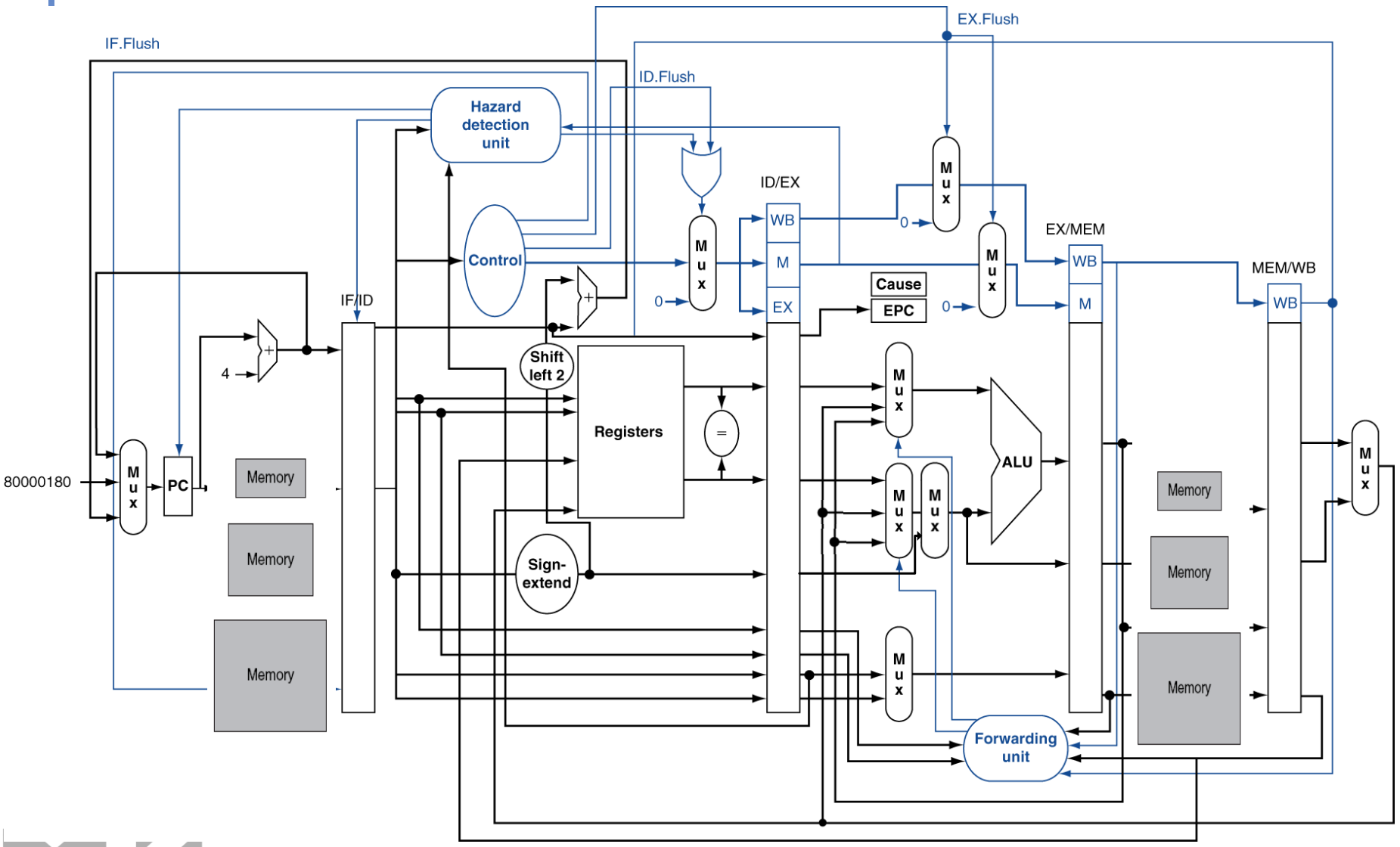
Topic 14

Memory Hierarchy

- Virtual Memory

MIPS Pipeline Architecture



Issues with Memory

- Computer may have a huge program (GByte)
  - Stored on a tera bytes of hard drive (TByte) – slow
  - But has to run on a Mbyte main memory – fast
- Computer may run multiple programs
  - Sharing the same main memory
  - We don’t want them to talk to each other
- CPU interacts with main memory (through cache)
  - CPU already has many other issues, doesn’t want to be bothered by issues brought by memory

Solutions to the Issues

- Make the programmer aware of the issues
  - Write smaller program
  - Carefully allocate different main memory sections to different programs

- Well, maybe a solution decades ago!

Solutions to the Issues

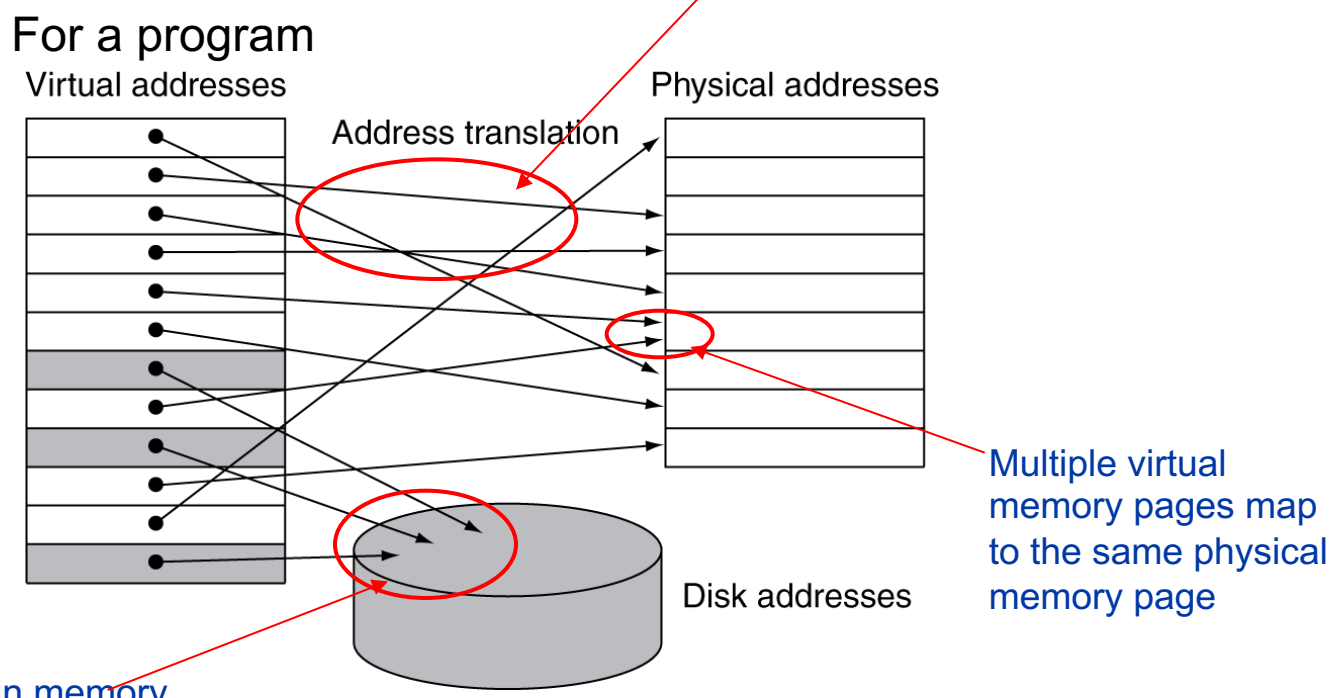
- Virtual Memory (VM)
  - An **imaginary huge and fast** memory from CPU's perspective – mapped to physical main memory or hard drive
  - Each program has a virtual (memory) space on hard drive – mapped to different sections of physical memory
  - Mapping is done by CPU or OS translating specific **virtual addresses** to specific **physical addresses**

