# Advanced Computer Architecture
# Memory Systems


Instructor

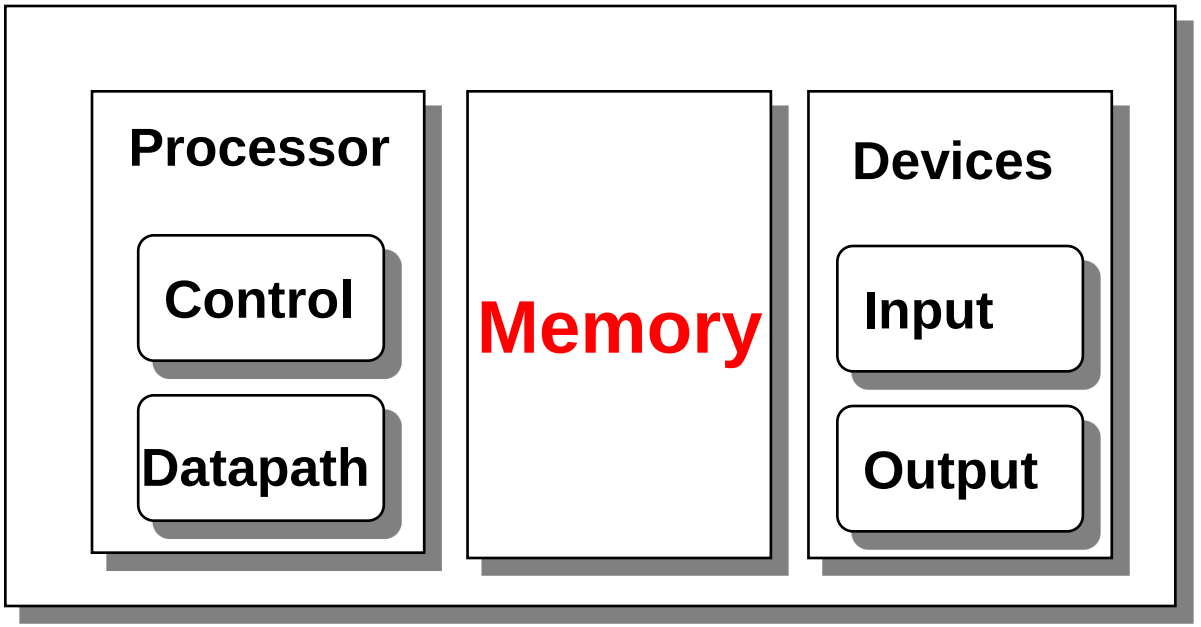
**Dr. Dinh-Duc Anh-Vu**

http://www.cse.hcmut.edu.vn/~anhvu

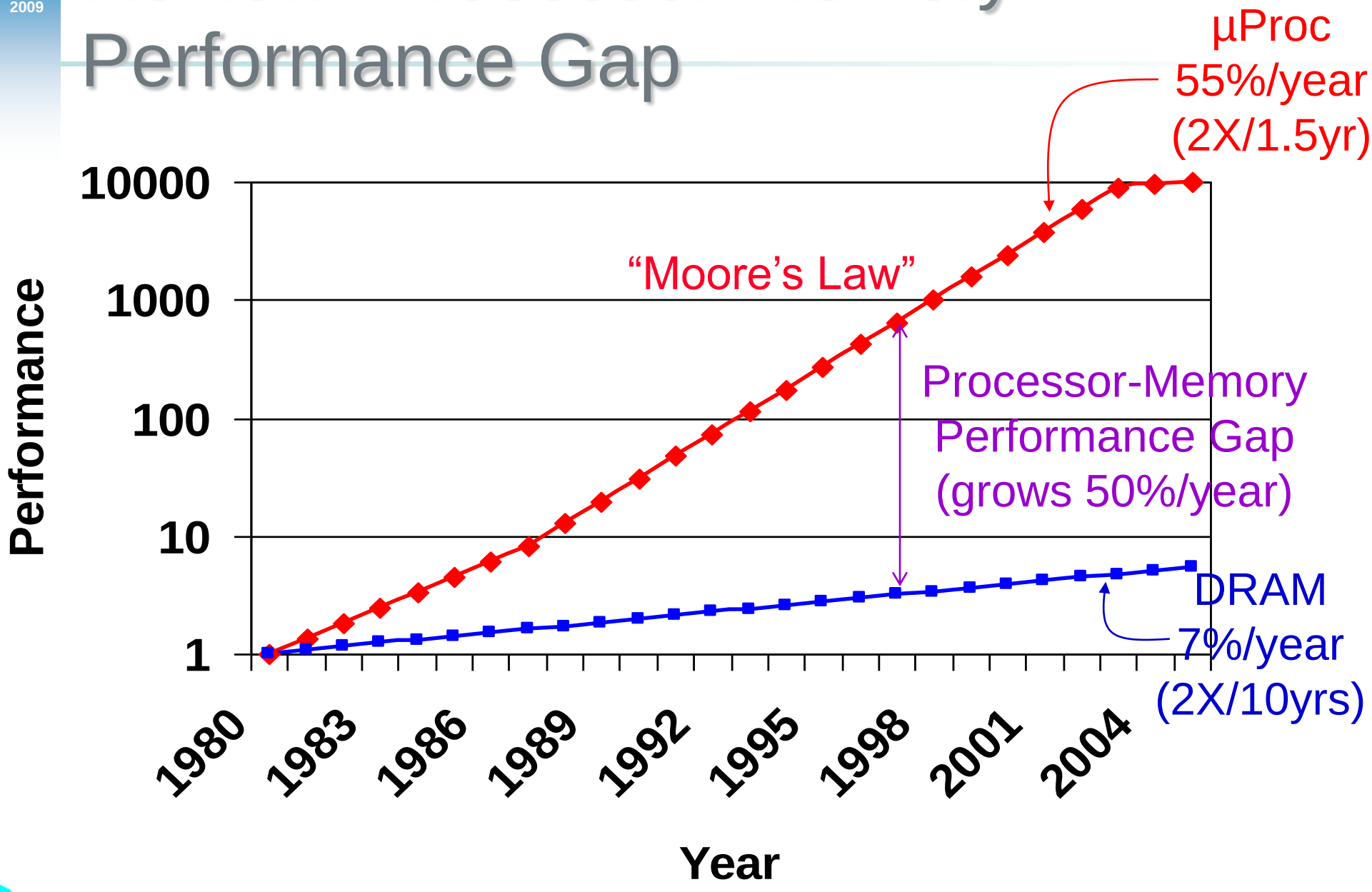
# Lecture 2 – Memory Systems

- Basic components
- Memory hierarchy
- Cache memory
- Internal Memory
- External Memory
- Virtual Memory

# Review: Major Components of a Computer

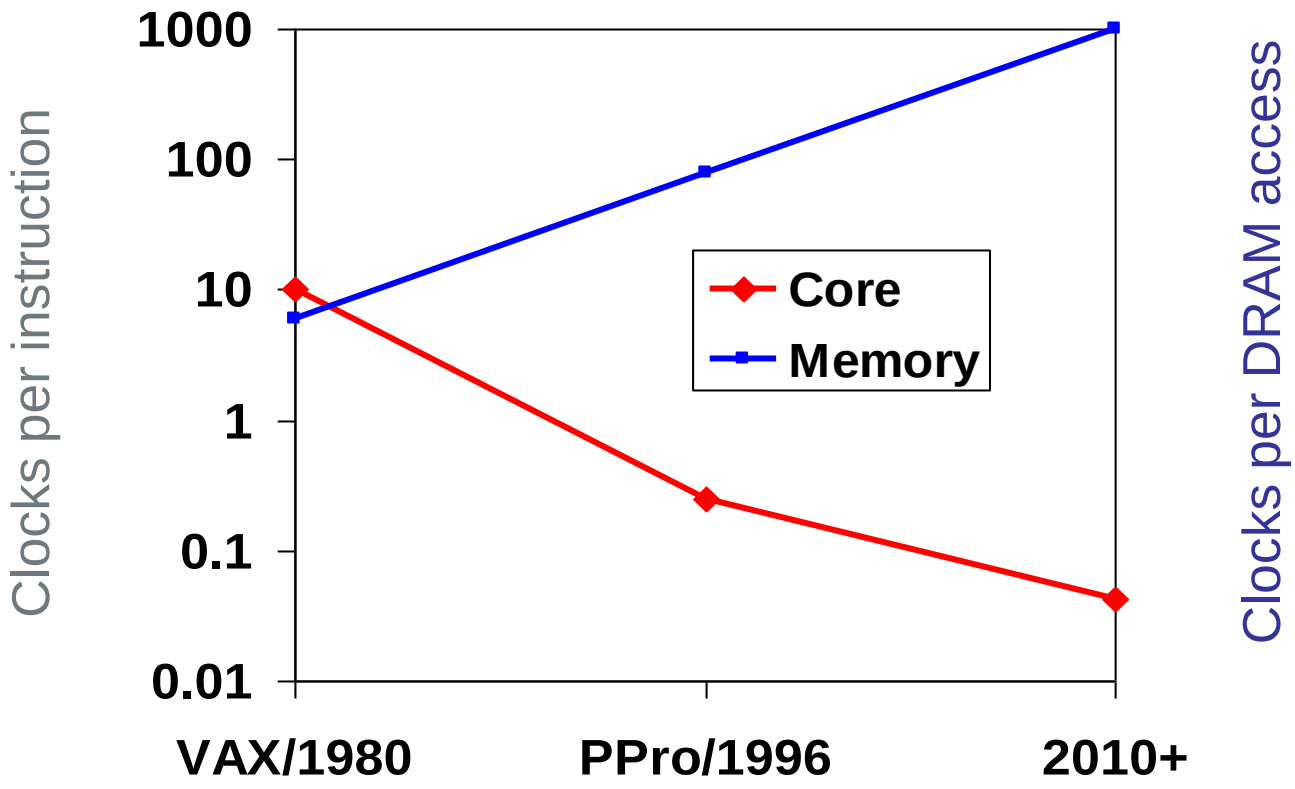


# Review: Processor-Memory Performance Gap



# Review: The "Memory Wall"

- Logic vs DRAM speed gap continues to grow



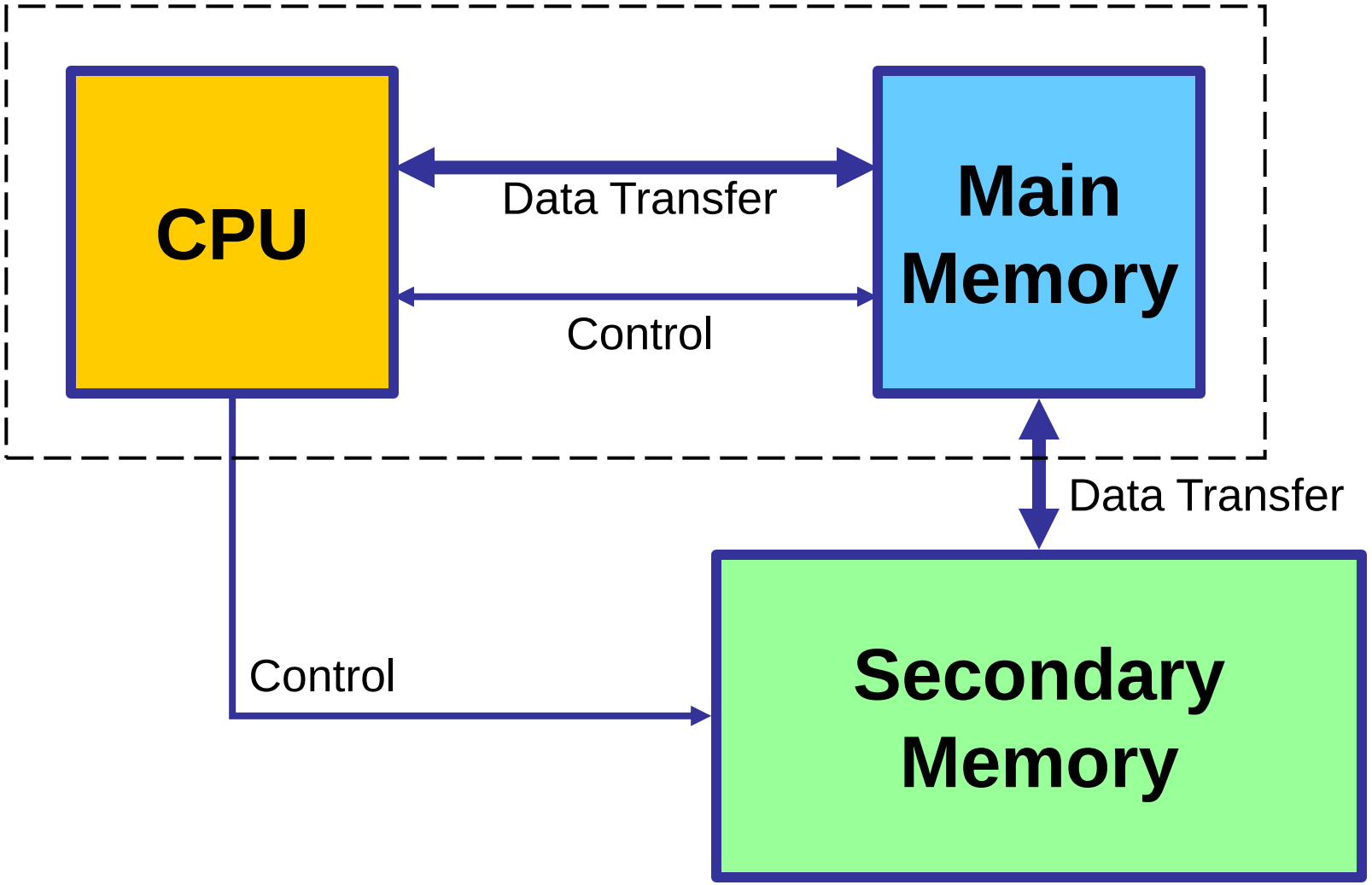
# Memory Characteristics

- Location
- Capacity
- Unit of transfer
- Access method
- Performance
- Physical type
- Physical characteristics
- Organisation

# Characteristics – Location

- Refers to whether memory is internal and external to the computer
  - Internal
    - The main memory is used to store the program and data which are currently manipulated by the CPU.
  - External
    - The secondary memory provides the long-term storage of large amounts of data and program.
    - Accessible to the processor via I/O controllers
  - Before the data and program in the secondary memory can be manipulated by the CPU, they must first be loaded into the main memory

# Internal and External Memories



# Characteristics – Capacity

- Word size
  - The natural unit of organisation
    - Equal to the number of bits used to represent an integer and
    - Equal to the instruction length
  - 8, 16 or 32 bits length
- Number of words
  - or Bytes
    - Internal memory
      - Expressed in terms of bytes or words
    - External memory
      - Expressed in terms of bytes

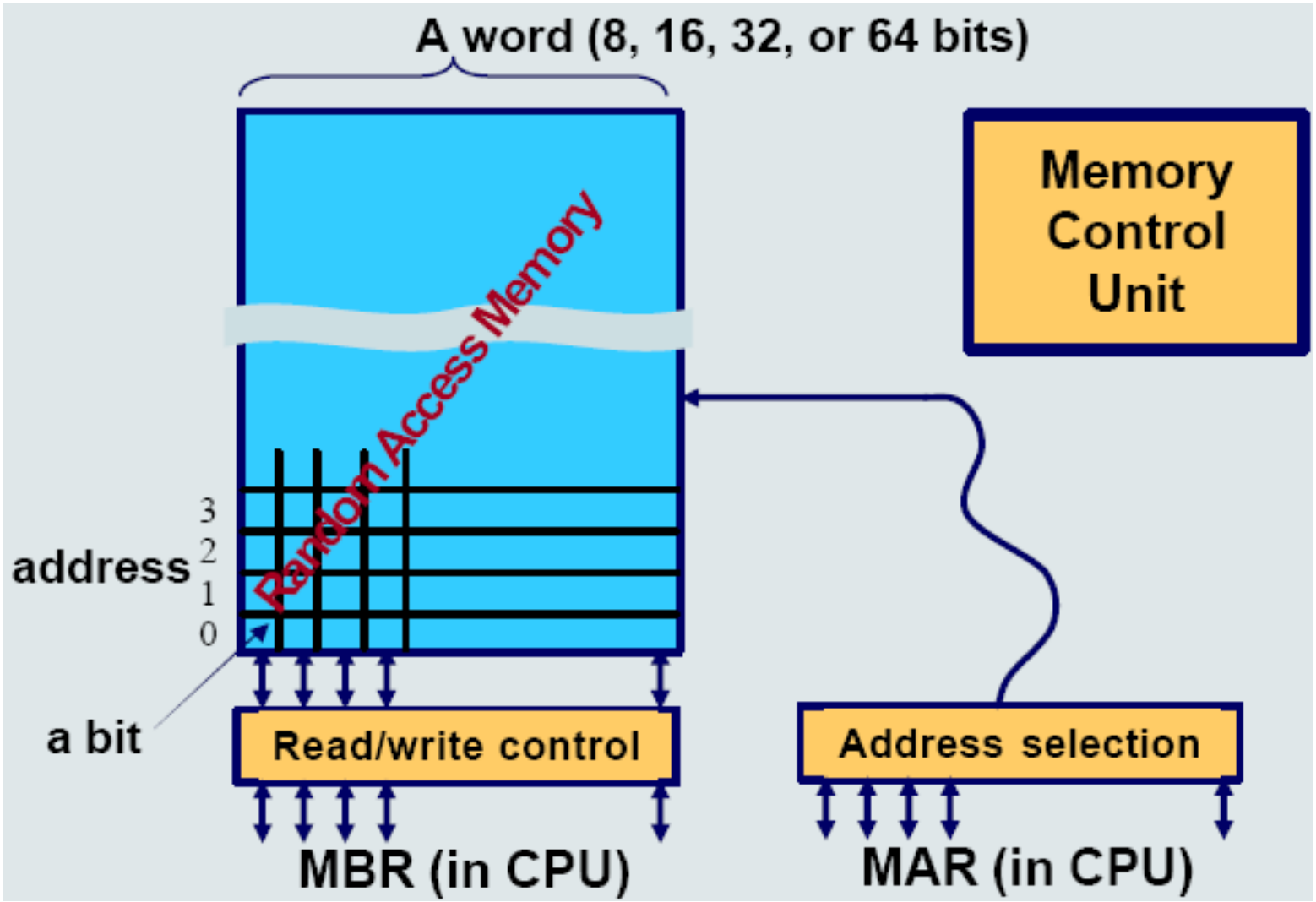
# Characteristics – Unit of Transfer

- Addressable unit
  - Smallest location which can be uniquely addressed
  - Word internally
  - Relationship between the length in bits A of an address and the number N of addressable unit: $2^A = N$
  - Cluster on M$ disks
- Unit of transfer
  - Number of bits read out of or written into memory at a time
  - Need not equal a word or an addressable unit
- Internal
  - Usually governed by data bus width
- External
  - Usually a block which is much larger than a word

# Main Memory

- The MM can be viewed as a set of storage cells, each of which can be used to store a word.
- Each cell is assigned a unique address and the addresses are numbered sequentially: 0,1,2,...
- The number of address bits determines the maximal size of the memory (e.g., 16 bits – 65K, 24 bits – 16M, 32 bits – 4G).
- There are a read/write control mechanism and an address selection mechanism, which are part of the memory control unit.

# Main Memory Model



# Characteristics – Access Methods (1)

- Sequential
  - Memory is organized into units of data (records)
  - Start at the beginning and read through in order
  - Access time depends on location of data and previous location
  - e.g. tape
- Direct
  - Individual blocks have unique address
  - Access is by jumping to vicinity plus sequential search
  - Access time depends on location and previous location
  - e.g. disk

# Characteristics – Access Methods (2)

- Random
  - Individual addresses identify locations exactly
  - Access time is independent of location or previous access
  - e.g. RAM
- Associative
  - Data is located by a comparison with contents of a portion of the store
  - Access time is independent of location or previous access
  - e.g. cache

# Characteristics – Performance

- Access time
  - For RAM, it's the time between presenting the address and getting the valid data.
  - For non-RAM, it's the time between positioning the read/write mechanism at the desired location.
- Memory cycle time
  - Time may be required for the memory to "recover" before next access
  - Cycle time is access + recovery

# Characteristics – Performance

- Transfer Rate
  - Rate at which data can be moved in or out of a memory unit
  - For RAM, transfer rate = 1/(cycle time)
  - For non-RAM,

$$T_N = T_A + \frac{N}{R}$$

Where

$T_N$ = Average time to read or write N bits

$T_A$ = Average access time

N = Number of bits

R = Transfer rate, in bps

# Characteristics – Physical Types

- Semiconductor
  - RAM
- Magnetic
  - Disk & Tape
- Optical
  - CD & DVD
- Others
  - Bubble
  - Hologram

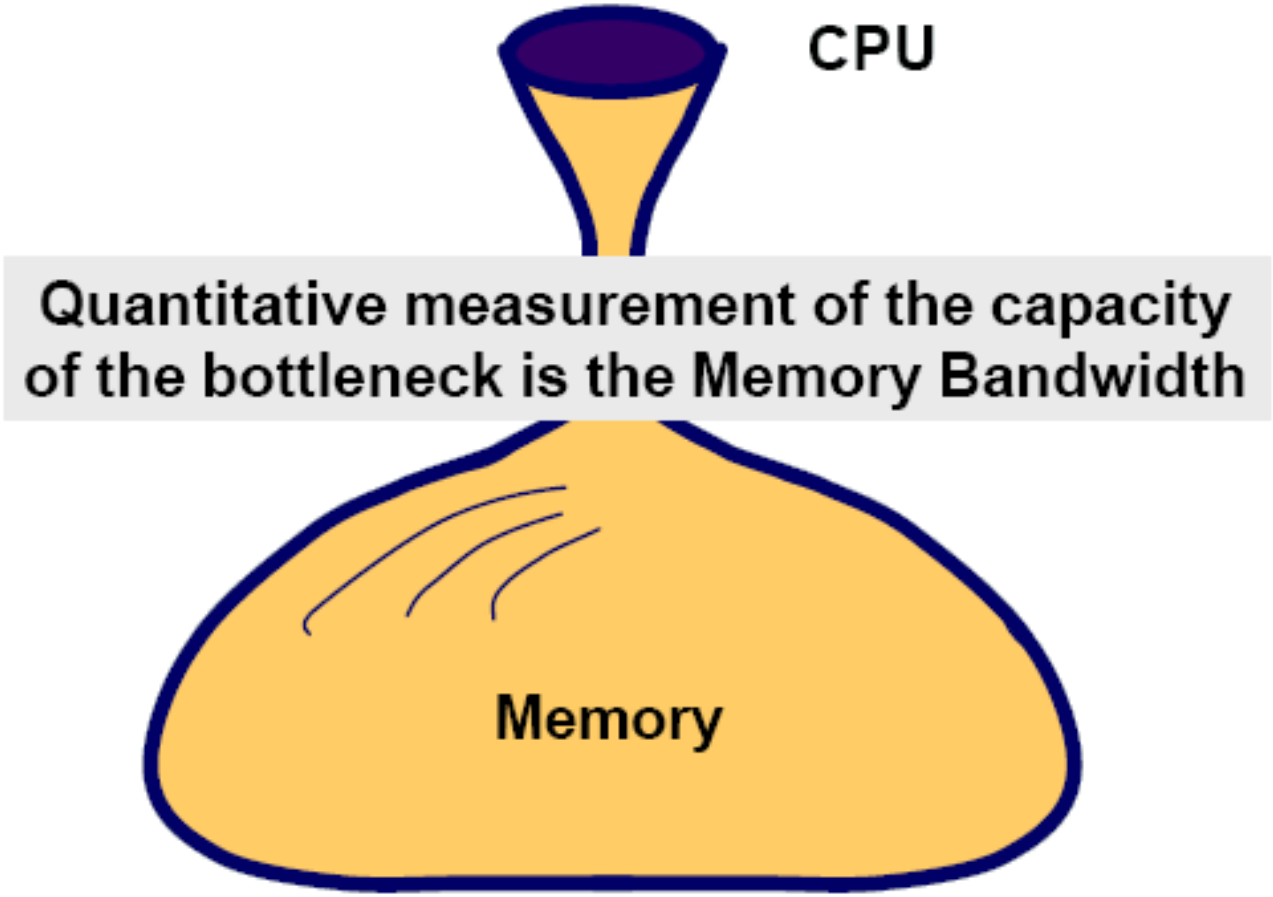
# Physical Characteristics

- Decay
- Volatility
- Erasable
- Power consumption

# Organisation

- Physical arrangement of bits into words
- Not always obvious
- e.g. interleaved

# Memory Access Bottleneck



# Memory Bandwidth

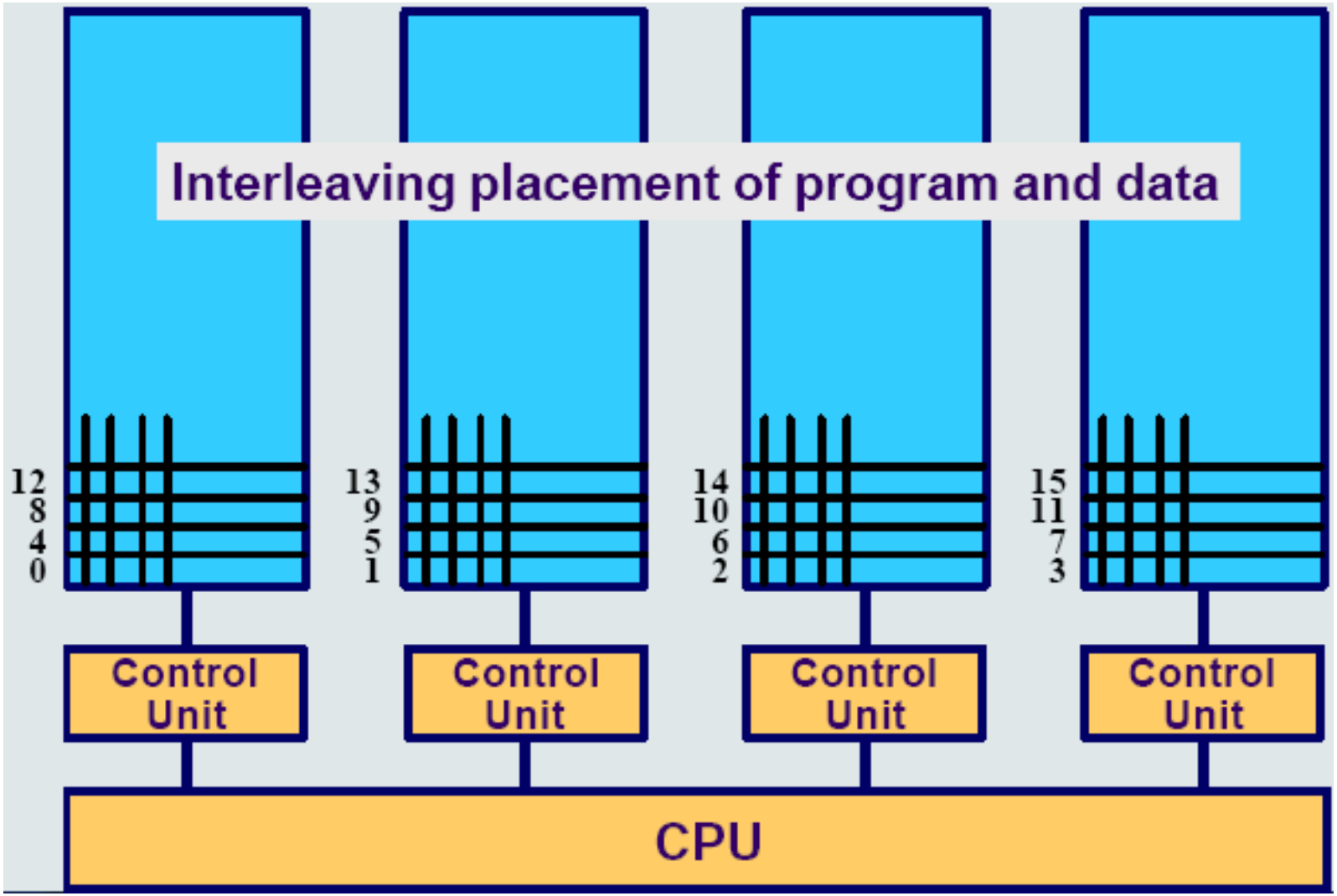
- Memory bandwidth denotes the amount of data that can be accessed from a memory per second:

$$\text{M-Bandwidth} = \frac{1}{\text{memory cycle time}} \cdot \text{amount of data per access}$$

Ex. MCT = 100 nano second and 4 bytes (a word) per access:
M-bandwidth = 40 MBps

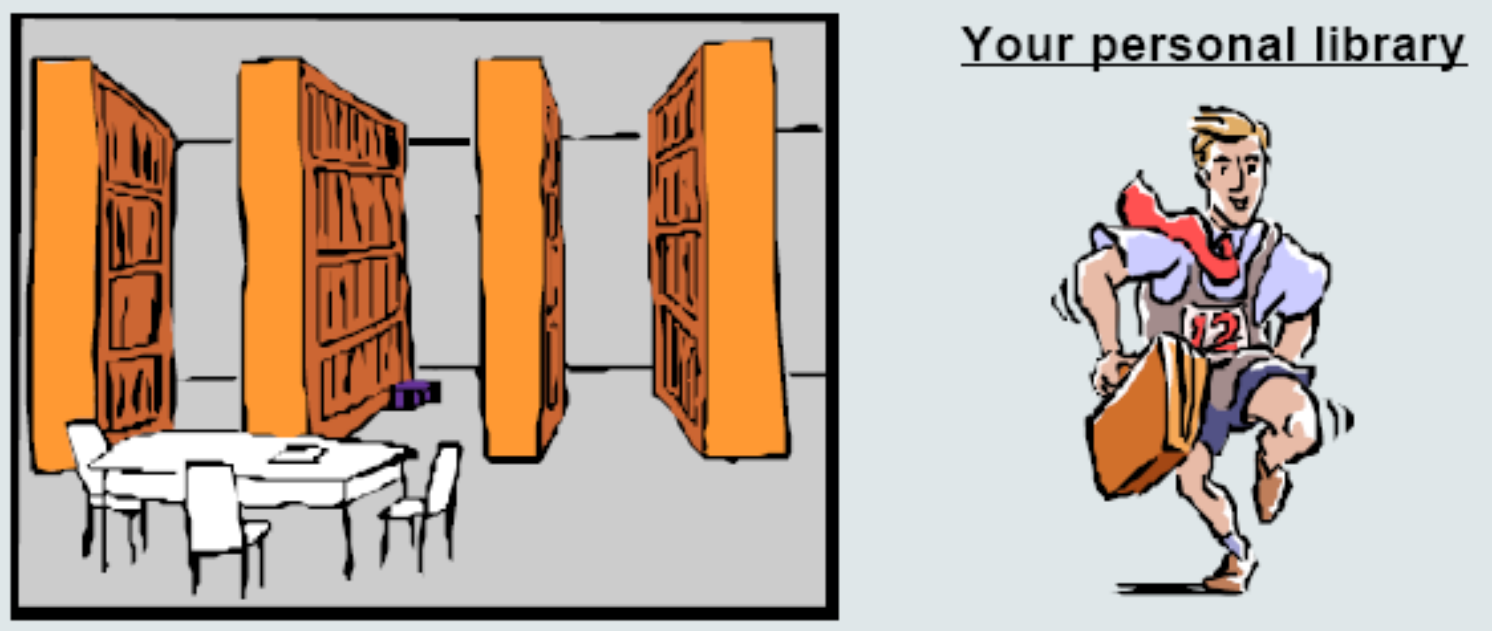
- There are two basic techniques to increase the bandwidth of a given memory:
  - Reduce the memory cycle time
    - Expensive
    - Memory size limitation
  - Divide the memory into several banks, each of which has its own control unit.

# Memory Banks



# The Bottom Line

- The most important characteristics of a memory:
  - How fast? – speed — as fast as possible;
  - How much? – size — as large as possible;
  - How expensive? – cost — reasonable price.
- They are determined by the technology used for implementation.



