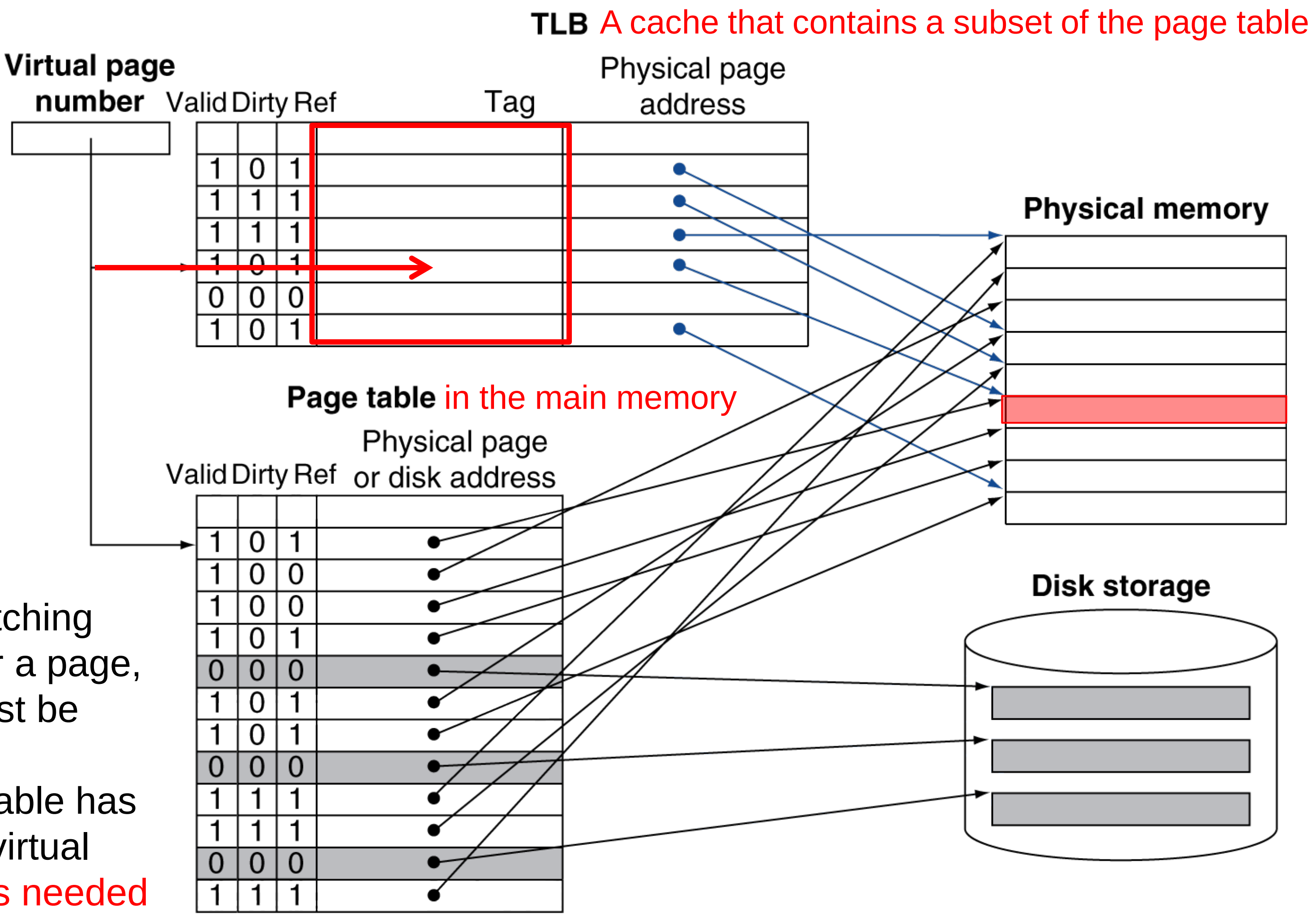
Chapter 5C

Large and Fast: Exploiting Memory Hierarchy

Two kinds of TLB Misses

- Check the valid bit of a page table entry
  - If it is on, the page is in memory (a simple TLB miss).
  - Otherwise, the page is not in memory (a page fault)
- A simple TLB miss (refill TLB)
  - Load the PTE into TLB and retry → TLB hit
  - What if the page table is not in memory?
    - A page fault for the page table
  - Could be handled in hardware or in software
    - Little performance difference between the two approaches.
- A page fault (for instruction or data)
  - OS handles fetching the corresponding page from the disc and updating the page table
  - Then restart the faulting instruction

A TLB Hit



◆If there is no matching entry in the TLB for a page, the page table must be examined.

◆Since the page table has an entry for every virtual page, no tag field is needed

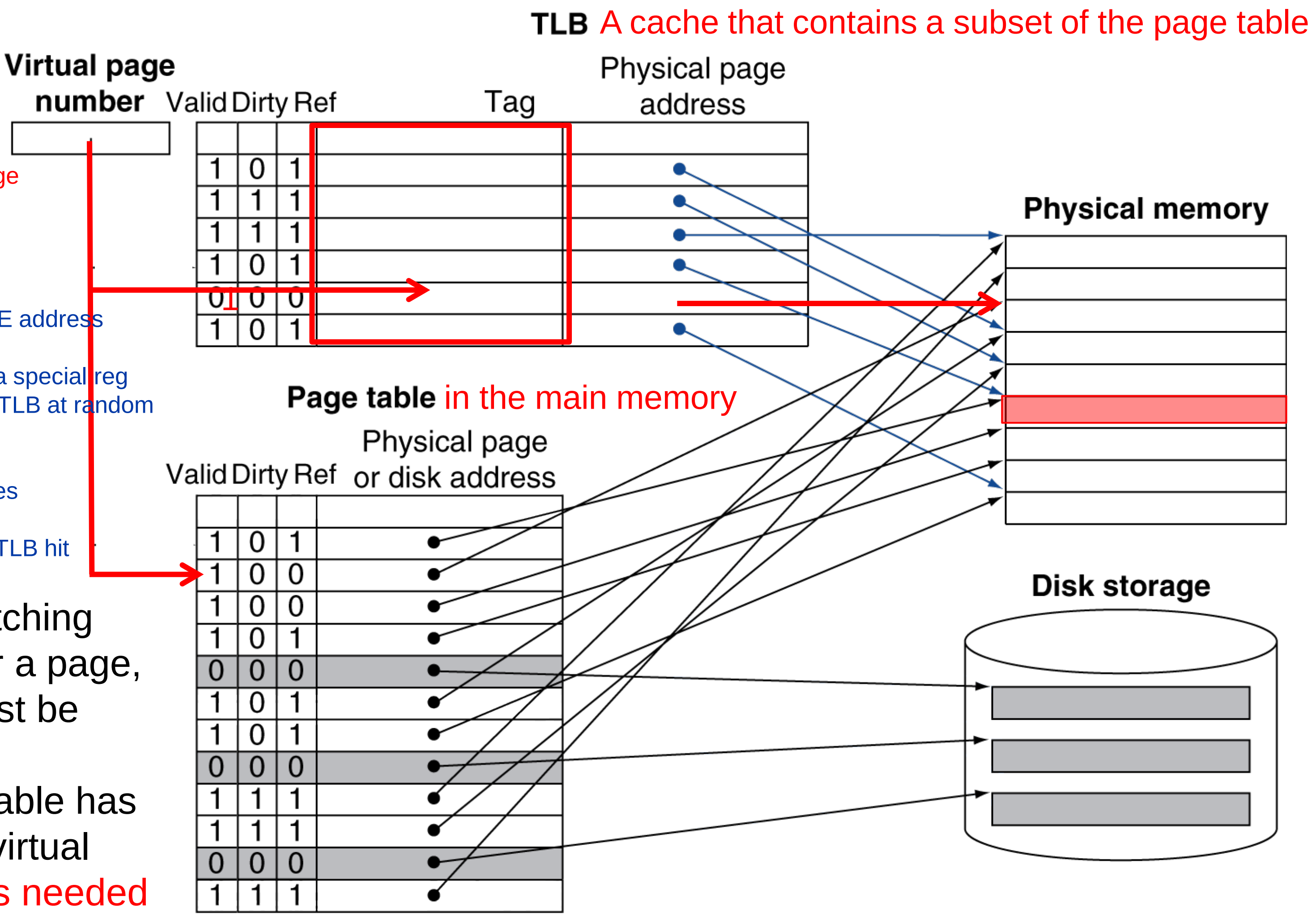
A simple TLB Miss

1. TLB miss handler copy the page table entry without checking its valid bit.

```
TLBmiss:
 mfc0   $k1, Context    # copy PTE address
 lw     $k1, 0($k1)     # get PTE
 mtc0   $k1, EtnryLo    # put into a special reg
 tlbwr                  # put into TLB at random
 eret                   # return
```

