

Advanced Computer Architecture

Memory Hierarchy

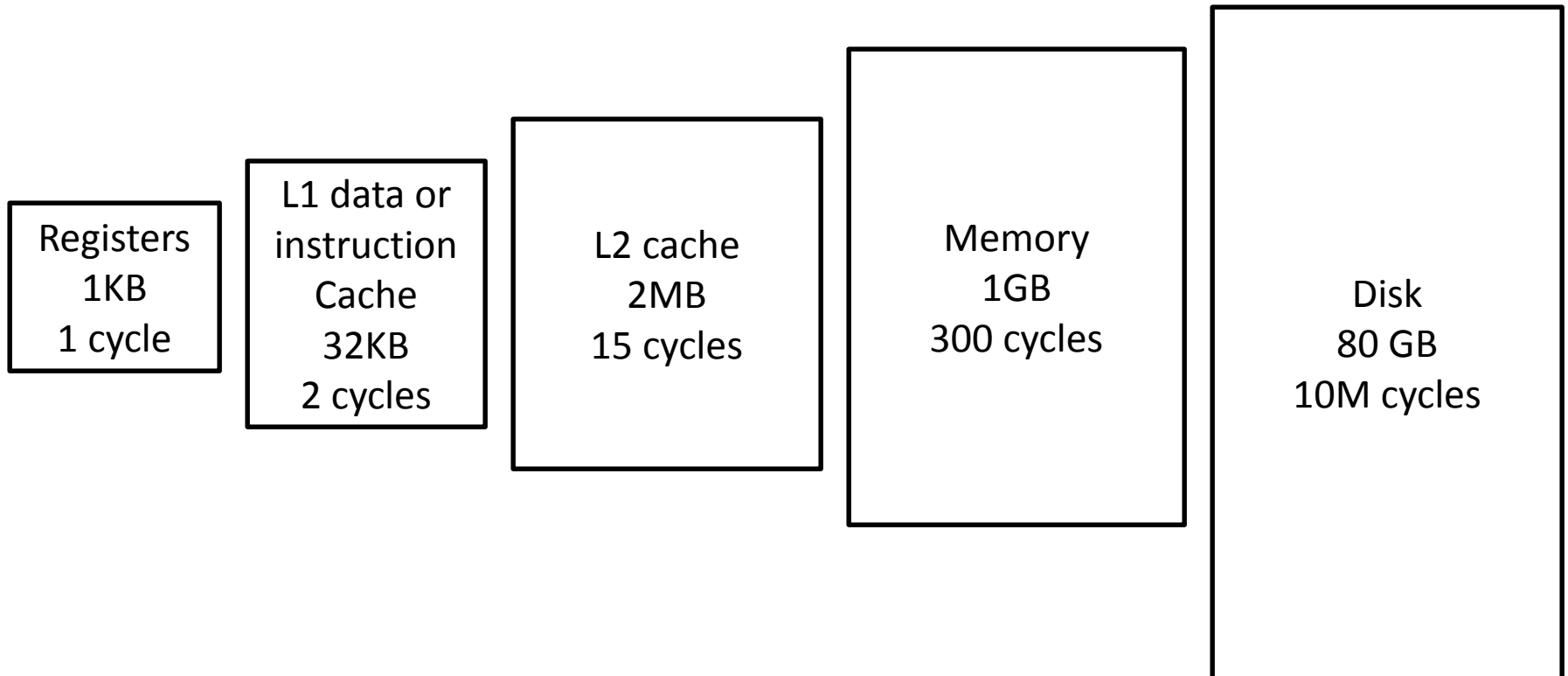


DRAM vs. SRAM

- *DRAM* (Dynamic Random Access Memory):
 - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
 - very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
 - used for *main memory*
- *SRAM* (Static Random Access Memory):
 - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
 - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
 - used for *cache*

