2018年度(平成30年度)版

Ver. 2018-10-10a

Course number: CSC.T363

# コンピュータアーキテクチャ
# Computer Architecture

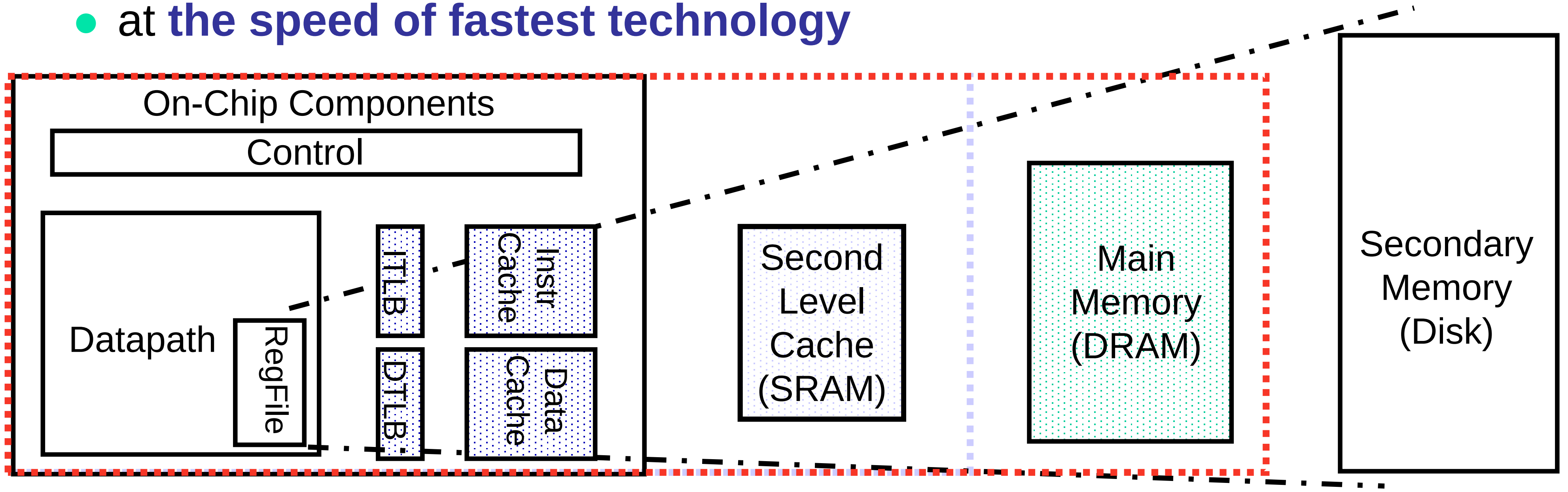
## 4. キャッシュ:ダイレクトマップ方式
## Caches: Direct-Mapped

www.arch.cs.titech.ac.jp/lecture/CA/
Room No.W321
Tue 13:20-16:20, Fri 13:20-14:50

吉瀬 謙二 情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

## A Typical Memory Hierarchy

- By taking advantage of **the principle of locality**（局所性）
  - Present **much memory** in **the cheapest technology**
  - at **the speed of fastest technology**