It takes about a dozen clock cycles

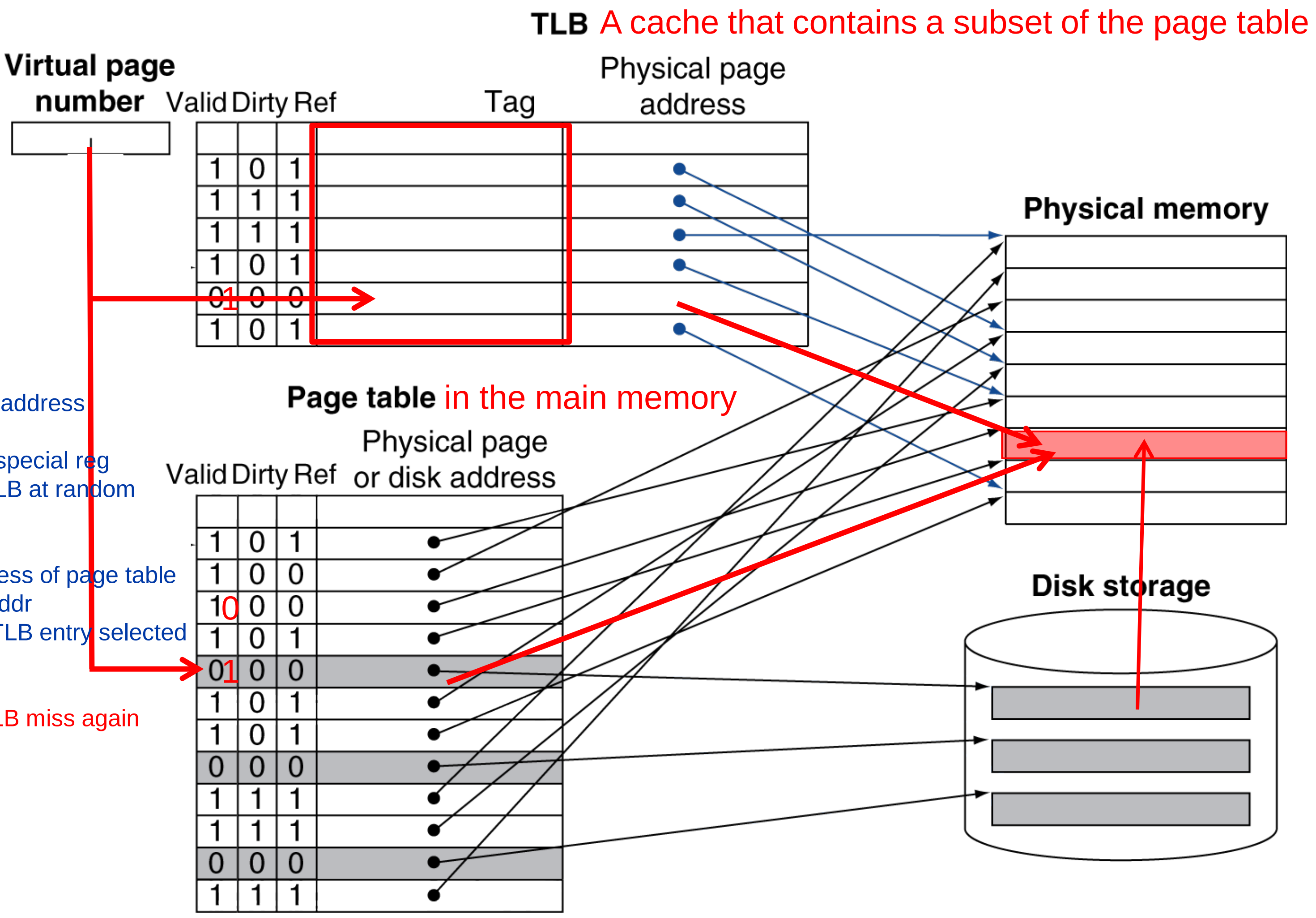
2. re-try the address translation: TLB hit



◆If there is no matching entry in the TLB for a page, the page table must be examined.

◆Since the page table has an entry for every virtual page, no tag field is needed

Page fault



1. TLB miss handler copy the page table entry without checking its valid bit.

```
TLBmiss:
 mfc0   $k1, Context    # copy PTE address
 lw     $k1, 0($k1)     # get PTE
 mtc0   $k1, EtnryLo    # put into a special reg
 tlbwr                  # put into TLB at random
 eret                   # return
```

Context: upper 12 bits: bases address of page table
lower 18 bits from BADVaddr
Tlbwr: copy from EntryLo into the TLB entry selected by Random

2. re-try the address translation: TLB miss again

Exception Handler

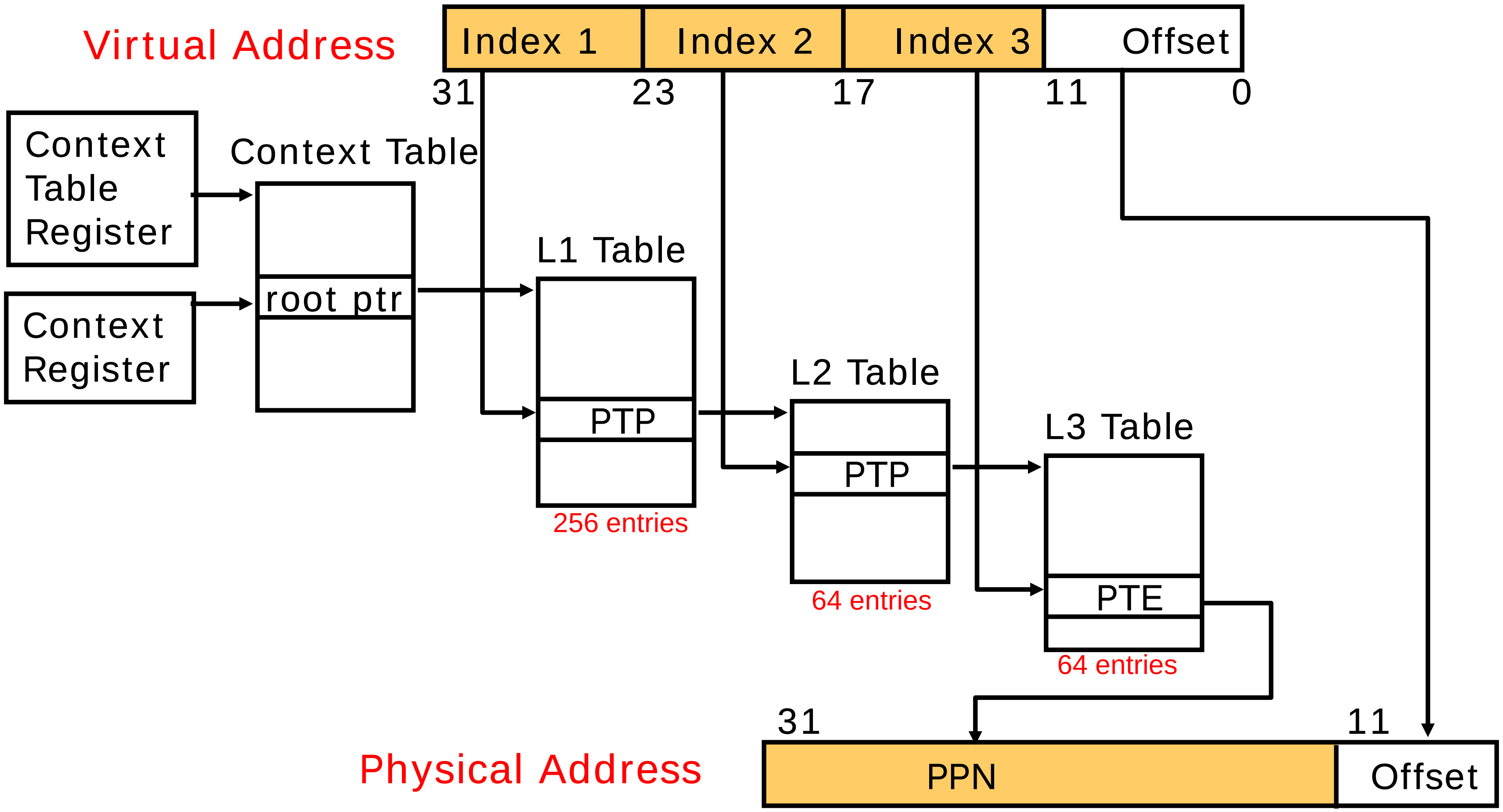
- Handling a TLB miss or a page fault requires using the exception mechanism
  - Interrupt the active process
  - Transfer control to the OS
  - (exception handling)
  - Resume execution of the interrupted process
- The OS is particularly vulnerable
  - between the time we begin executing the exception handler and the time that OS has saved all the state of the process because another exception can occur.
  - set the supervisor mode and disable the exceptions.
  - After the OS saves just enough state to allow it to recover, then re-enable the exceptions..