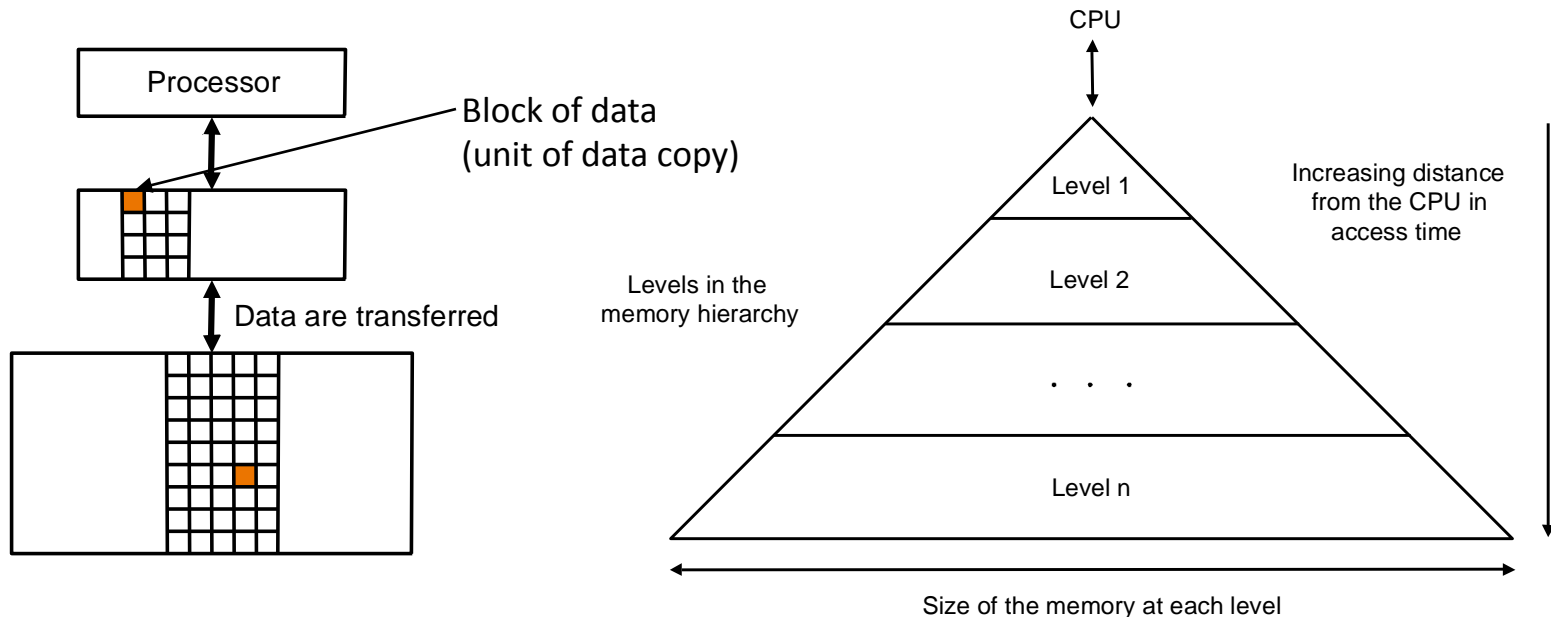
Memory Hierarchy

- As you go further, capacity and latency increase



Memory Hierarchy

- Users want large and fast memories...
 - expensive and they don't like to pay...
- Make it seem like they have what they want...
 - *memory hierarchy*
 - hierarchy is *inclusive*, every level is *subset* of lower level
 - performance depends on *hit rates*



Locality

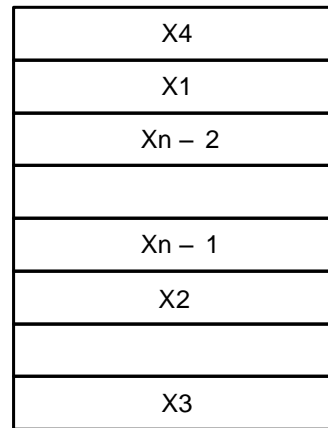
- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
 - *temporal locality*: it will tend to be *again* referenced soon
 - *spatial locality*: *nearby items* will tend to be referenced soon

Hit and Miss

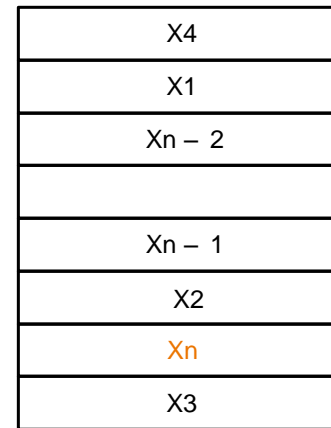
- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
 - *block*: minimum unit of data to move between levels
 - *hit*: data requested is in upper level
 - *miss*: data requested is not in upper level
 - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
 - *miss rate*: fraction of memory accesses that are not hits
 - $\text{miss rate} = 1 - \text{hit rate}$
 - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
 - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Caches Addressing Schemes

- Issues:
 - *how do we know if a data item is in the cache?*
 - *if it is, how do we find it?*
 - *if not, what do we do?*
- Solution depends on *cache addressing scheme...*



a. Before the reference to Xn



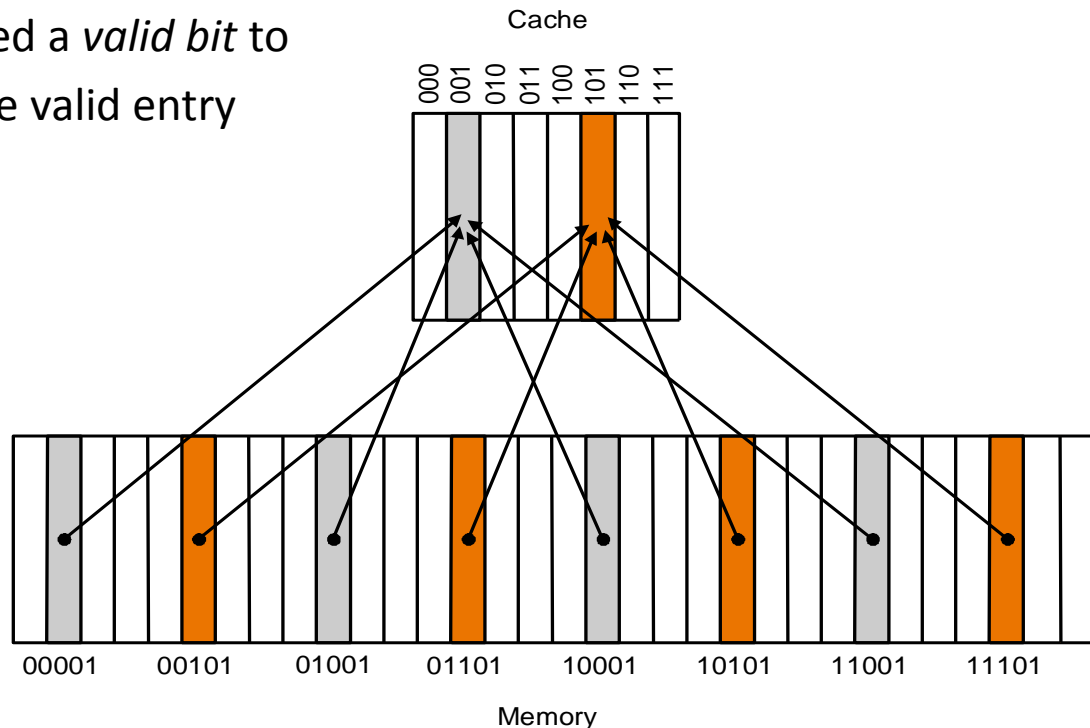
b. After the reference to Xn

Reference to X_n causes miss so it is fetched from memory

- By simple example
 - assume block size = one word of data

Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
 - cache block address = memory block address *mod* cache size (*unique*)
 - if cache size = 2^m , cache address = lower m bits of n -bit memory address
 - remaining upper $n-m$ bits kept as *tag bits* at each cache block
 - also need a *valid bit* to recognize valid entry