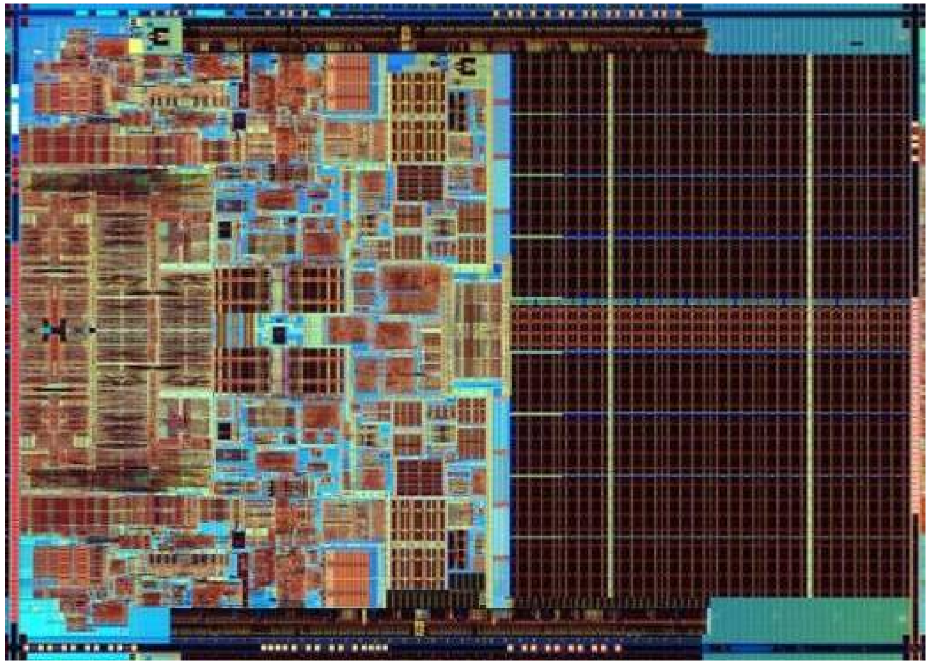


| | | | | | |
|---|---|---|---|---|---|
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

TLB: Translation Lookaside Buffer

## Cache

- *Cache memory* consists of a small, fast memory that acts as a buffer for the large memory.
- The nontechnical definition of *cache* is a safe place for hiding things.

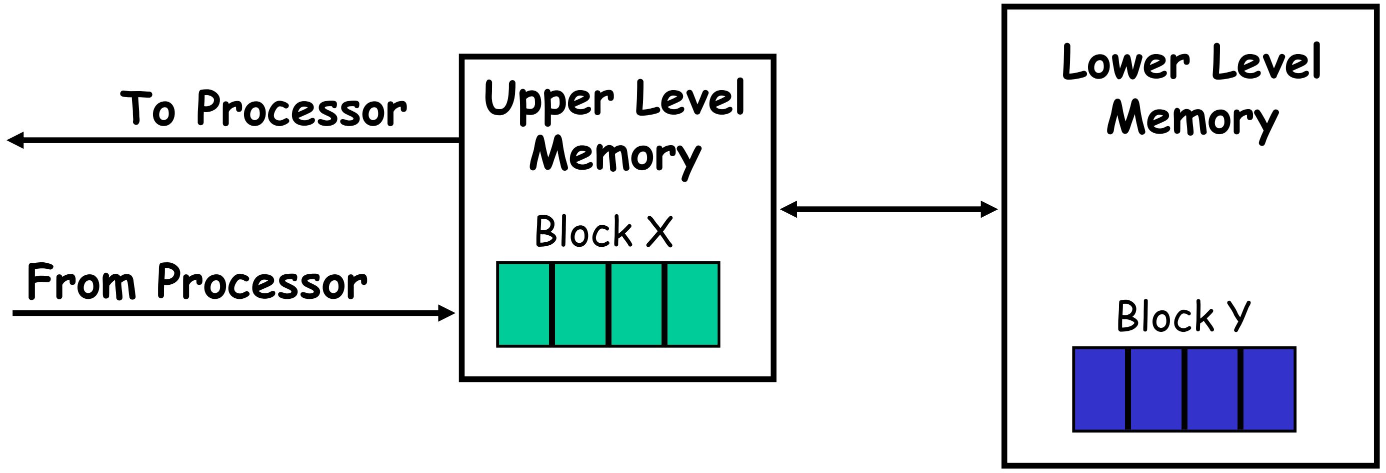

Intel Core 2 Duo

## パレートの法則

- Vilfredo Federico Damaso Pareto
  - イタリアの経済学者(1948 – 1923)
- パレートの法則
  - 全体の数値の大部分は，全体を構成するうちの一部の要素が生み出している
  - 80:20の法則

