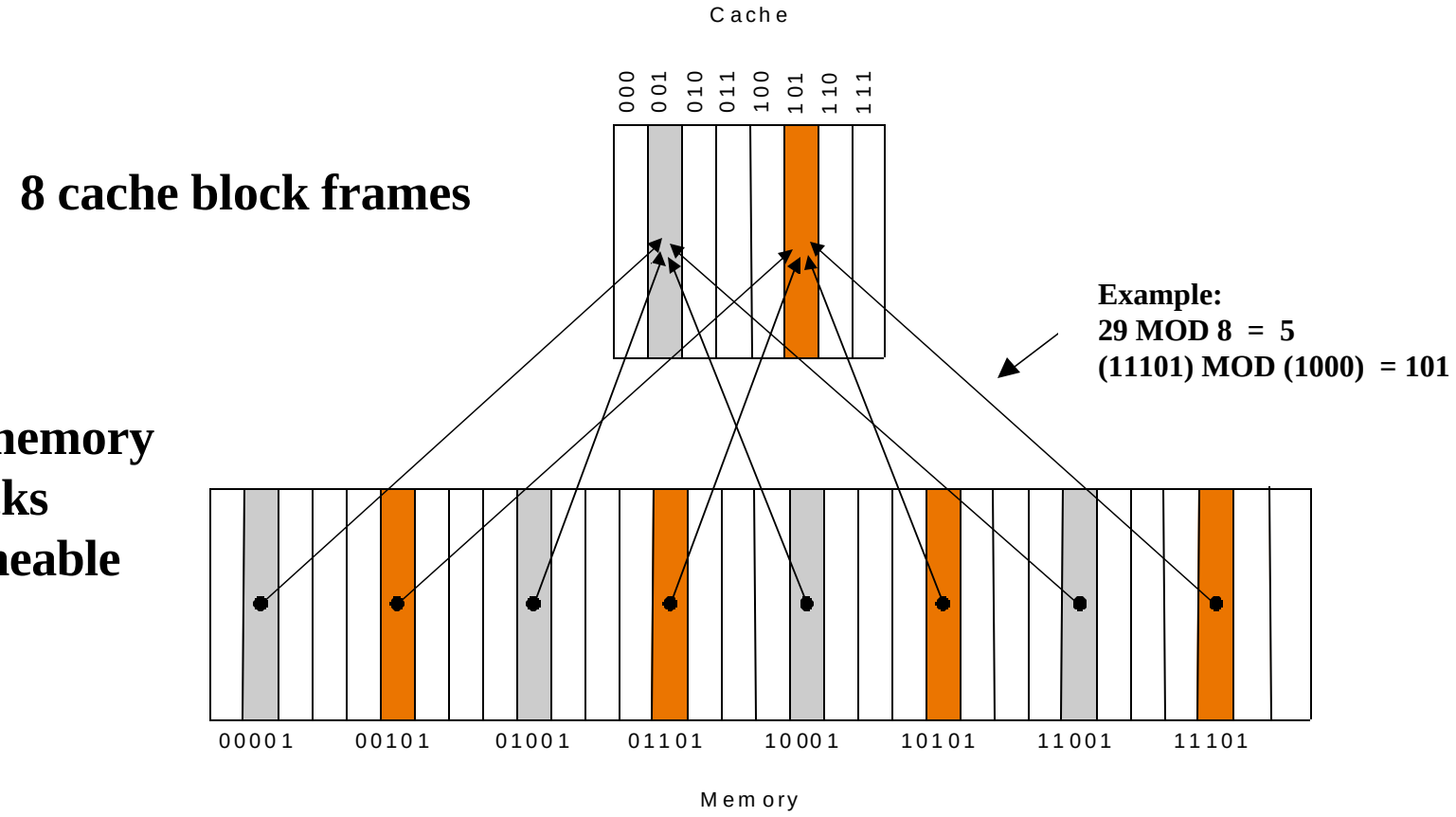
If there are *n* blocks in a set the cache placement is called *n*-way set-associative.

Cache Organization:
Direct Mapped Cache

A block can be placed in one location only, given by:

(Block address) MOD (Number of blocks in cache)

In this case, mapping function: (Block address) MOD (8)



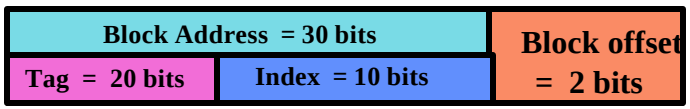
4KB Direct Mapped Cache Example

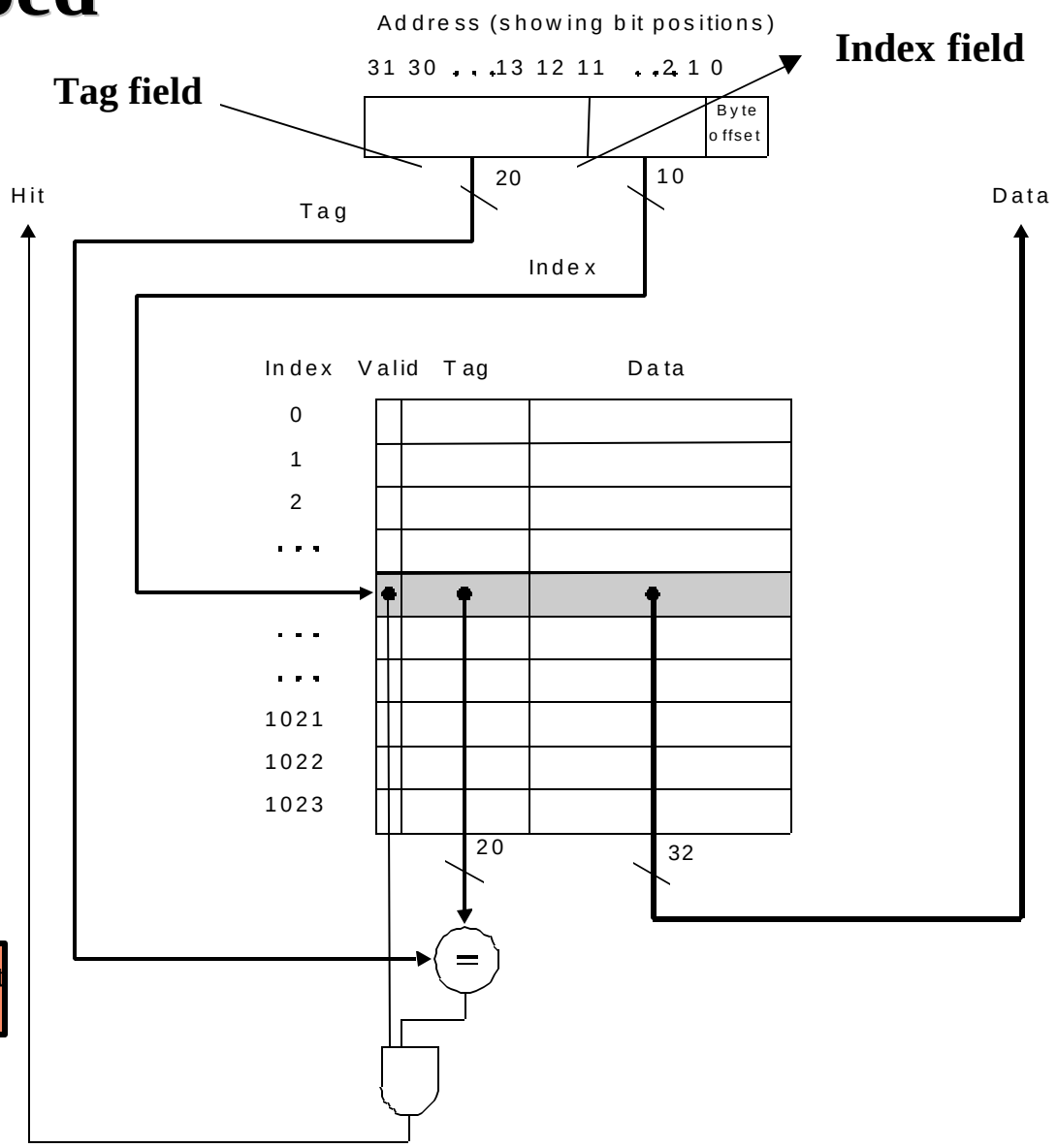
1K = 1024 Blocks
Each block = one word

Can cache up to
2^{32} bytes = 4 GB
of memory

Mapping function:

