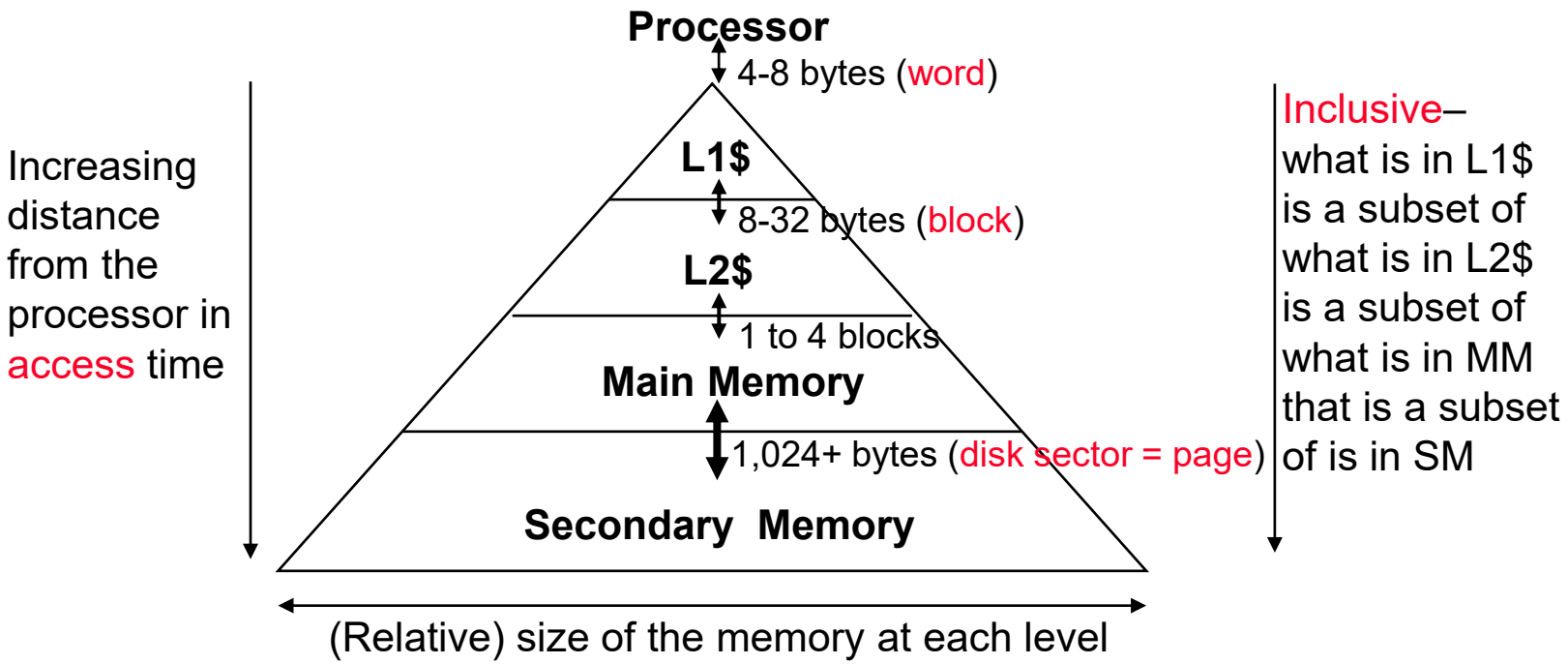
ELT3047 Computer Architecture

Lecture 13: Memory (cont.)

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

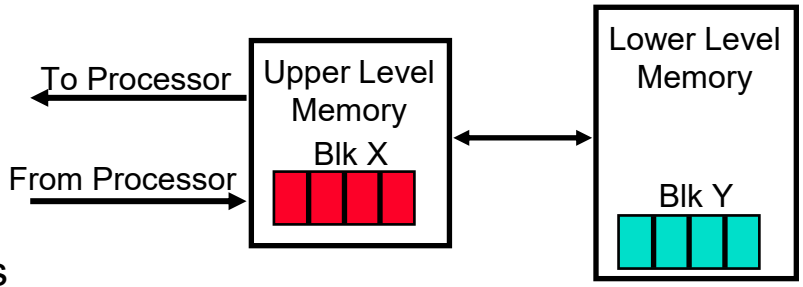
Last lecture review (1)

- The Memory Hierarchy
  - 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.



Last lecture review (2)

- Temporal Locality
  - Keep **most recently** accessed data items **closer** to the processor.
- Spatial Locality
  - Move **blocks** consisting of contiguous words to the upper levels



- Hit Time << Miss Penalty
  - **Hit**: data appears in some block in the upper level (Blk X)
    - **Hit rate**: fraction of accesses found in the upper level
    - **Hit Time**: RAM access time + time to determine hit/miss
  - **Miss**: data needs to be retrieve from a lower level block (Blk Y)
    - **Miss rate**: $\frac{\#\text{miss}}{\#\text{accesses}} = 1 - (\text{Hit rate})$
    - **Miss penalty:** Time to replace a block in the upper level with a block from the lower level + Time to deliver this block's word to the processor
    - **Miss types**: Compulsory, Conflict, Capacity

Measuring Cache Performance

- The processor stalls on a cache miss
  - When fetching instructions from the **Instruction Cache (I-cache)**
  - When loading or storing data into the **Data Cache (D-cache)**
  - Miss penalty is assumed equal for I-cache & D-cache
  - Miss penalty is assumed equal for Load and Store
- Components of CPU time:
  - Program execution cycles (includes cache hit time)
  - Memory stall cycles (mainly from cache misses)
  - CPU time = IC × CPI × CC = IC × ($\underbrace{(CPI_{ideal} + \text{Memory-stall cycles})}_{CPI_{stall}}$) × CC
    - CPI_{ideal} = CPI for ideal cache (no cache misses)
    - CPI_{stall} = CPI in the presence of memory stalls
    - Memory stall cycles **increase** the CPI!