## The Memory Hierarchy: Why Does it Work?

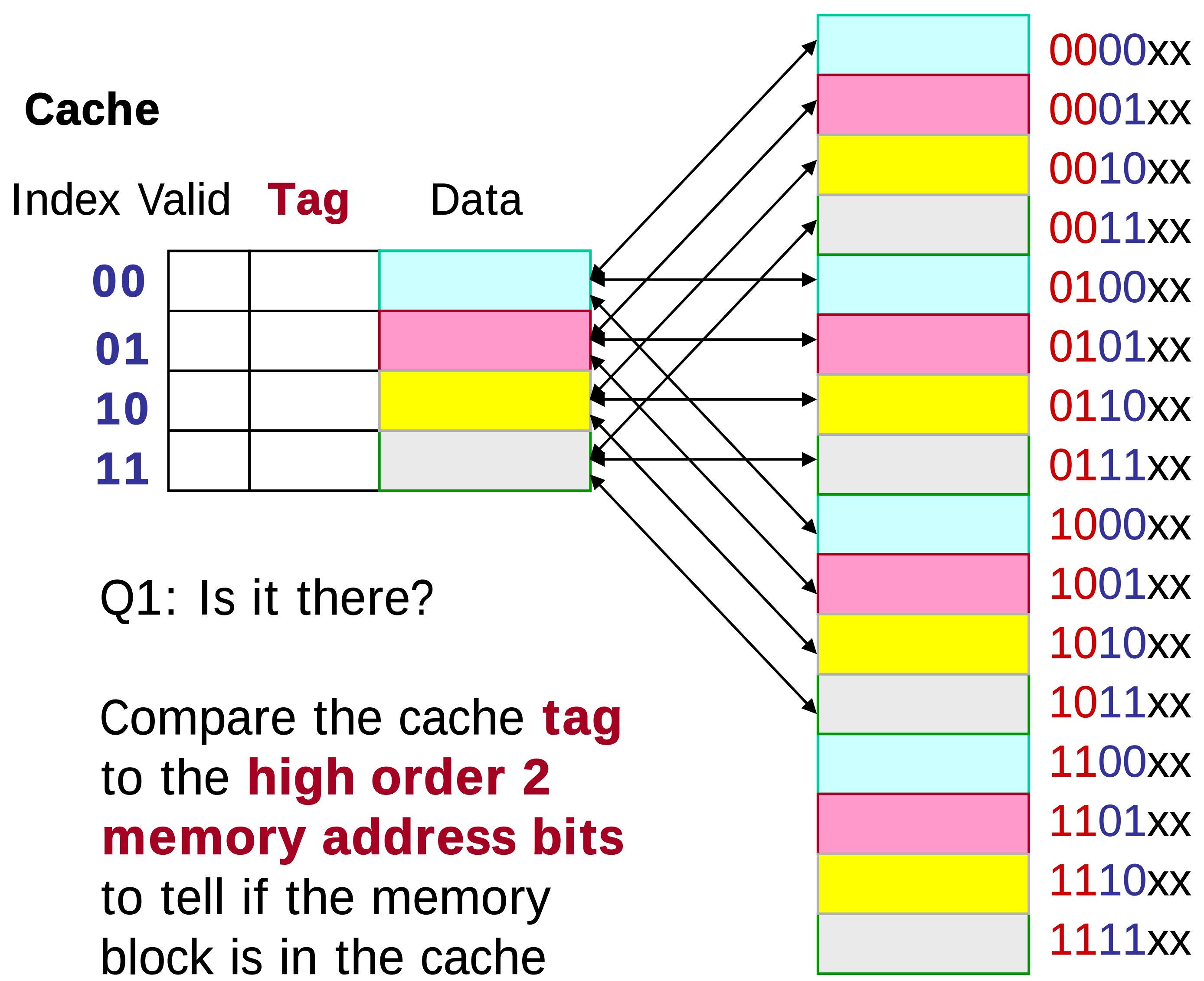
- **Temporal Locality** (時間的局所性, Locality in Time):
  ⇒ Keep **most recently accessed** data items closer to the processor
- **Spatial Locality** (空間的局所性, Locality in Space):
  ⇒ Move blocks consisting of **contiguous words** to the upper levels



## 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?**
- **Direct mapped**
  - For each item of data at the lower level, there is **exactly one location** in the cache where it might be - so lots of items at the lower level must **share** locations in the upper level
  - Address mapping:
    **(block address) modulo (# of blocks in the cache)**
  - First, consider block sizes of **one word**

