Chapter 5

Virtual Memory

Outline

- Virtual Machines
- **Virtual Memory**
- A Common Framework for Memory Hierarchies
- Using a Finite State Machine to Control A Simple Cache
- Parallelism and Memory Hierarchies: **Cache Coherence**
- The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
- Real Stuff: RISC-V System and Special Instructions
- Going Faster: Cache Blocking and Matrix Multiply
- Fallacies and Pitfalls
- Concluding Remarks

Virtual Machines

- Host computer emulates guest operating system and machine resources
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- Virtualization has some performance impact
  - Feasible with modern high-performance computers
- Examples
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machine Monitor

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest

Example: Timer Virtualization

- In native machine, on timer interrupt
  - OS suspends current process, handles interrupt, selects and resumes next process
- With Virtual Machine Monitor
  - VMM suspends current VM, handles interrupt, selects and resumes next VM
- If a VM requires timer interrupts
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs

Instruction Set Support

- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers
- Renaissance of virtualization support
  - Current ISAs (e.g., x86) adapting

Virtual Memory Origin

- To execute a program larger than available memory size
- Load from disk parts of a program to memory
  - Load on-demand when a ”page” is needed.
  - Some “page” in memory will be replaced.
- Not so useful because of slow HD access time.
  - **Thrashing** --- when next required page is always in disk

Virtual Address

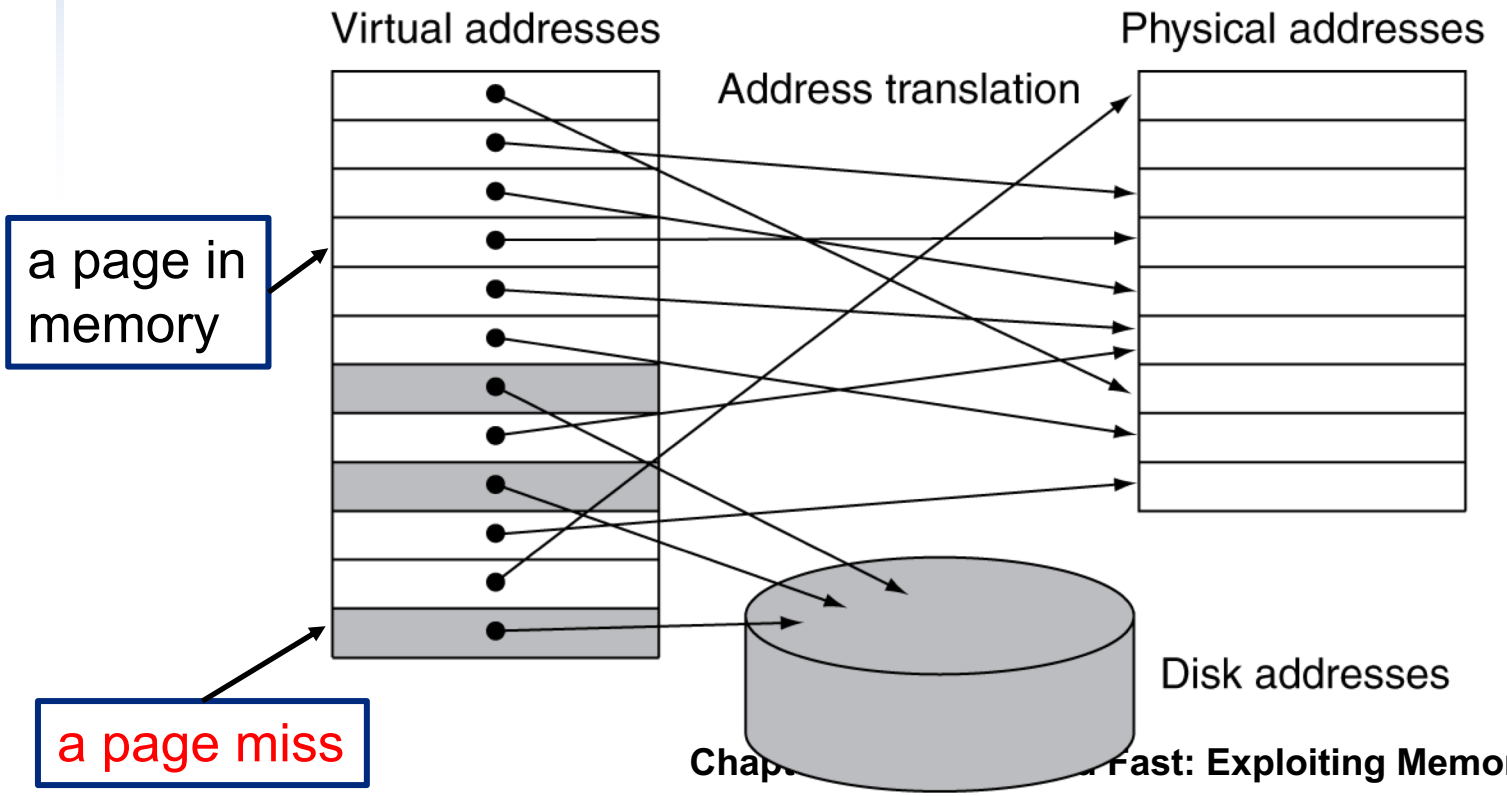
- To manage a program's memory, it's convenient to use a uniform memory map
  - from 0x00000000 – 0xFFFFFFFF
  - This is called **virtual address**
  - Now programmers are free from assigning physical addresses and media
- O.S. handles the mapping to physical memory address or disk space

Virtual Memory Usage

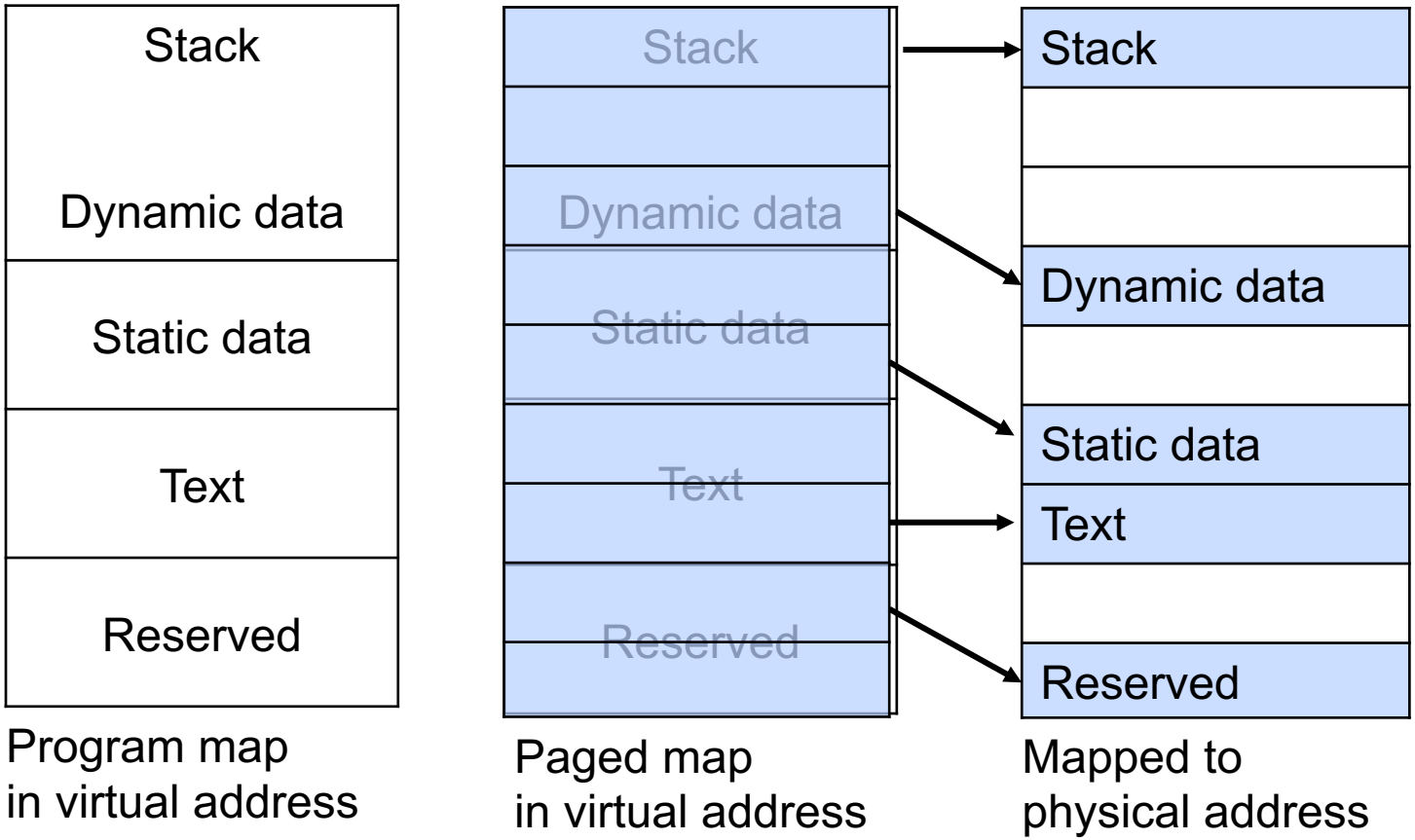
- Multiple programs share the same main memory
  - Each gets a private virtual address space
- Can protect access from other programs
  - Ownership of physical pages is enforced by O.S.
  - One program cannot arbitrarily access another program's memory pages

Virtual vs. Physical Address

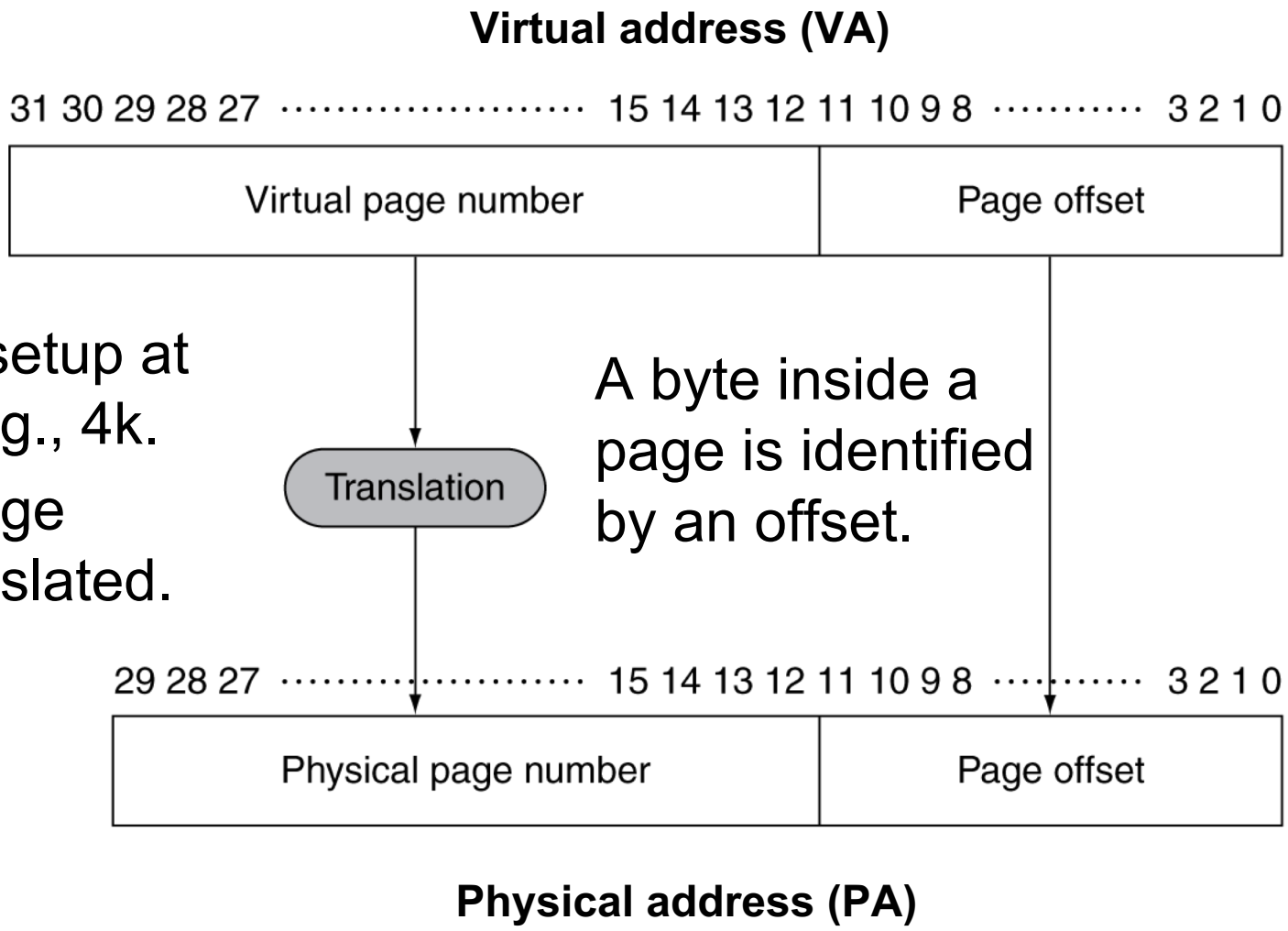
- CPU and OS translate virtual addresses to physical addresses
- VM “block” is called a page
- VM translation “miss” is called a page fault