What is Virtual Memory (VM)

- Big
  - It can be as big as needed
  - It's an illusion of a program
- Private
  - VM is a memory that a program owns entirely
  - Each program has a separate and private VM space holding its data and instructions
- Cover
  - It hides constrains and complications of memory from the CPU and programmer

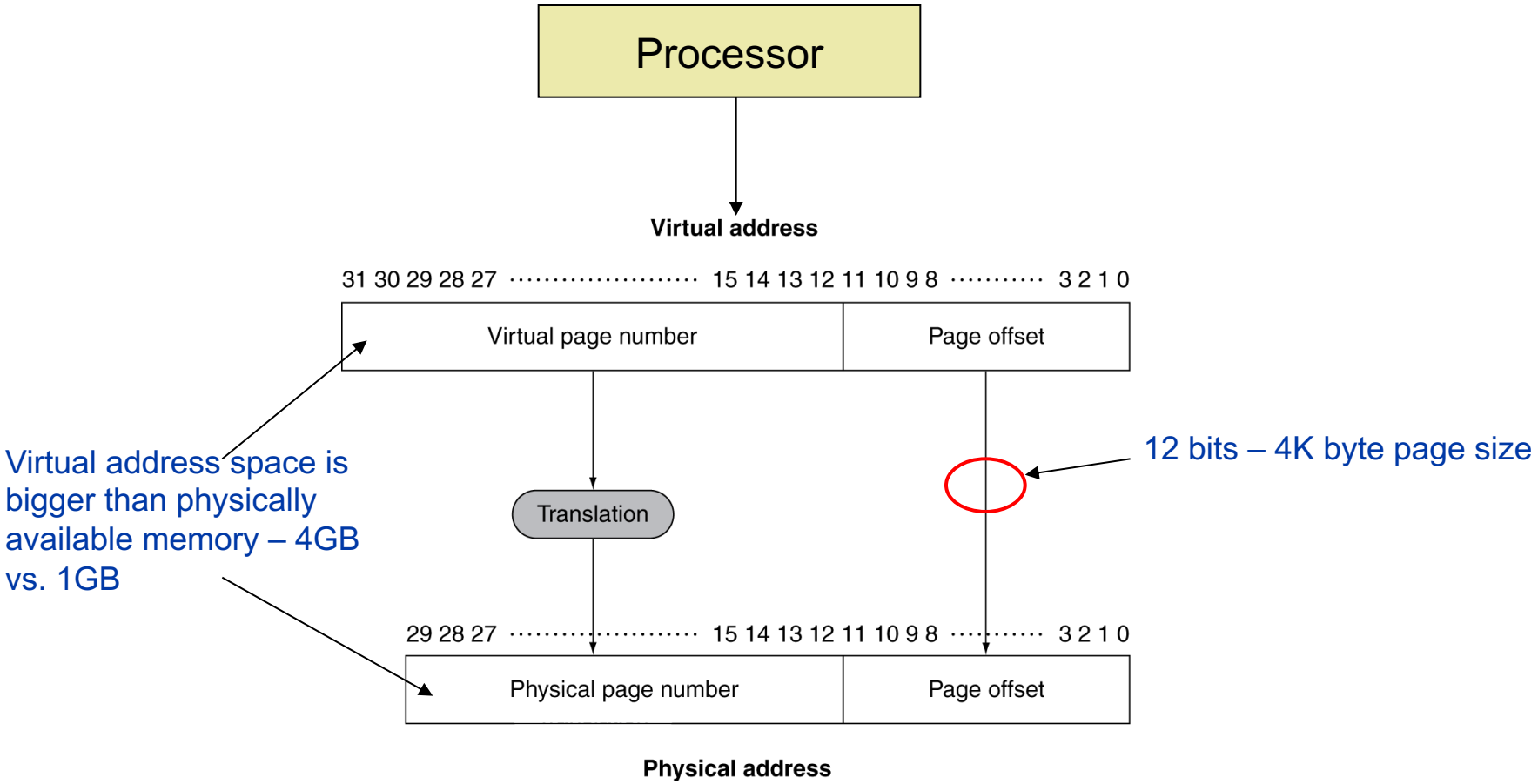
VM Terminology

- Use main memory as a “cache” for hard disk
- Concepts in VM
  - VM “block” is called a **page** (bigger)
  - VM “miss” is called a **page fault**



Address Translation

- Assuming fixed-size pages (e.g., 4K Bytes)



Address Mapping

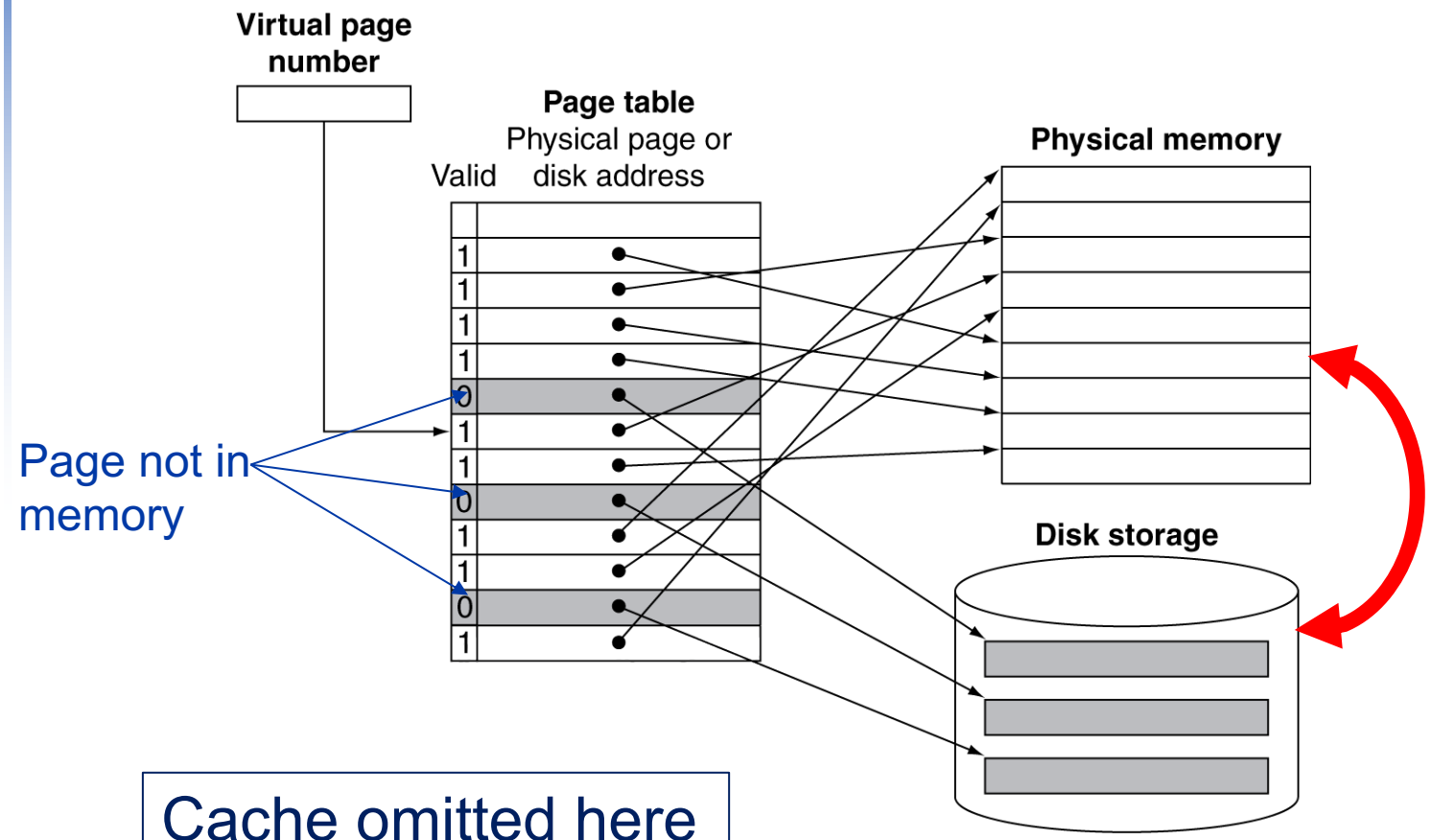
Translator – Page Tables

- Each program has one translator – page table
  - Stores the mapping of all virtual to physical addresses
  - Indexed by virtual page numbers
  - Located in main memory
  - **Page table register** in CPU points to the beginning of page table of the program running in the physical memory currently