Cache Block frame number =
(Block address) MOD (1024)

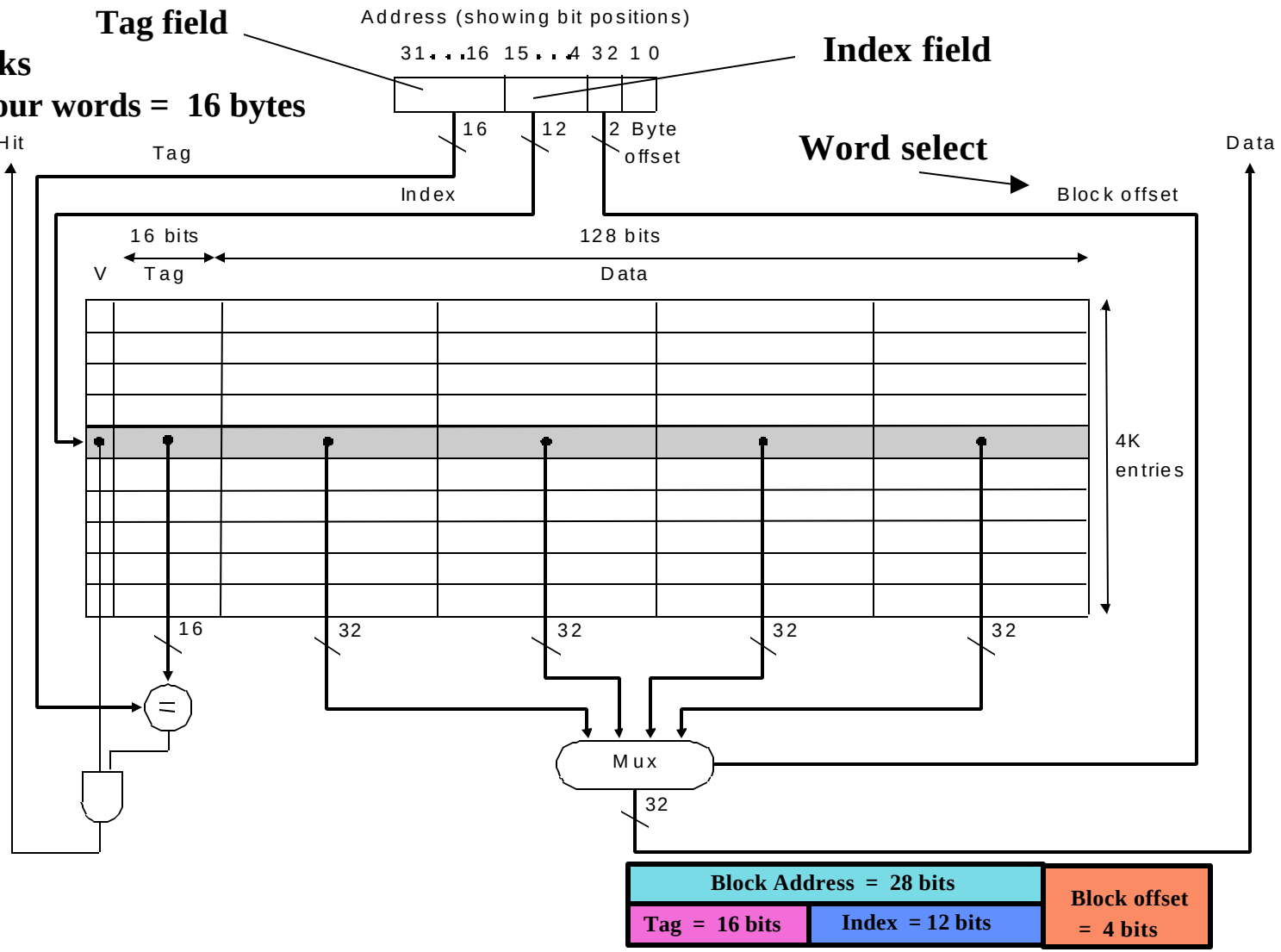




64KB Direct Mapped Cache Example

4K= 4096 blocks
Each block = four words = 16 bytes

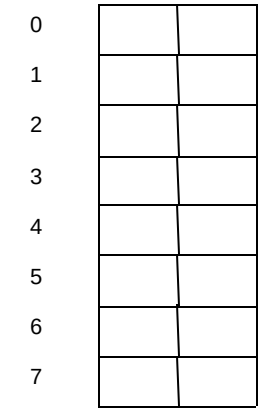
Can cache up to 2^{32} bytes = 4 GB of memory



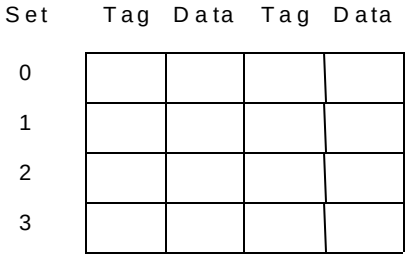
Mapping Function: Cache Block frame number = (Block address) MOD (4096)
Larger blocks take better advantage of spatial locality

Cache Organization:
Set Associative Cache

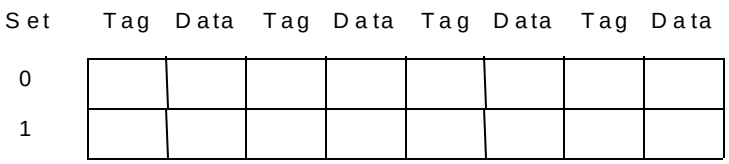
One-way set associative
(direct mapped)



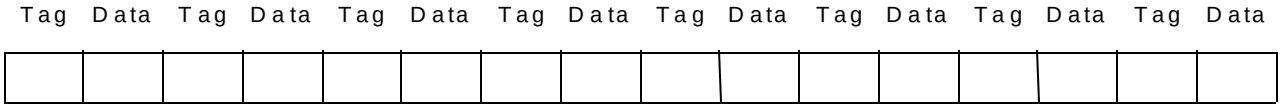
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)