Memory Stall Cycles

- Sum of read-stalls and write-stalls (due to cache misses)
  - 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
- Memory stall cycles = (I-Cache Misses + D-Cache Misses) × Miss Penalty
  - I-Cache Misses = I-Count × I-Cache Miss Rate
  - D-Cache Misses = LS-Count × D-Cache Miss Rate
    - LS-Count (Load & Store) = I-Count × LS Frequency
- With simplifying assumptions:

  Memory stall cycles = I-Count x misses/instruction x miss penalty

  I-Cache Miss Rate + LS Frequency × D-Cache Miss Rate
  - Memory stall cycles/**instruction** = I-Cache Miss Rate × Miss Penalty + LS Frequency × D-Cache Miss Rate × Miss Penalty
  - For **write-through** caches: Memory-stall cycles = miss rate × miss penalty

Memory Stall Cycles: example

- **Example**: Compute misses/instruction and memory stall cycles for a program with the given characteristics
  - Instruction count (I-Count) = 10^6 instructions
  - 30% of instructions are loads and stores
  - D-cache miss rate is 5% and I-cache miss rate is 1%
  - Miss penalty is 100 clock cycles for instruction and data caches
- **Solution**:
  - misses/instruction=1%+30%x5%=0.025;
  - memory stall cycles/instruction=0.025x100=2.5 cycles
  - total memory stall cycles=2.5x10^6=2,500,000 cycles

Impacts of Cache Performance

- **Relative** cache penalty *increases* as processor performance *improves* (faster clock rate and/or lower CPI)
  - 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
- **Example:** Given
  - I-cache miss rate = 2%, D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions

  Questions:
  - What is CPI_{stall}? 2+(2%+36%x4%)x100 = 5.44, % time on memory stall = 63%
  - What if the CPI_{ideal} is reduced to 1? % time on memory stall = 77%
  - What if the processor clock rate is doubled? Miss penalty = 200, CPI_{stall} = 8.88

Average Memory Access Time (AMAT)

- Hit time is also important for performance
  - A larger cache will have a longer access time → an increase in hit time will likely add another stage to the pipeline.
  - At some point, the increase in hit time for a larger cache will overcome the improvement in hit rate leading to a decrease in performance.
- Average Memory Access Time (**AMAT**) is the average time to access memory considering both hits and misses.

  AMAT = Hit time + Miss rate × Miss penalty
- **Example**: Find the AMAT for a cache with
  - Cache access time (Hit time) of 1 cycle = 2 ns
  - Miss penalty of 20 clock cycles
  - Miss rate of 0.05 per access
- **Solution:**
  - AMAT = 1 + 0.05 × 20 = 2 cycles = 4 ns
  - Without the cache, AMAT will be equal to miss penalty = 20 cycles = 40 ns

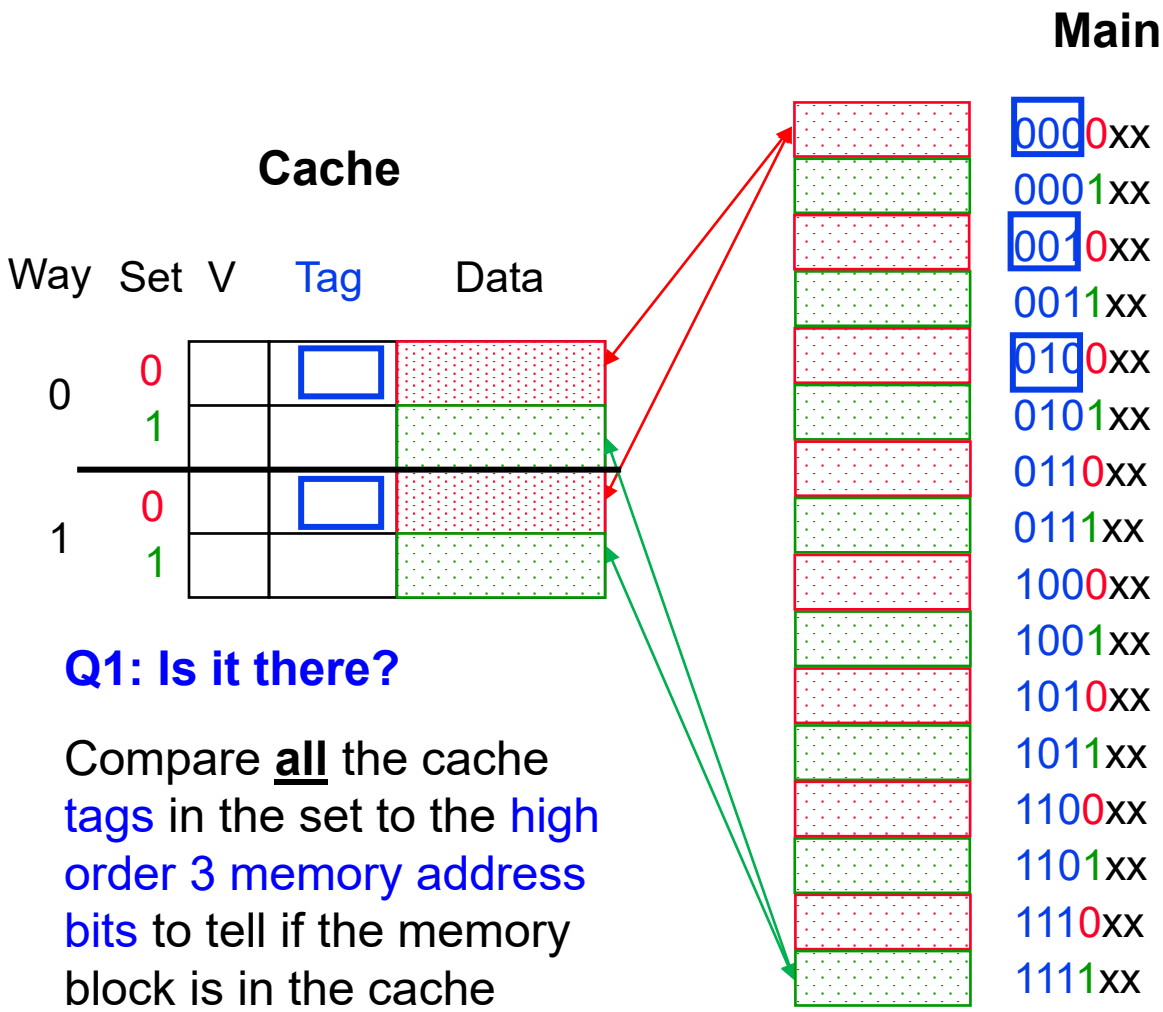
Reducing cache miss rates #1: cache associativity

- 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** (no indexing)
- 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).