Translator – Page Tables

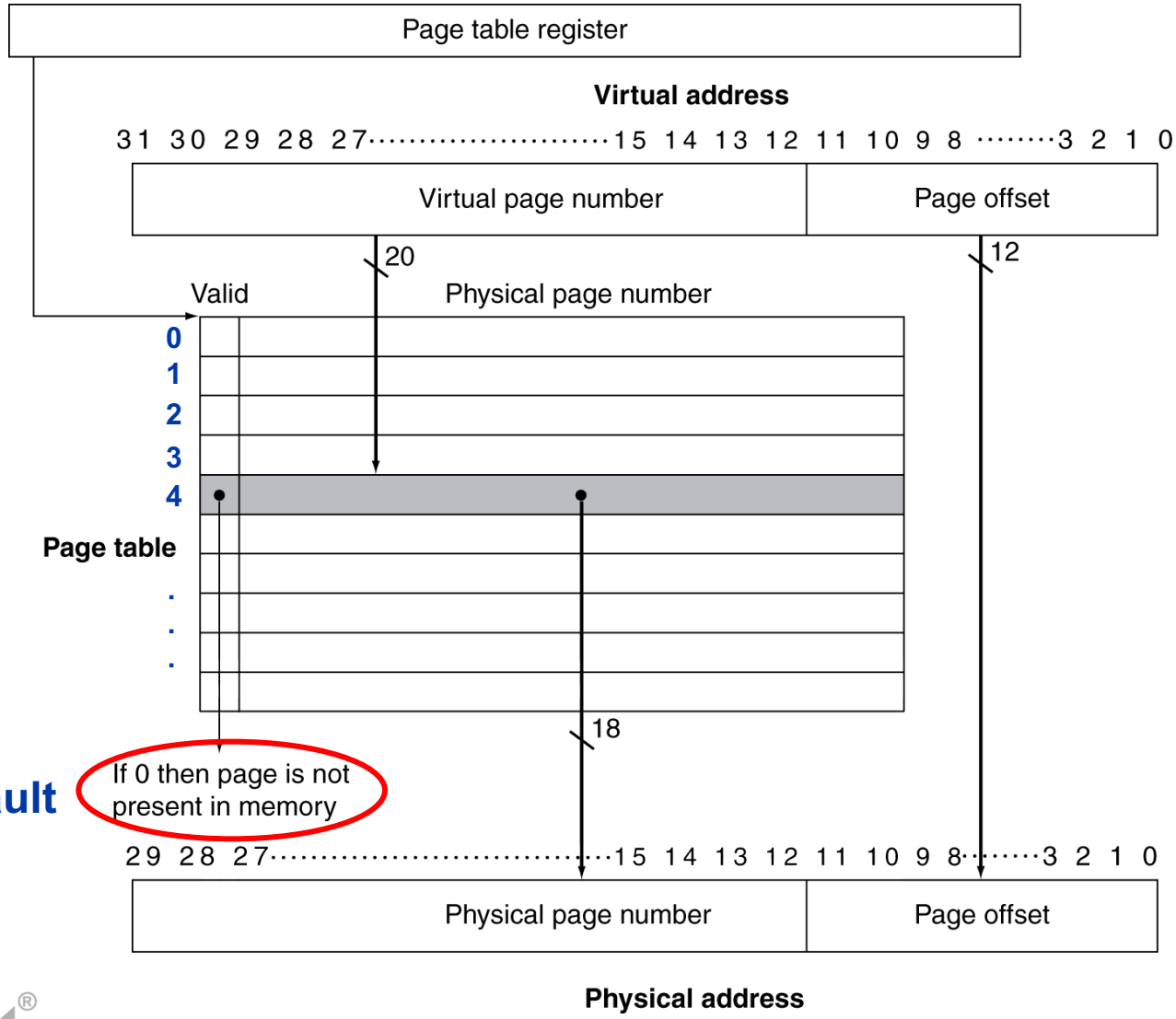
- If page is present in memory
  - Page table stores the physical page address
  - Plus other status bits (valid, dirty, …)
- If page is not present in memory – page fault
  - All virtual pages for a program are mapped to a unique **swap space** on disk
  - Page table can refer to locations in swap space of a program on disk

