CS60003: High Performance Computer Architecture

Memory Hierarchy Design – 1: Fundamentals

IIT KHARAGPUR

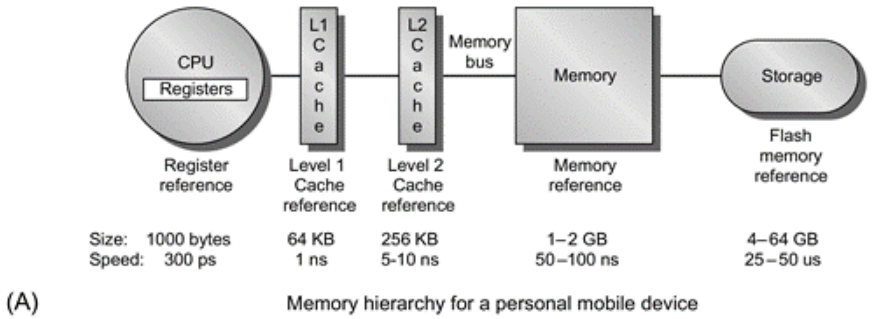
Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

Different Instruction Set Architectures

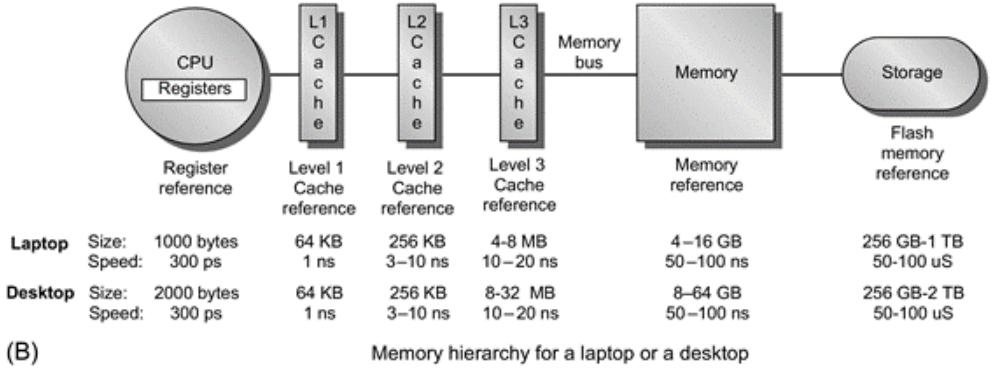
Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD

The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage and much smaller memories and caches. Increasingly, FLASH is replacing magnetic disks, at least for first level file storage. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2017; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk/FLASH storage includes both the media and the buffered interfaces.

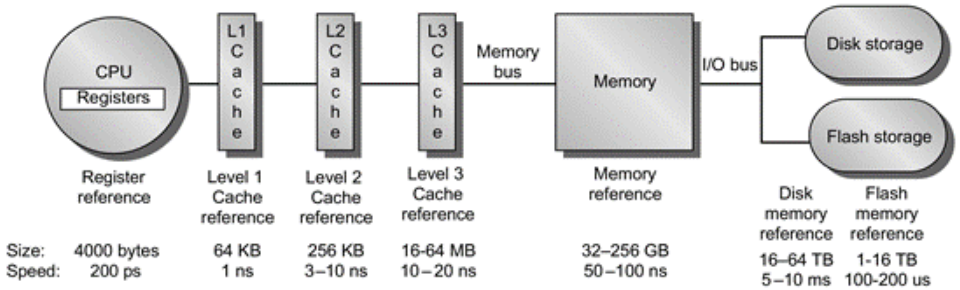
Typical Memory Hierarchy



(A) Memory hierarchy for a personal mobile device



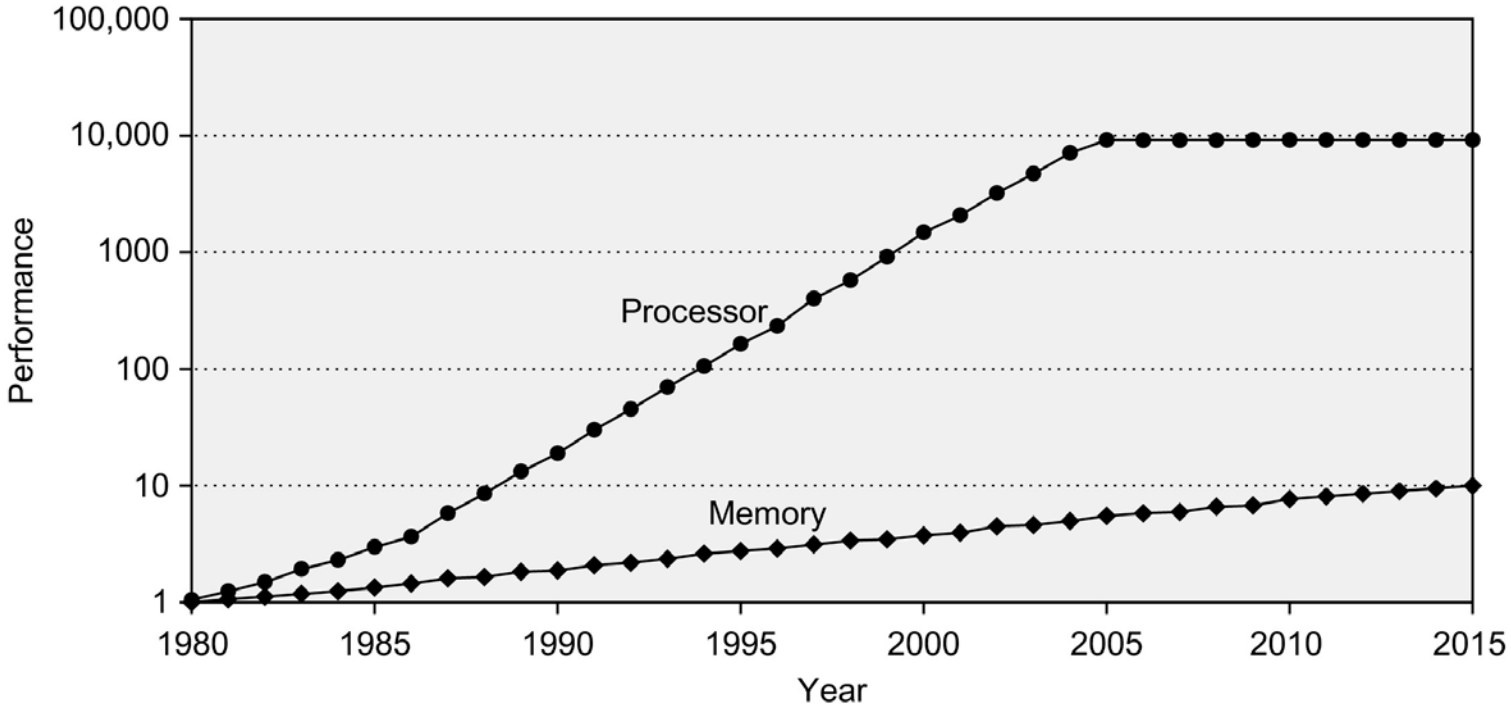
(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

The levels in a typical memory hierarchy in a personal mobile device (PMD), such as a cell phone or tablet (A), in a laptop or desktop computer (B), and in a server (C). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9 from picoseconds to milliseconds in the case of magnetic disks and that the size units change by a factor of 10^{10} from thousands of bytes to tens of terabytes. If we were to add warehouse-sized computers, as opposed to just servers, the capacity scale would increase by three to six orders of magnitude. Solid-state drives (SSDs) composed of Flash are used exclusively in PMDs, and heavily in both laptops and desktops. In many desktops, the primary storage system is SSD, and expansion disks are primarily hard disk drives (HDDs). Likewise, many servers mix SSDs and HDDs.

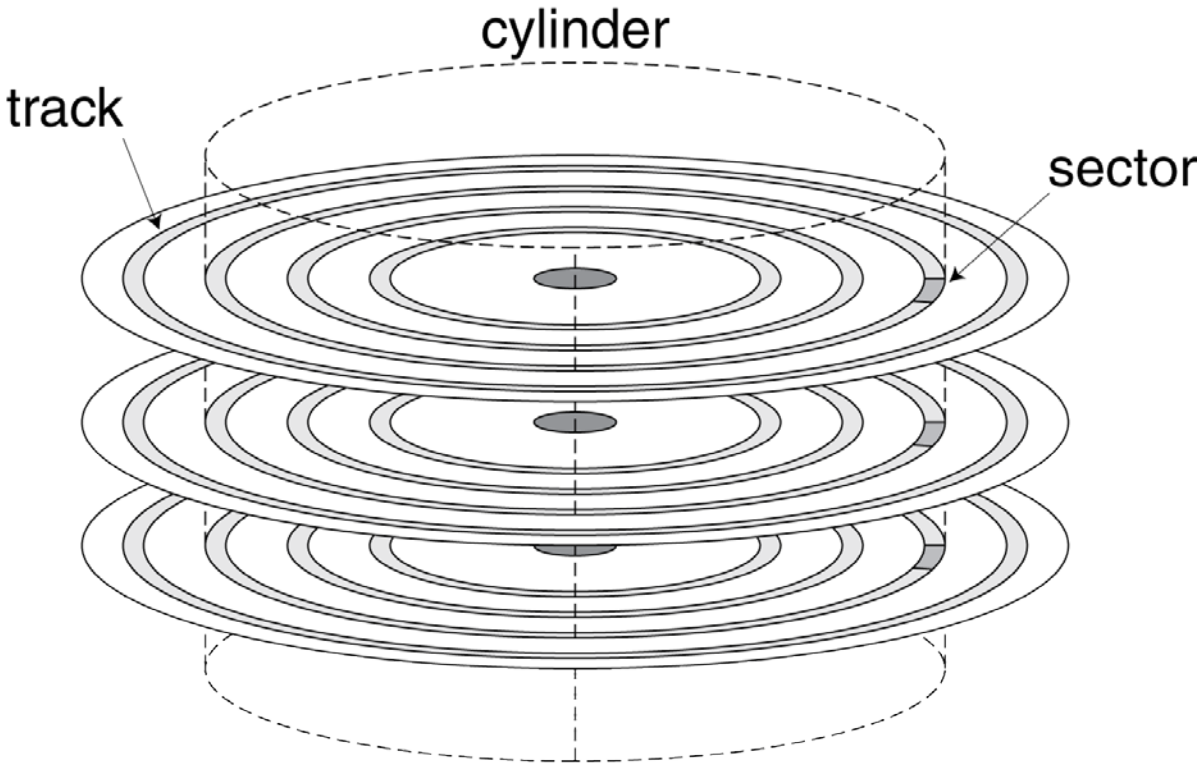
Processor-Memory Performance Gap



Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. In mid-2017, AMD, Intel and Nvidia all announced chip sets using versions of HBM technology. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.4 on page 88). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you can see, until 2010 memory access times in DRAM improved slowly but consistently; since 2010 the improvement in access time has reduced, as compared with the earlier periods, although there have been continued improvements in bandwidth.