Handling a TLB Miss

- Software (MIPS, Alpha)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB.
  - A privileged “untranslated” (unmapped) addressing mode used for page table walk
  - Unmapped: a portion of the address space that cannot have page fault.
- Hardware (SPARC v8, x86, PowerPC)
  - A memory management unit (MMU) walks the page tables and reloads the TLB

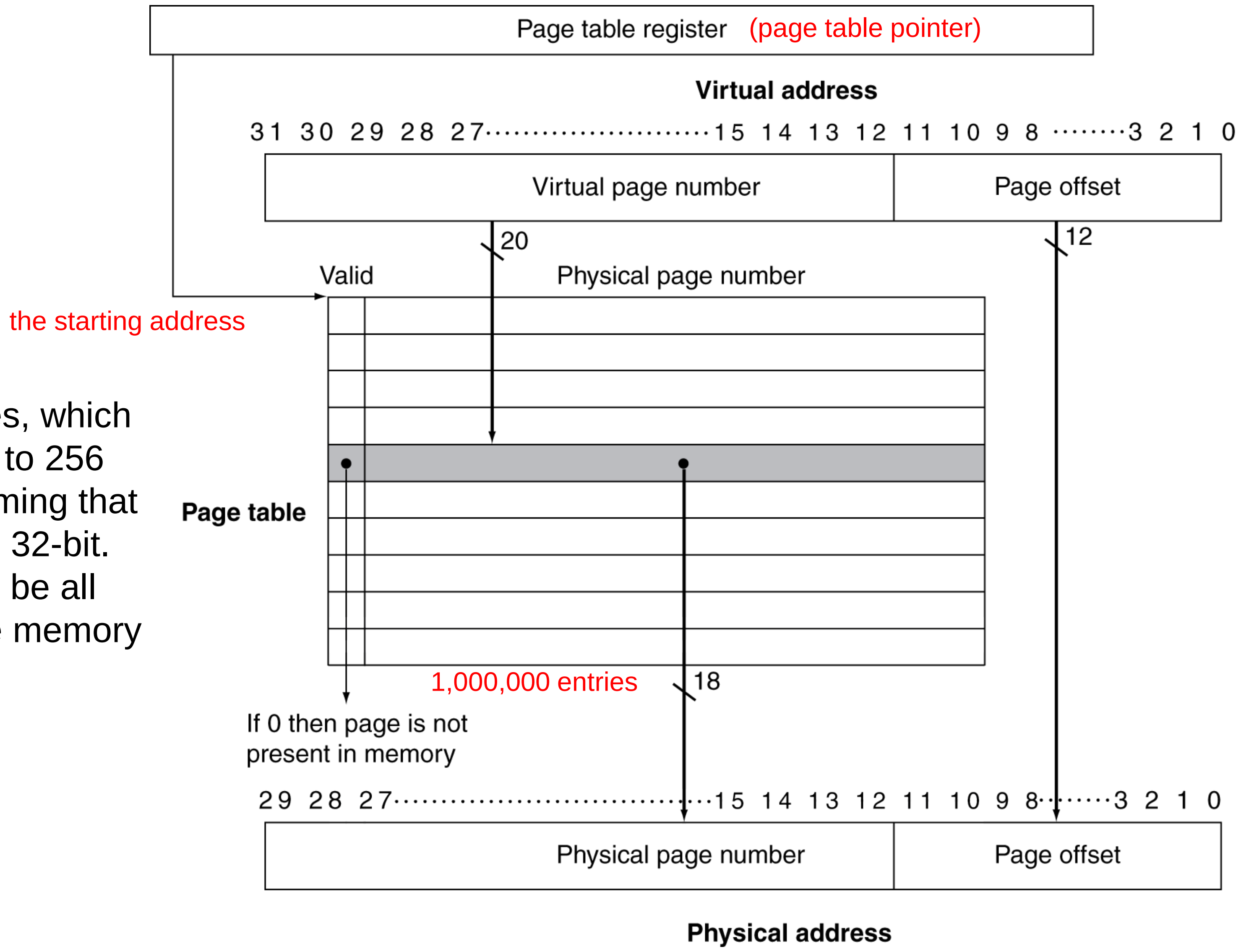
  If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Page Table Walk: SPARC v8



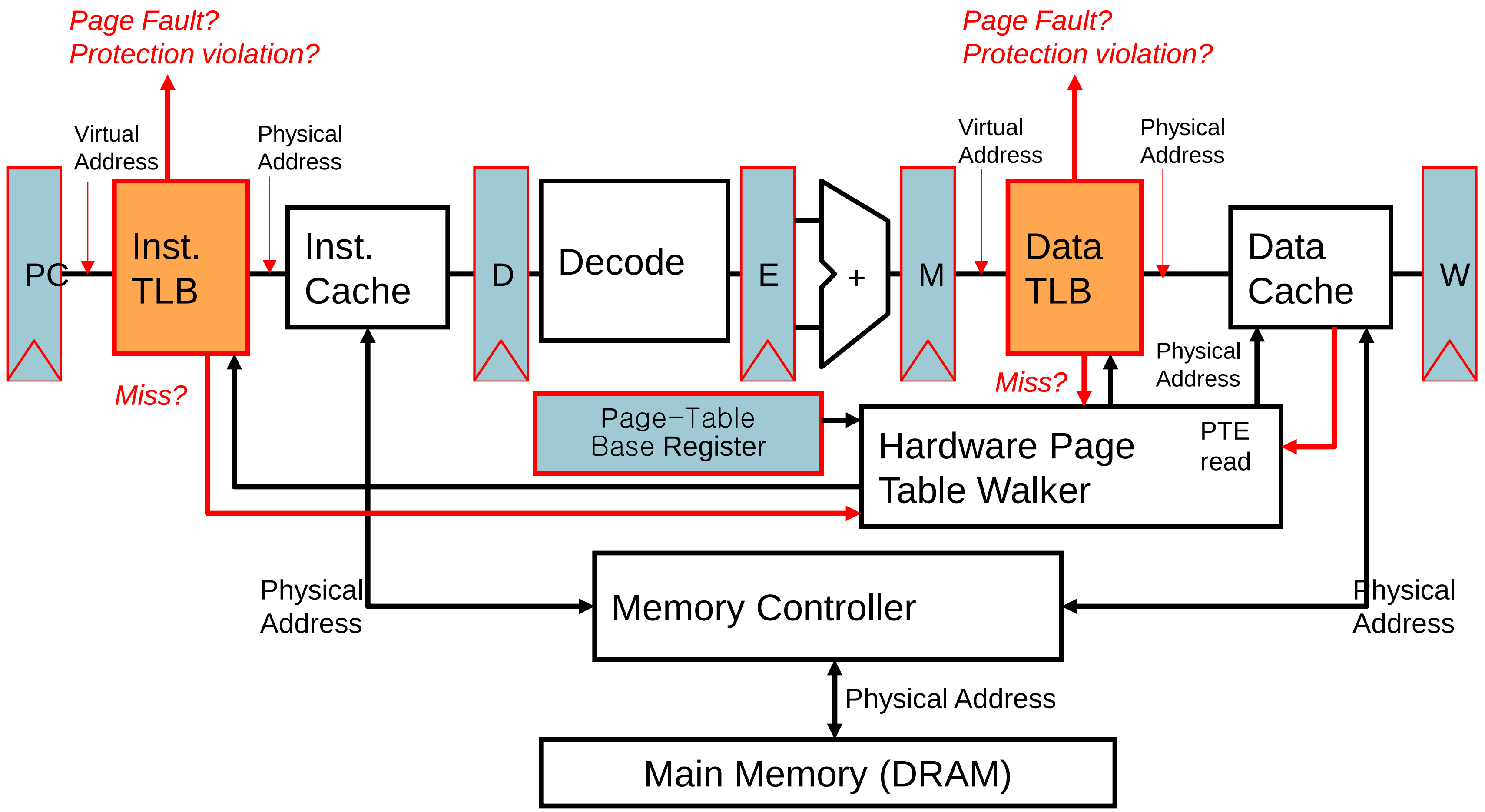
MMU does this page table walk in hardware on a TLB miss

A linear Page Table



- 1 M entries, which corresponds to 256 pages, assuming that each entry is 32-bit.
- too big to be all entries in the memory