Accessing Cache

- Example:

(0) Initial state:

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

(1) Address referred 10110 (*miss*):

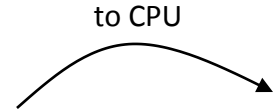
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(2) Address referred 11010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(3) Address referred 10110 (*hit*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

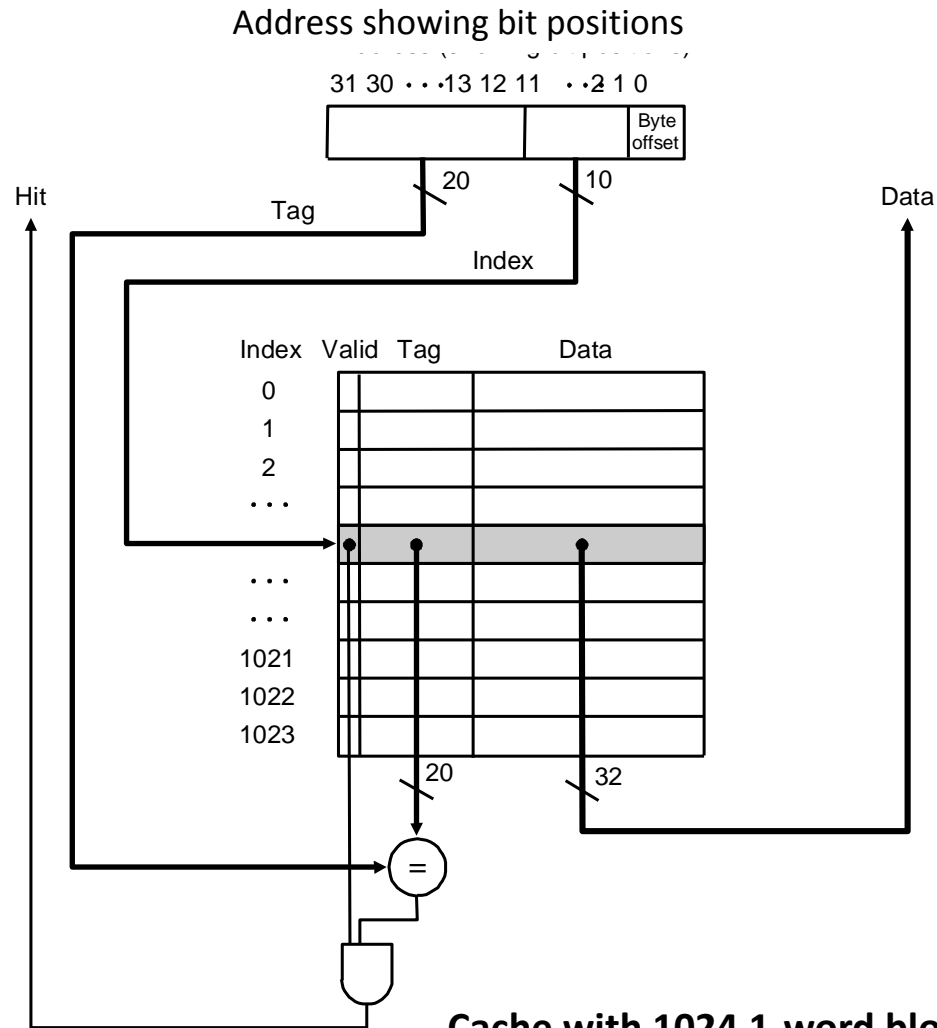


(4) Address referred 10010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

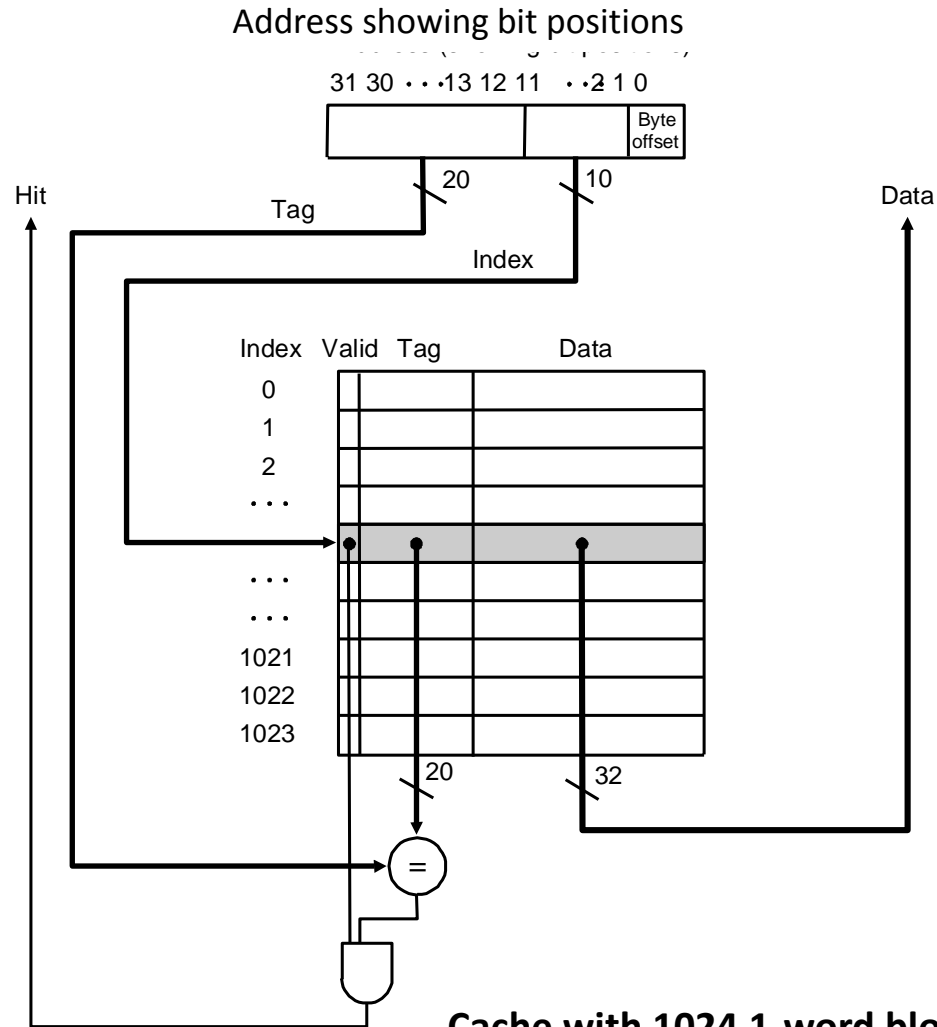
Direct Mapped Cache

- In MIPS



Direct Mapped Cache

- In MIPS



Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache

Cache Read Hit/Miss

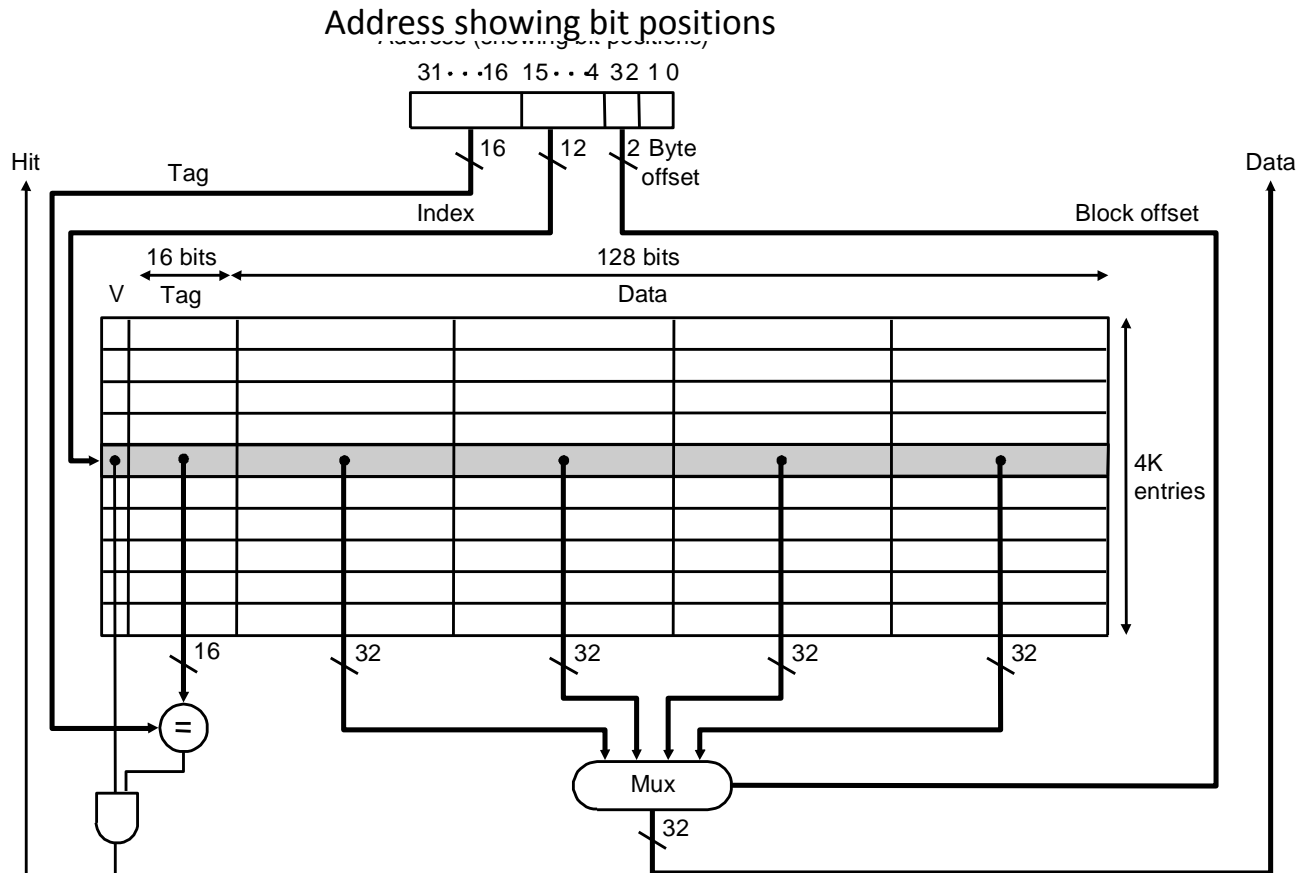
- *Cache read hit*: no action needed
- *Instruction cache read miss*:
 1. *Send original PC value (current PC – 4, as PC has already been incremented in first step of instruction cycle) to cache.*
 2. *Instruct main memory to perform read and wait for memory to complete access – stall on read*
 3. *After read completes write cache entry*
 4. *Restart instruction execution at first step to refetch instruction*
- *Data cache read miss*:
 1. *Similar to instruction cache miss*
 2. *To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete until the word is required – stall on use*

Cache Write Hit/Miss

- *Write-through* scheme
 - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
 - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
 - Write-through is slow because of always required memory write
- *Write-back* scheme
 - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
 - more efficient than write-through, more complex to implement

