# Caches

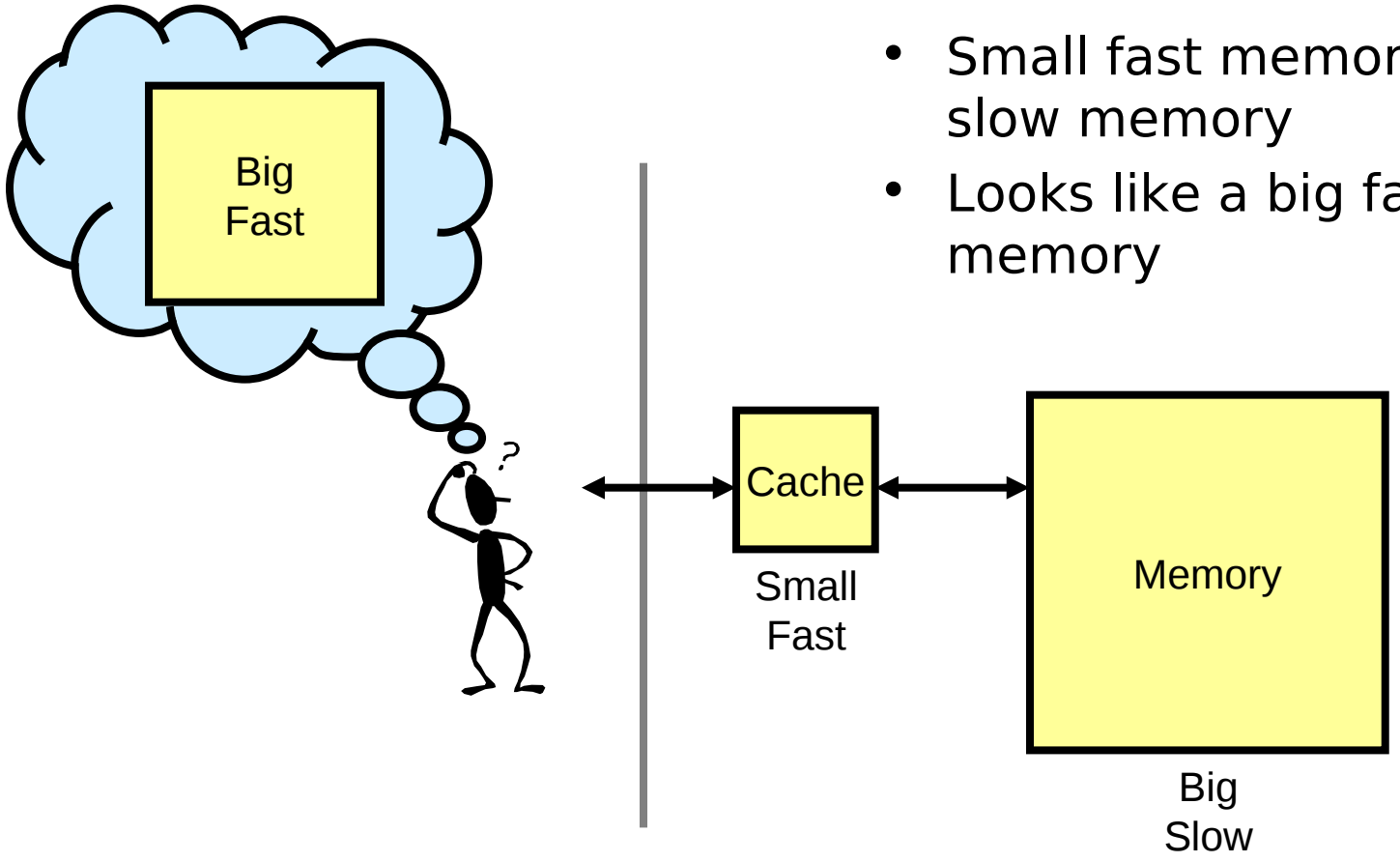
Giorgio Richelli

# The Memory Bottleneck

- Typical CPU clock rate
  - >2GHz (0.5 ns cycle time)
- Typical DRAM access time
  - 30ns (about 60 cycles)
- Typical main memory access
  - 100ns (200 cycles)
    - DRAM (60), precharge (20), chip crossings (60), overhead (60).

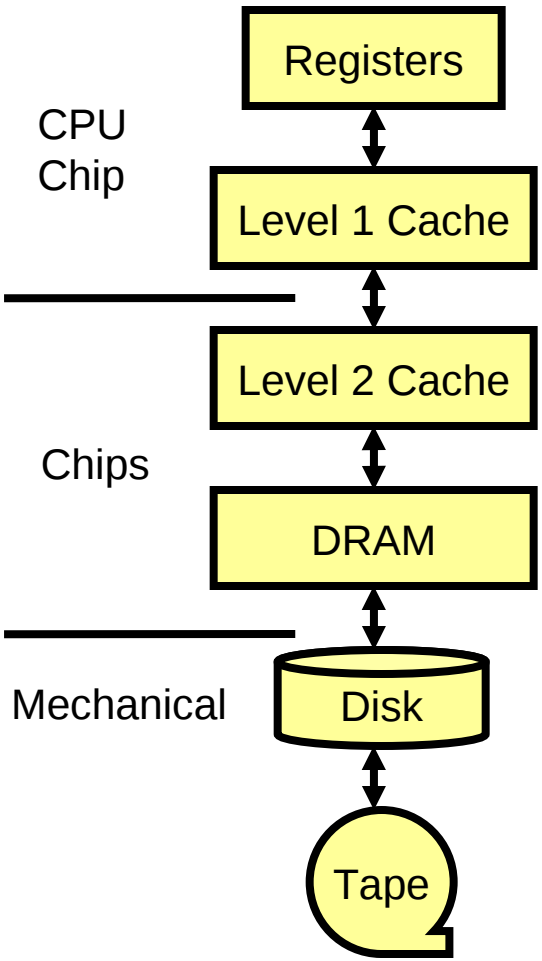
- This problem gets worse
  - CPUs get *faster*
  - Memories get *bigger*
- Memory delay is mostly communication time
  - reading/writing a bit is *fast*
  - it takes time to
    - select the right bit
    - route the data to/from the bit
- Big memories are *slow*
- Small memories can be made *fast*

# Cache Memory




- Small fast memory + big slow memory
- Looks like a big fast memory

# The Memory Hierarchy




| | Latency | Bandwidth | |
|---|---|---|---|
| | 1 cyc | 3-10 words/cycle<br>< 1KB | compiler managed |
| | 1-5cy | 1-2 words/cycle<br>32KB -1MB | hardware managed |
| | 6-15cy | 1 word/cycle<br>1MB - 4MB | hardware managed |
| | 60-200cy | 0.5 words/cycle<br>64MB - 4GB | OS managed |
| | $10^6$-$10^7$cy | 0.01 words/cycle<br>4GB+ | OS managed |