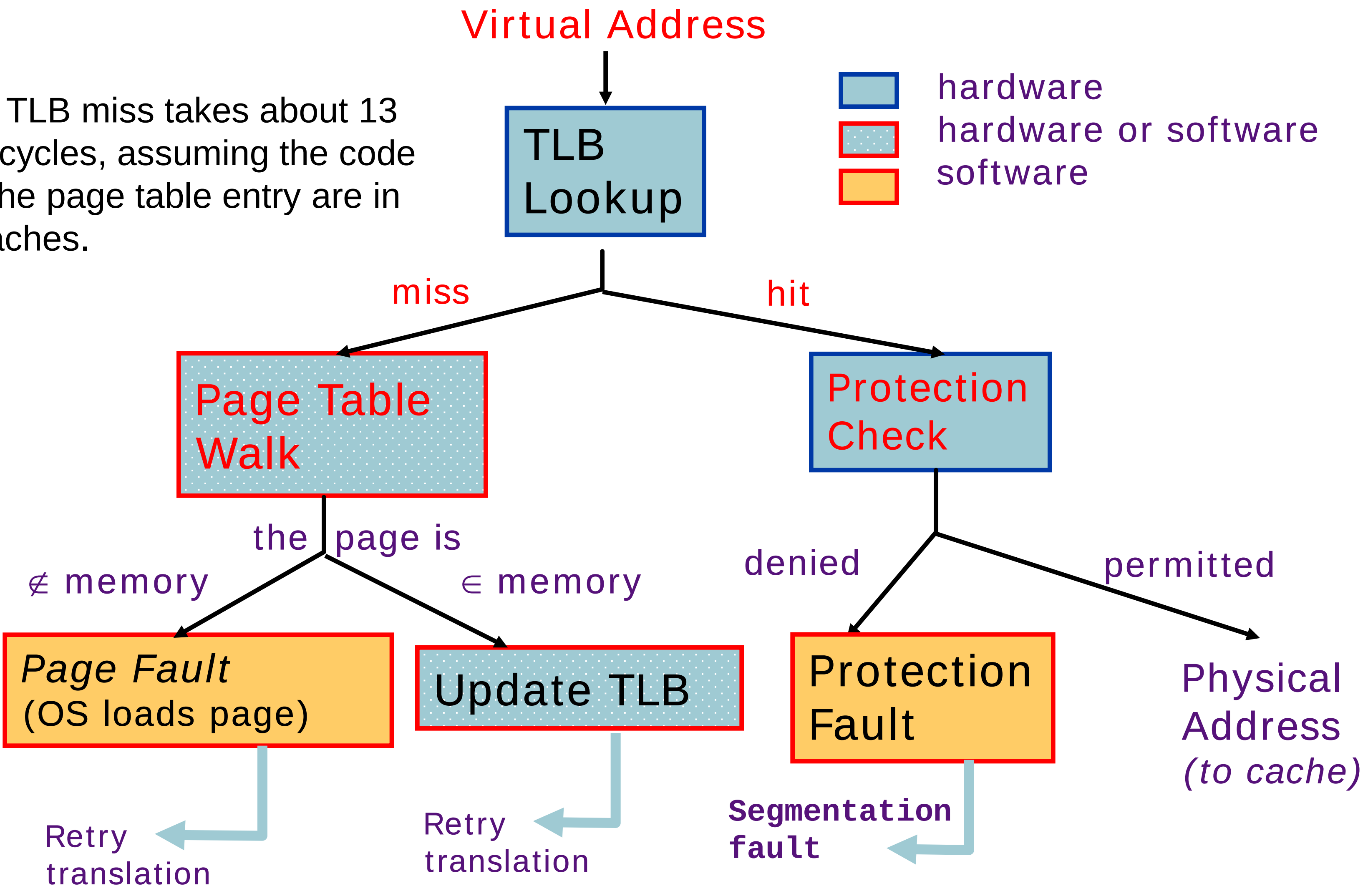
Page-Based Virtual-Memory



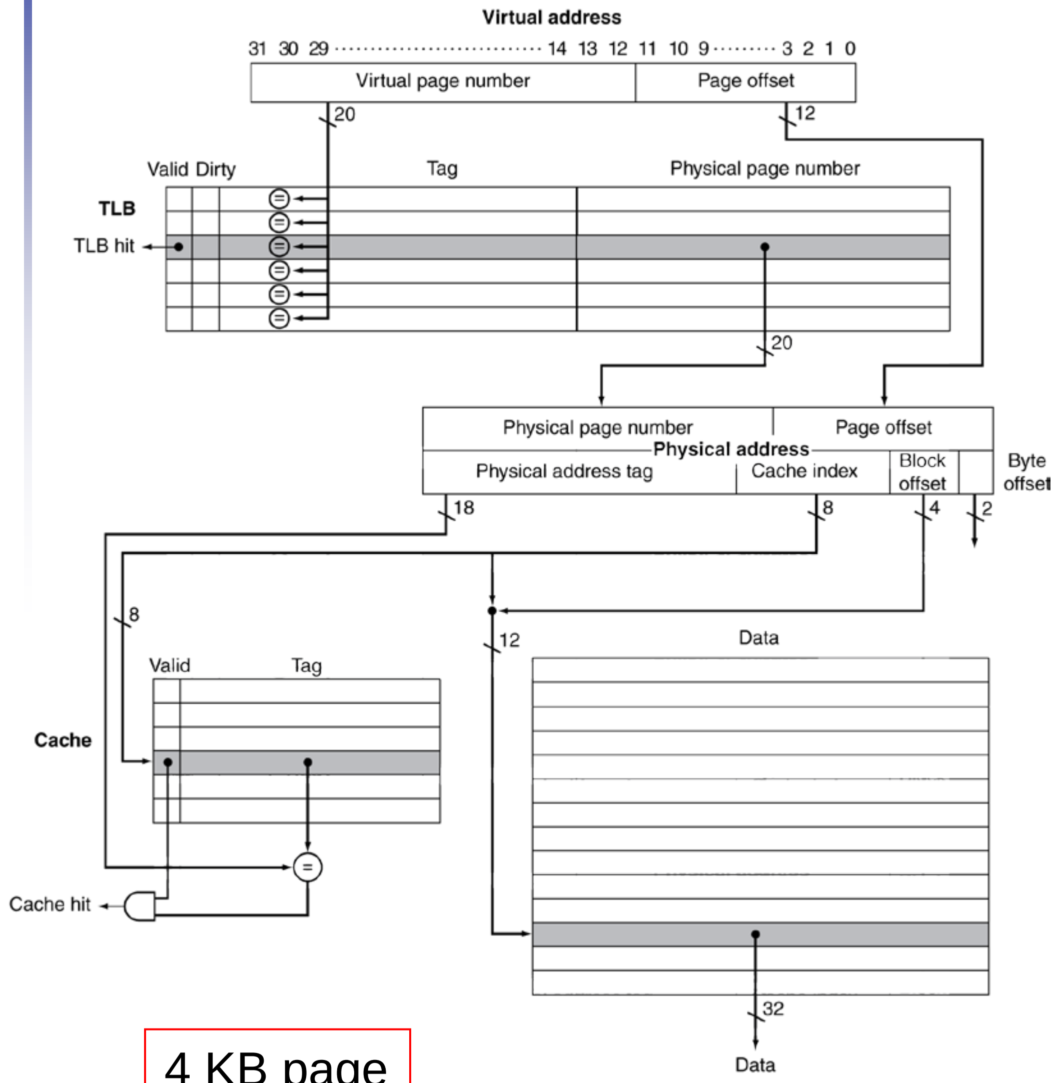
- Assumes page tables held in untranslated (upmapped) physical memory

Address Translation: detailed?

e.g. A TLB miss takes about 13 clock cycles, assuming the code and the page table entry are in the caches.



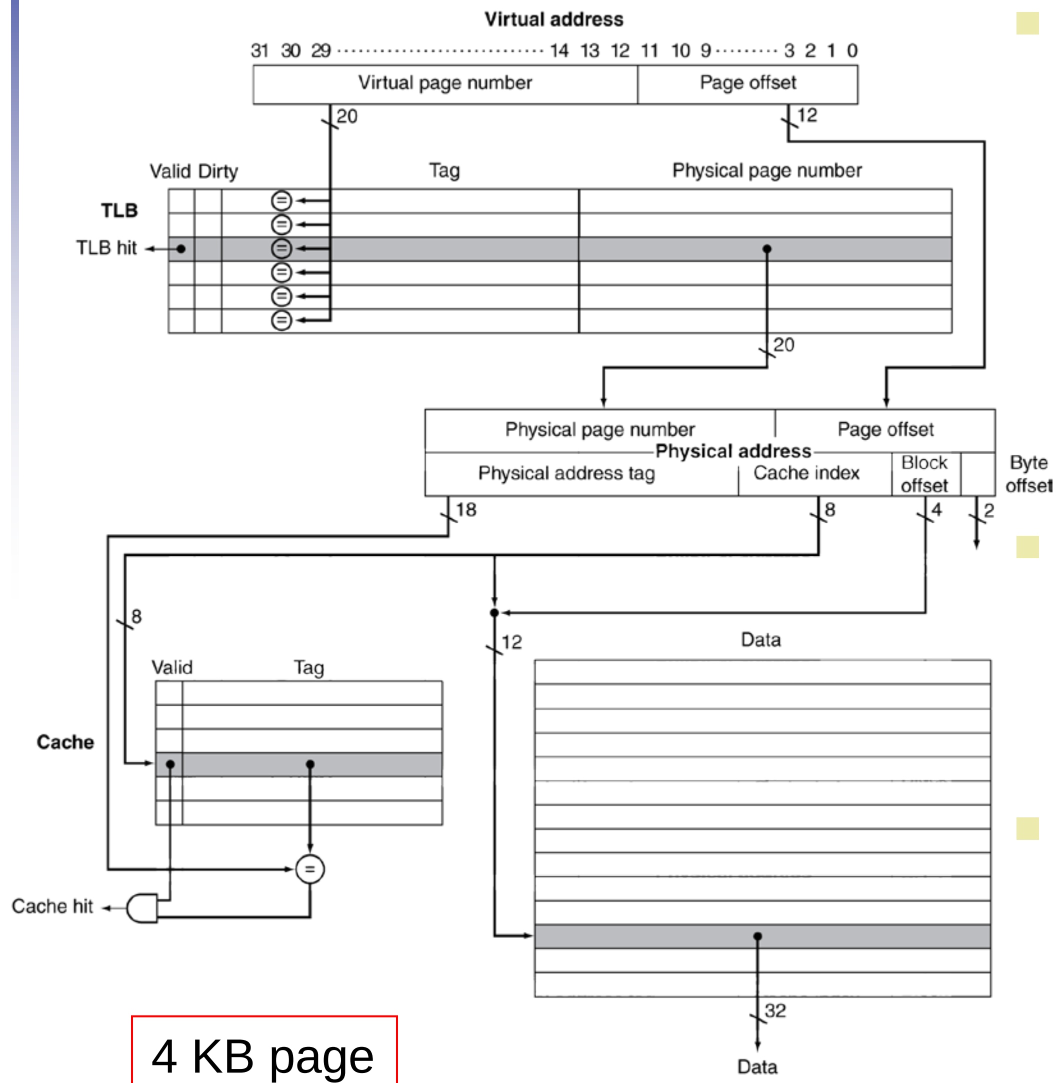
TLB and Cache Interaction



4 KB page

- Note that the tag and data RAMs are split.
- By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor.
- While the cache is direct mapped, the TLB is fully associative.