    Set index = (block address) modulo (# sets in the cache)
- **Example**: consider the main memory word reference for the following string

  0 4 0 4 0 4 0 4
  - Start with an empty cache - all blocks initially marked as not valid

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

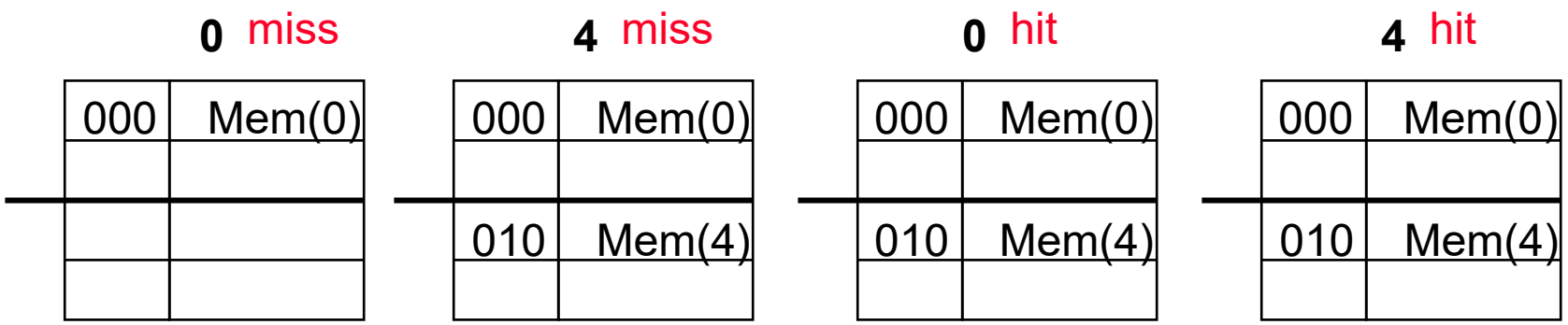
One word blocks
Two low order bits define the byte in the word (32b words)

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)

Set associative cache example: reference string mapping

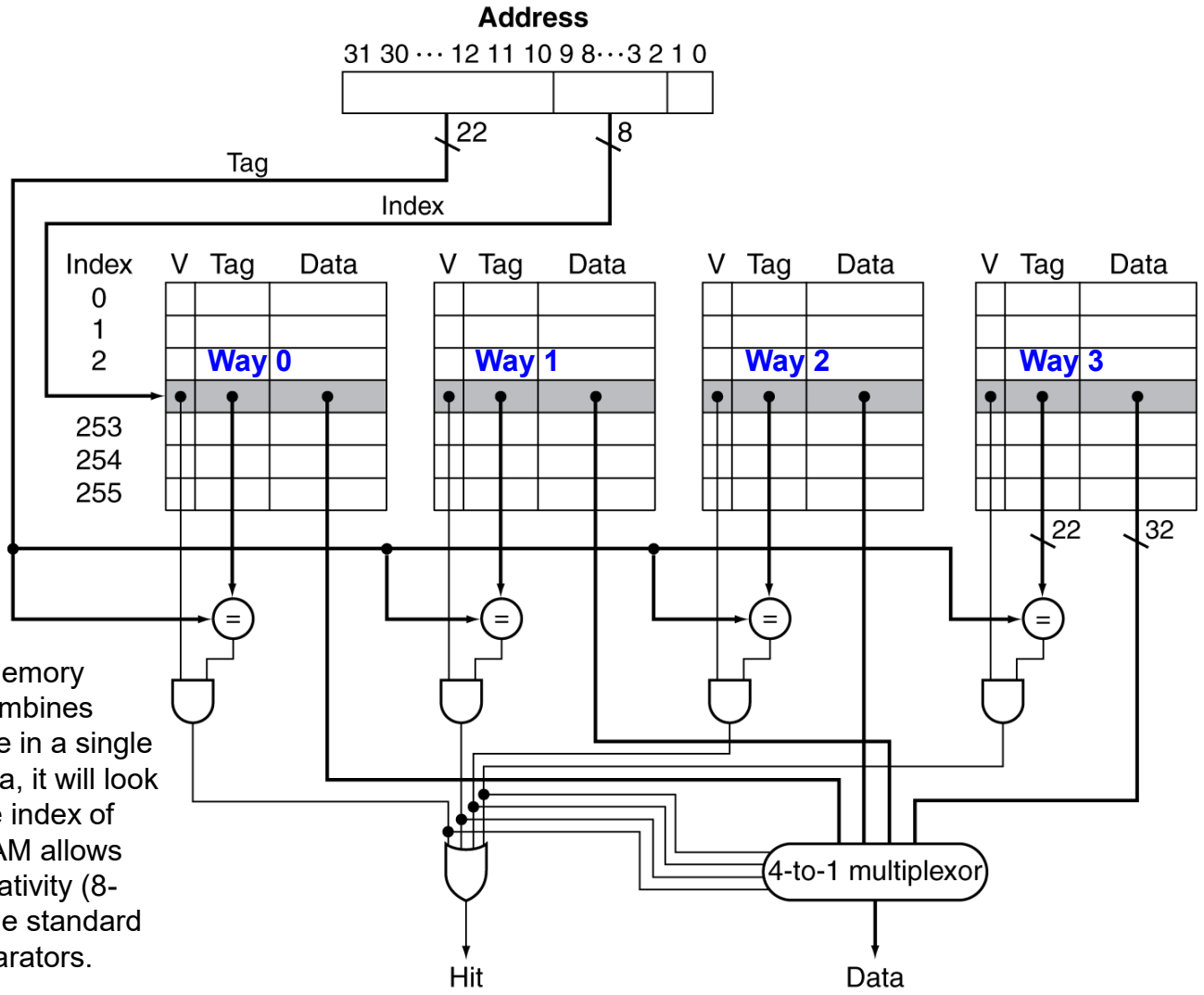
0 4 0 4 0 4 0 4



- 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!