## Caching: A Simple First Example



**Main Memory**

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

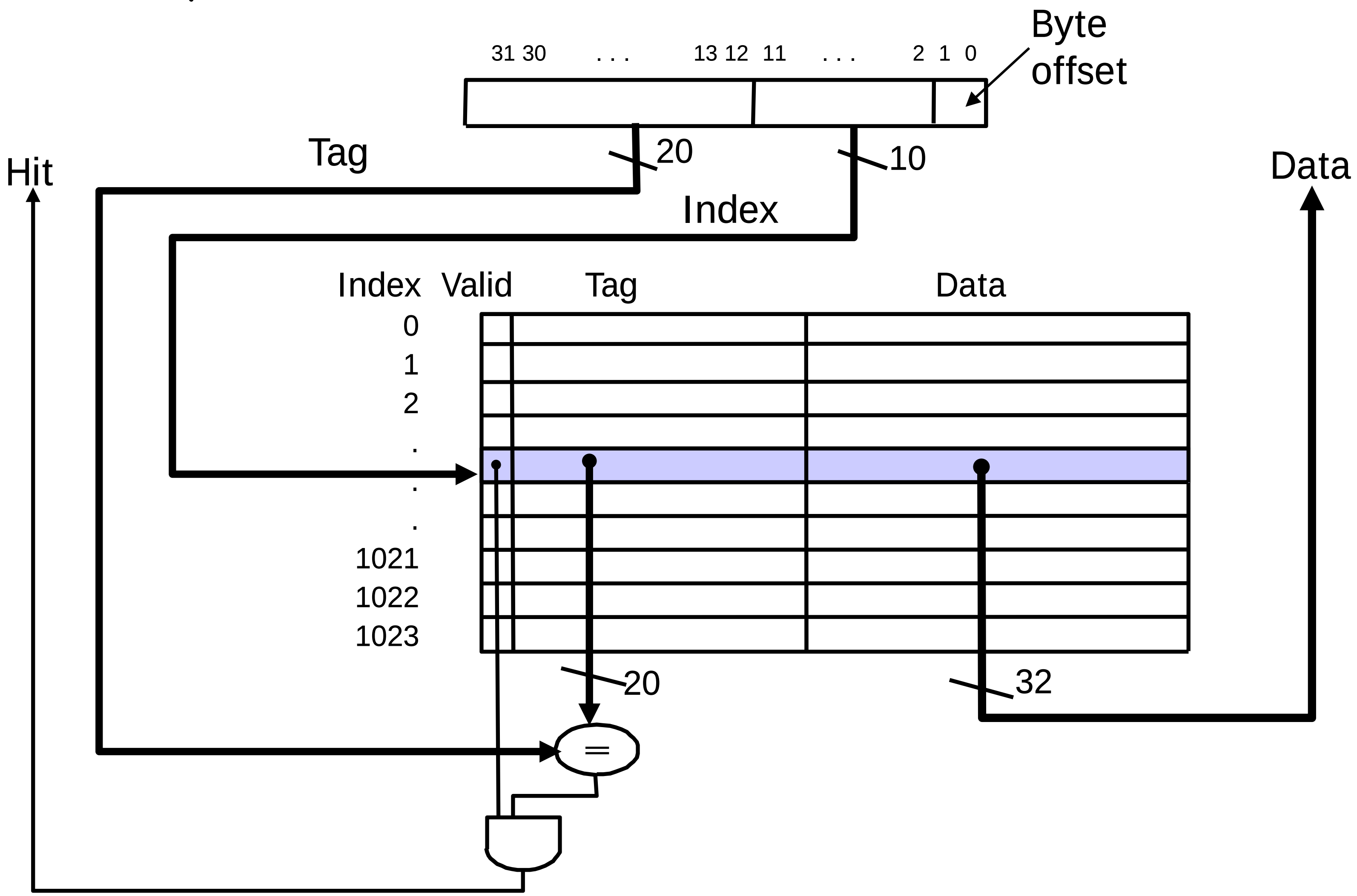
Q2: How do we find it?

Use **next 2 low order memory address bits** – the **index** – to determine which cache block

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

# MIPS Direct Mapped Cache Example

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



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

# Example Behavior of Direct Mapped Cache

- Consider the main memory word reference string (word addresses) 0 1 2 3 4 3 4 15

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

**0** miss

| Tag | |
|---|---|
| 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

| | |
|---|---|
| 01 ~~00~~ | ~~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