Program Memory Map



Address Translation



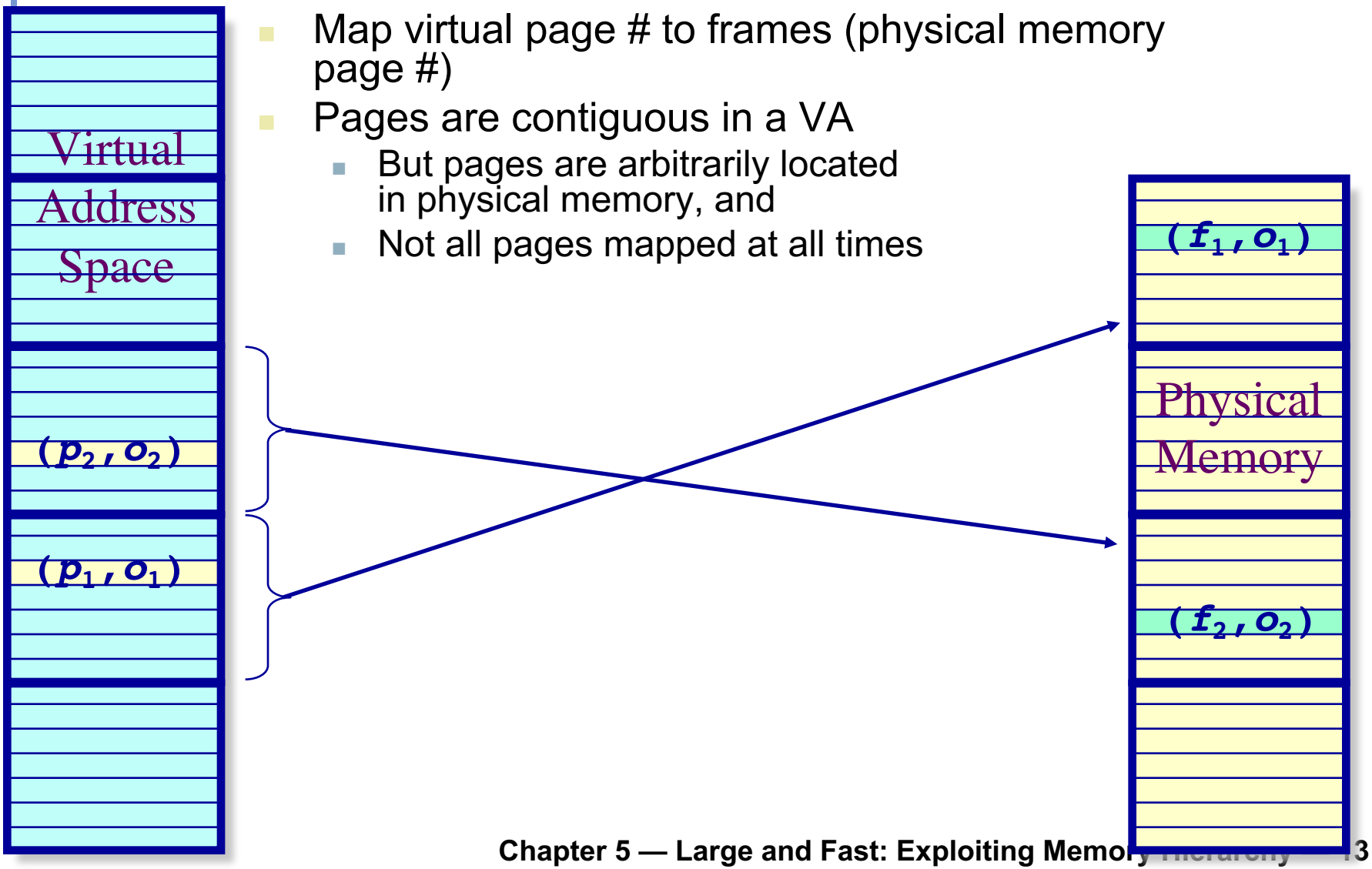
Each page is setup at a fixed size, e.g., 4k.

Only virtual page number is translated.

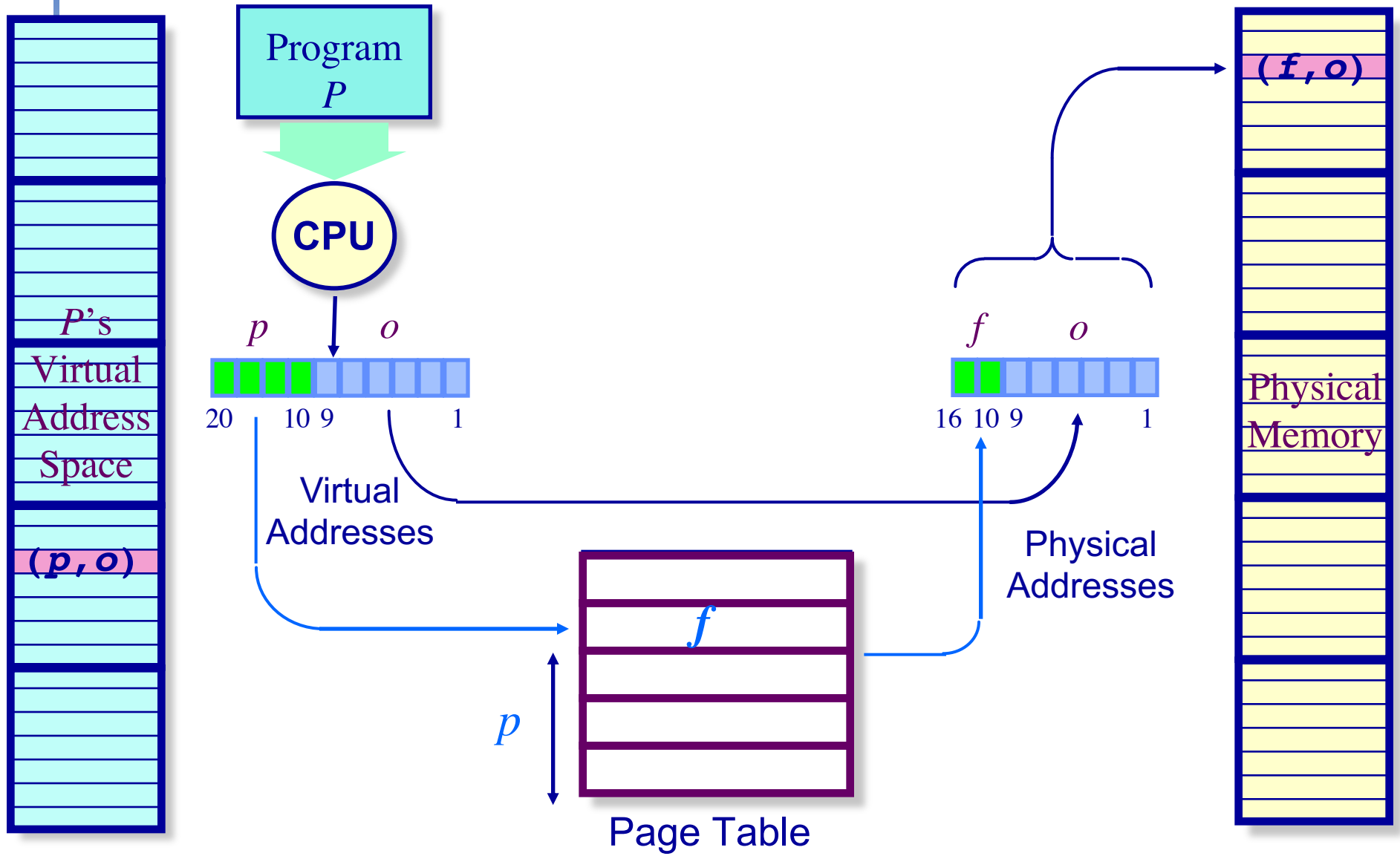
A byte inside a page is identified by an offset.

Map VA to PA

- Map virtual page # to frames (physical memory page #)
- Pages are contiguous in a VA
  - But pages are arbitrarily located in physical memory, and
  - Not all pages mapped at all times



Virtual Address Translation



Memory Access Latency

- All memory accesses now need virtual to physical translation
  - instruction fetch, function address, global data, etc.
- Increased access latency
- Overlap to optimize (more on this later).

Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

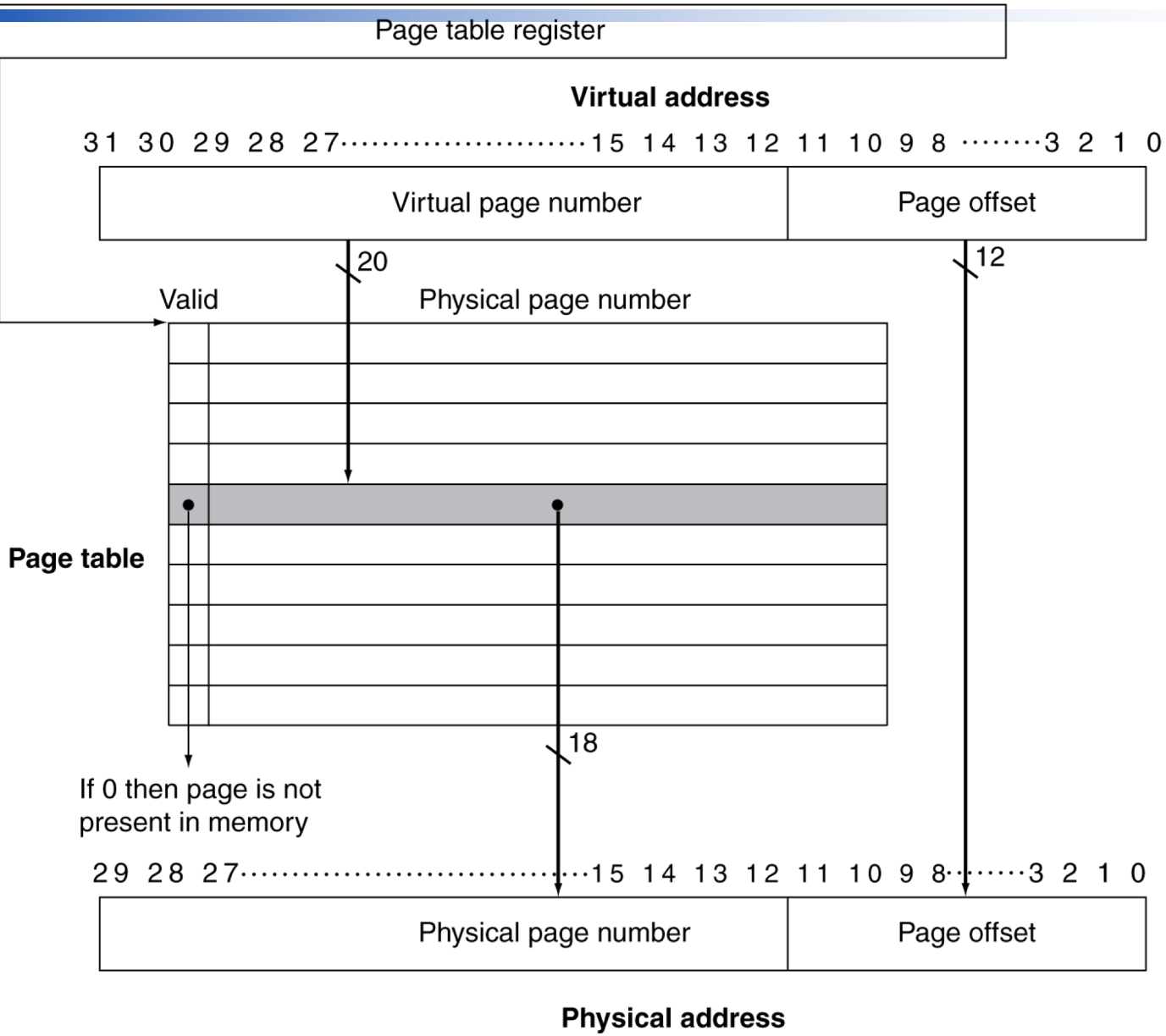
Page Fault Process

- Page fault handling steps:
  - Processor runs the interrupt handler
  - OS blocks the running process
  - OS starts read of the unmapped page
  - OS resumes/initiates some other process
  - Read of page completes
  - OS maps the missing page into memory
  - OS restart the faulting process