Mapping Pages to Storage

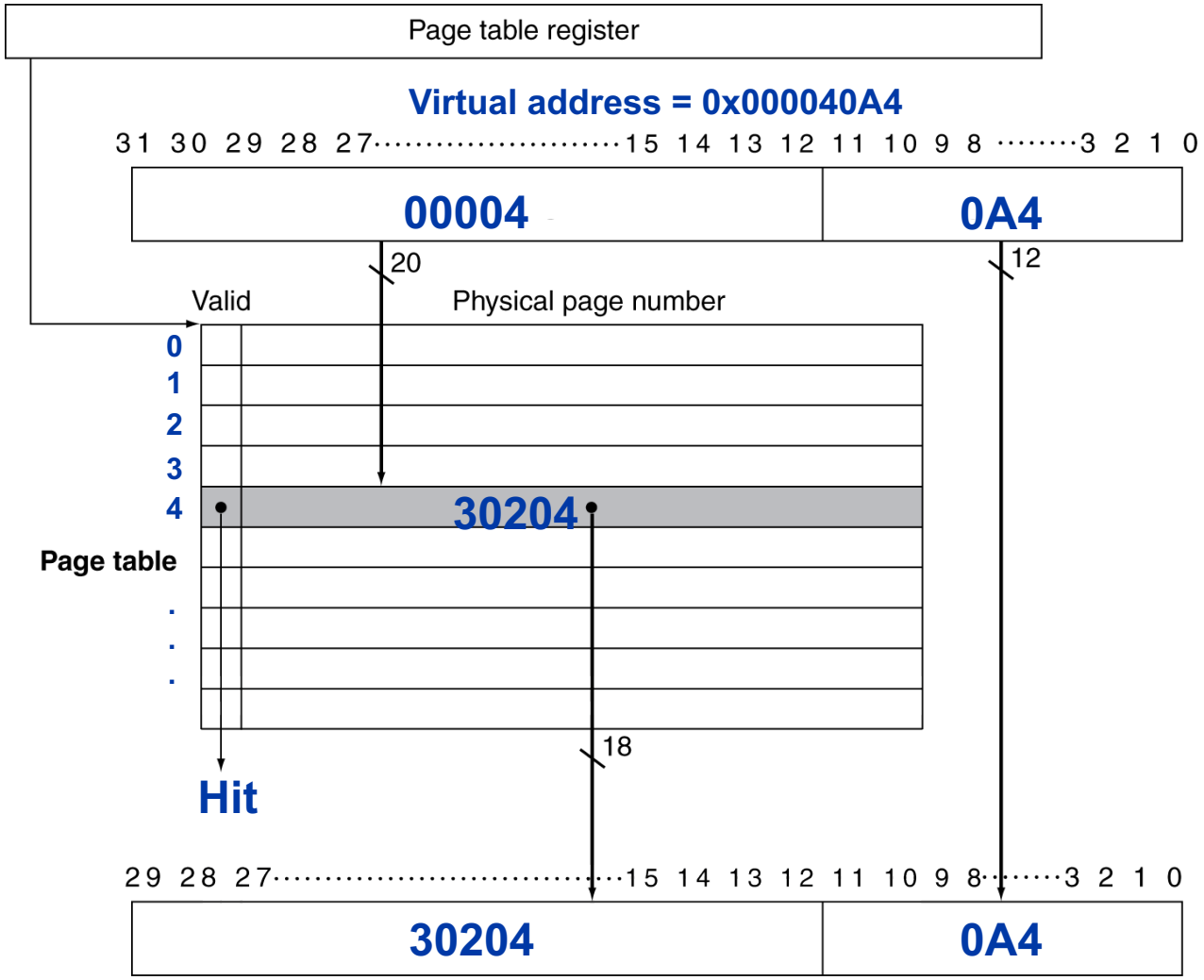


Cache omitted here for now

Translation Using a Page Table



Example


Page table register
Virtual address = 0x000040A4
31 30 29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
00004
0A4
20
12
Valid
Physical page number
0
1
2
3
4
30204
Page table
18
Hit
29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
30204
0A4
Physical address = 0x302040A4

Page Table Size

- Example:
  - Page size: 4KB
  - 32-bit virtual byte address (4GB)
  - 4 bytes per page table entry
- Number of page table entries = $2^{(32-12)} = 2^{20}$
  - Up to, might be customized for each program and usually less than that
- Size of page table = number of page table entries x bytes/page table entry
  - Page table size = 2^{20} x 4 = 4 MB

Reducing Page Table Size

- Limit register
  - Restricts number of page table entries
  - Page table expands as needed
- Multiple levels of page table
  - E.g. segment table → page table
  - Total size not smaller, but non-contiguous storage for page table
- Allow page table to go virtual
  - While only having part of the page table in memory
  - The other part in disk

Page Fault

- Page Fault
  - Requested page in virtual address space not mapped to page in main memory
- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS
- Should try to minimize page fault rate

Reduce Page Fault Rate and Penalty

- Most of the time is for getting the first word in the page – access time – very long
  - Should have **large page size**, so one access fetches more data, also reduces page fault rate
  - Reduce page fault rate by **full associativity**
  - Handle page fault by software – more sophisticated and less expensive
  - Use **write-back**

Handling Page Fault

- On page fault (page table valid bit is 0)
  - Find the page on disk
  - Fetch and put it in main memory
- Locate a page from disk
  - All virtual pages for a program are in a unique swap space on disk
  - Some page table entries contain disk addresses
  - Or could have a separate page table for disk pages

Page Writes

- Disk writes take millions of cycles
  - Write through is impractical, even with write buffer
    - Millions of processor clock cycles
  - Use write-back
    - Dirty bit in page table is set when page is written
    - Write-back first if dirty bit is on
  - Writing entire page is more time efficient than writing a word
  - CPU switches to another process/program while waiting – context switch

Page Replacement

- Least-recently used (LRU) replacement
  - Lower page fault rate – temporal locality
  - Reference bit (aka use bit) in page table
    - 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 – to be replaced

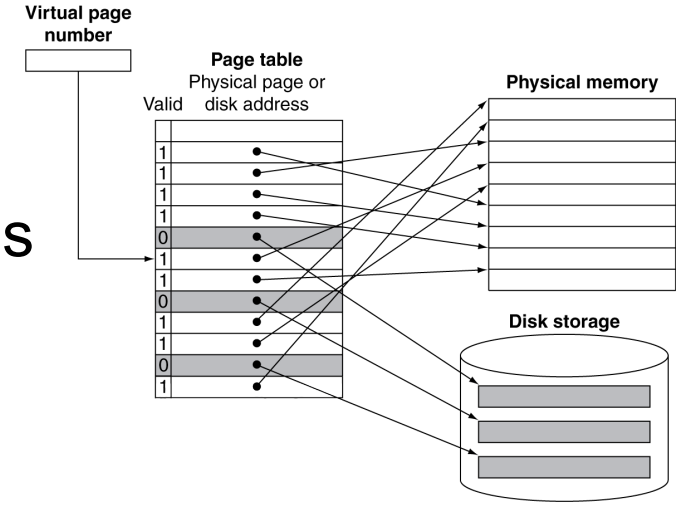
Class Exercise

- Given
  - 4KB page size, 16KB physical memory, LRU replacement
  - Virtual address: byte addressable, 20 bits (how many bytes?)
  - Page table for program A stored in page #0 of physical memory, starting at address 0x0100, assume only 2 valid entries in page table:
    - Virtual page number 0 => physical page number 1
    - Virtual page number 1 => physical page number 2
- Show physical memory including page table
- Complete following table

Virtual Address	Virtual page number	Page fault?	Physical Address
0x00F0C	00	N	1F0C
0x01F0C	01	N	21-0C
0x20F0C	20	Y	3F0C
0x00100		N	1100
0x00200		N	1200
0x30000		Y	2000
0x01FFF		Y	31-F
0x00200		N	1200

Fast Translation Using a TLB

- Address translation requires extra memory access
  - One to access the page table
  - Then the actual memory access