# Lecture 2 – Memory Systems

- Basic components
- Memory hierarchy
- Cache memory
- Internal Memory
- External Memory
- Virtual Memory

# Memory System

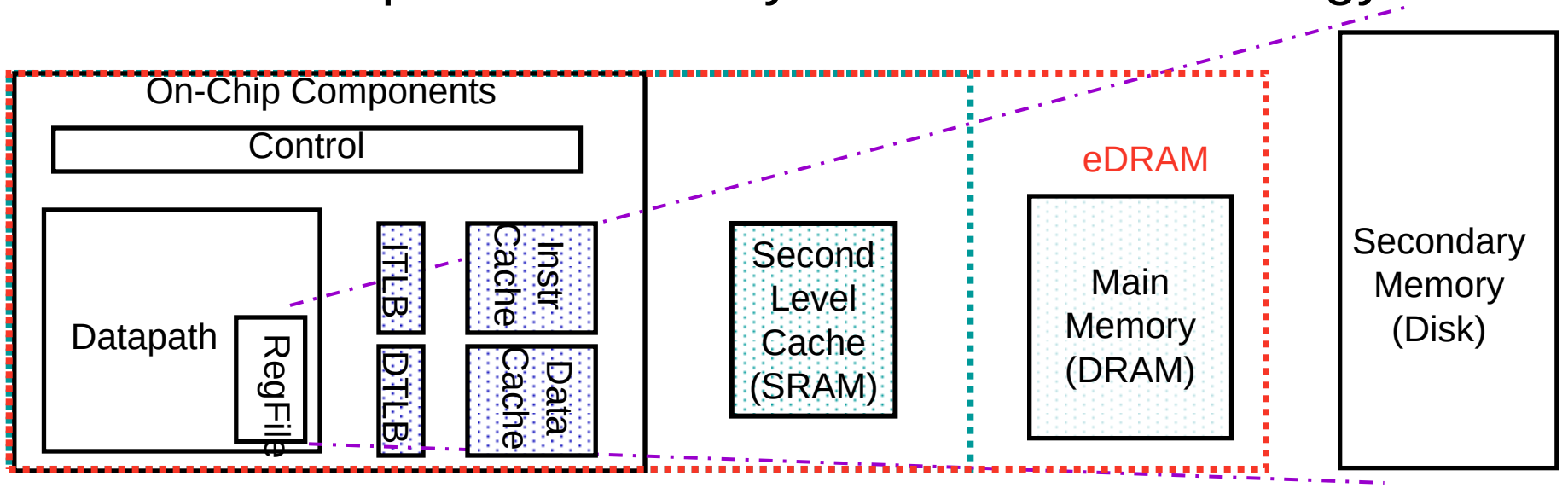
- What do we need?
  - A memory to store very large programs and to work at a speed comparable to that of the CPU.
- The reality is:
  - the larger a memory, the slower it will be;
  - the larger a memory, the smaller the cost/bit;
  - the faster the memory, the greater the cost/bit.
- A solution:
  - To build a composite memory system which combines a small and fast memory with a large and slow memory, and behaves (most of the time) like a large fast memory.
  - The two level principle above can be extended into a hierarchy of many levels.
  - The effectiveness of such a memory hierarchy is based on property of programs called the locality of reference (to be detailed later)

# Memory Hierarchy

- Registers
  - In CPU
- Internal or Main memory
  - May include one or more levels of cache
  - “RAM”
- External memory
  - Backing store

# A Typical Memory Hierarchy

- By taking advantage of the principle of locality
  - Can present the user with as much memory as is available in the cheapest technology
  - at the speed offered by the fastest technology




| | | | | | |
|---|---|---|---|---|---|
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

# Memory Hierarchy

Faster

CPU

Registers

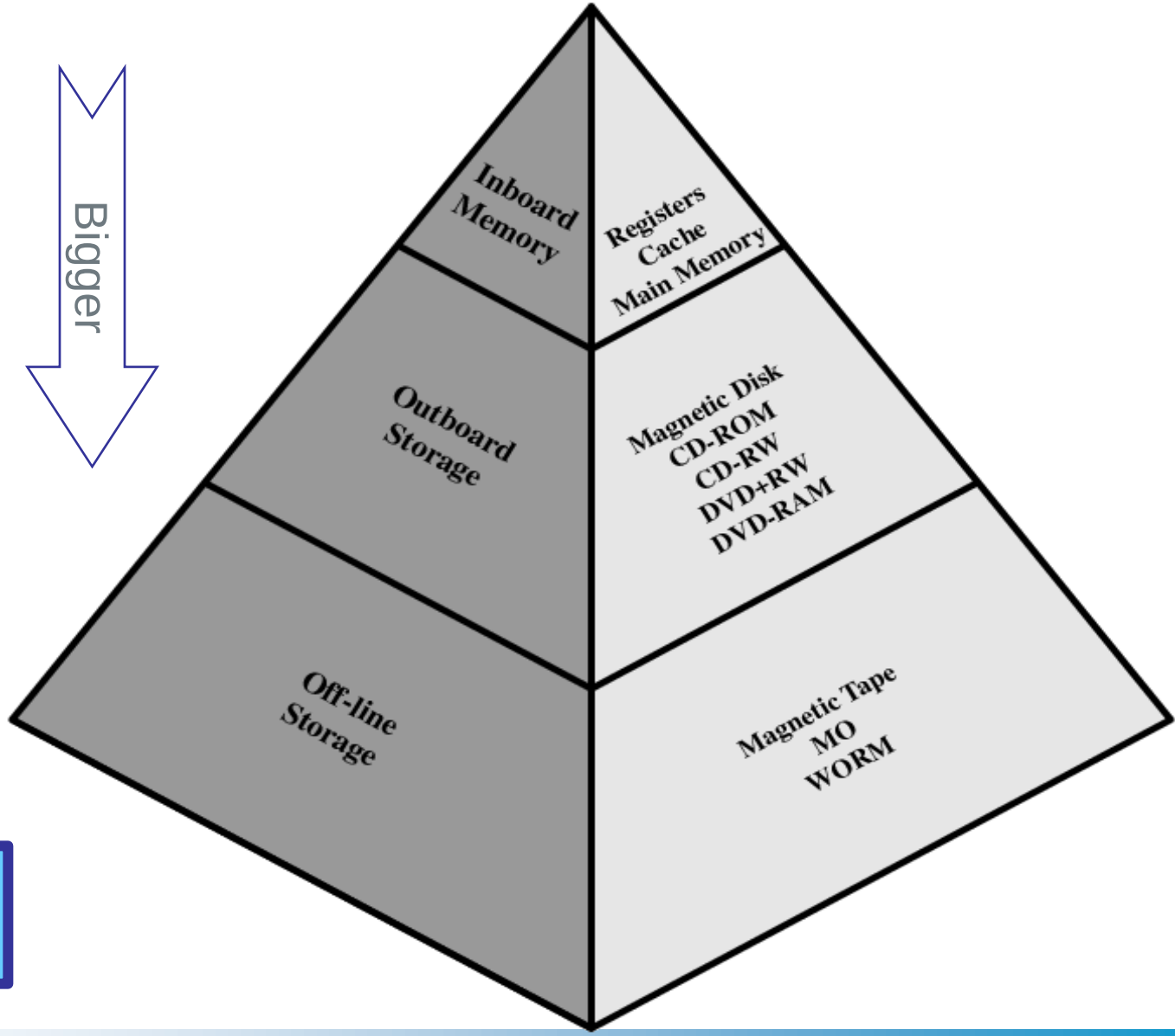
Cache

Main Memory

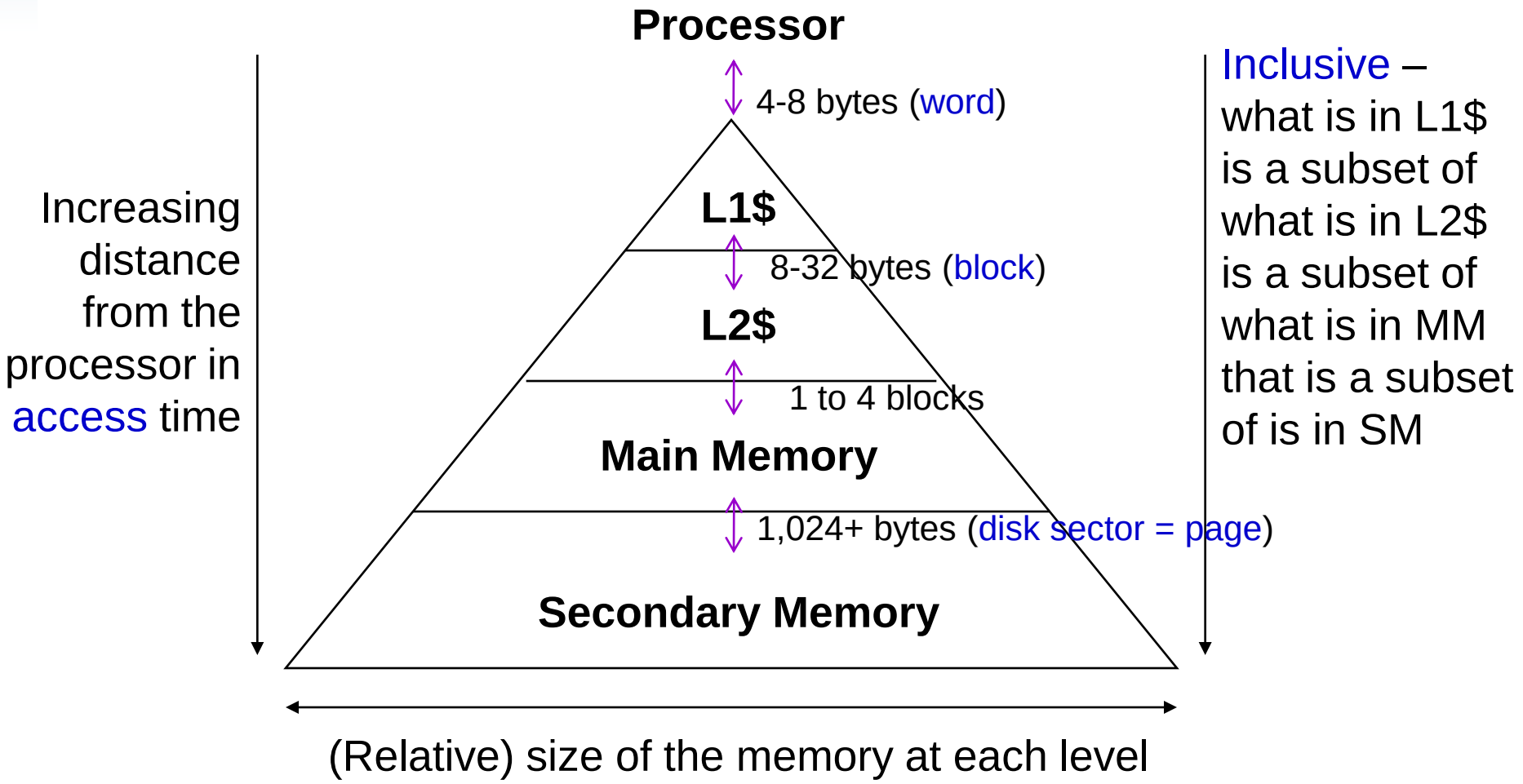
Secondary Memory of direct access type

Secondary Memory of archive type

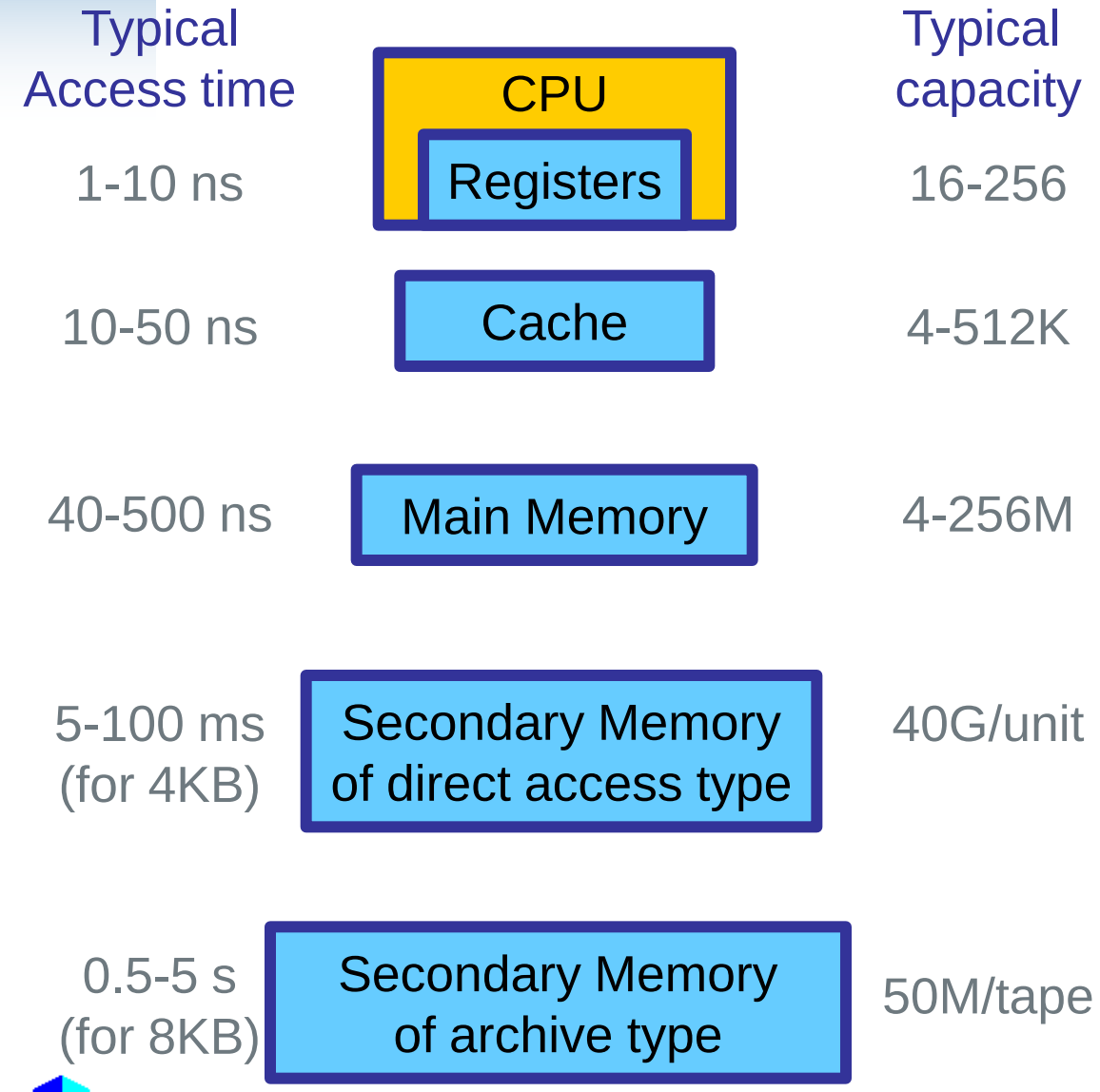
Bigger



# Characteristics of the Memory Hierarchy



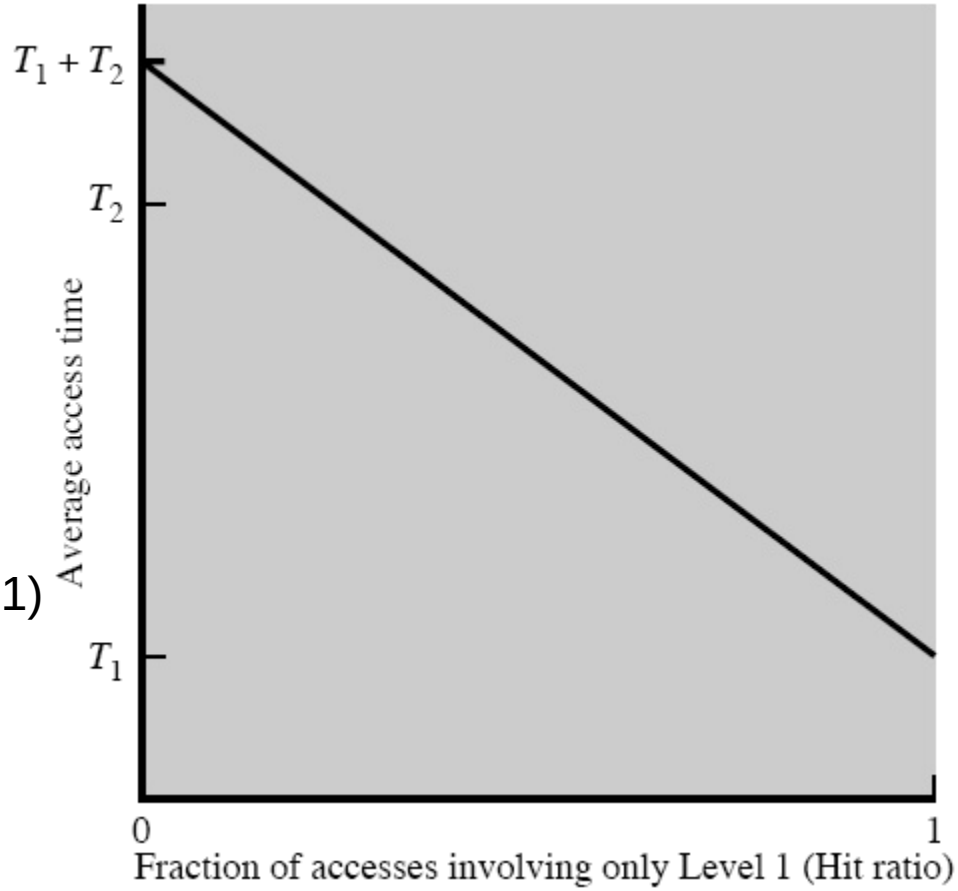
# Memory Hierarchy




- As one goes down the hierarchy, the following occur:
  - Decreasing cost/bit.
  - Increasing capacity.
  - Increasing access time.
  - Decreasing frequency of access by the CPU.

# Performance of a simple 2-level memory

- Level 1's access time $T_1$ = 0.01µs
- Level 2's access time $T_2$ = 0.1µs
- The word is transferred to level 1 before accessed by the processor
  - A **hit** occurs if the accessed word is found in the faster memory (level 1)
  - A **miss** occurs if the accessed word is not found in the faster memory
- Time required for the processor to determine whether the word is in level 1 or level 2 is ignored



# Layered Memory Performance

Average Access Time ≈ $P_{hit}$ x $T_{cache\_access}$ + (1 – $P_{hit}$) x ($T_{mm\_access}$ + $T_{cache\_access}$) x Block_size + $T_{checking}$

where

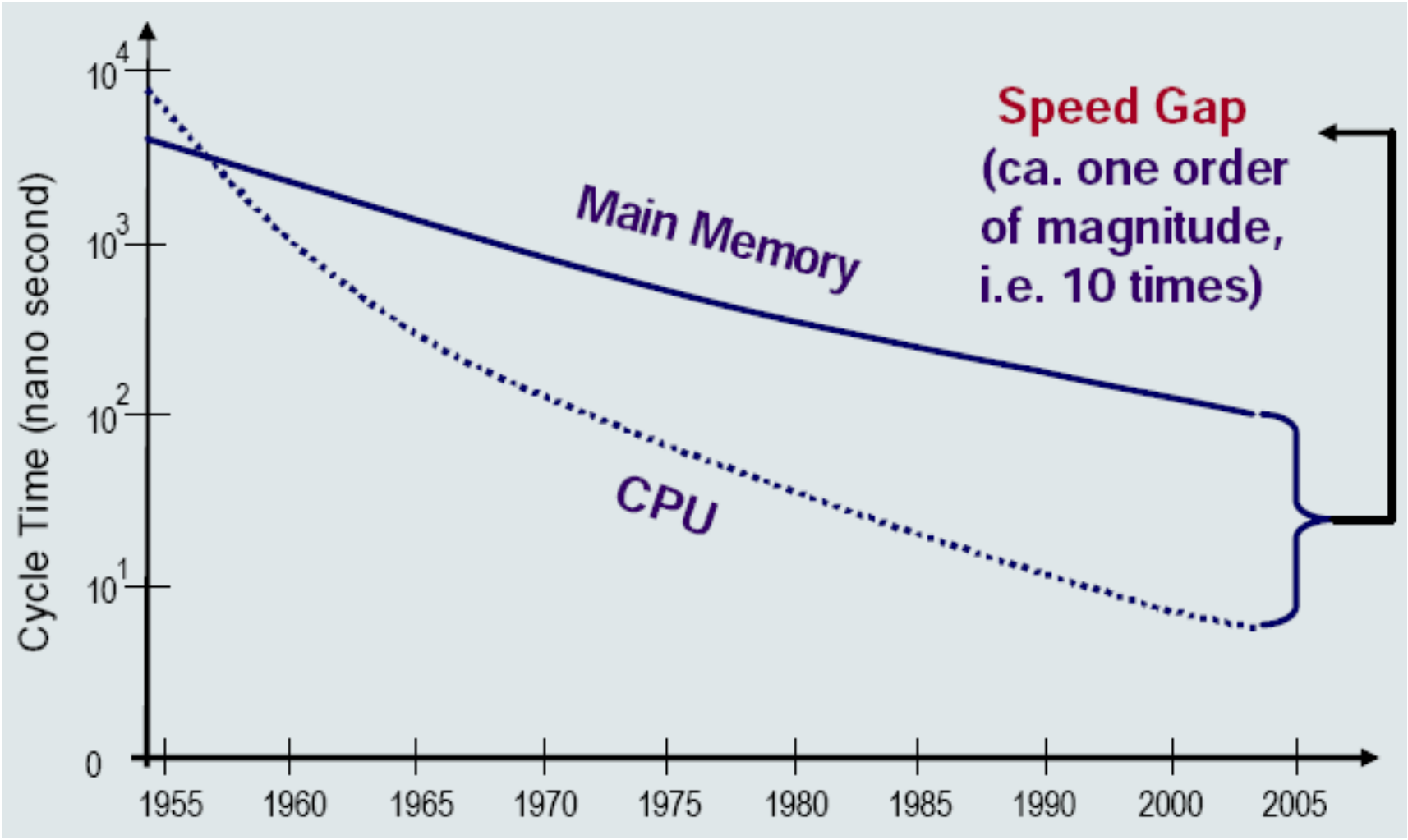
- $P_{hit}$ = the probability of cache hit, cache hit ratio;
- $T_{cache\_access}$ = cache access time;
- $T_{mm\_access}$ = main memory access time;
- Block_size = number of words in a cache block;
- $T_{checking}$ = the time needed to check for cache hit or miss.

- Ex. A computer has 8MB MM with 100 ns access time, 8KB cache with 10 ns access time, BS=4, and $T_{checking}$ = 0, $P_{hit}$ = 0.97, AAT will be 22.9 ns.

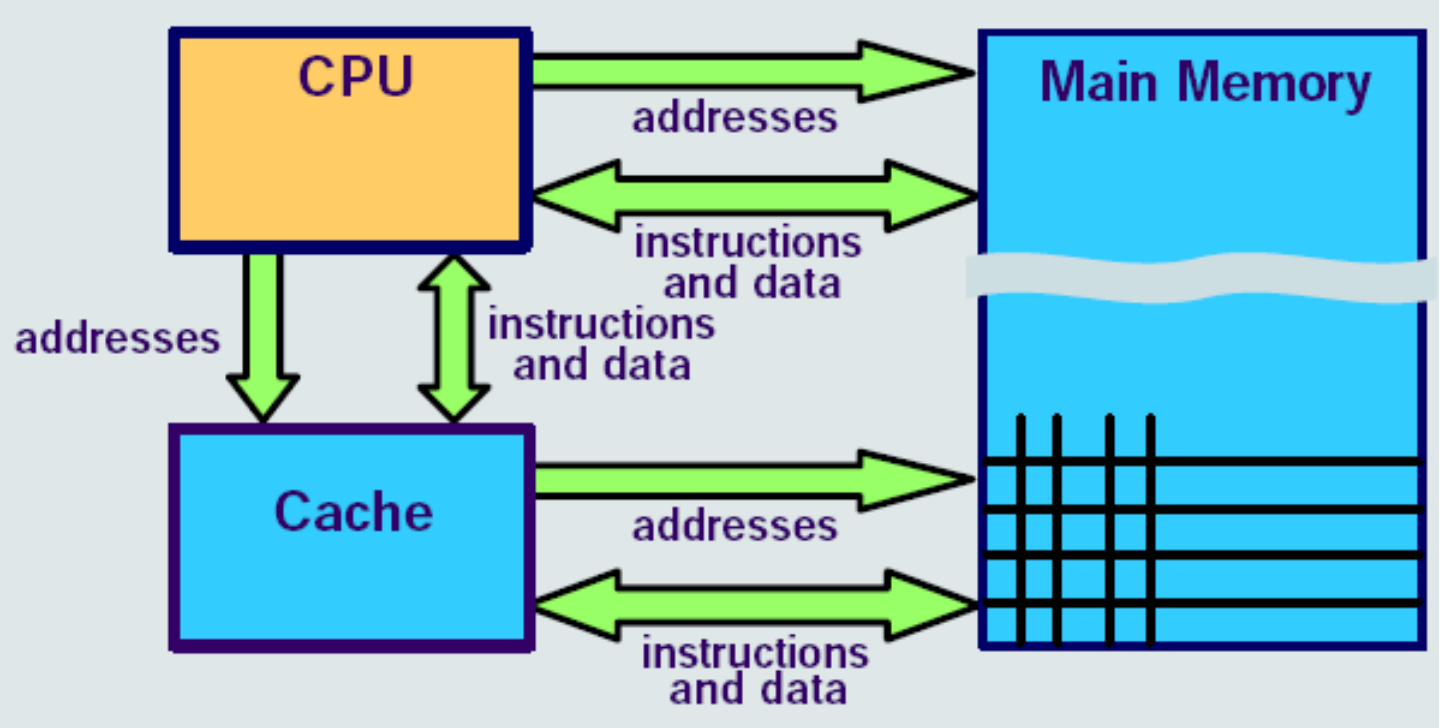
# Lecture 2 – Memory Systems

- Basic components
- Memory hierarchy
- Cache memory
- Internal Memory
- External Memory
- Virtual Memory

# Mismatch of CPU and MM Speeds



# Cache Memory



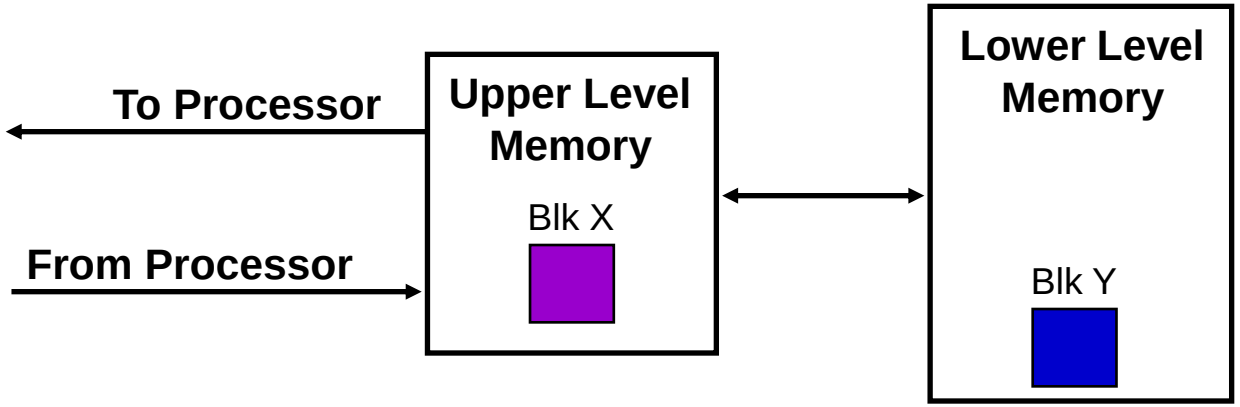

- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology
- A cache is a very fast memory which is put between the main memory and the CPU, and used to hold segments of program and data of the main memory.
  - May be located on CPU chip or module

# Locality of Reference

- **Temporal locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
- **Spatial locality** (Locality in Space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.
- This access pattern is referred as locality of reference principle, which is a intrinsic features of the von Neumann architecture:
  - Sequential instruction storage.
  - Loops and iterations (e.g., subroutine calls).
  - Sequential data storage (e.g., array).

# The Memory Hierarchy: Terminology

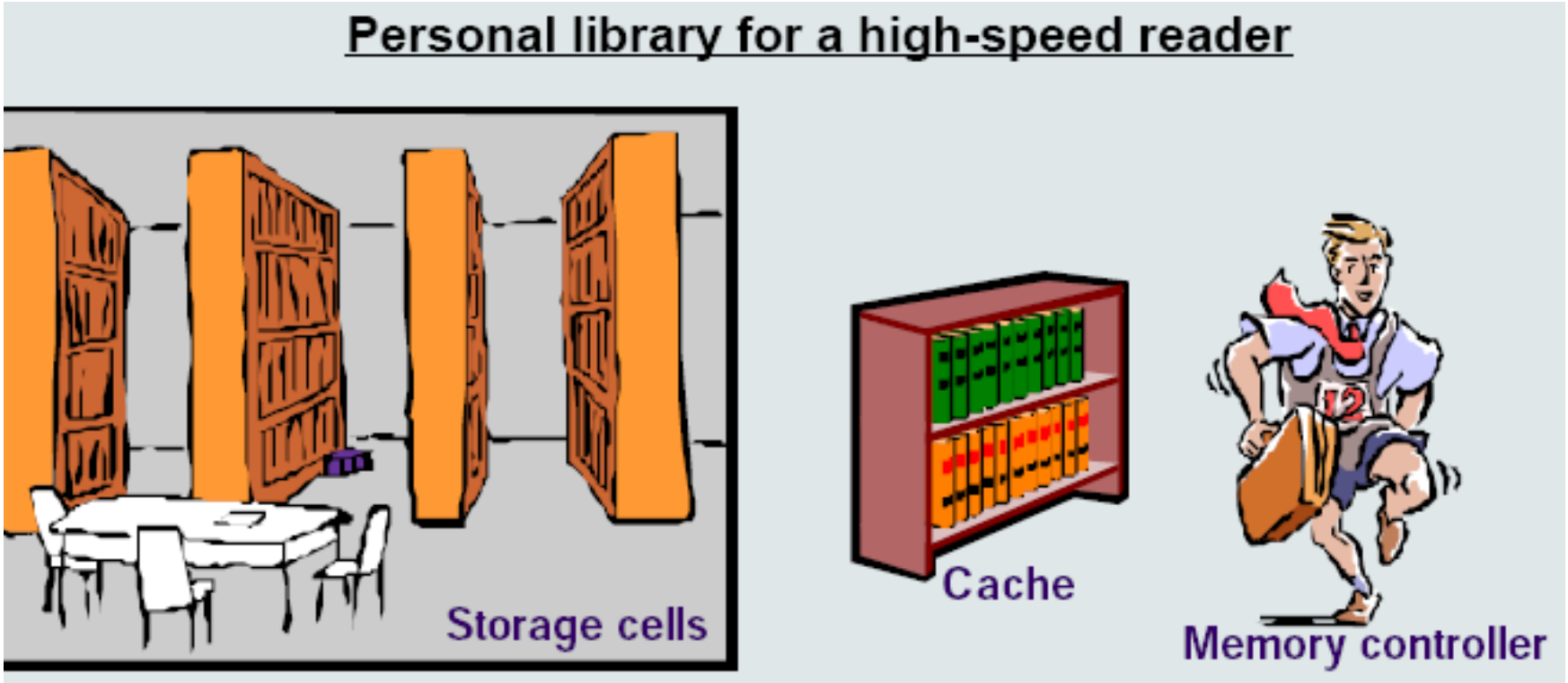
- Hit: data is in some block in the upper level (Blk X)
  - Hit Rate: the fraction of memory accesses found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss



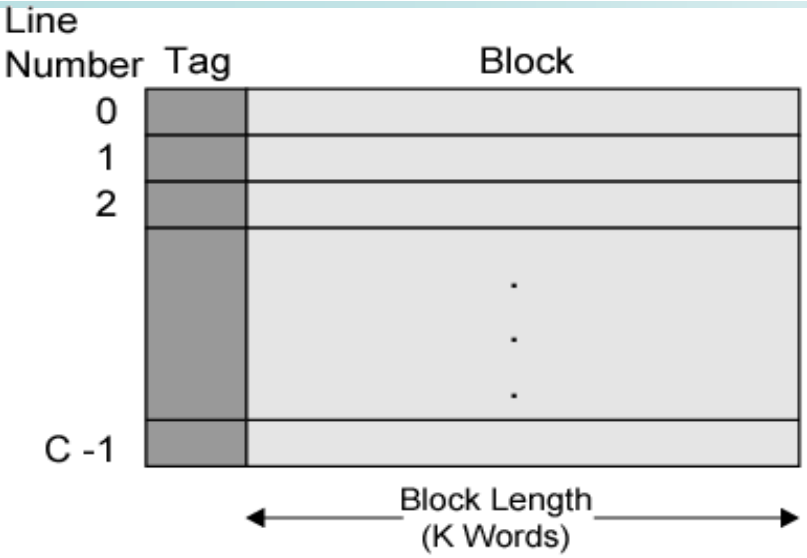

- Miss: data is not in the upper level so needs to be retrieve from a block in the lower level (Blk Y)
  - Miss Rate = 1 – (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level
    + Time to deliver the block the processor
  - Hit Time << Miss Penalty

# Cache Memory Model

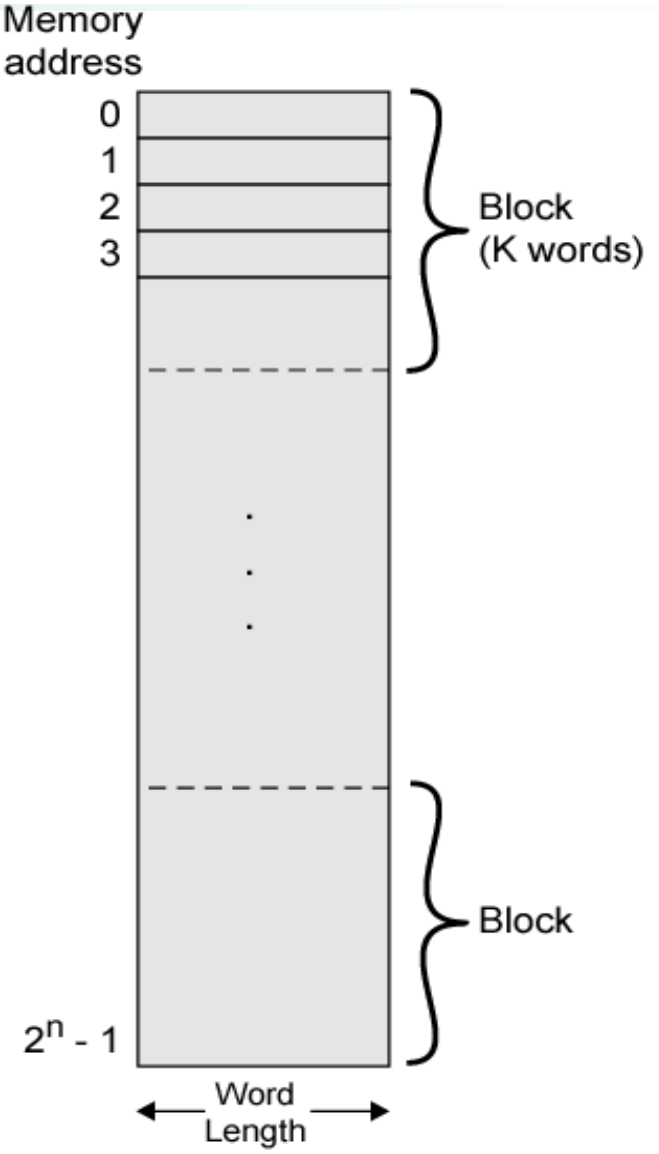
- A computer is a “predictable and iterative reader,” therefore high cache hit ratio, e.g, 96%, is achievable, even with a relatively small cache.



# Cache/Main Memory Structure




(a) Cache

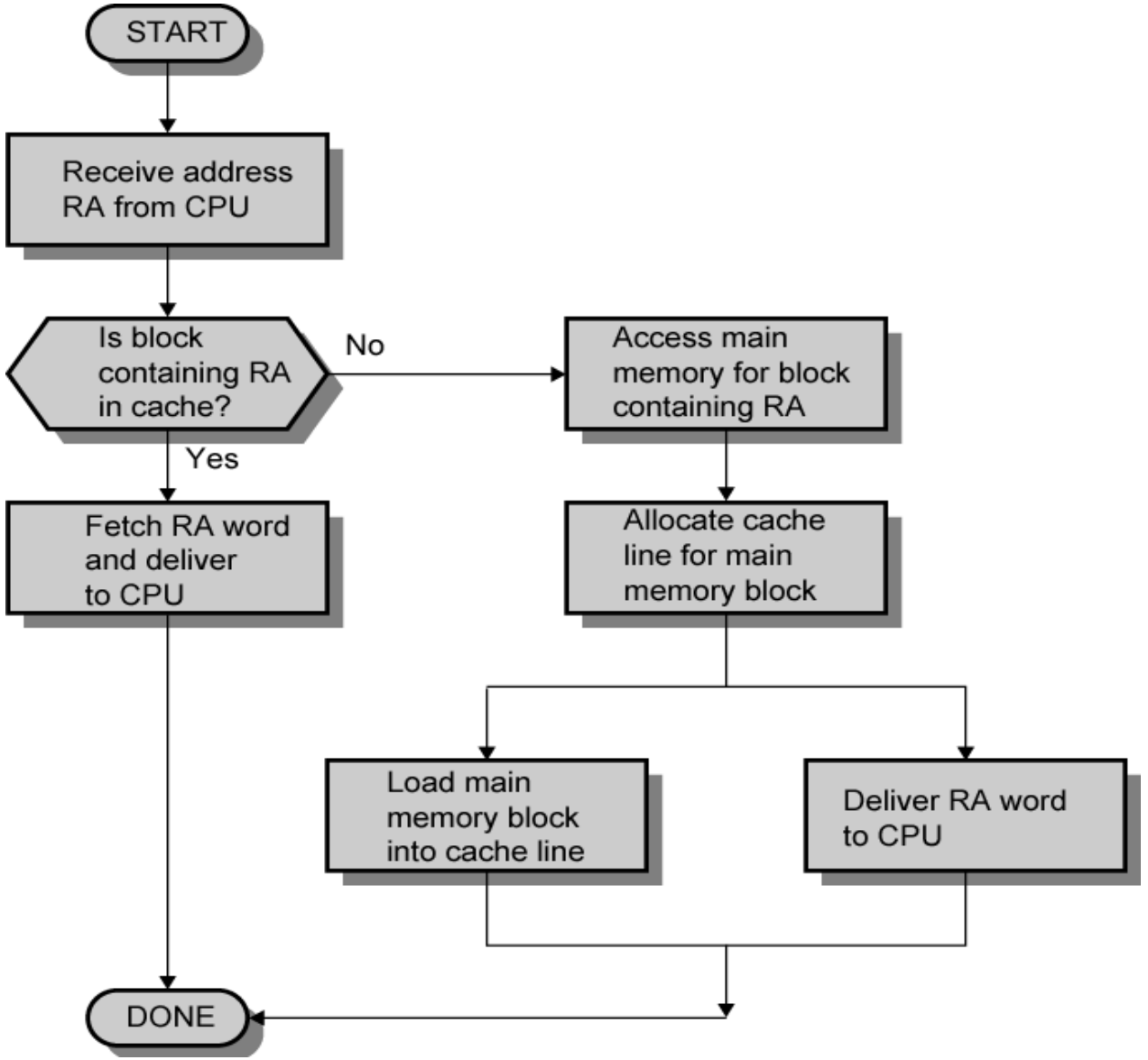



(b) Main memory

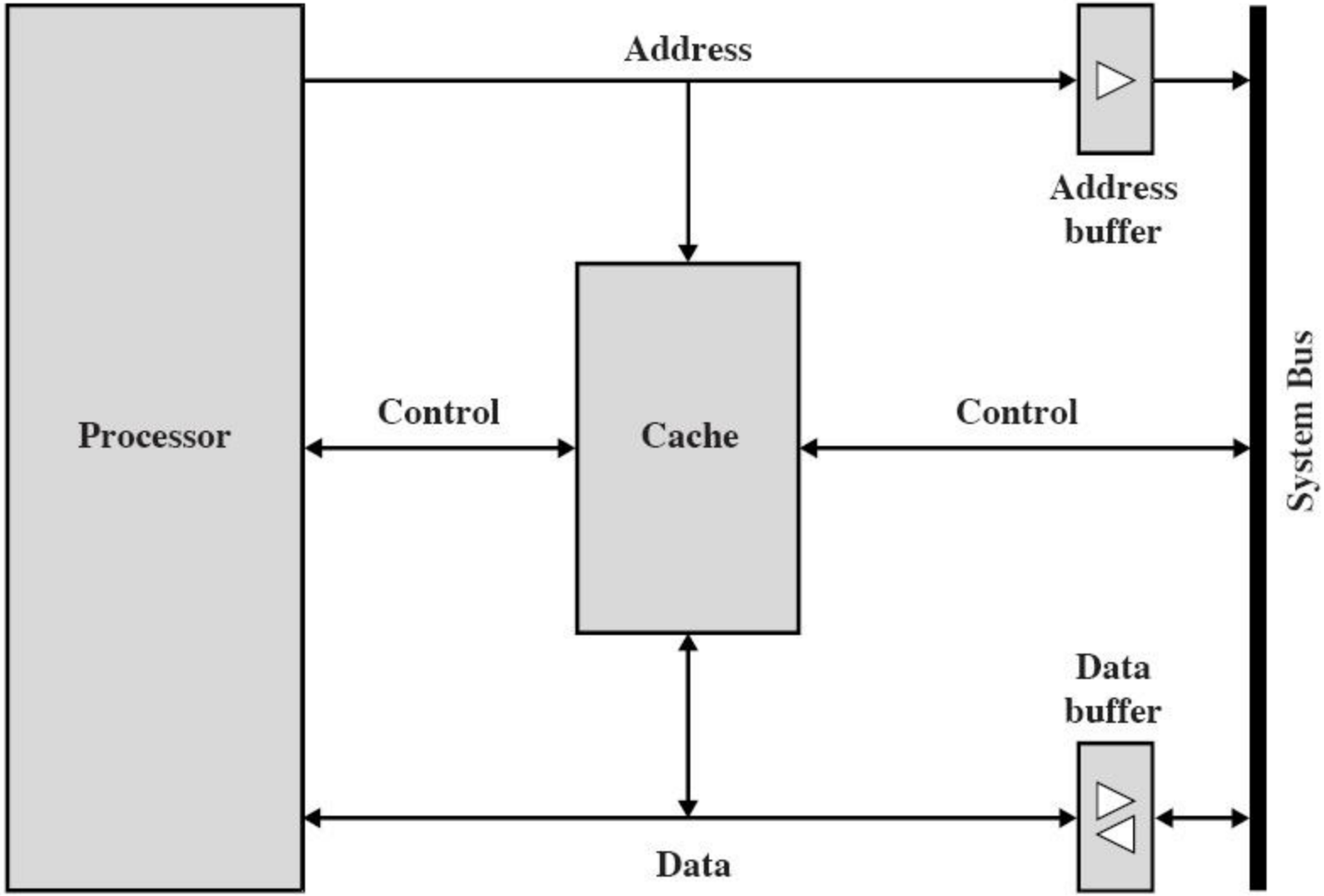
# Cache Operation – Overview

- 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

# Cache Read Operation – Flowchart



# Cache Organization



# Cache Design

- The size and nature of the copied block must be carefully designed, as well as the algorithm to decide which block to be removed from the cache when it is full:
  - Cache block size.
  - Total cache size.
  - Mapping function.
  - Replacement method.
  - Write policy.
  - Numbers of caches:
    - Single, two-level, or three-level cache.
    - Unified vs. split cache.