Page Tables

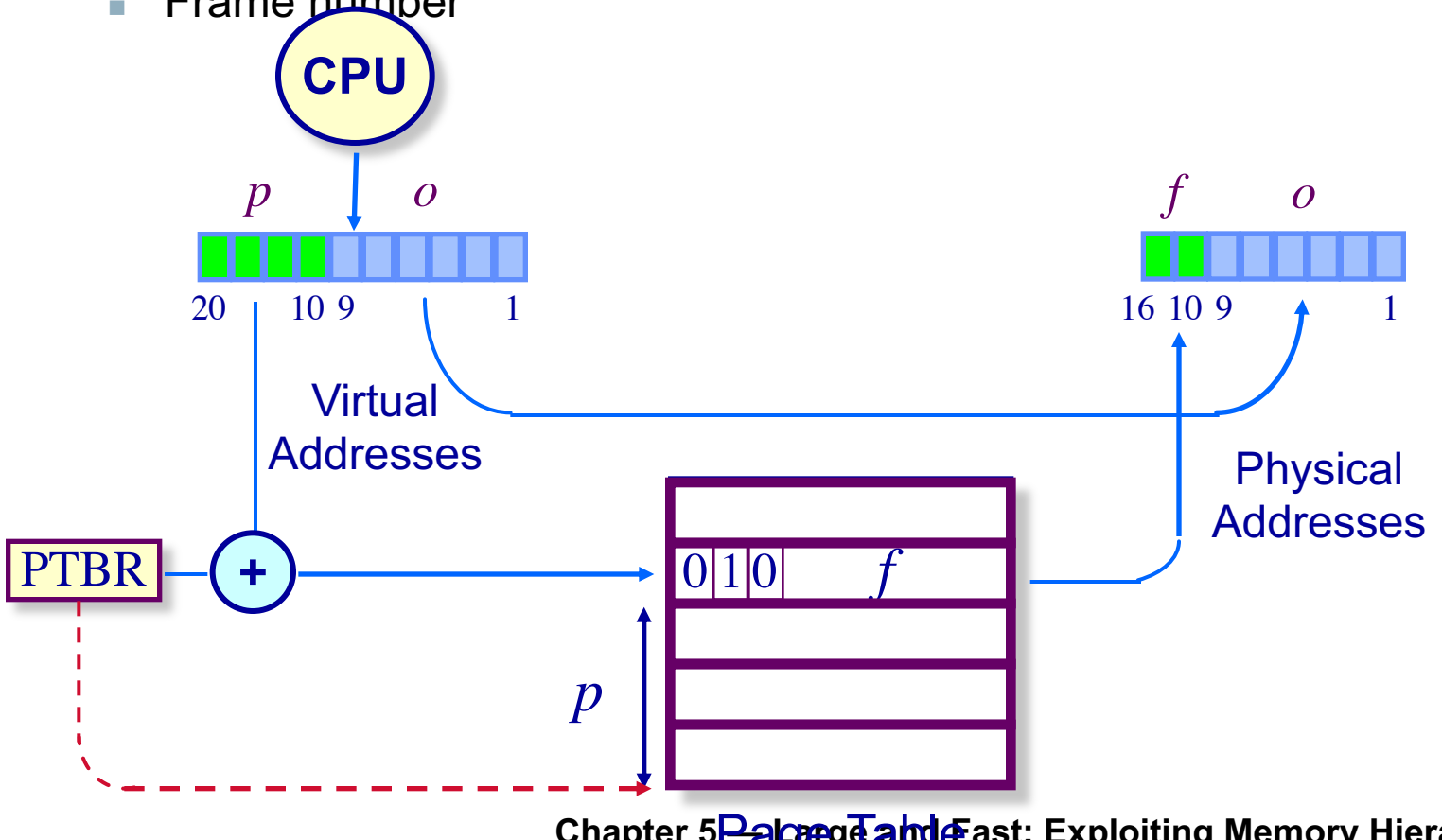
- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Page Table Structure

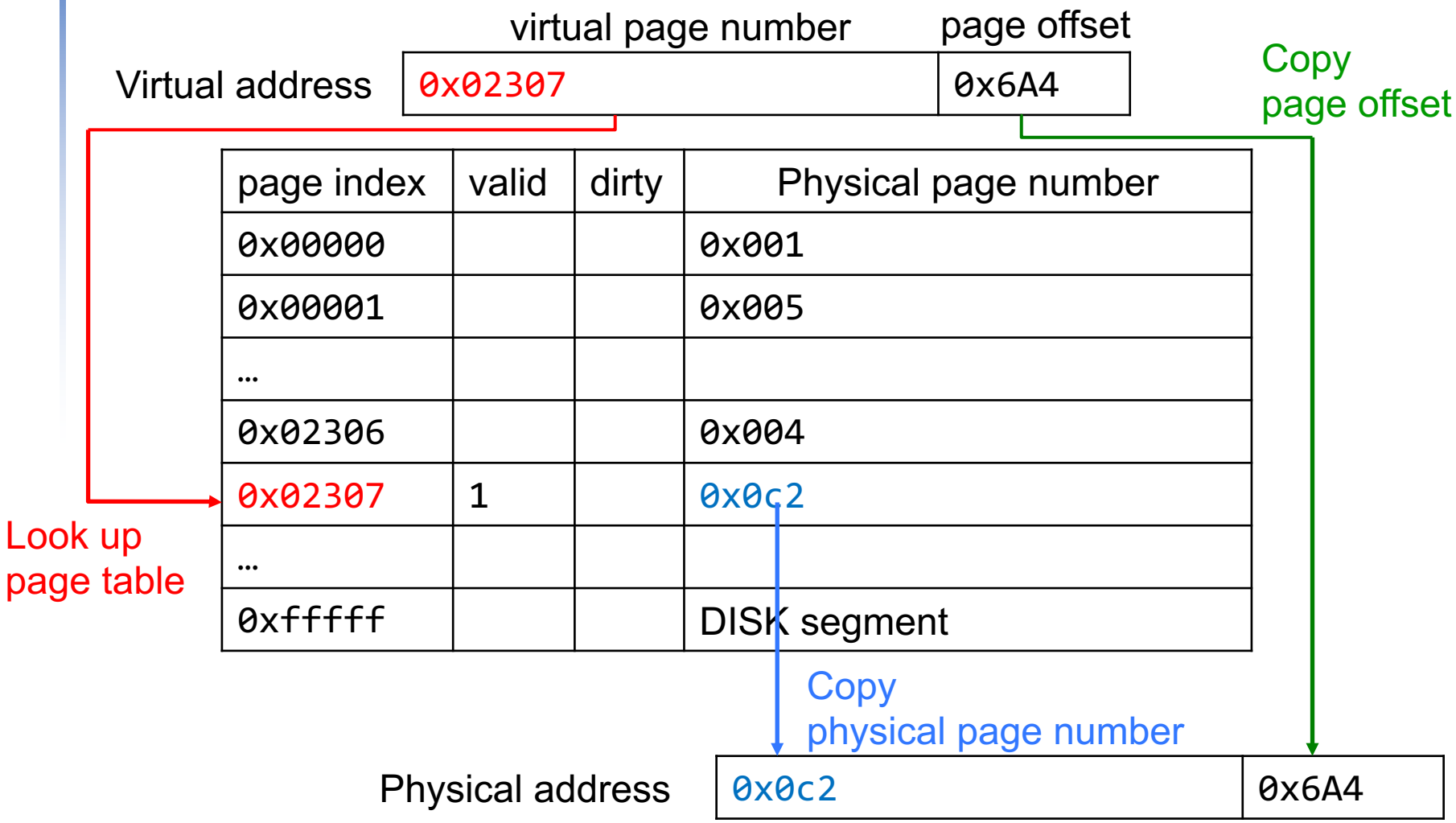
- 1 table per process
  - Access with PTBR (page table base register)
- Contain
  - Flags — dirty bit, valid bit, clock/reference bit
  - Frame number



Page Table Flags

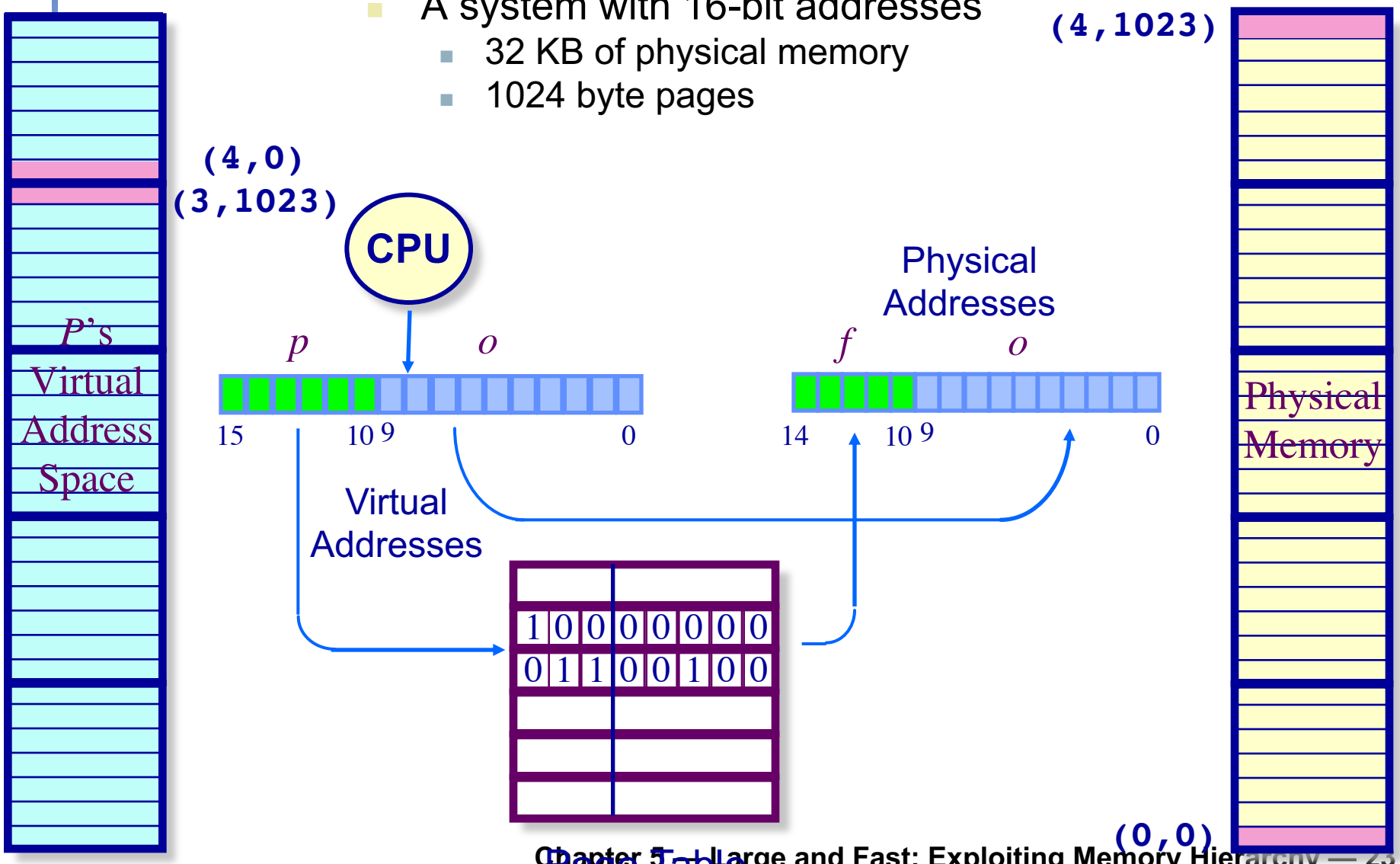
- Valid
  - The table entry has a valid frame number
- Dirty
  - The page has been written
- Reference
  - To identify the last visited page
  - For replacement usage

Translation Example



Another Translation Example

- A system with 16-bit addresses
  - 32 KB of physical memory
  - 1024 byte pages

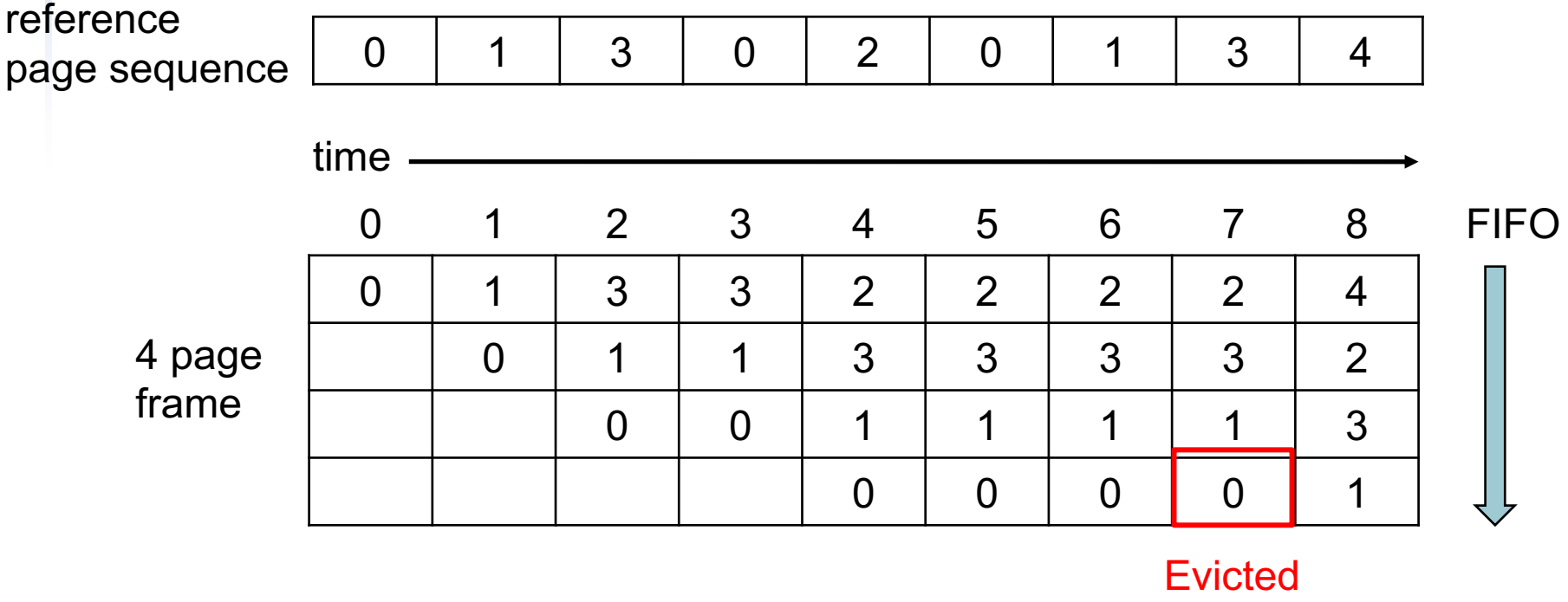


Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 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 PTE set when page is written