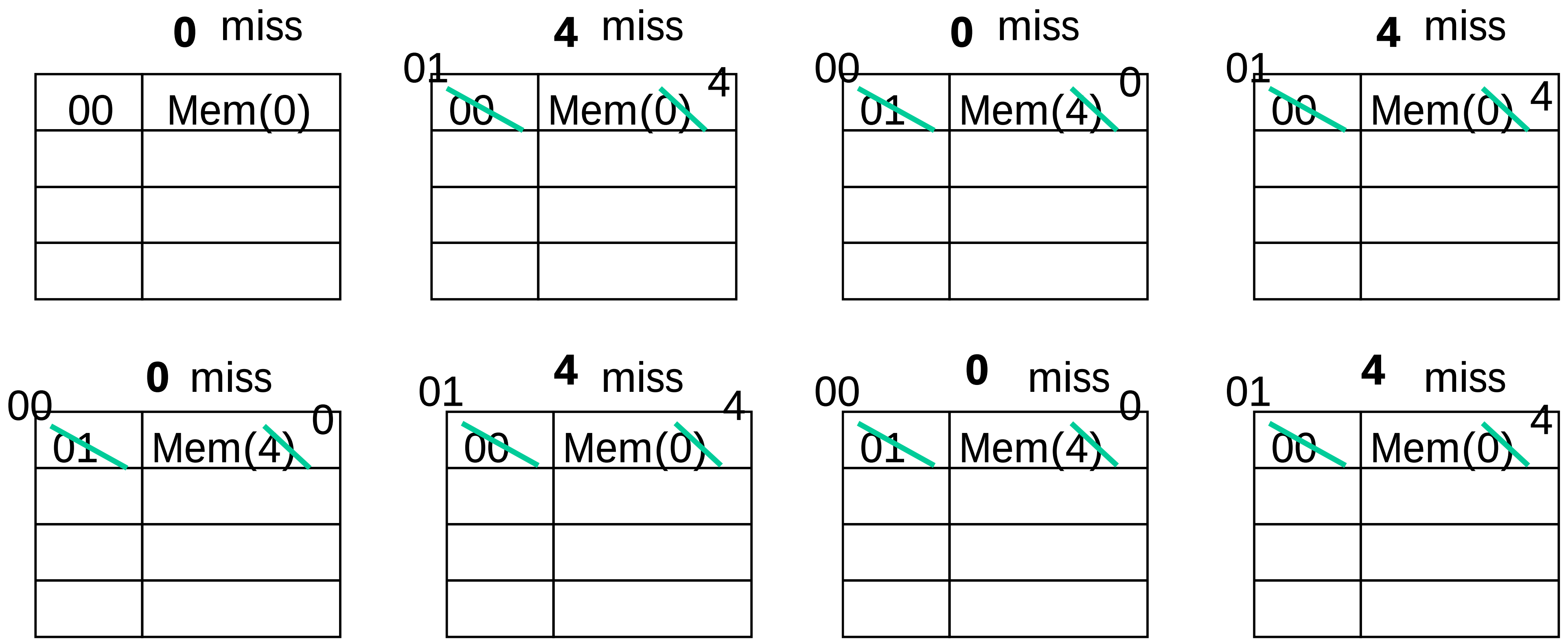
| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 11 ~~00~~ | ~~Mem(3)~~ 15 |