Cache Organization Example

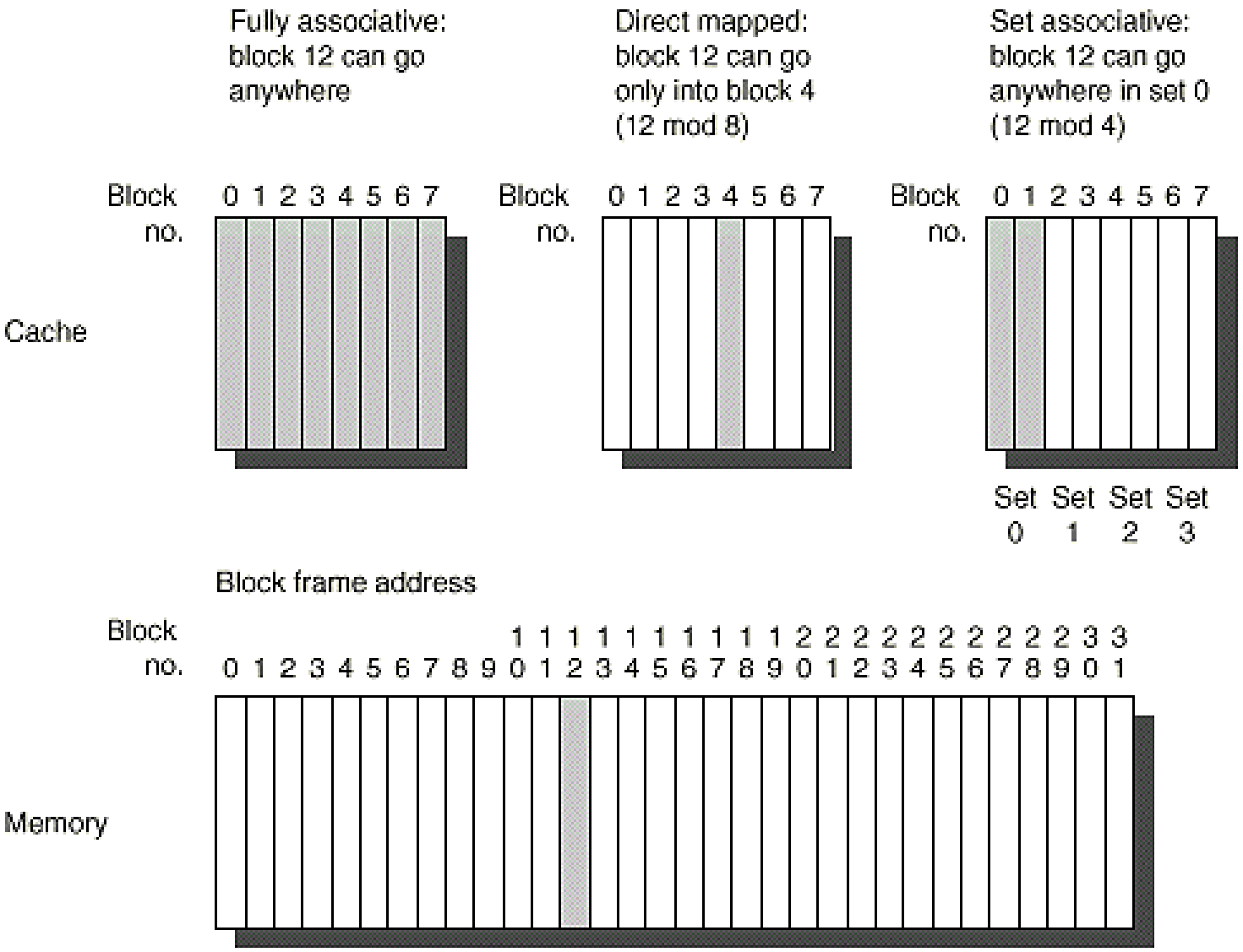
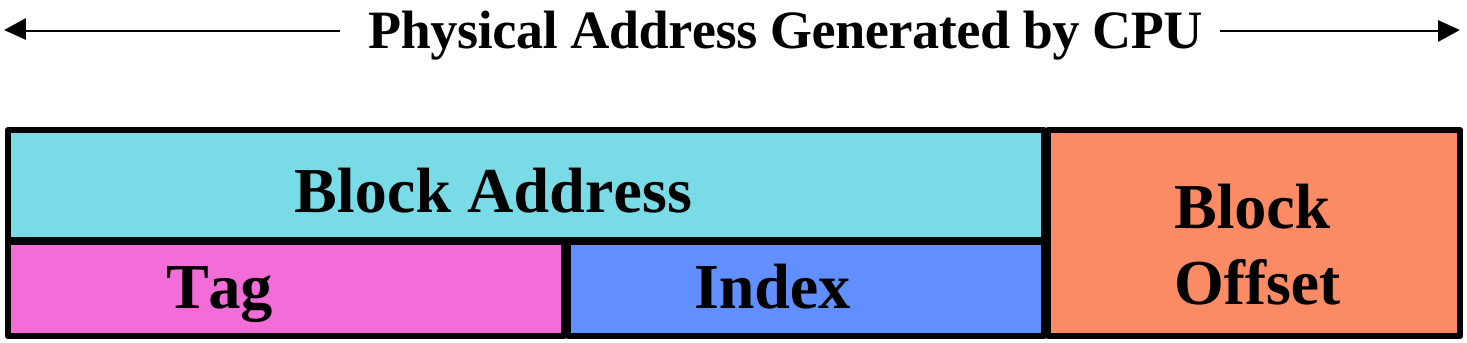


FIGURE 5.2 This example cache has eight block frames and memory has 32 blocks.

Locating A Data Block in Cache

- Each block frame in cache has an address tag.
- The tags of every cache block that might contain the required data are checked or searched in parallel.
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The address from the CPU to cache is divided into:
  - A block address, further divided into:
    - An index field to choose a block set in cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block.

Address Field Sizes



Block offset size = log2(block size)

Index size = log2(Total number of blocks/associativity)

Tag size = address size - index size - offset size

Number of Sets

Mapping function:

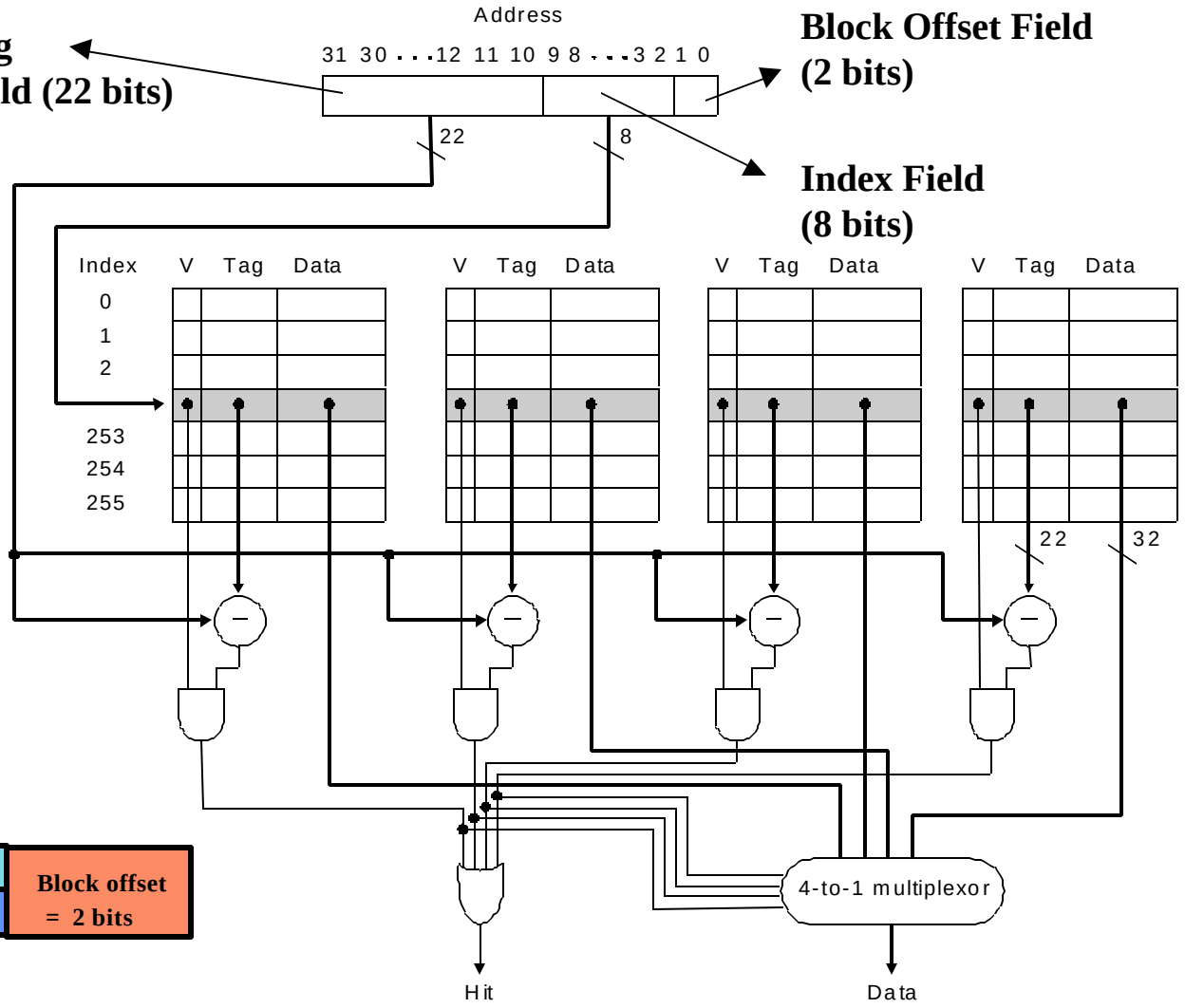
Cache set or block frame number = Index =