# Size does matter

- Small enough so that the overall average cost per bit is close to that of main memory alone
- Large enough so that the overall average access time is close to that of the cache alone
- Cost
  - More cache is expensive
- Speed
  - More cache is faster (up to a point)
  - Checking cache for data takes time

# Comparison of Cache Sizes

| Processor | Type | Year of Introduction | L1 cache | L2 cache | L3 cache |
|---|---|---|---|---|---|
| IBM 360/85 | Mainframe | 1968 | 16 to 32 KB | — | — |
| PDP-11/70 | Minicomputer | 1975 | 1 KB | — | — |
| VAX 11/780 | Minicomputer | 1978 | 16 KB | — | — |
| IBM 3033 | Mainframe | 1978 | 64 KB | — | — |
| IBM 3090 | Mainframe | 1985 | 128 to 256 KB | — | — |
| Intel 80486 | PC | 1989 | 8 KB | — | — |
| Pentium | PC | 1993 | 8 KB/8 KB | 256 to 512 KB | — |
| PowerPC 601 | PC | 1993 | 32 KB | — | — |
| PowerPC 620 | PC | 1996 | 32 KB/32 KB | — | — |
| PowerPC G4 | PC/server | 1999 | 32 KB/32 KB | 256 KB to 1 MB | 2 MB |
| IBM S/390 G4 | Mainframe | 1997 | 32 KB | 256 KB | 2 MB |
| IBM S/390 G6 | Mainframe | 1999 | 256 KB | 8 MB | — |
| Pentium 4 | PC/server | 2000 | 8 KB/8 KB | 256 KB | — |
| IBM SP | High-end server/ supercomputer | 2000 | 64 KB/32 KB | 8 MB | — |
| CRAY MTA | Supercomputer | 2000 | 8 KB | 2 MB | — |
| Itanium | PC/server | 2001 | 16 KB/16 KB | 96 KB | 4 MB |
| SGI Origin 2001 | High-end server | 2001 | 32 KB/32 KB | 4 MB | — |
| Itanium 2 | PC/server | 2002 | 32 KB | 256 KB | 6 MB |
| IBM POWER5 | High-end server | 2003 | 64 KB | 1.9 MB | 36 MB |
| CRAY XD-1 | Supercomputer | 2004 | 64 KB/64 KB | 1MB | — |

# Cache

- Two questions to answer (in hardware):
  - Q1: How do we know if a data item is in the cache?
  - Q2: If it is, how do we find it?

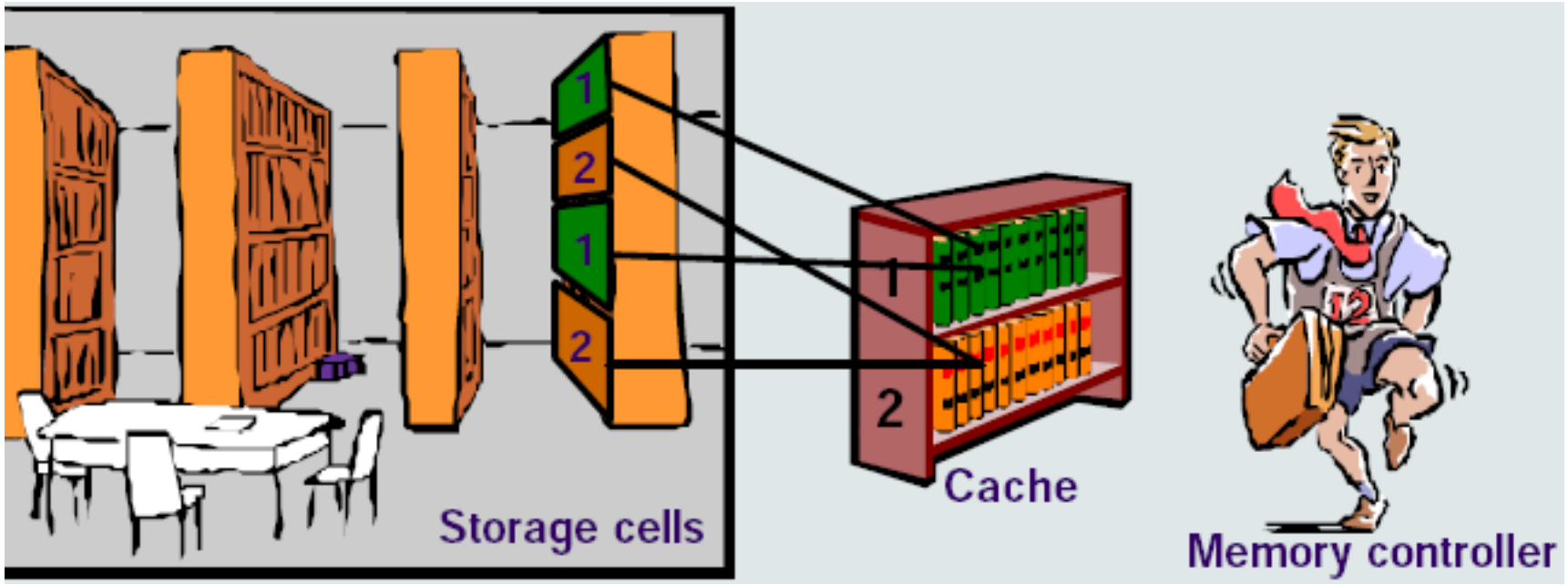
- Mapping Function

# Mapping Function

- Needed for mapping MM blocks into cache lines and for determining which MM block currently occupies a cache line
- The choice of the mapping function dictates how the cache is organized
- Example in use
  - Cache of 64kByte
  - Data are transferred in blocks of 4 bytes each
  - Cache block of 4 bytes
    - i.e. cache is 16k ($2^{14}$) lines of 4 bytes
  - 16MBytes main memory
  - 24 bit address
    - ($2^{24}$=16M)
    - 4M blocks of 4 bytes each

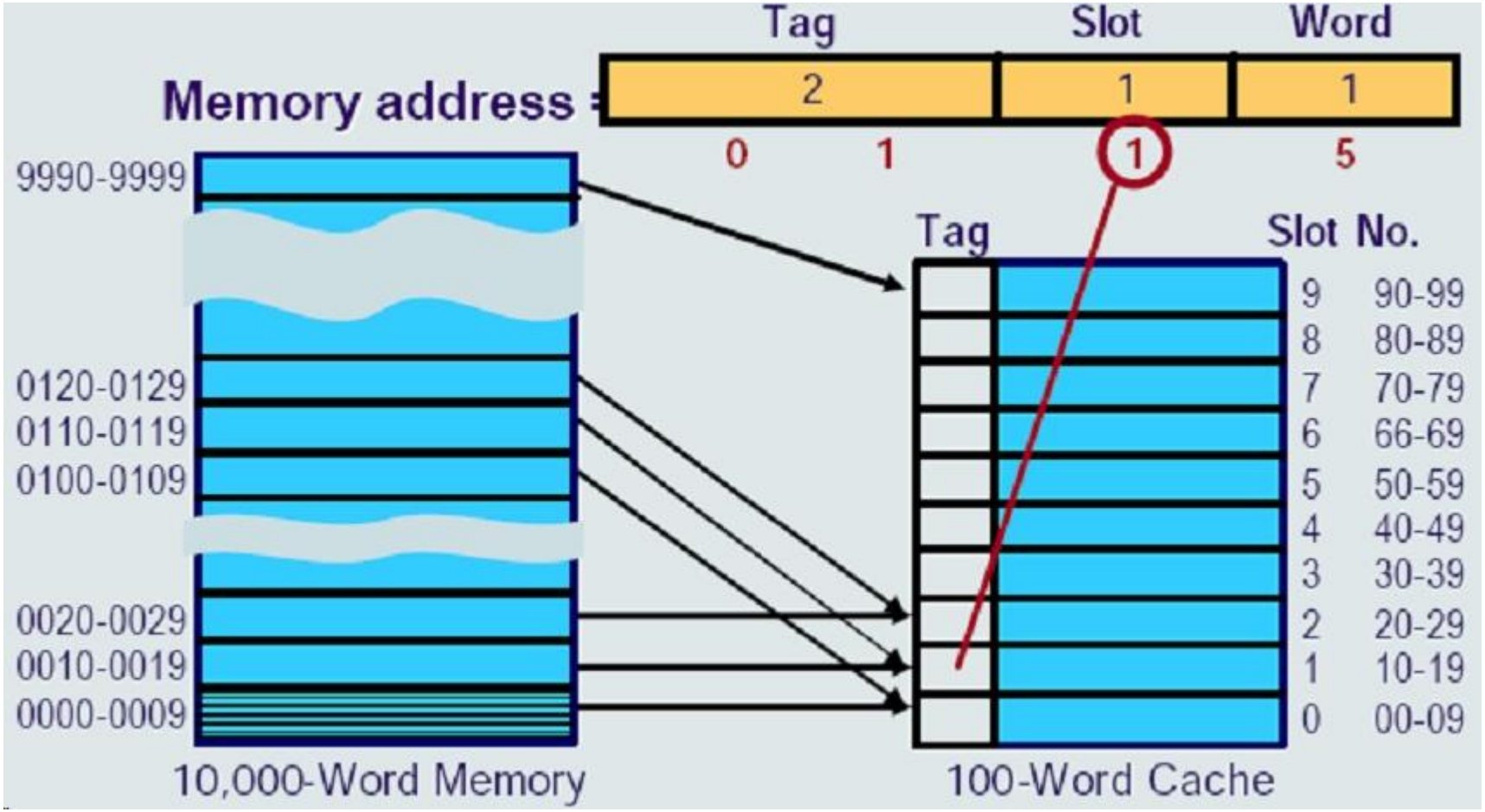
# Direct Mapping Cache

- Direct mapping – Each block of the main memory is mapped into a fixed cache slot.
  - So lots of items at the lower level must share locations in the upper level
- *i = j modulo m*
  - i = cache line number
  - j = MM block number
  - m = number of lines in the cache

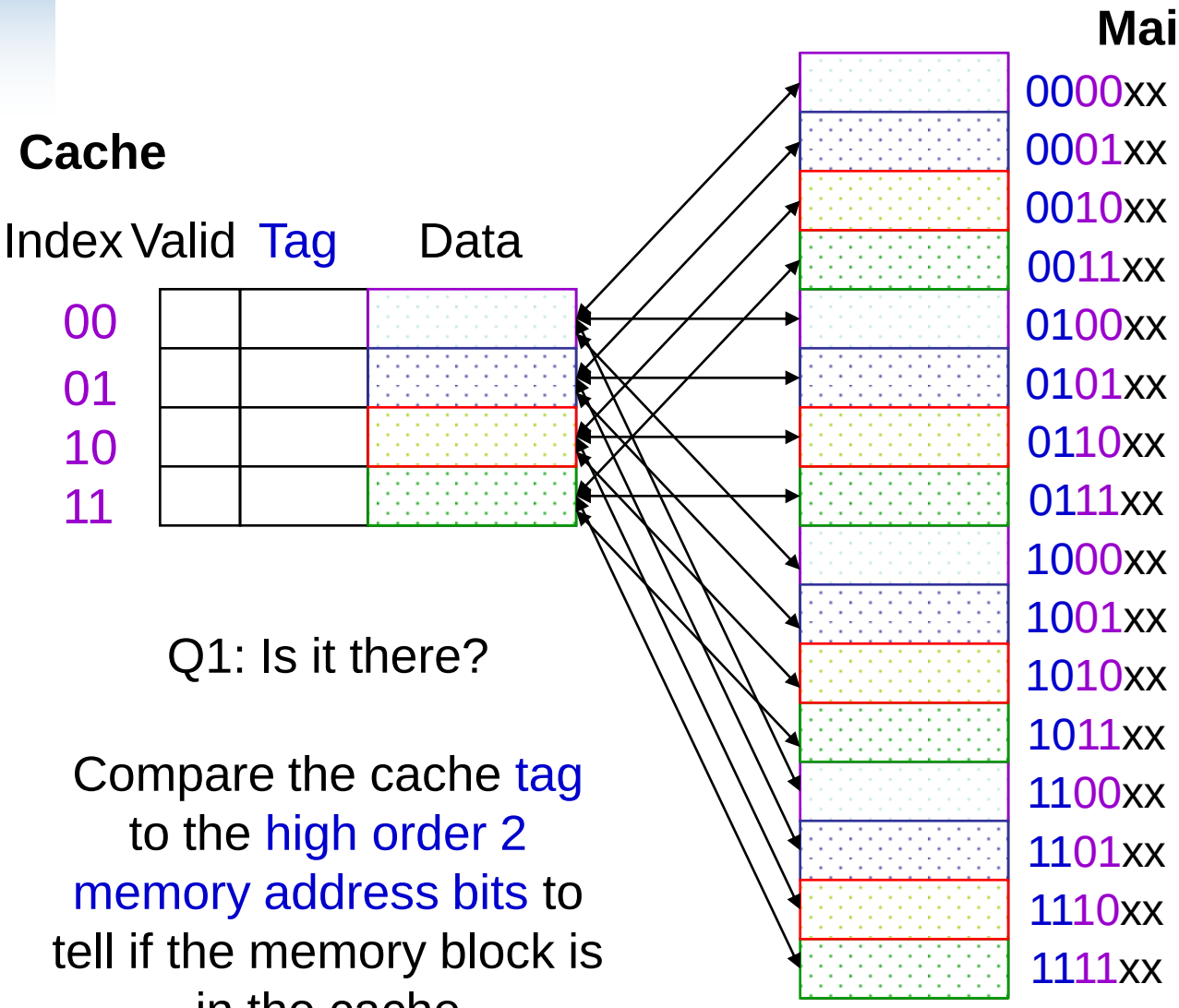


# Direct Mapping Cache Example

- We have a 10,000-word MM and a 100-word cache. 10 memory cells are grouped into a block.



# Caching: A Simple First Example




Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

# Direct Mapped Cache

- Consider the main memory word reference string

**0 1 2 3 4 3 4 15**

Start with an empty cache - all blocks initially marked as not valid

8 requests, 6 misses

**0** miss

| | |
|---|---|
| 00 | Mem(0) |
| | |
| | |
| | |

**1** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| | |
| | |

**2** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| | |

**3** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** miss

| | |
|---|---|
| ~~00~~ (01) | ~~Mem(0)~~ (4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**3** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**15** miss

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| ~~00~~ (11) | ~~Mem(3)~~ (15) |

# Direct Mapping Cache Line Table

| Cache line | Main Memory blocks held |
|---|---|
| 0 | 0, m, 2m, 3m,…, $2^s - m$ |
| 1 | 1,m+1, 2m+1,…, $2^s - m+1$ |
| … | … |
| m – 1 | m – 1, 2m – 1,3m – 1,…, $2^s - 1$ |

# Direct Mapping Cache (Cont'd)

- Address is in two parts
- Least significant w bits identify unique word within a block of main memory
- Most significant s bits specify one memory block ($2^s$ blocks of main memory)
- The MSBs are split into a cache line field r and a tag of s-r (most significant)

# Direct Mapping Address Structure

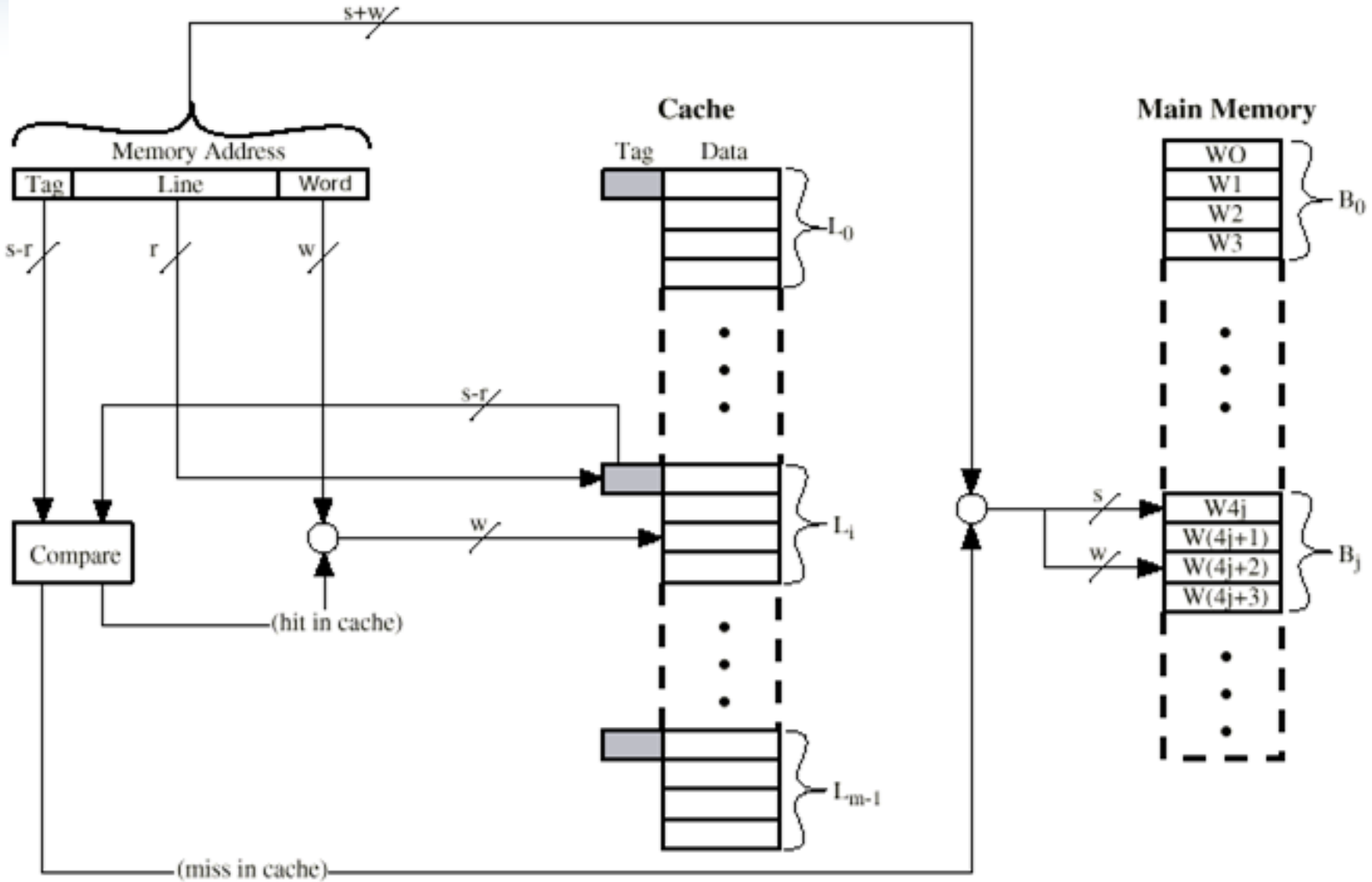
## Main memory address

| Tag s-r | Line or Slot r | Word w |
|---|---|---|
| 8 | 14 | 2 |

- 24 bit address
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
  - 8 bit tag (= 22 – 14)
  - 14 bit slot or line

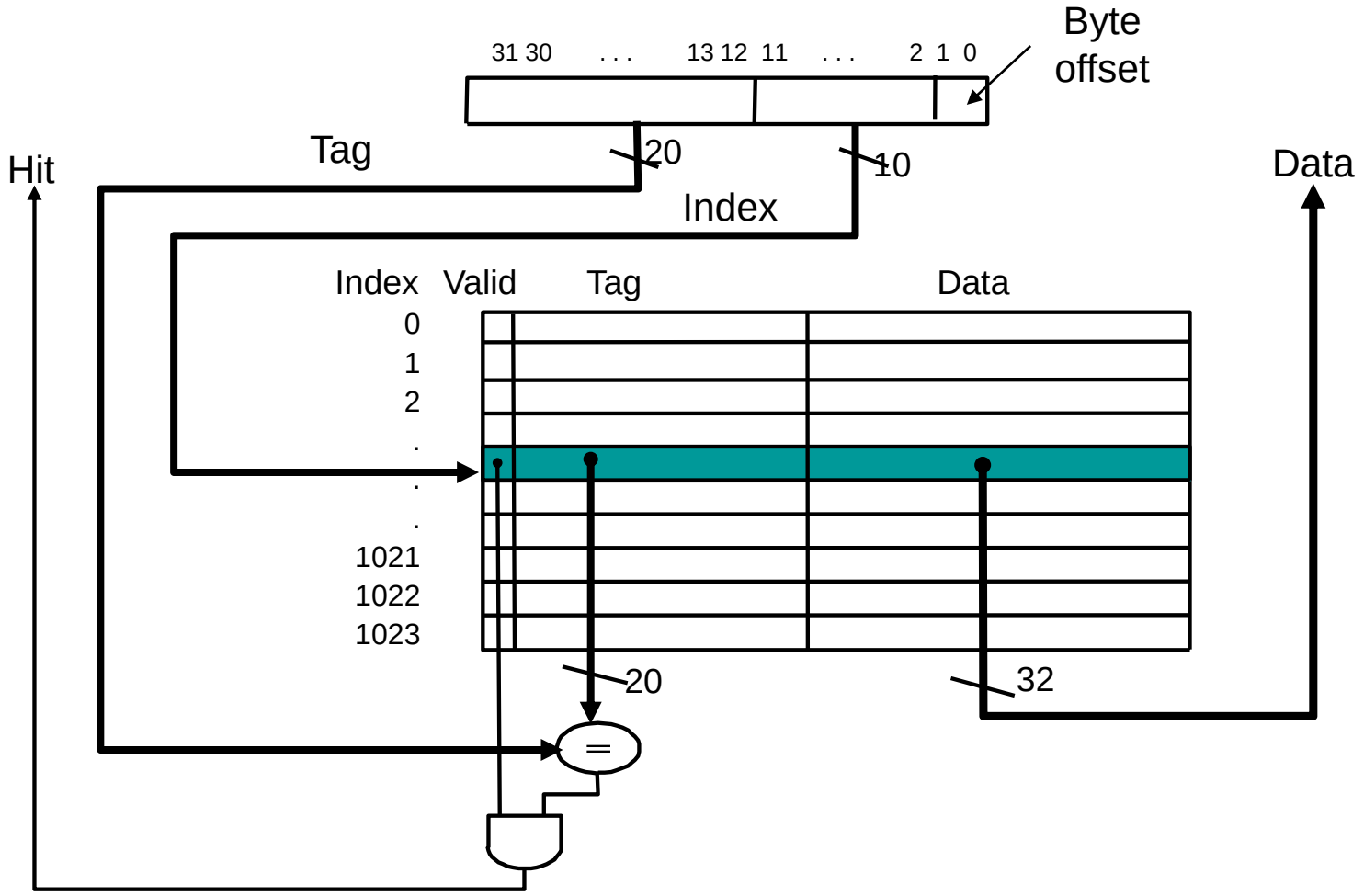
- No two blocks in the same line have the same Tag field
- Check contents of cache by finding line and checking Tag

# Direct Mapping Cache – Organization



# MIPS Direct Mapped Cache Example

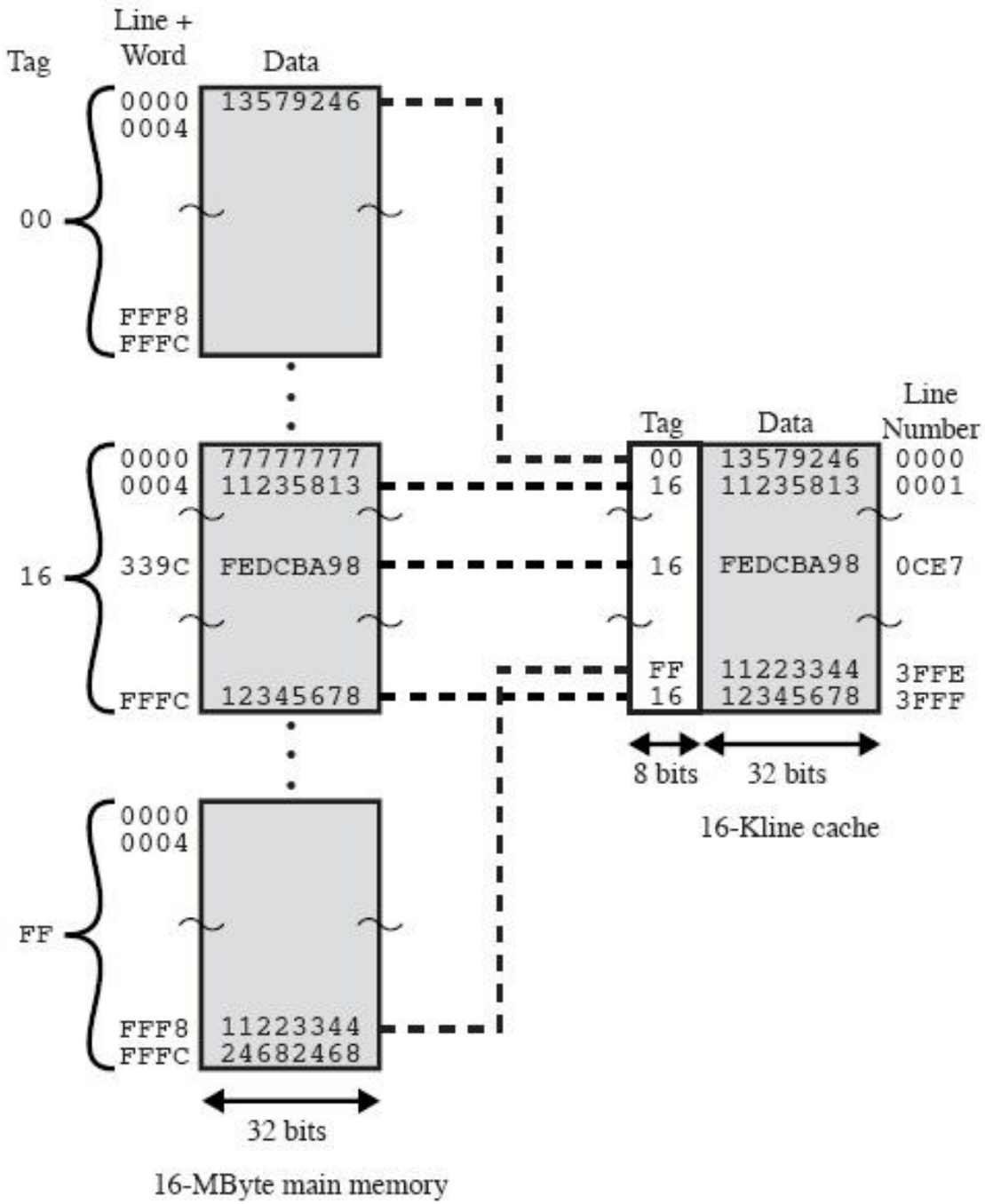
- One word/block, cache size = 1K words




*What kind of locality are we taking advantage of?*

# Direct Mapping Cache Example

- m = 16K = $2^{14}$
- i = j modulo $2^{14}$

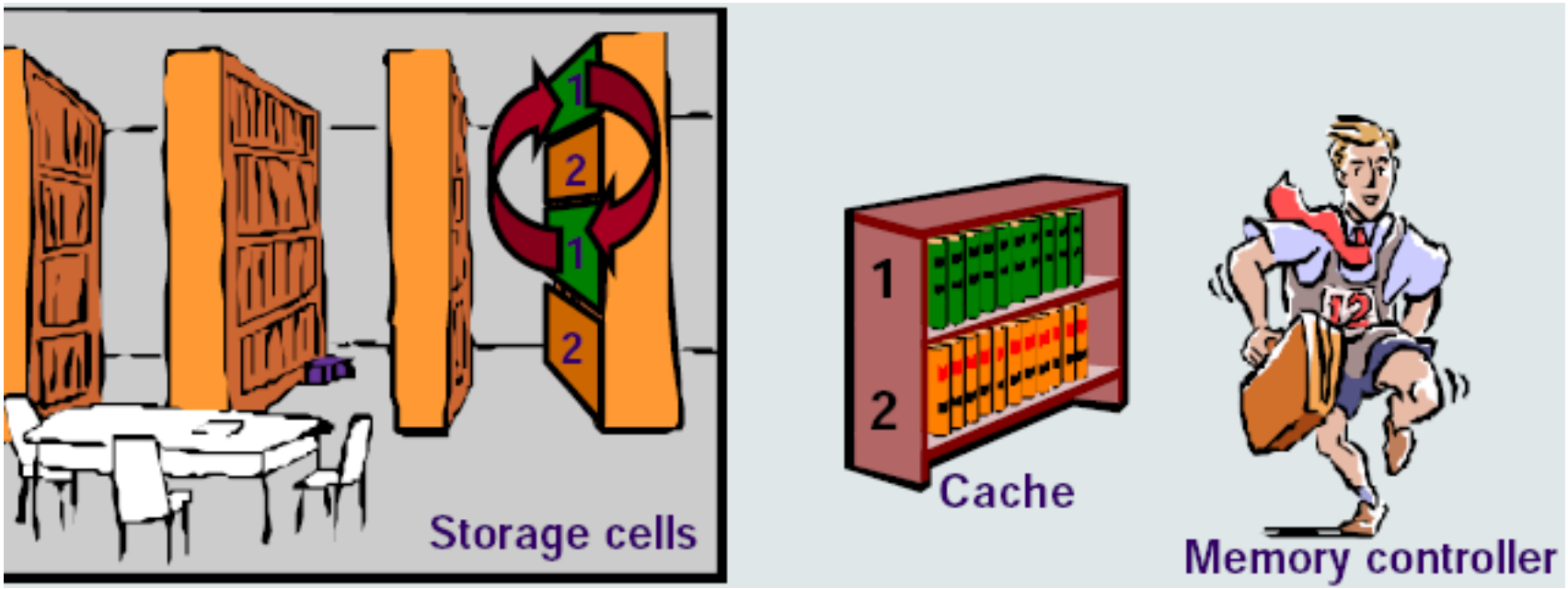


# Direct Mapping Cache – Summary

- Address length = (s + w) bits
- Number of addressable units = $2^{s+w}$ words or bytes
- Block size = line size = $2^{w}$ words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in cache = m = $2^{r}$
- Size of tag = (s – r) bits

# Direct Mapping Pros & Cons

- Simple to implement and therefore inexpensive.
- Fixed location for blocks.
  - If a program accesses 2 blocks that map to the same cache slot repeatedly, cache miss rate is very high (known as *thrashing* phenomenon)