Disk Storage

- **Nonvolatile, rotating magnetic storage**
- **Important concepts: sector, track, cylinder**



Disk Access Example

- **Given:**
  - **512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk**

- **Average read time:**
  - **4ms seek time**
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms

- **If actual average seek time is 1 ms:**
  - **Average read time = 3.2 ms**

Disk Performance Issues

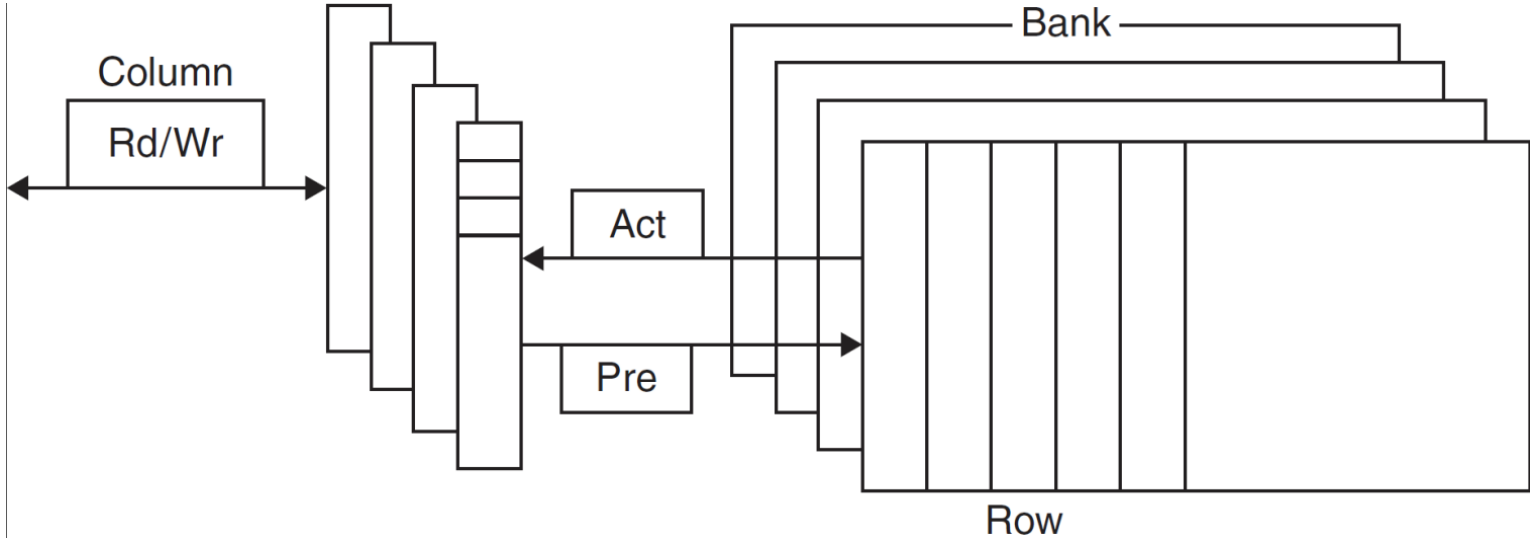
- **Manufacturers quote average seek time**
  - **Based on all possible seeks**
  - **Locality and OS scheduling lead to smaller actual average seek times**
- **Smart disk controller allocate physical sectors on disk**
  - **Present logical sector interface to host**
  - **SCSI, ATA, SATA**
- **Disk drives include caches**
  - **Pre-fetch sectors in anticipation of access**
  - **Avoid seek and rotational delay**

Memory Technology

- **Static RAM (SRAM)**
  - **0.5ns – 2.5ns, $2000 – $5000 per GB**
- **Dynamic RAM (DRAM)**
  - **50ns – 70ns, $20 – $75 per GB**
- **Ideal memory should have:**
  - **Access time of SRAM**
  - **Capacity and cost/GB of disk**

Main Memory: DRAM

- **Data stored as a charge in a capacitor**
  - **Single transistor used to access the charge**
  - **Must periodically be refreshed (read content and write again)**



H&P CA:A QA (6th. Ed.)

DRAM Size

			Best case access time (no precharge)			Precharge needed
Production year	Chip size	DRAM type	RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

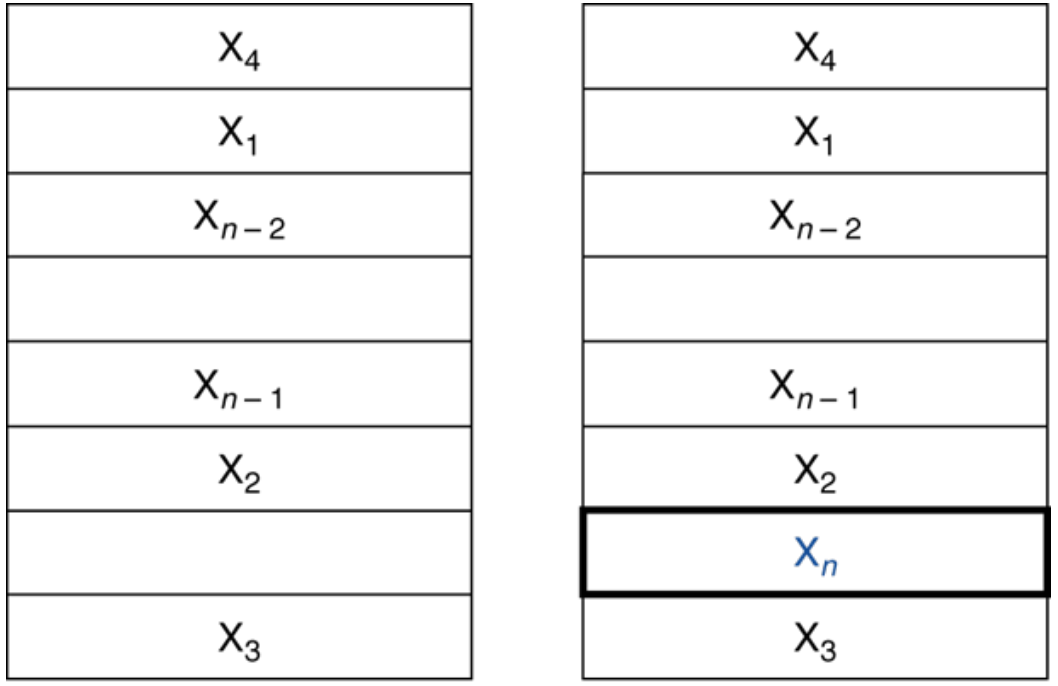
Capacity and access times for DDR SDRAMs by year of production. Access time is for a random memory word and assumes a new row must be opened. If the row is in a different bank, we assume the bank is precharged; if the row is not open, then a precharge is required, and the access time is longer. As the number of banks has increased, the ability to hide the precharge time has also increased. DDR4 SDRAMs were initially expected in 2014, but did not begin production until early 2016.

DRAM Performance Improvement

- **Row buffer**
  - **Allows several words to be read and refreshed in parallel**
- **Synchronous DRAM**
  - **Allows for consecutive accesses in bursts without needing to send each address**
  - **Improves bandwidth**
- **DRAM banking:**
  - **Allows simultaneous access to multiple DRAMs**
  - **Improves bandwidth**

Cache Memory

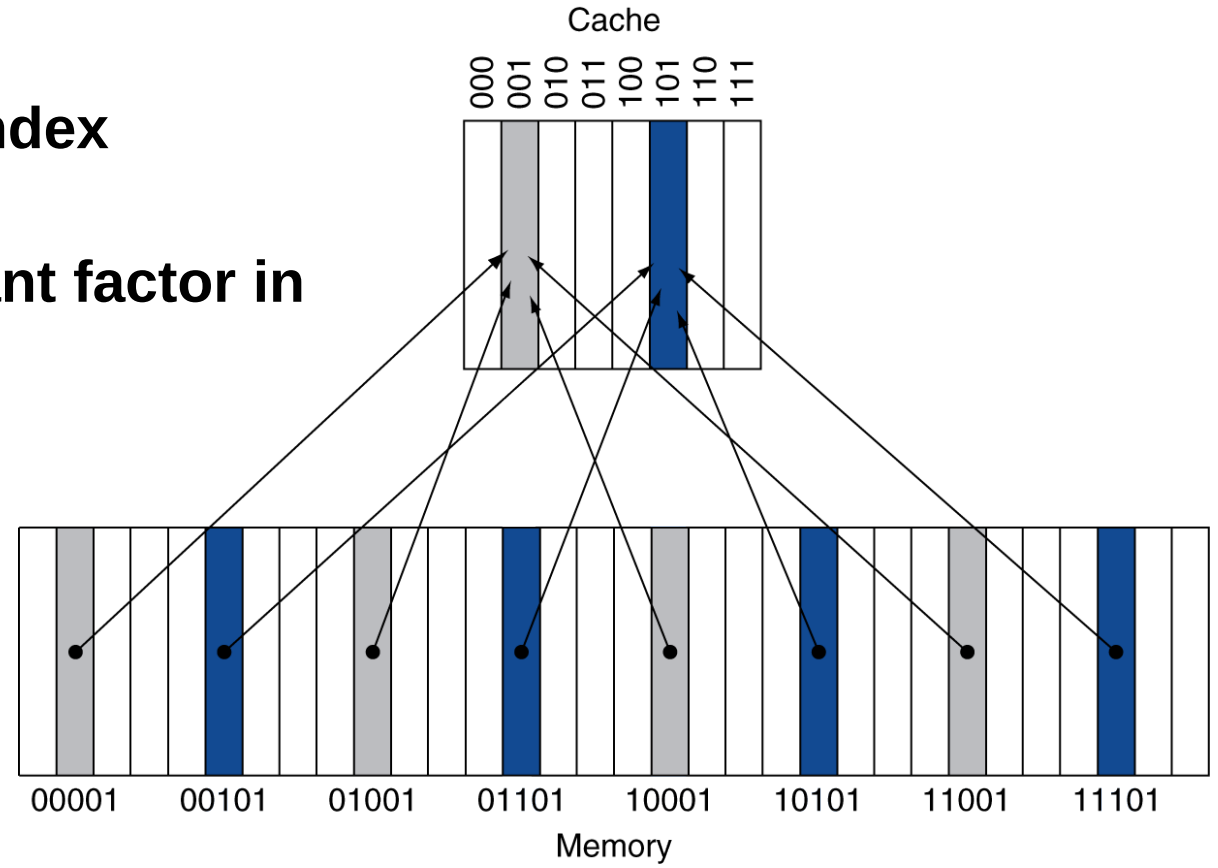
- **Cache memory: the level of memory closest to the CPU**
  - **SRAM-type circuits, faster than DRAM, but larger and more power-hungry!**
- **Given accesses X_1, X_2, …. , X_{n-1}, X_n , decide:**
  - **Is data present at all?**
  - **Where do we look for it?**