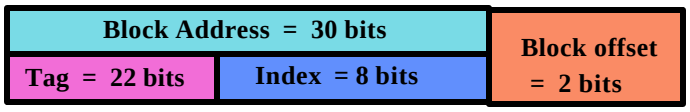
= (Block Address) MOD (Number of Sets)

4K Four-Way Set Associative Cache: MIPS Implementation Example

1024 block frames
Each block = one word
4-way set associative
256 sets

Can cache up to
2^{32} bytes = 4 GB
of memory



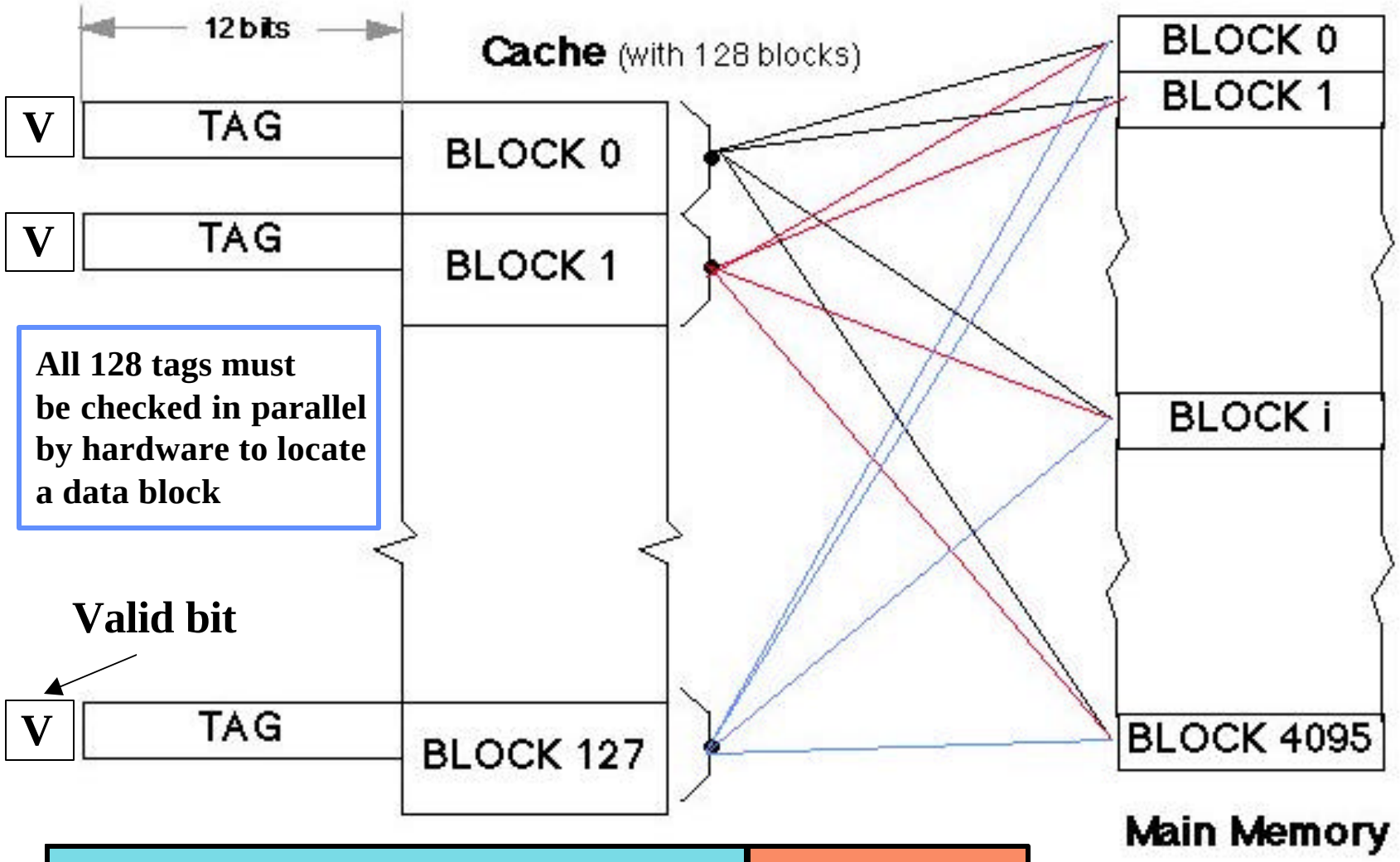


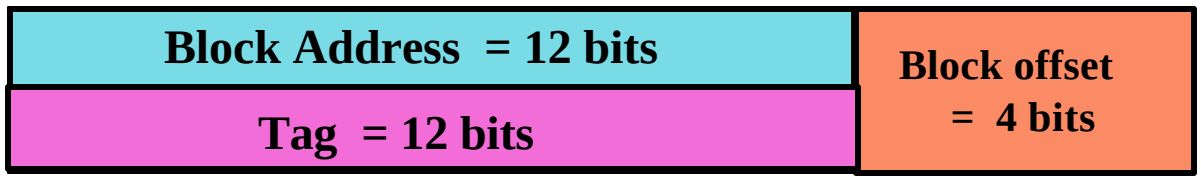
Mapping Function: **Cache Set Number = (Block address) MOD (256)**

Cache Organization/Addressing Example

- **Given the following:**
  - **A single-level L_1 cache with 128 cache block frames**
    - **Each block frame contains four words (16 bytes)**
  - **16-bit memory addresses to be cached (64K bytes main memory or 4096 memory blocks)**
- **Show the cache organization/mapping and cache address fields for:**
    - **Fully Associative cache.**
    - **Direct mapped cache.**
    - **2-way set-associative cache.**