- 8 requests, 6 misses

## Another Reference String Mapping

- Consider the main memory word reference string

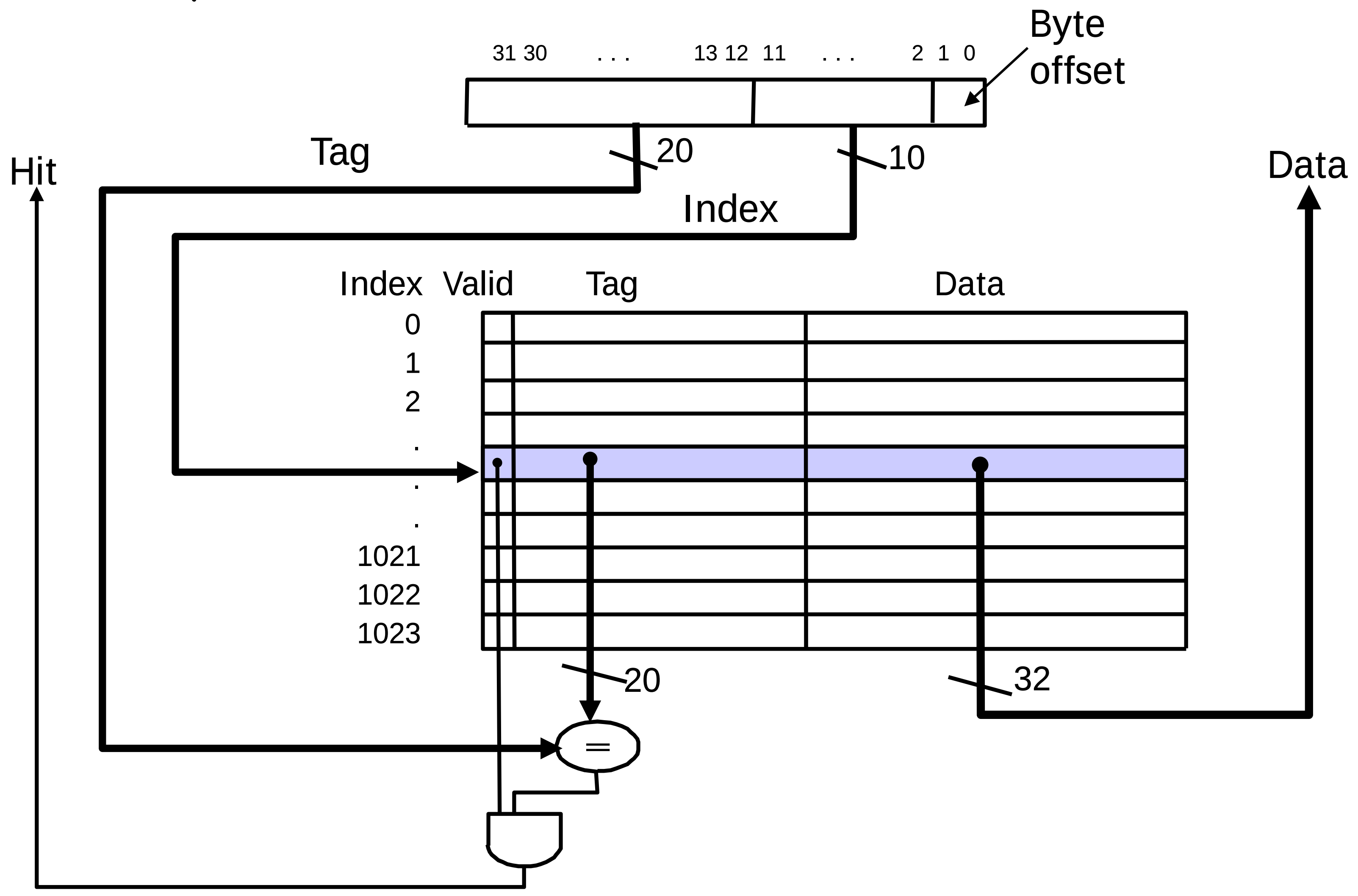
  0 4 0 4 0 4 0 4



- 8 requests, 8 misses
  - Ping pong effect due to **conflict** misses - two memory locations that map into the same cache block