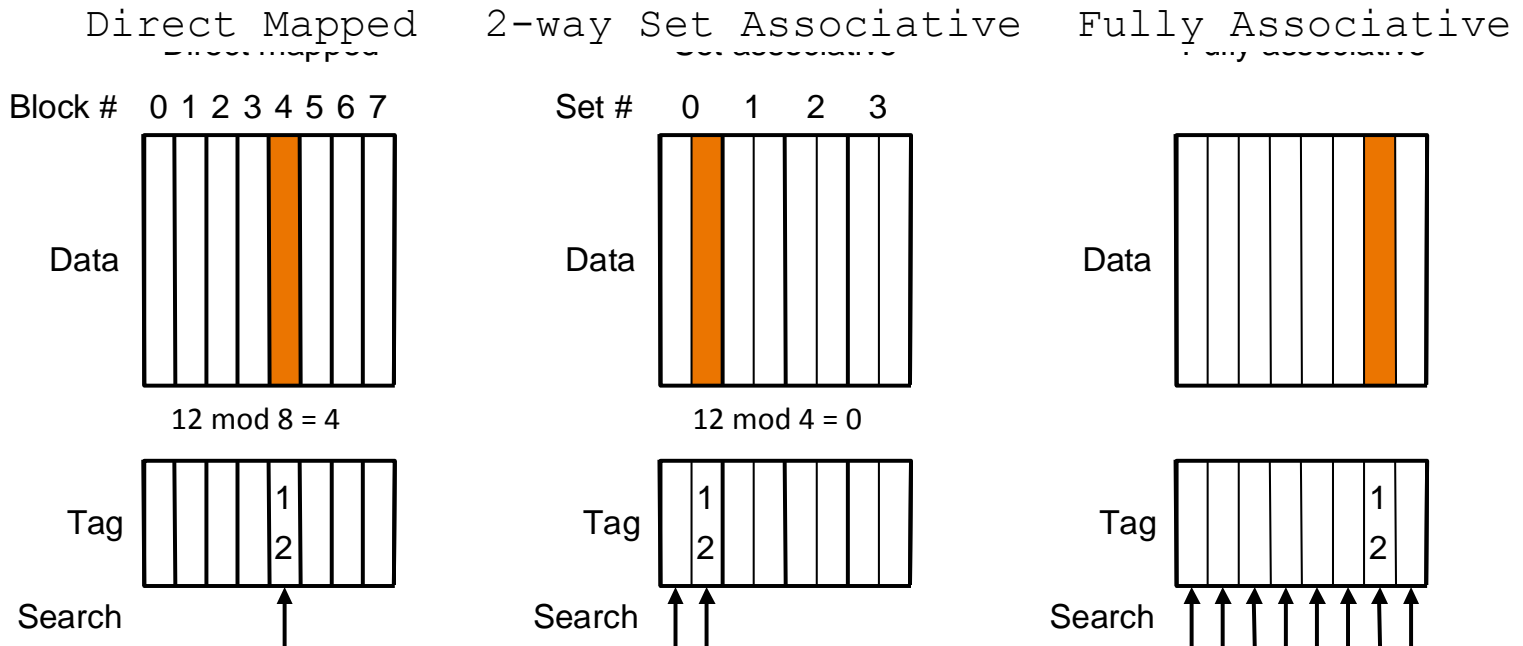
Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:



Cache with 4K 4-word blocks: *byte offset* (least 2 significant bits) is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache

Direct Mapped vs Set Associative vs Fully Associative Cache



Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity

Direct Mapped vs Set Associative vs Fully Associative Cache

- *Direct mapped*: one *unique* cache location for each memory block
 - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
 - *all* cache entries are searched to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size *n* it is *n*-way set-associative
 - cache set address = memory block address *mod* number of sets in cache
 - all cache entries in the corresponding set are searched to locate block
- Increasing degree of associativity
 - *reduces miss rate*
 - *increases hit time* because of the search and then fetch

Direct Mapped vs Set Associative vs Fully Associative Cache

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Configurations of an 8-block cache with different degrees of associativity

Example Problems

- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*

0, 8, 0, 6, 8

for each of the following cache configurations

- 1. direct mapped*
- 2. 2-way set associative (use LRU replacement policy)*
- 3. fully associative*

Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 ($= 0 \text{ mod } 4$)
6	2 ($= 6 \text{ mod } 4$)
8	0 ($= 8 \text{ mod } 4$)

Block address translation in direct-mapped cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – blue indicates new entry added

- 5 misses

Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 (= 0 mod 2)
6	0 (= 6 mod 2)
8	0 (= 8 mod 2)

Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – blue indicates new entry added

- Four misses

Solution (cont.)