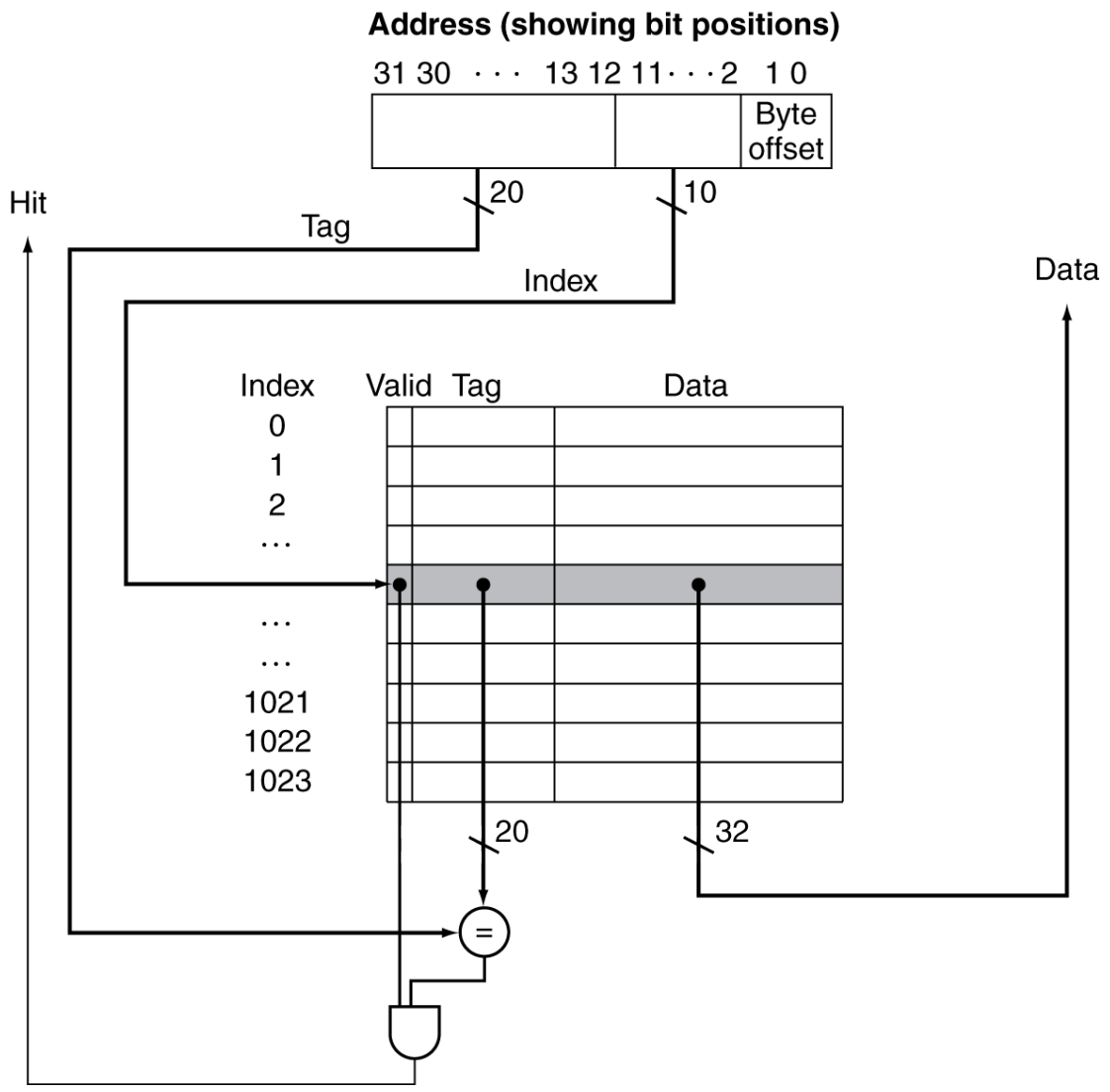
a. Before the reference to X_n b. After the reference to X_n

Direct Mapped Cache

- **Location determined by address**
- **Cache Block (or Cache Line): unit of data storage in cache**
- **Direct mapped: only one choice**
  - **Cache Index = (Block address) % (No. of blocks in cache)**
  - **No. of blocks is a power of 2**
  - **Use low-order bits of block address as index**
    - **e.g. 8 blocks => lower 3-bits are index bits**
  - **Block size (usually in kB) is very important factor in calculations!**



Direct Mapped Cache Organization: Tags and Valid Bits



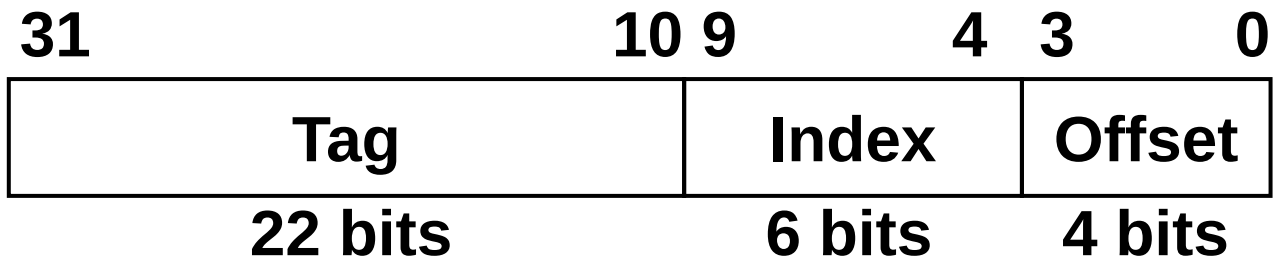
- **Assume: Block size = 1 word**
  - **Assume: 1 word = 4 bytes**
  - **Block offset = word offset = 2 bits in address**
- **Tag bits: uniquely identify the block**
- **Valid bit: is the data valid?**
  - **Valid bit: 1 = present, 0 = not present**
  - **Initial value of valid bit: 0**

Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?

- Block address = $\lfloor 1200/16 \rfloor = 75$

- Block number = 75 (mod 64) = 11



Cache Operation Example

- **Total 8 blocks in cache, 1 word/block, direct mapped**
- **Initial state:**

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

Cache Operation Example (contd.)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Cache Operation Example (contd.)

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

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		

Cache Operation Example (contd.)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

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		

Cache Operation Example (contd.)

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

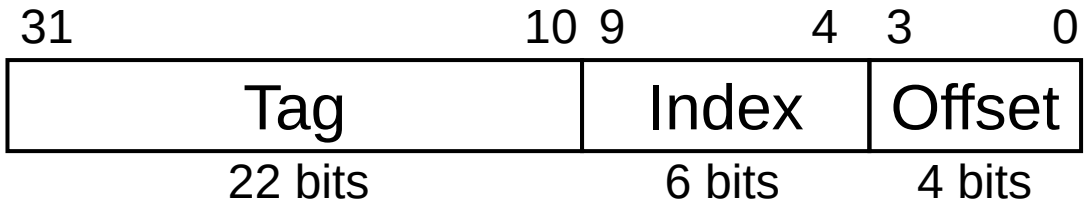
Cache Operation Example (contd.)

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example: Larger Block Size

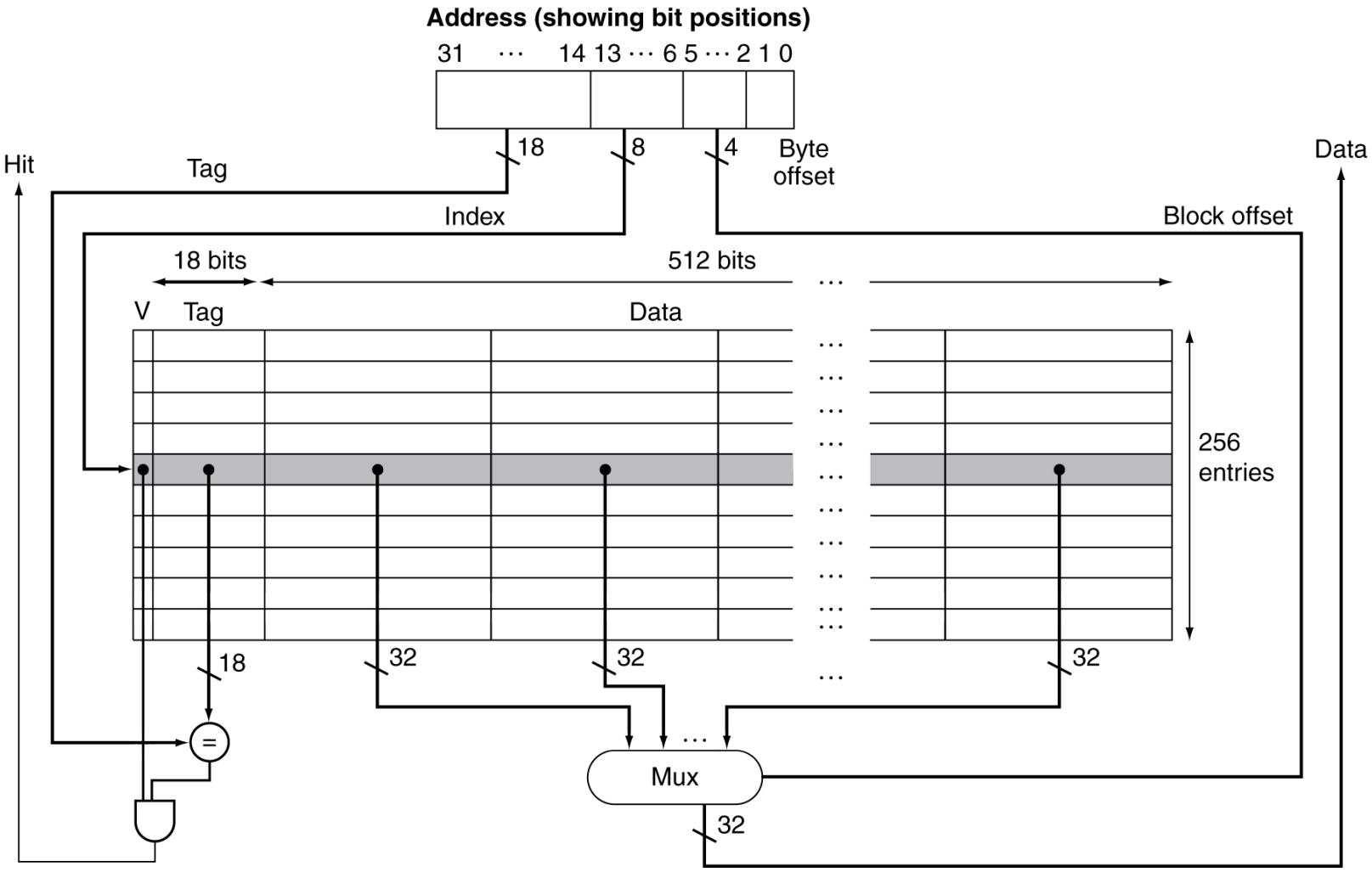
- **64 blocks, 16 bytes/block**
  - **To what block number does address 1200 map?**
- **Block address = $\lfloor 1200/16 \rfloor = 75$**
- **Block number = 75 modulo 64 = 11**