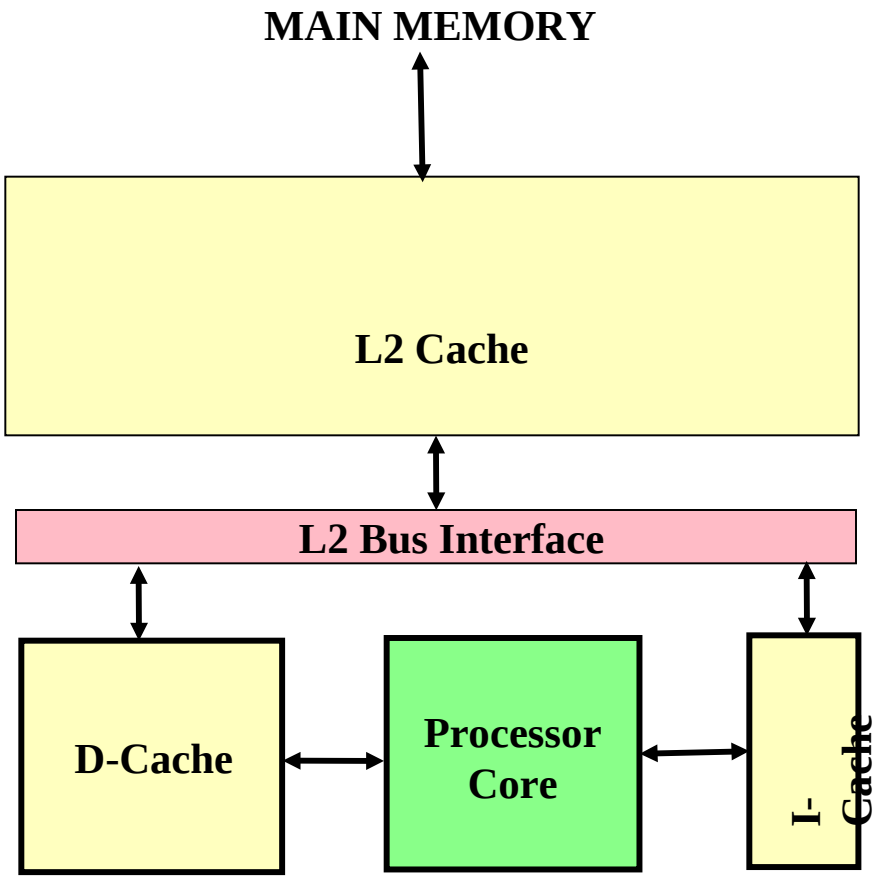
# Typical Cache Organization

**Pentium2: 16+16KB I+D L1**
**512 KB L2**

**POWER6: 64+64KB I+D L1**
**4 MB L2**
**32 MB L3 (offchip)**

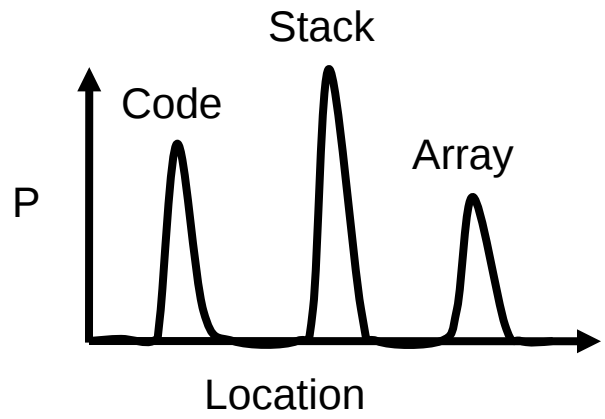
**Montecito: 16+16KB I+D L1**
**1024+256KB I+D L2**
**12 MB L3 (onchip)**

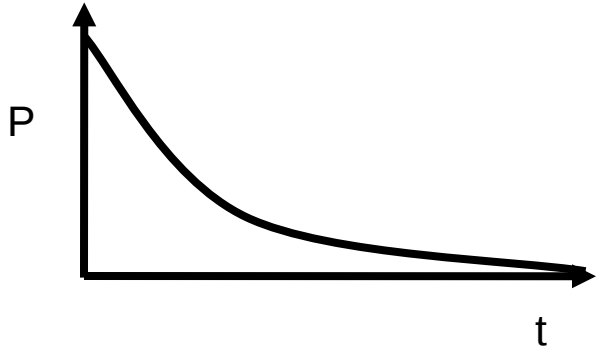
**POWER7: 32+32 KB I+D L1**
**256 KB L2**
**4-32 MB L3 fluid eDRAM**



# Locality of Reference

- Spatial Locality
  - likely to reference data *near* recent references
- Temporal Locality
  - likely to reference the same data that was referenced recently



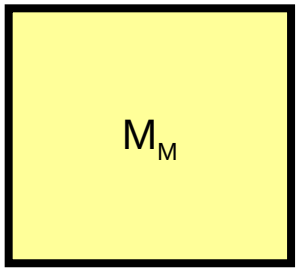



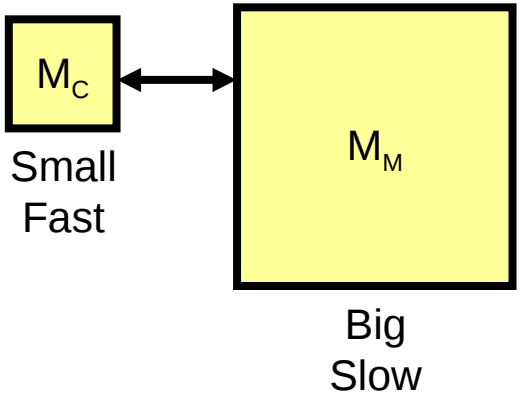
# Program Behavior

- Locality depends on type of program
- Some programs ‘behave’ well
  - small loop operating on data on stack
- Some programs don’t
  - frequent calls to nearly random subroutines
  - traversal of large, sparse data set
    - essentially random data references with no reuse
- Most programs exhibit some degree of locality

# Example

**What is the average memory access time?**



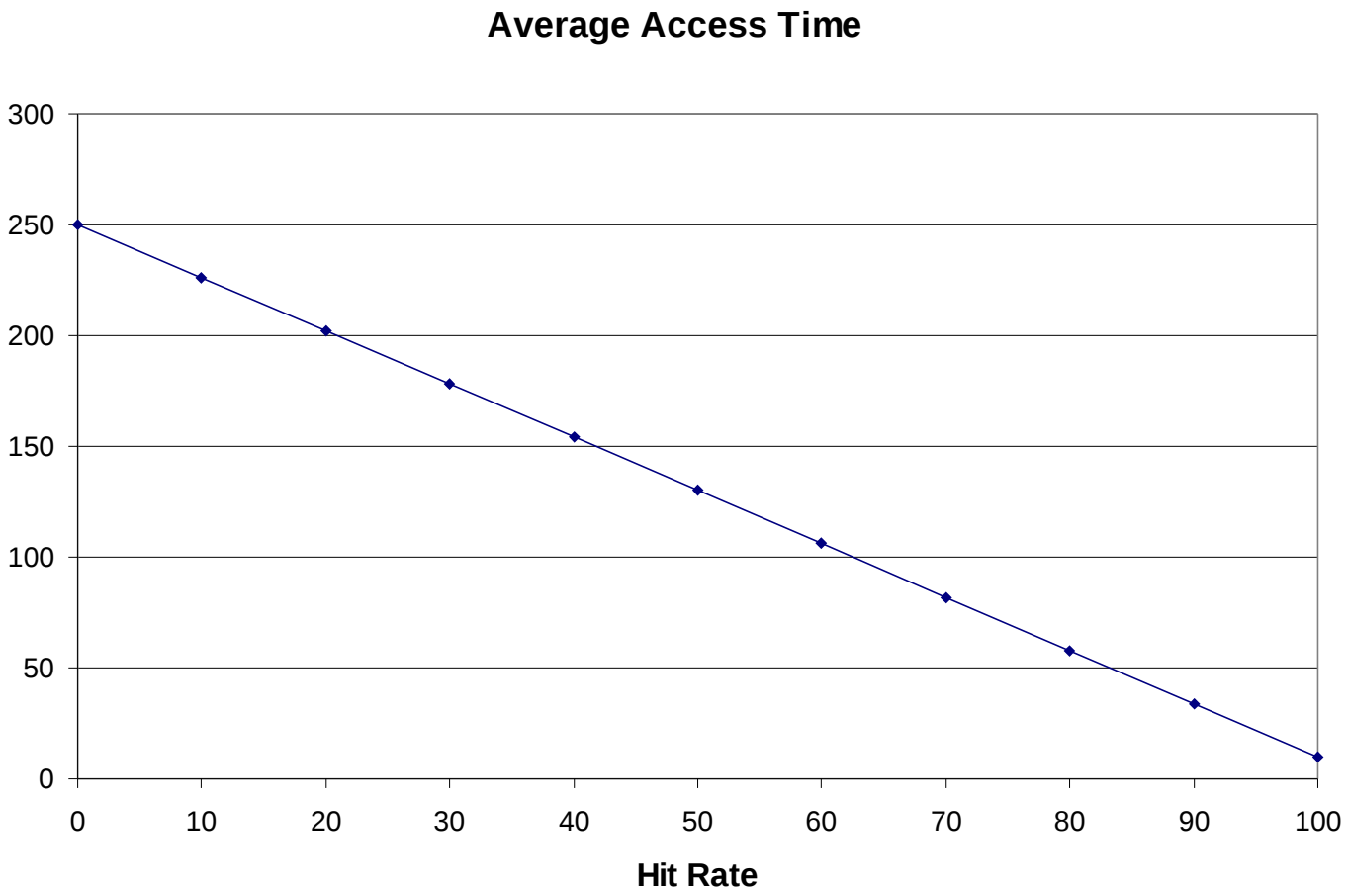




- **95% of references hit in cache**
- **Cache hits take 10 cycles**
- **Main memory references take 250 cycles**