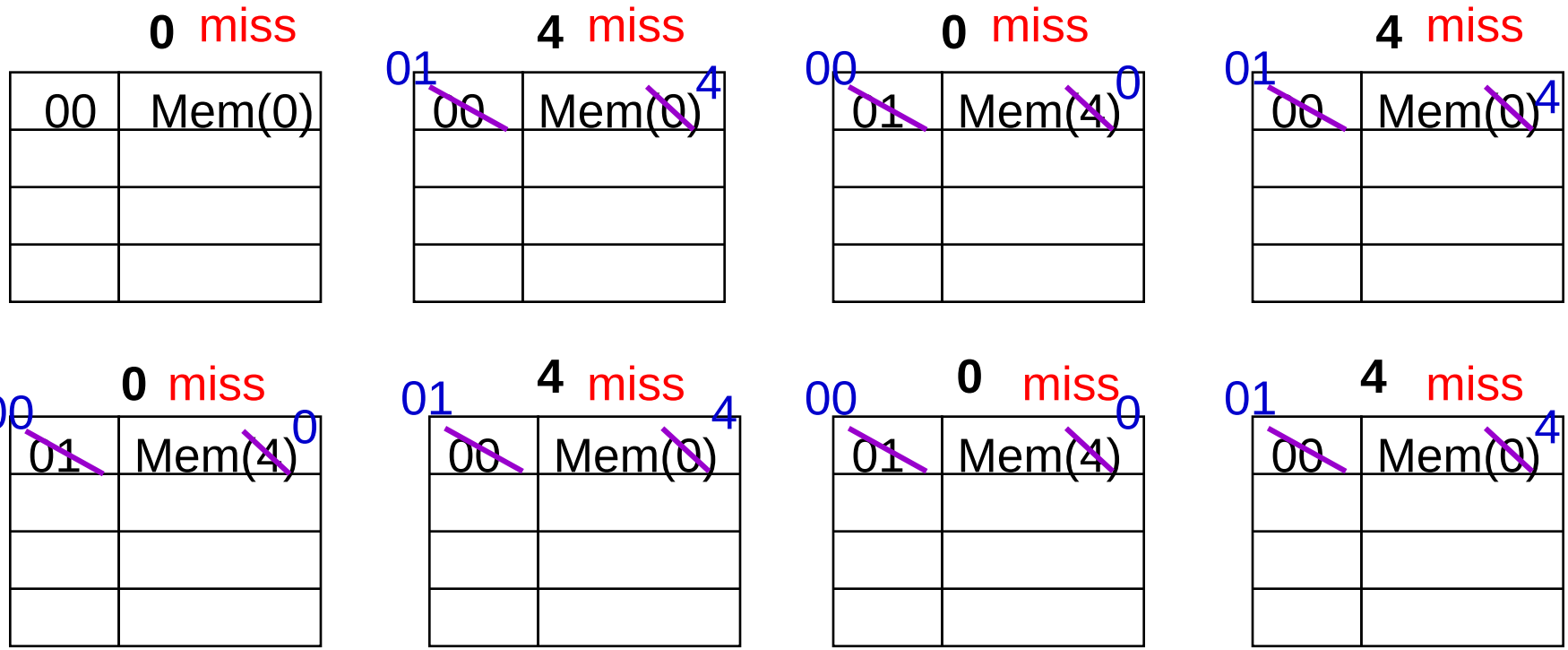


# Thrashing

- Consider the main memory word reference string

0 4 0 4 0 4 0 4

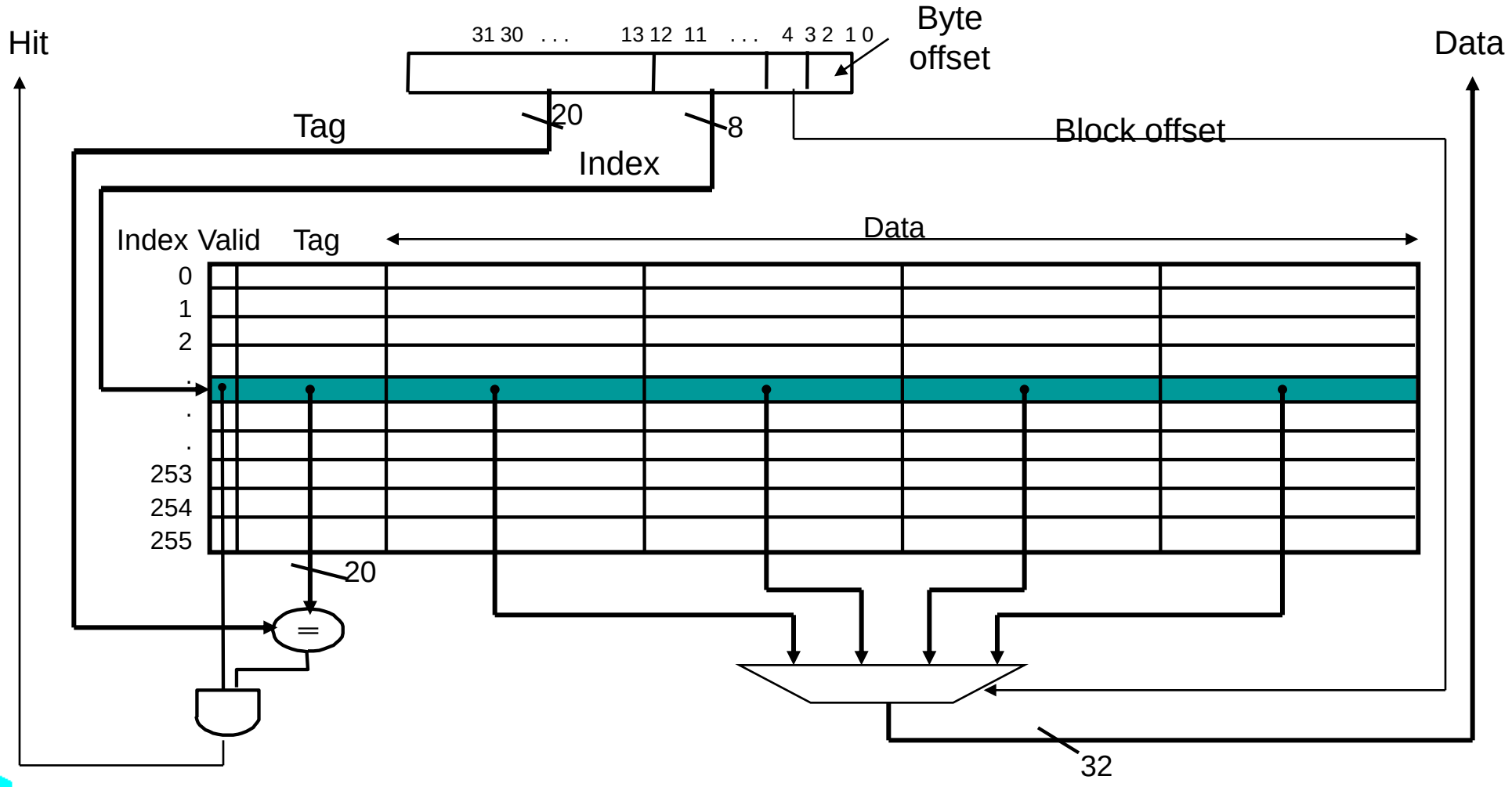
Start with an empty cache - all blocks initially marked as not valid




- Ping pong effect due to conflict misses – two memory locations that map into the same cache block

# Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



# Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

**0** miss

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| | | |

**1** hit

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| | | |

**2** miss

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

**3** hit

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

**4** miss

| | | |
|---|---|---|
| 01 ~~00~~ | Mem(1) 5 | Mem(0) 4 |
| 00 | Mem(3) | Mem(2) |

**3** hit

| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

**4** hit

| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

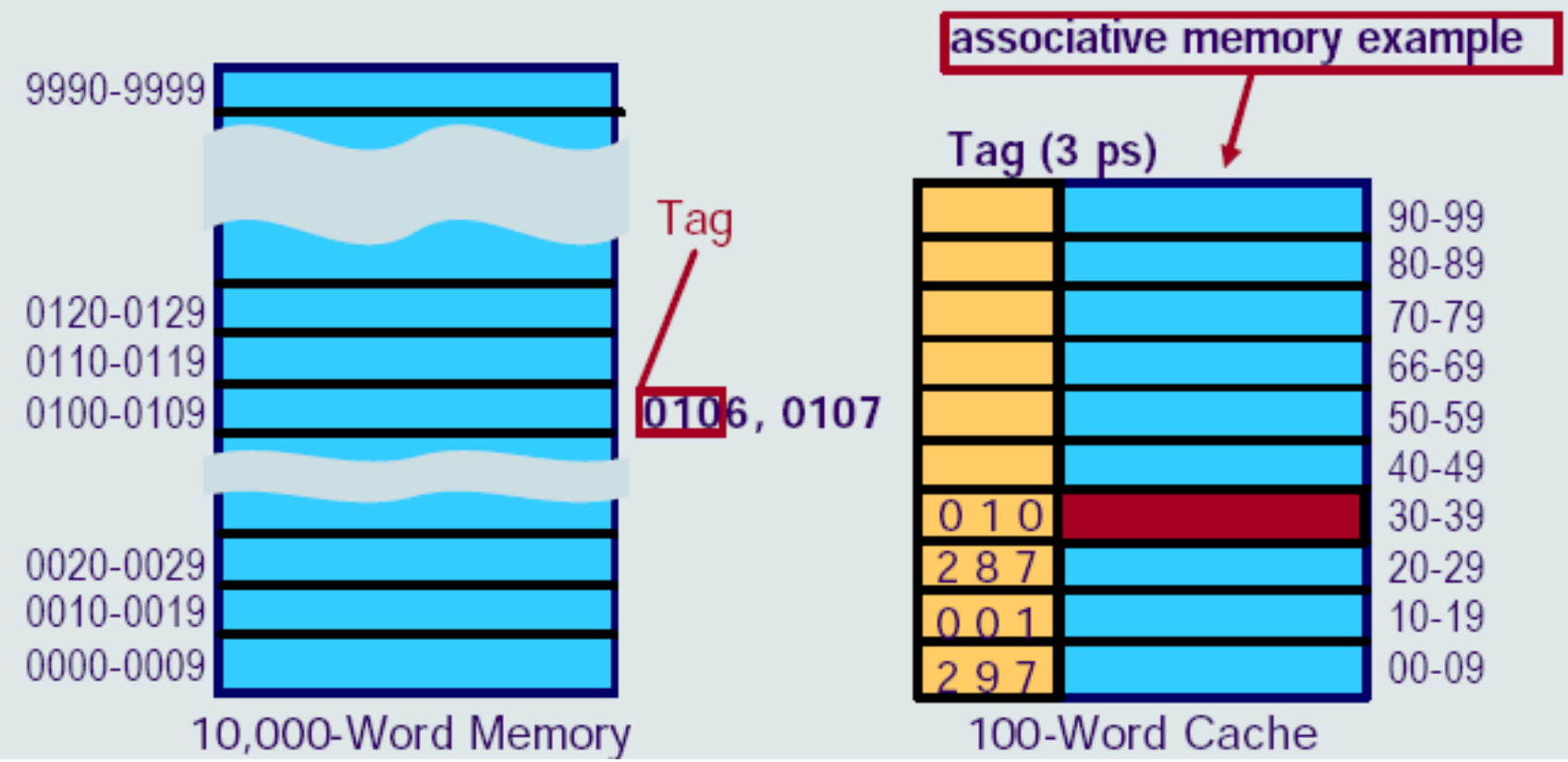
**15** miss

| | | |
|---|---|---|
| 11 ~~01~~ | Mem(5) | Mem(4) |
| 00 | Mem(3) 15 | Mem(2) 14 |

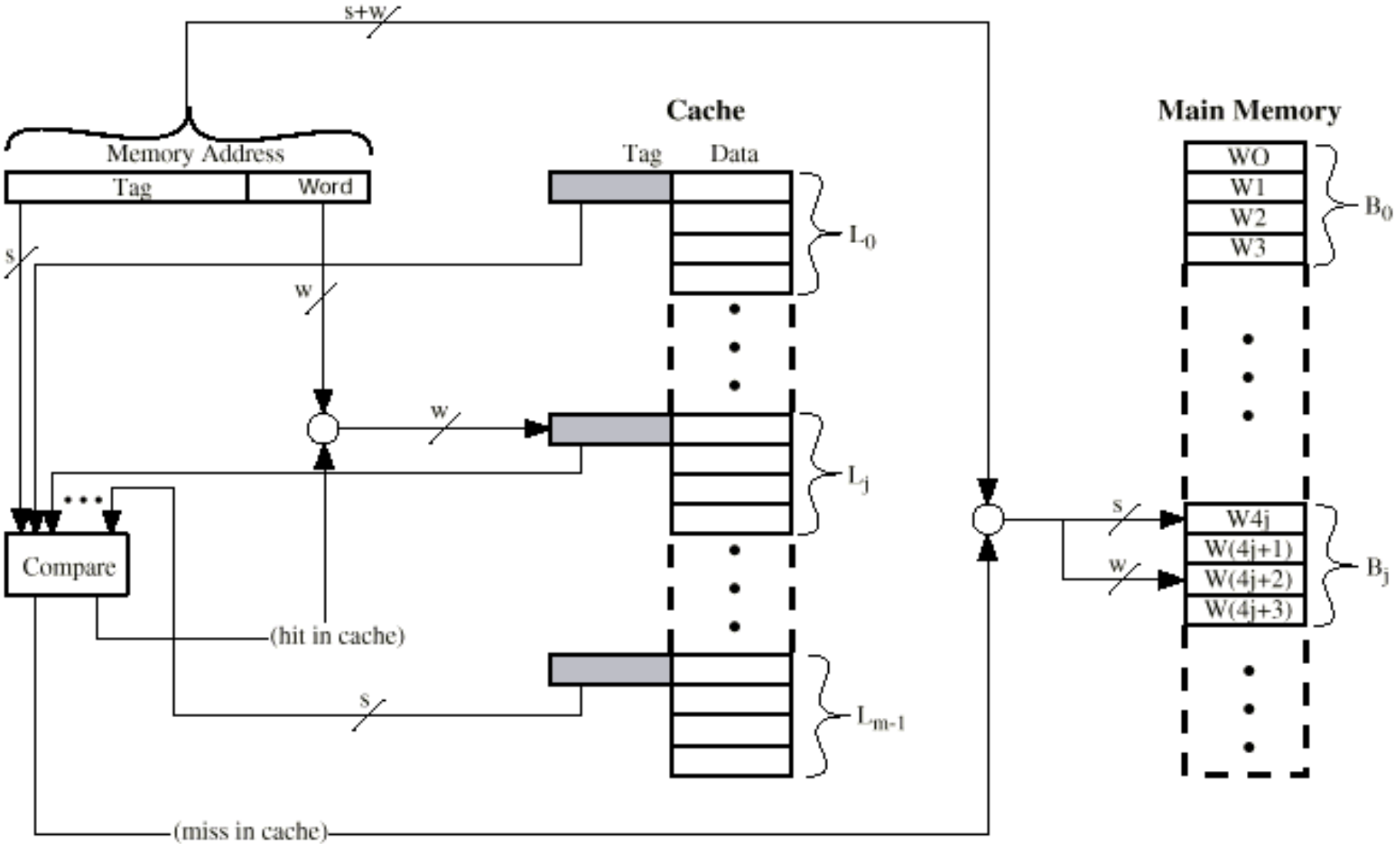
# Associative Mapping

- A main memory block can be loaded into any slot of the cache.
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- To determine if a block is in the cache, a mechanism is needed to simultaneously examine every slot's tag.
  - Cache searching gets expensive

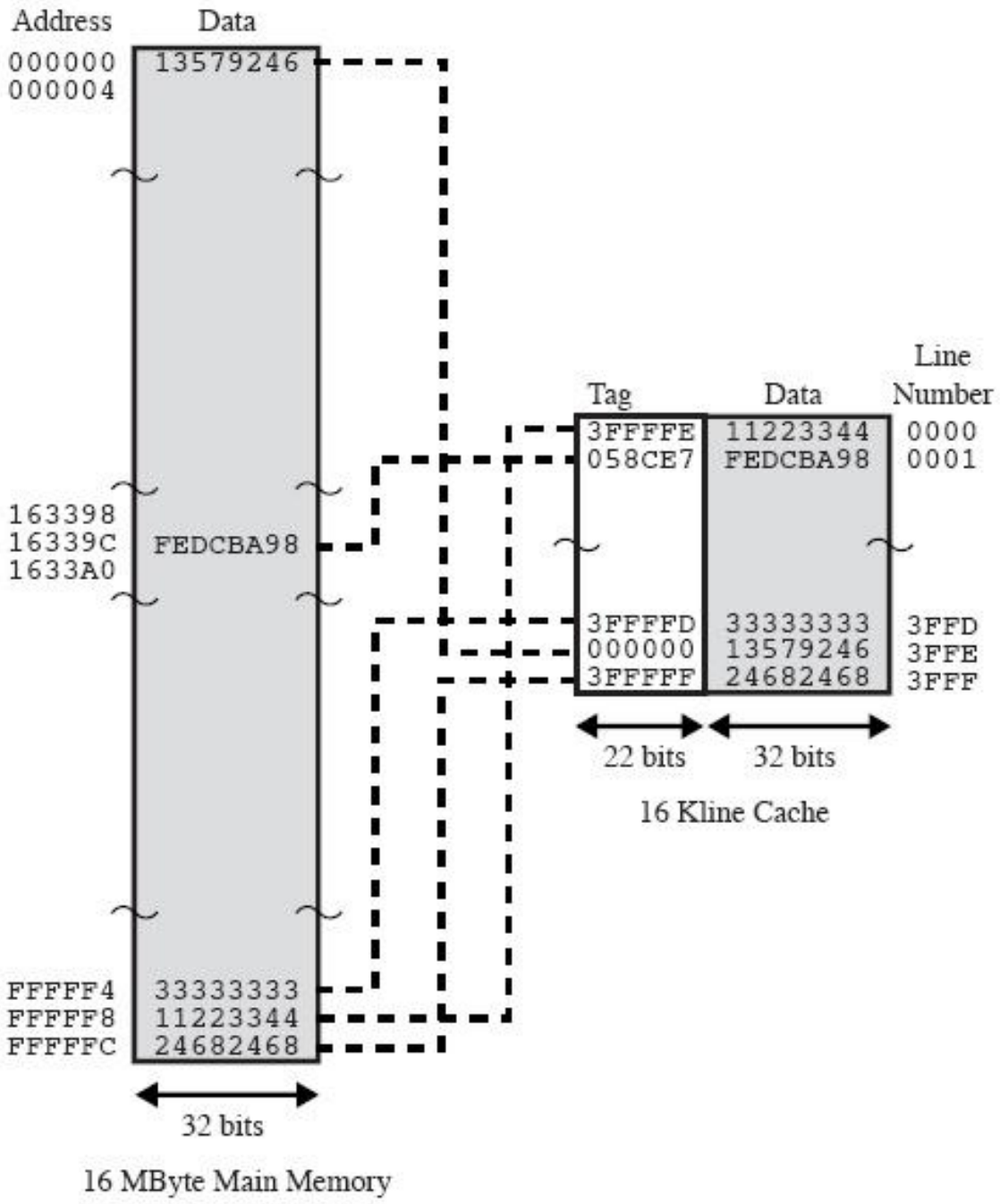
# Associative Mapping



# Fully Associative Cache Organization



# Associative Mapping Example



# Associative Mapping Address Structure


- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.
  - Address Tag Data Cache line
  - FFFFFC 3FFFFF 24682468 3FFF

# Associate Mapping – Summary

- Address length = (s + w) bits
- Number of addressable units = $2^{s+w}$ words or bytes
- Block size = line size = $2^{w}$ words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in cache = undetermined
- Size of tag = s bits

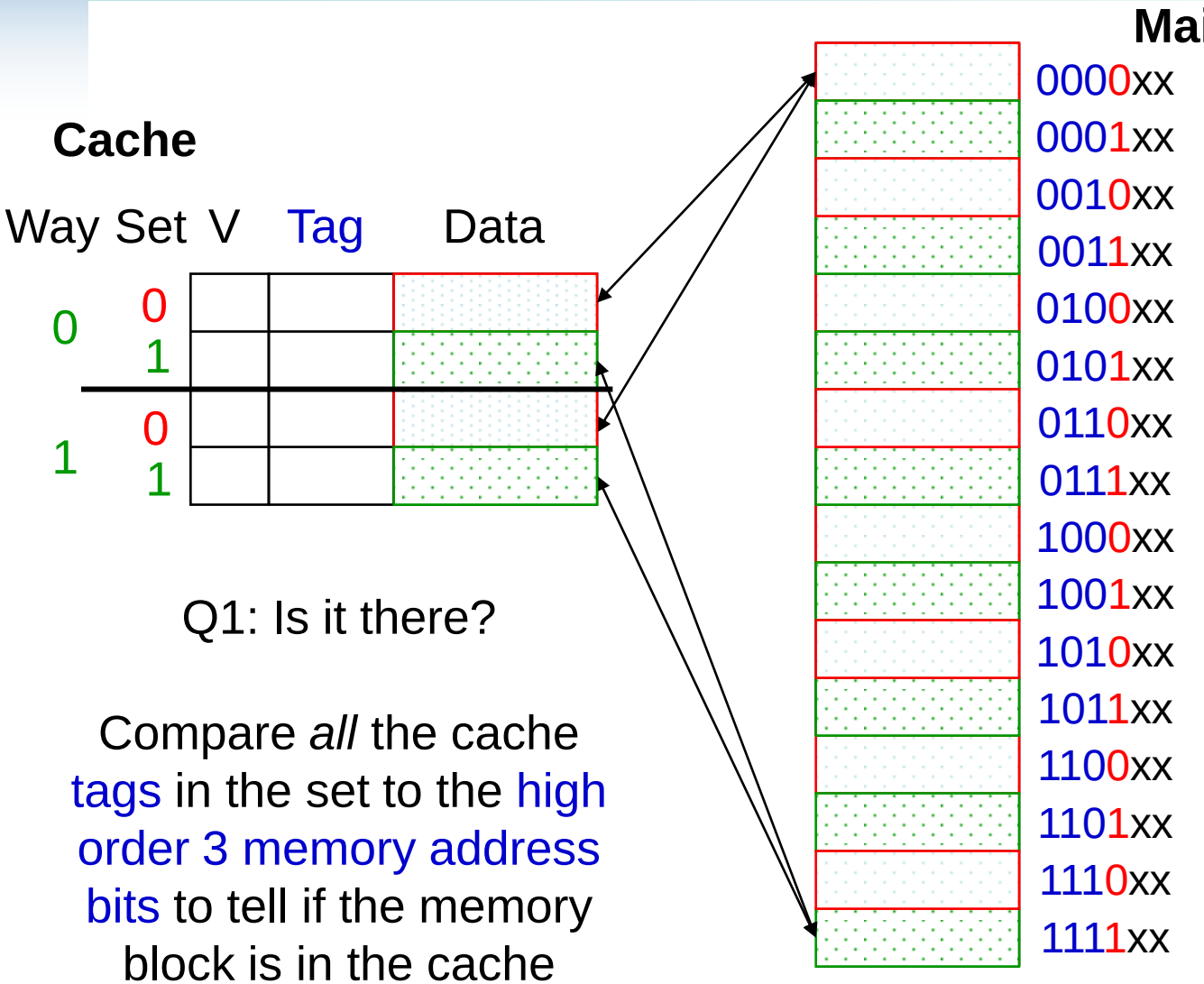
# Set Associative Mapping

- The cache is divided into a number of sets (v).
- Each set contains a number of slots (k).
  - $m = k.v$
- A given block maps to any slot in a given set.
  - $i = j$ modulo $v$ (i = cache set number)
  - e.g. block $j$ can be in any slot of set $i$.
- For example, 2 slots per set (k = 2):
  - 2-way associative mapping
  - A given block can be in one of 2 slots in only one given set.
- Direct mapping: k = 1 (no alternative).
- Fully associative: v = 1 (k = total number of all slots in the cache, all mappings possible).

# Set Associative Mapping Example

- 13 bit set number
- Block number in main memory is modulo $2^{13}$
- Blocks 000000, 008000, 00A000, 00C000, 00E000, … FF8000 map into same cache set 0

# Set Associative Cache Example




Q1: Is it there?

Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

**Two low order bits define the byte in the word (32-b words) One word blocks**

Q2: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

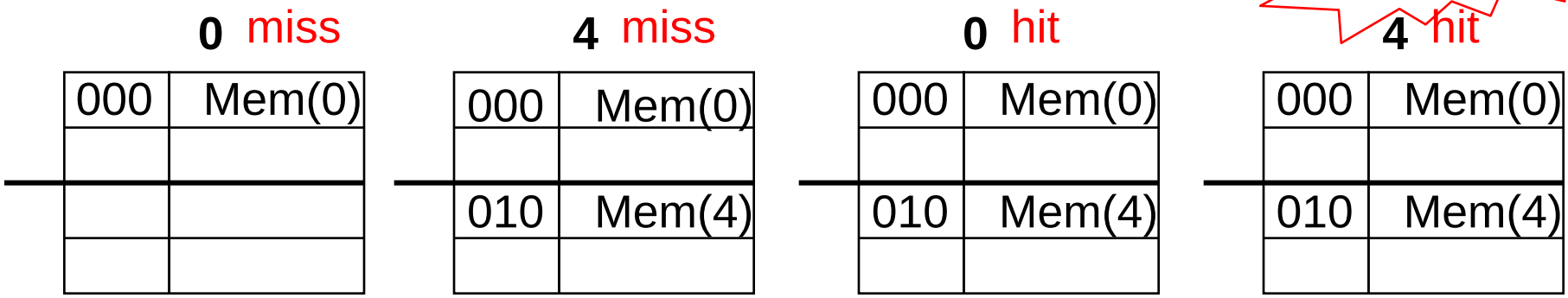
# Another Reference String Mapping

- Consider the main memory word reference string

0 4 0 4 0 4 0 4

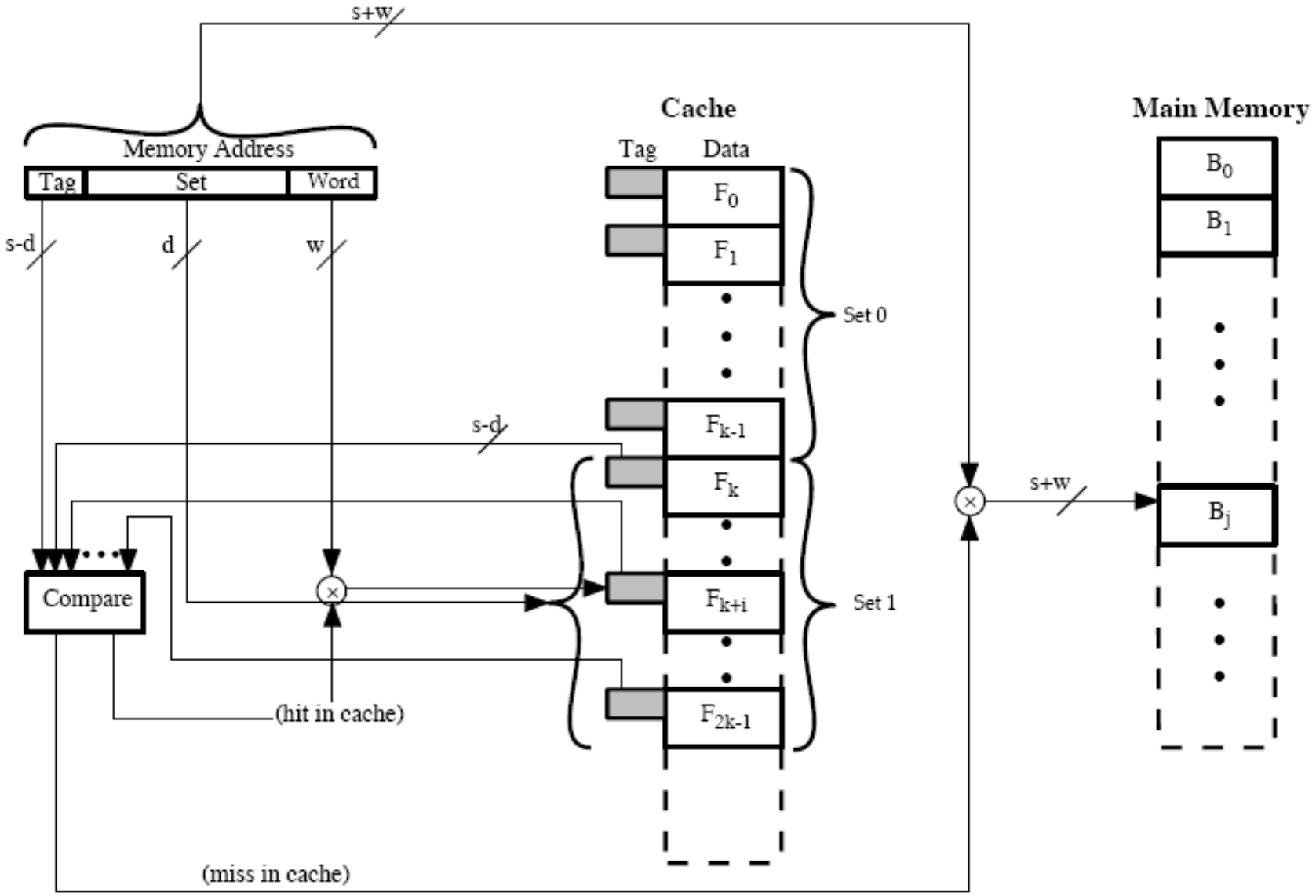
Start with an empty cache - all blocks initially marked as not valid

8 requests, 2 misses



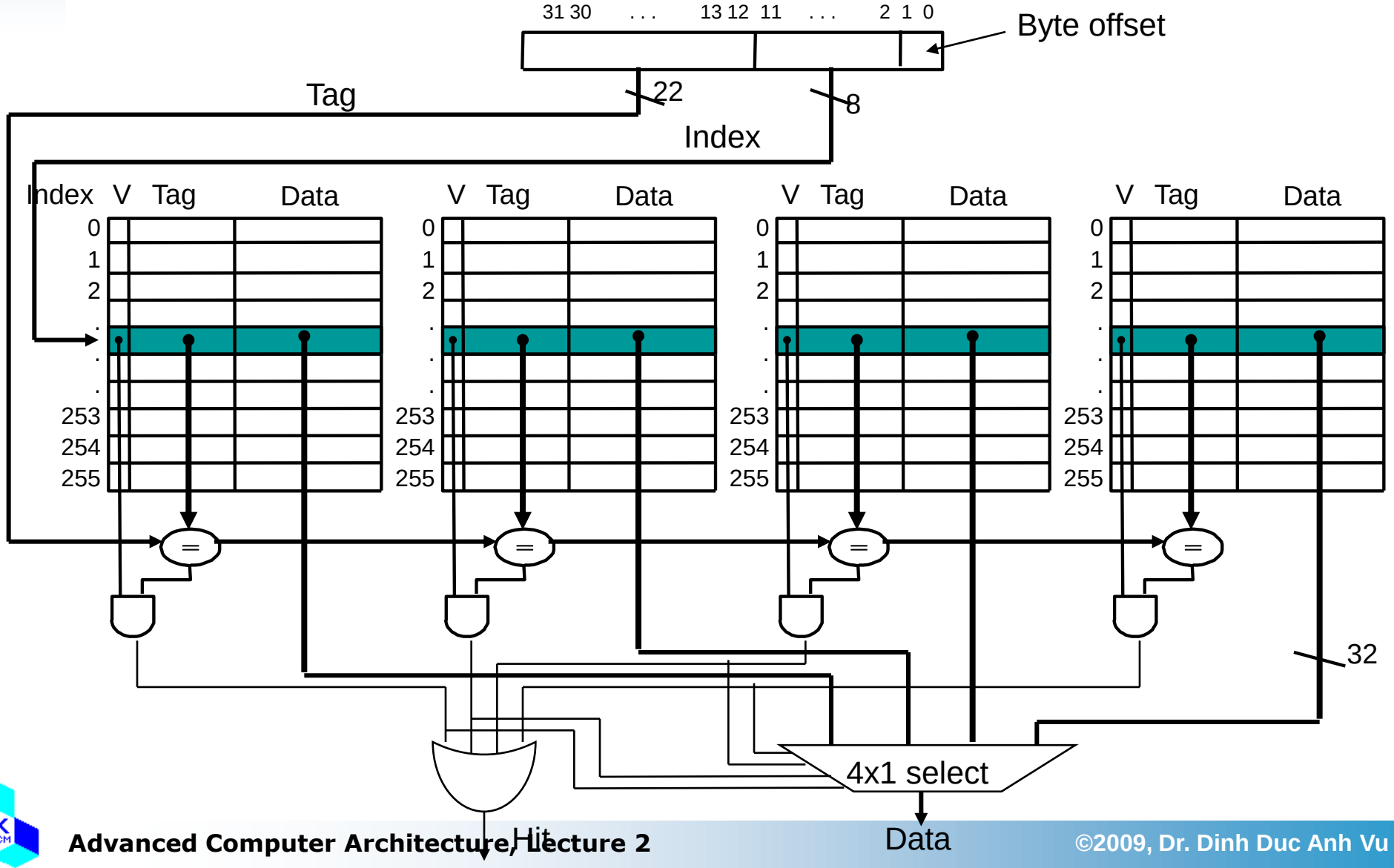

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

# k-Way Set Associative Cache Organization



# Four-Way Set Associative Cache

- $2^8$ = 256 sets each with four ways (each with one block)



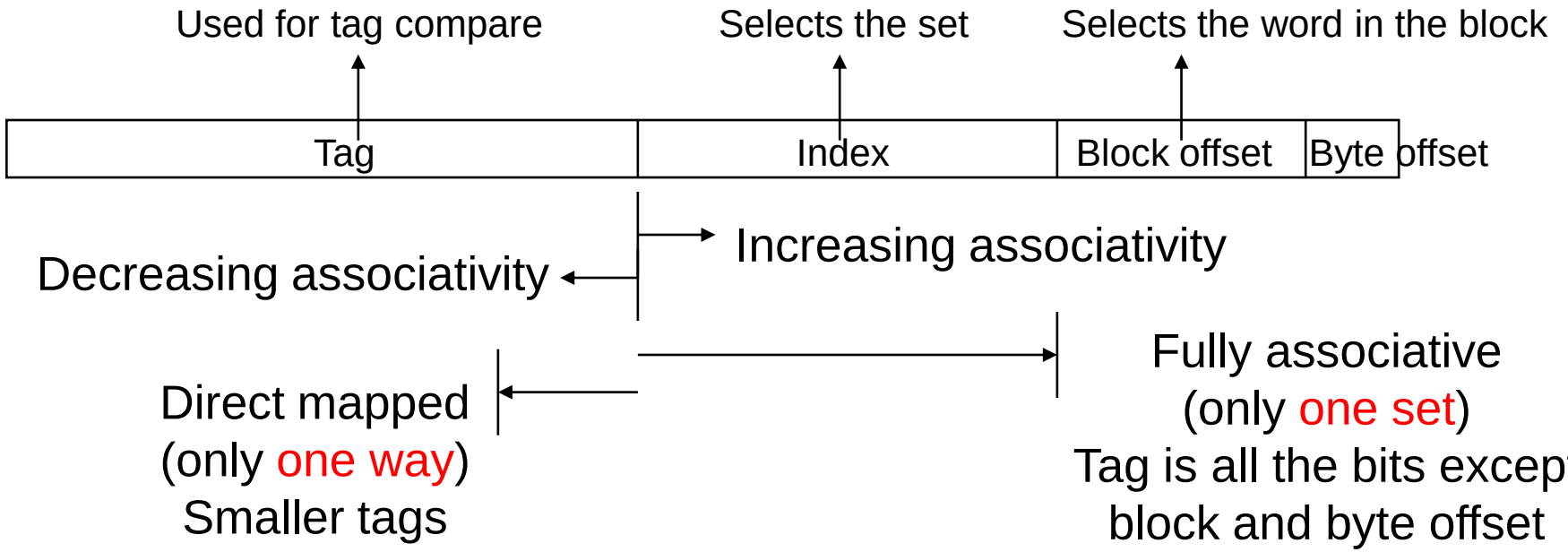
# Set Associative Mapping Address Structure

| Tag 9 bit | Set 13 bit | Word 2 bit |
|---|---|---|

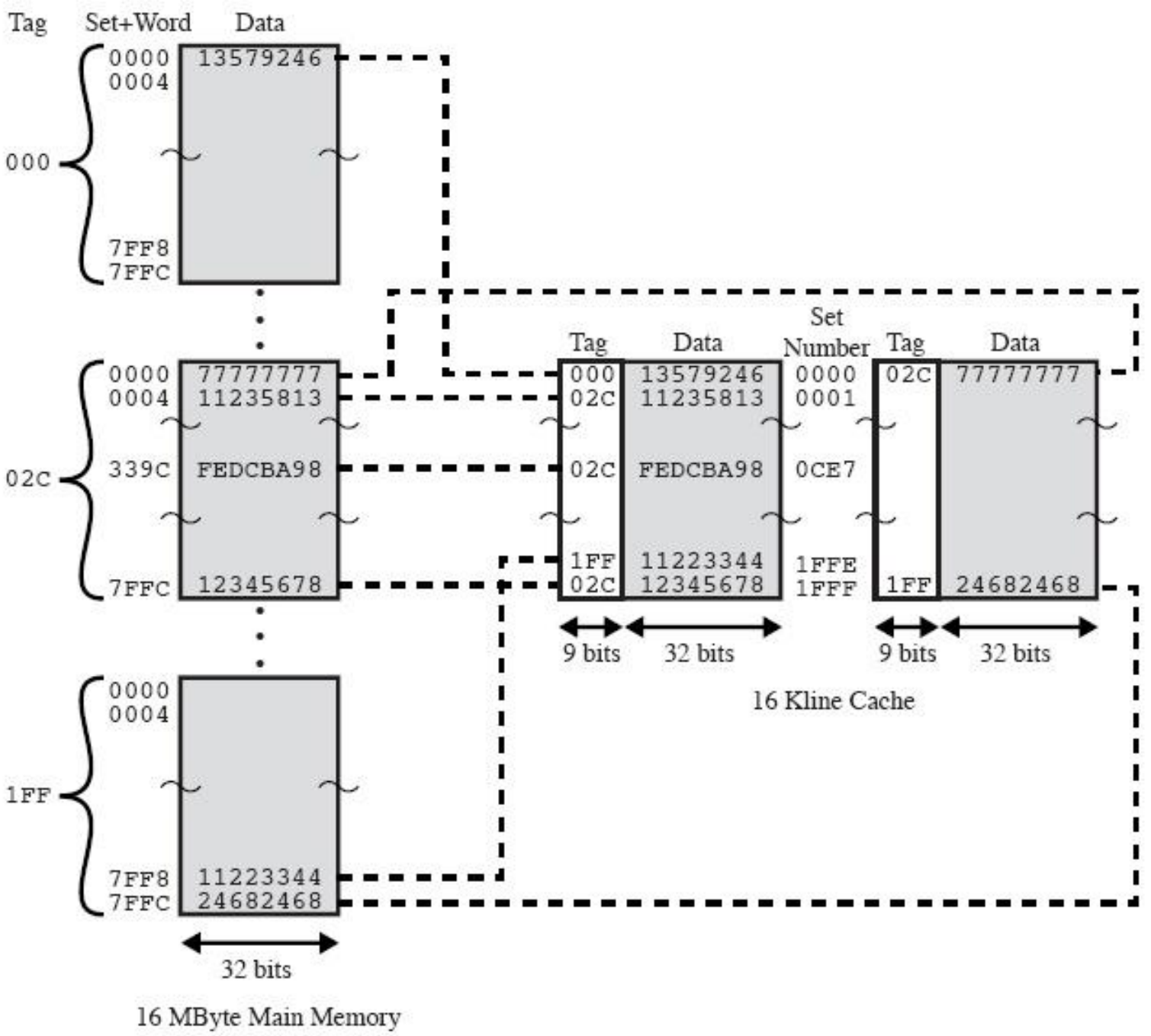
- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g
  - Address Tag Data Set number
  - 1FF 7FFC 1FF 24682468 1FFF
  - 02C 7FFC 02C 12345678 1FFF

# Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



# 2-Way Set Associative Mapping Example



# Set Associate Mapping – Summary

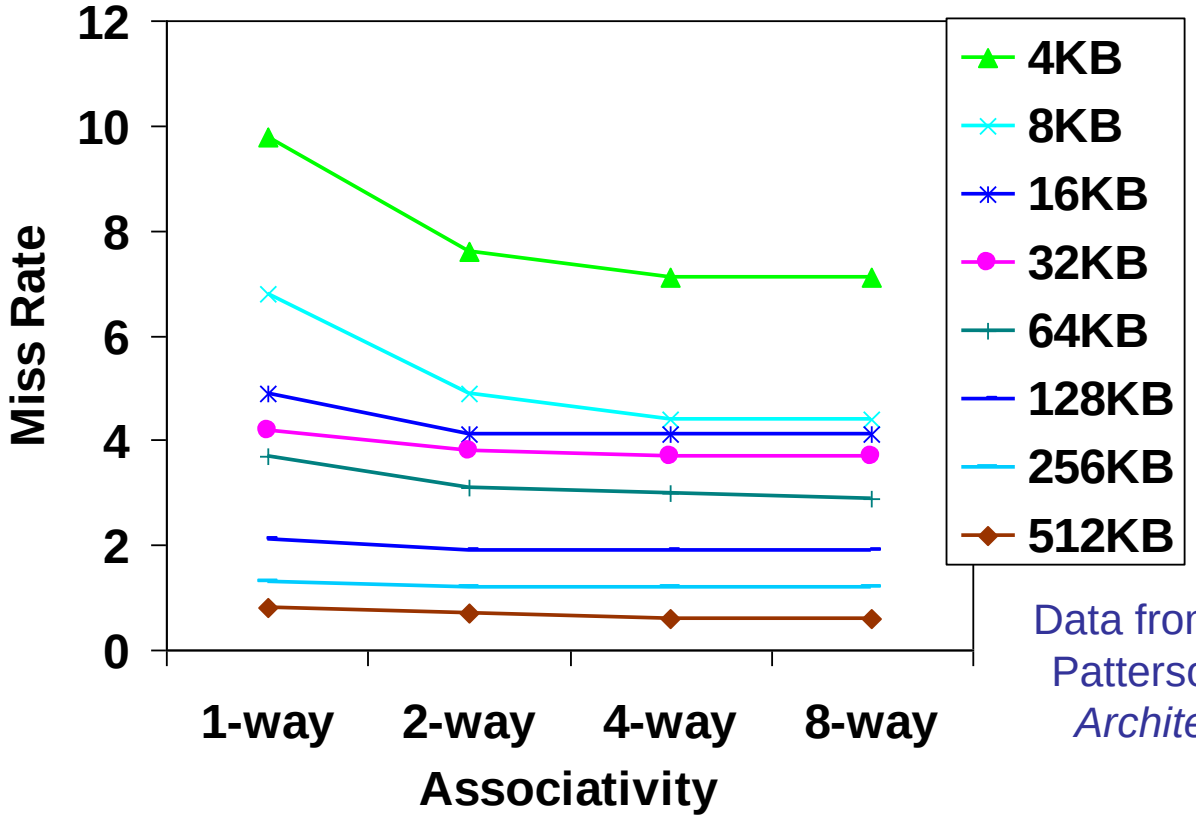
- Address length = (s + w) bits
- Number of addressable units = $2^{s+w}$ words or bytes
- Block size = line size = $2^{w}$ words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in set = k
- Number of sets = v = $2^{d}$
- Number of lines in cache = k.v = k.$2^{d}$
- Size of tag = (s – d) bits

# Costs of Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement?
  - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)
- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So its not possible to just assume a hit and continue and recover later if it was a miss

# Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation

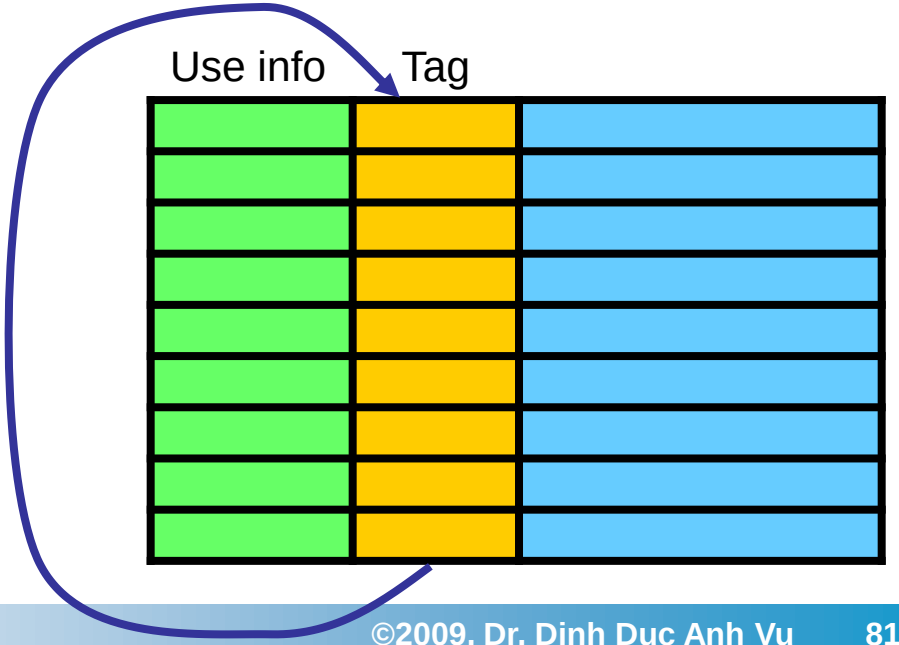



Data from Hennessy & Patterson, *Computer Architecture*, 2003

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate

# Replacement Algorithms

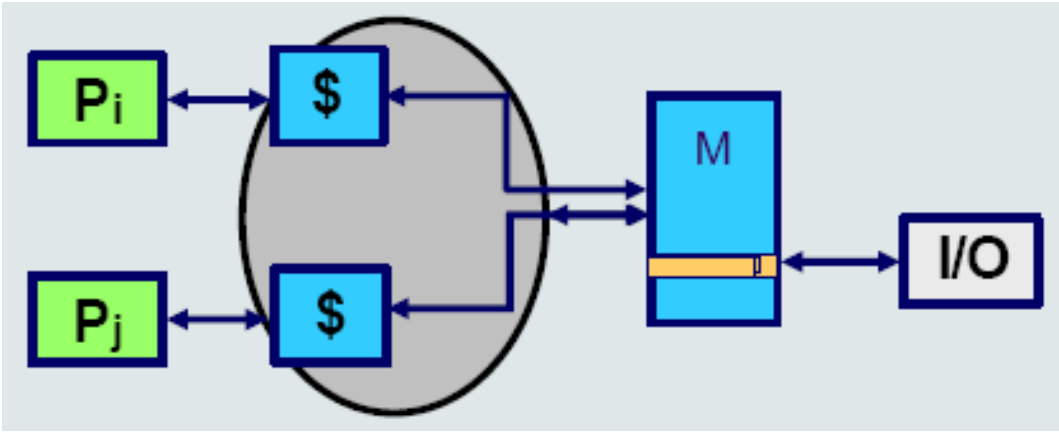
- With direct mapping, it is no need.
- With associative mapping, a replacement algorithm is needed in order to determine which block to replace:
  - Hardware implemented algorithm (speed)
  - Least-recently used (LRU) – replace the block that has been in the cache longest with no reference to it.
  - First-in-first-out (FIFO) – replace block that has been in cache longest
  - Least-frequently used (LFU) – replace the block that has experienced the fewest references.
  - Random.



# Write Policy

- **The problem**:
  - How to keep cache content and the content of main memory consistent without losing too much performance?
  - Must not overwrite a cache block unless main memory is up to date
  - I/O may address main memory directly
  - Multiple CPUs may have individual caches

# What Happen with Multiprocessor?

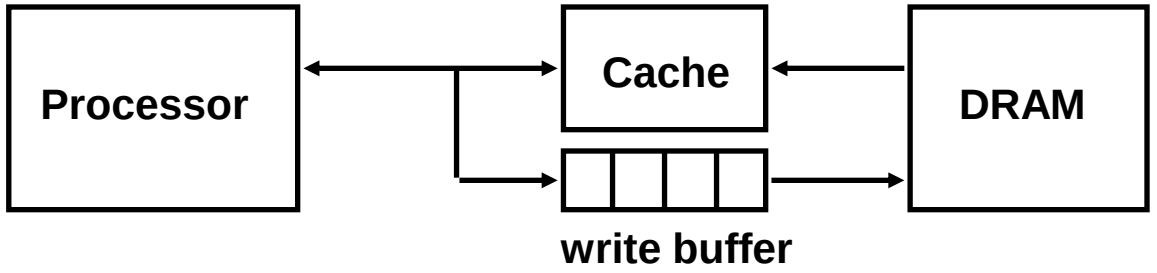



- Different processors may access values at same memory location
- Multiple copies of the same data in different caches.
  - How to ensure data integrity at all times?
- An update by a processor at time t should be available for other processors at time t+1.
  - I/O may address main memory directly.

# Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!
- Write hits (D$ only)
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache contents to the next level in the memory hierarchy when that cache block is “evicted”)
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through) so don’t need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer, so only have to stall if the write buffer is full

# Write Buffer for Write-Through Caching




- Write buffer between the cache and main memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: writes contents of the write buffer to memory
- The write buffer is just a FIFO
  - Typical number of entries: 4
  - Works fine if store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare
  - When the store frequency (w.r.t. time) → 1 / DRAM write cycle leading to write buffer saturation
    - One solution is to use a write-back cache; another is to use an L2 cache (next lecture)

# Sources of Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it
  - If you are going to run “millions” of instruction, compulsory misses are insignificant
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (next lecture)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size

# Handling Cache Misses

- Read misses (I$ and D$)
  - stall the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume

  or (normally used in write-back caches)

  2. Write allocate – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall

  or (normally used in write-through caches with a write buffer)

  3. No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full; must invalidate the cache block since it will be inconsistent (now holding stale data)

# Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – datapath resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and datapath) first
  - Nonblocking cache – allows the datapath to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - Can't use write allocate or will end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block), so must fetch the block from memory first and pay the stall time

# Write Through

- All write operations are passed to main memory; if the addressed location is currently hold in the cache, the cache is updated so that it is coherent with the main memory.
- For writes, the processor always slows down to main memory speed.
- Since the percentage of writes is small (ca. 15%), the scheme doesn't lead to large performance reduction.

**For multiprocessor:**

- There is inconsistency due to other cache copies of the same memory location.
- Multiple CPUs must monitor main memory traffic to keep local cache up to date.
  - Lots of traffic.

# Write Through (Cont'd)

- **Write through with buffered write:**
  - The same as write-through, but instead of slowing the processor down by writing directly to main memory, the write address and data are stored in a high-speed write buffer; the write buffer transfers data to main memory while the processor continues its task.
  - Higher speed, but more complex hardware.

# Write Back

- Write operations update only the cache memory which is not kept coherent with main memory. When the slot is replaced from the cache, its content has to be copied back to memory.
- Update bit for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
- N.B. 15% of memory references are writes
- Good performance (usually several writes are performed on a cache block before it is replaced), but more complex hardware is needed

**For multiprocessor:**

- Other caches get also out of sync!
- I/O must access main memory through cache.

# Cache Coherence – Software Solutions

- Based on code analysis:
  - Determine which data items may become unsafe for caching.
  - Mark them, so that they are not cached.
  - Alternatively, determine the unsafe periods and insert code to enforce cache coherence.
- Compiler and operating system deal with problem.
- Overhead transferred to compile time.
- Design complexity transferred from hardware to software.
- However, software tends to make conservative decisions
  - Inefficient cache utilization.

# Cache Coherence – Hardware Solution

- Cache coherence protocols
- Dynamic recognition of potential problems
- Run time
- More efficient use of cache
- Transparent to programmer
- Two main categories:
  - Directory protocols
  - Snoopy protocols

# Directory Protocols

- Collect and maintain information about copies of data in cache
- Directory stored in main memory
- Requests are checked against directory
- Appropriate transfers are performed
  - Between memory and caches or between caches
- Creates central bottleneck
- Effective in large scale systems with complex interconnection schemes

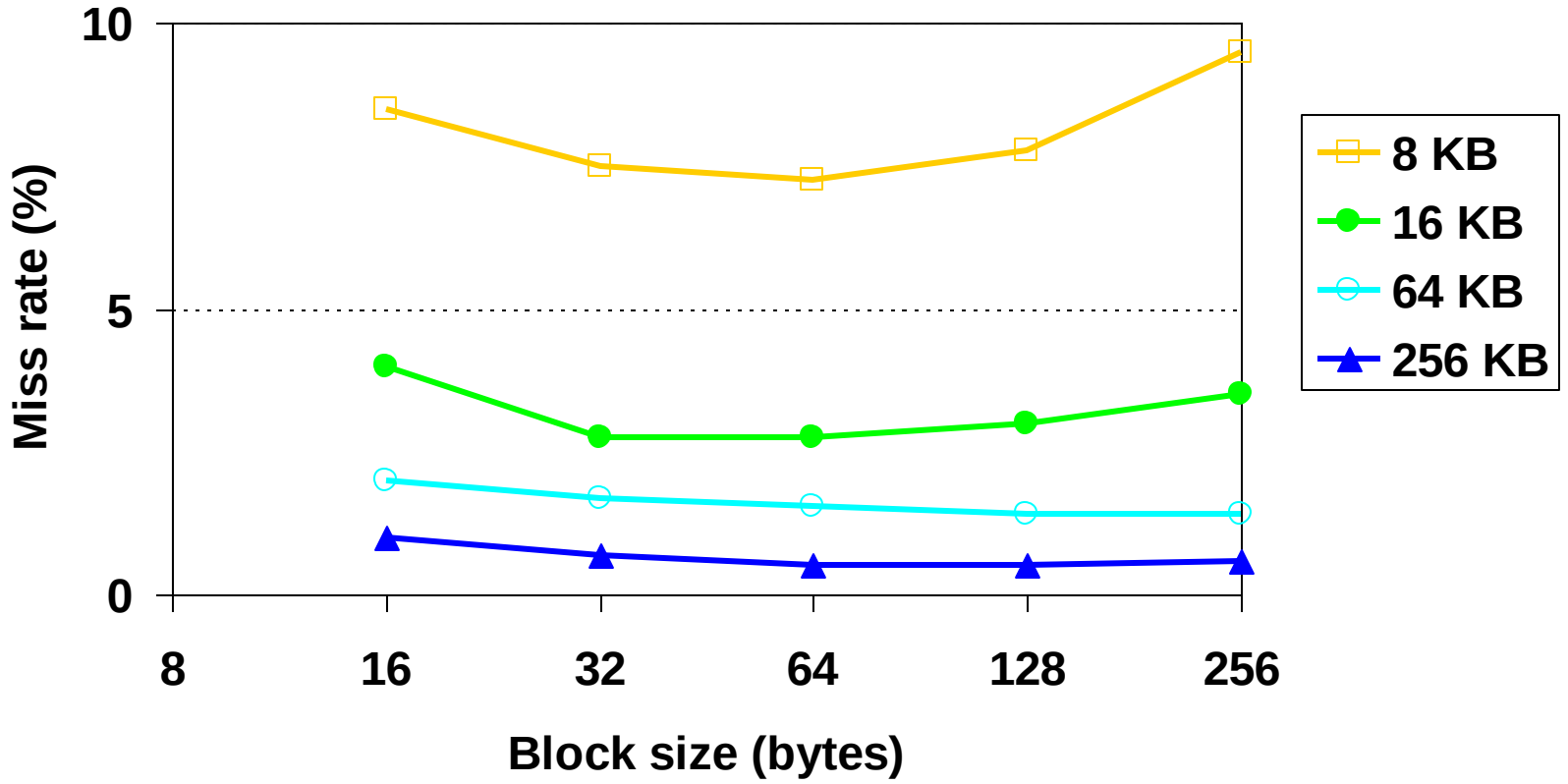
# Snoopy Protocols

- Distribute cache coherence responsibility among cache controllers
- Cache recognizes that a line is shared
- Updates announced to other caches
- Suited to bus-based multiprocessor
- Increases bus traffic

# Line Size

- The relationship between block size and hit ratio is complex
  - Block size increases
    - Hit ratio increase because of the principle of locality
    - Hit ratio will be decrease as the block size becomes even bigger
- Larger blocks reduce the number of blocks that fit into a cache
- As a block becomes larger, each additional word is farther from the requested word, therefore less likely to be needed in the near future

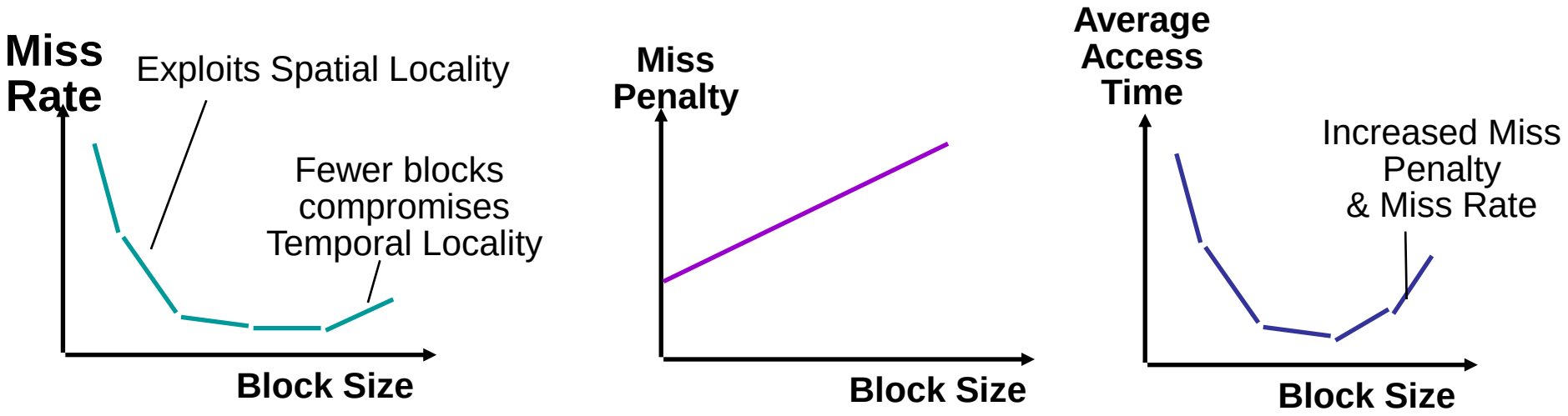
# Miss Rate vs Block Size vs Cache Size




- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

# Block Size Tradeoff

- Larger block sizes take advantage of spatial locality but
  - If the block size is too big relative to the cache size, the miss rate will go up
  - Larger block size means larger miss penalty
    - Latency to first word in block + transfer time for remaining words




- In general, Average Memory Access Time = Hit Time + Miss Penalty x Miss Rate

# Number of caches

- More recently, the use of multiple caches has become the norm
  - Multilevel cache
  - Unified or split caches
- Multilevel cache
  - Internal cache (L1 cache):
    - On-chip
    - Reduces the processor's external bus activity => increases system performance
  - External cache (L2 cache):
    - Off-chip
    - Separate data path is used (instead of the system bus) for transfer between the L2 cache and the processor
    - Could be incorporated on the processor chip to improve performance
  - The use of multilevel cache improve performance, but complicate all of the design issues related to caches

# Split Data and Instruction Caches?

- Unified caches:
  - Better balance the load between instruction and data fetches depending on the dynamics of the program execution.
  - Design and implementation are cheaper.
  - Less performance.
- Split caches (Harvard Architectures):
  - Competition for the cache between instruction processing and execution units is eliminated.
  - Instruction fetch can proceed in parallel with memory access from the CPU for operands.
  - One may be overloaded while the other is under utilized.

# Intel Cache Evolution

| Problem | Solution | Processor on which feature first appears |
|---|---|---|
| External memory slower than the system bus. | Add external cache using faster memory technology. | 386 |
| Increased processor speed results in external bus becoming a bottleneck for cache access. | Move external cache onchip, operating at the same speed as the processor. | 486 |
| Internal cache is rather small, due to limited space on chip | Add external L2 cache using faster technology than main memory | 486 |
| Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place. | Create separate data and instruction caches. | Pentium |
| Increased processor speed results in external bus becoming a bottleneck for L2 cache access. | Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache. | Pentium Pro |
| | Move L2 cache on to the processor chip. | Pentium II |
| Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small | Add external L3 cache. | Pentium III |
| | Move L3 cache on-chip. | Pentium 4 |

# Cache Architecture Examples

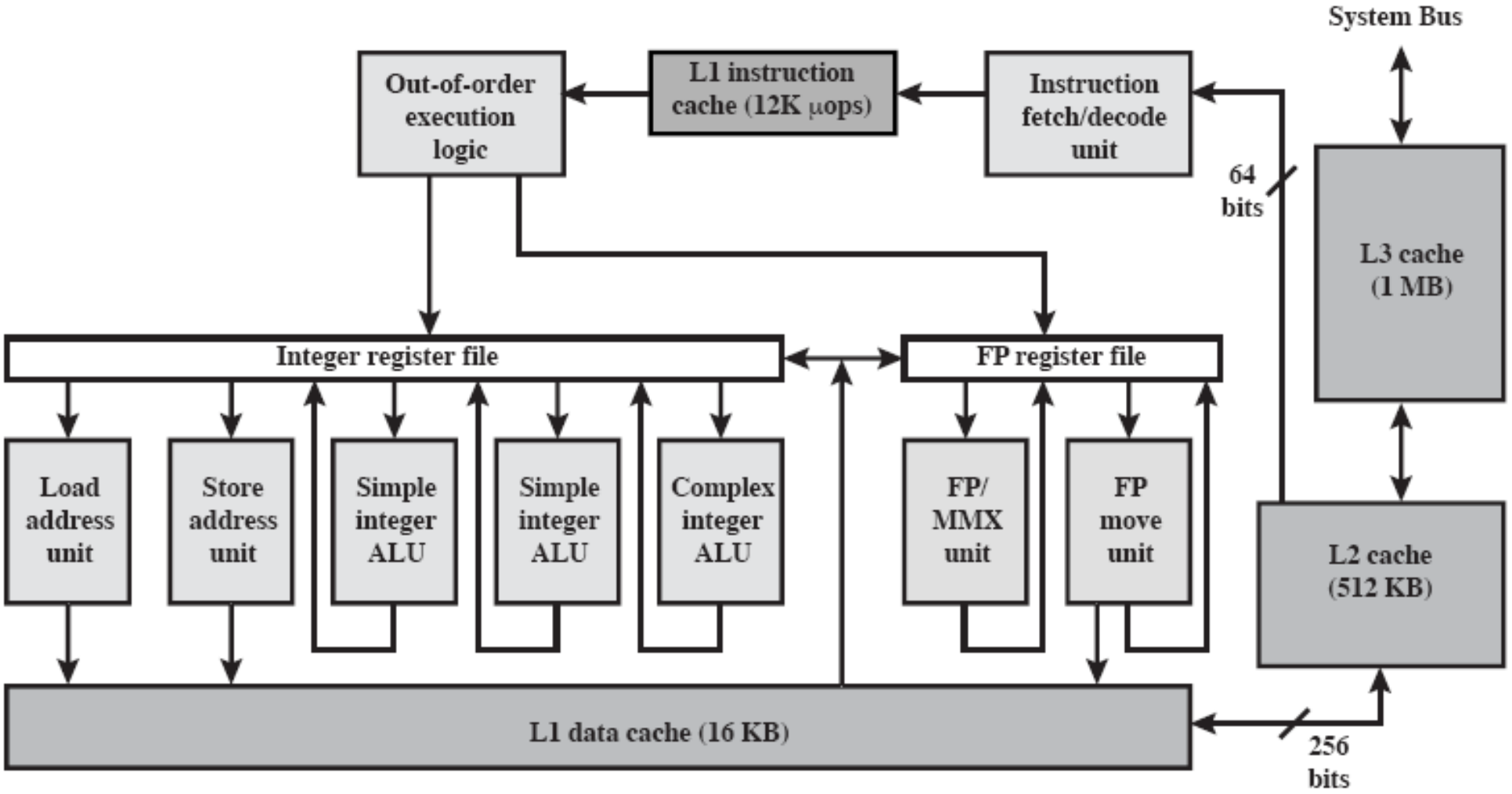
- Intel 80486 (introduced 1989)
  - a single on-chip cache of 8 Kbytes
  - line size: 16 bytes
  - 4-way set associative organization
- Intel Pentium (introduced 1993)
  - two on-chip caches, one for data and one for instructions
  - each cache: 8 Kbytes
  - line size: 32 bytes
  - 2-way set associative organization
- IBM PowerPC 620 (introduced 1995)
  - two on-chip caches, one for data and one for instructions
  - each cache: 32 Kbytes
  - line size: 64 bytes
  - 8-way set associative organization

# Cache Architecture Examples (Cont'd)

- Intel Itanium 2 (introduced 2002) – three levels of cache:

| | L1 | L2 | L3 |
|---|---|---|---|
| Contents | Split D and I | Unified D + I | Unified D + I |
| Size | 16 Kbytes each | 256 Kbytes | 3 Mbytes |
| Line size | 64 bytes | 128 bytes | 128 bytes |
| Associativity | 4 way | 8 way | 12 way |
| Access time | 1 cycle | 5-7 cycles | 14-17 cycles |
| Store policy | Write-through | Write-back | Write-back |

# Pentium 4 Block Diagram



# Pentium 4 Core Processor

- **Fetch/Decode unit**: fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the result in the L1 instruction cache.
- **Out-of-order execution logic**: schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream.
- **Execution units**: these units execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- **Memory subsystem**: this unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss, and to access the system IO resources.

# Pentium 4 Design Reasoning

- Decodes instructions into RISC like micro-ops before L1 cache
- Micro-ops fixed length
  - Enable the use of superscalar pipelining and scheduling
- Pentium instructions long & complex
- Performance improved by separating decoding from scheduling & pipelining
- Data cache is write back
  - Can be configured to write through
- L1 cache controlled by 2 bits in register
  - CD = cache disable
  - NW = not write through
  - 2 instructions to invalidate (flush) cache and to write back then invalidate
- L2 and L3 8-way set-associative
  - Line size 128 bytes

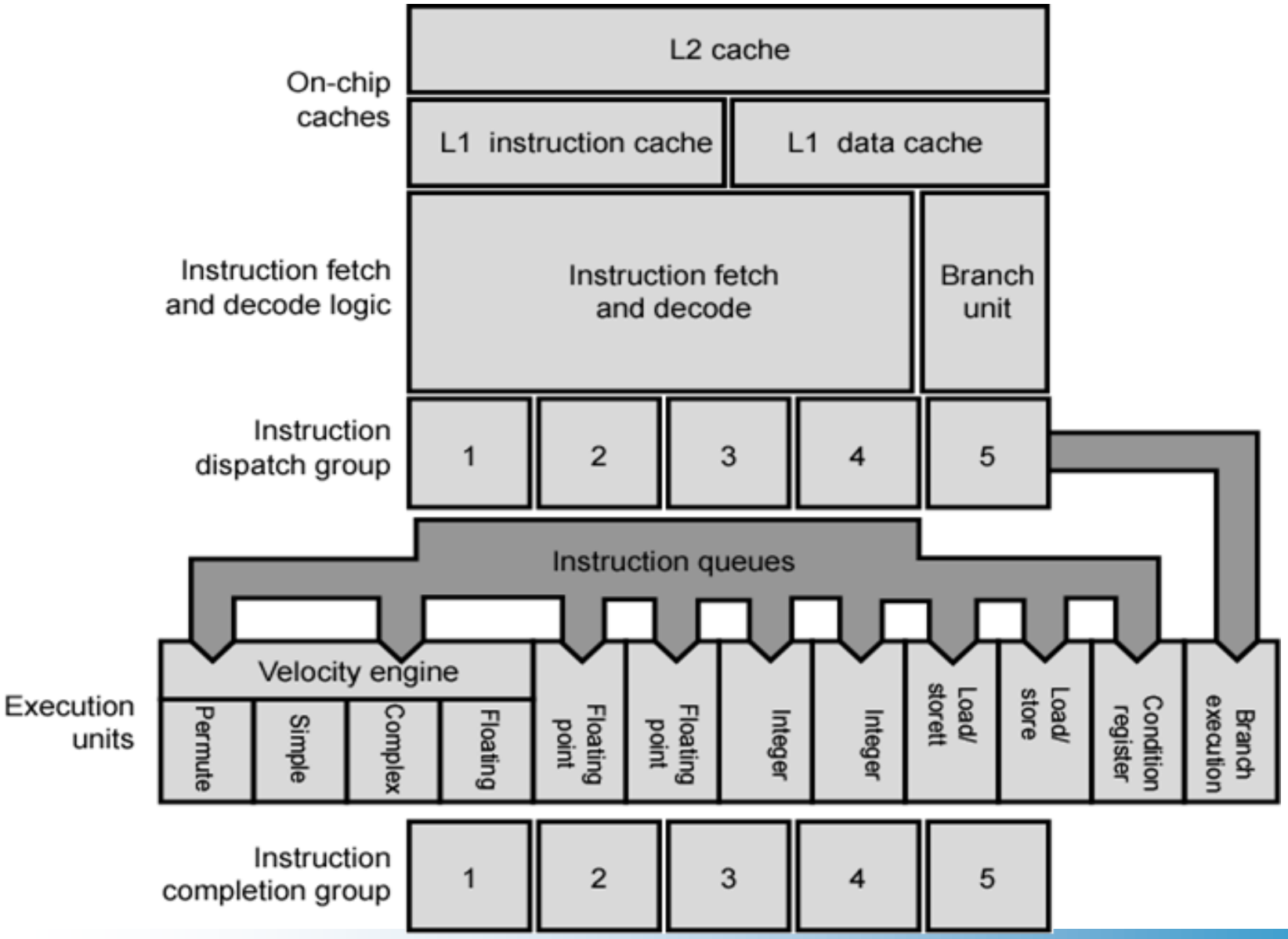
# PowerPC Cache Organization

- 601 – single 32kb 8 way set associative
- 603 – 16kb (2 x 8kb) two way set associative
- 604 – 32kb
- 620 – 64kb
- G3 & G4
  - 64kb L1 cache
    - 8 way set associative
  - 256k, 512k or 1M L2 cache
    - two way set associative
- G5
  - 32kB instruction cache
  - 64kB data cache

# PowerPC Internal L1 Caches

| Model | Size | Bytes/Line | Organization |
|---|---|---|---|
| PowerPC 601 | 1 32-KByte | 32 | 8-way set associative |
| PowerPC 603 | 2 8-KByte | 32 | 2-way set associative |
| PowerPC 604 | 2 16-KByte | 32 | 4-way set associative |
| PowerPC 620 | 2 32-KByte | 64 | 8-way set associative |
| PowerPC G3 | 2 32-KByte | 64 | 8-way set associative |
| PowerPC G4 | 2 32-KByte | 32 | 8-way set associative |
| PowerPC G5 | 1 32-Kbyte (Instruction)<br>1 64-Kbyte (Data) | 32 | 8-way set associative |

# PowerPC G5 Block Diagram



# 4 Questions for the Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

# Q1&Q2: Where can a block be placed/found?

| | # of sets | Blocks per set |
|---|---|---|
| Direct mapped | # of blocks in cache | 1 |
| Set associative | (# of blocks in cache)/ associativity | Associativity (typically 2 to 16) |
| Fully associative | 1 | # of blocks in cache |

| | Location method | # of comparisons |
|---|---|---|
| Direct mapped | Index | 1 |
| Set associative | Index the set; compare set's tags | Degree of associativity |
| Fully associative | Compare all blocks tags | # of blocks |

# Q3: Which block should be replaced on a miss?

- Easy for direct mapped – only one choice
- Set associative or fully associative
  - Random
  - LRU (Least Recently Used)

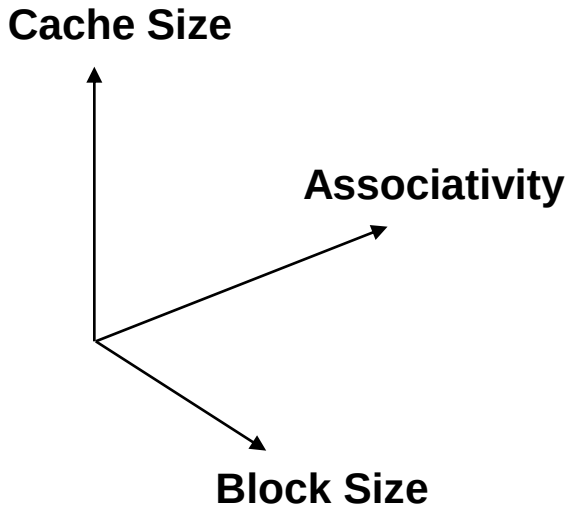
- For a 2-way set associative cache, random replacement has a miss rate about 1.1 times higher than LRU.
- LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly

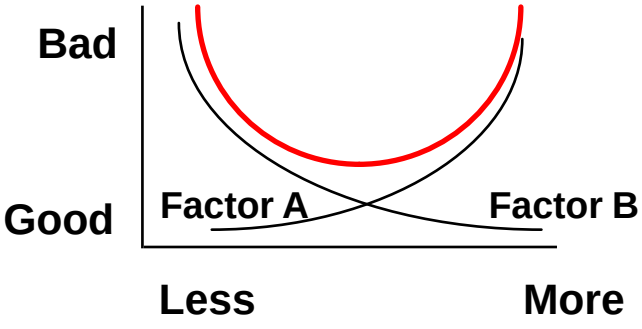
# Q4: What happens on a write?