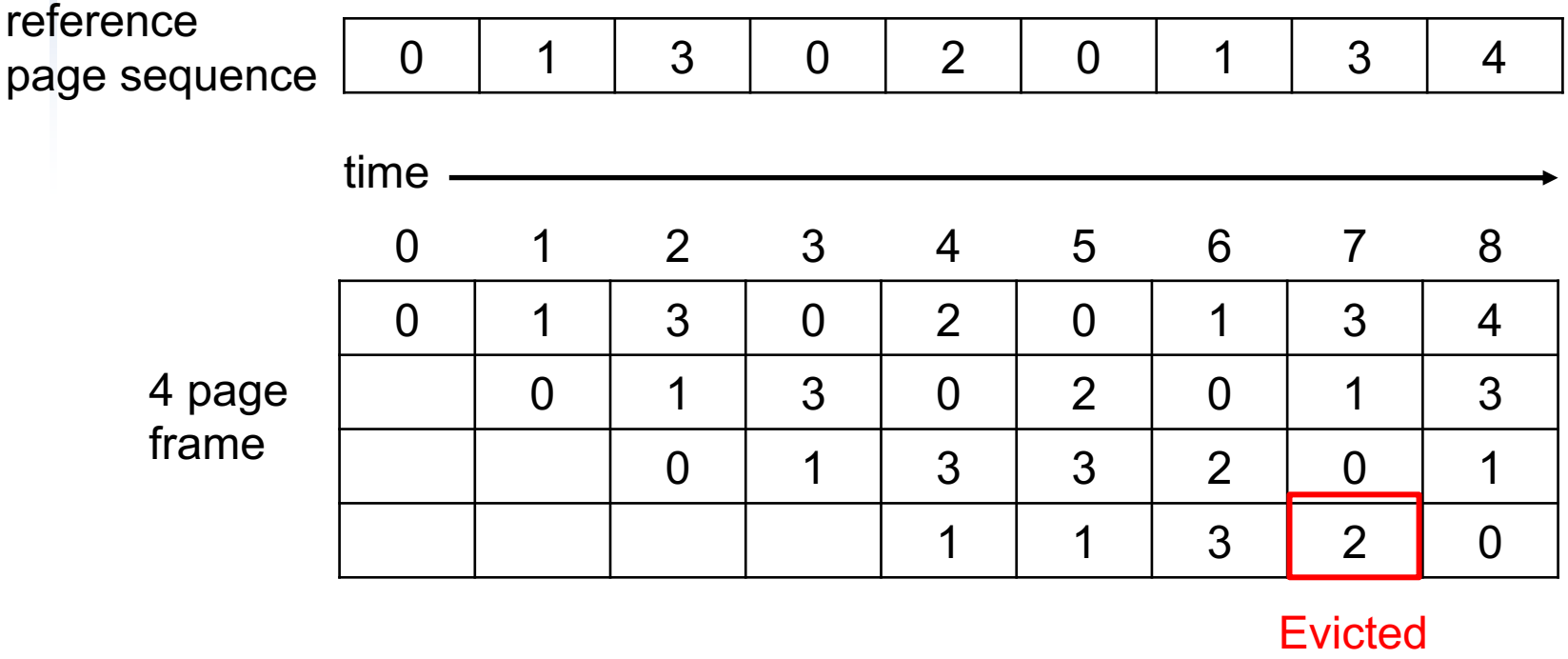
First-in First-out (FIFO) Replacement

- Evict oldest page
  - Defined by when a page is used.
  - The earlier, the older.



Least Recently Used (LRU) Replacement

- Evict page that has not been used for the longest time
  - When a page is referenced, it is placed at the end of a Queue



Size of Page Table

- Assume
  - 32-bit virtual address
  - 4k page size → 12 bit page offset
  - 20 bits for virtual page number
- Assume 4bytes per page entry, we need $2^{20}*4$ ~ **4Mbytes** page table size
- We need one table for each process.

Where is the page table?

- Page table is too big for CPU
  - Actually most of it is empty
- It is too slow to translate with OS and main memory

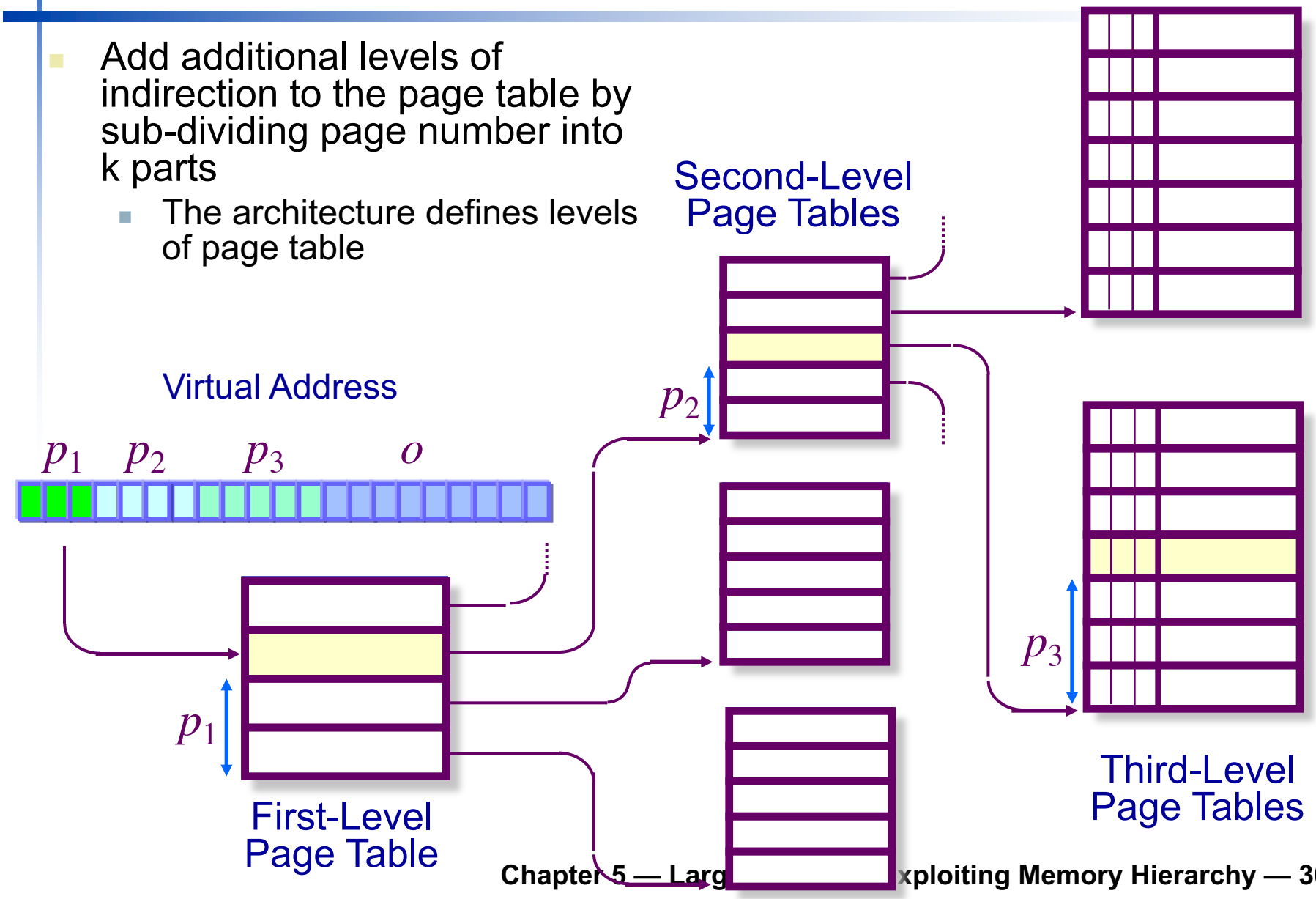
- Solution: Put page table in memory and cache part of it in CPU
- Translation Lookaside Buffer (TLB)

How to compress page table?

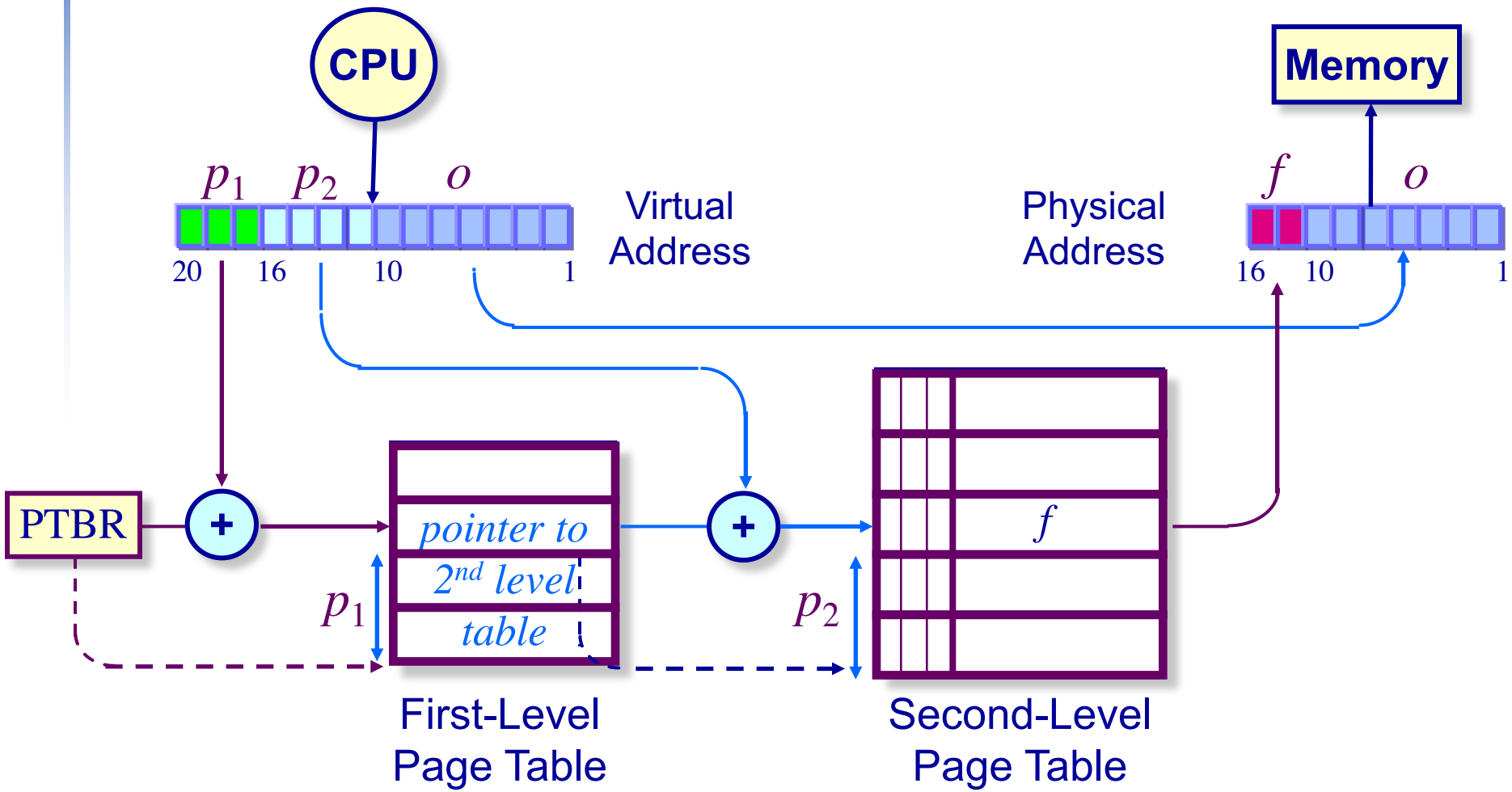
- Inverted page table
  - Store only mapped PA frames
  - Store both VA and PA, and use hash table to search
- Multi-level page table
  - Divide virtual page number into blocks
  - Each block index a table
  - Can handle stack and heap memory