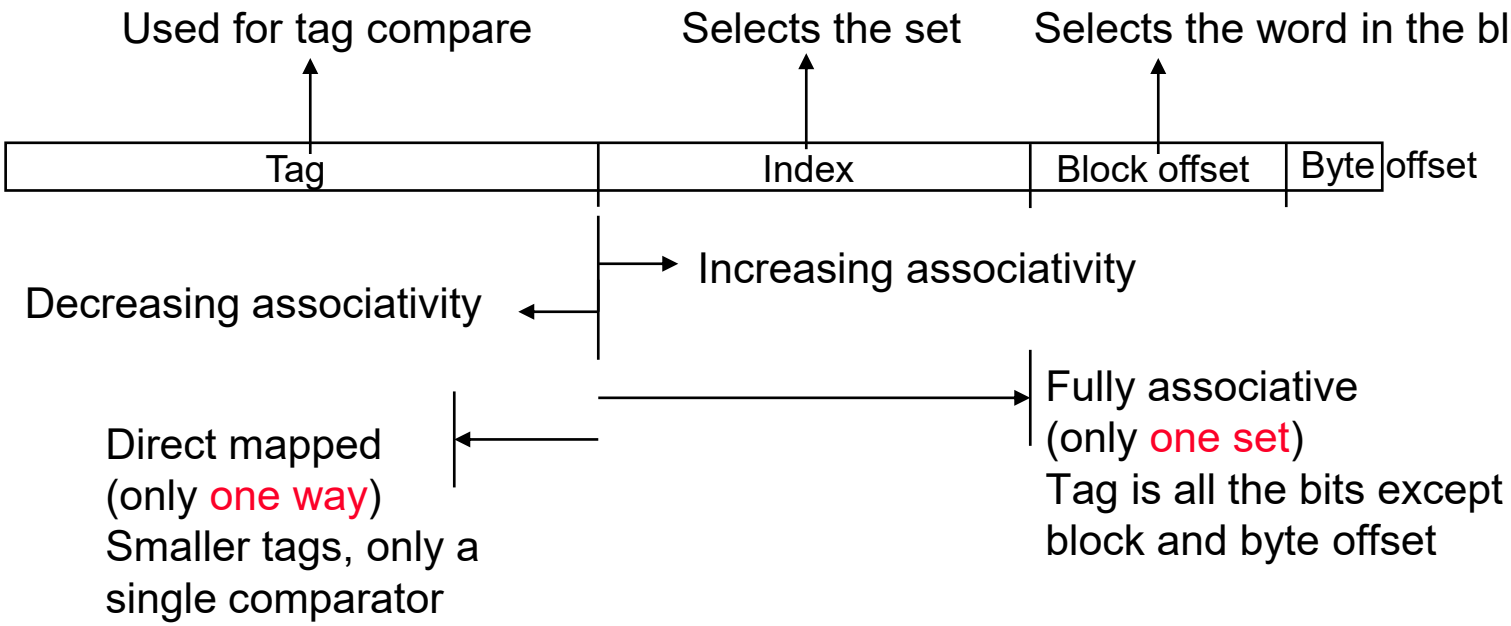
Four-Way Set Associative Cache Organization

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



Content Addressable Memory (**CAM**): a circuit that combines comparison and storage in a single device - supply the data, it will look for a copy & returns the index of the matching row → CAM allows much higher set associativity (8-way and above) than the standard HW of SRAMs + comparators.

Range of Set Associative Caches



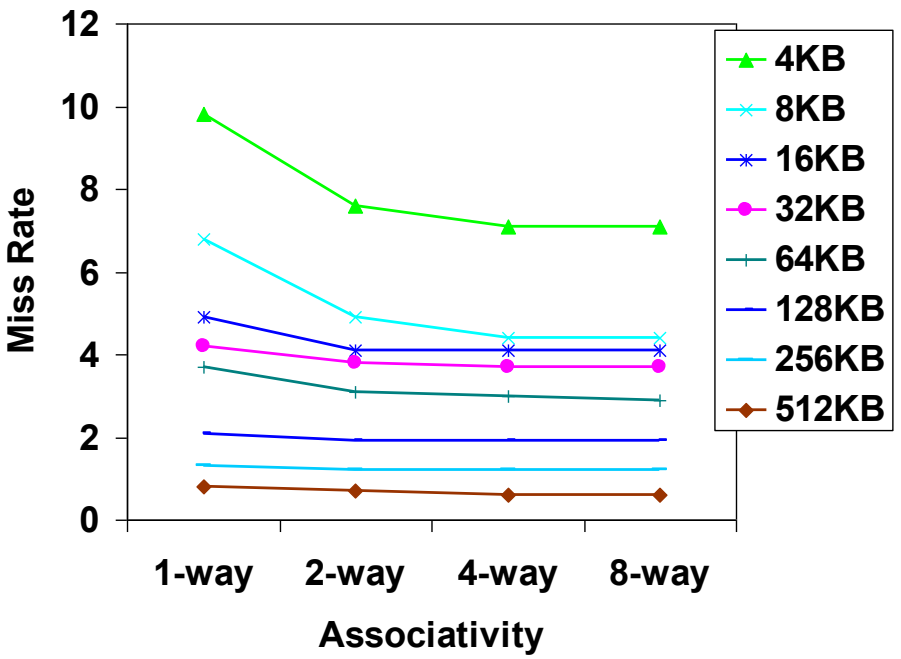
- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (= the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

Replacement Policy

- A miss occurred, which way's block do we pick for replacement?
  - Direct mapped: **no** choice.
  - Set associative: non-valid entry, then choose among entries in the set.
- First In First Out (**FIFO**): replace the oldest block in set
  - Use one counter per set to specify the oldest block. On a cache miss replace the block specified by counter & increment the counter.
- Least Recently Used (**LRU**): replace the one that has been unused for the longest time
  - Requires 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)
  - Manageable for 4-way, too hard beyond that.
- **Random**
  - Gives approximately the same performance as LRU for high associativity.

How Much Associativity?

- Increased associativity decreases miss rate
  - But with diminishing returns
- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation.
- 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 (c.f. direct mapped cache: the cache block is available **before** the Hit/Miss decision) → can be an important consideration (why?).



Reducing Cache Miss Rates #2: multi-level caches

- Use multiple levels of caches
  - Primary (L1) cache attached to CPU
  - Larger, slower, L2 cache services misses from primary cache. With advancing technology → have more than enough room on the die for L2, normally a unified cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache.

- **Example:** Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns

  Questions:
  - Compute the actual CPI with just primary cache.
  - Compute the performance gain if we add L2 cache with
    - Access time = 5ns
    - Global miss rate to main memory = 0.5%

Multi-level cache: example solution

- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - $CPI_{stall} = 1 + 0.02 \times 400 = 9$
- With added L2 cache
  - Primary miss with L2 hit: penalty = 5ns/0.25ns = 20 cycles
  - Primary miss with L2 miss: penalty = L2 access stall + Main memory stall = 20 + 400 = 420 cycles
  - $CPI_{stall} = 1 + (0.02 - 0.005) \times 20 + 0.005 \times 420 = 3.4$ cycles
  - [Alternatively, CPI_{stall} = 1 + L1 stalls/instruction + L2 stalls/instruction = 1 + 0.02 x 20 + 0.005 x 400 = 3.4 cycles]
  - Performance gain = 9/3.4=2.6 times.