- *Write-through* – The information is written to both the block in the cache and to the block in the next lower level of the memory hierarchy
  - Write-through is always combined with a write buffer so write waits to lower level memory can be eliminated (as long as the write buffer doesn't fill)
- *Write-back* – The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Need a dirty bit to keep track of whether the block is clean or dirty
- Pros and cons of each?
  - Write-through: read misses don't result in writes (so are simpler and cheaper)
  - Write-back: repeated writes require only one write to lower level

# Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins



# Cache Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three major categories of cache misses:
  - Compulsory misses: sad facts of life. Example: cold start misses
  - Conflict misses: increase cache size and/or associativity Nightmare Scenario: ping pong effect!
  - Capacity misses: increase cache size
- Cache design space
  - total size, block size, associativity (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)

# Measuring Cache Performance

- 2 techniques for improving cache performance
  - Reducing the miss rate by reducing the probability that 2 different memory blocks will contend for the same location
  - Reducing the miss penalty by adding an additional level to the hierarchy

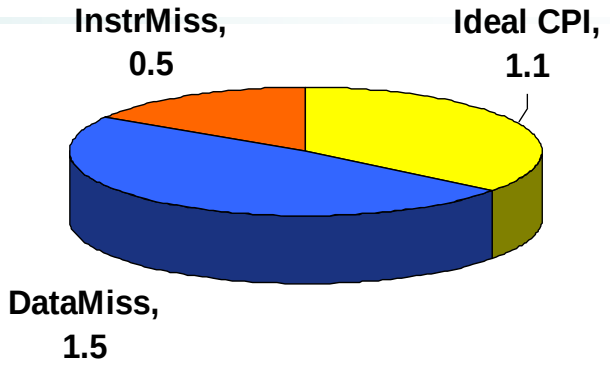
- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \quad \text{CPI} \quad \text{CC}$$

$$= \text{IC} \quad \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{stall}}} \quad \text{CC}$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)
  - Read-stall cycles = reads/program × read miss rate × read miss penalty
  - Write-stall cycles = (writes/program × write miss rate × write miss penalty) + write buffer stalls
- For write-through caches, we can simplify this to
  - Memory-stall cycles = miss rate × miss penalty

# Memory Performance Impact on System Performance




- Suppose a processor executes at
  - ideal CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

  and that 10% of data memory operations miss with a 50 cycle miss penalty

- CPI = ideal CPI + average stalls per instruction
  = 1.1(cycle) + ( 0.30 (datamemops/instr)
  x 0.10 (miss/datamemop) x 50 (cycle/miss) )
  = 1.1 cycle + 1.5 cycle = 2.6

- So 58% of the time the processor is stalled waiting for memory!
- A 1% instruction miss rate would add an additional 0.5 to the CPI!

# Impacts of Cache Performance

- Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI)
  - The memory speed is unlikely to improve as fast as processor cycle time. When calculating $CPI_{stall}$, the cache miss penalty is measured in *processor* clock cycles needed to handle a miss
  - The lower the $CPI_{ideal}$, the more pronounced the impact of stalls

- A processor with a $CPI_{ideal}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I$ and 4% D$ miss rates

$$\text{Memory-stall cycles} = 2\% \quad 100 + 36\% \quad 4\% \quad 100 = 3.44$$

$$\text{So} \quad CPI_{stalls} = 2 + 3.44 = 5.44$$

- What if the $CPI_{ideal}$ is reduced to 1? 0.5? 0.25?
- What if the processor clock rate is doubled (doubling the miss penalty)?

# Reducing Cache Miss Rates #1

1. Allow more flexible block placement

- In a direct mapped cache a memory block maps to exactly one cache block
- At the other extreme, could allow a memory block to be mapped to any cache block – fully associative cache

- A compromise is to divide the cache into sets each of which consists of n "ways" (n-way set associative). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

  (block address) modulo (# sets in the cache)

# Reducing Cache Miss Rates #2

2. Use multiple levels of caches

- With advancing technology have more than enough room on the die for bigger L1 caches *or* for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache
- For our example, $CPI_{ideal}$ of 2, 100 cycle miss penalty (to main memory), 36% load/stores, a 2% (4%) L1I$ (D$) miss rate, add a UL2$ that has a 25 cycle miss penalty and a 0.5% miss rate

$$CPI_{stalls} = 2 + .02 \ 25 + .36 \ .04 \ 25 + .005 \ 100 + .36 \ .005 \ 100 = 3.54$$

(as compared to 5.44 with no L2$)

# Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

# Key Cache Design Parameters

| | L1 typical | L2 typical |
|---|---|---|
| Total size (blocks) | 250 to 2000 | 4000 to 250,000 |
| Total size (KB) | 16 to 64 | 500 to 8000 |
| Block size (B) | 32 to 64 | 32 to 128 |
| Miss penalty (clocks) | 10 to 25 | 100 to 1000 |
| Miss rates (global for L2) | 2% to 5% | 0.1% to 2% |

# Two Machines' Cache Parameters

| | Intel P4 | AMD Opteron |
|---|---|---|
| L1 organization | Split I$ and D$ | Split I$ and D$ |
| L1 cache size | 8KB for D$, 96KB for trace cache (~I$) | 64KB for each of I$ and D$ |
| L1 block size | 64 bytes | 64 bytes |
| L1 associativity | 4-way set assoc. | 2-way set assoc. |
| L1 replacement | ~ LRU | LRU |
| L1 write policy | write-through | write-back |
| L2 organization | Unified | Unified |
| L2 cache size | 512KB | 1024KB (1MB) |
| L2 block size | 128 bytes | 64 bytes |
| L2 associativity | 8-way set assoc. | 16-way set assoc. |
| L2 replacement | ~LRU | ~LRU |
| L2 write policy | write-back | write-back |

# Improving Cache Performance

0. Reduce the time to hit in the cache
   - smaller cache
   - direct mapped cache
   - smaller blocks
   - for writes
     - no write allocate – no “hit” on cache, just write to write buffer
     - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks

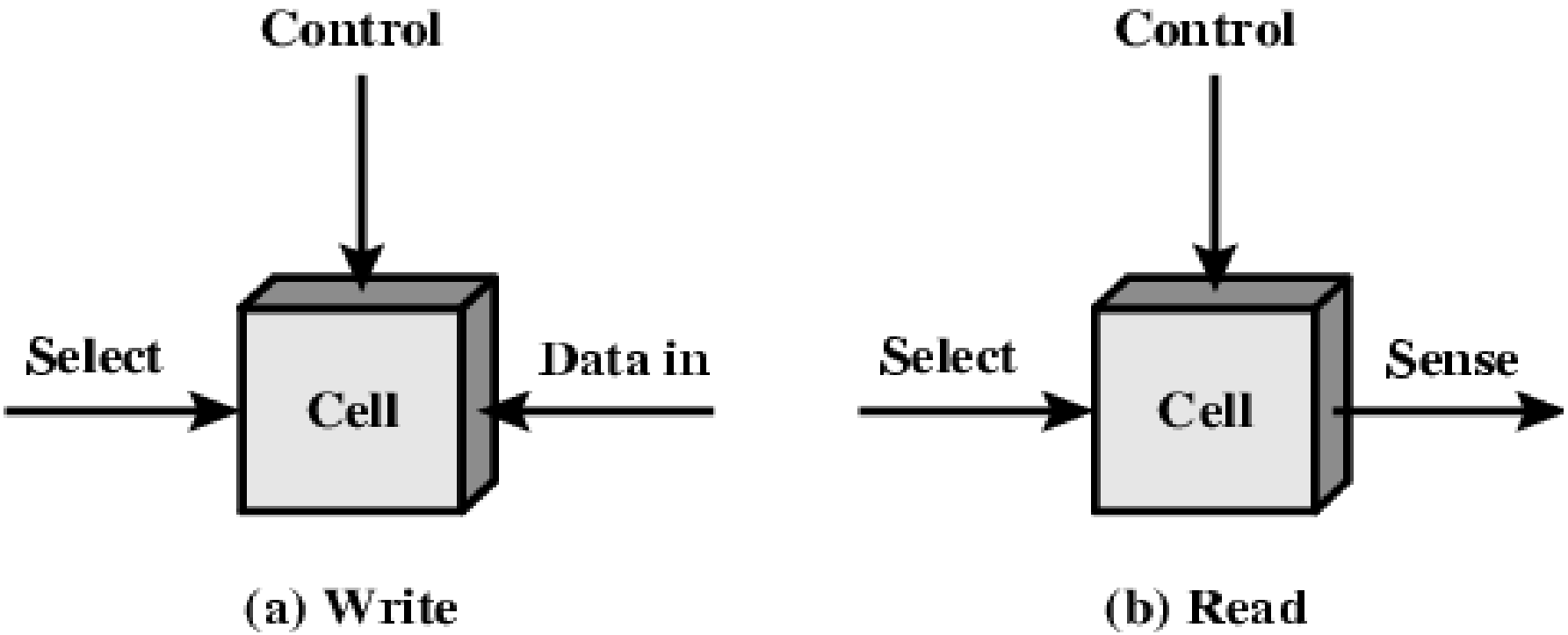
# Improving Cache Performance

2. Reduce the miss penalty
   - smaller blocks
   - use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
   - check write buffer (and/or victim cache) on read miss – may get lucky
   - for large blocks fetch critical word first
   - use multiple cache levels – L2 cache not tied to CPU clock rate
   - faster backing store/improved memory bandwidth
     - wider buses
     - memory interleaving, page mode DRAMs

# Lecture 2

- Basic components
- Memory hierarchy
- Cache memory
- Internal Memory
- External Memory
- Virtual Memory

# Memory Cell Operation



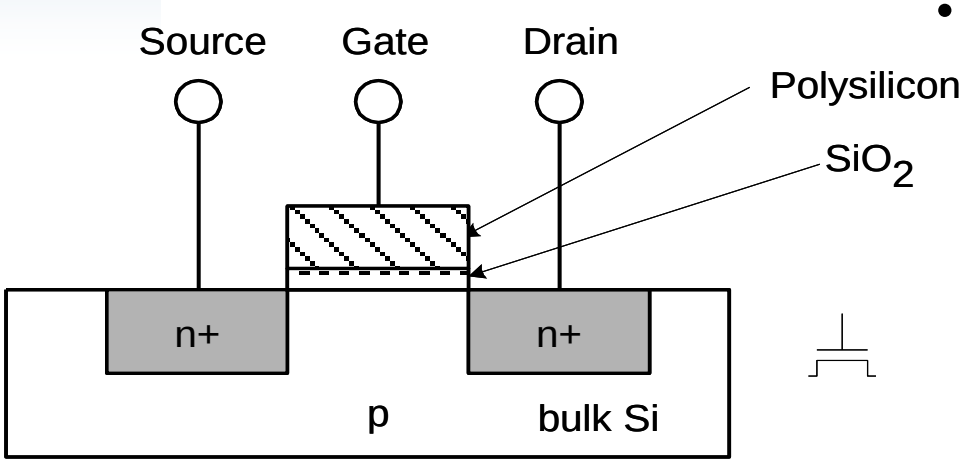
# Semiconductor Memory Types

| Memory Type | Category | Erasure | Write Mechanism | Volatility |
|---|---|---|---|---|
| Random-access memory (RAM) | Read-write memory | Electrically, byte-level | Electrically | Volatile |
| Read-only memory (ROM) | Read-only memory | Not possible | Masks | Nonvolatile |
| Programmable ROM (PROM) | | | Electrically | |
| Erasable PROM (EPROM) | Read-mostly memory | UV light, chip-level | | |
| Electrically Erasable PROM (EEPROM) | | Electrically, byte-level | | |
| Flash memory | | Electrically, block-level | | |

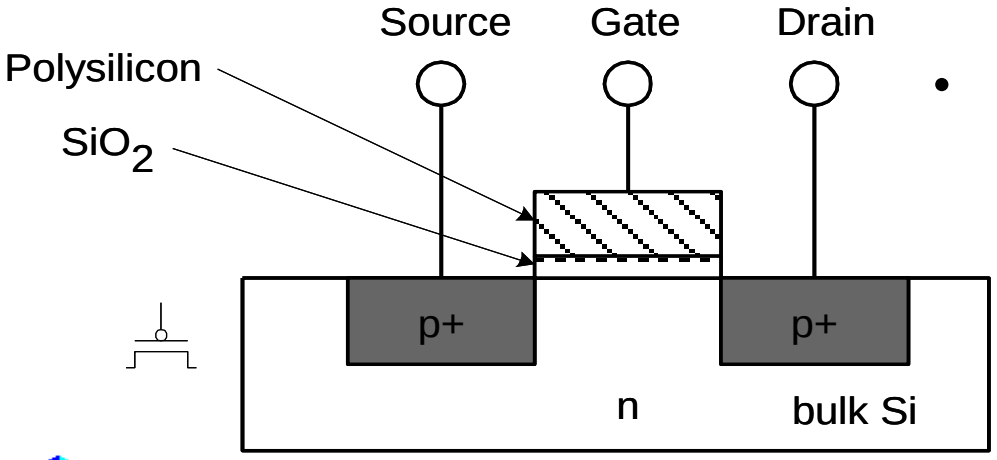
# Random-access memory (RAM)

- Misnamed as all semiconductor memory is random access
- Read/Write
- Volatile
- Temporary storage
- Static or dynamic

# A Transistor



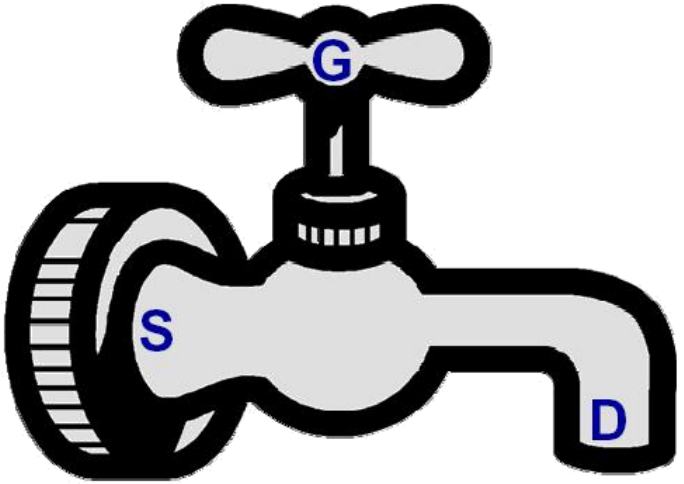

- nMOS
  - Body is commonly tied to ground (0 V)
  - When the gate is at a low voltage:
    - P-type body is at low voltage
    - Source-body and drain-body diodes are OFF
    - No current flows, transistor is OFF
  - When the gate is at a high voltage:
    - Positive charge on gate of MOS capacitor
    - Negative charge attracted to body
    - Inverts a channel under gate to n-type
    - Now current can flow through n-type silicon from source through channel to drain, transistor is ON

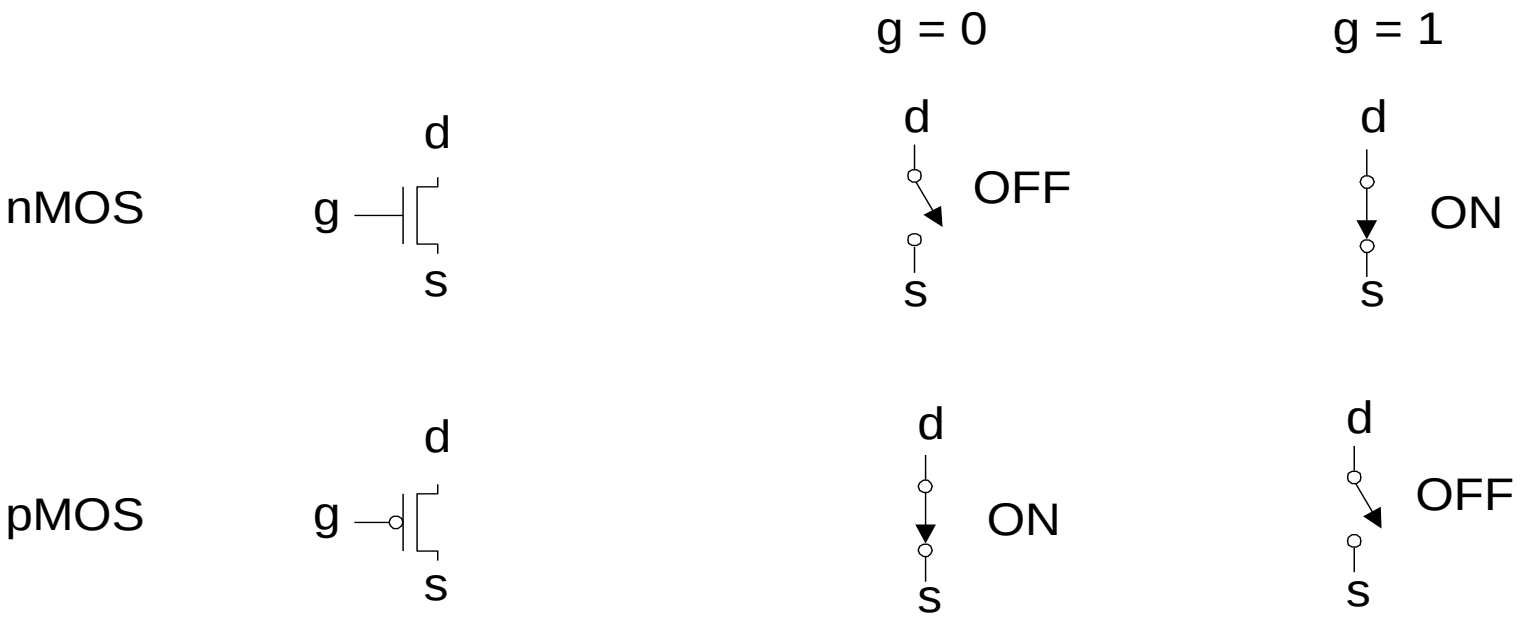



- pMOS: similar, but doping and voltages reversed
  - Body tied to high voltage ($V_{DD}$)
  - Gate low: transistor ON
  - Gate high: transistor OFF
  - Bubble indicates inverted behavior

# Transistors as Switches

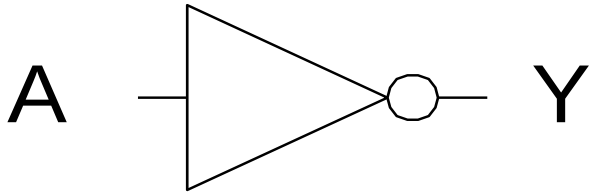
- We can view MOS transistors as electrically controlled switches
- Voltage at gate controls path from source to drain

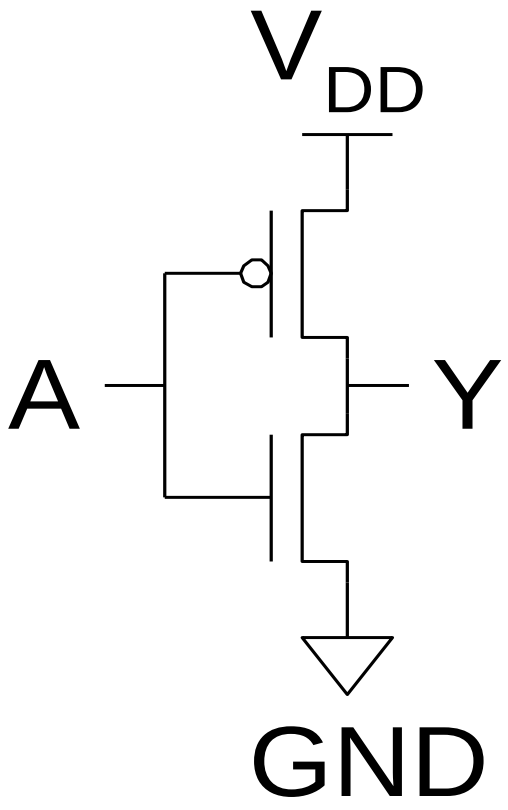


# CMOS Inverter

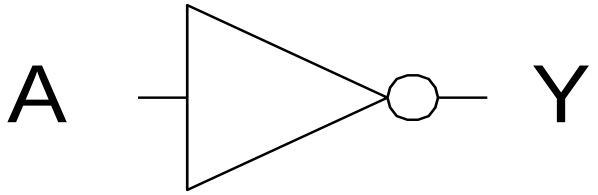
| A | Y |
|---|---|
| 0 | |
| 1 | |

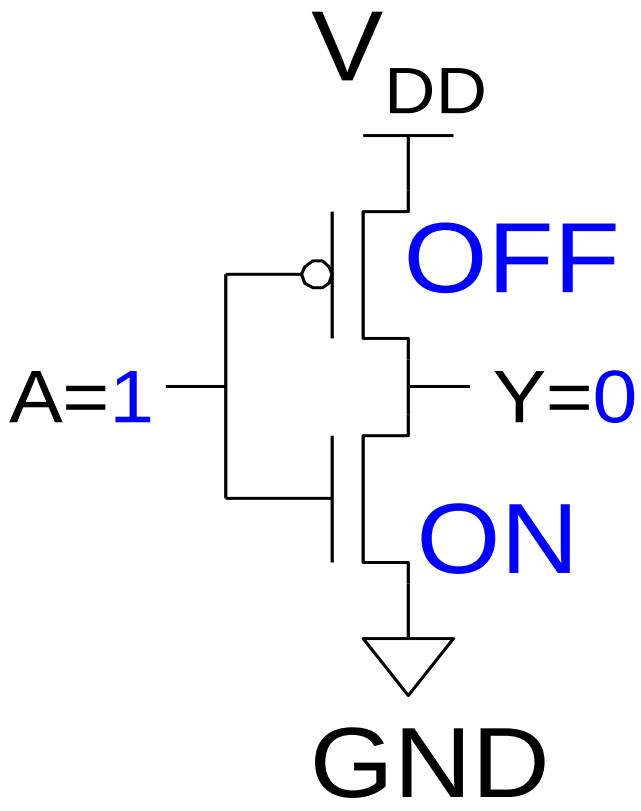


# CMOS Inverter

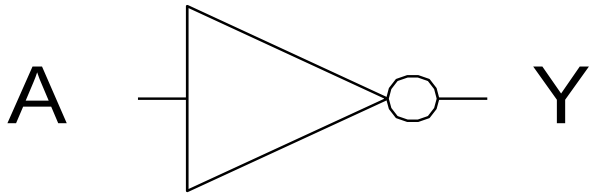
| A | Y |
|---|---|
| 0 | |
| **1** | **0** |

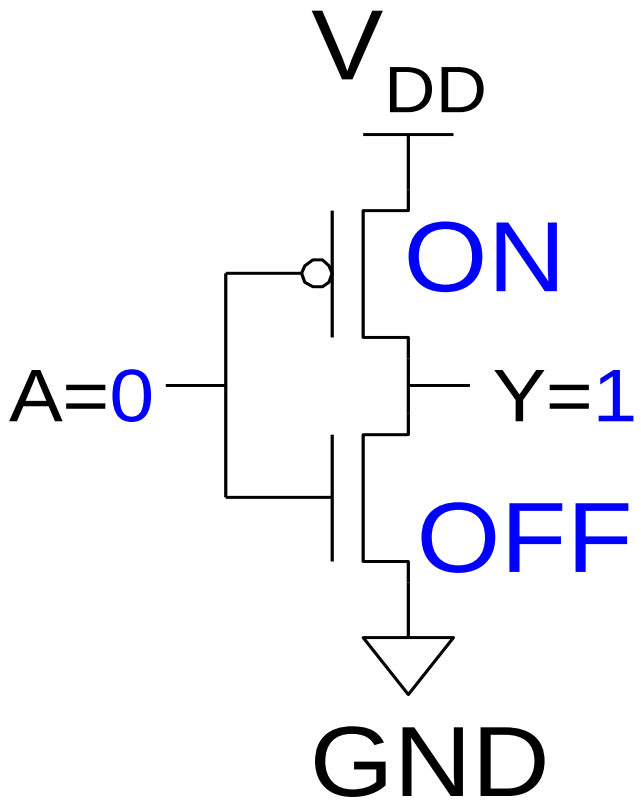


# CMOS Inverter

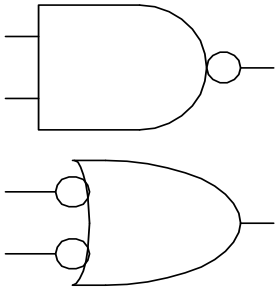
| A | Y |
|---|---|
| **0** | **1** |
| 1 | 0 |

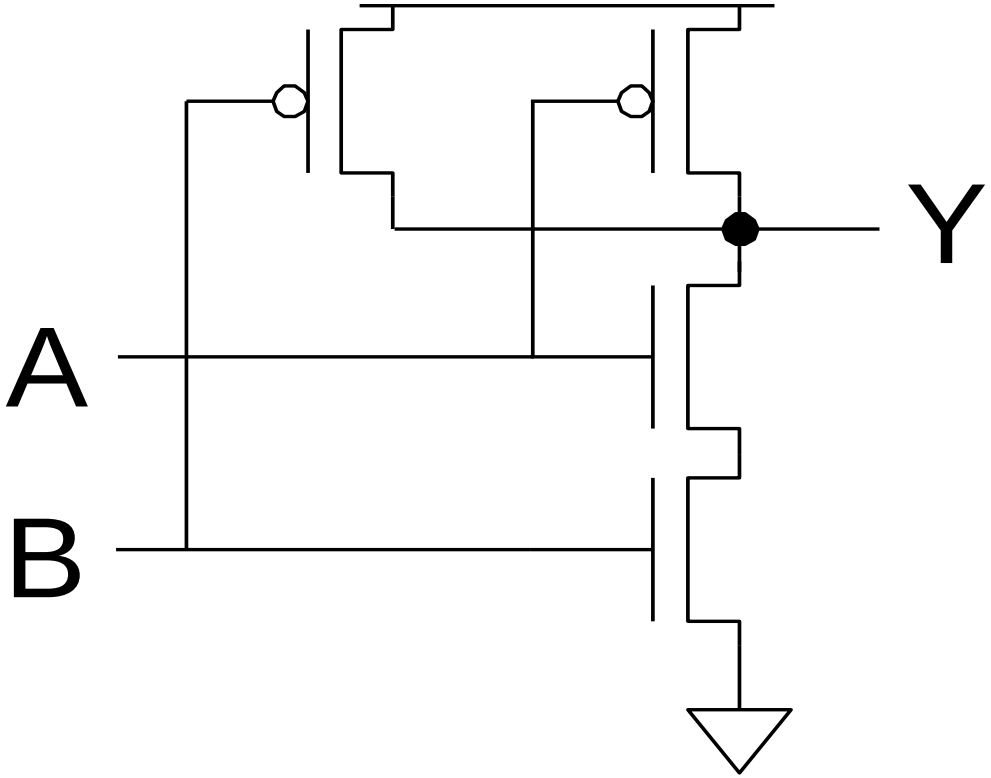


# CMOS NAND Gate

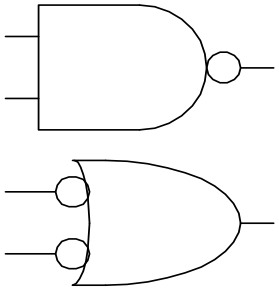
| A | B | Y |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

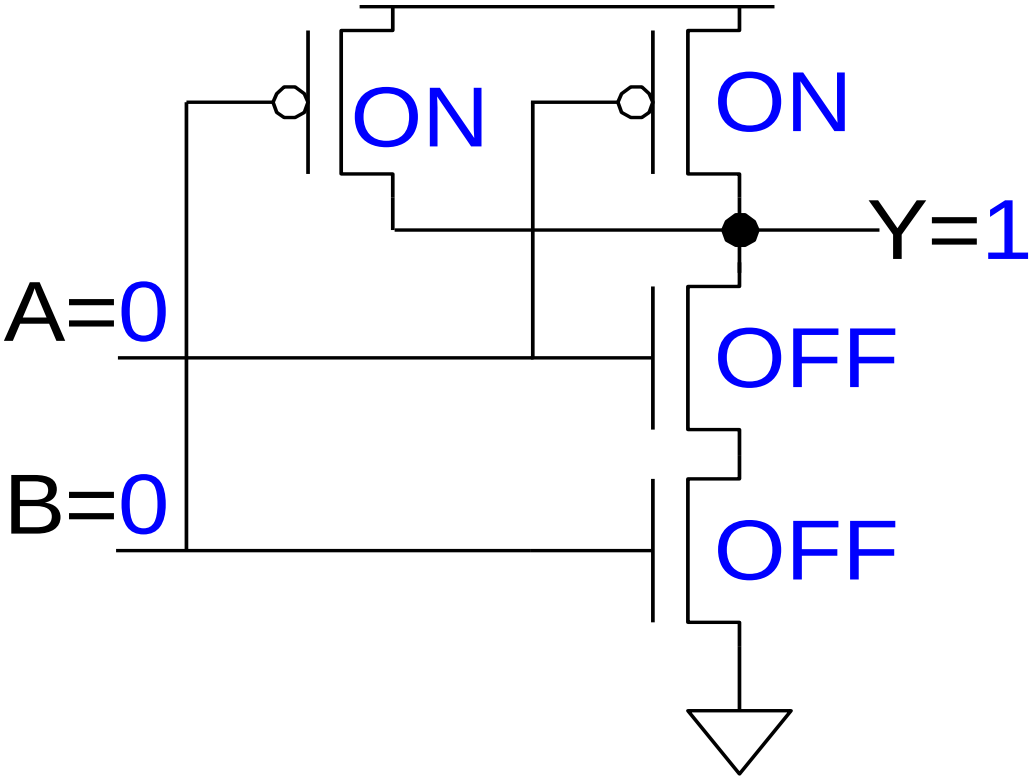




# CMOS NAND Gate

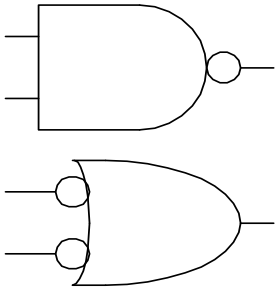
| A | B | Y |
|---|---|---|
| **0** | **0** | **1** |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

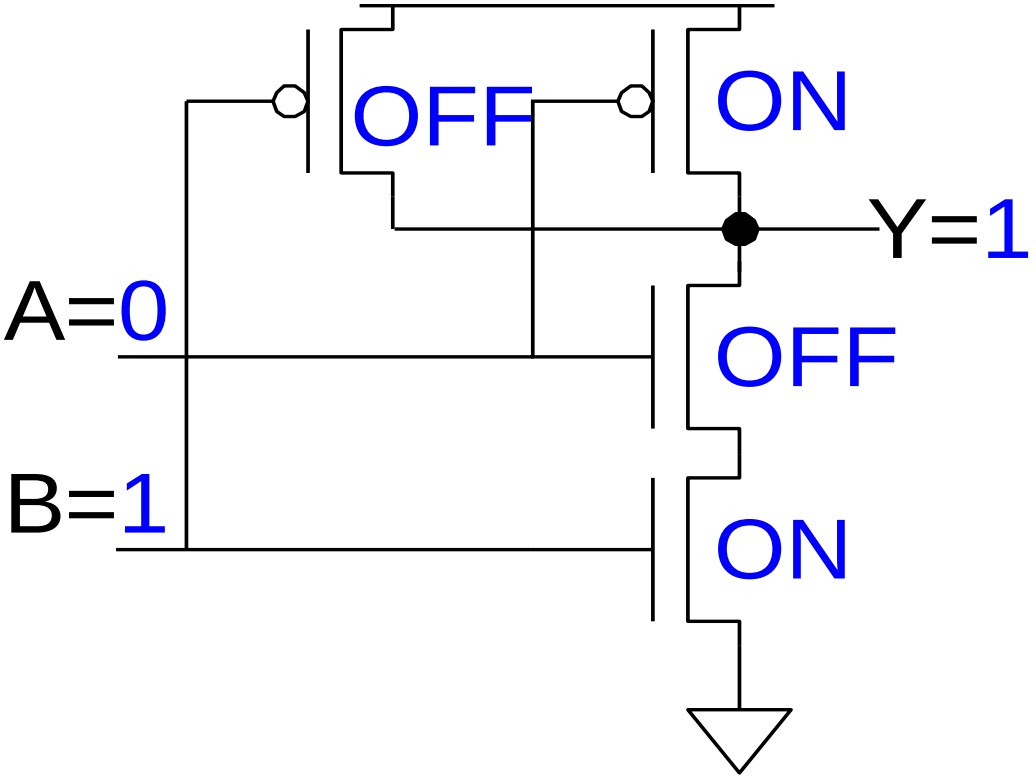




# CMOS NAND Gate

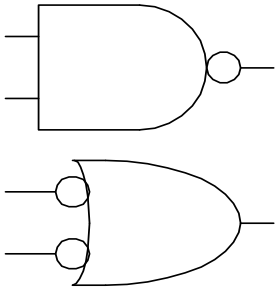
| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| **0** | **1** | **1** |
| 1 | 0 | |
| 1 | 1 | |

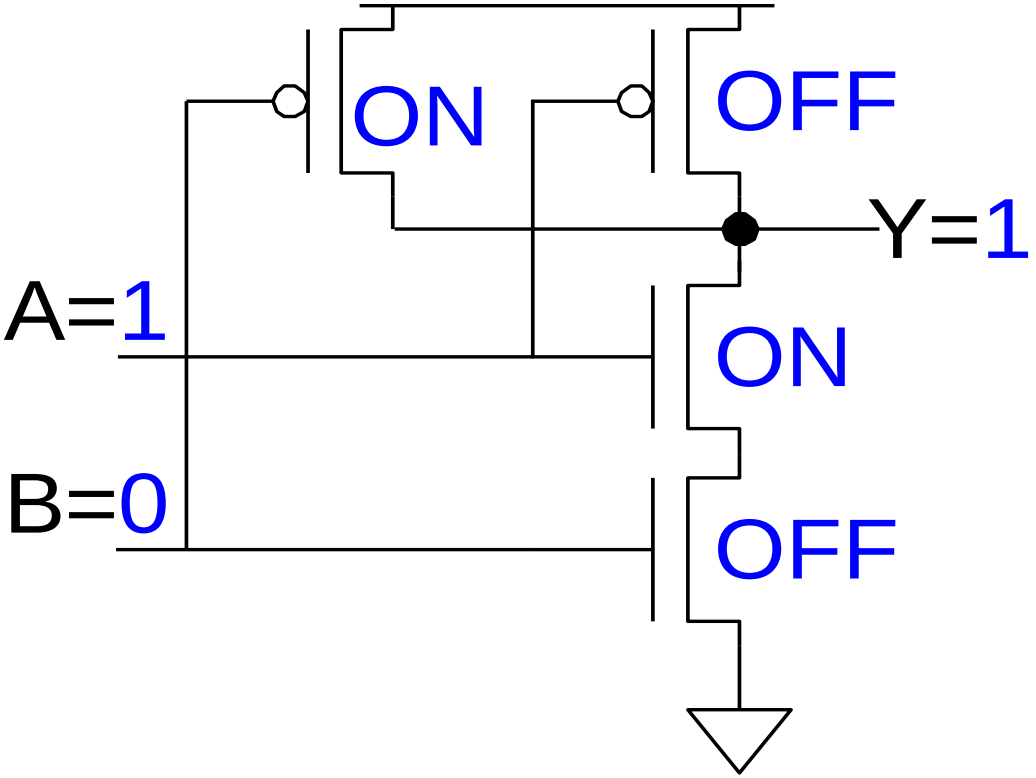




# CMOS NAND Gate

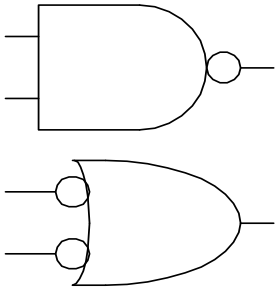
| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| **1** | **0** | **1** |
| 1 | 1 | |