Multi-level Paging

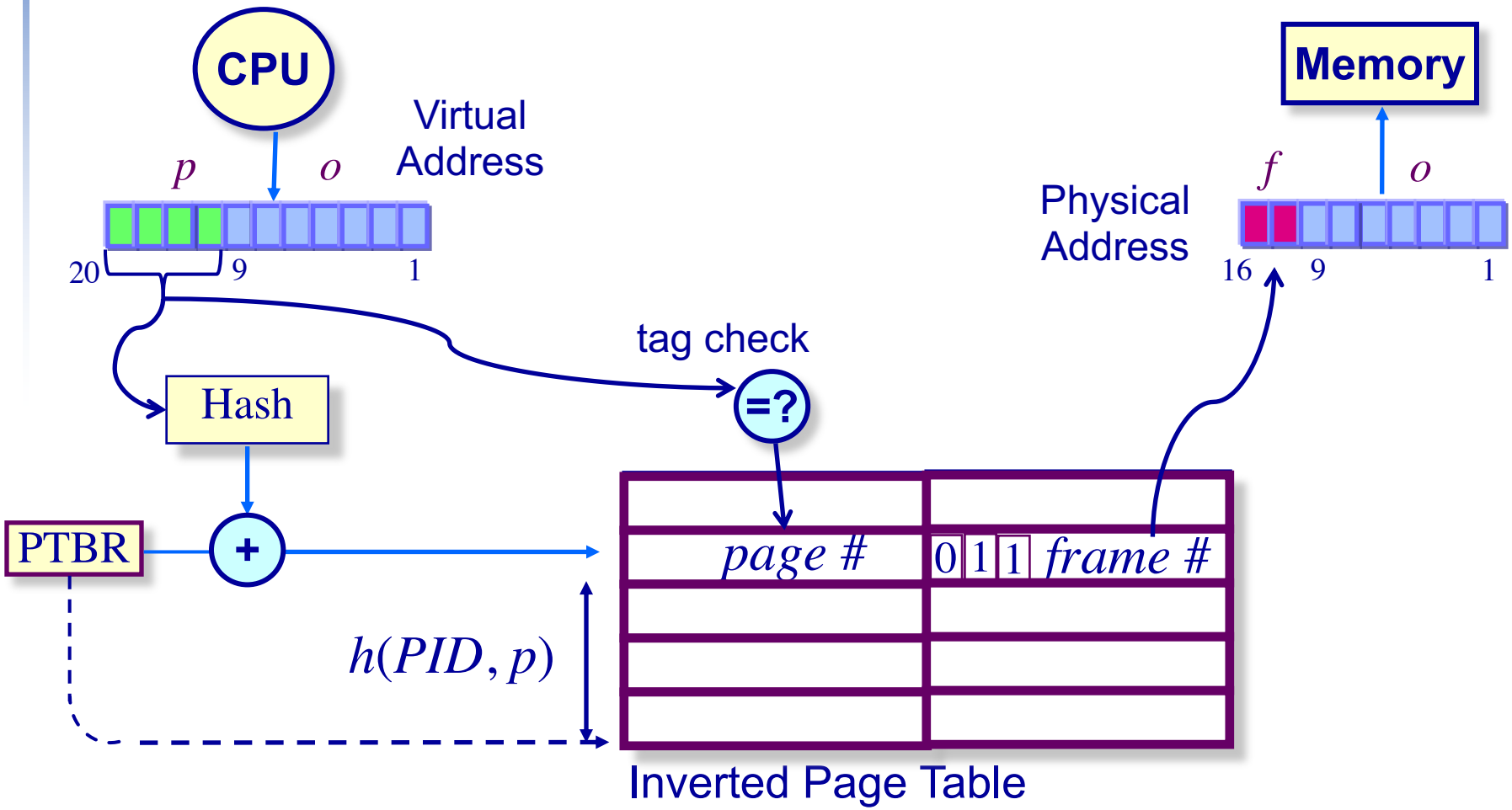
- Add additional levels of indirection to the page table by sub-dividing page number into k parts
  - The architecture defines levels of page table



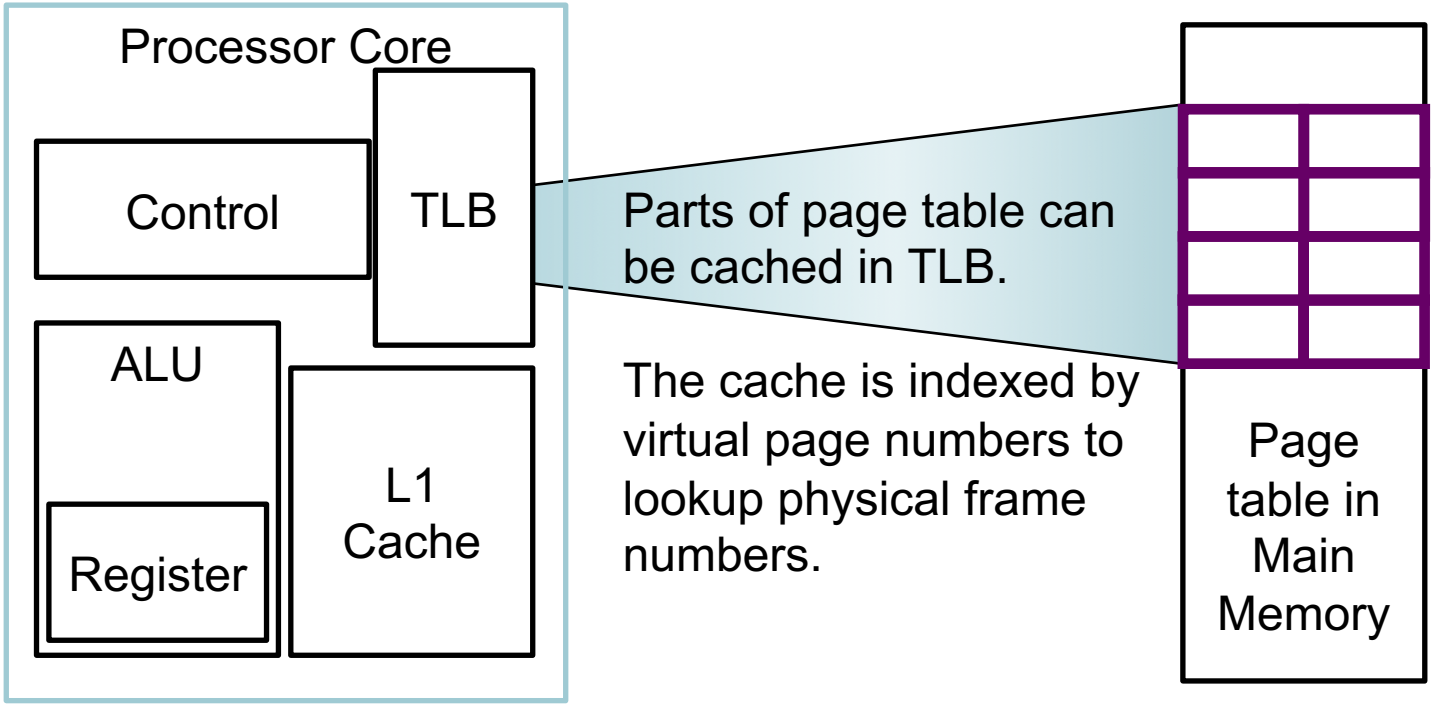
Example: Two-level Paging



Inverted Hashed Page Tables



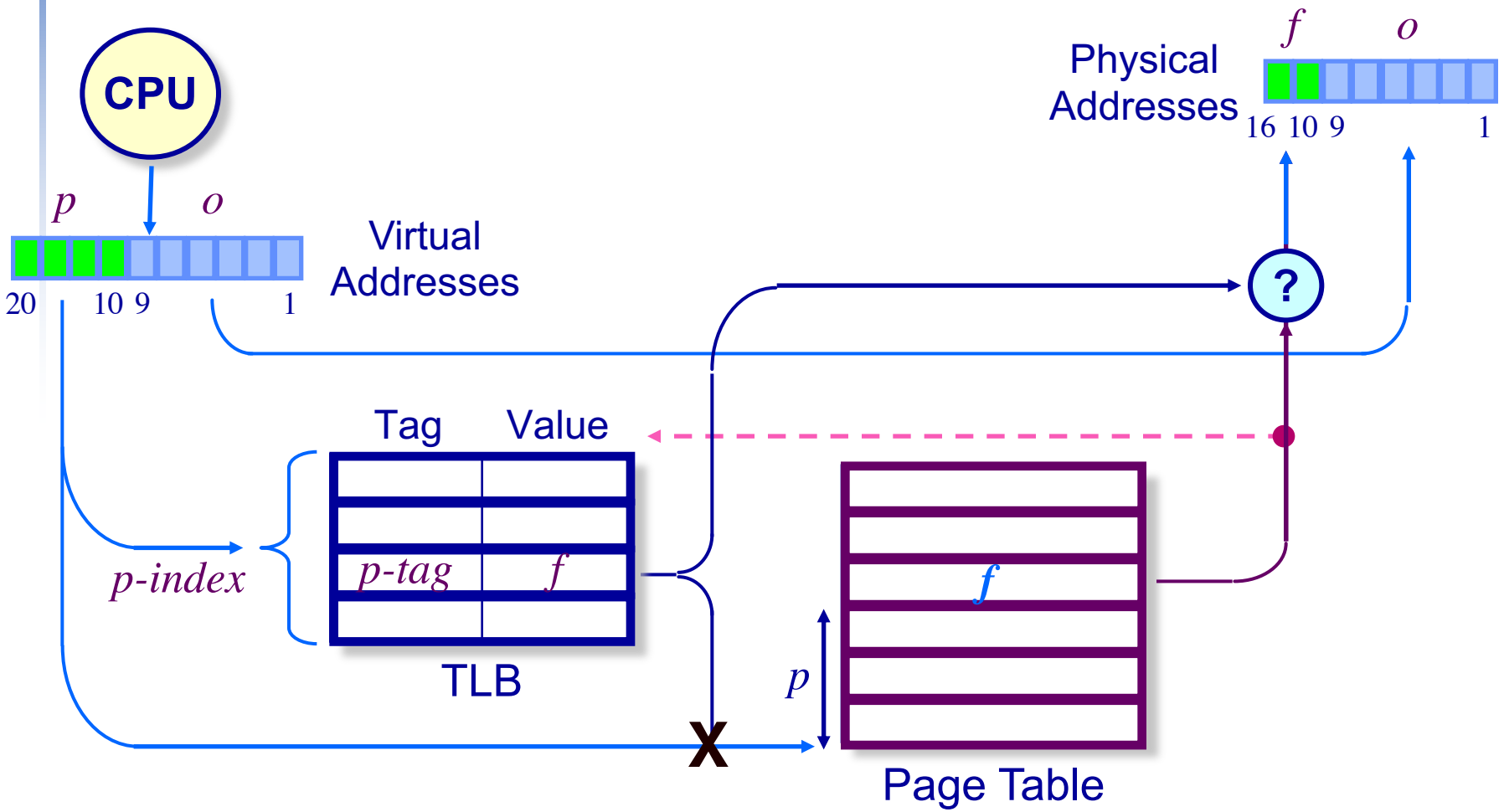
Speedup Page Lookup



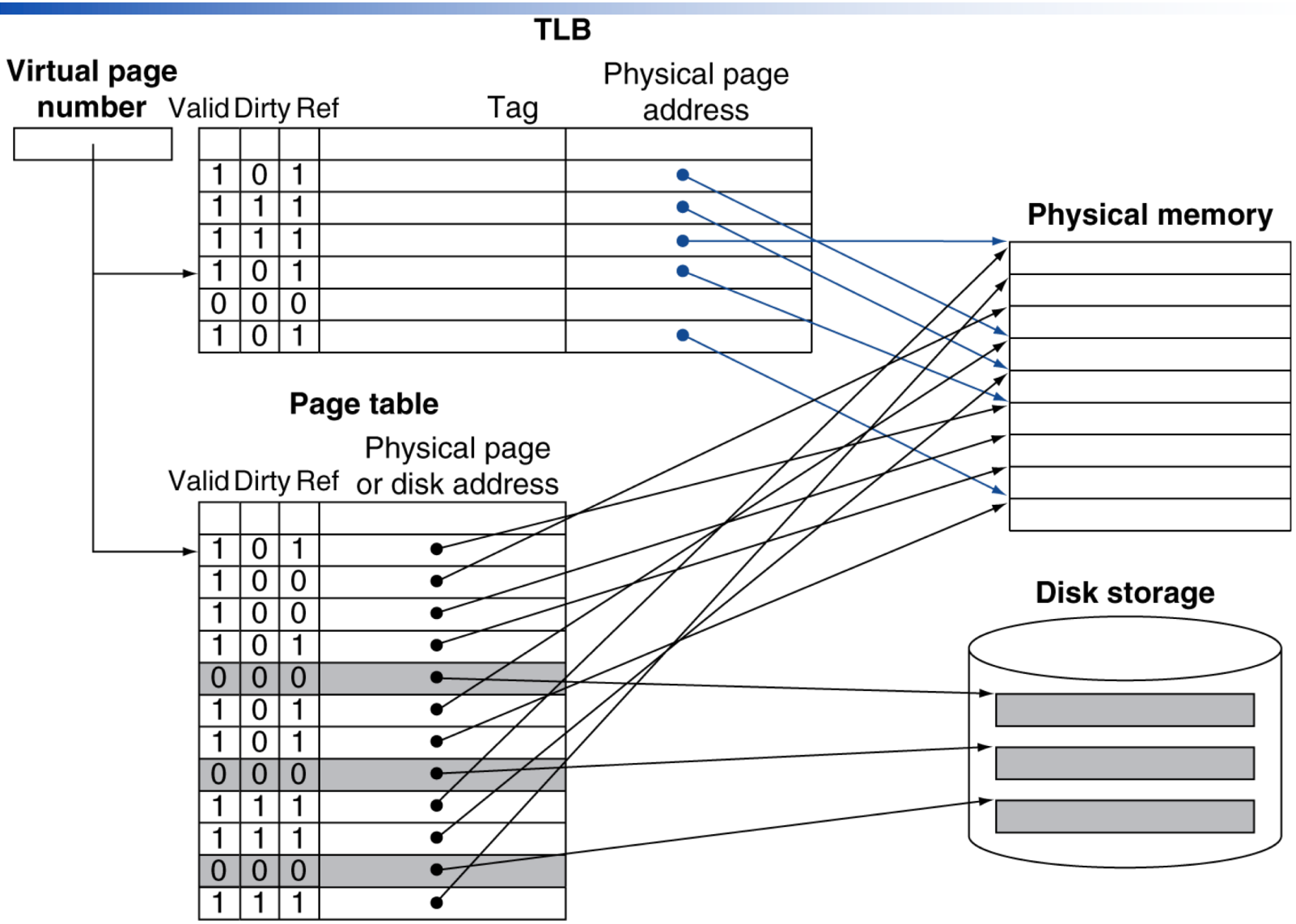
Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good 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
  - Misses could be handled by hardware or software