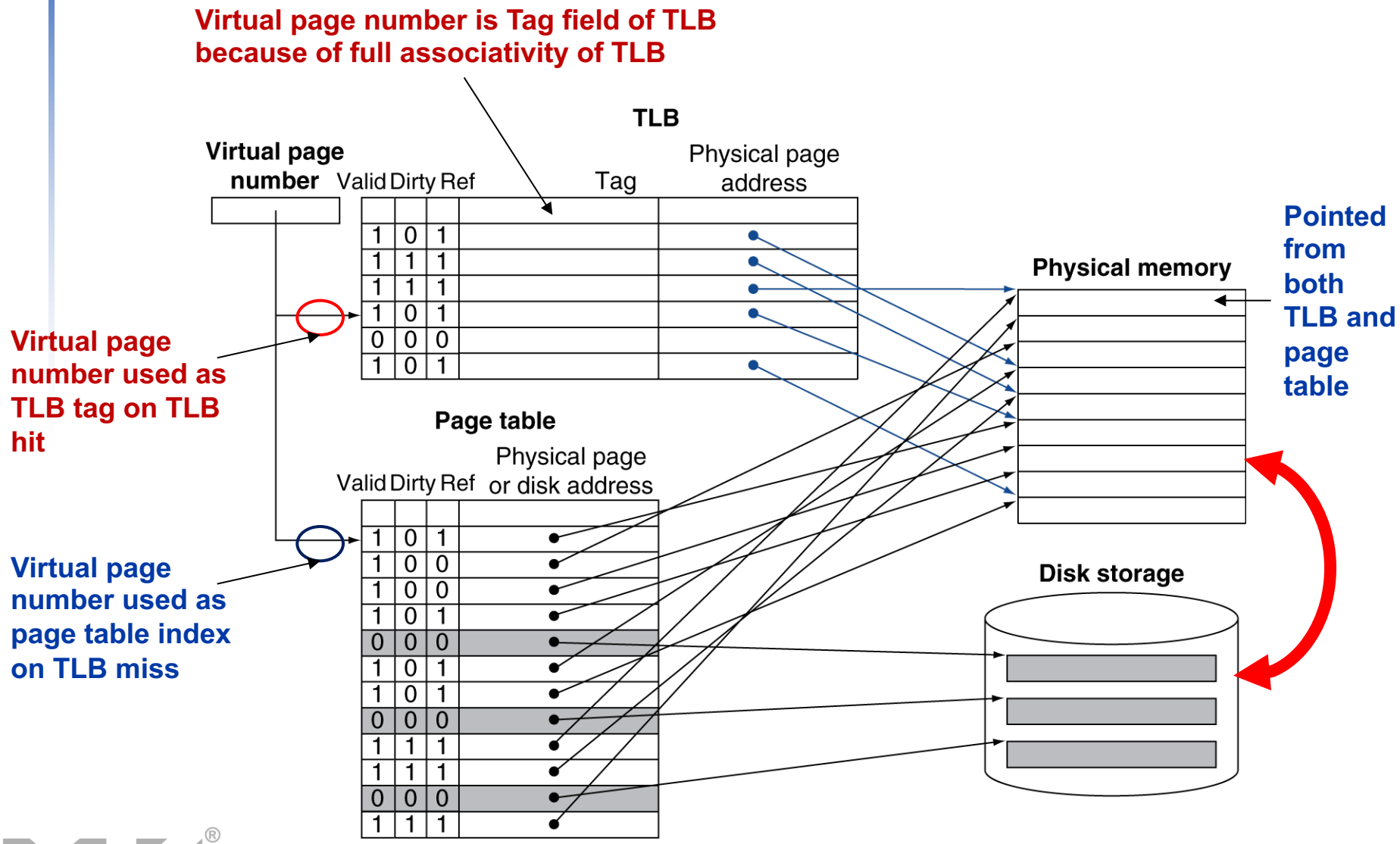


- But access to page tables has good locality
  - So use a fast **cache** of page tables within the CPU
  - Called a **Translation Look-aside Buffer (TLB)**
  - (TLB ← page table) = (cache ← main memory)

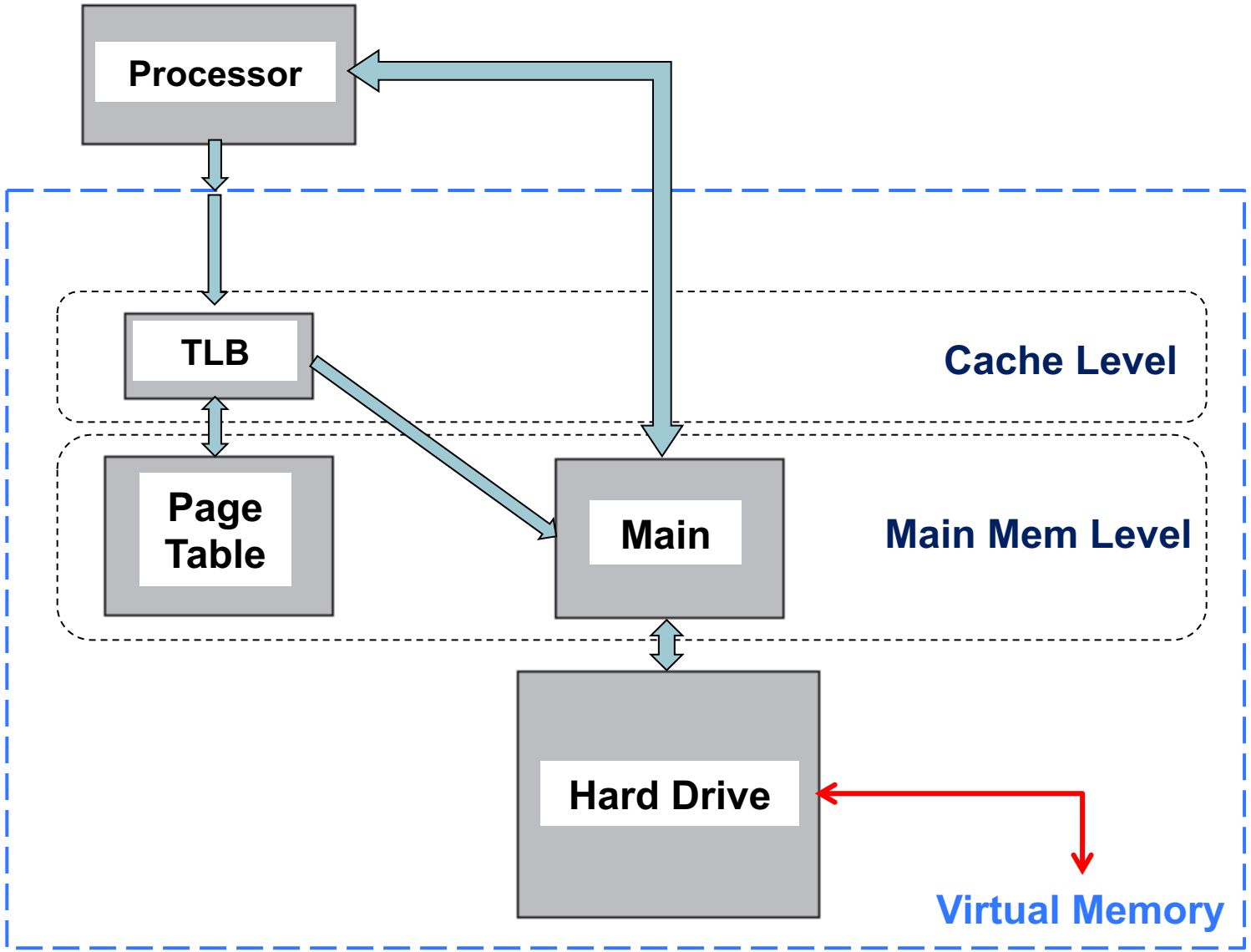
TLB

- Load part of the Page Table in cache
- One translation per TLB entry
- Typical: 16–512 entries, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
- Status bits
  - Valid: empty or not
  - Dirty: target memory was changed
  - Reference: target memory was used
- Full Associativity
  - Lower miss rate
  - Small TLB, access time not a major concern
  - LRU or random replacement

Fast Translation Using a TLB



Access Mem through TLB



TLB Hit

- Virtual address found in TLB – hit
  - Provide physical address
  - Reference bit on
  - Dirty bit on if physical address used for write (might not be needed if dirty bit maintained in physical memory)

TLB Miss

- Virtual address not in TLB – TLB miss
  - If page is in memory (page table valid bit = 1)
    - Load the page table entry from memory to TLB
      - Why not just use page table since it's accessed anyway?
    - Could be handled in hardware or in software
  - If page is not in memory (page fault)
    - Page fault exception – OS handles fetching the page and updating the page table and TLB
    - Then restart the faulting instruction

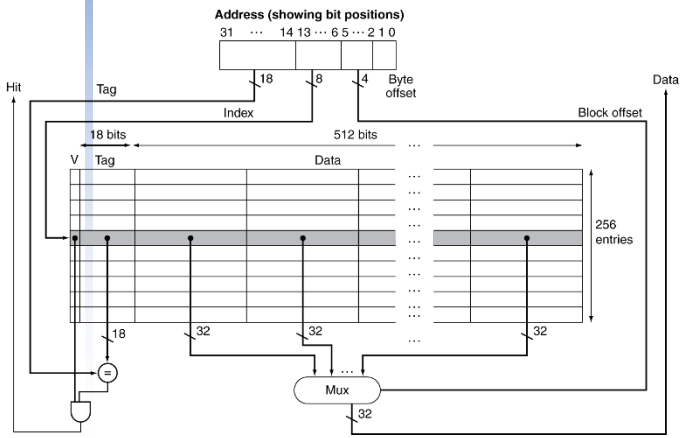
TLB Miss

- Recognize a TLB miss or page fault
  - TLB miss – by cache tag and cache valid bit
  - Page fault – by page table valid bit

Handling TLB Miss

- Copy page table entry to TLB
- If TLB is full, replace a TLB entry
  - Ref bit and dirty bit of replaced entry should be copied back to page table
    - These bits need to be copied back to page table only when entry is replaced – write back
  - Not for valid bit – why?
  - Different meaning of Ref bit – why?
  - Other techniques used to keep track of ref/dirty, no need to write back