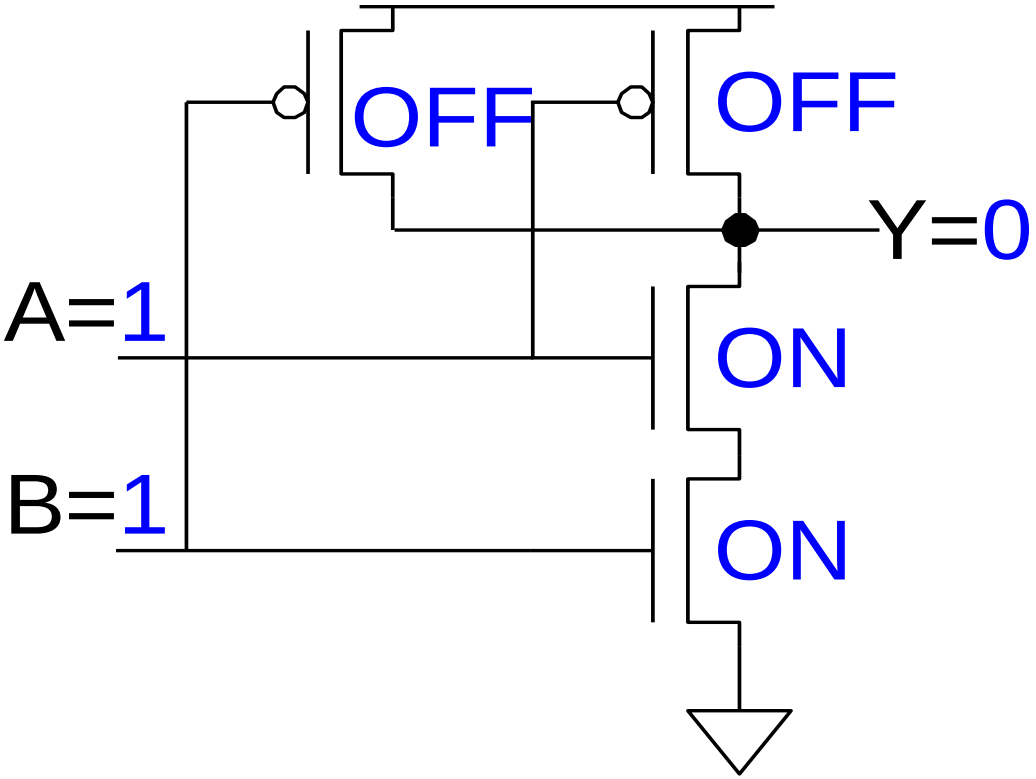




# CMOS NAND Gate

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| **1** | **1** | **0** |



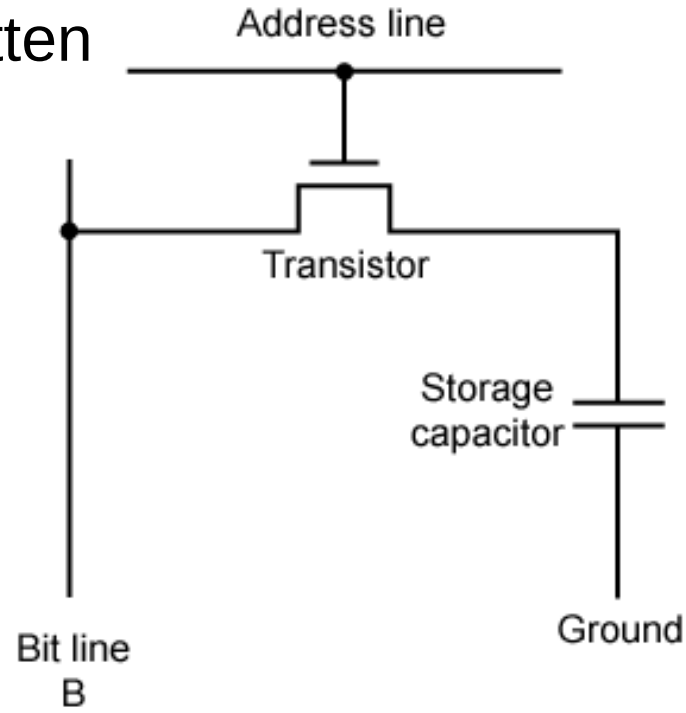


# Dynamic RAM

- Bits stored as charge in capacitors
- Charges leak
- Need refreshing even when powered
- Simpler construction
- Smaller per bit
- Less expensive
- Need refresh circuits
- Slower
- Main memory
- Essentially analogue
  - Level of charge determines value

# DRAM Operation

- Address line active when bit read or written
  - Transistor switch closed (current flows)
- Write
  - Voltage to bit line
    - High for 1, low for 0
  - Then signal address line
    - Transfers charge to capacitor
- Read
  - Address line selected
    - transistor turns on
  - Charge from capacitor fed via bit line to sense amplifier
    - Compares with reference value to determine 0 or 1
  - Capacitor charge must be restored

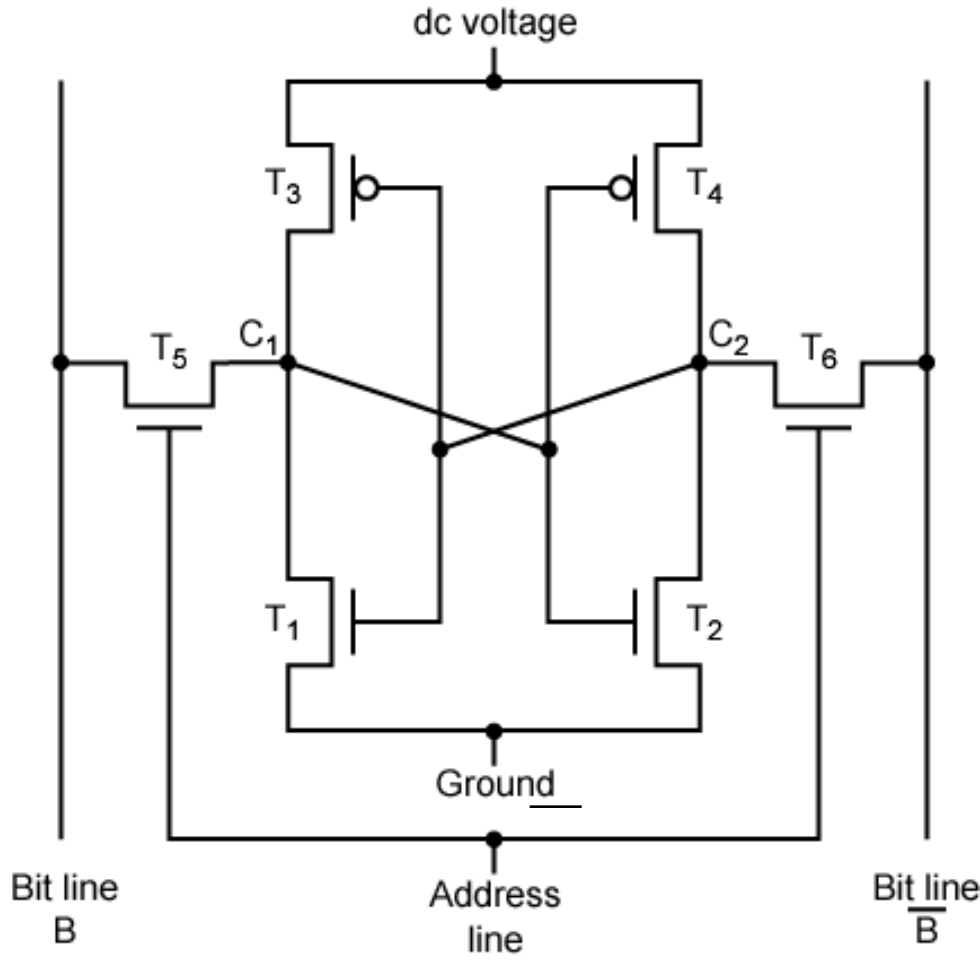


# Static RAM

- Bits stored as on/off switches
- No charges to leak
- No refreshing needed when powered
- More complex construction
- Larger per bit
- More expensive
- Does not need refresh circuits
- Faster
- Cache
- Digital
  - Uses flip-flops

# Static RAM Operation

- Transistor arrangement gives stable logic state
- State 1
  - $C_1$ high, $C_2$ low
  - $T_1$ $T_4$ off, $T_2$ $T_3$ on
- State 0
  - $C_2$ high, $C_1$ low
  - $T_2$ $T_3$ off, $T_1$ $T_4$ on
- Address line transistors $T_5$ $T_6$ is switch
- Write – apply value to B & compliment to B
- Read – value is on line B



# SRAM vs DRAM

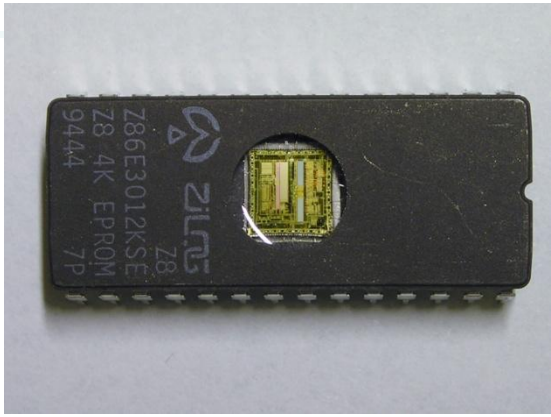
- Both volatile
  - Power needed to preserve data
- Dynamic cell
  - Simpler to build, smaller
  - More dense
  - Less expensive
  - Needs refresh
  - Larger memory units
- Static
  - Faster
  - Cache

# Read Only Memory (ROM)

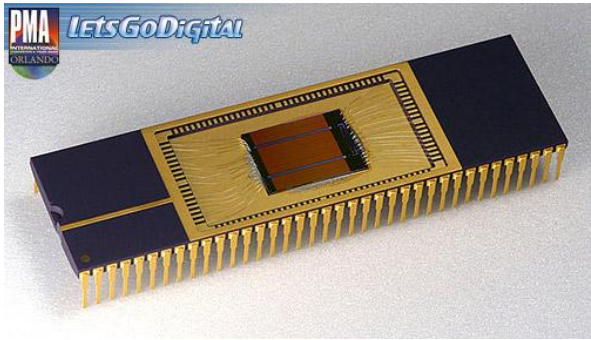
- Permanent storage
  - Nonvolatile
- Microprogramming
- Library subroutines
- Systems programs (BIOS)
- Function tables

# Types of ROM

- Written during manufacture
  - Very expensive for small runs
- Programmable (once)
  - PROM
  - Needs special equipment to program
- Read “mostly”
  - Erasable Programmable (EPROM)
    - Erased by UV
  - Electrically Erasable (EEPROM)
    - Takes much longer to write than read
  - Flash memory
    - Erase whole memory electrically



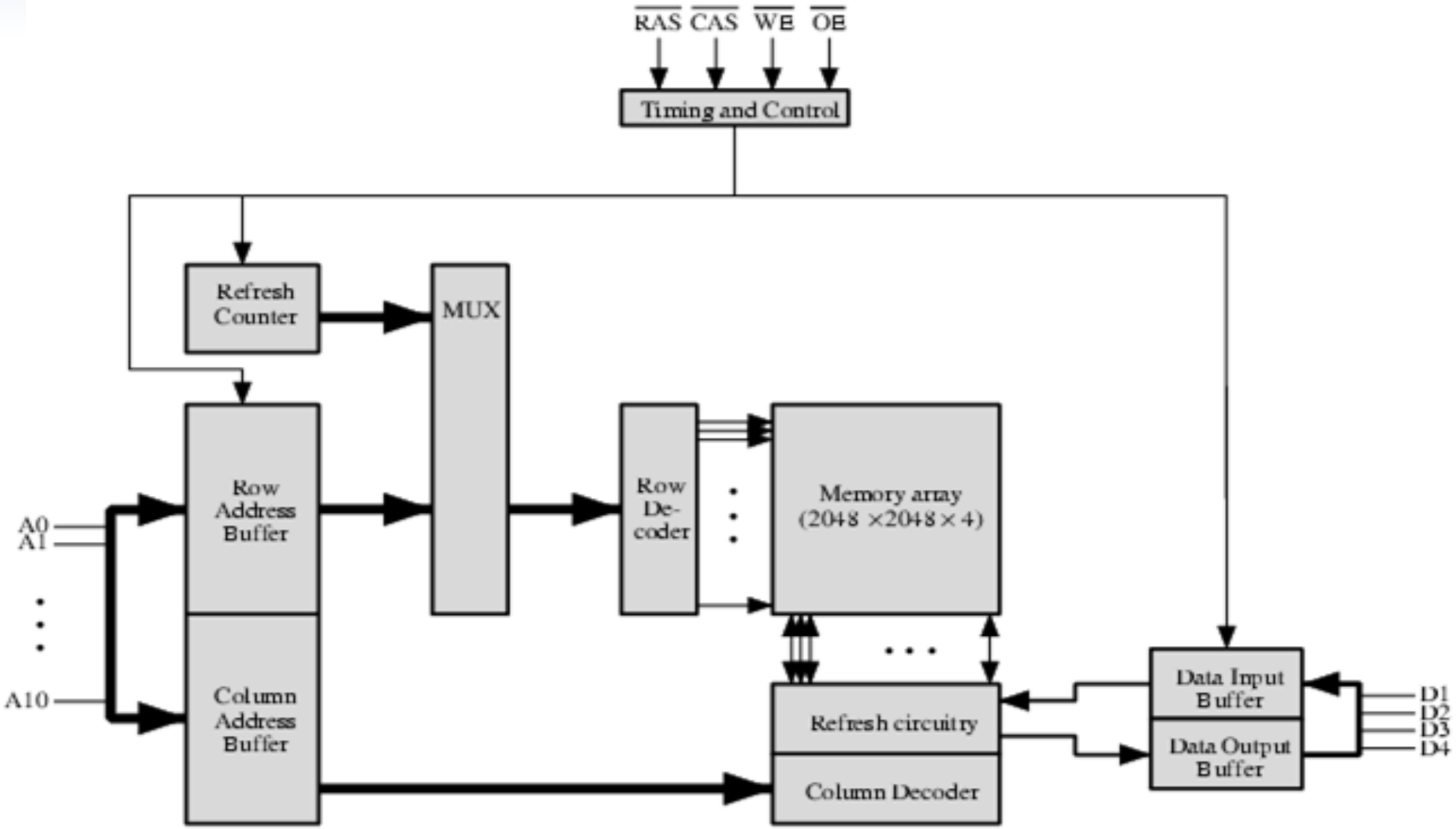




# Organisation in detail

- A 16Mbit chip can be organised as 1M of 16-bit words
- A bit per chip system has 16 lots of 1Mbit chip with bit 1 of each word in chip 1 and so on
- A 16Mbit chip can be organised as a 2048 x 2048 x 4bit array
  - Reduces number of address pins
    - Multiplex row address and column address
    - 11 pins to address ($2^{11}$=2048)
    - Adding one more pin doubles range of values so x4 capacity

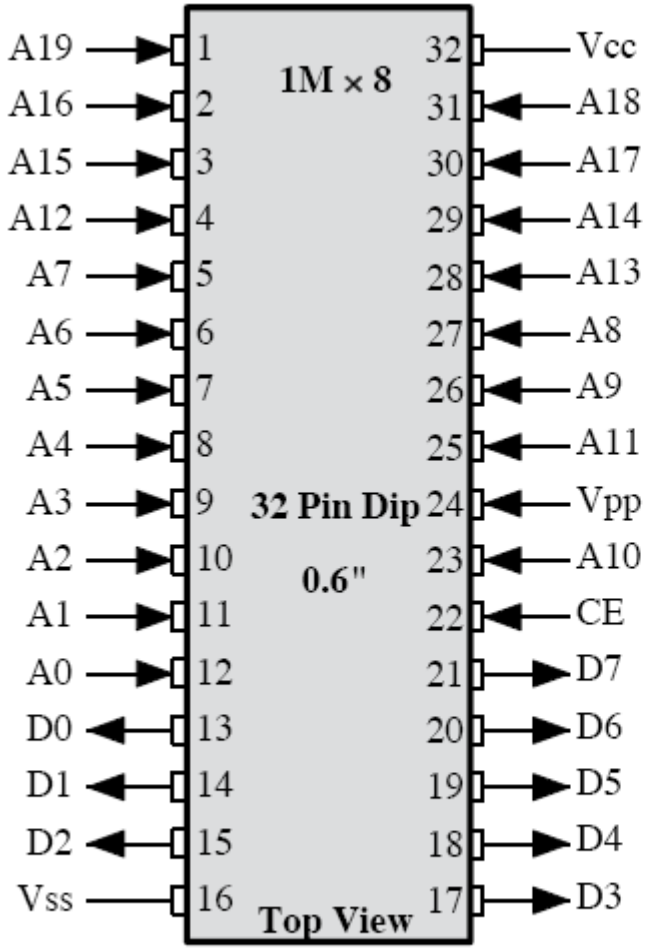
# Typical 16 Mb DRAM (4M x 4)



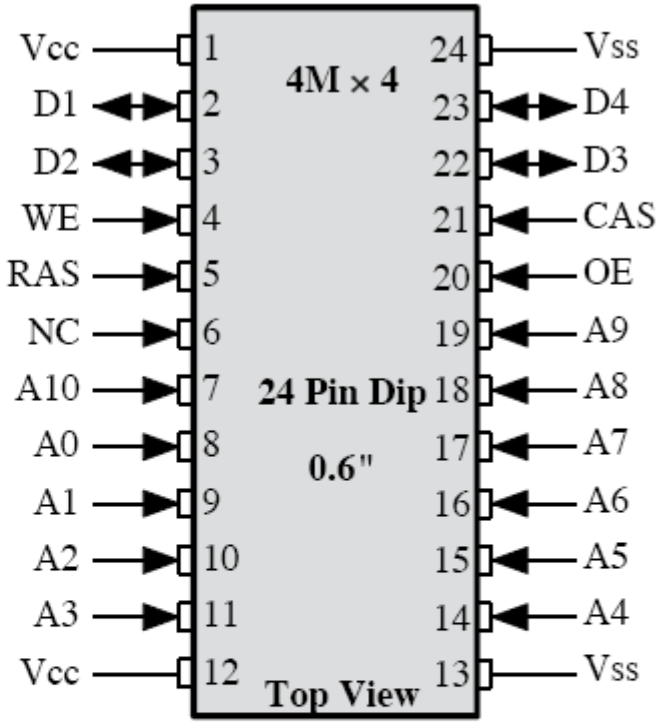
# Refreshing

- Refresh circuit included on chip
- Disable chip
- Count through rows
- Read & Write back
- Takes time
- Slows down apparent performance

# Packaging

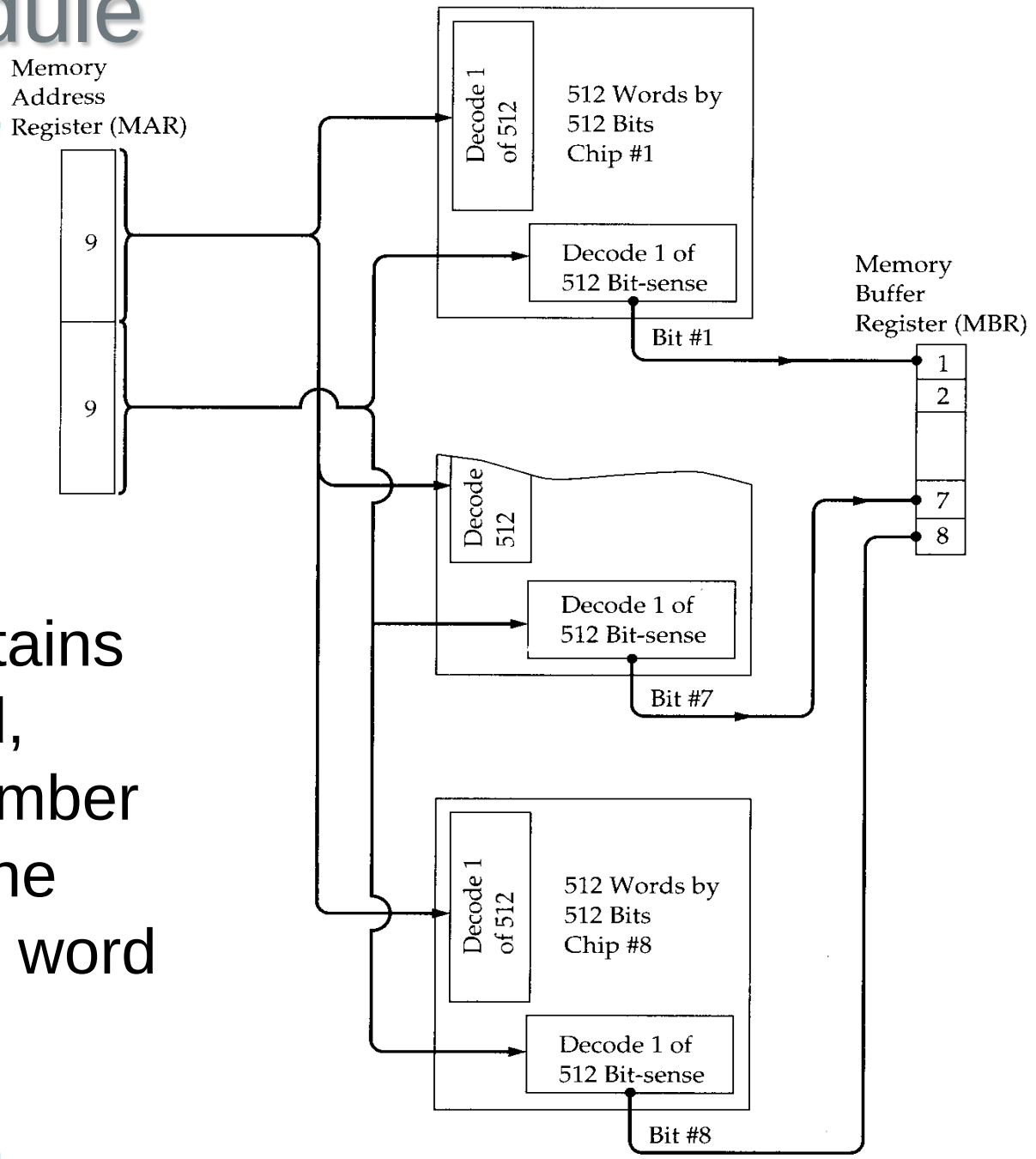



(a) 8 Mbit EPROM




(b) 16 Mbit DRAM

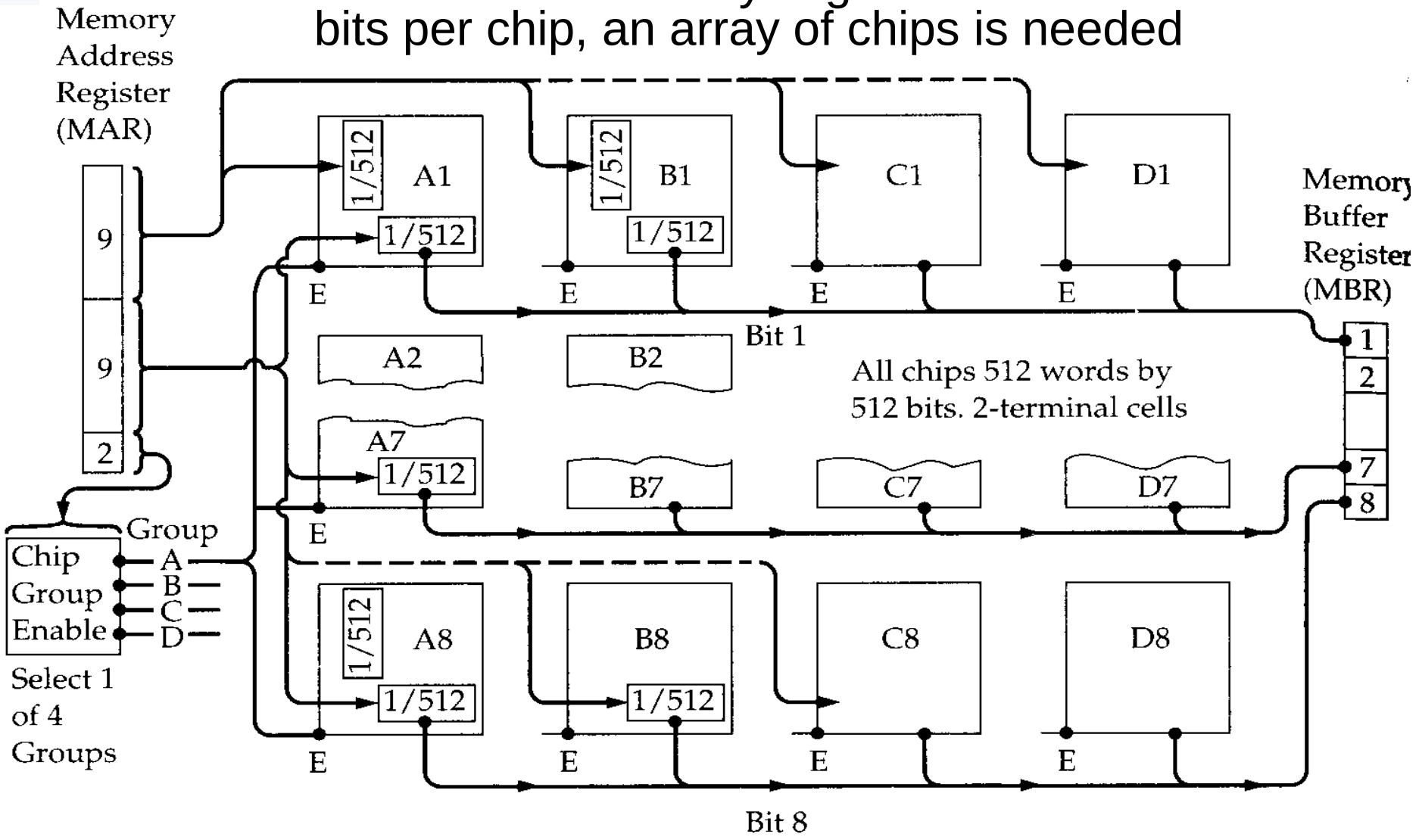
# 256KByte Module Organisation




- If a RAM chip contains only 1 bit per word, need at least a number of chips equal to the number of bits per word

# 1MByte Module Organisation

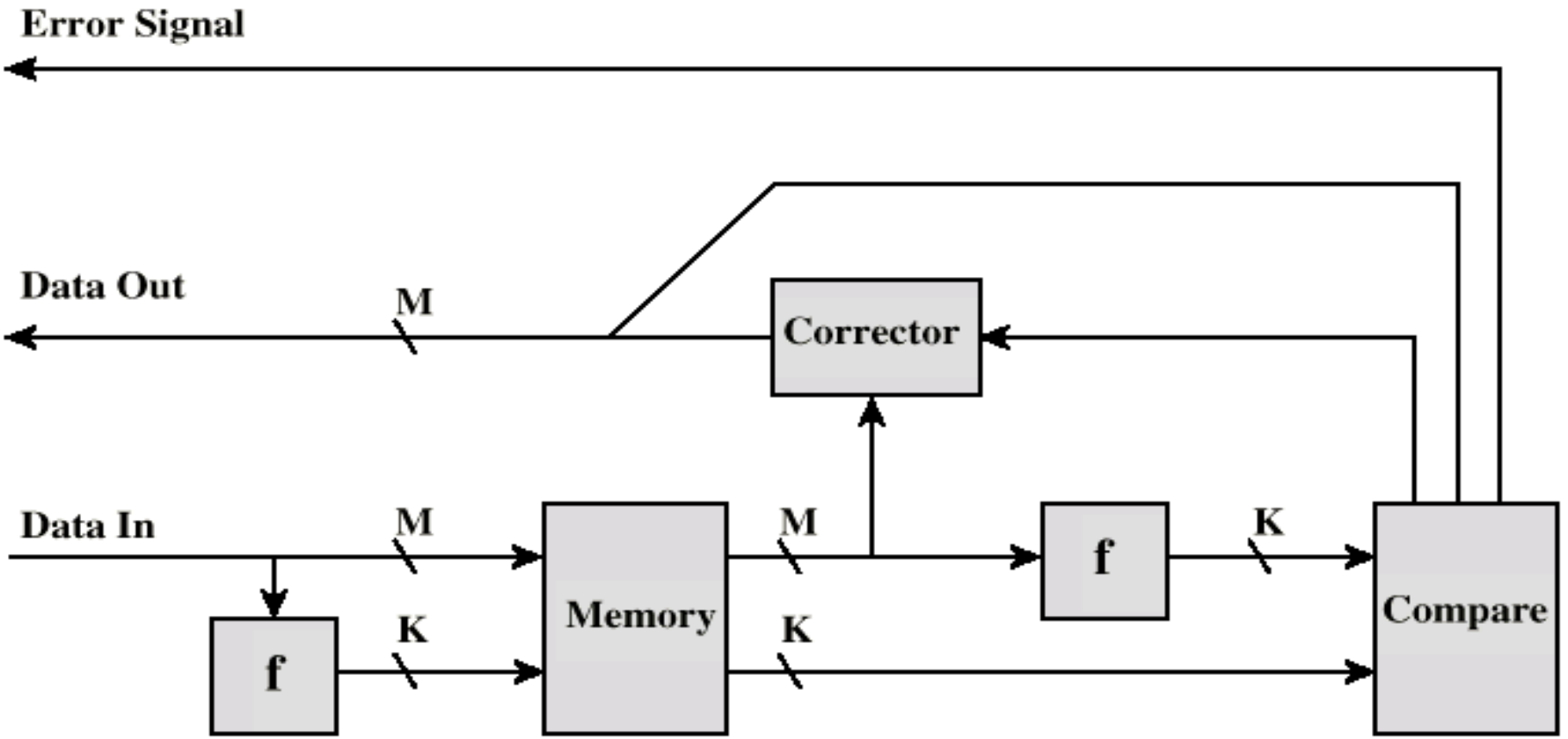
- If the size of memory is greater than the number of bits per chip, an array of chips is needed



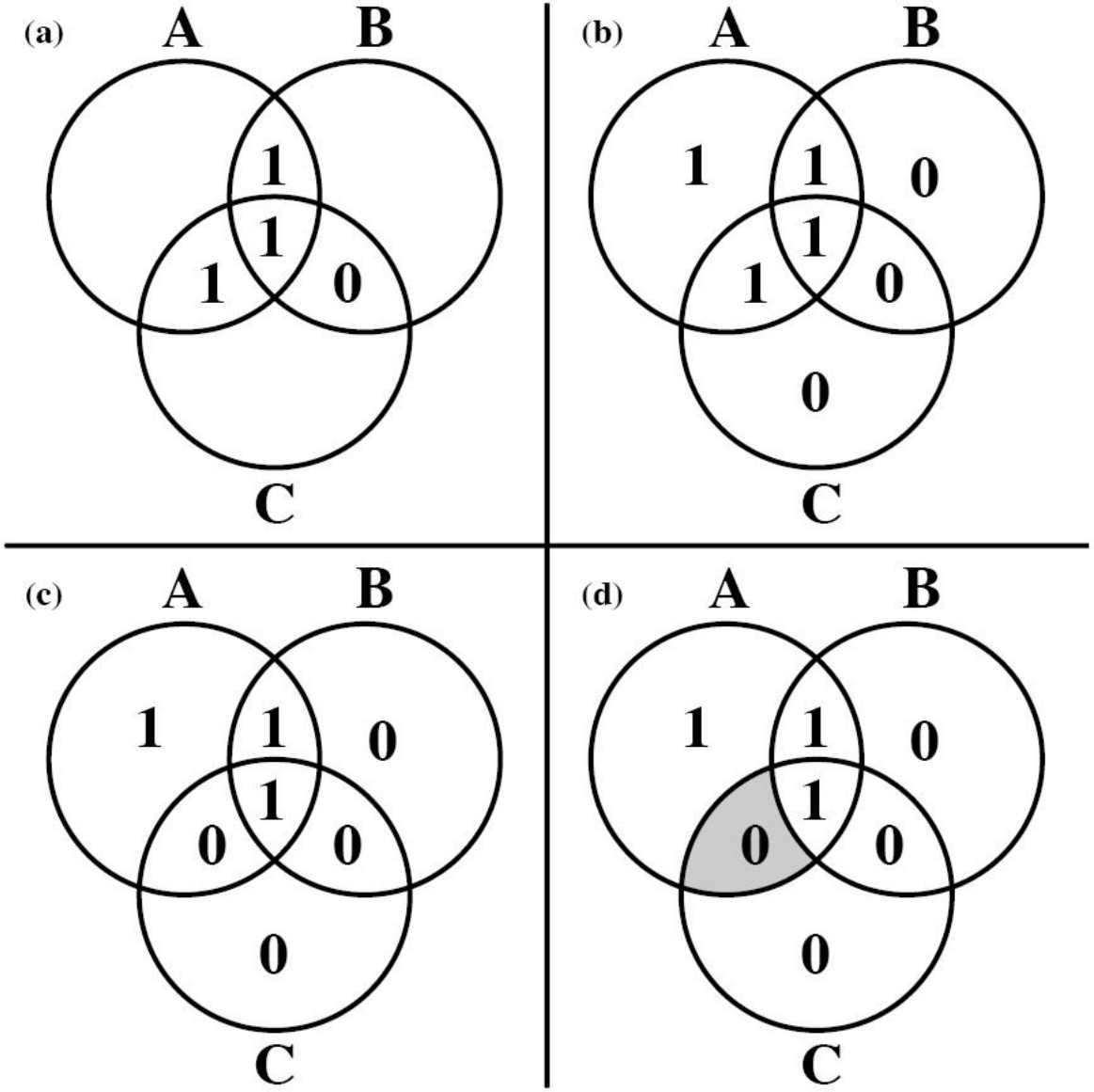
# Error Correction

- Semiconductor memory is subject to errors
- Hard Failure
  - Permanent physical defect
  - Stuck at 0/1 or switch erratically between 0 and 1
- Soft Error
  - Random, non-destructive
  - No permanent damage to memory
- Detected using Hamming error correcting code

# Error Correcting Code Function



# Venn Diagrams



# Detect and Correct Single-bit Errors in 8-bit Words

- Syndrome word: output of the compare process (implemented by XOR)
  - Each bit of the syndrome is 0/1 according to if there is/isnot a match in that bit position
- K bit syndrome word: range between 0 and $2^K - 1$
  - 0 indicates that no error was detected
  - $2^K - 1$ values indicate which bit was in error
- Error can be found in any of the M data bits or K check bits
  - $2^K - 1 \geq M + K$
  - For a word of 8 data bits (M=8), $K \geq 4$

| | Single-Error Correction | | Single-Error Correction/ Double-Error Detection | |
|---|---|---|---|---|
| Data Bits | Check Bits | % Increase | Check Bits | % Increase |
| 8 | 4 | 50 | 5 | 62.5 |
| 16 | 5 | 31.25 | 6 | 37.5 |
| 32 | 6 | 18.75 | 7 | 21.875 |
| 64 | 7 | 10.94 | 8 | 12.5 |
| 128 | 8 | 6.25 | 9 | 7.03 |
| 256 | 9 | 3.52 | 10 | 3.91 |