$$T_{acc} = T_{hit} * hit\_ratio + T_{miss} * (1 - hit\_ratio)$$

$$T_{acc} = 10 * 0.95 + 250 * 0.05 = 22 \text{ cycles}$$

# Impact of Hit Rate

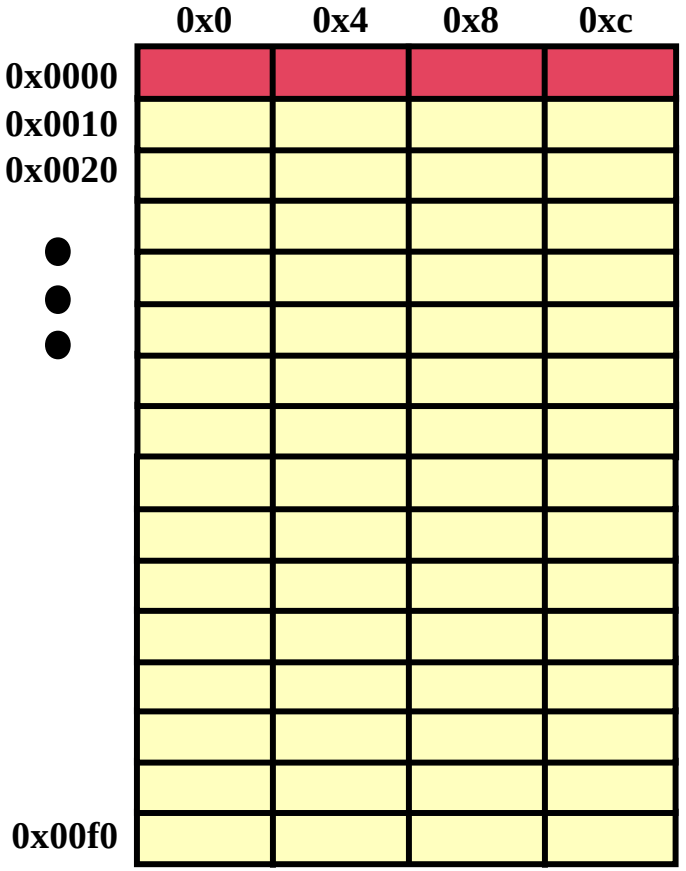


# Taking advantage of Spatial Locality

- Instead of each block in cache being just 1 word, what if we made it 4 words?
- When we get our 1 word instruction or 1 word of data from memory to put in the cache, get the next 3 as well, because they are likely to be used soon!
- Need to add a way to choose which of the 4 words in the block we want when we go to cache… called block offset.

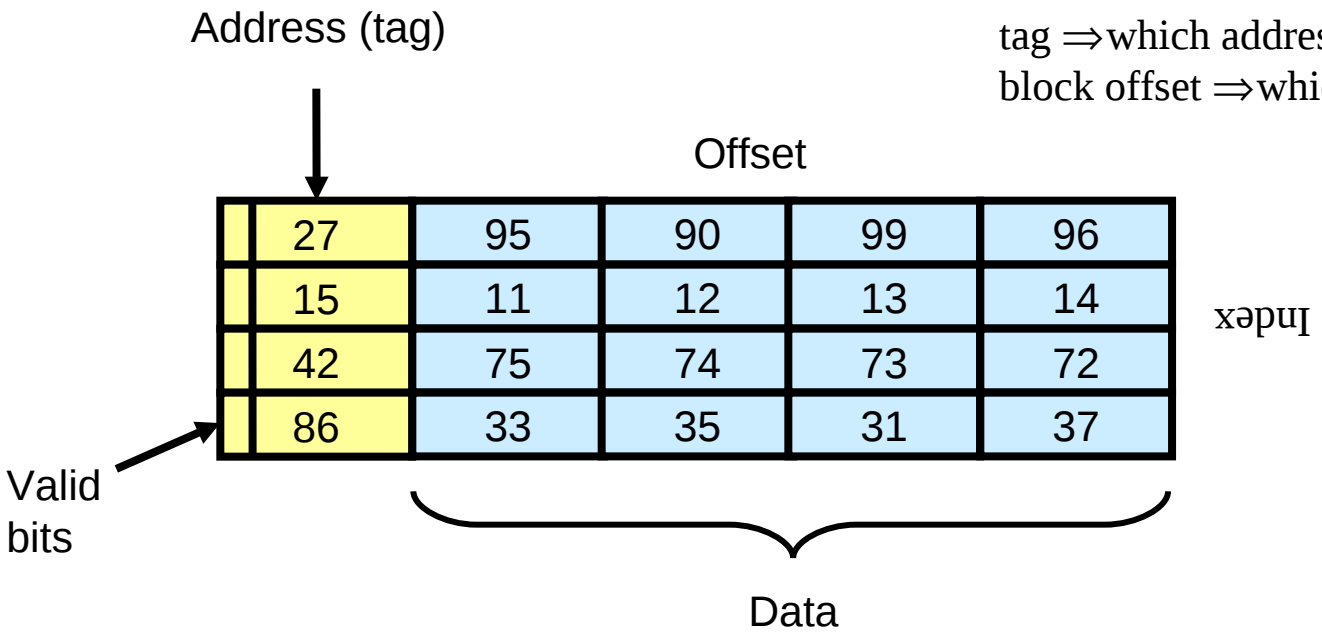
# Cache Definitions

- Cache block (= cache line)
- Index, Tag, Offset:
  - **Index** identifies the cache line corresponding to the data address
  - **Offset** gives a byte in a cache line
  - **Tag** compares with high order bits of the address to see if they match
- Miss rate, Miss penalty



# Cache Organization

valid bit ⇒is cache block good?
index ⇒which block in cache
tag ⇒which addresses in block
block offset ⇒which word in block



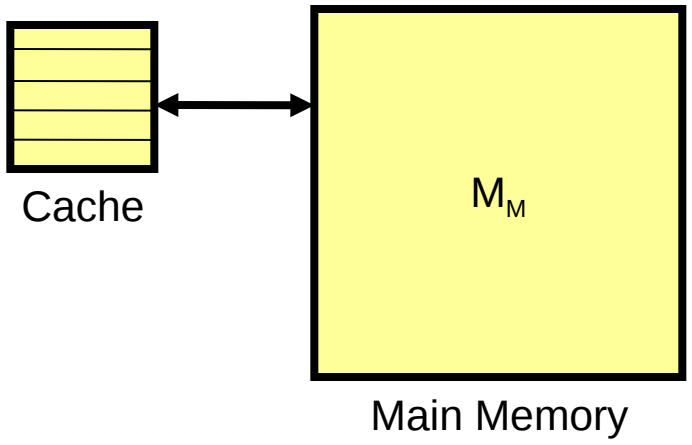

**Issues:**

- **Where does a block get placed?**
- **How do we find it?**
- **Which one do we replace when a new one is brought in?**
- **What happens on a write?**

# Block Placement

- Mapping function from Big Memory to Small memory
- On block-by-block basis
  - Direct Mapped: 1 place
  - Fully Associative: Anywhere
  - Set Associative: Subset of cache