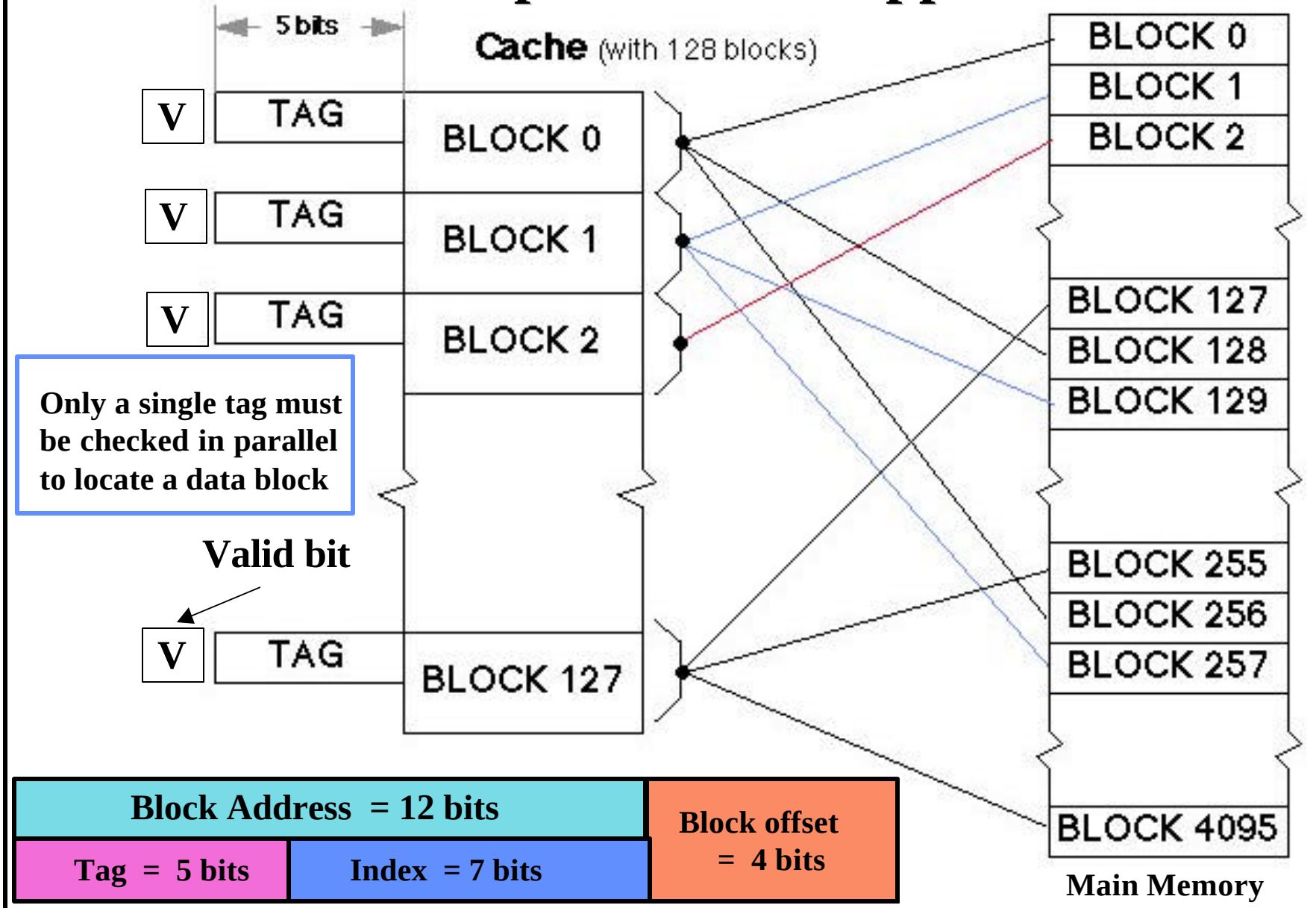
Cache Example: Fully Associative Case





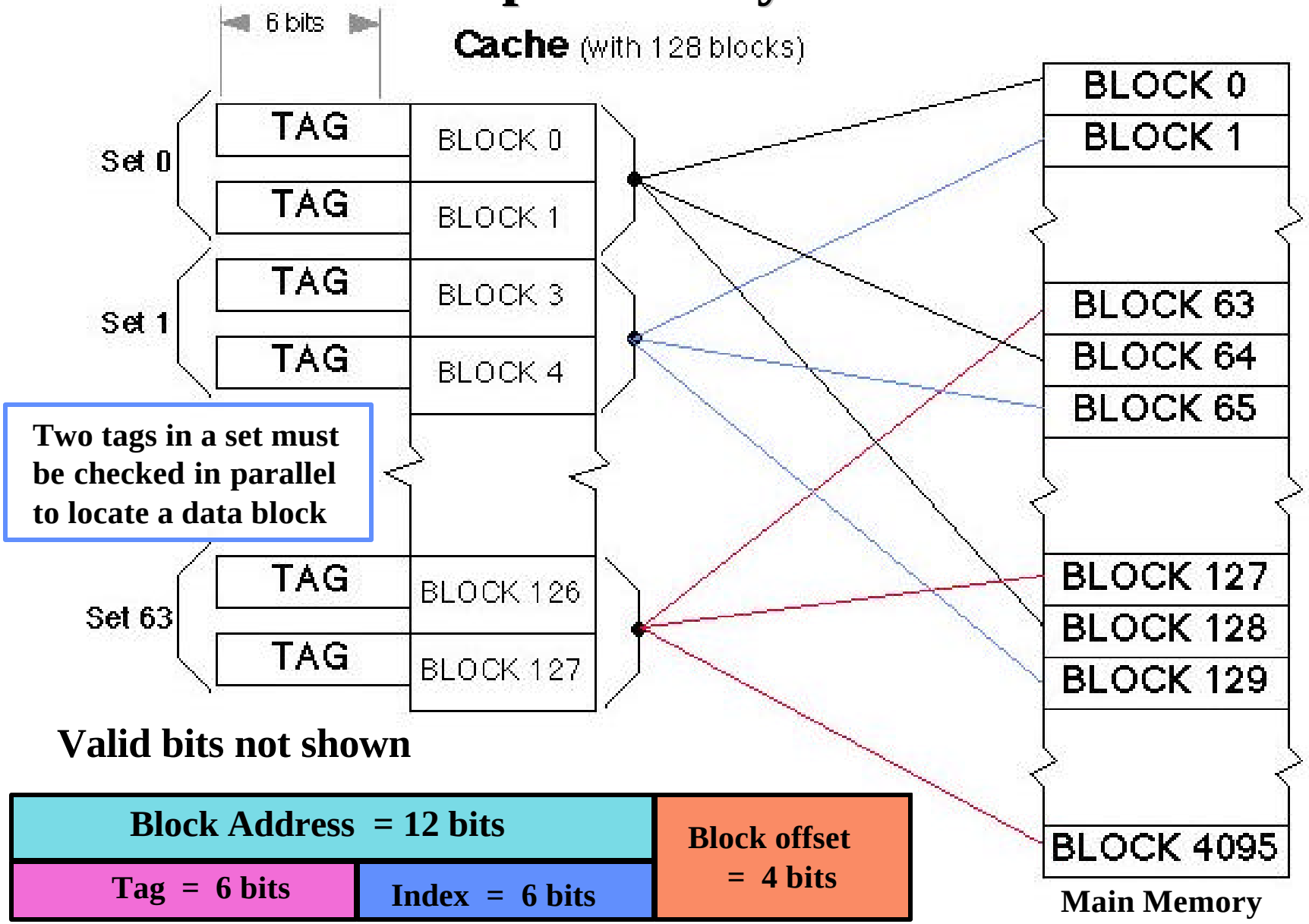
Mapping Function = none

Cache Example: Direct Mapped Case



Mapping Function: Cache Block frame number = Index = (Block address) MOD (128)