Use TLB to Speedup Page Lookup



Fast Translation Using a TLB



TLB Misses

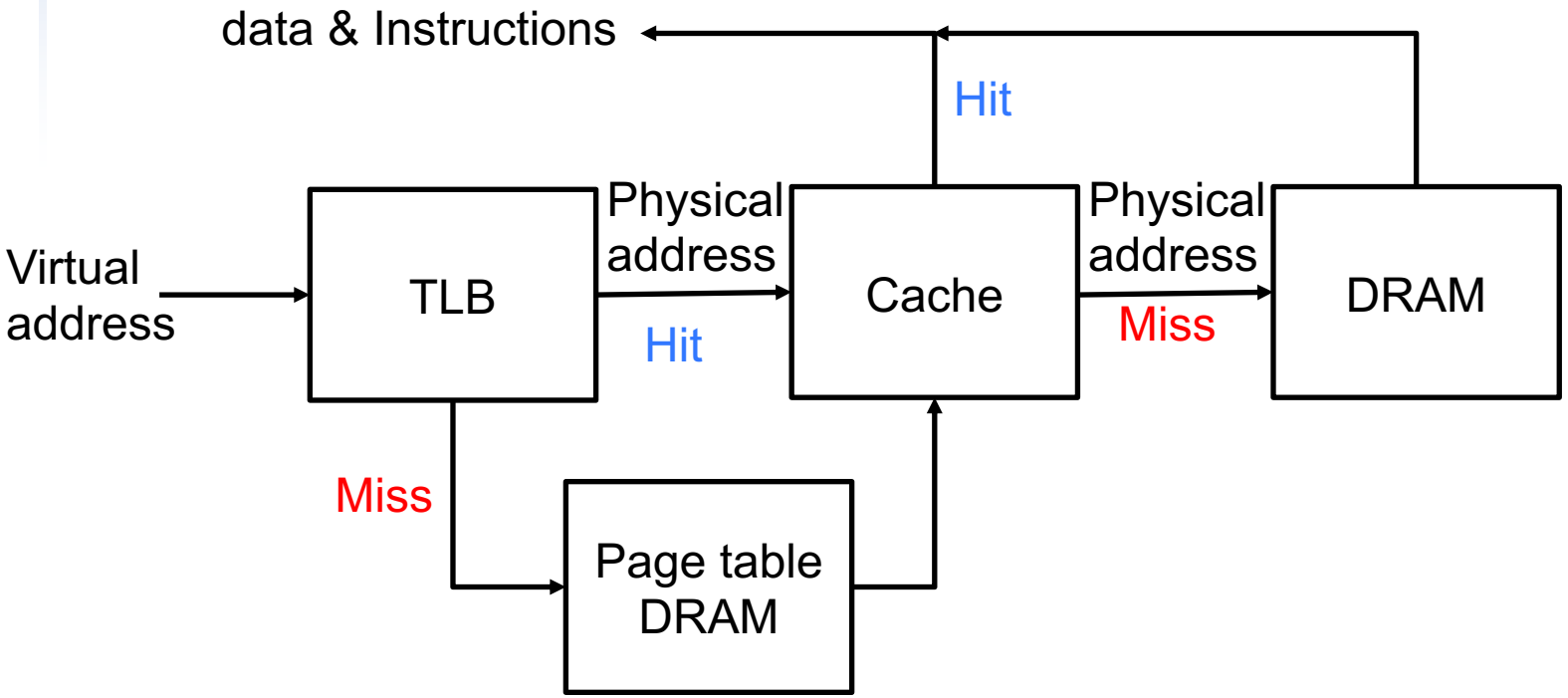
- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

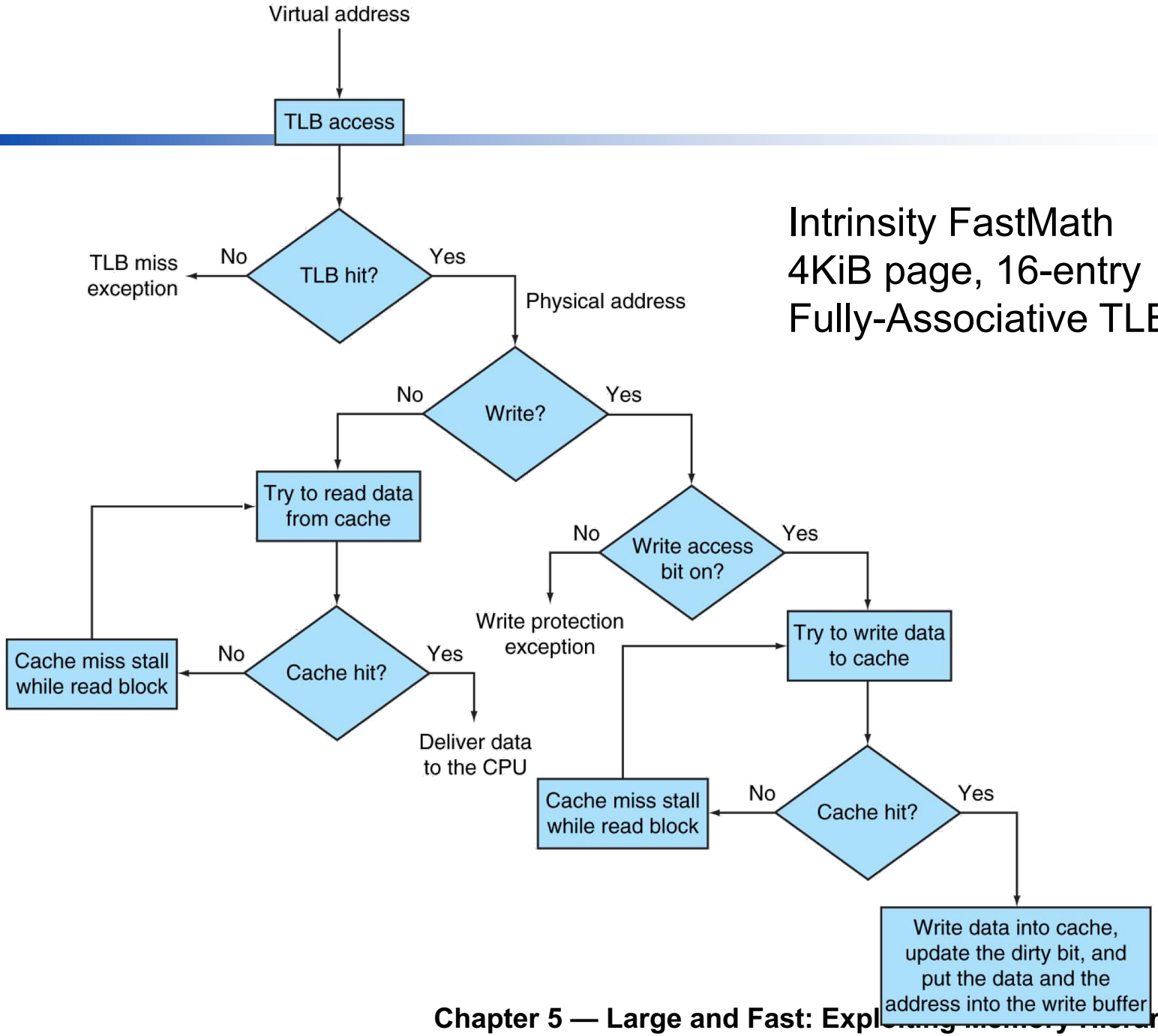
TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not preset
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

TLB and Cache

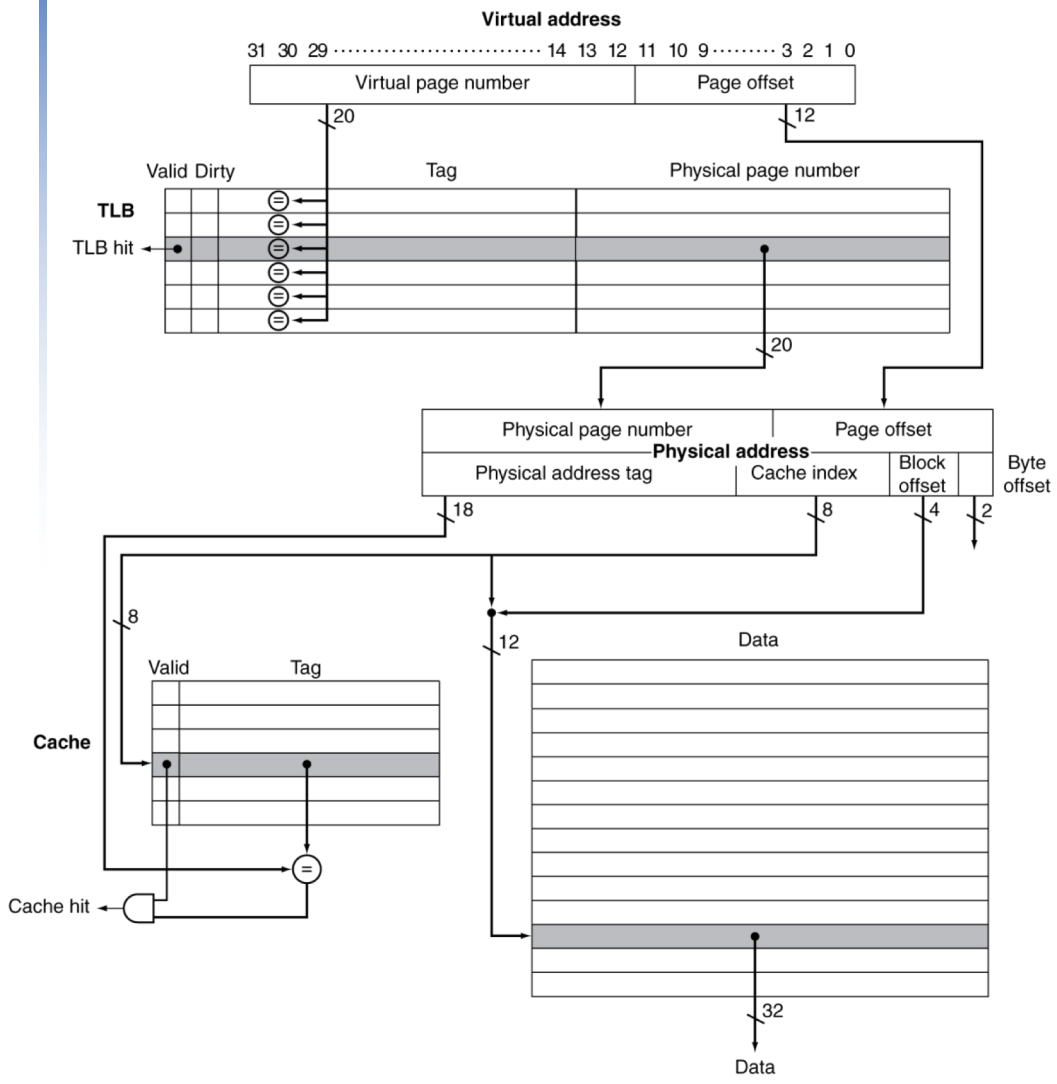
- If we lookup a physical address in cache, we will have to wait for TLB/OS/page fault, before we get actual instruction or data.





Intrinsity FastMath
4KiB page, 16-entry
Fully-Associative TLB

TLB and Cache Interaction



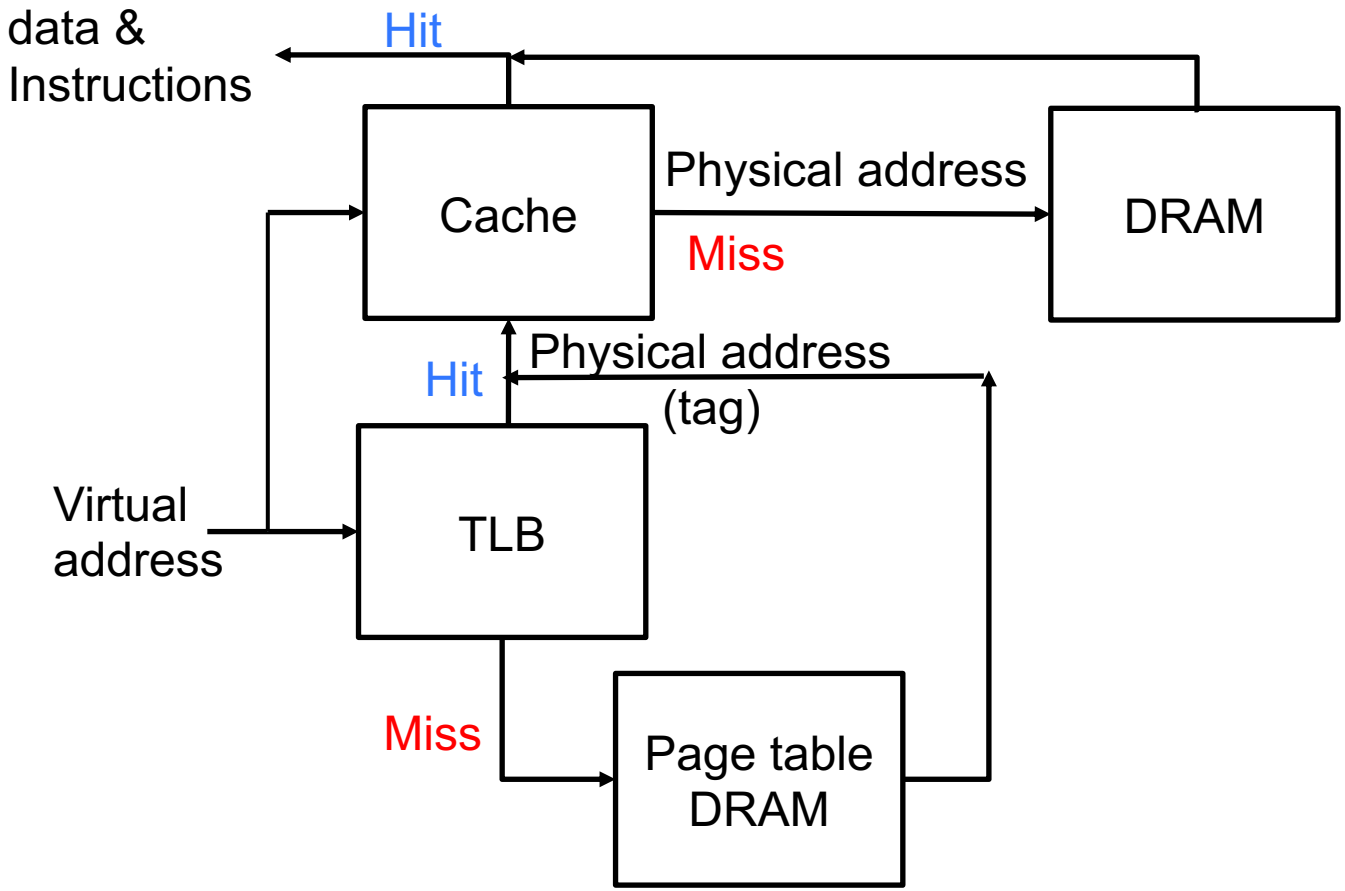
- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Cache Index/Tag Options