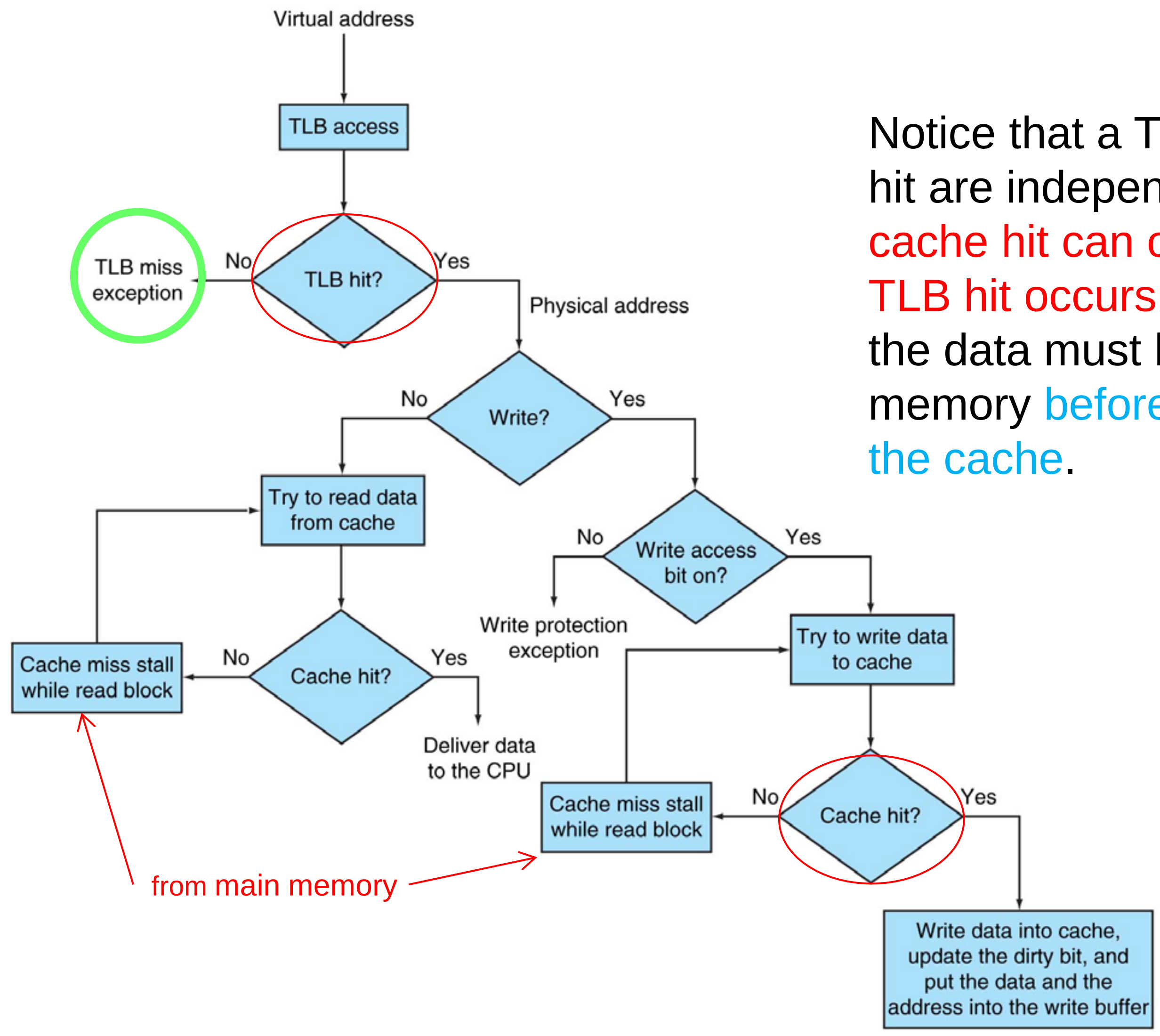
TLB and Cache Interaction



4 KB page

- If the valid bit of the matching TLB entry is on, the access is a TLB hit.
  - Bits from the physical page number together with bits from the page offset form the index that is used to access the cache.
- If the cache tag uses physical address
  - Need to translate it before cache lookup
- Alternative: use virtual address tag (?)
  - No need to translate for cache access

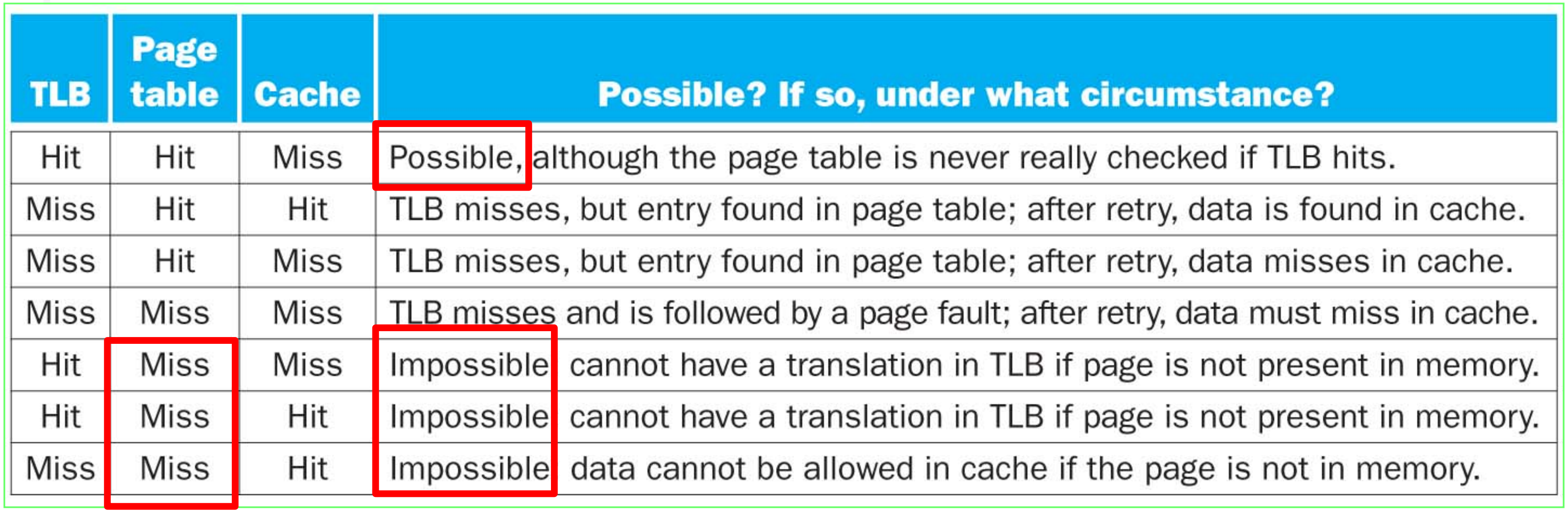
TLB and Cache Interaction


Virtual address
TLB access
TLB hit?
No
Yes
TLB miss exception
Physical address
Write?
No
Yes
Try to read data from cache
Write access bit on?
No
Yes
Write protection exception
Try to write data to cache
Cache miss stall while read block
Cache hit?
No
Yes
Deliver data to the CPU
Cache miss stall while read block
Cache hit?
No
Yes
Write data into cache, update the dirty bit, and put the data and the address into the write buffer
from main memory

Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory before being loaded into the cache.

Possible Combination of Events

- a TLB miss, a page fault, and a cache miss
- Assuming that the cache is physically indexed and physically tagged
- Three of these combinations are impossible
- One is possible (TLB hit, virtual memory hit, cache miss) but never detected.



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.

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 (e.g. writes for TLB and page table register)
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS), which is a special instruction that transfers control from user mode to a dedicated location in supervisor code space., invoking the exception mechanism in the process.
  - Return from exception (ERET): change to user mode as well as restoring PC

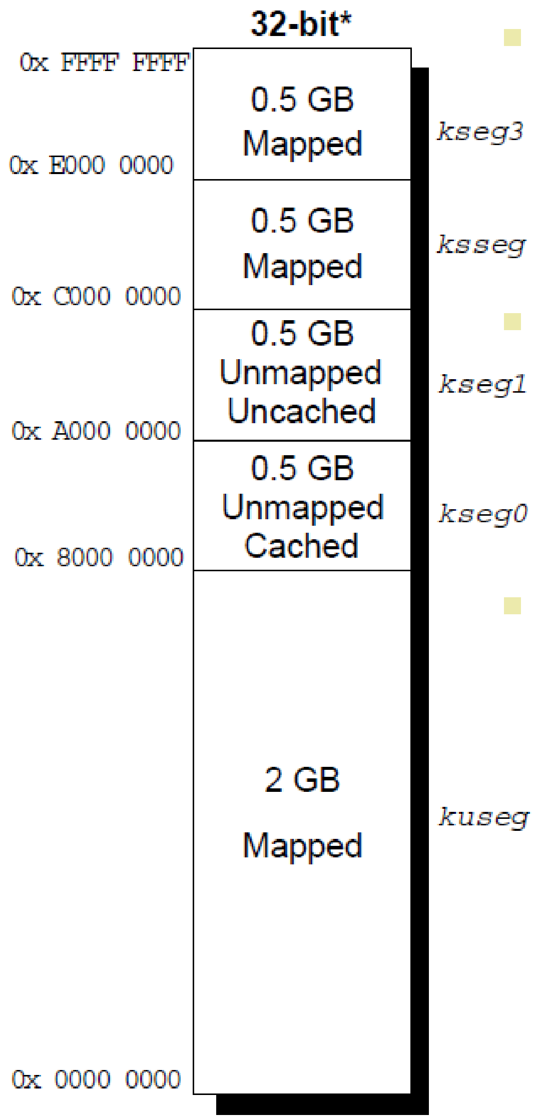
Page Fault Handler

- Transfer control to the OS to deal with a page fault.
  - Detect a page fault by checking the valid bit of the page table entry
- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first: very slow
- Read page into memory and update page table: very slow
- OS select another process to execute in the processor until the disk access completes: context switching
- OS can restore the state of the process originally caused the page fault and execute ERET
  - The user process restart from faulting instruction

Context switch

- Change the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.
  - When the OS decides to change from running process P1 to running process P2.
- If there is no TLB, it suffices to change the page table register to point to P2's page table.
- With a TLB, we must clear the TLB entries that belong to P1. – not efficient if process switch rate were high.
- Alternative: extend the virtual address space by adding a process identifier (task identifier) : e.g. 8-bit address space ID (ASID)
  - Similar problems can occur for a cache.