Cache Example: 2-Way Set-Associative



Two tags in a set must be checked in parallel to locate a data block

Valid bits not shown

Block Address = 12 bits		Block offset = 4 bits
Tag = 6 bits	Index = 6 bits	

Mapping Function: Cache Set Number = Index = (Block address) MOD (64)

Calculating Number of Cache Bits Needed

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?
  - 64 Kbytes = 16 K words = 2^{14} words = 2^{14} blocks
  - Block size = 4 bytes => offset size = 2 bits,
  - #sets = #blocks = 2^{14} => index size = 14 bits
  - Tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - Bits in cache = #blocks x bits/block = 2^{14} x 49 = 98 Kbytes
- How many total bits would be needed for a 4-way set associative cache to store the same amount of data?
  - Block size and #blocks does not change.
  - #sets = #blocks/4 = (2^{14})/4 = 2^{12} => index size = 12 bits
  - Tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - Bits in cache = #blocks x bits/block = 2^{14} x 51 = 102 Kbytes
- Increase associativity => increase bits in cache

Calculating Cache Bits Needed

- **How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word blocks, assuming a 32-bit address (it can cache 2^{32} bytes in memory)?**
  - **64 Kbytes = 2^{14} words = $(2^{14})/8 = 2^{11}$ blocks**
  - **block size = 32 bytes**

    => offset size = block offset + byte offset = 5 bits,
  - **#sets = #blocks = 2^{11} => index size = 11 bits**
  - **tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits**
  - **bits/block = data bits + tag bits + valid bit = 8 x 32 + 16 + 1 = 273 bits**
  - **bits in cache = #blocks x bits/block = 2^{11} x 273 = 68.25 Kbytes**
- **Increase block size => decrease bits in cache.**

Cache Replacement Policy

- When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is selected by one of two methods:
  - **Random**:
    - Any block is randomly selected for replacement providing uniform allocation.
    - Simple to build in hardware.
    - The most widely used cache replacement strategy.
  - **Least-recently used (LRU)**:
    - Accesses to blocks are recorded and and the block replaced is the one that was not used for the longest period of time.
    - LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated.

Cache Read/Write Operations

- **Statistical data suggest that reads (*including instruction fetches)* dominate processor cache accesses (writes account for 25% of data cache traffic).**
- **In cache reads, a block is read at the same time while the tag is being compared with the block address. If the read is a hit the data is passed to the CPU, if a miss it ignores it.**
- **In cache writes, modifying the block cannot begin until the tag is checked to see if the address is a hit.**
- **Thus in cache writes, tag checking cannot take place in parallel, and only the specific data requested by the CPU can be modified.**
- **Cache is classified according to the write and memory update strategy in place as: write through, or write back.**

Cache Write Strategies

1. **Write Though: Data is written to both the cache block and to a block of main memory.**
   - **The lower level always has the most updated data; an important feature for I/O and multiprocessing.**
   - **Easier to implement than write back.**
   - **A write buffer is often used to reduce CPU write stall while data is written to memory.**
2. **Write back: Data is written or updated only to the cache block frame. The modified cache block is written to main memory when it's being replaced from cache.**
   - **Writes occur at the speed of cache.**
   - **A status bit called a dirty bit, is used to indicate whether the block was modified while in cache; if not the block is not written to main memory.**
   - **Uses less memory bandwidth than write through.**

Cache Write Miss Policy

- **Since data is usually not needed immediately on a write miss two options exist on a cache write miss:**

Write Allocate:

The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:

The block is modified in the lower level (lower cache level, or main memory) and not loaded into cache.

While any of the above two write miss policies can be used with either write back or write through:

- **Write back caches use write allocate to capture subsequent writes to the block in cache.**
- **Write through caches usually use no-write allocate since subsequent writes still have to go to memory.**

Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

Sample Data

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

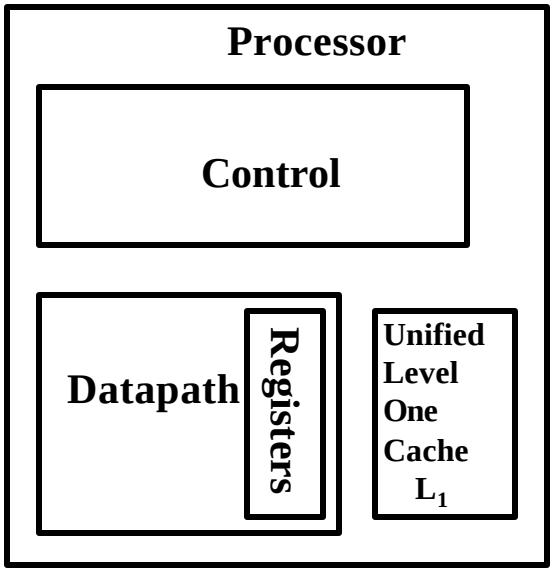
Unified vs. Separate Level 1 Cache

- **Unified Level 1 Cache (Princeton Memory Architecture).**

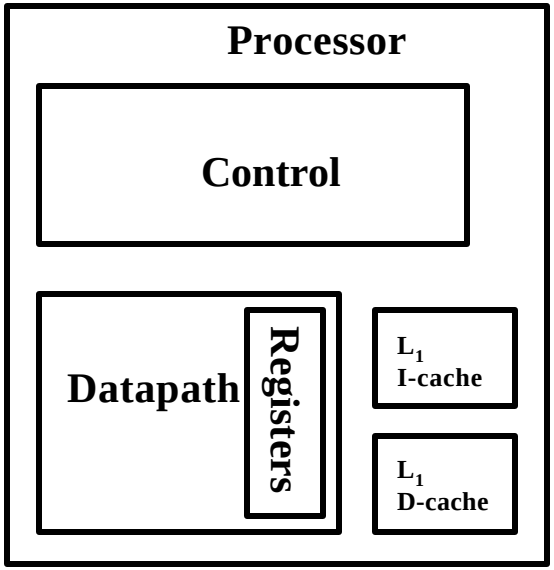
  A single level 1 cache is used for both instructions and data.

- **Separate instruction/data Level 1 caches (Harvard Memory Architecture):**

  The level 1 (L_1) cache is split into two caches, one for instructions (instruction cache, L_1 I-cache) and the other for data (data cache, L_1 D-cache).