# Characteristics of Syndrome

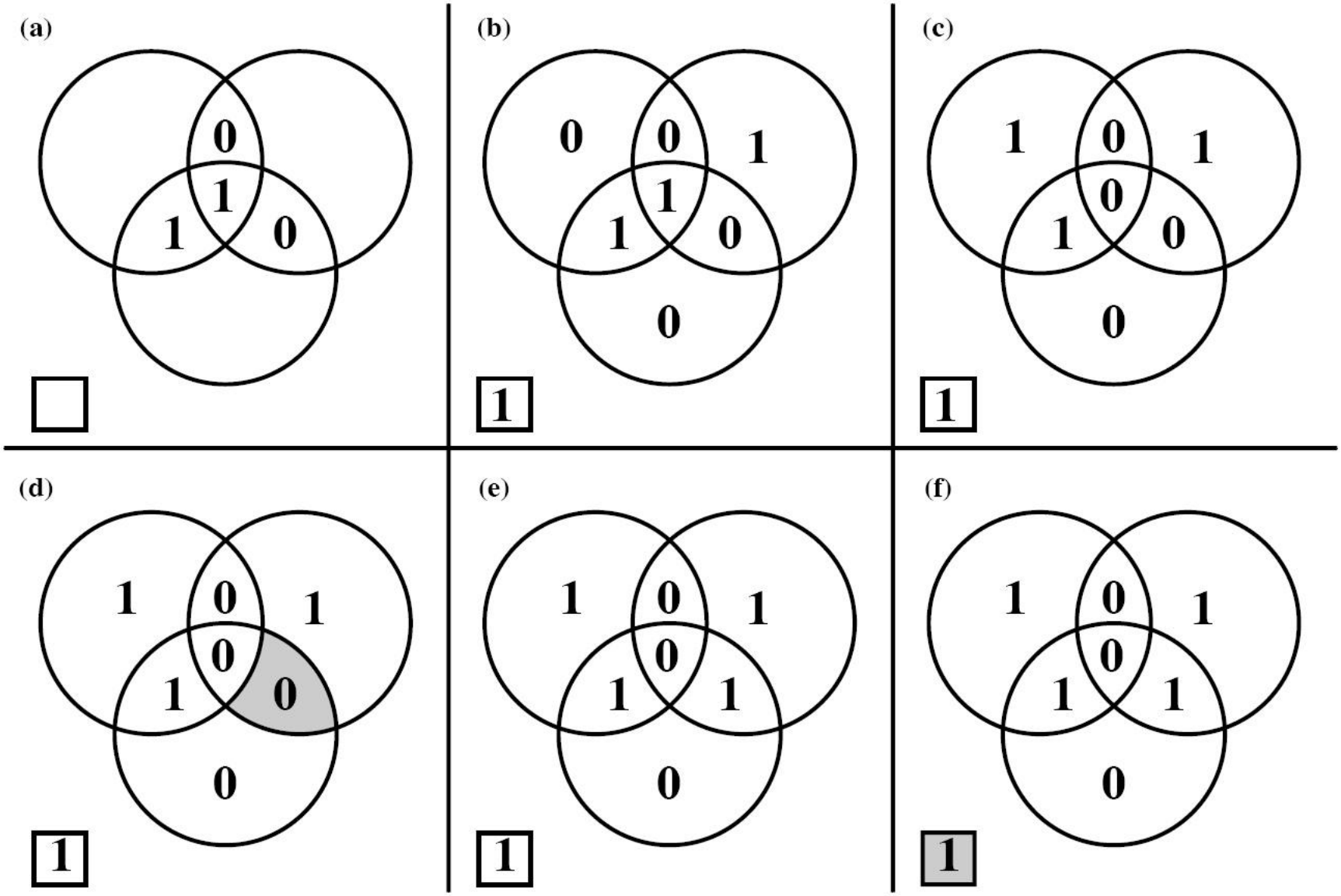
- 4-bit syndrome for an 8-bit data word
  - If the syndrome contains all 0s, no error has been detected
  - If the syndrome contains one and only one bit set to 1, an error has occurred in one of the 4 check bits. No correction

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C1 | = | D1 | ⊕ | D2 | ⊕ | | | D4 | ⊕ | D5 | ⊕ | | | D7 | | |
| C2 | = | D1 | ⊕ | | | D3 | ⊕ | D4 | ⊕ | | | D6 | ⊕ | D7 | | |
| C3 | = | | | D2 | ⊕ | D3 | ⊕ | D4 | ⊕ | | | | | | | D8 |
| C4 | = | | | | | | | | | D5 | ⊕ | D6 | ⊕ | D7 | ⊕ | D8 |

- Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit

| Bit Position | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position Number | 1100 | 1011 | 1010 | 1001 | 1000 | 0111 | 0110 | 0101 | 0100 | 0011 | 0010 | 0001 |
| Data Bit | D8 | D7 | D6 | D5 | | D4 | D3 | D2 | | D1 | | |
| Check Bit | | | | | C8 | | | | C4 | | C2 | C1 |

# Hamming SEC-DEC Code



# Error-Correcting Code

- Enhances the reliability of the memory at the cost of added complexity
- For 1-bit-per-chip organization, a SEC-DEC code is generally considered adequate
  - IBM 30xx, 8-bit SEC-DEC code for each 64 bits of data, 12% larger of size
  - VAX, 7-bit SEC-DEC code for each 32 bits of memory, 22% overhead
  - Modern DRAM use 9 check bits for each 128 data bits, 7% overhead

# Advanced DRAM Organization

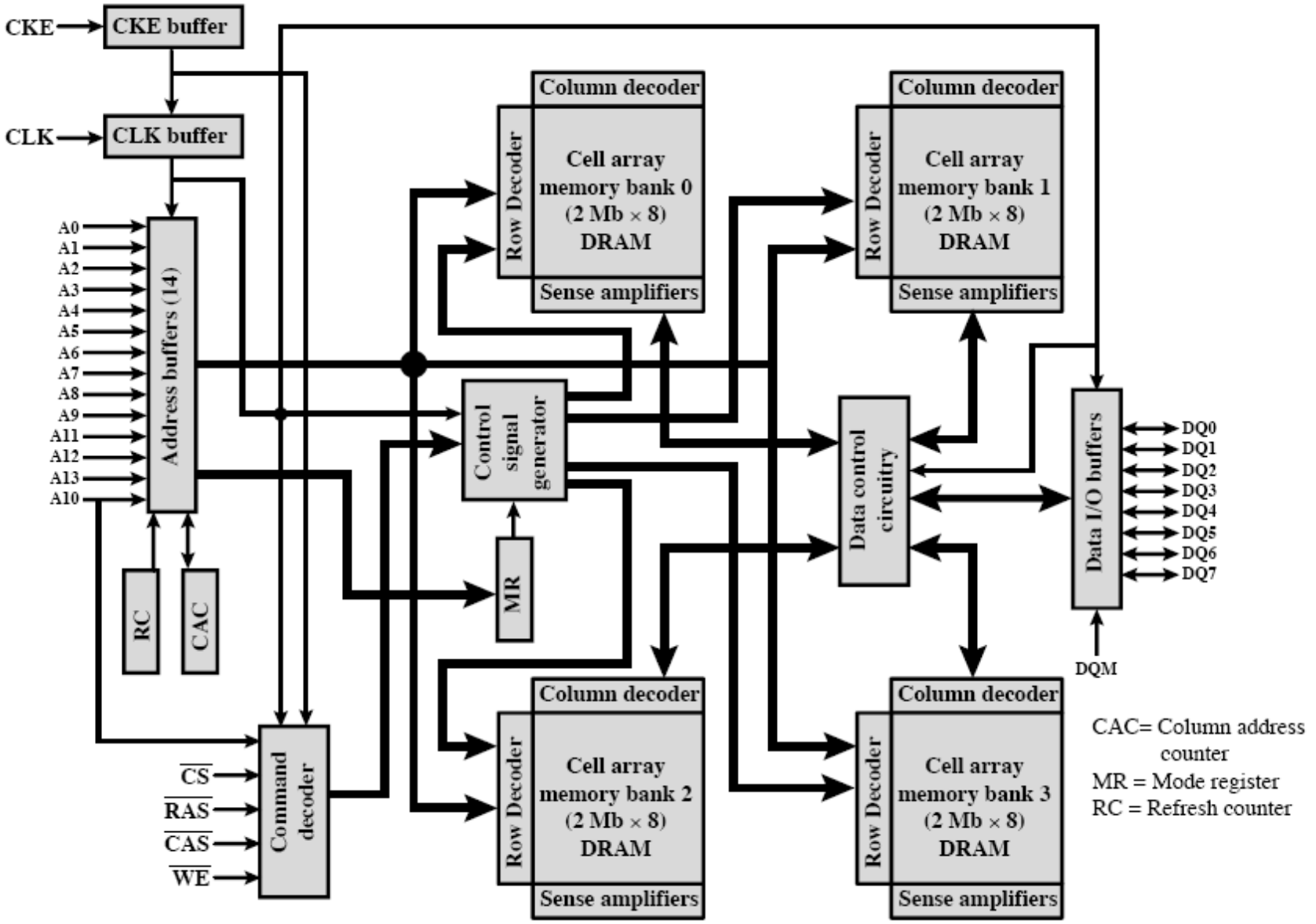
- Basic DRAM same since first RAM chips
- Enhanced DRAM
  - Contains small SRAM as well
  - SRAM holds last line read (c.f. Cache!)
- Cache DRAM
  - Larger SRAM component
  - Use as cache or serial buffer

| | Clock frequency (MHz) | Transfer rate (GB/s) | Access time (ns) | Pin count |
|---|---|---|---|---|
| SDRAM | 166 | 1.3 | 18 | 168 |
| DDR | 200 | 3.2 | 12.5 | 184 |
| RDRAM | 600 | 4.8 | 12 | 162 |

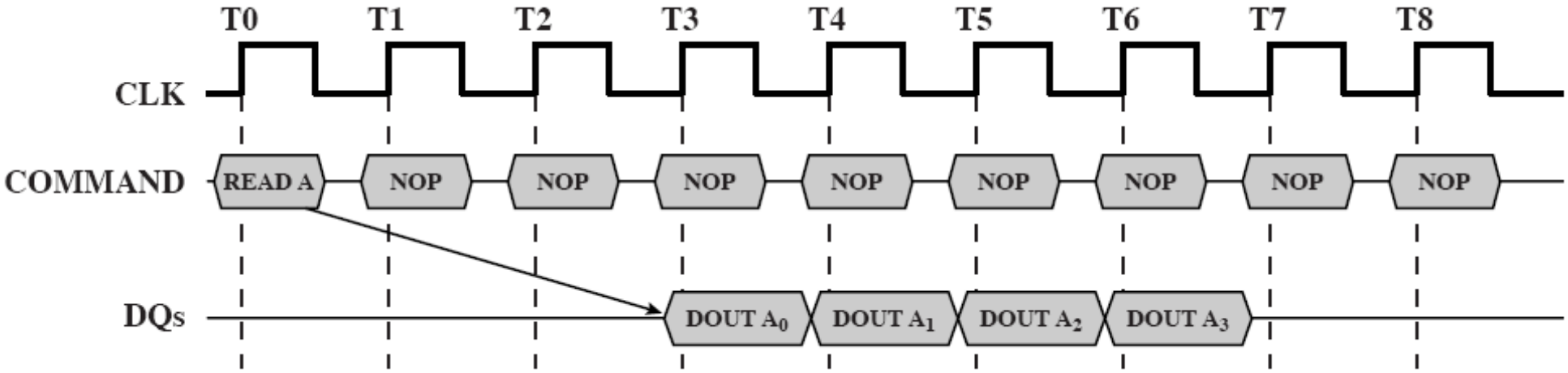
# Synchronous DRAM (SDRAM)

- Access is synchronized with an external clock
- Address is presented to RAM
- RAM finds data (CPU waits in conventional DRAM)
- Since SDRAM moves data in time with system clock, CPU knows when data will be ready
- CPU does not have to wait, it can do something else
- Burst mode allows SDRAM to set up stream of data and fire it out in block
- DDR-SDRAM sends data twice per clock cycle (leading & trailing edge)

# SDRAM


CKE
CKE buffer
CLK
CLK buffer
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A11
A12
A13
A10
Address buffers (14)
RC
CAC
MR
Control signal generator
Column decoder
Row Decoder
Cell array memory bank 0 (2 Mb × 8) DRAM
Sense amplifiers
Column decoder
Row Decoder
Cell array memory bank 1 (2 Mb × 8) DRAM
Sense amplifiers
Data control circuitry
Data I/O buffers
DQ0
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQM
Column decoder
Row Decoder
Cell array memory bank 2 (2 Mb × 8) DRAM
Sense amplifiers
Column decoder
Row Decoder
Cell array memory bank 3 (2 Mb × 8) DRAM
Sense amplifiers
CS
RAS
CAS
WE
Command decoder
CAC= Column address counter
MR = Mode register
RC = Refresh counter

# SDRAM Read Timing

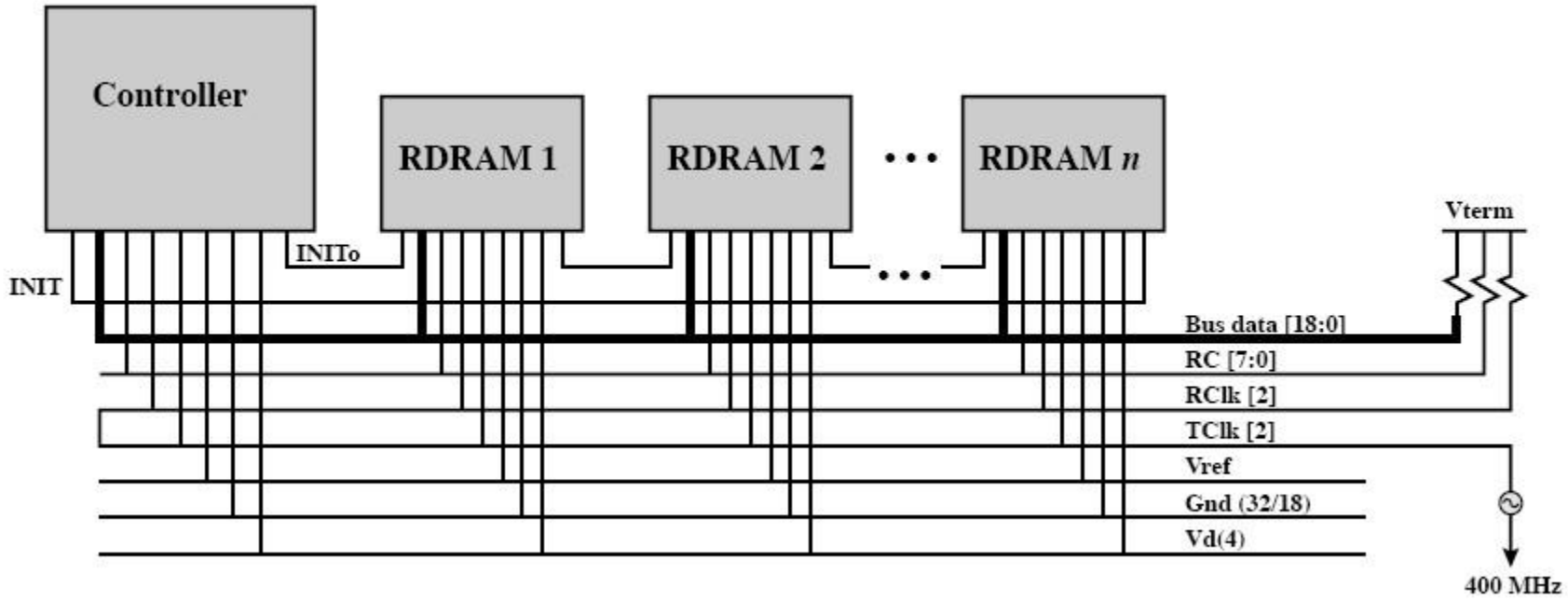



Burst length = 4, $\overline{\text{CAS}}$ latency = 2

# RAMBUS DRAM

- Adopted by Intel for Pentium & Itanium
- Main competitor to SDRAM
- Vertical package – all pins on one side
- Data exchange over 28 wires < 12 cm long
- Bus addresses up to 320 RDRAM chips at 1.6Gbps
- Asynchronous block protocol
  - 480ns access time
  - Then 1.6 Gbps

# RAMBUS Diagram



# DDR SDRAM

- SDRAM can only send data once per clock
- Double-data-rate SDRAM can send data twice per clock cycle
  - Rising edge and falling edge

# Cache DRAM

- Mitsubishi
- Integrates small SRAM cache (16 kb) onto generic DRAM chip
- Used as true cache
  - 64-bit lines
  - Effective for ordinary random access
- To support serial access of block of data
  - E.g. refresh bit-mapped screen
    - CDRAM can prefetch data from DRAM into SRAM buffer
    - Subsequent accesses solely to SRAM

# Lecture 2

- Basic components
- Memory hierarchy
- Cache memory
- Internal Memory
- External Memory
- Virtual Memory

# Types of External Memory

- Magnetic Disk
  - RAID
  - Removable
- Optical
  - CD-ROM
  - CD-Recordable (CD-R)
  - CD-R/W
  - DVD
- Magnetic Tape

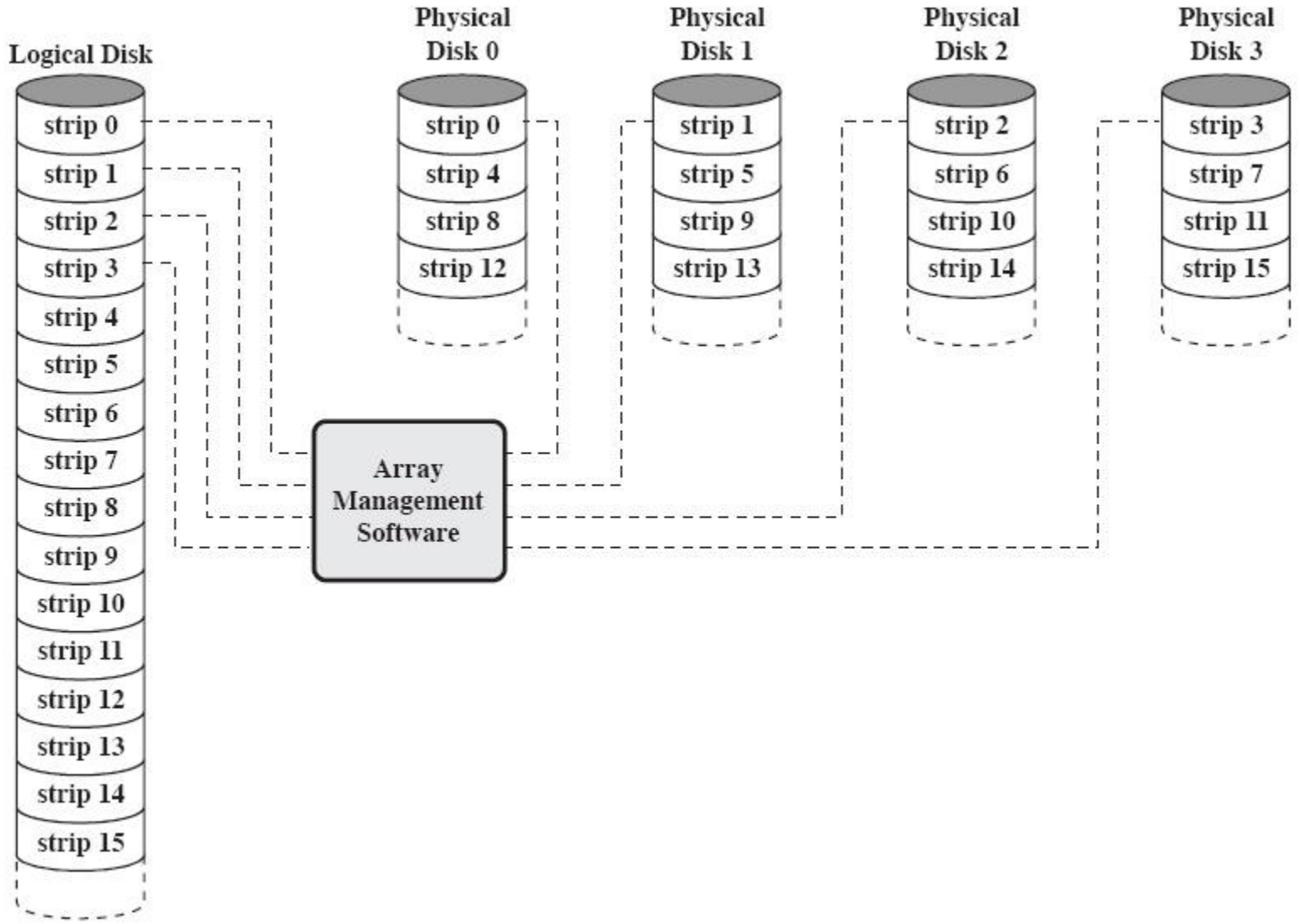
# RAID

- Redundant Array of Independent Disks
- Redundant Array of Inexpensive Disks
- 7 levels in common use
- Not a hierarchy
- Common characteristics
  - Set of physical disks viewed as single logical drive by O/S
  - Data distributed across physical drives
  - Can use redundant capacity to store parity information

# RAID 0

- No redundancy
- Data striped across all disks
- Round Robin striping
- Stripe
- Increase speed
  - Multiple data requests probably not on same disk
  - Disks seek in parallel
  - A set of data is likely to be striped across multiple disks

# Data Mapping For RAID 0



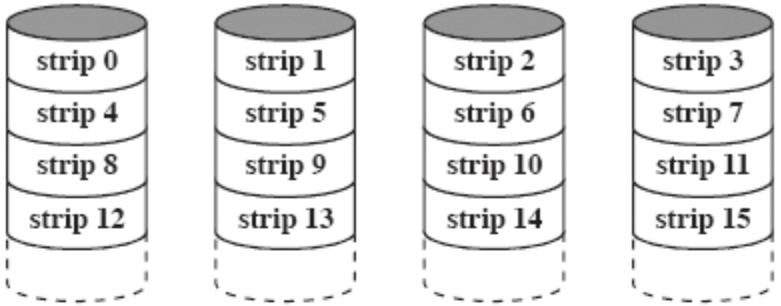
# RAID 1

- Mirrored Disks
- Data is striped across disks
- 2 copies of each stripe on separate disks
- Read from either
- Write to both
- Recovery is simple
  - Swap faulty disk & re-mirror
  - No down time
- Expensive

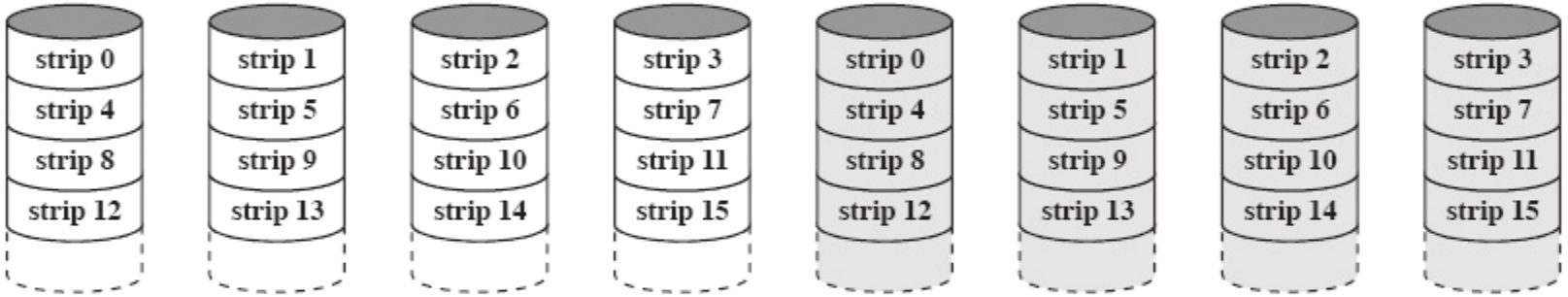
# RAID 2

- Disks are synchronized
- Very small stripes
  - Often single byte/word
- Error correction calculated across corresponding bits on disks
- Multiple parity disks store Hamming code error correction in corresponding positions
- Lots of redundancy
  - Expensive
  - Not used

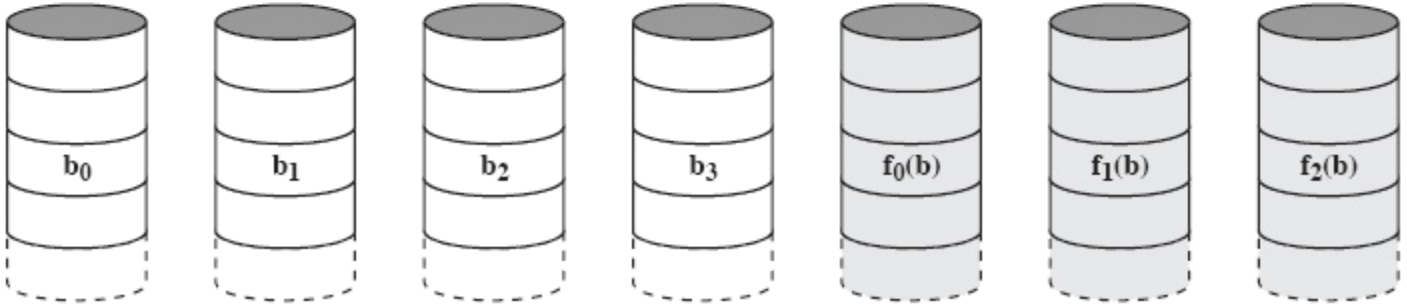
# RAID 0, 1, 2




(a) RAID 0 (non-redundant)




(b) RAID 1 (mirrored)




(c) RAID 2 (redundancy through Hamming code)

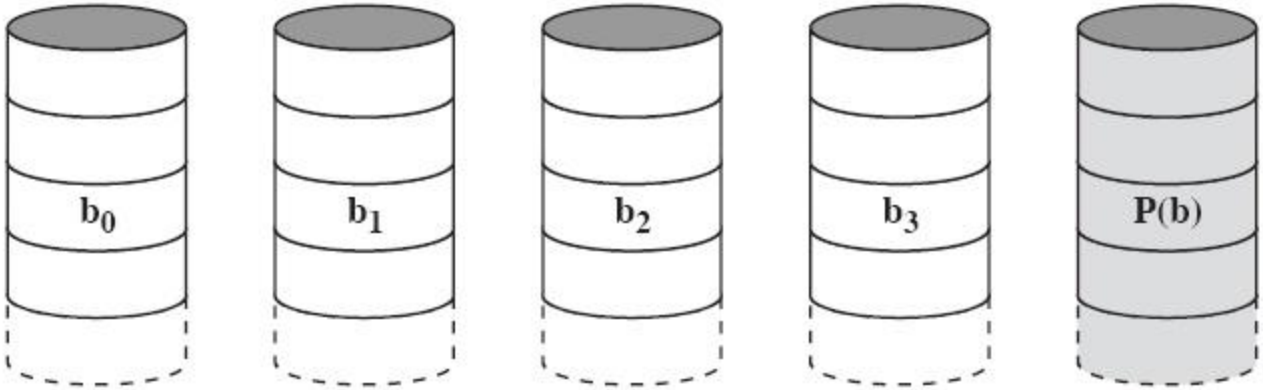
# RAID 3

- Similar to RAID 2
- Only one redundant disk, no matter how large the array
- Simple parity bit for each set of corresponding bits
- Data on failed drive can be reconstructed from surviving data and parity info
- Very high transfer rates

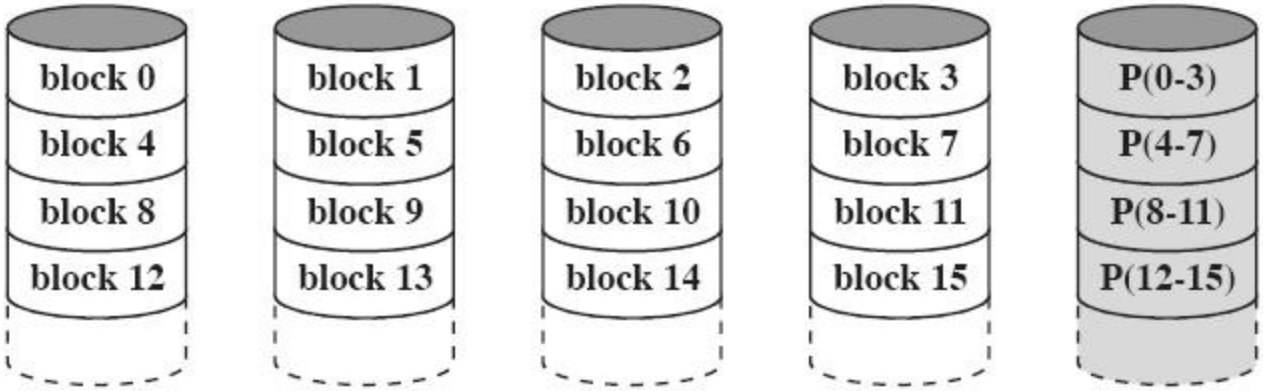
# RAID 4

- Each disk operates independently
- Good for high I/O request rate
- Large stripes
- Bit by bit parity calculated across stripes on each disk
- Parity stored on parity disk

# RAID 3 & 4




(a) RAID 3 (bit-interleaved parity)




(b) RAID 4 (block-level parity)

# RAID 5

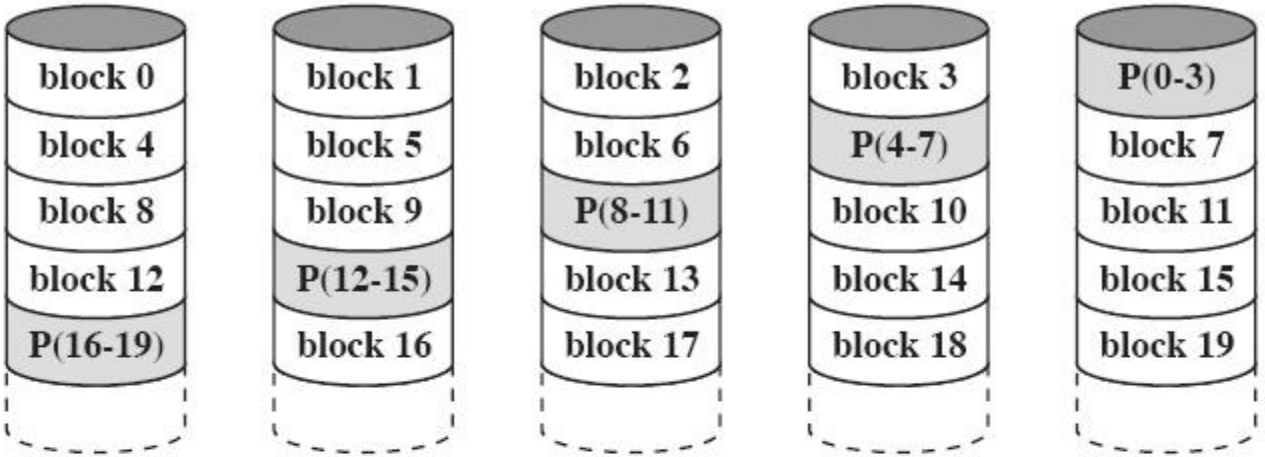
- Like RAID 4
- Parity striped across all disks
- Round robin allocation for parity stripe
- Avoids RAID 4 bottleneck at parity disk
- Commonly used in network servers

- N.B. DOES NOT MEAN 5 DISKS!!!!!

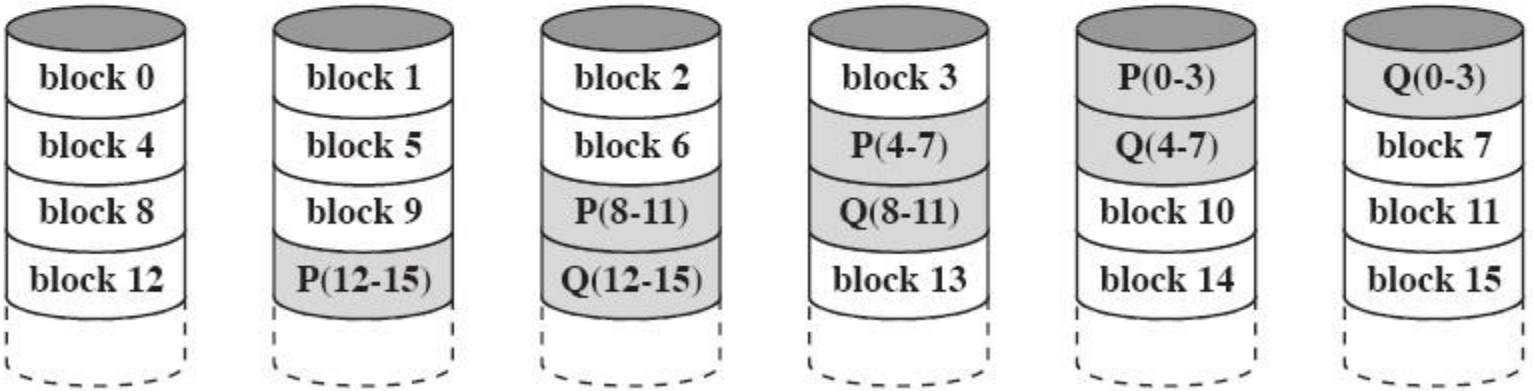
# RAID 6

- Two parity calculations
- Stored in separate blocks on different disks
- User requirement of N disks needs N+2
- High data availability
  - Three disks need to fail for data loss
  - Significant write penalty

# RAID 5 & 6




(c) RAID 5 (block-level distributed parity)




(d) RAID 6 (dual redundancy)

# RAID Levels

| Category | Level | Description | Disks required | Data availability | Large I/O data transfer capacity | Small I/O request rate |
|---|---|---|---|---|---|---|
| Striping | 0 | Nonredundant | $N$ | Lower than single disk | Very high | Very high for both read and write |
| Mirroring | 1 | Mirrored | $2N$, $3N$, etc. | Higher than RAID 2, 3, 4, or 5; lower than RAID 6 | Higher than single disk for read; similar to single disk for write | Up to twice that of a single disk for read; similar to single disk for write |
| Parallel access | 2 | Redundant via Hamming code | $N + m$ | Much higher than single disk; higher than RAID 3, 4, or 5 | Highest of all listed alternatives | Approximately twice that of a single disk |
| | 3 | Bit-interleaved parity | $N + 1$ | Much higher than single disk; comparable to RAID 2, 4, or 5 | Highest of all listed alternatives | Approximately twice that of a single disk |
| Independent access | 4 | Block-interleaved parity | $N + 1$ | Much higher than single disk; comparable to RAID 2, 3, or 5 | Similar to RAID 0 for read; significantly lower than single disk for write | Similar to RAID 0 for read; significantly lower than single disk for write |
| | 5 | Block-interleaved distributed parity | $N + 1$ | Much higher than single disk; comparable to RAID 2, 3, or 4 | Similar to RAID 0 for read; lower than single disk for write | Similar to RAID 0 for read; generally lower than single disk for write |
| | 6 | Block-interleaved dual distributed parity | $N + 2$ | Highest of all listed alternatives | Similar to RAID 0 for read; lower than RAID 5 for write | Similar to RAID 0 for read; significantly lower than RAID 5 for write |

# RAID Comparison (1)

| Level | Advantages | Disadvantages | Applications |
|---|---|---|---|
| 0 | I/O performance is greatly improved by spreading the I/O load across many channels and drives<br>No parity calculation overhead is involved<br>Very simple design<br>Easy to implement | The failure of just one drive will result in all data in an array being lost | Video production and Editing<br>Image editing<br>Pre-press applications<br>Any application requiring high bandwidth |
| 1 | 100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk<br>Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures<br>Simplest RAID storage subsystem design | Highest disk overhead of all RAID types (100%) - inefficient | Accounting<br>Payroll<br>Financial<br>Any application requiring very high availability |
| 2 | Extremely high data transfer rates possible<br>The higher the data transfer rate required, the better the ratio of data disks to ECC disks<br>Relatively simple controller design compared to RAID levels 3,4 & 5 | Very high ratio of ECC disks to data disks with smaller word sizes - inefficient<br>Entry level cost very high - requires very high transfer rate requirement to justify | No commercial implementations exist / not commercially viable |

# RAID Comparison (2)

| | | | |
|---|---|---|---|
| 3 | Very high read data transfer rate<br>Very high write data transfer rate<br>Disk failure has an insignificant impact on throughput<br>Low ratio of ECC (parity) disks to data disks means high efficiency | Transaction rate equal to that of a single disk drive at best (if spindles are synchronized)<br>Controller design is fairly complex | Video production and live streaming<br>Image editing<br>Video editing<br>Prepress applications<br>Any application requiring high throughput |
| 4 | Very high Read data transaction rate<br>Low ratio of ECC (parity) disks to data disks means high efficiency | Quite complex controller design<br>Worst write transaction rate and Write aggregate transfer rate<br>Difficult and inefficient data rebuild in the event of disk failure | No commercial implementations exist / not commercially viable |
| 5 | Highest Read data transaction rate<br>Low ratio of ECC (parity) disks to data disks means high efficiency<br>Good aggregate transfer rate | Most complex controller design<br>Difficult to rebuild in the event of a disk failure (as compared to RAID level 1) | File and application servers<br>Database servers<br>Web, e-mail, and news servers<br>Intranet servers<br>Most versatile RAID level |
| 6 | Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures | More complex controller design<br>Controller overhead to compute parity addresses is extremely high | Perfect solution for mission critical applications |

# Summary

- A memory system has to fit very large programs and still provide fast access.
- No single type of memory can provide all the need of a computer system.
- Usually several different storage mechanisms are organized in a layer hierarchy.
  - Cache is a hardware solution to improve memory access which is transparent to the programmers.
  - Virtual memory provides a much larger address space than the available physical space with the help of the OS (software solution).
- The layer structure works very well partly due to the locality of reference principle.