Multilevel Cache Design Considerations

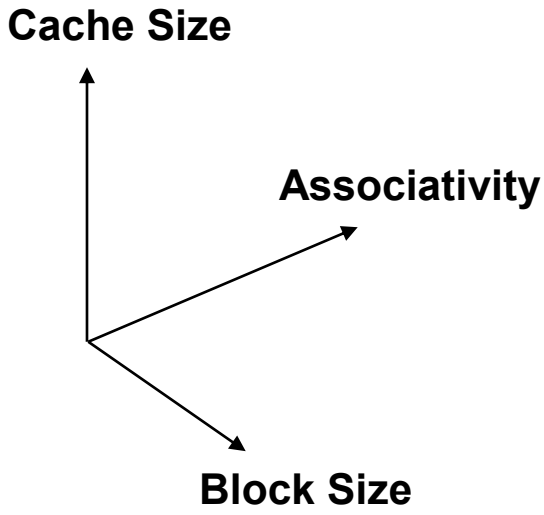
- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on **minimizing hit time** in support of a shorter clock cycle → smaller with smaller block sizes.
  - Secondary cache(s) should focus on **reducing miss rate** to reduce the penalty of long main memory access times → larger with larger block sizes & higher levels of associativity.
- 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** >> the **global miss rate**
    - Local miss rate = fraction of references to one level of a cache that miss
    - Global miss rate = fraction of references that miss in all levels of a multi-level cache → dictates how often we must access the main memory.

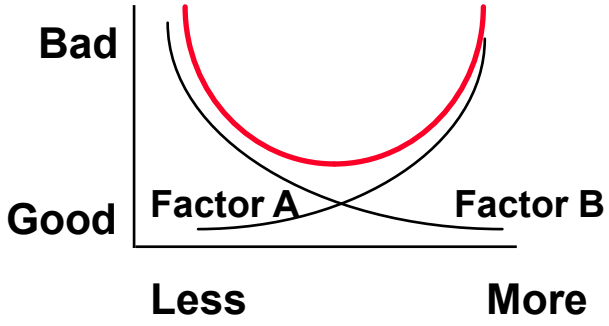
Multi-level cache parameters: two real-life examples

	Intel Nehalem	AMD Barcelona
L1 cache organization & size	Split I$ and D$; 32KB for each per core; 64B blocks	Split I$ and D$; 64KB for each per core; 64B blocks
L1 associativity	4-way (I), 8-way (D) set assoc.; ~LRU replacement	2-way set assoc.; LRU replacement
L1 write policy	write-back, write-allocate	write-back, write-allocate
L2 cache organization & size	Unified; 256MB (0.25MB) per core; 64B blocks	Unified; 512KB (0.5MB) per core; 64B blocks
L2 associativity	8-way set assoc.; ~LRU	16-way set assoc.; ~LRU
L2 write policy	write-back, write-allocate	write-back, write-allocate
L3 cache organization & size	Unified; 8192KB (8MB) shared by cores; 64B blocks	Unified; 2048KB (2MB) shared by cores; 64B blocks
L3 associativity	16-way set assoc.	32-way set assoc.; evict block shared by fewest cores
L3 write policy	write-back, write-allocate	write-back; write-allocate