Block Size Considerations

- **Larger blocks should reduce miss rate**
  - **Due to spatial locality**
- **But in a fixed-sized cache**
  - **Larger blocks ⇒ fewer of them**
    - **More competition ⇒ increased miss rate**
  - **Larger blocks ⇒ pollution**
- **Larger miss penalty**
  - **Can override benefit of reduced miss rate**
  - **Early restart and critical-word-first can help**

Example: Cache in Intrinsity FastMATH



- Direct mapped
- 32-bit addresses
- 8-bit index => 256 blocks
- Word size: 4 bytes (32 bits)
- Block size: 16 words => 64 bytes (512 bits)
- Cache size: (for the data blocks only)

  256 blocks * (64 bytes/block)

  = 16 kB
- Block offset (4 bits) selects one of the 16 possible words, and returns to processor

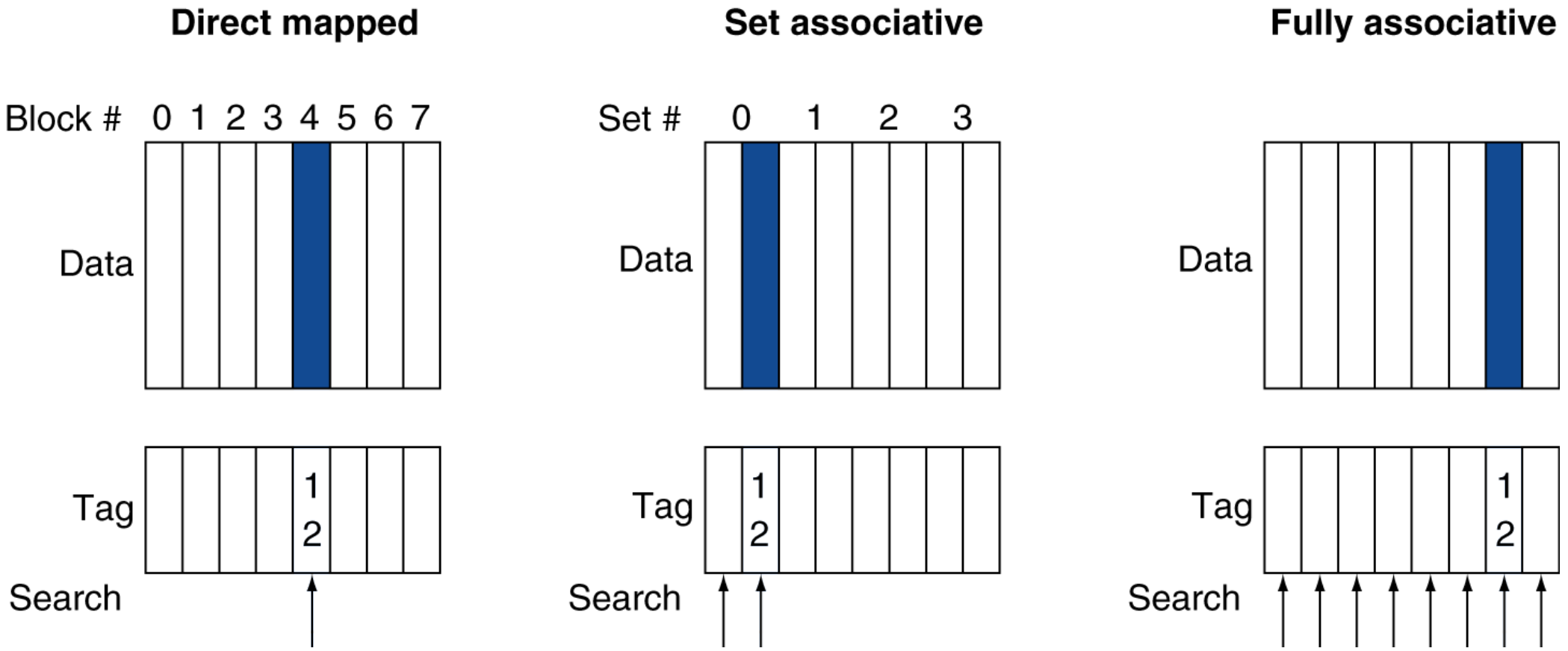
Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy (~100 clock cycles penalty)
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

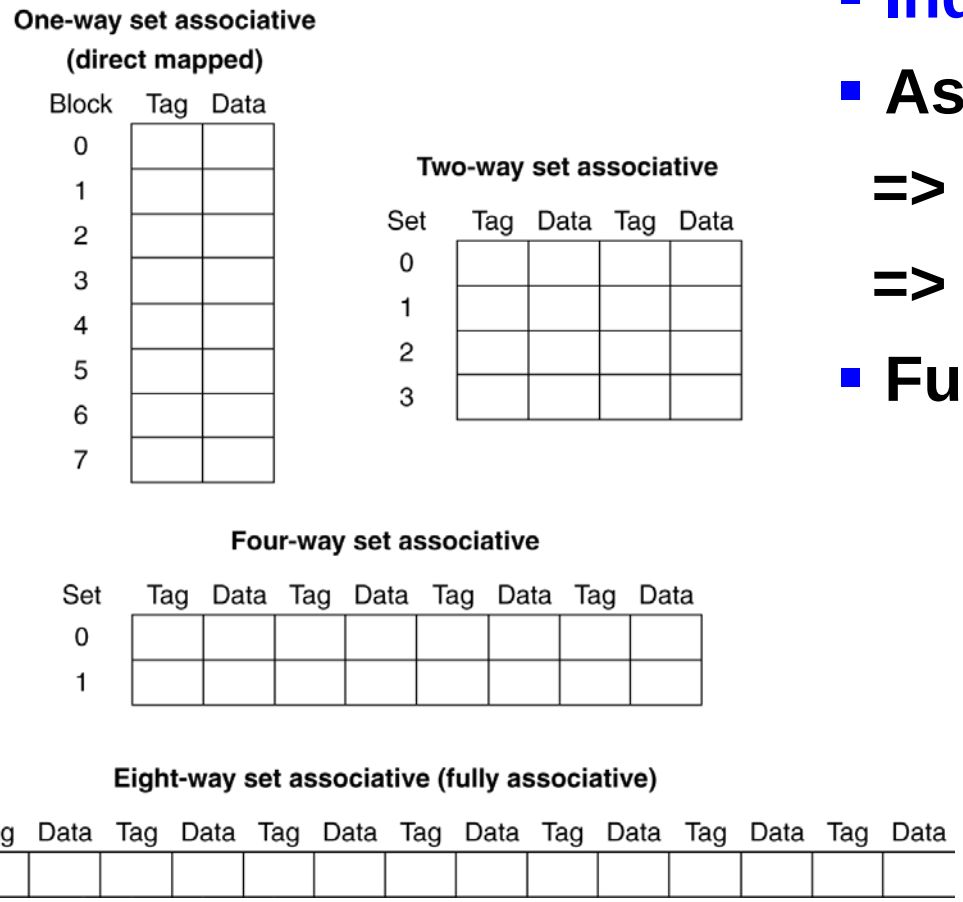
Associative Caches

- **Basic idea: give more options to map a memory block to the cache**
- **Fully Associative**
  - **Allow a given block to go in any cache entry**
  - **Requires all entries to be searched at once**
  - **Tag comparator per entry => expensive**
- ***n*-way Set Associative**
  - **Each set contains *n* entries**
  - **Block number determines set:**
    - **Set no. = Block no. % (#no. of sets)**
  - **Search all entries in a given set at once**
  - ***n* tag comparators => less expensive than fully associative cache**
  - **Note: Direct mapped cache is 1-way set associative!**

Associative Cache Examples



Spectrum of Associativity (cache with total 8 blocks)