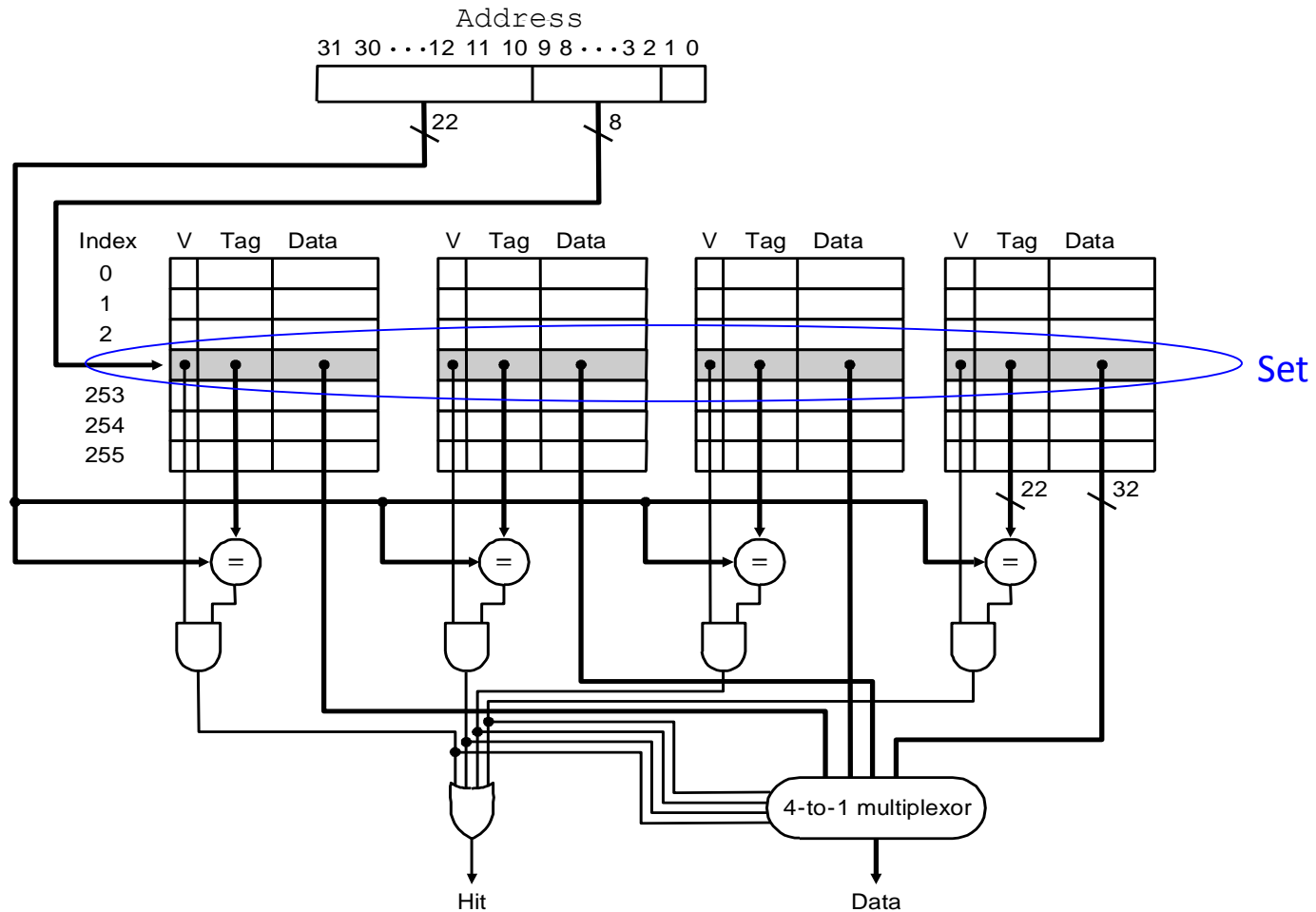
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Cache contents after each reference – blue indicates new entry added

- 3 misses

Implementation of a Set-Associative Cache



**4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor:
size of cache is 1K blocks = 256 sets * 4-block set size**

Multilevel Caches

- Add a *second-level* cache
 - primary cache is on the same chip as the processor
 - use SRAMs to add a second-level cache, sometimes off-chip, *between main memory and the first-level cache*
 - if miss occurs in primary cache second-level cache is accessed
 - if data is found in second-level cache miss penalty is access time of second-level cache which is much less than main memory access time
 - if miss occurs again at second-level then main memory access is required and large miss penalty is incurred

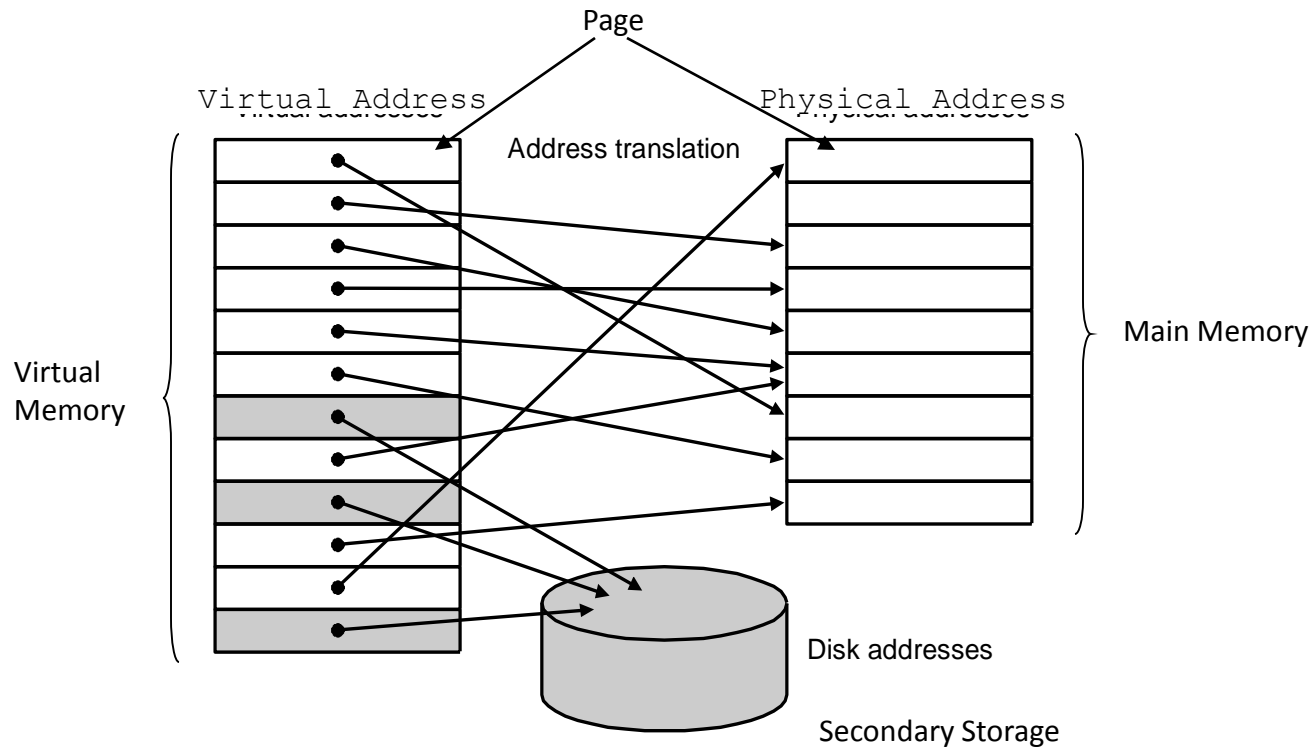
Virtual Memory

- *Virtual address space*, i.e., space addressable by a program is determined by ISA
 - typically: main memory size \leq disk size \leq virtual address space size
- Program can “pretend” it has main memory of the size of the disk – which is *smaller than* the *virtual memory* (= whole virtual address space), but *bigger than* the actual *physical memory* (=DRAM main memory)