Unified Level 1 Cache
(Princeton Memory Architecture)



Separate Level 1 Caches
(Harvard Memory Architecture)

Cache Performance:

Average Memory Access Time (AMAT), Memory Stall cycles

- **The Average Memory Access Time (AMAT): The number of cycles required to complete an average memory access request by the CPU.**
- **Memory stall cycles per memory access: The number of stall cycles added to CPU execution cycles for one memory access.**
- **For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.**
- **Memory stall cycles per average memory access = (AMAT -1)**
- **Memory stall cycles per average instruction =**

Memory stall cycles per average memory access

x Number of memory accesses per instruction

= (AMAT -1) x (1 + fraction of loads/stores)

Instruction Fetch

Cache Performance

Princeton (Unified L1) Memory Architecture

For a CPU with a single level (L1) of cache for both instructions and data (Princeton memory architecture) and no stalls for cache hits:

With ideal memory

CPU time = (CPU execution clock cycles +
Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty) +
(Writes x Write miss rate x Write miss penalty)

If write and read miss penalties are the same:

Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty

Cache Performance

Princeton Memory Architecture

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction = Mem accesses per instruction x Miss rate x Miss penalty

CPUtime = IC x ($CPI_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x ($CPI_{execution}$ + Misses per instruction x Miss penalty) x Clock cycle time

Memory Access Tree For Unified Level 1 Cache

CPU Memory Access

L_1

L1 Hit:
% = Hit Rate = H1
Access Time = 1
Stalls= H1 x 0 = 0
(No Stall)

L1 Miss:
% = (1- Hit rate) = (1-H1)
Access time = M + 1
Stall cycles per access = M x (1-H1)

AMAT = H1 x 1 + (1 -H1) x (M+ 1) = 1 + M x (1 -H1)

Stall Cycles Per Memory Access = AMAT - 1 = M x (1 -H1)

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per memory access

M = Miss Penalty
H1 = Level 1 Hit Rate
1- H1 = Level 1 Miss Rate

Cache Performance Example

- **Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache (unified).**
- **$CPI_{execution}$ = 1.1**
- **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
- **Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.**

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction =

Mem accesses per instruction x Miss rate x Miss penalty

Mem accesses per instruction = 1 + .3 = 1.3

Instruction fetch **Load/store**

Mem Stalls per instruction = 1.3 x .015 x 50 = 0.975

CPI = 1.1 + .975 = 2.075

The CPU with ideal cache (no misses) is 2.075/1.1 = 1.88 times faster

Cache Performance Example

- **Suppose for the previous example we double the clock rate to 400 MHZ, how much faster is this machine, assuming similar miss rate, instruction mix?**
- **Since memory speed is not changed, the miss penalty takes more CPU cycles:**

Miss penalty = 50 x 2 = 100 cycles.

CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

Speedup = $(CPI_{old} \times C_{old})/(CPI_{new} \times C_{new})$

= 2.075 x 2 / 3.05 = 1.36

The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

→ ***CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.***

Cache Performance

Harvard Memory Architecture

For a CPU with separate level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

CPUtime = Instruction count x CPI x Clock cycle time

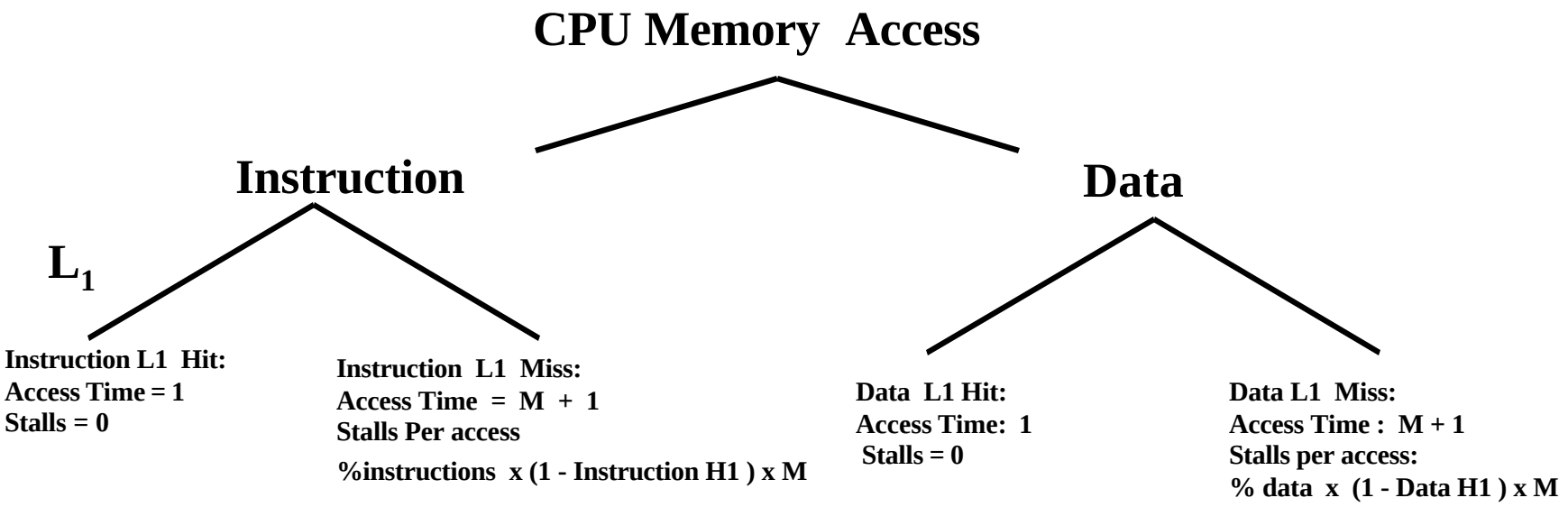
CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction =

Instruction Fetch Miss rate x Miss Penalty +
Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Access Tree
For Separate Level 1 Caches



Stall Cycles Per Memory Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M

AMAT = 1 + Stall Cycles per access

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per memory access

Cache Performance Example

- **Suppose a CPU uses separate level one (L1) caches for instructions and data (Harvard memory architecture) with different miss rates for instruction and data access:**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
  - **$CPI_{execution}$ = 1.1**
  - **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
  - **Assume a cache miss rate of 0.5% for instruction fetch and a cache data miss rate of 6%.**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes. Find the resulting CPI using this cache? How much faster is the CPU with ideal memory?**

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stall cycles per instruction = Instruction Fetch Miss rate x Miss Penalty +
Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Mem Stall cycles per instruction = 0.5/100 x 200 + 0.3 x 6/100 x 200 = 1 + 3.6 = 4.6