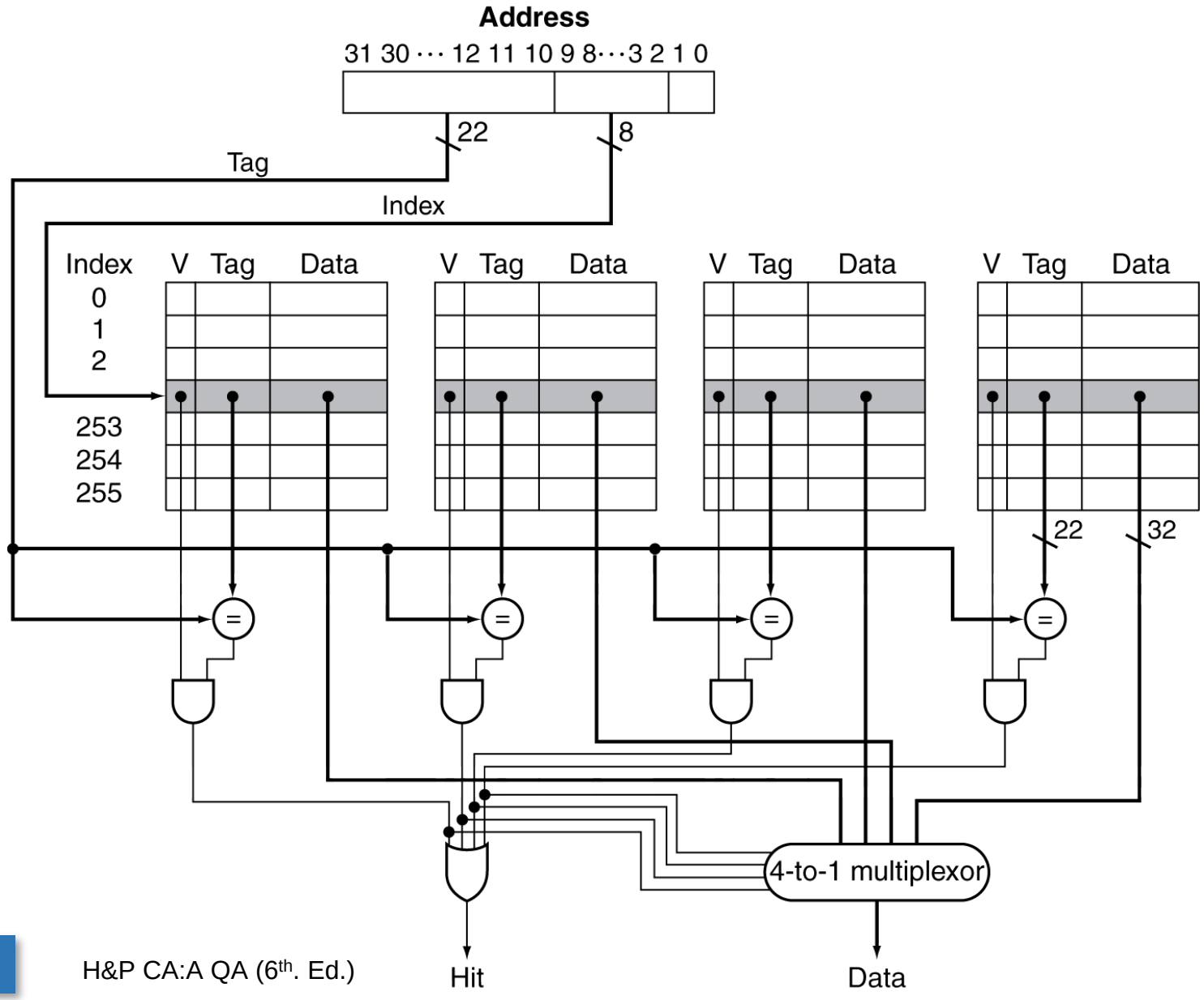


- **Index is basically set no.!**
- **As associativity increases**

  => # of sets decreases

  => # of index bits decreases
- **Fully associative cache: no index!**

Example: 4-way Set Associative Cache Organization



How Much Associativity?

- **Observation: Associativity increase => decrease of miss rate**
  - **But with diminishing returns!**
- **Simulation of a system with 64 kB D-cache, block size = 16 words, SPEC2000 benchmark suite, gives miss rates:**
  - **1-way (direct mapped): 10.3%**
  - **2-way: 8.6%**
  - **4-way: 8.3%**
  - **8-way: 8.1%**
- **Associativity > 8 is rare**
- **Fully associative caches are usually small, and usage is limited!**

Which Block to Replace? (Replacement Policies)

- Direct-mapped => no choice!
- Set associative:
  - Prefer non-valid entry (valid bit =0), if there is one
  - Otherwise, have to choose among entries in the set
- Two main policies: Random, Least-recently Used (LRU), First-in first-out (FIFO)
- Random replacement: randomly select the block to replace
  - Simple to implement
  - Uses hardware pseudo-random number generator
- LRU replacement: replace the one unused for the longest time
  - Difficult to implement, especially for higher associativity
  - Usually approximations are used
- FIFO: simple to implement (an approximation of LRU)
- Random replacement gives approx. similar performance as LRU for high associativity!

Associativity Example

- **Total no. of blocks: 4**
- **Block address access sequence: 0, 8, 0, 6, 8**

- **Direct mapped:**

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Associativity Example (contd.)

- **2-way set associative => # of sets = (# of blocks) / 2 = 4 / 2 = 2**
- **LRU replacement policy**
- **Note: set index = (block address) % (# of sets)**

Block address	Cache index (Set Index)	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

Associativity Example (contd.)

- **Fully associative: cache still has space for 4 blocks**
- **Note: no index bits!**

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Types of Misses

- **Compulsory Miss (a.k.a. cold-start miss / first-reference miss):**
  - **The very first access to a block *cannot* be in the cache**

- **Capacity Misses:**
  - **In addition to compulsory misses**
  - **The cache cannot contain all the blocks needed during execution**
  - **Repeatedly, blocks will be discarded and later retrieved after miss**

- **Conflict Misses (a.k.a. collision misses):**
  - **In addition to compulsory and capacity misses**
  - **Happens mostly in direct-mapped and set-associative caches**
  - **Too many requests (> *n*) for a *n*-way Set Associative Cache**

Write-Hit Policies

- Simple option: on data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write-through: also update memory
- But makes each memory write take (much) longer!
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = $1 + 0.1 \times 100 = 11$
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Hit Policies (contd.)

- **Alternative: Write-Back: On data-write hit, just update the block in cache**
  - **Keep track of whether each block is dirty**

- **When a dirty block is replaced**
  - **Write it back to memory**
  - **Can use a write buffer to allow replacing block to be read first**
    - **Step-1: Store block to be updated in memory in write buffer**
    - **Step-2: Update block in cache by reading from memory**
    - **Step-3: Then, send block waiting in write buffer to memory**

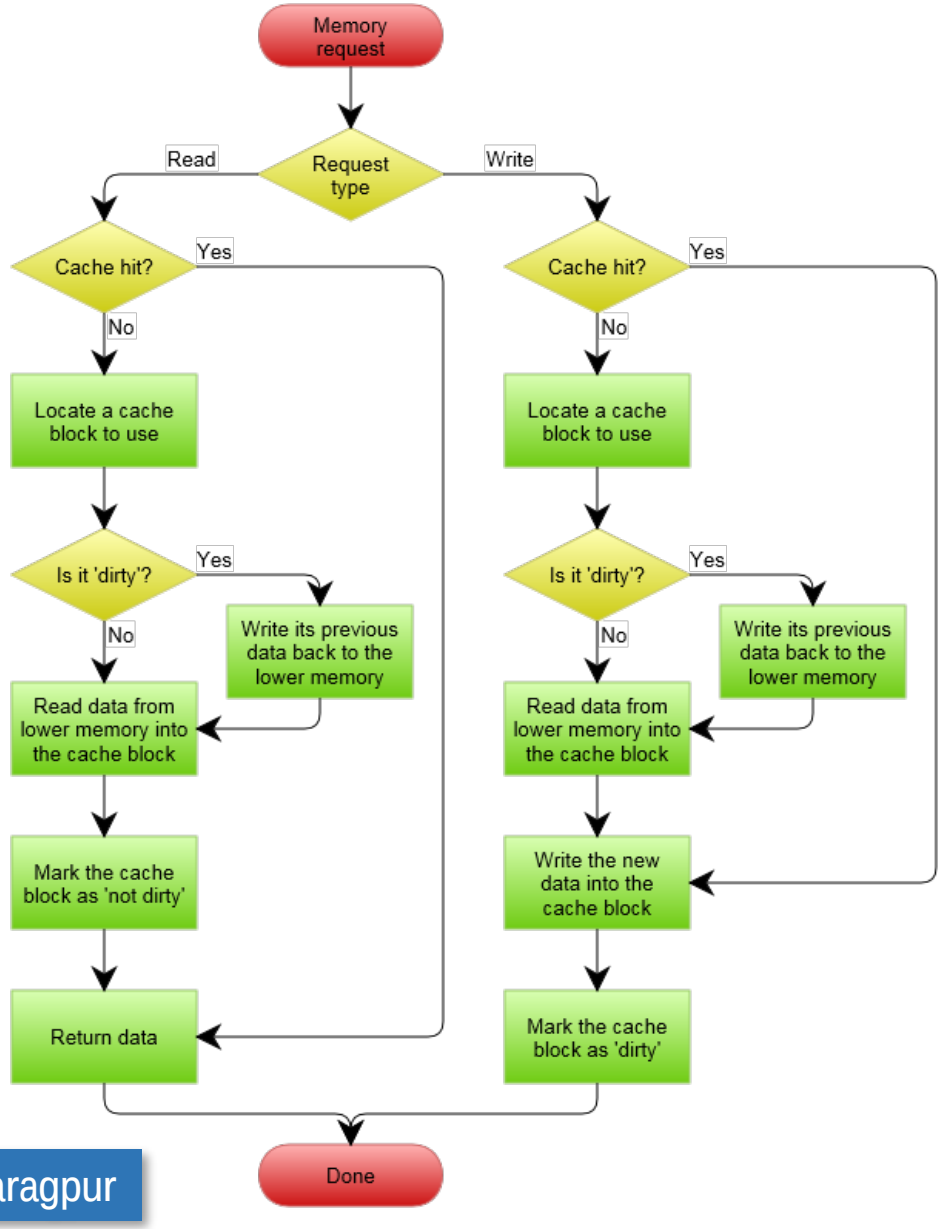
- **More difficult to implement!**

What Happens on Write-Miss?

- **Two options (based on cache block update policies):**
  - **Write-Allocate:**
    - **A block is allocated on a write miss**
    - **If block to be replaced is dirty, first it is written to lower levels**
      - **Similar to how read misses are handled**
    - **Subsequently, a write hit happens because of the updated block**
  - **No-Write-Allocate:**
    - **Write misses do not affect the contents of cache!**
    - **Instead, the block is modified only in the lower-level memory!**

- **Usually followed combinations of (Write Policy, Write Miss):**
  - **Write-Back-with-Write-Allocate**
  - **Write-Through-with-No-Write-Allocate**