Virtual Memory

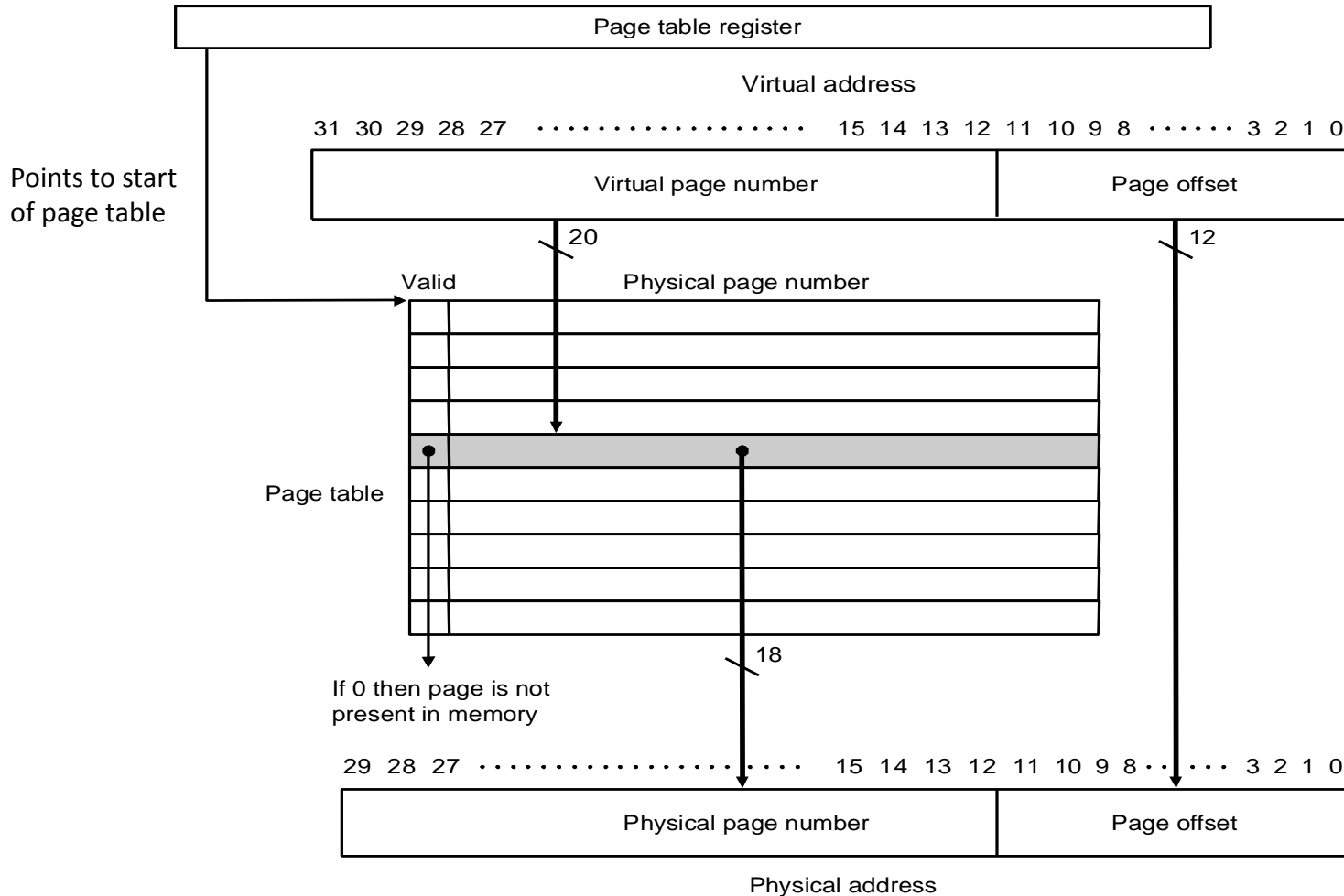
- *Page table* (in software) transparently converts a virtual memory address to a physical memory address, *if the data is already in main; if not*, it issues call to fetch the data from disk into main
- Virtual memory is organized in fixed-size (power of 2, typically at least 4 KB) blocks, called *pages*. Physical memory is also considered a collection of pages of the same size.
 - the unit of data transfer between disk and physical memory is a page