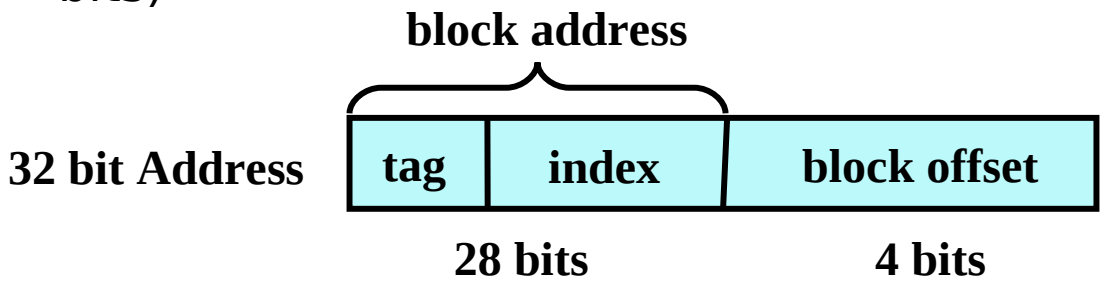
- Use address to do mapping and lookup



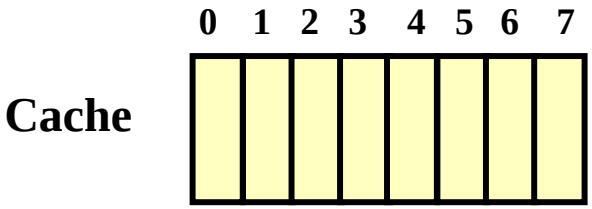
# Direct Mapped

- Each address has 3 fields: Tag, Index, Offset
  - Index: which line is mapped to
  - Offset: which byte within cache line
  - Tag: rest of address (to be compared)

- Example:
  - Main memory address space = 32 bits (= 4GBytes)
  - Block size = 4 words = 16 bytes (4 bits)
  - Cache capacity (index) = 8 blocks (3 bits) = 128 bytes (7 bits)

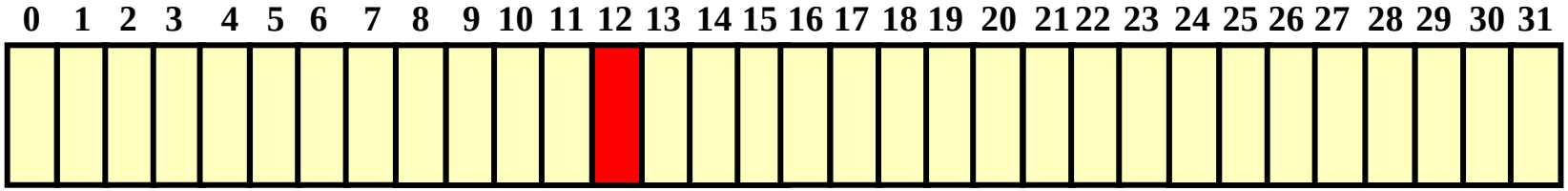


# Where Does a Block Go in the Cache?




- **Size = 8 word**
- **Block = 1 word**

**Main Memory**

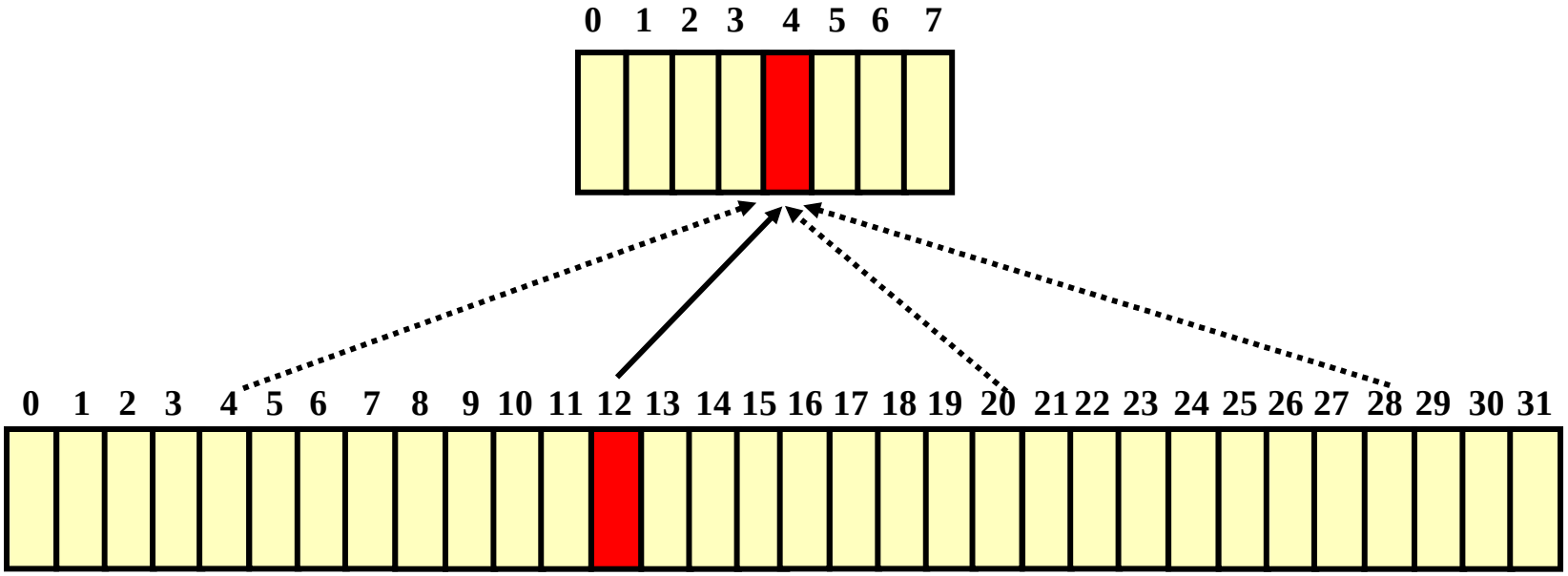



- Where do we put block 12?