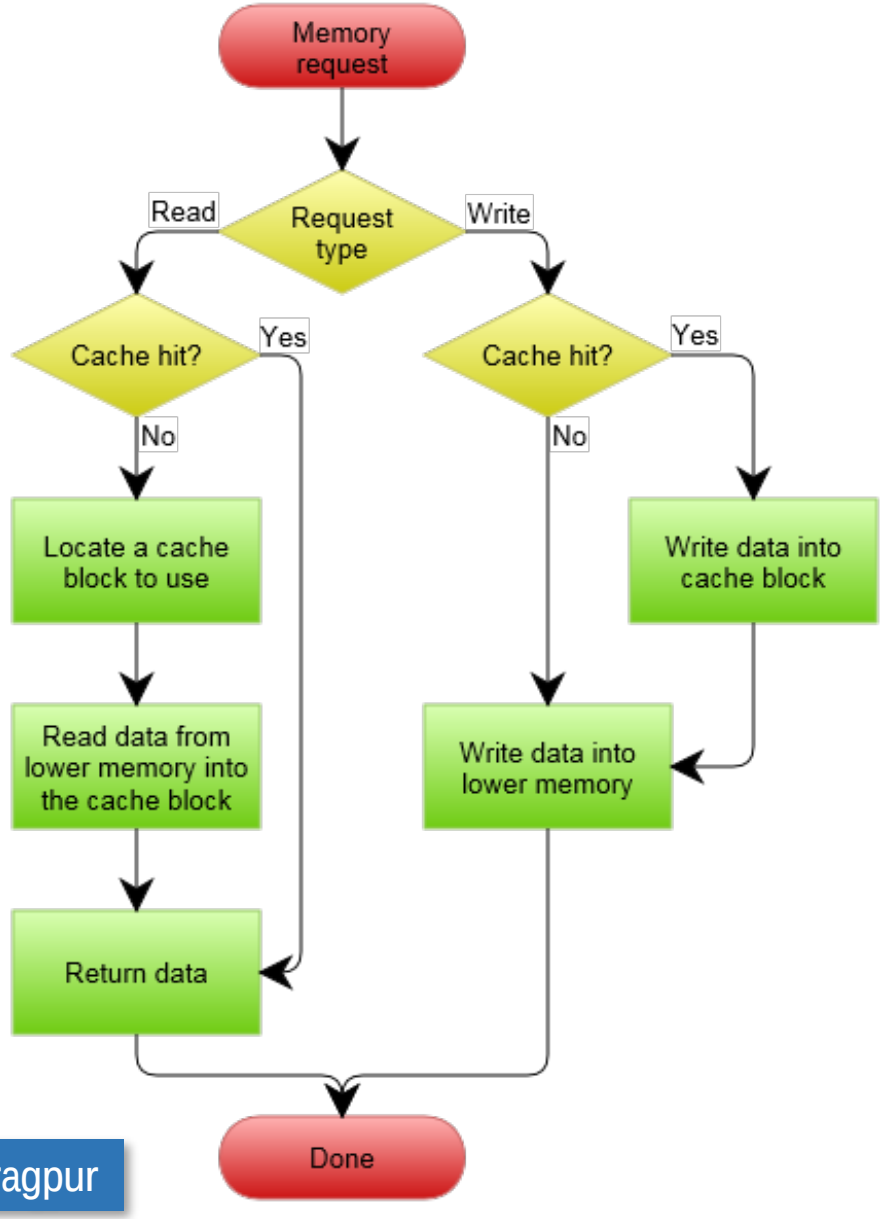
Write-Back-with-Write-Allocate



Observation: blocks that are written repeatedly but never read, remain in the cache, keep on getting updated repeatedly because of processor write operations on it, until replaced because of conflict!

Source: Wikipedia

Write-Through-with-No-Write-Allocate



- **Reasoning behind this peculiar strategy:** even if the cache block is updated, anyways it has to be still immediately updated in lower memory because of Write-Through), so what's the point?
- **Observation:** cache blocks which are being written repeatedly, will result in repeated write misses, until there is a read miss and the cache block is updated

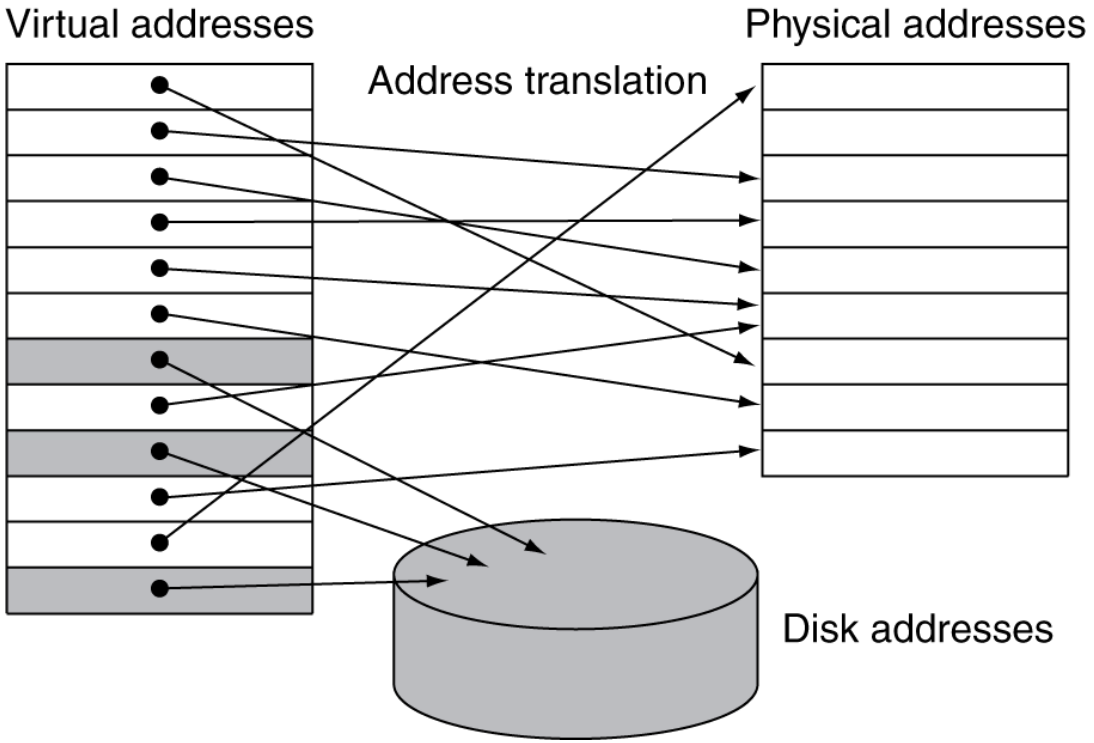
Source: Wikipedia

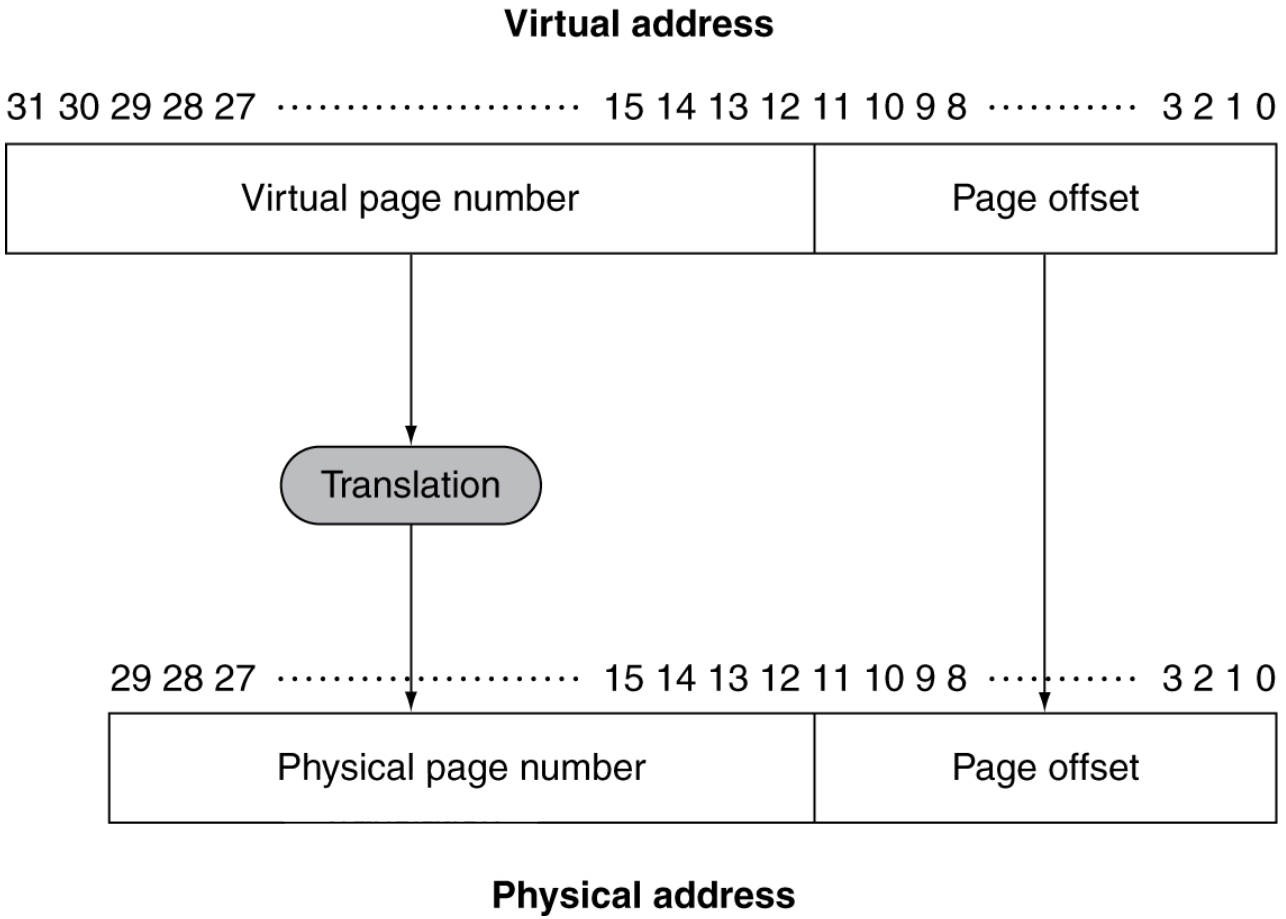
Virtual Memory

- **Use main memory as a “cache” for secondary (disk) storage**
  - **Managed jointly by CPU hardware and OS**
- **Programs share main memory**
  - **Each gets a private virtual address space holding its frequently used code and data**
  - **Protection from other programs**
- **CPU and OS translate V.A. to P.A.**
  - **V.M. quanta of memory is called “page”**
  - **P.M. page: “frame”**
  - **“Page table” (in main memory): contains information about V.A. to P.A. translation**
  - **“Page fault”: required frame is not in main memory => millions of clock cycles required to handle the fault (OS-assisted)**