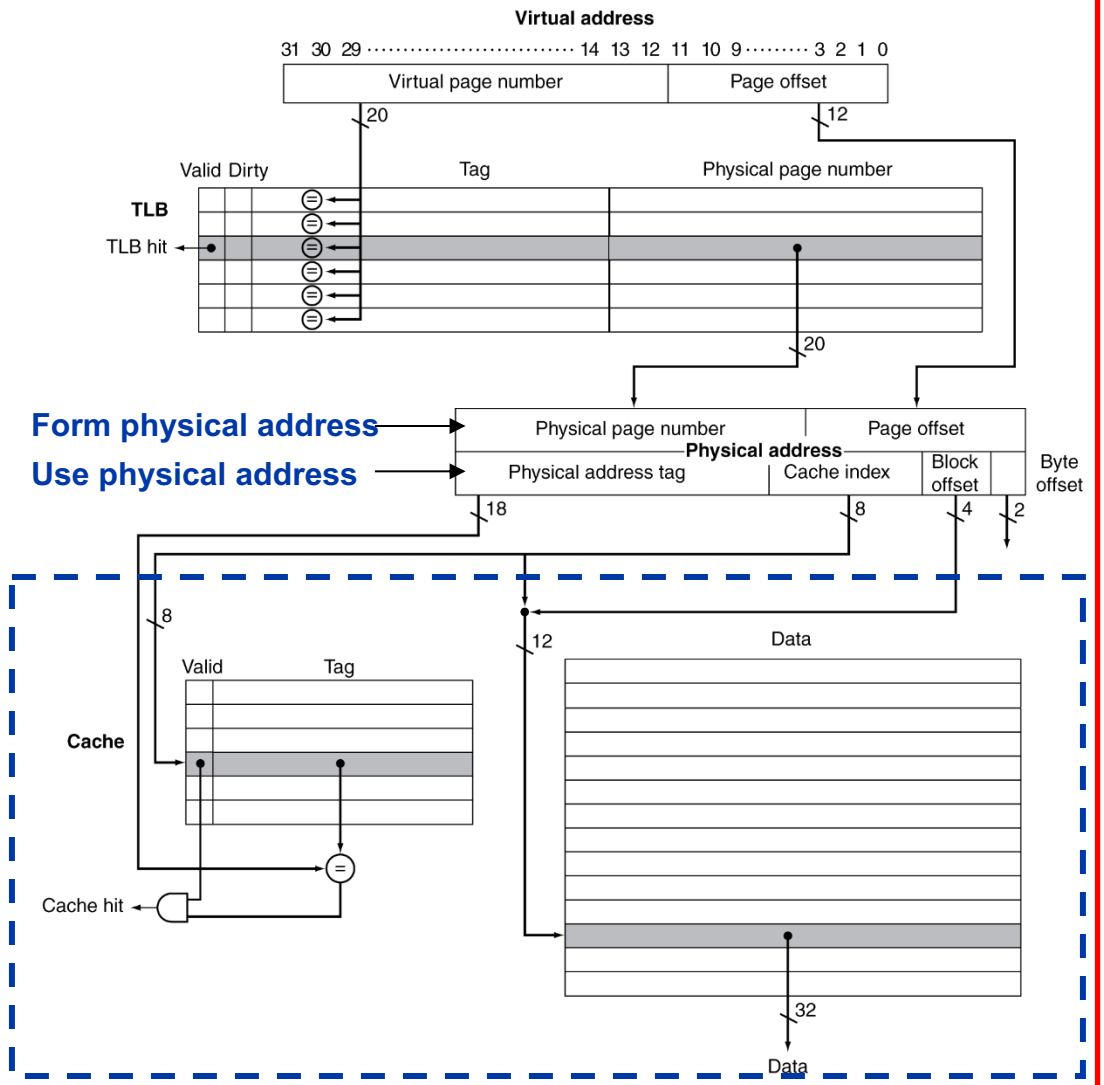
TLB and Cache Interaction



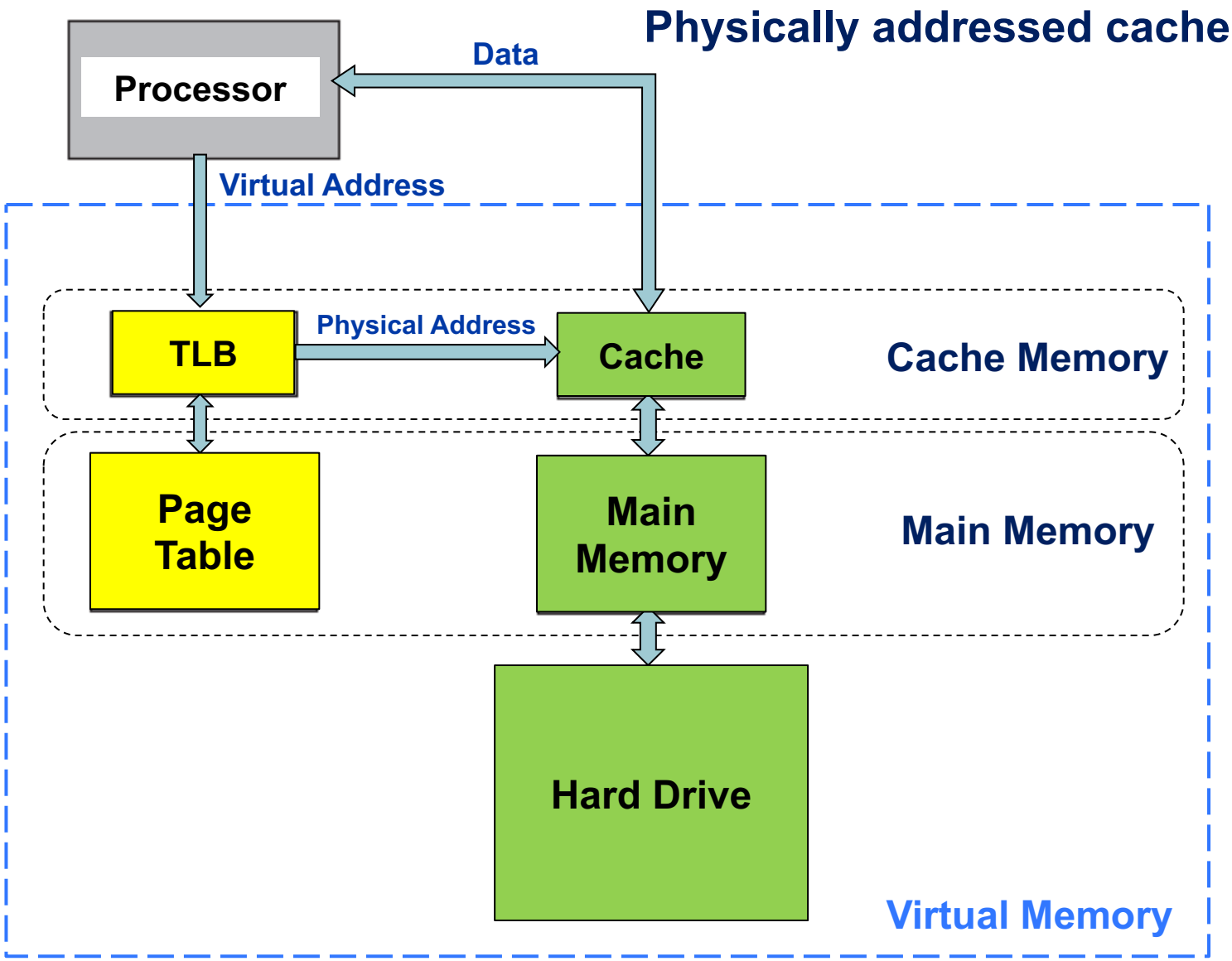
Small RAM for valid and tags
Big RAM aligned in word for data

- Indexed by cache index and block offset
- **No need for big MUX, improved cache access time**