MIPS R3000



- Kernal mode address
  - 0xx : kuseg 2GB: mapped like user process
  - 100 : kseg0 0.5GB: cached and unmapped
  - 101 : kseg1 0.5GB: uncached and unmapped
  - 11x : kseg2 1GB: mapped cachable (user page table)
- Address mapping: 31 bits (top bit is 0)+ 6-bit process id
  - Upper 19 bits + 6-bit process id → physical page #
  - Use a linear page table in kseg2
  - We also use paging kseg2 using a linear page table
  - The page table for kseg2 is stored in kseg0
- Given a 32-bit virtual address from kuseg,
  - Top 9 bits: index of kseg2 page table stored in kseg0 (unmapped)
  - After this lookup, if the page table is not resident in kseg2, read it back from disk
  - Middle 10 bits: index of the page table
  - After this lookup of the physical page #, , if the page table is not resident in kuseg, read it back from disk
  - Lower 12 bits: offset of the physical page in kuseg

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 (page table)

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss (page fault) 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 Pollution

- defined as the displacement of a cache element by a less useful one.
- In the context of processor caches, cache pollution occurs whenever a non-reusable cache line is installed into a cache set, displacing a reusable cache line.
- Reusability is determined by the number of times a cache line is accessed after it is initially installed into the cache but before its eviction.

Virtual Machines

- Idea related to VM is as old as virtual memoy
- 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
- VMM is the software that support VMs
  - Host: underlying hardware
  - Guests: VMs that share the resources

System Virtual Machines

- VM:(broadest definition) basically all emulation methods that provide a standard software interface, such as Java VM
- System VM:(Here) provide a complete system-level environment at the binary ISA level.
  - But some VM run different ISA in the VM from the native hardware
- Present the illusion that the users have an entire computer themselves, including a copy of OS
  - A computer runs multiple VMs and can support various OSes.
- Examples: IBM VM/370 (1970s technology!), VMWare, Microsoft Virtual PC

Virtual Machine Monitor

- VMM or Hypervisor
- VMM determines how to map virtual resources to physical resources (Memory, I/O devices, CPUs)
  - Much smaller than a traditional OS
  - Isolation part of the VMM : only 10K lines of code
- VMs provide three major benefits
  - Provide improved protection
  - Managing software: provide an abstraction that can run the complete software stacks: old, stable, new OSes
  - Managing hardware: VM allows separate software stacks to run independently yet share hardware. Some VMM support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware

Virtualization Cost

- Depends on workload
  - Processor-bound program: zero overhead (OS is rarely invoked)
  - IO-intensive program: high overhead (OS-intensive)
- The overhead is determined by both the number of instructions that must be emulated by the VMM and by how much time each takes to emulate
- VMM must control just about everything
  - Access to privileged state
  - Address translation
  - I/O
  - Exceptions, and interrupts

Virtual Machine Monitor

- What must a VMM do?
  - It presents a software interface to guest software
  - It must isolate the state of guests form each other
  - It must protect itself from guest software, including guest OSes.
- 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 on timer interrupt
  - 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 for VM

- Basic requirements
  - 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
- Virtualizable ISA
  - IBM 370 architecture: modified IBM 360 ISA
  - X86, MIPS, ARM are not a virtualizable ISA
- Renaissance of virtualization support
  - Current ISAs (e.g., x86) adapting

ISA for protection

- Protection is a joint effort of architecture and operating system → Need to change ISA
- Example:
    - X86 instruction POPF: load the flag registers from TOS, including IE (interrupt enable)
        - In user mode: trap it and simply change all flags except IE
        - In system mode: it does change IE (since a guest OS runs in user mode inside the VM, it is a problem!)
- Three steps to improve VM performance
    1. Reduce cost of processor virtualization
    2. Reduce interrupt overhead cost due to virtualization
    3. Reduce interrupt cost by steering interrupts to the proper VM without invoking the VMM
    - In 2006, Intel and AMD try to address the first point

VM and virtual memory

- Virtualization of virtual memory: introduce another indirection on each memory access
  - Virtual memory, real memory, physical memory
  - Different terms: Virtual memory, physical memory, machine memory
- Guest OS: map virtual memory to real memory
- VMM: map real memory to physical memory

VM and virtual memory

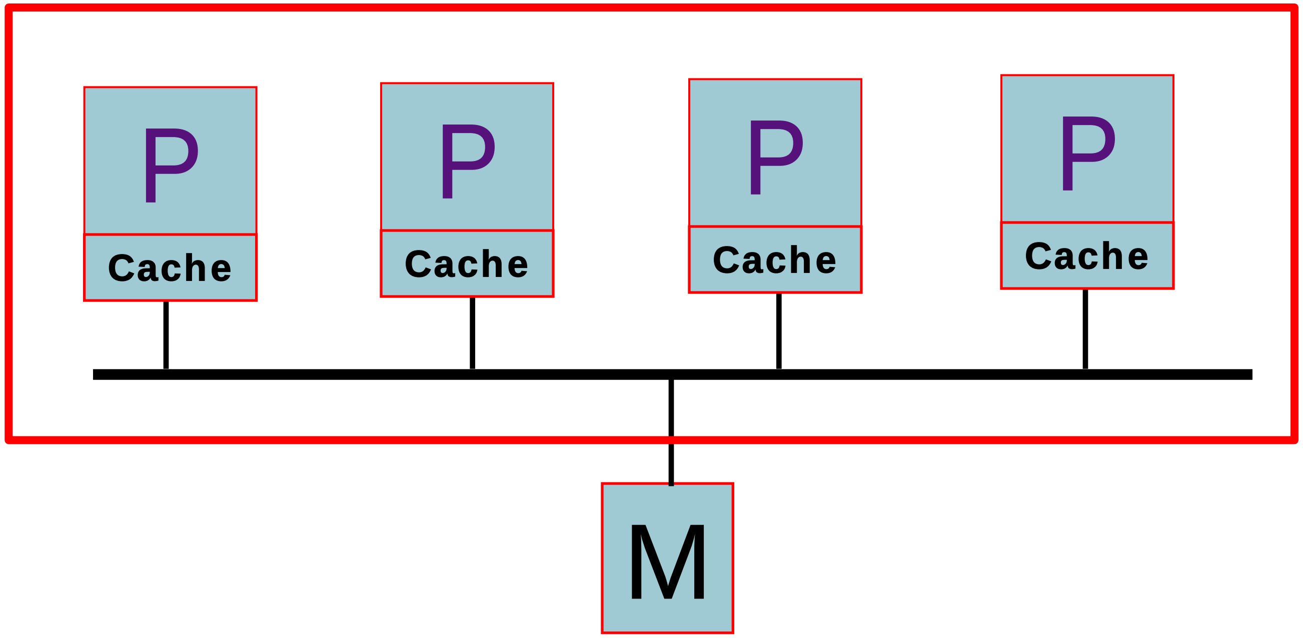
- Rather than pay an extra level of indirection, VMM maintains a shadow page table that maps directly for the guest virtual address to the physical address space of the hardware.
- VMM must trap any attempts by guest OS to change its page table or to access page table pointer, which is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS.

Virtualization of I/O

- Virtualization of I/O is by far the most difficult part of system virtualization
  - Because of the increasing number of I/O devices attached to a computer and the increasing diversity of I/O device types
- Another difficult is sharing of a real device among multiple VMs
- The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and leaving it to the VMM to handle real I/O

Parallelism and Memory Hierarchies

- The processors on a single chip shares a common physical address space
- Each processor has its own cache



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

Different value: Cache coherent problem !

Coherence and Consistency

- A memory system is coherent if reads return most recently written value.
  - This definition, although intuitively appealing, is vague and simplistic
  - The reality is much more complex
- This simple definition contains two different aspects of memory system behaviour.
  - Both of which are critical to writing correct shared memory program
  - Coherence: what values can be returned by a read
    - Most recently (or last) written value
  - Consistency: when a written value will be returned by a read (or when a read can see a written value)

A memory system is coherence if

- (1. preserving program order) A read by a processor P to a location X that follows a write by the same processor P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. -- uniprocessor
- (2. a coherent view of memory) A read by a processor P1 to location X that follows a write by another processor P2 to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

  Not addressing (consistency) exactly when P1 see a write of X by P2
- (3. write serialization) Two writes to the same location by any two processors are seen in the same order by all processors.