Example: Address Translation

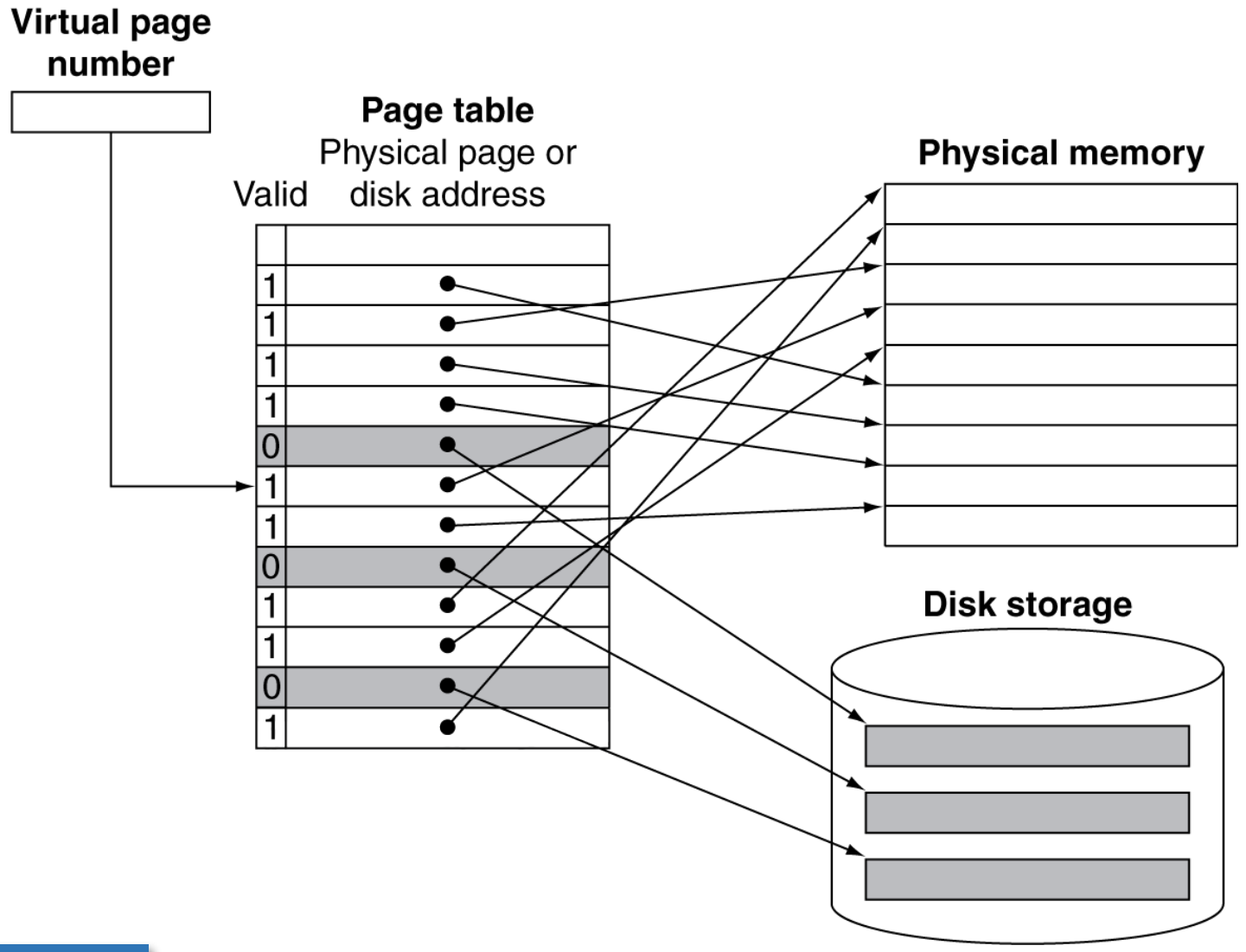
- **Assume fixed-sized (4 kB) pages, 32-bit V.A., 30-bit P.A.**





H&P CA:A QA (6th. Ed.)

Example: Mapping Pages to Storage

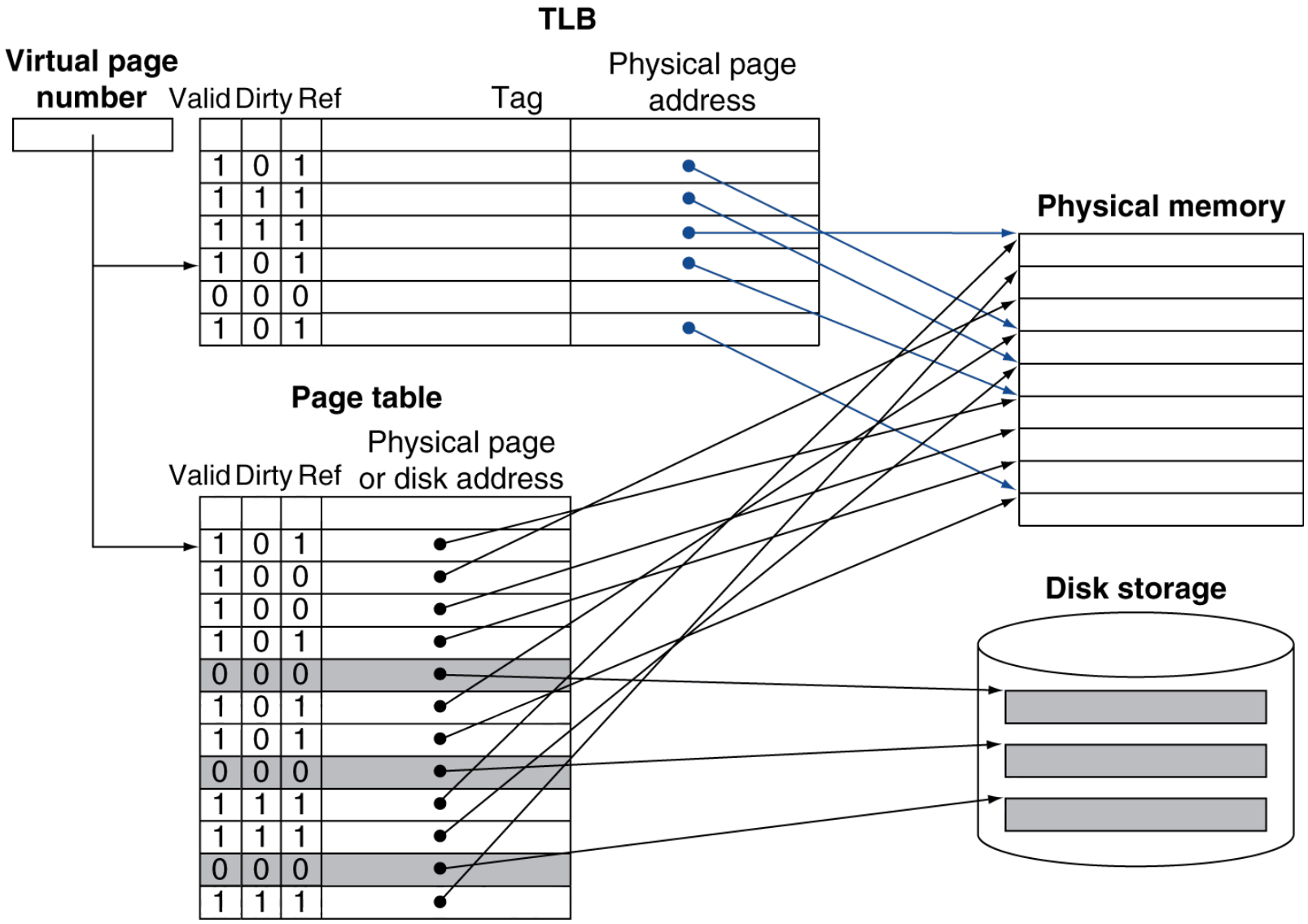


Fast Address Translation

- **Address translation would appear to require an extra memory reference**
  - **One to access the PTE, and next to access the page in memory!**

- **Observation: access to page tables also follows principle of locality!**
  - **So, use a fast cache of PTEs within the CPU**
  - **Called a Translation Look-aside Buffer (TLB)**
  - **Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate**
  - **One type of cache in which Fully Associative structure is common!**
  - **Misses could be handled by hardware or software**

Example: Using TLB



TLB Misses

- **TLB miss indicates:**
  - **Case-(1): page present in memory, but PTE not in TLB**
  - **case-(2): page not present in memory**

- **Must recognize TLB miss before destination register is overwritten**
  - **Raise (hardware) exception**

- **Handler copies PTE from memory to TLB**
  - **Then restarts instruction**
  - **If page is not present, page fault will occur!**

Page Fault Handler

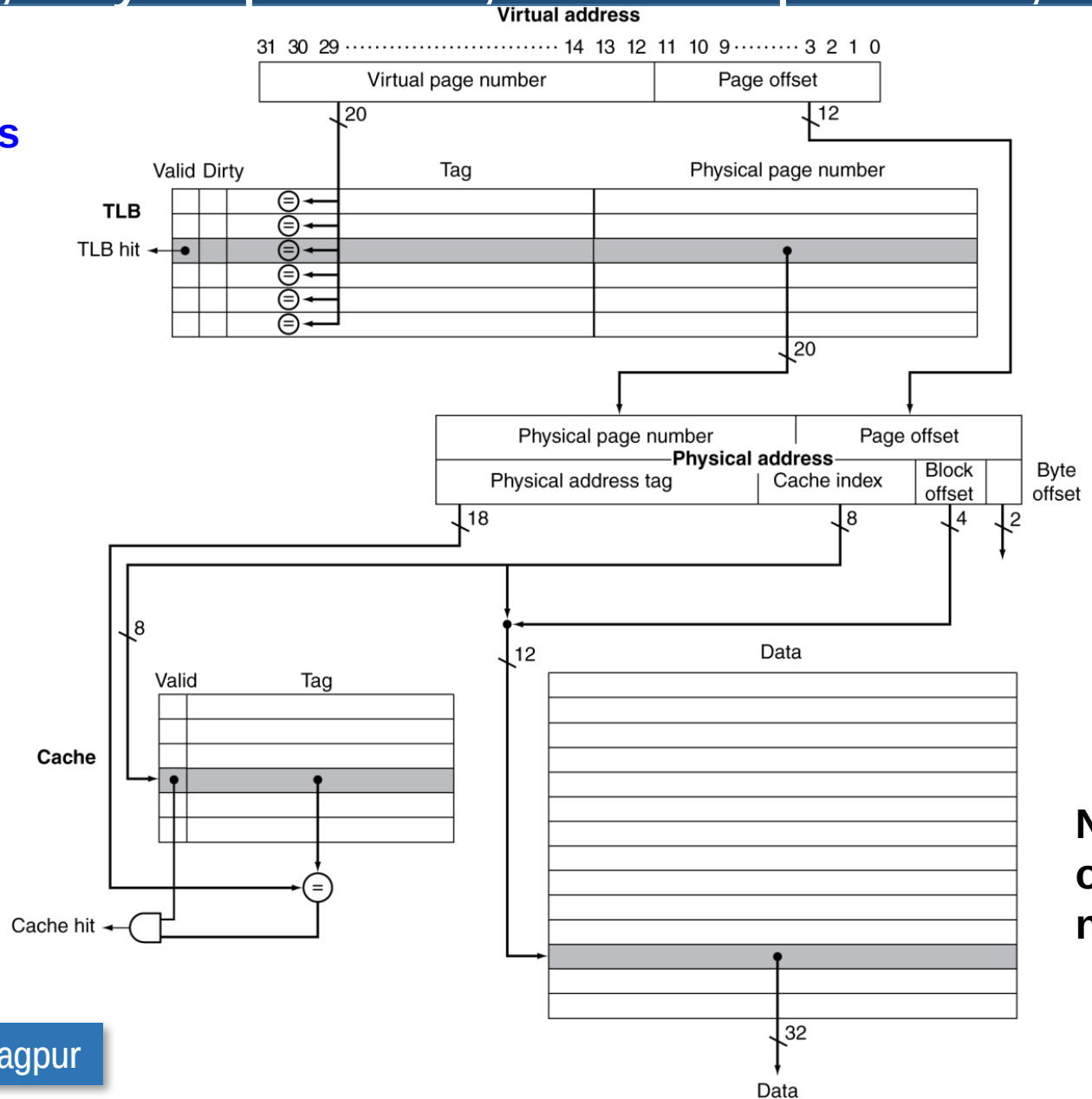
- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again (bring to “Ready” state)
  - Restart from faulting instruction