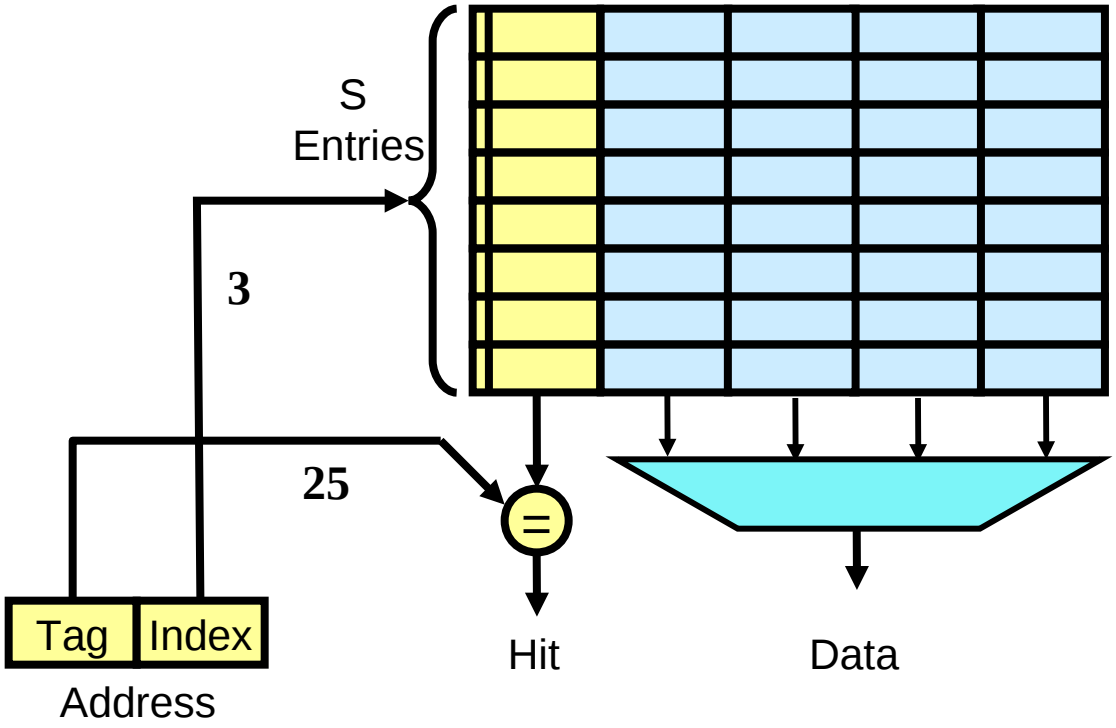
# Direct Mapped

- **Each block mapped to exactly 1 cache location**

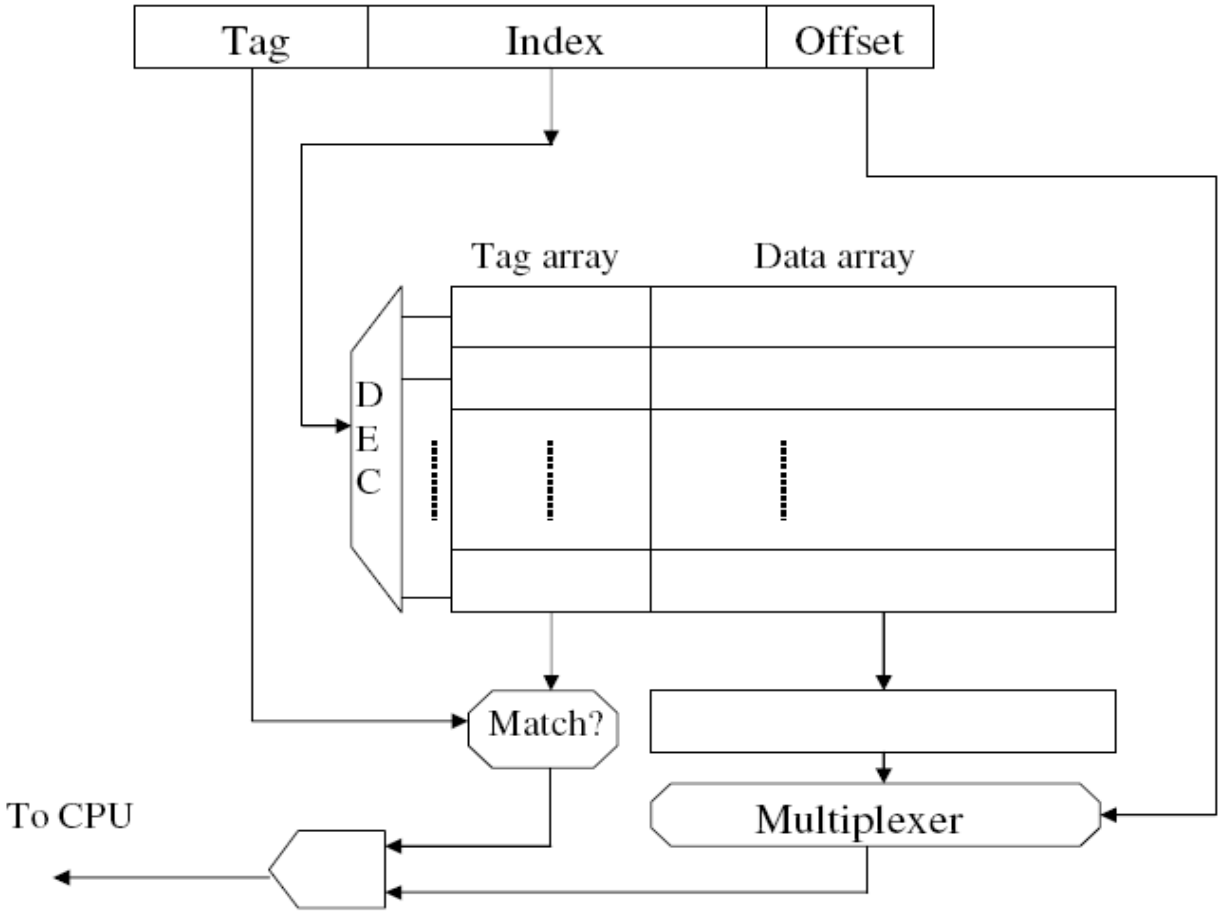
**Cache location = (block address) MOD (# blocks in cache)**



# Finding a Block: Direct-Mapped



# Hardware for a Direct Mapped Cache



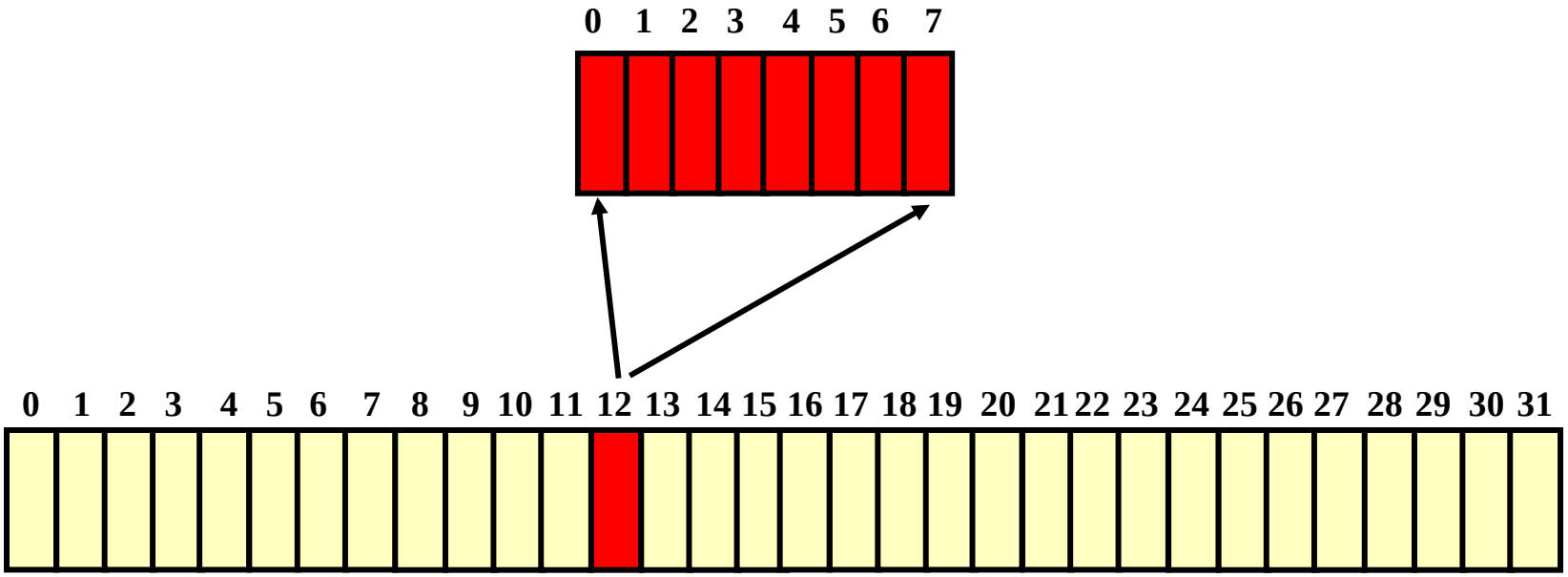
# Fully Associative

- Direct-mapped cache has high conflict misses
- Why not place a block at any free location?
- Fully associative cache:
  - No restriction as to where to place a cache line
  - each address has 2 fields: Tag & Offset
    - Cache is accessed by associative search, matching tags
  - No line interferences, only capacity misses