Basic schemes for coherence

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration (copy) of shared data to local caches
    - Reduces both bandwidth and latency for shared memory
  - Replication of read-shared data in a local cache
    - Reduces both latency and contention for access
- Supporting this migration and replication is critical to performance in accessing shared data
  - Introduce a hardware protocol to maintain coherent data.
- Key to implementing a cache coherent protocol is tracking the state of any sharing of a data block.

Cache Coherence Protocols

- Snooping protocols: most popular
  - Every cache that has a copy of the shared data also has a copy of the sharing status of the block
  - No centralized status is kept.
  - The cache are accessible via some broadcast medium (a bus or network)
  - Each cache controller monitors the medium to determine whether or not they have a copy of a block that is requested on a bus.
- Directory-based protocols
  - Caches and memory record sharing status of blocks in just one location, called the directory
  - More scalable: higher overhead but lower traffic between caches

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is 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	$\xleftarrow{\text{migration}}$	0
CPU B reads X	Cache miss for X	0	$\xrightarrow{\text{replication}}$ 0	0
CPU A writes 1 to X	Invalidate for X	1	$\xrightarrow{\text{invalidate}}$	0
CPU B read X	Cache miss for X	1	$\xrightarrow{\text{replication}}$ 1	$\xrightarrow{\text{write-back}}$ 1

- The CPU cache and memory contents show the value after the processor and bus activity have both completed.
- When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated.
- This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced.
- This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

False Sharing

- Block size plays an important role in cache coherency
  - For example, take the case of snooping on a cache with a block size of 8 words, with a single word alternatively written and read by two processors.
  - Most protocols exchange full blocks between processors, thereby increasing coherency bandwidth demands.
- Large blocks can also cause what is called false sharing:
  - When two unrelated shared variables are located in the same cache block, the full block is exchanged between processors.

Memory Consistency Model

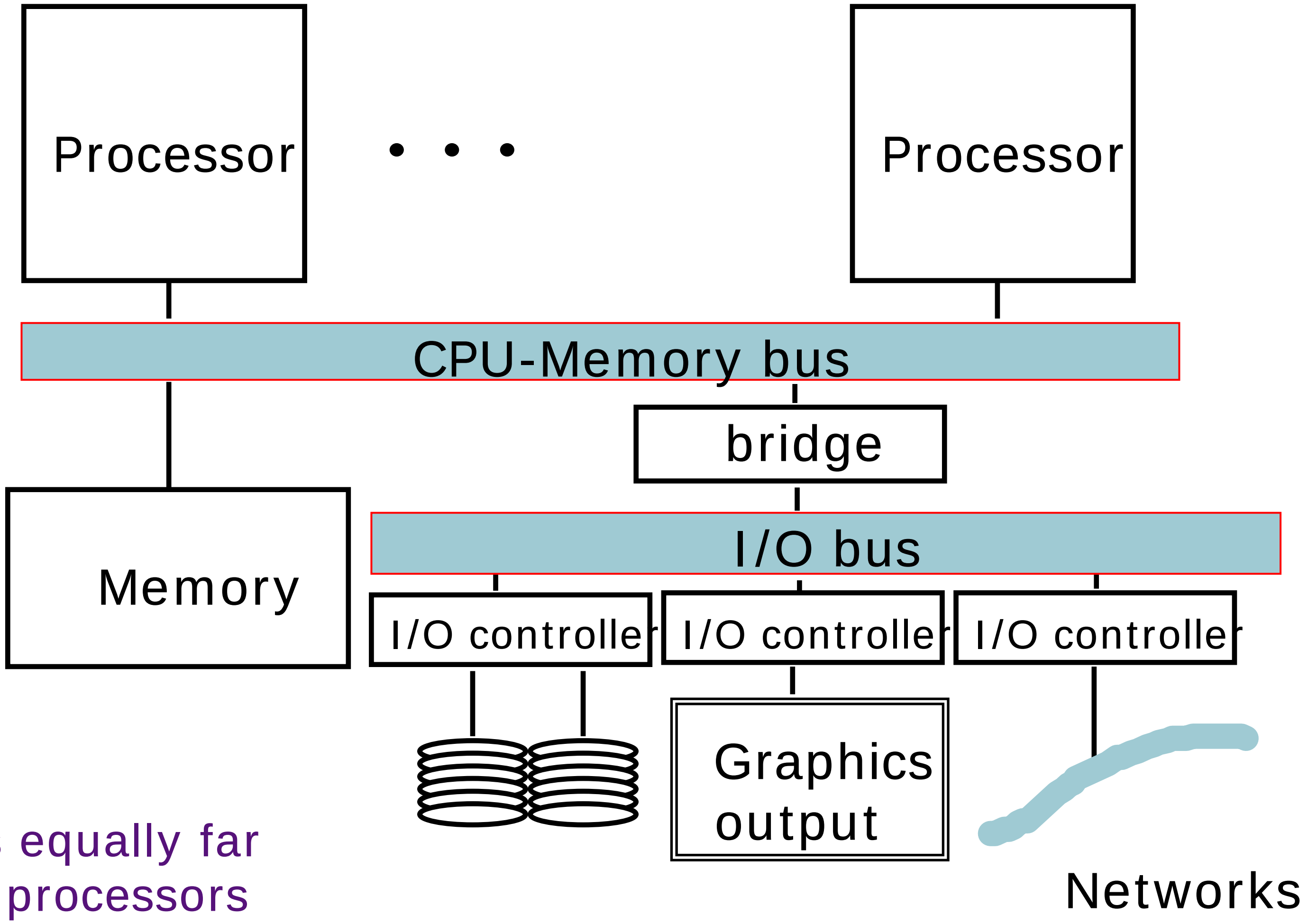
- Suppose two CPU cores share a physical address space
  - Write-through caches

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

Different value: Cache coherent problem !

When does CPU B see a write of X by CPU A?
Depends on memory consistency model

Symmetric Multiprocessors



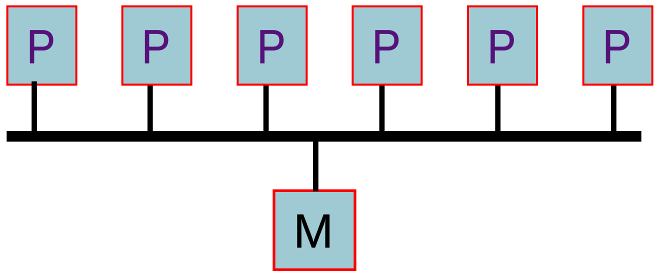
symmetric

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Memory Consistency Model

- When are writes seen by other processors?
  - "Seen" means that 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 (read or write) accesses
- Consequence
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Therefore, processors can reorder reads, but not writes

Sequential Consistency (SC)



" A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency = (severe constraint)
arbitrary *order-preserving interleaving*
of memory references of sequential programs

A straightforward implementation

- Need to satisfy the following two requirements
  - (program order requirement) a processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order.
  - (write atomicity requirement pertaining only to cache-based system) it requires that writes to the same location be serialized. (i.e. writes to the same location be made visible in the same order to all processors) and that the value of a write not be returned by a read until all invalidates or updates generated by the write are acknowledges (i.e. until the write become visible to all processors).

A simple example

- The question of how consistent memory must be seems simple but remarkably complex.

- Two code segments form processes P1 and P2
  ```
  P1:    A=0;          P2:   B=0;
         ….                  ….
         A=1;                B=1;
  L1:    if (B==0)…    L2:   if (A==0)…
  ```
- Assumptions
  - The processes are running on different processors
  - The locations A and B are originally cached by both processors

The programmer's View

- The SC model has a performance disadvantage, but the advantage of simplicity.
- A program is synchronized if all accesses to shared data are ordered by synchronization operations. – data race free
- Consider a simple example where a variable is read and updated by two different processors.
  - each processor surround the read and update with a lock and an unlock both to ensure mutual exclusion for the update and to ensure that the read is consistent.
- Most programs must be synchronized