TLB and Cache Interaction

- **Virtually-addressed caches: “Virtual cache”**
  - **Have many shortcomings (size limitations, “aliasing”, etc.)**
- **CPU generates virtual (logical) addresses**
- **If L1 cache tag uses physical address:**
  - **Needs to translate each address before cache lookup**
  - **It will make it difficult to make L1 cache fast!**
- **Alternative: use virtual address tag!**
  - **Combine best of both worlds!**
  - **Virtually Addressed, Physically Tagged (VIPT) Cache**
  - **Very commonly adopted**

Example-1: VIPT Cache
(L1, Direct Mapped, 4 bytes per word, 16 words per block, i.e. 64 bytes per block)

32-bit Virtual Addresses
32-bit Physical Addresses

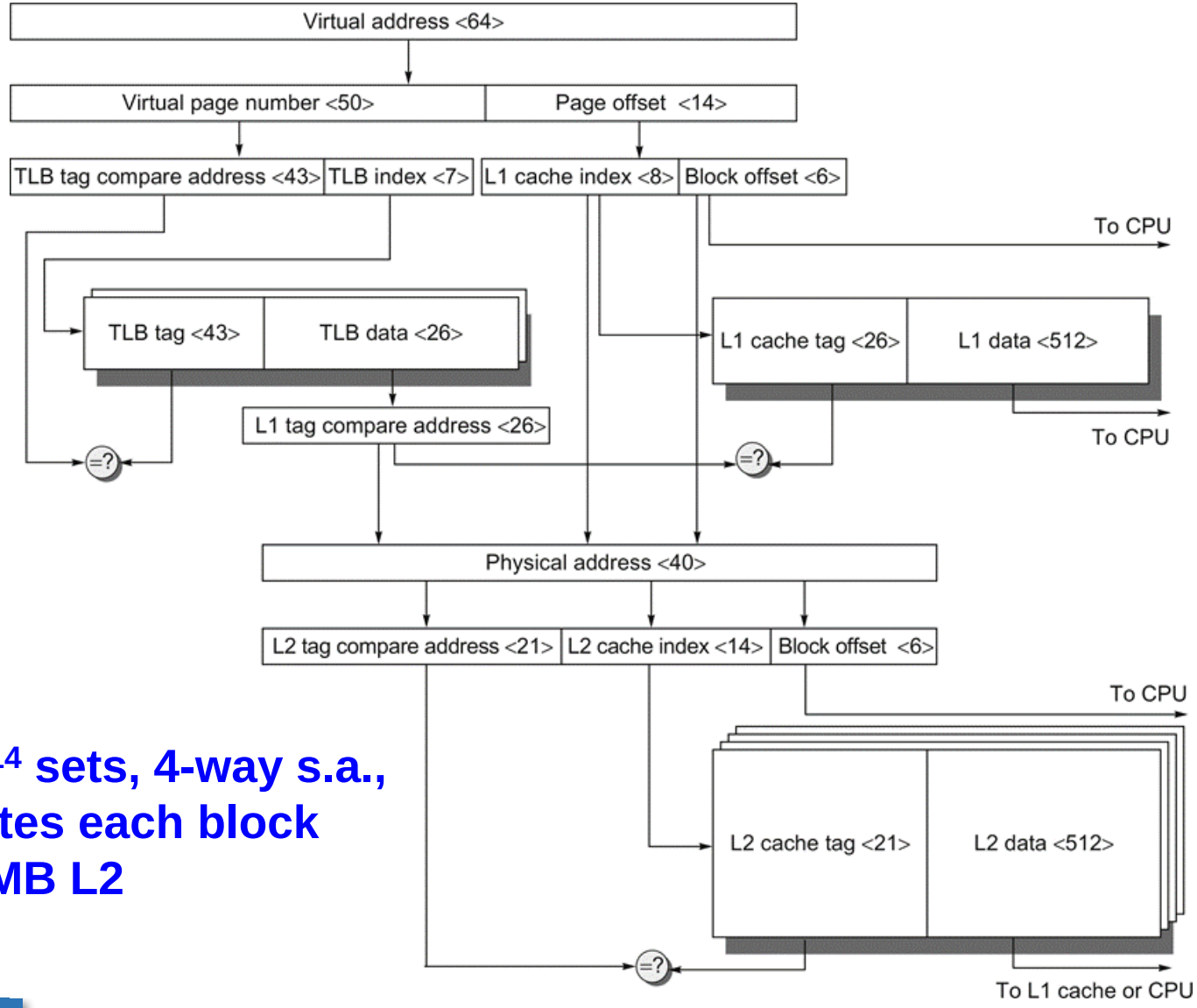


Note: multiplexor to select 1 out of 16 words in a block is not shown here!

Example-2: VIPT Cache (L1 and L2)

16kB pages
64 byte blocks

TLB: 2^7 sets, 2-way s.a.



L1: direct mapped, 2^8 blocks, 64 bytes each block => 16 kB L1

L2: 2^{14} sets, 4-way s.a., 64 bytes each block => 4 MB L2

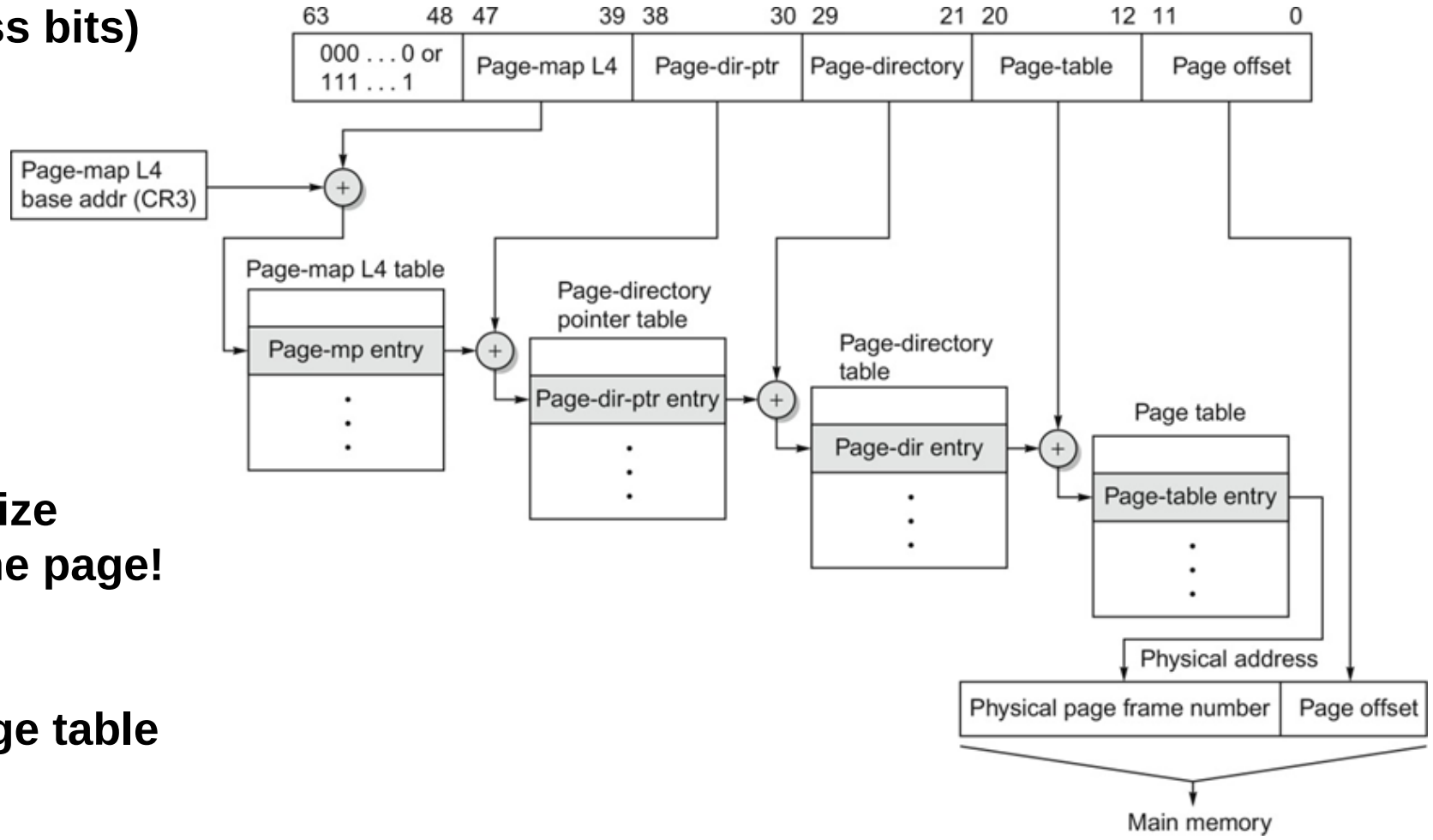
H&P CA:A QA (6th. Ed.)

Managing Large V.A.: Multi-level Page Tables (AMD Opteron)

48-bit V.A. used (out of 64 address bits)
Last 16 bits sign-extended
4 levels of page tables
512 entries in each page table
Each PTE: 8 bytes
CR: control register
CR3 also known as:
Page Table Base Register (PTBR)

12-bit page offset => 4 kB page size
Each page table fits in exactly one page!

Allows 4 MB pages => 3-level page table



Thank you