Assuming 4K page size



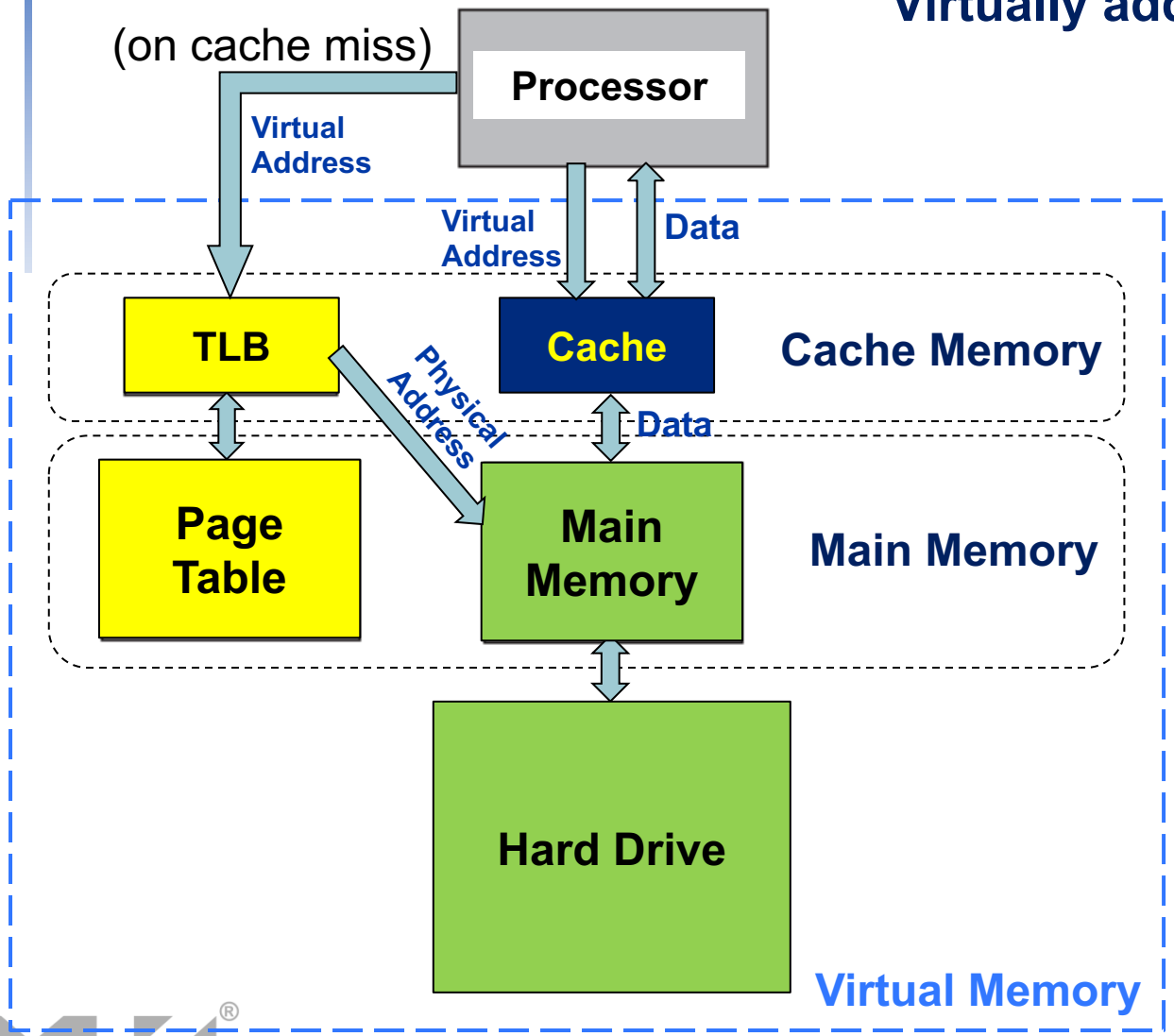
TLB vs. Cache


Physically addressed cache
Processor
Data
Virtual Address
TLB
Physical Address
Cache
Cache Memory
Page Table
Main Memory
Main Memory
Hard Drive
Virtual Memory

TLB and Cache Interaction

- If cache uses physical address
  - Need to translate before access cache
  - Virtual Addr → TLB → cache
  - Having TLB on the critical path, taking longer time