Virtual Memory



Mapping of pages from a virtual address to a physical address or disk address

Page Table Implements Virtual to Physical Address Translation



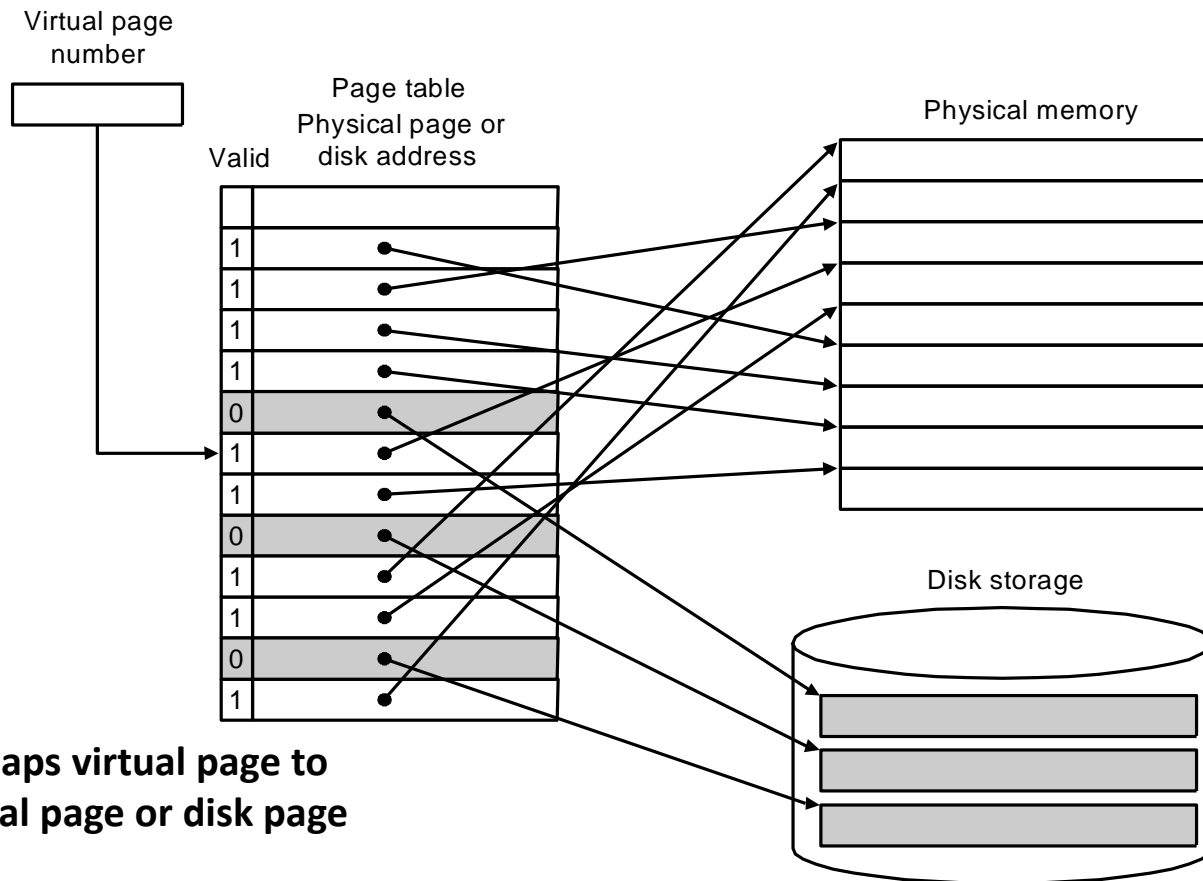
**Page table: page size 4 KB, virtual address space 4 GB,
physical memory 1 GB**

Page Faults

- *Page fault*: page is not in memory, must retrieve it from disk
 - *enormous miss penalty* = millions of cycles
 - therefore, page size should be *large* (e.g., 32 or 64 KB)
 - to make one trip to disk worth a lot
 - reducing page faults is *critical*

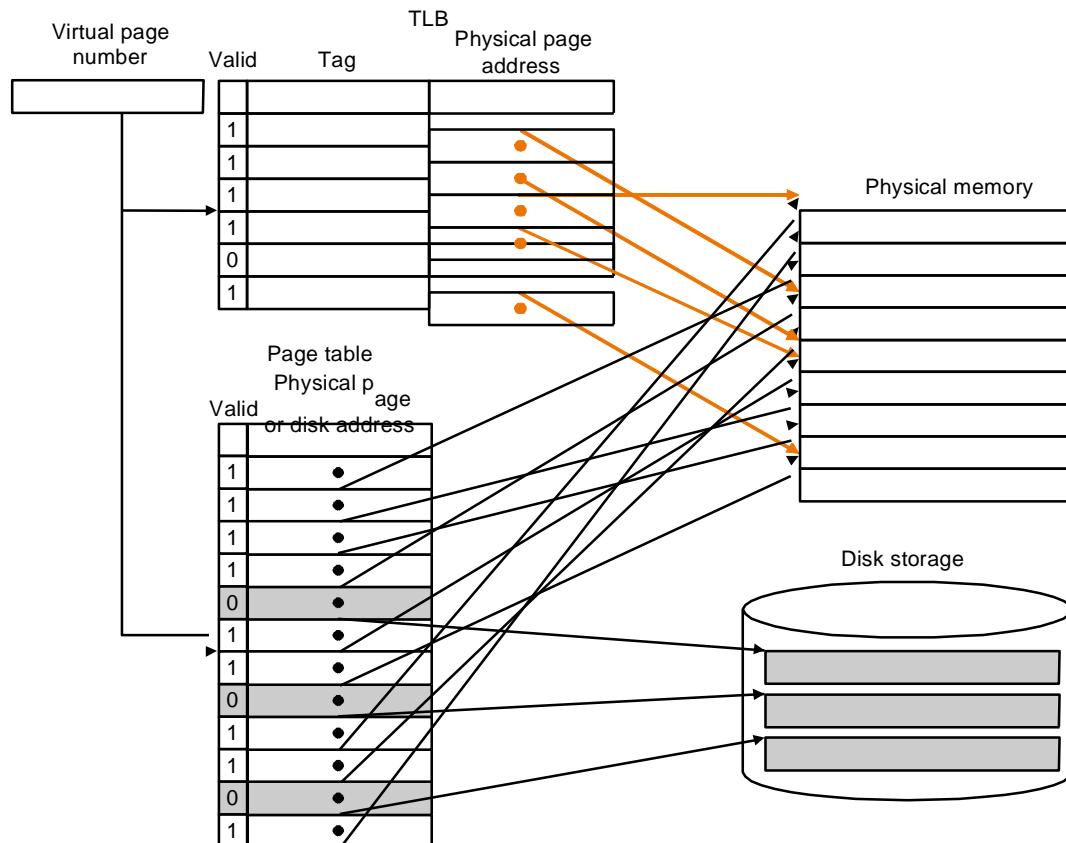
Resolving Page Faults using the Page Table to Access Disk

- There is a data structure, either part of or auxiliary to the page table, which records *where each virtual page is stored on disk* (cylinder, sector, block, etc.)



Making Address Translation Fast with the Translation-lookaside Buffer

- A cache for address translations – *translation-lookaside buffer (TLB)*:



On a page reference, first look up the virtual page number in the TLB; if there is a TLB miss look up the page table; if miss again then true page fault