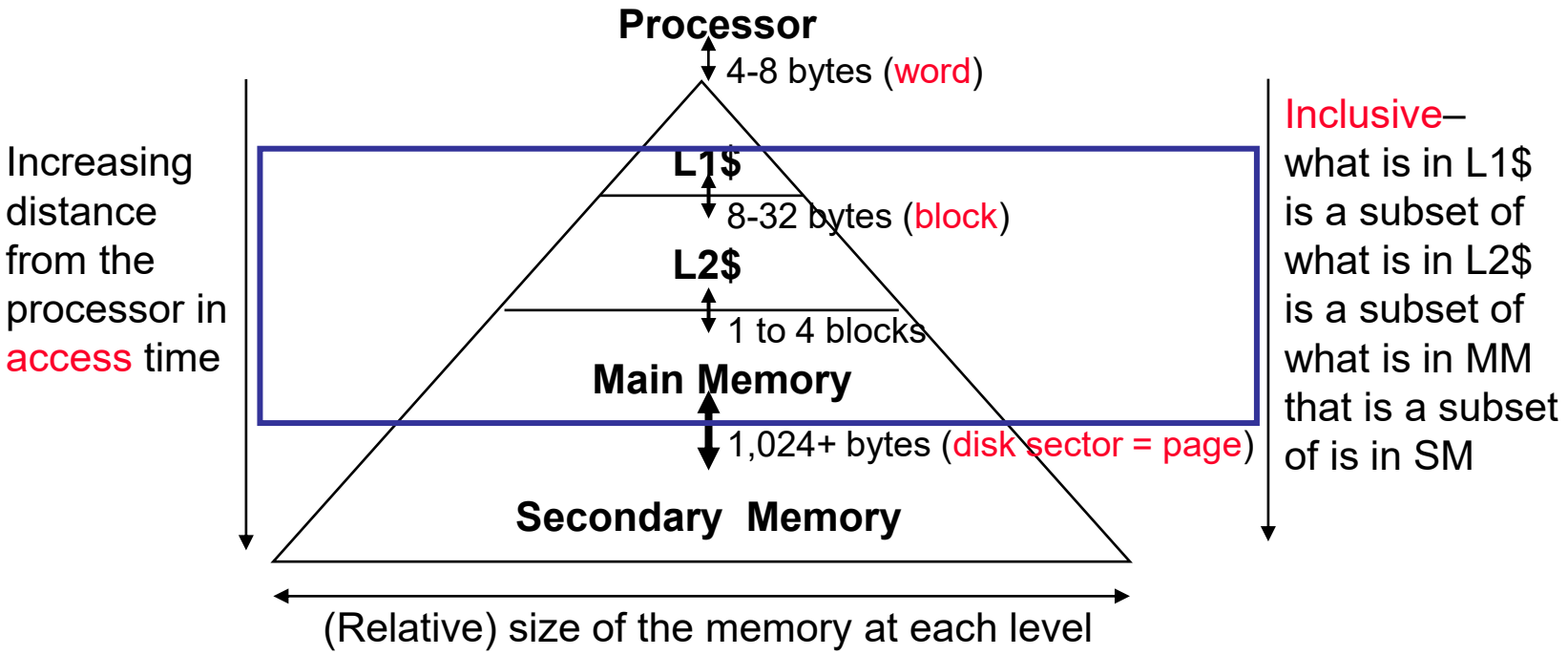
The Cache Design Space

- 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





Memory: the next hierarchy

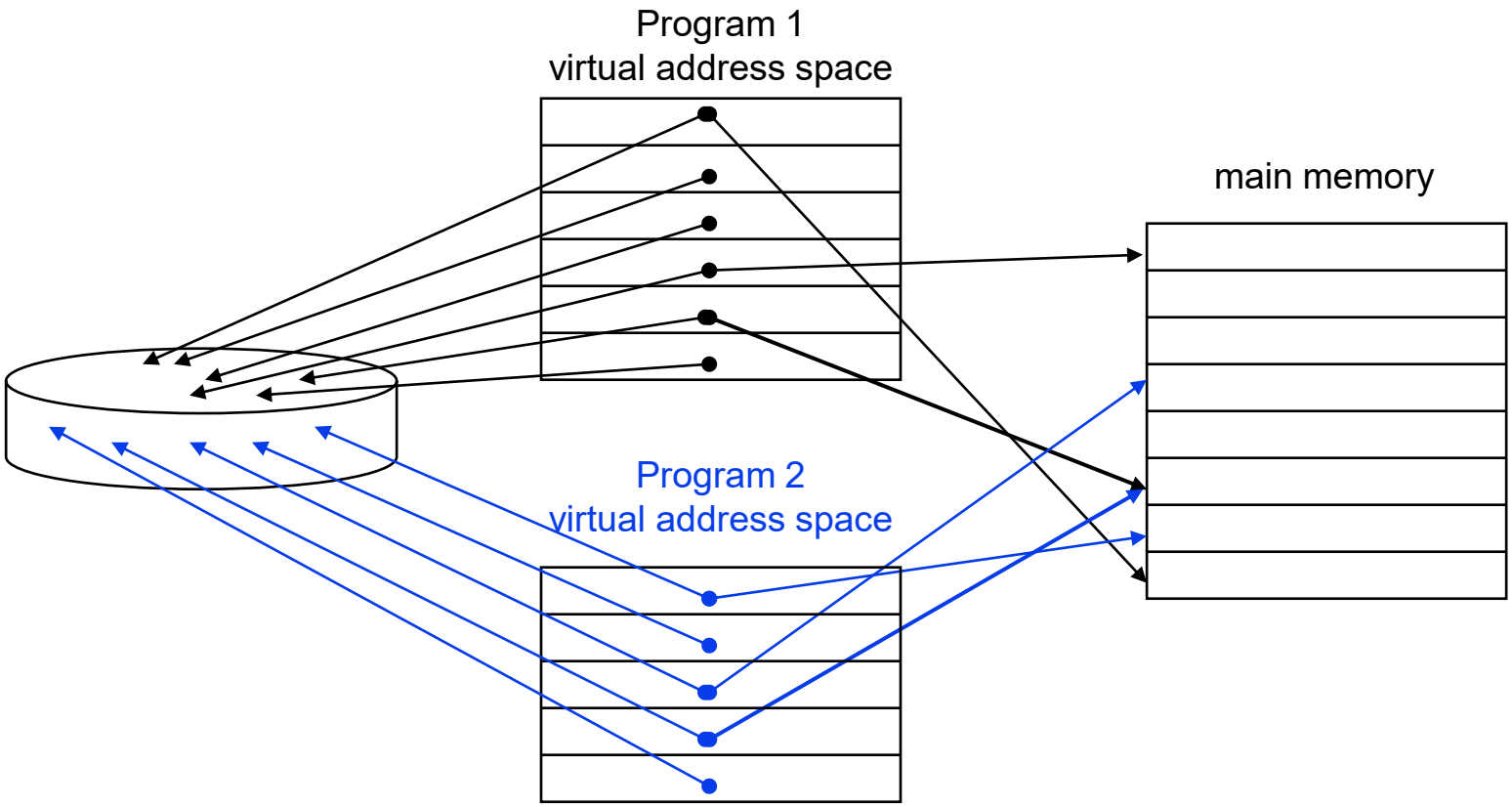


Virtual Memory

- A technique that uses RAM as a "cache" for secondary storage
  - Allows efficient and **safe** sharing of memory among multiple programs
  - Provides the ability to run programs larger than the size of physical memory
  - Simplifies loading a program for execution by enabling code relocation.
- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a "**virtual**" address space
  - The processor generates virtual addresses while the memory is accessed using **physical** addresses (real locations in main memory) → each virtual address **must** be translated to a physical address.
  - Some chunks of virtual memory can be present on disk, not in main memory.
  - Multiple programs can use (different chunks of physical) memory at same time.

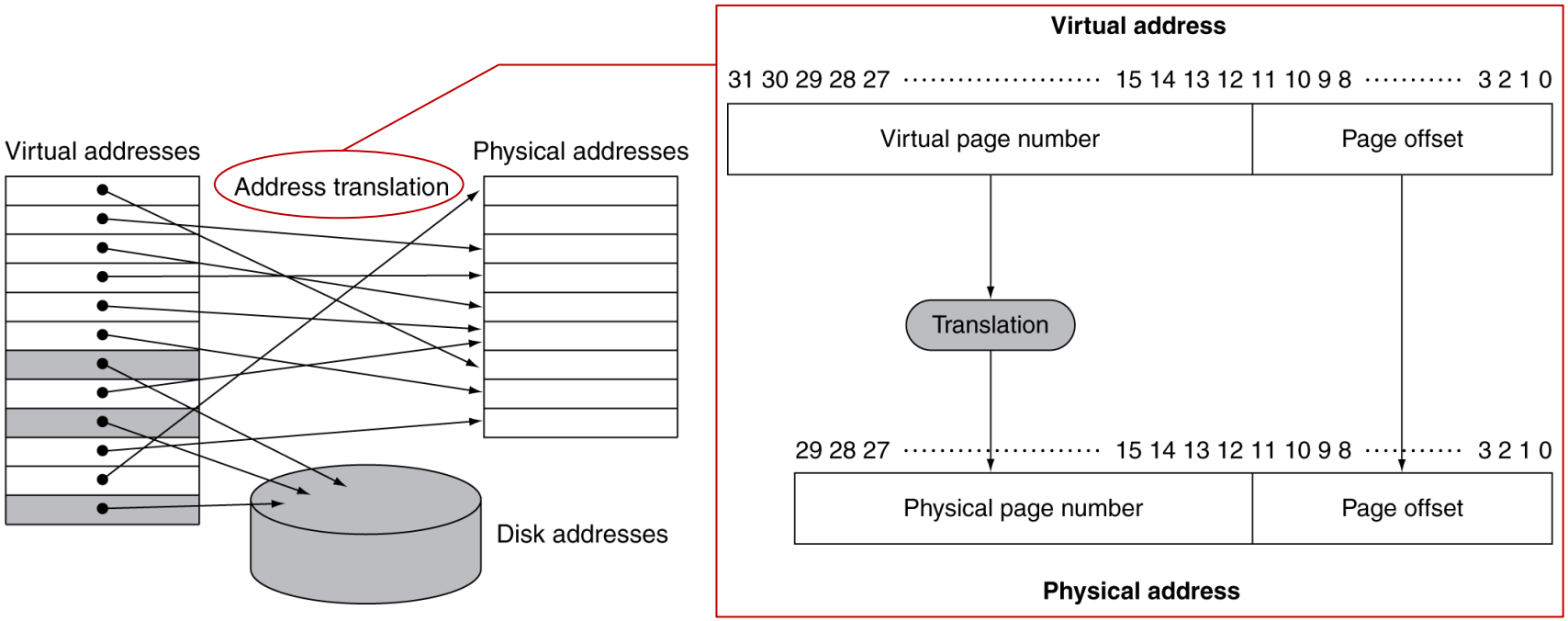
Virtual memory: two programs sharing physical memory