- Physically indexed, physically tagged (PIPT)
  - Wait for full address translation
  - Then use physical address for both indexing and tag comparison
- Virtually indexed, physically tagged (VIPT)
  - Use portion of the virtual address for indexing then wait for address translation and use physical address for tag comparisons
  - Easiest when index portion of virtual address w/in offset (page size) address bits, otherwise aliasing may occur
- Virtually indexed, virtually tagged (VIVT)
  - Use virtual address for both indexing and tagging…No TLB access unless cache miss
  - Requires invalidation of cache lines on context switch or use of process ID as part of tags

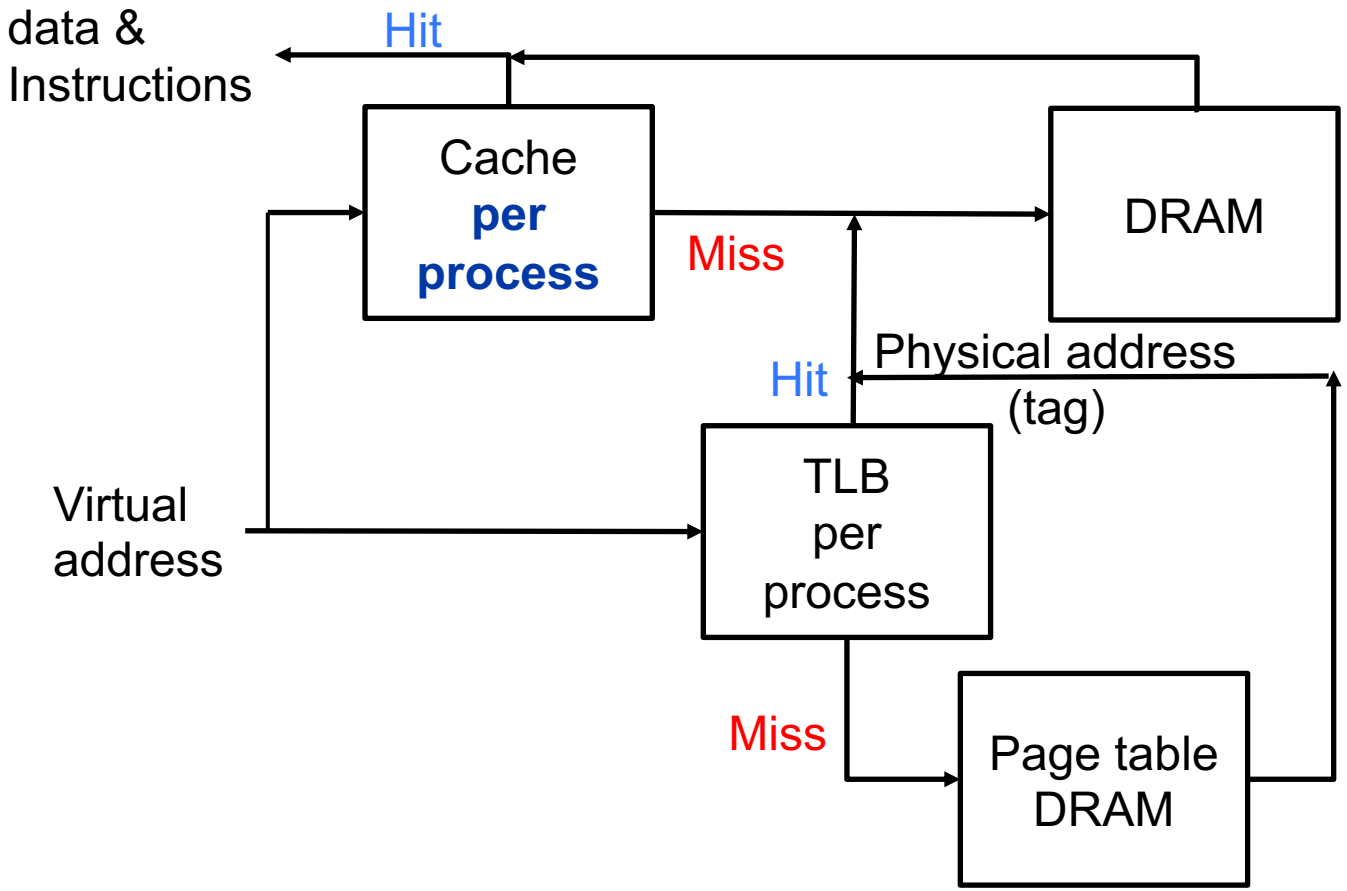
VIPT Flow

- Lookup a virtual address in cache and match with physical tag (from TLB).
- Intel



VIVT Flow

- Lookup a virtual address in cache and match with virtual tag per process.
- Either too much hardware or too much time for context switching



Multi-programming and Protection

- Machines can run multiple processes
  - Share a single or multiple cores
  - Share the same physical memory space

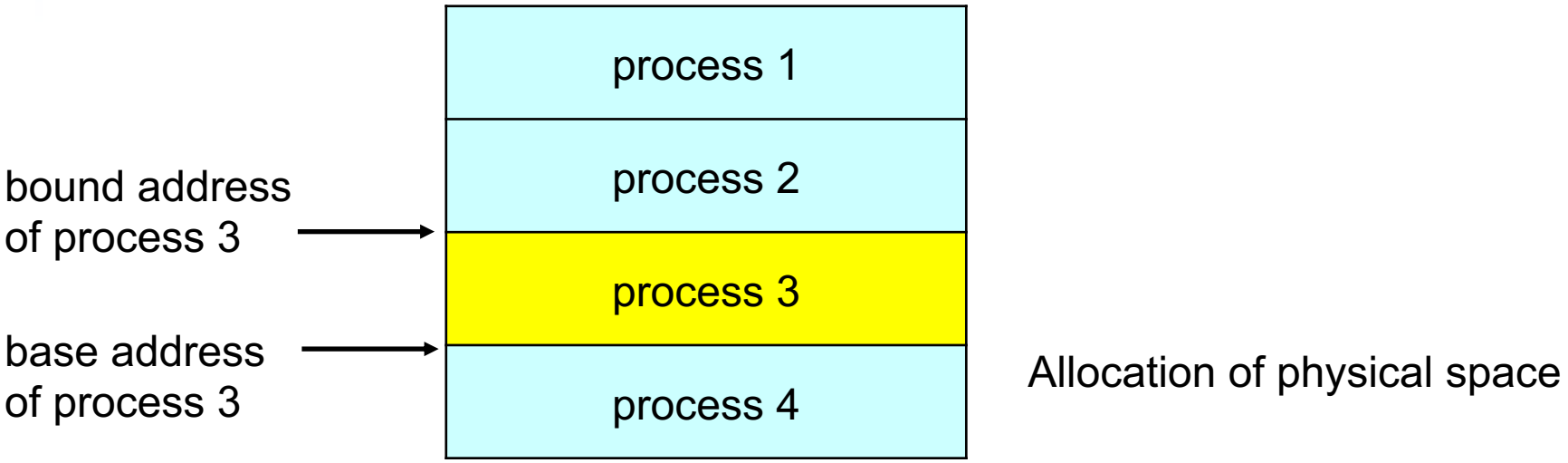
- OS coordinates the resource sharing and protection
  - To avoid one process read or write to other's data/instructions

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

Protection with bounds

- Each process is limited between base and bound addresses
  - Checked by two registers
- Only OS can update these registers
  - in privileged mode and instructions

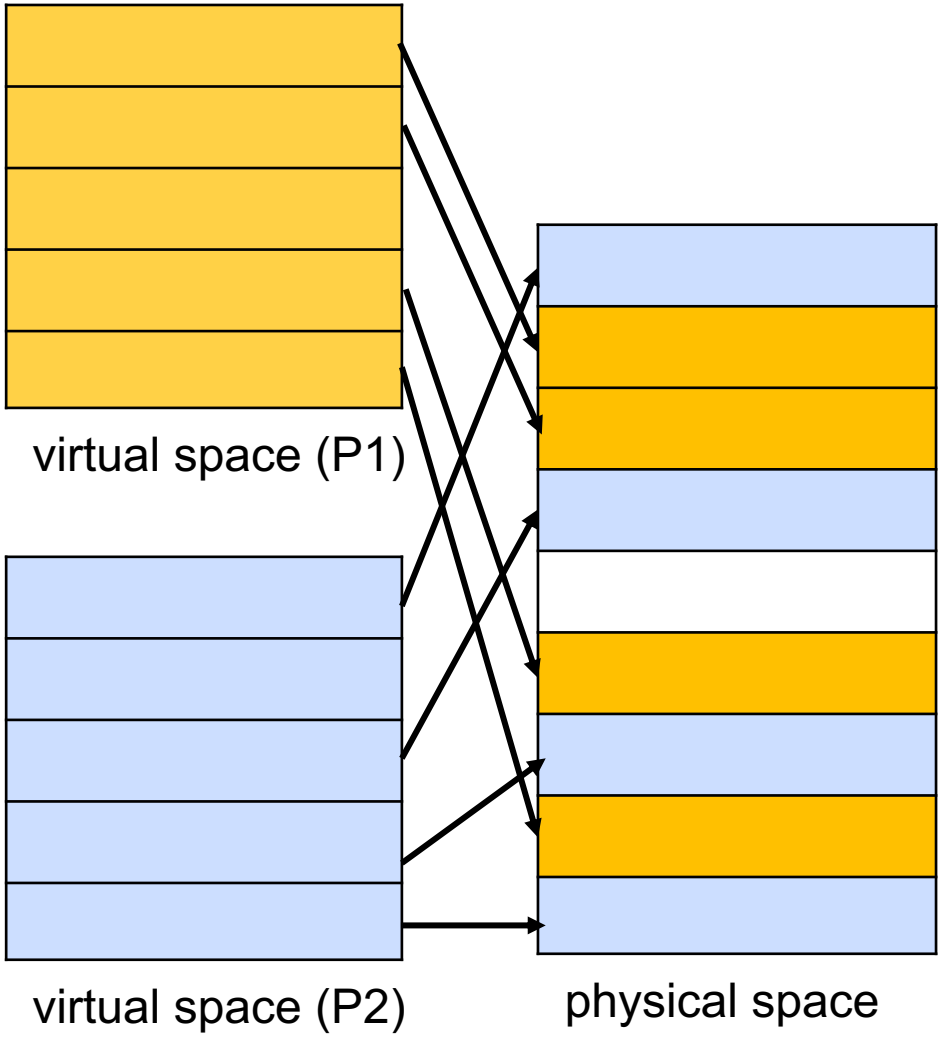


Protection with page tables

- Each process has one page table
- Each process can only access physical addresses in the table
- Page tables are in OS kernel space
  - only OS can change them

- TLB to support protection and avoid virtual address aliasing
  - Add process ID to TLB
  - Or flush TLB on context switch

Different Page Table for Each Process



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 caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

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 disk write latency

Sources of Misses

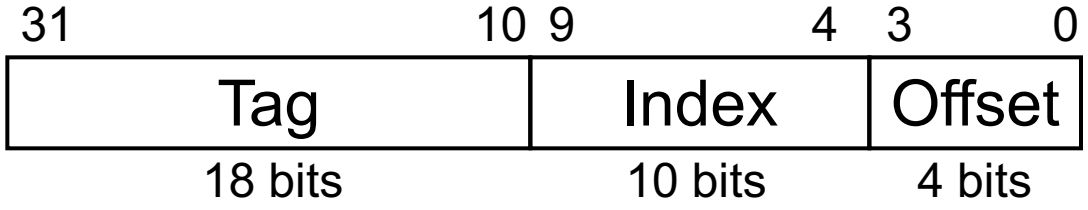
- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur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