## MIPS Direct Mapped Cache Example

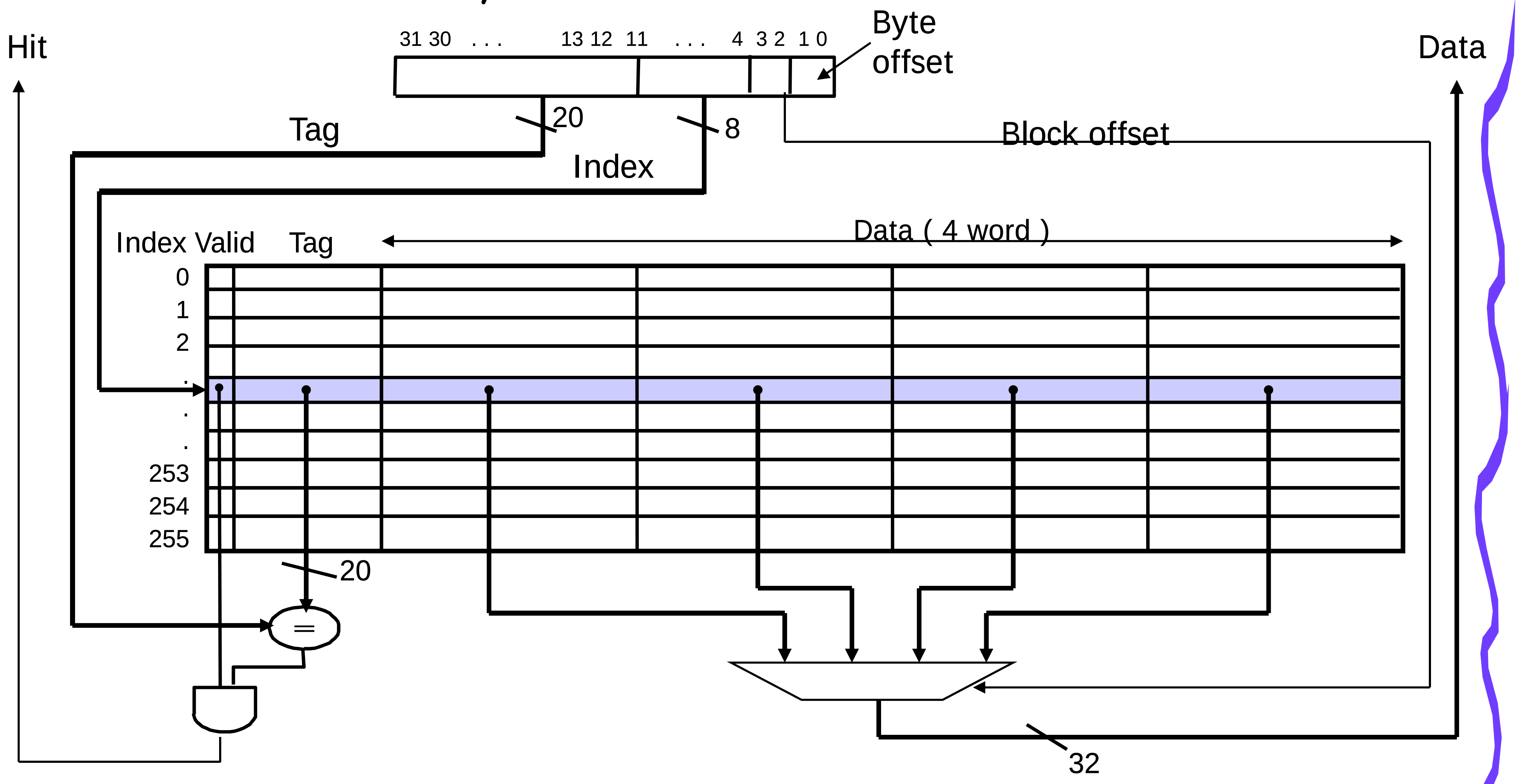
- One word/block, cache size = 1K words



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

# Multiword Block Direct Mapped Cache

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



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

## Taking Advantage of **Spatial Locality**

- Let cache block hold more than one word

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

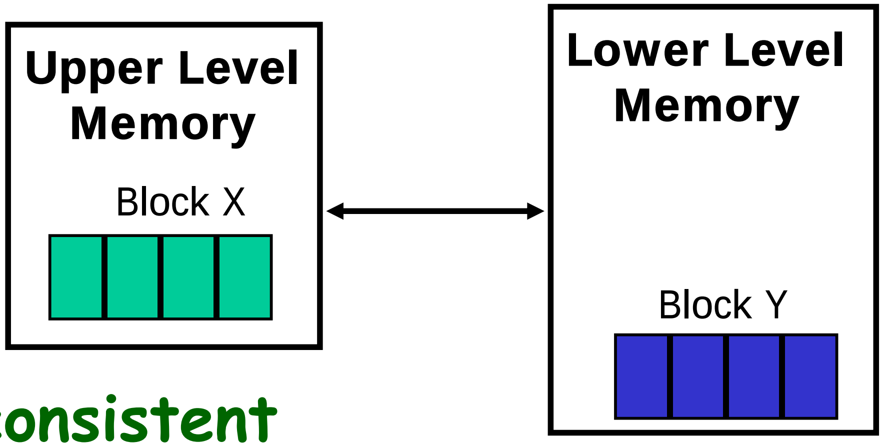
| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

**15** miss

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

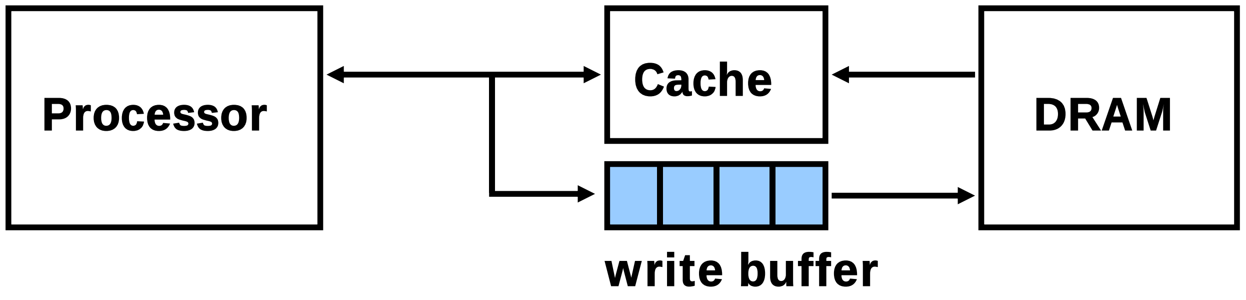
- 8 requests, 4 misses

# Handling Cache Hits (Miss is the next issue)



- **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**)
    - 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