- A program's address space is divided into **pages** (all one fixed size) or **segments** (variable sizes)
  - The starting location of each page (either in main memory or in secondary memory) is contained in the program's **page table**

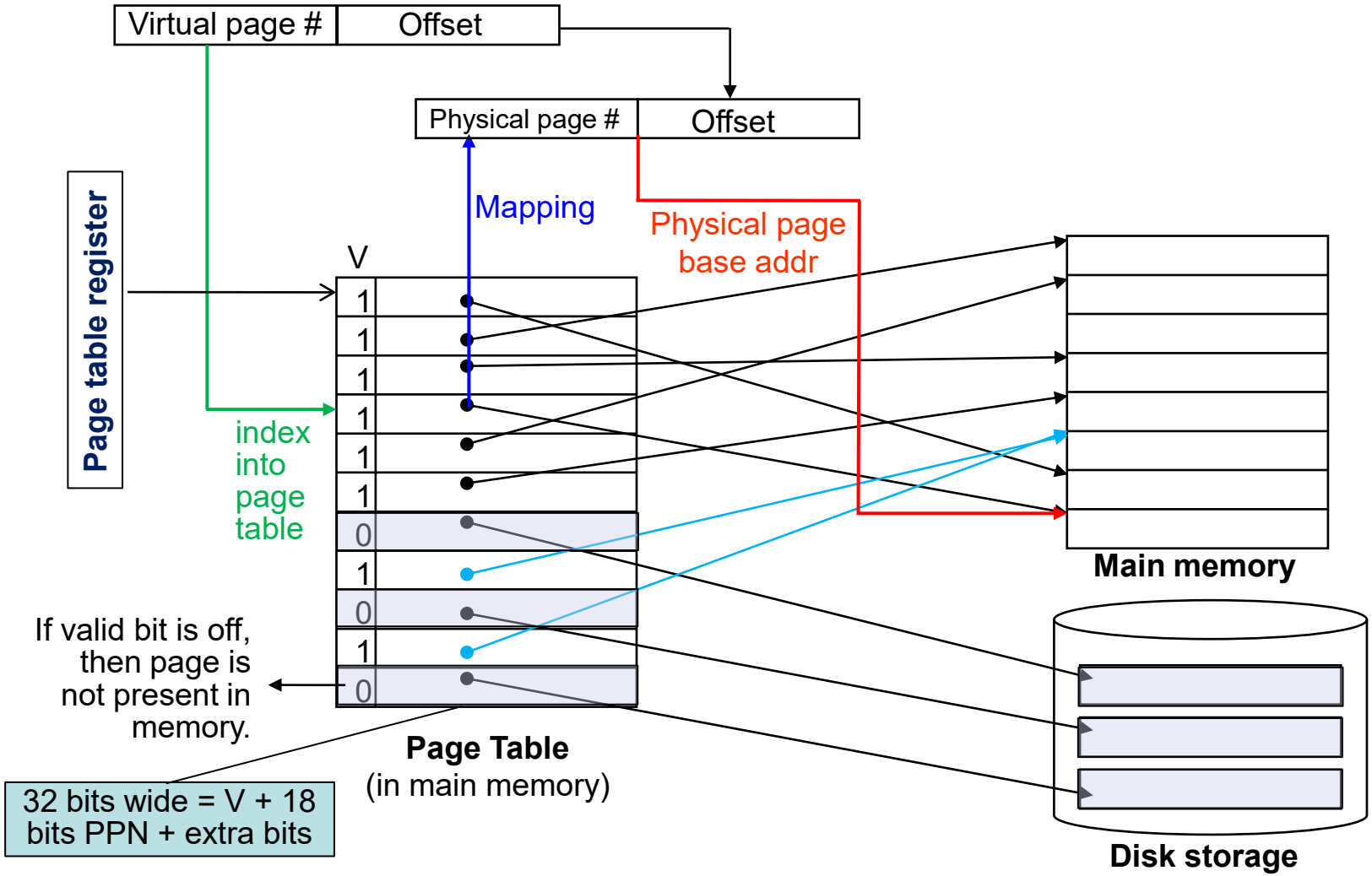


Virtual memory: address translation

- Assuming fixed-size pages, each memory request **first** requires an address **translation** from virtual space to physical space
  - Done by a combination of hardware and software
  - **page fault**: virtual memory miss (i.e., the page is not in physical memory). Page fault penalty is very costly, often takes millions of clock cycles



Address Translation Mechanisms (1)



Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - **Reference bit** (aka use bit) in the page table entry set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 means it has not been used recently
- Disk writes take millions of cycles
  - **Block** at once, not individual locations
  - Write through is impractical
  - Use **write-back**
  - **Dirty bit** in the page table entry set when page is written

Handling page fault & space optimization

- A page fault is like a cache miss
  - Must find page in lower level of hierarchy
  - If valid bit is zero, the Physical Page Number points to a page on disk
- When OS starts new process, it creates space on disk for all the pages of the process (all valid bits in page table = zero)
  - called **Demand Paging** - pages of the process are loaded from disk only as needed
- Page Table too big!
  - 4GB Virtual Address Space ÷ 4 KB page
  - 1 million Page Table Entries ≈ 4 MB just for Page Table of a single process!
- Variety of solutions to tradeoff Page Table size for slower performance
  - E.g., Multi-level page table, Paging page tables, etc.