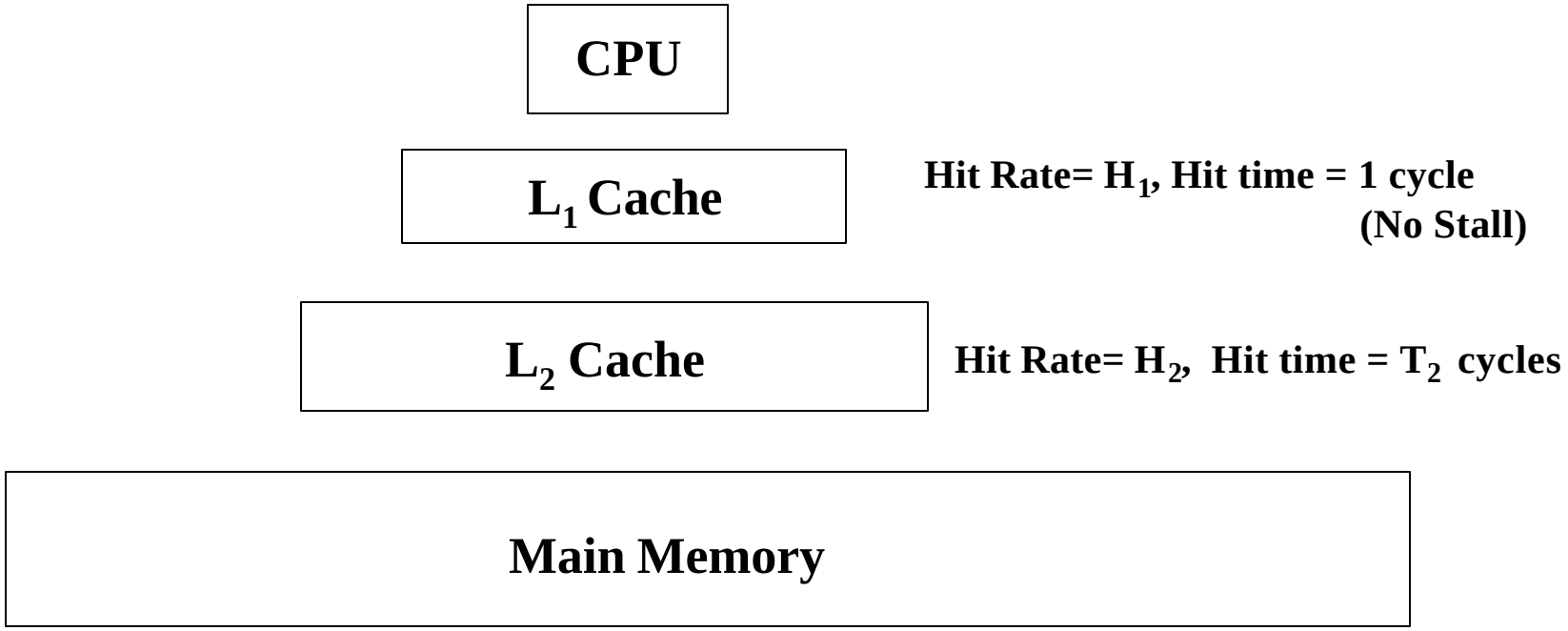
CPI = $CPI_{execution}$ + mem stalls per instruction = 1.1 + 4.6 = 5.7

The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster

With no cache the CPI would have been = 1.1 + 1.3 X 200 = 261.1 !!

Improving Cache Performance:

2 Levels of Cache: L_1, L_2



Miss Rates For Multi-Level Caches

- **Local Miss Rate: This rate is the number of misses in a cache level divided by the number of memory accesses to this level. Local Hit Rate = 1 - Local Miss Rate**
- **Global Miss Rate: The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.**
- **Since level 1 receives all CPU memory accesses, for level 1:**
  - **Local Miss Rate = Global Miss Rate = 1 - H1**
- **For level 2 since it only receives those accesses missed in 1:**
  - **Local Miss Rate = Miss rate$_{L2}$= 1- H2**
  - **Global Miss Rate = Miss rate$_{L1}$ x Miss rate$_{L2}$**

    = (1- H1) x (1 - H2)

2-Level Cache Performance

CPUtime = IC x (CPI$_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- **For a system with 2 levels of cache, assuming no penalty when found in L_1 cache:**

Stall cycles per memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2

+ Miss rate L_2 x Memory access penalty)] =

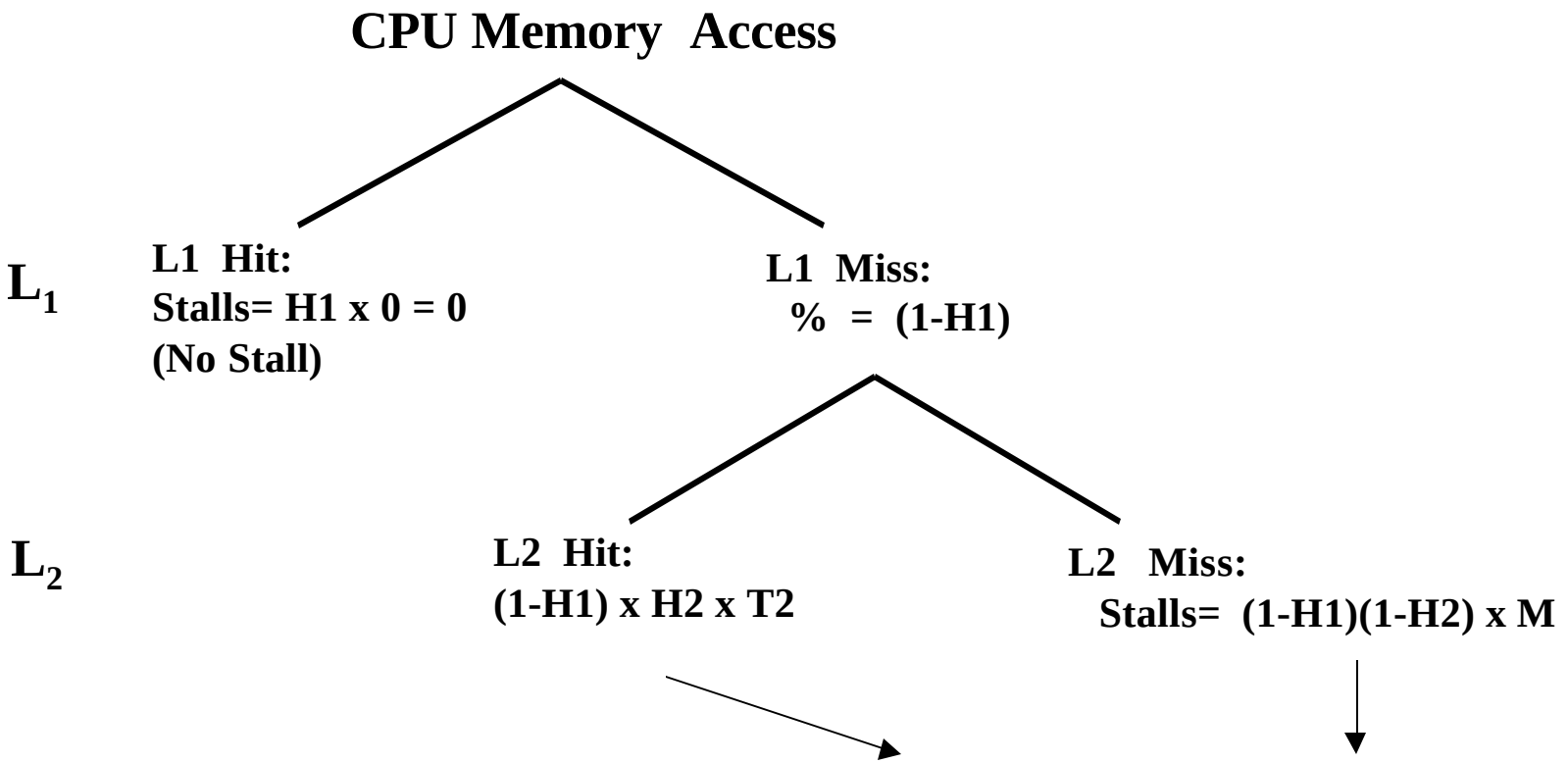
(1-H1) x H2 x T2 + (1-H1)(1-H2) x M

L1 Miss, L2 Hit

L1 Miss, L2 Miss:
Must Access Main Memory

2-Level Cache Performance Memory Access Tree

CPU Stall Cycles Per Memory Access



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Two-Level Cache Example

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHZ
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHZ with a miss rate of 5%
- L_2 cache operates at 250 MHZ with local miss rate 40%, (T_2 = 2 cycles)
- Memory access penalty, M = 100 cycles. **Find CPI.**

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single unified L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

= .05 x .6 x 2 + .05 x .4 x 100

= .06 + 2 = 2.06

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.06 x 1.3 = 2.678

CPI = 1.1 + 2.678 = 3.778

Speedup = 7.6/3.778 = 2