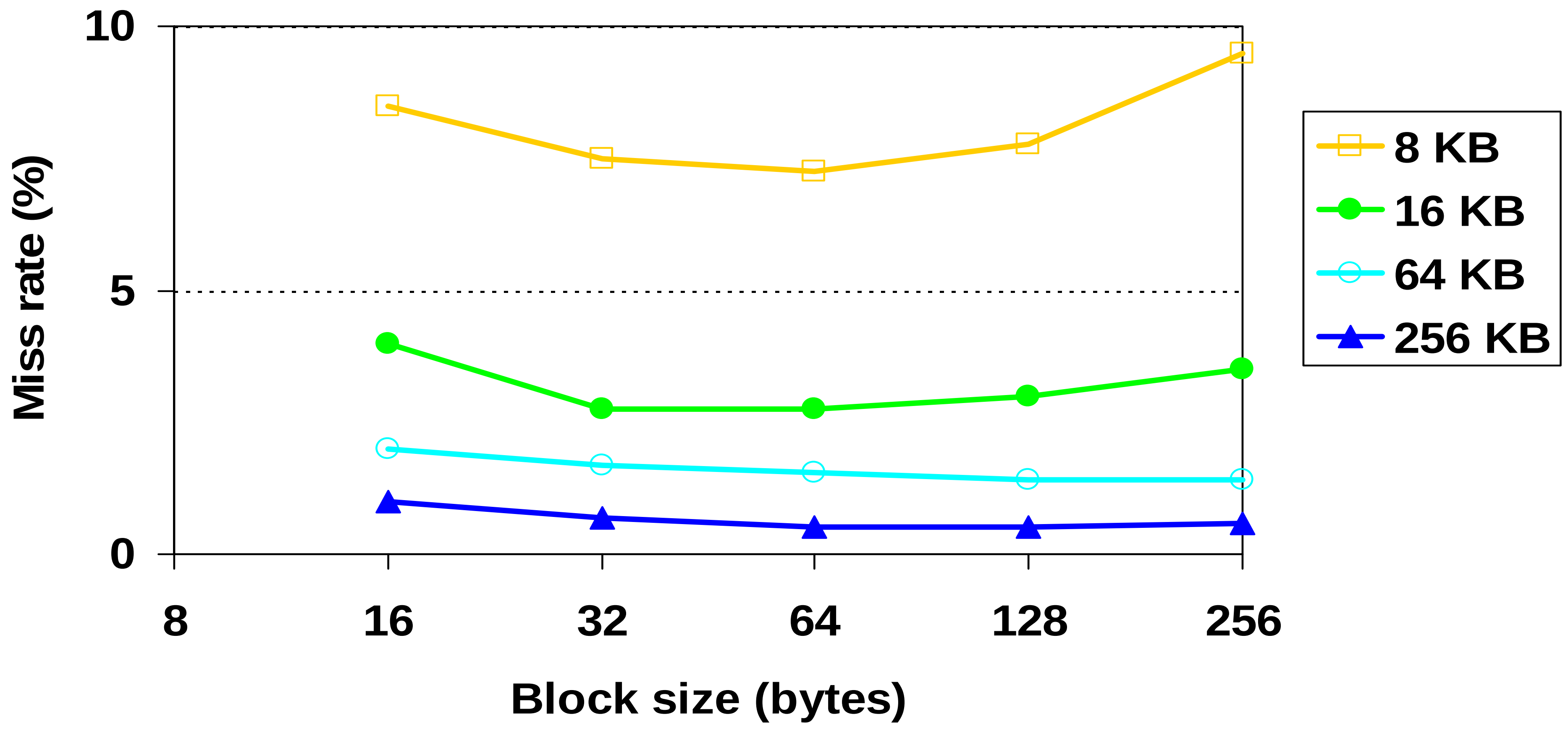


- **Write buffer** between the cache and main memory
  - Processor: writes data into the cache and the write buffer
  - **Memory controller**: writes contents of the write buffer to memory
- The write buffer is just a **FIFO**
  - Typical number of entries: 4
  - Works fine if store frequency is low
- Memory system designer's nightmare, Write buffer **saturation**
  - One solution is to use a write-back cache; another is to use an L2 cache

## 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)**
  - Write allocate
    - (a) write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall
    - **(b) stall** the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache, write the word from the processor to the cache, then let the pipeline resume
  - 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**

## 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