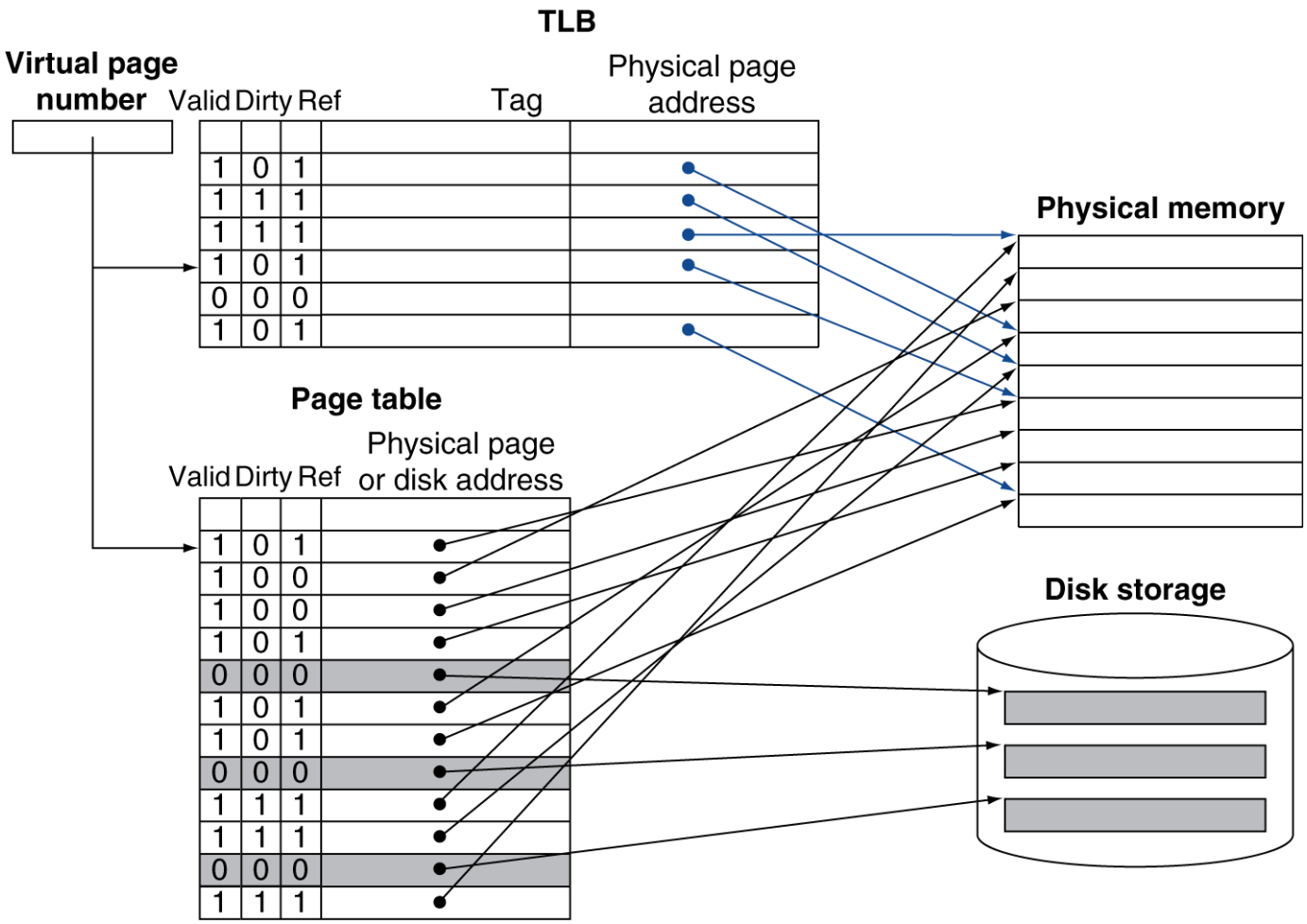
Address translation optimization

- Virtual Memory would appear to require extra memory references
  - one to translate Virtual Address into Physical Address (page table lookup) - Page Table is in physical memory
  - one to transfer the actual data (hopefully cache hit)
- But access to page tables has good locality
  - So use a fast cache of page tables within the CPU

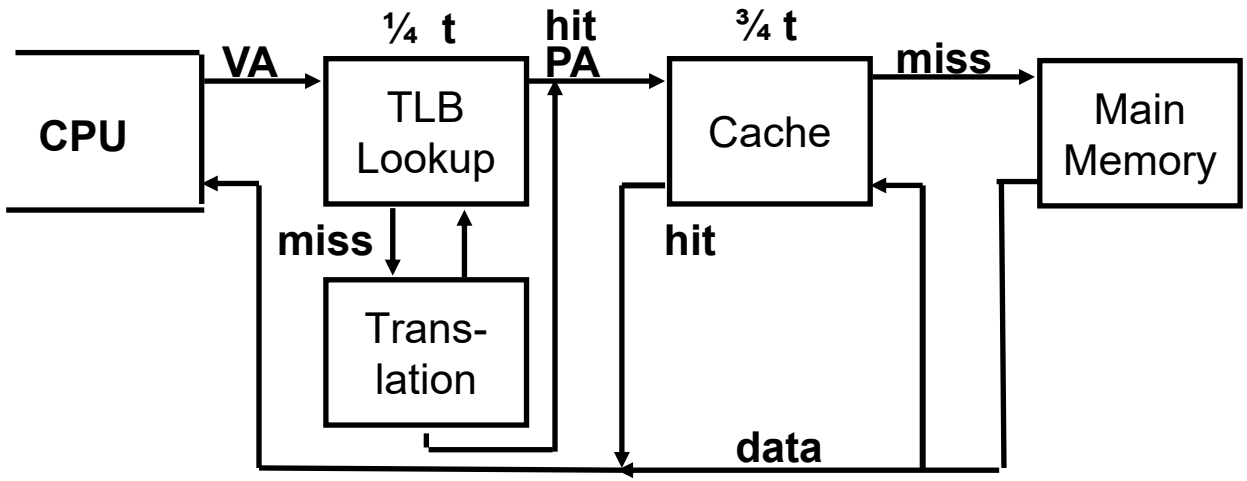
V	Virtual Page #	Physical Page #	Dirty	Ref	Access

  - Called a **T**ranslation **L**ook-aside **B**uffer (**TLB**)
  - Typical: 16–512 entries, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Making Address Translation Fast (2)



A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory → TLB miss can be handled by loading the translation information from the page table into the TLB (takes 10's of cycles to find and load the translation info into the TLB)
  - If the page is not in main memory, then it's a true page fault (takes millions of cycles to service a page fault)
- TLB misses are much more frequent than true page faults

Summary: steps in memory access