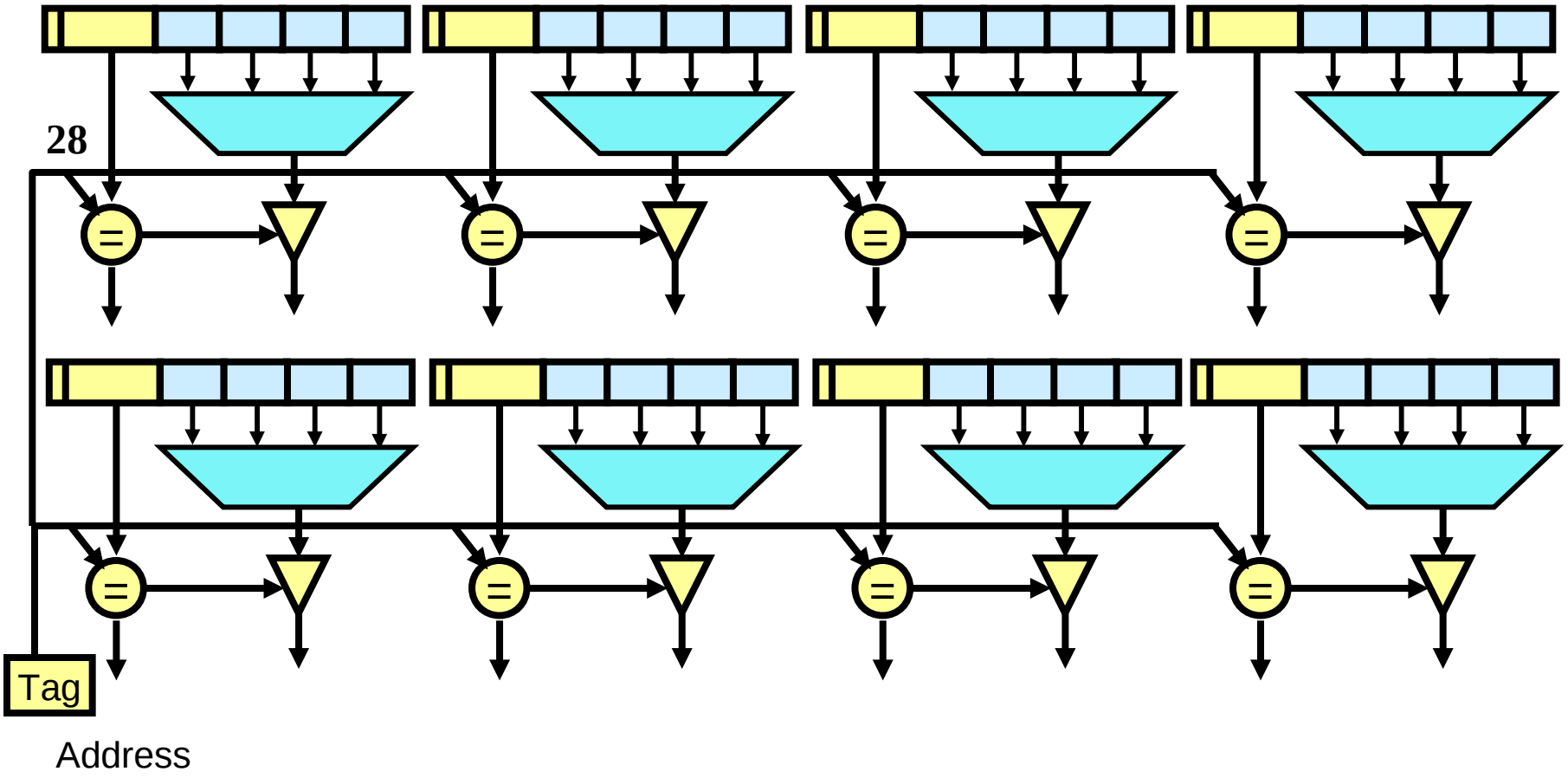
# Fully Associative

- **Each block mapped to any cache location**

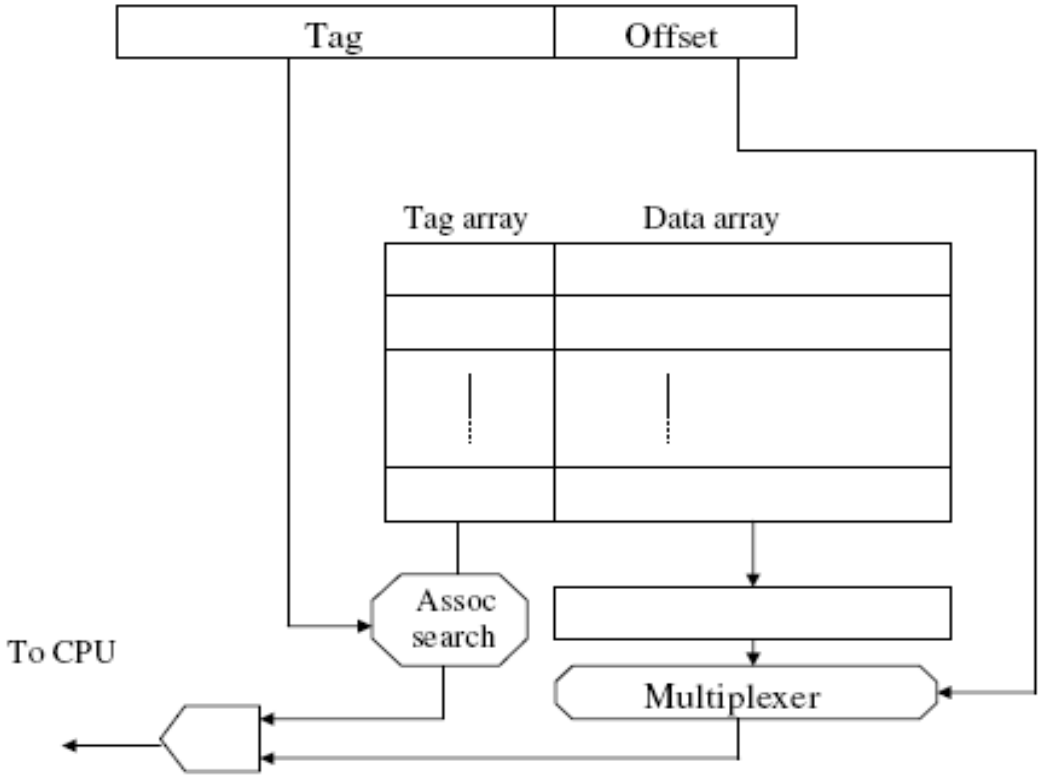
**Cache location = any**



# Finding A Block: Fully Associative



# Fully Associative



# Set Associative

- A compromise/hybrid of the previous two: associative map within a set and direct-map among sets
- Flexibility with a set to place data
  - Simple indexing logic to identify a set
  - d-way associative means d lines in a set
  - Cache lines are divided into groups => sets
- Reduce conflict misses and less complicated/faster than fully associative