TLB vs. Cache

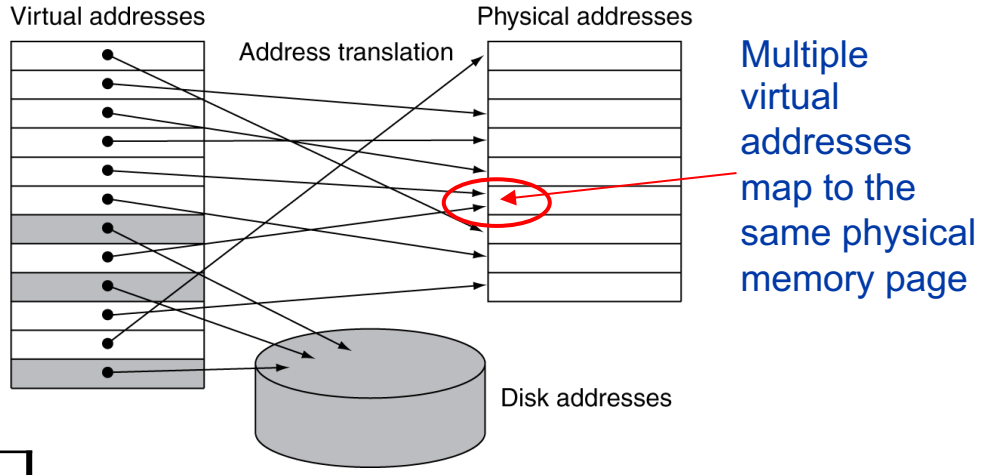


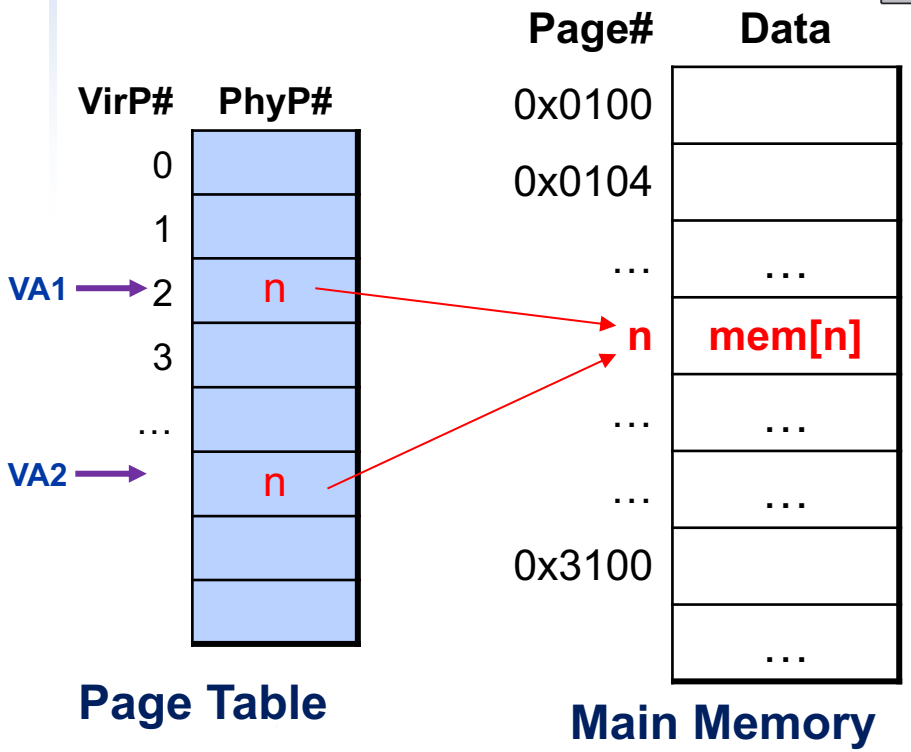
Virtually addressed cache

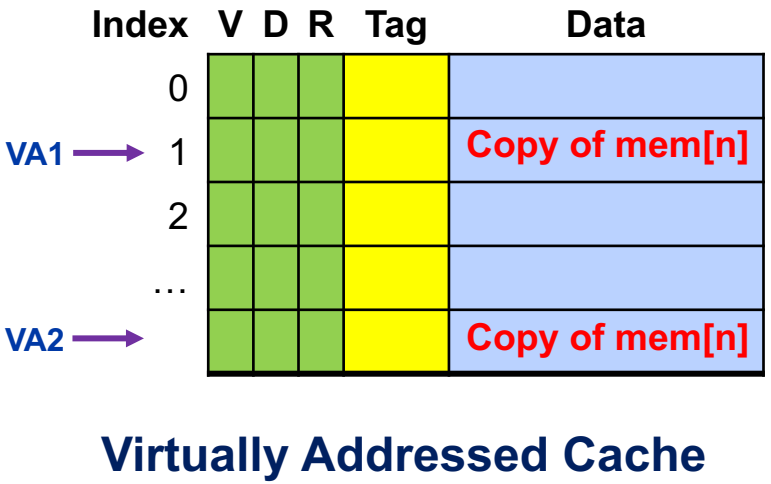
- No need for translation
- Translate when cache miss
- Complications due to aliasing
  - When different virtual spaces map to the same physical main memory location

Virtually Addressed Cache

Aliasing: the same object has two different names







Class Exercise with TLB

- Given
  - 4KB page size, 16KB physical memory, LRU replacement
  - Virtual address: byte addressable, 20 bits (how many bytes?)
  - Page table for program A stored in page #0 of physical memory, starting at address 0x0100, assume only 2 valid entries in page table:
    - Virtual page number 0 => physical page number 1
    - Virtual page number 1 => physical page number 2
- Fully associative TLB, 2 entries
- Show the memory structure, complete following table

Virtual Address	Virtual page number	TLB miss?	Page fault?	Physical Address
0x00F0C				
0x01F0C				
0x20F0C				
0x00100				
0x00200				
0x30000				
0x01FFF				
0x00200				

Relationships in Memory Hierarchy

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

- Assume cache uses physical addresses
- All data in a memory must also be present in its lower level
  - Impossible to copy data cross levels

Memory Protection

- Different programs can have the same virtual addresses
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions available in supervisor mode
    - System call exception (e.g., syscall in MIPS)
  - Page tables and other status information only updated in supervisor mode by operating system
    - Write enable bit protects memory from being written
  - Different page tables ensure different translations
    - Different translation ensures separate memory locations

Summary of The Memory Hierarchy

The BIG Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware costs
  - Reduce comparisons to reduce cost
- Virtual memory
  - Lookup table full associativity
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Needs tracking mechanism (reference bit updated and periodically refreshed)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU for high associtivity, easier to implement

Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given unaffordable disk write latency

Sources of Misses (3C model)

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for blocks in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

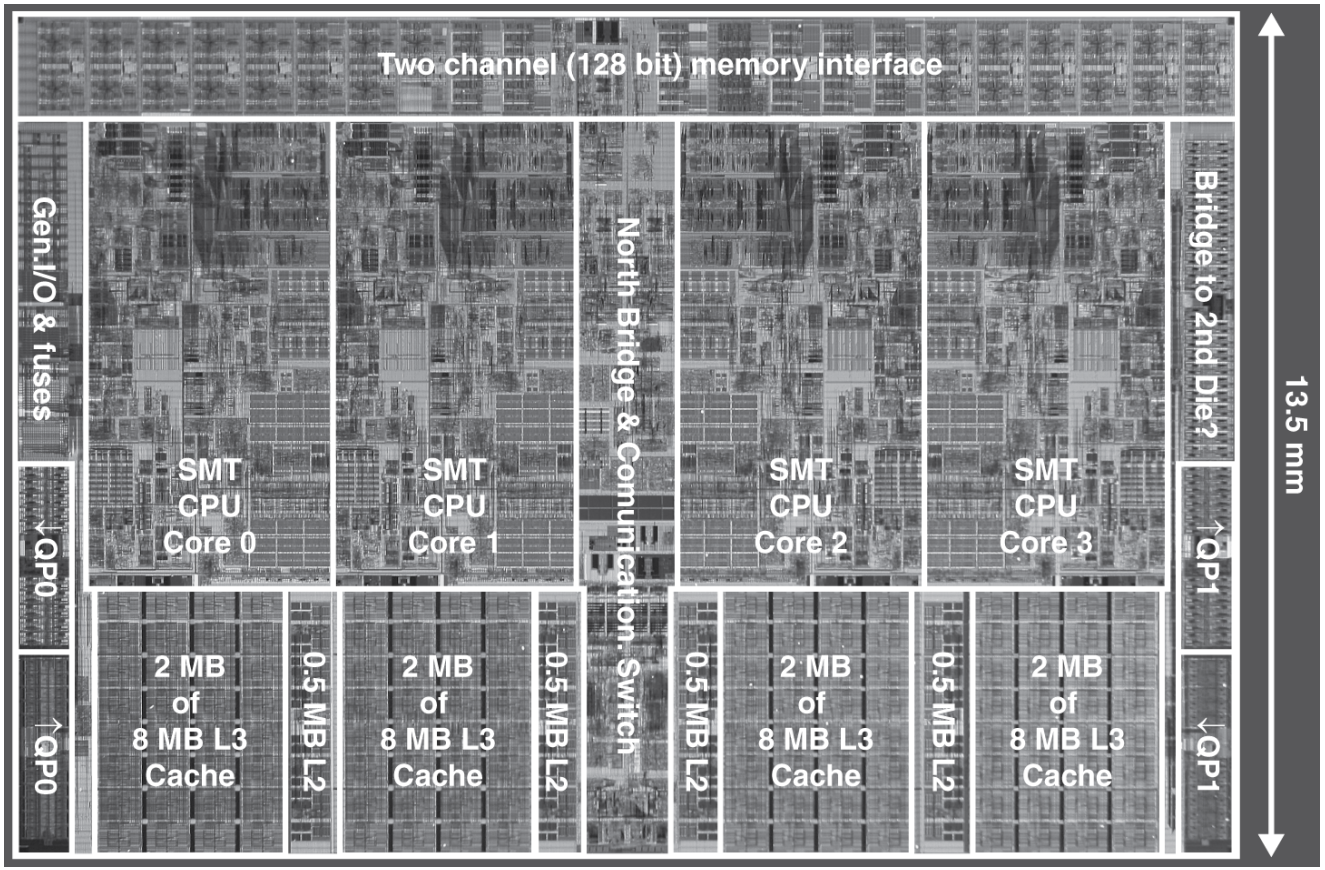
Design change	*Effect on miss rate*	*Negative performance effect*
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	Will increase access time
Increase block size	Decrease compulsory misses May increase conflict misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Miss Penalty Reduction

- Return requested word first
  - Then back-fill rest of block
- Non-blocking miss processing
  - Hit under miss: allow hits to proceed
  - Miss under miss: allow multiple outstanding misses
    - Need parallel processing capability
- bank interleaved cache/main memory
  - multiple concurrent accesses per cycle

Example: Intel Nehalem 4-core processor

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

2-Level TLB

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- Principle of locality
  - Programs use a small part of their memory space frequently
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
  - Vitual space → L1 TLB → L2 TLB → (page table) → physical space
- Memory system design is critical for multiprocessors