Cache Design Trade-offs

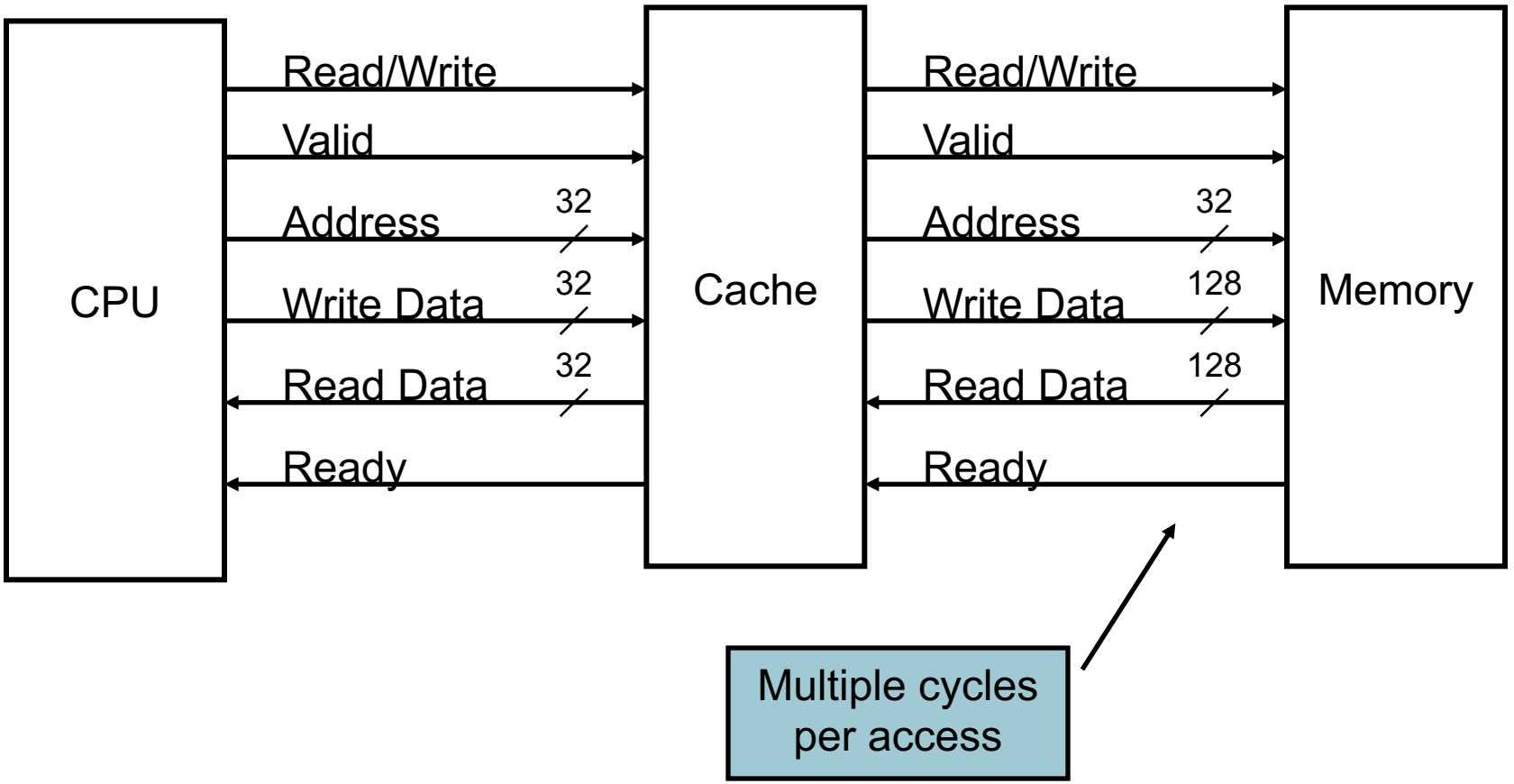
Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size,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

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

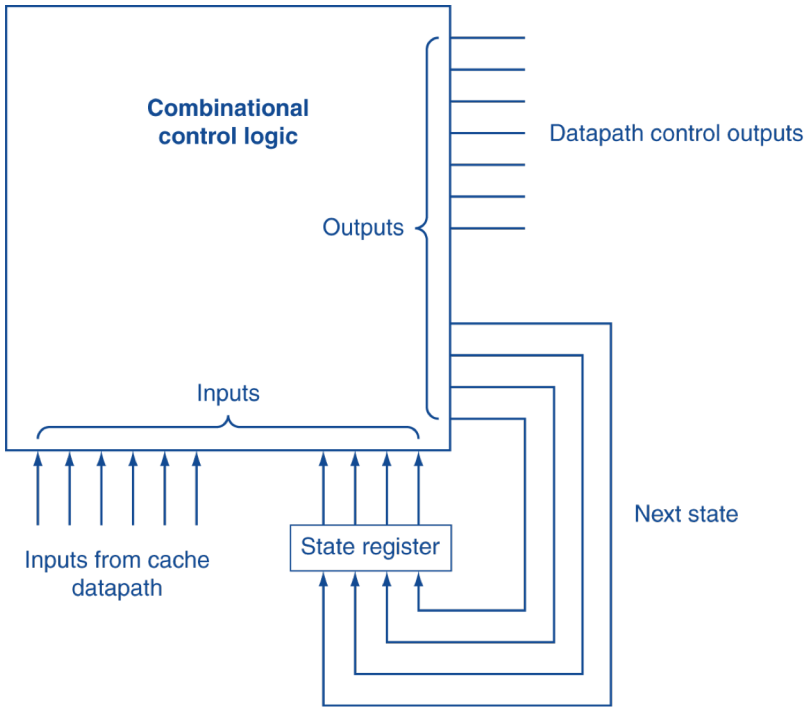


Interface Signals

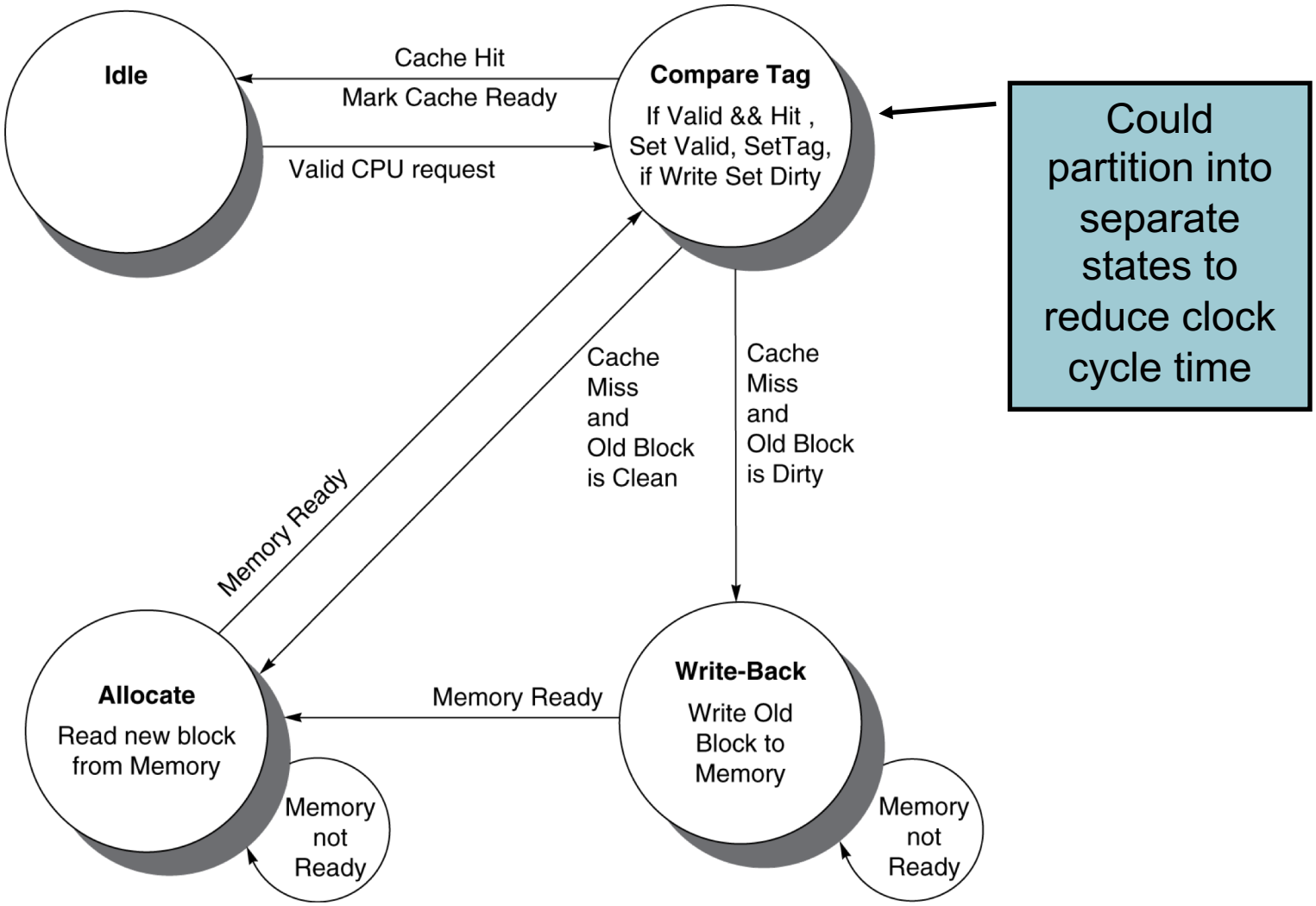


Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state = f_n (current state, current inputs)
- Control output signals = f_o (current state)



Cache Controller FSM



Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- When are writes seen by other processors
  - “Seen” means a read returns the written value
  - Can’t be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    ⇒ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

2-Level TLB Organization

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Supporting Multiple Issue

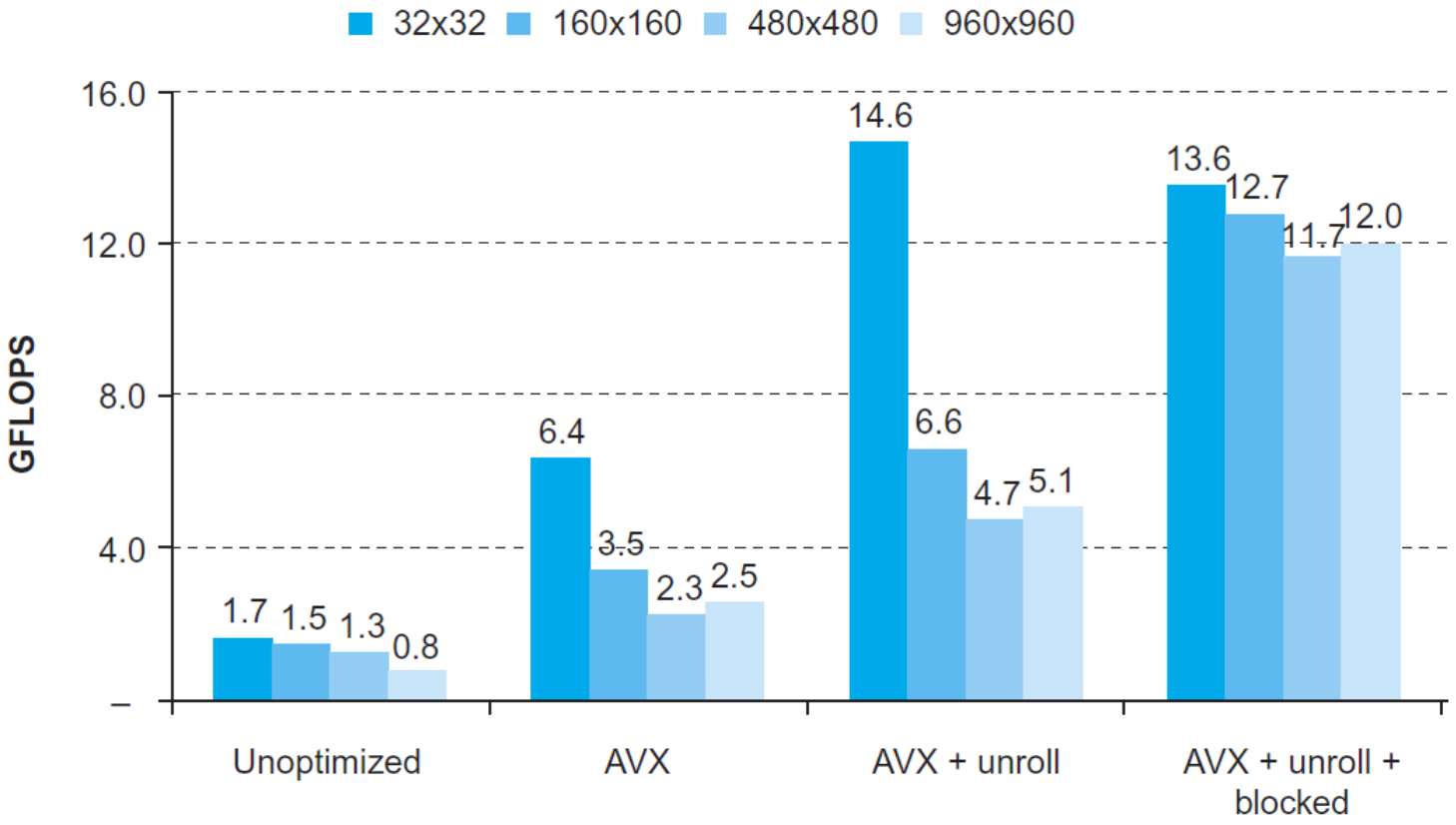
- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts
- Core i7 cache optimizations
  - Return requested word first
  - Non-blocking cache
    - Hit under miss
    - Miss under miss
  - Data prefetching

RISC-V System Instructions

Type	Mnemonic	Name
Mem ordering	fence.i	Instruction fence
	fence	Fence
	sfence.vm	Address translation fence
CSR access	csrrwi	CSR read/write immediate
	csrrsi	CSR read/set immediate
	csrrci	CSR read/clear immediate
	csrrw	CSR read/write
	csrrs	CSR read/set
	csrrc	CSR read/clear
System	ecall	Environment call
	ebreak	Environment breakpoint
	sret	Supervisor exception return
	wfi	Wait for interrupt

DGEMM

- Combine cache blocking and subword parallelism



Pitfalls

- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- Ignoring memory system effects when writing or generating code
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated
- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

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
- Memory system design is critical for multiprocessors