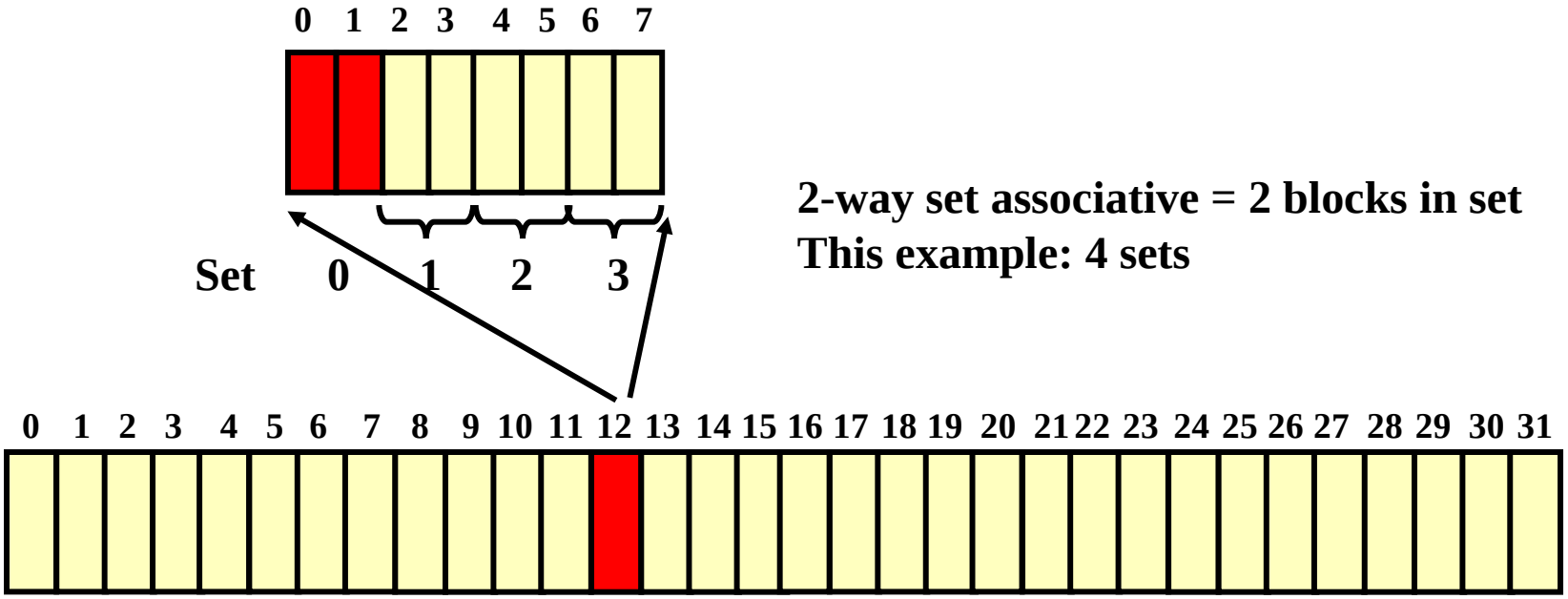
# Set Associative Cache

- S - sets
- A - elements in each set
  - A-way associative
- E.g: S=4, A=2
  - 2-way associative 8-entry cache (4 sets)
- All of main memory is divided into S sets
  - All addresses in set N map to same set of the cache
    - Addr = N mod S
    - “A” locations available in each set
- Shares costly comparators across sets
- Lower address bits select set
  - 2 in example
- Higher address bits are *tag*, used to associatively search the selected set
- Extreme cases
  - A=1: Direct mapped cache
  - S=1: Fully associative
- A need not be a power of 2

# Set Associative

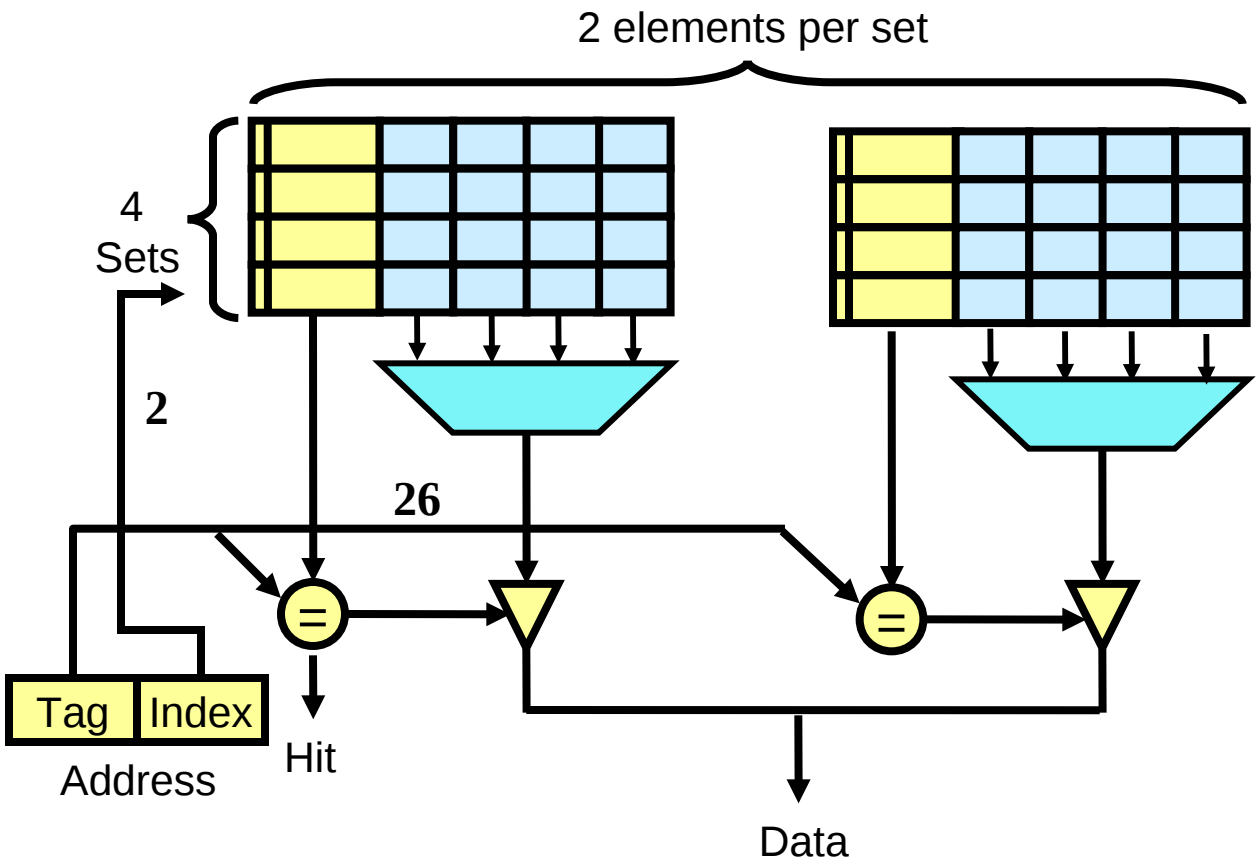
- **Each block mapped to subset of cache locations**

**Set selection = (block address) MOD (# sets in cache)**

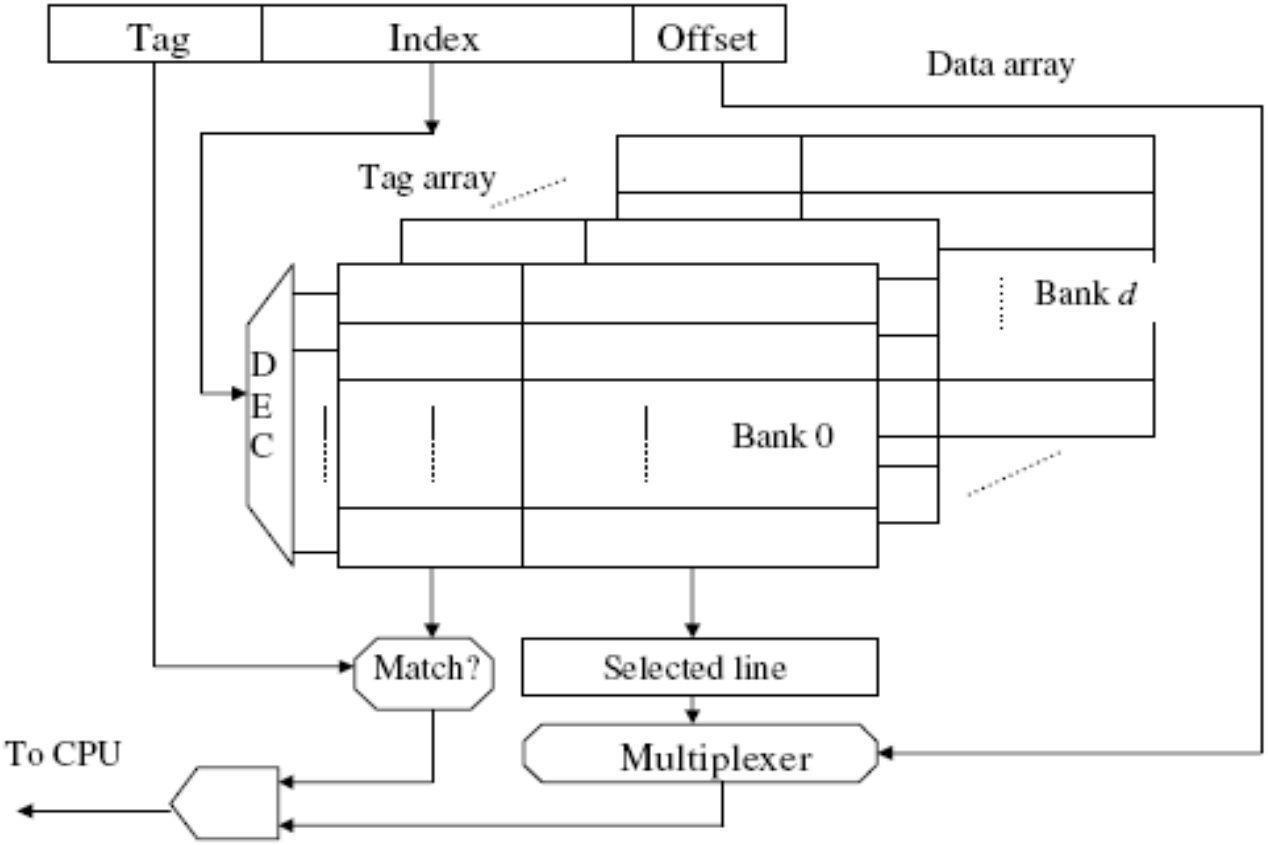



**2-way set associative = 2 blocks in set**
**This example: 4 sets**

# Finding A Block: 2-Way Set-Associative



# Set Associative



# Which Block Should Be Replaced on Miss?

- Direct Mapped
  - Choice is easy - only one option
- Associative
  - Randomly select block in set to replace (easy)
  - Least-Recently used (LRU) (hard)
  - Least-Recently entered
  - Pseudo LRU

# What Happens on a Store?

- Need to keep cache consistent with main memory
  - Reads are easy - no modifications
  - Writes - when do we update main memory?

- Write-Through
  - On cache write - always update main memory as well
  - Use a write buffer to enqueue writes to main memory

- Write-Back
  - On cache write - remember that block is modified (dirty bit)
  - Update main memory when dirty block is replaced
  - Sometimes need to flush cache (I/O, multiprocessing)

# If Store Causes Miss

- Write-Allocate
  - Bring written block into cache
  - Update word in block
  - Anticipate further use of block

- No-write Allocate
  - Main memory is updated
  - Cache contents unmodified
  - Useful with with Write-Through caches

# Miss Classifications

- Compulsory misses
  - First time data is accessed
- Capacity misses
  - When working set is larger than cache size
- Conflict misses
  - One set fills up, but room in other sets

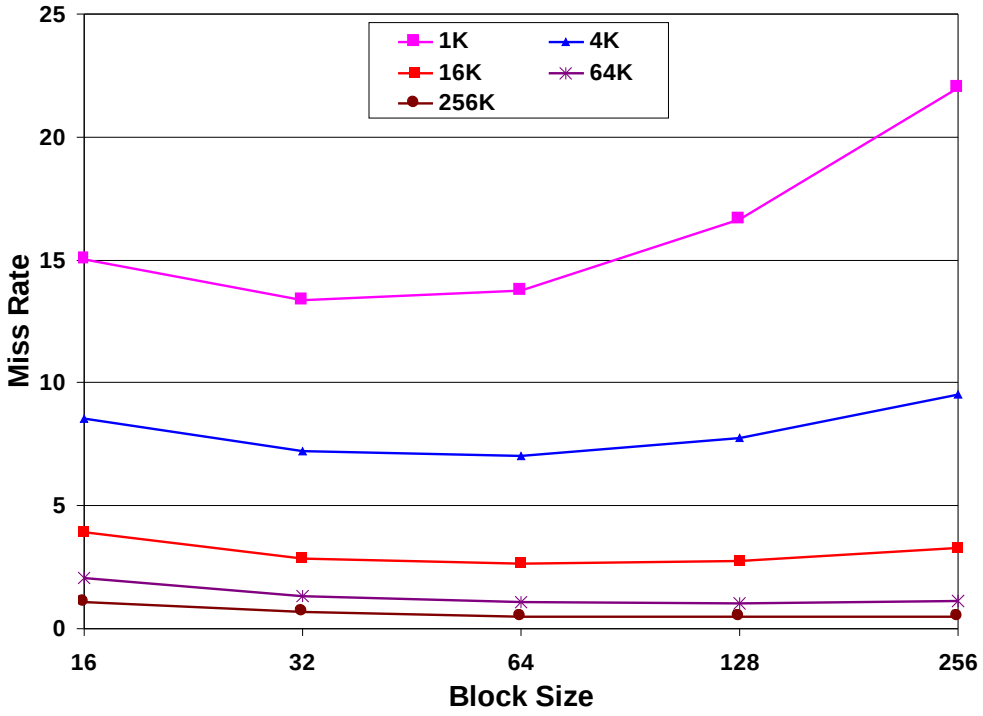
# How Do We Improve Cache Performance?

$$T_{access} = t_{hit} + rate_{miss} \times penalty_{miss}$$

- Reduce miss rate
- Reduce miss penalty
- Reduce hit time

# Reducing Miss Rate: Increase Block Size

- Fetch more data with each cache miss
  - 16 bytes ⇒ 64, 128, 256, 512 bytes
  - Works because of Locality (spatial)



# Reducing Miss Rate: Increase Associativity

- Reduce conflict misses
- Rules of thumb
  - 8-way = fully associative
  - Direct mapped size N = 2-way set associative size N/2

- However:
  - Size N associative is larger than Size N direct mapped
  - Associative typically slower that direct mapped ($t_{hit}$ larger)

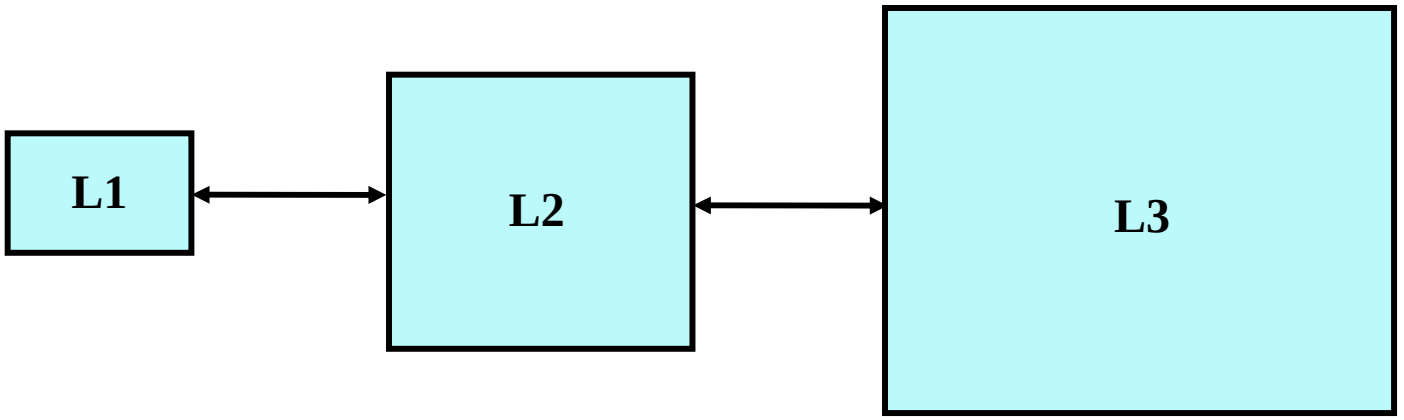
# Reduce Miss Penalty: More Cache Levels

Average access time = $HitTime_{L1}$ + $MissRate_{L1}$ * $MissPenalty_{L1}$

$MissPenalty_{L1}$ = $HitTime_{L2}$ + $MissRate_{L2}$ * $MissPenalty_{L2}$

... etc ...

- Size/Associativity of higher level caches ?



# Reduce Miss Penalty: Transfer Time

- Can increase the width of memory and the bus in order to decrease access times and transfer times to move things into cache.
- *Interleaving:* multiple memory “banks” allow to read or write multiple words in 1 access time
- The critical word can be *forwarded* to CPU while fetching the rest of the line

# Reducing Hit Time

- Make Caches small and simple
  - Hit Time = few cycles is good
  - L1 - low associativity, relatively small
- Even L2 caches can be broken into sub-banks
  - Can exploit this for faster hit time in L2