Sequential Consistency model

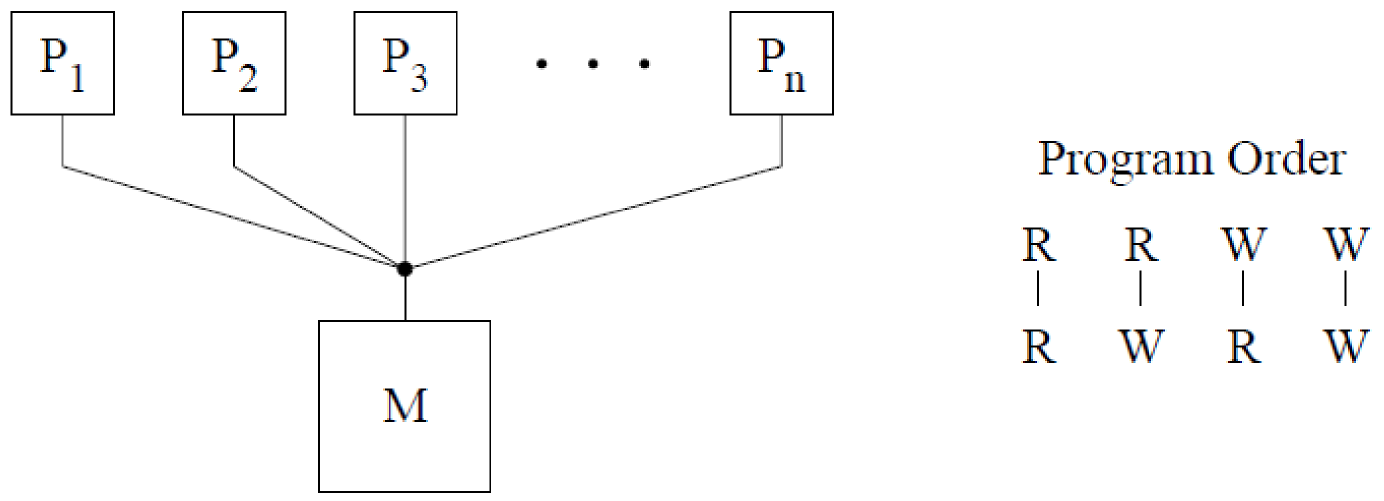
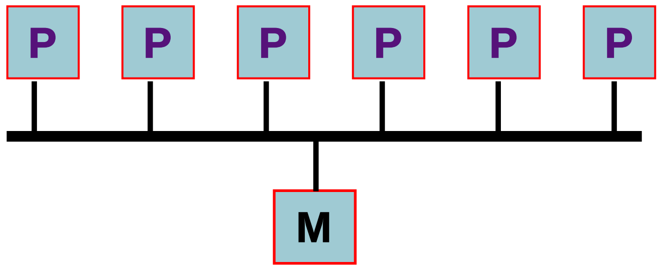


Figure 2.11: Representation for the sequential consistency (SC) model.

Issues in Implementing SC



Implementation of SC is complicated by two issues

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a $\neq$ b
Store(a); Load(b)	*yes if* a $\neq$ b
Store(a); Store(b)	*yes if* a $\neq$ b

- *Caches*

  Caches can prevent the effect of a store from being seen immediately by other processors

No common commercial architecture has a sequentially consistent memory model!

Relaxed Consistency Models

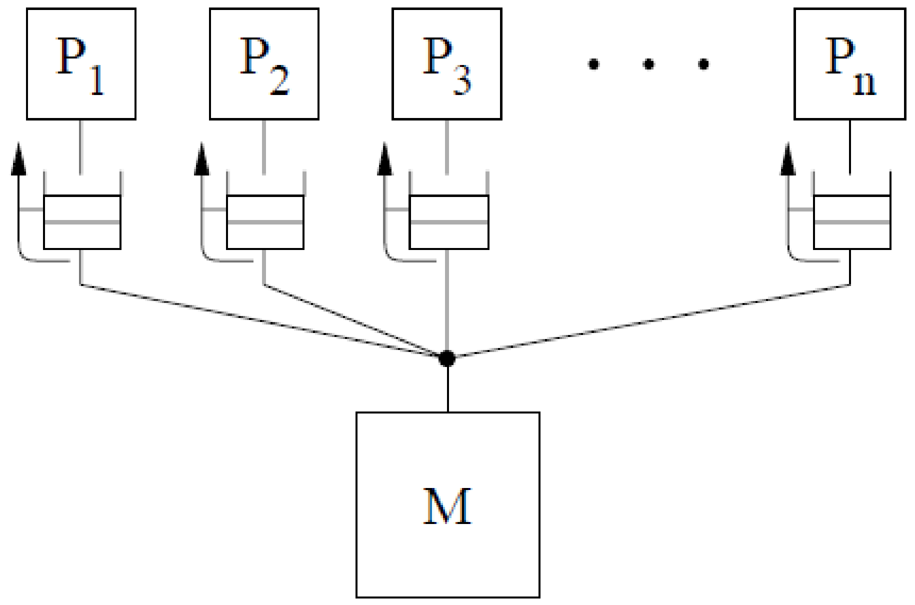
Allows read and writes to complete out of order to obtain performance advantages.

Three major sets of orderings that are relaxed are

1. Relax W→R ordering : total store ordering (TSO) or processor consistency, retains orders among writes
2. Relax W→W ordering: partial store ordering (PSO)
3. Relax R→W and R→R orderings: various models including weak ordering (WO) and release consistency

Since synchronization is highly specific and error prone, the expectation is that most programmers will use standard synchronization libraries and will write synchronized program, making the choices of a weak consistency model invisible to the programmers and yielding higher performance.

TSO Memory Model



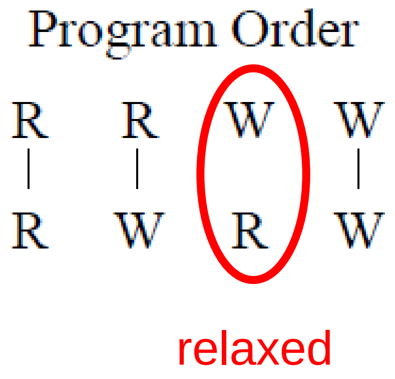
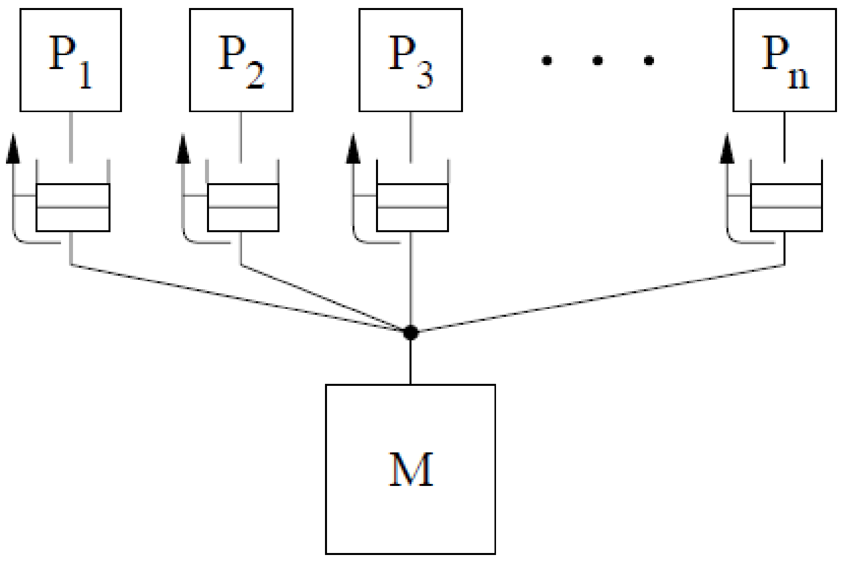


Figure 2.13: The total store ordering (TSO) memory model.

- Proposed for the SPARC V8 architecture
- The value of a write in the buffer is allowed to be forwarded to a read.

PSO Memory Model



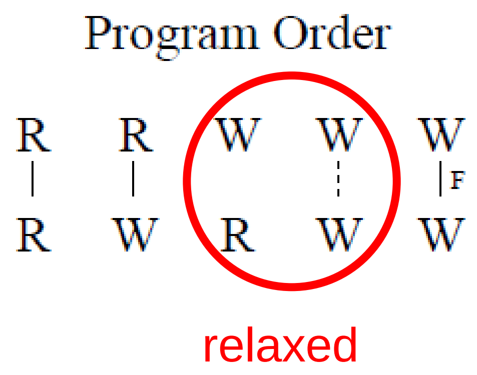
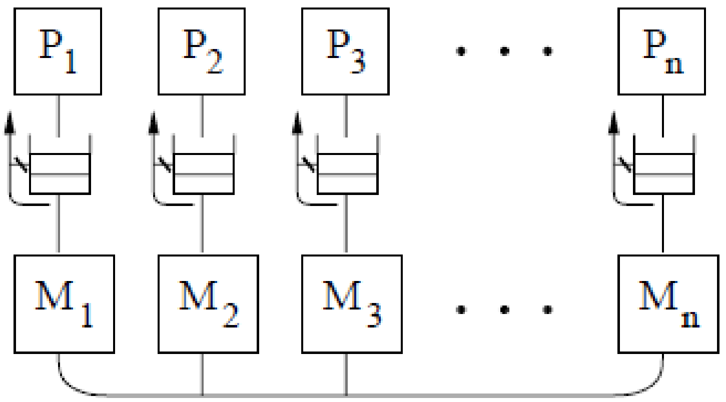
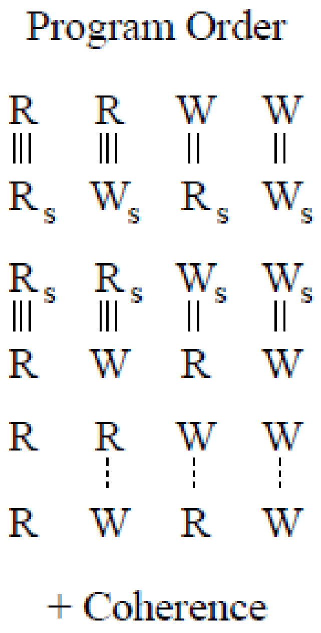


Figure 2.16: The partial store ordering (PSO) model.

- An extension of the TSO model
- Writes to different locations are allowed to execute out of program order
- PSO also provide a fence instruction, called the store barrier or STBAR, that may be used to enforce the program order between writes.

WO Memory Model

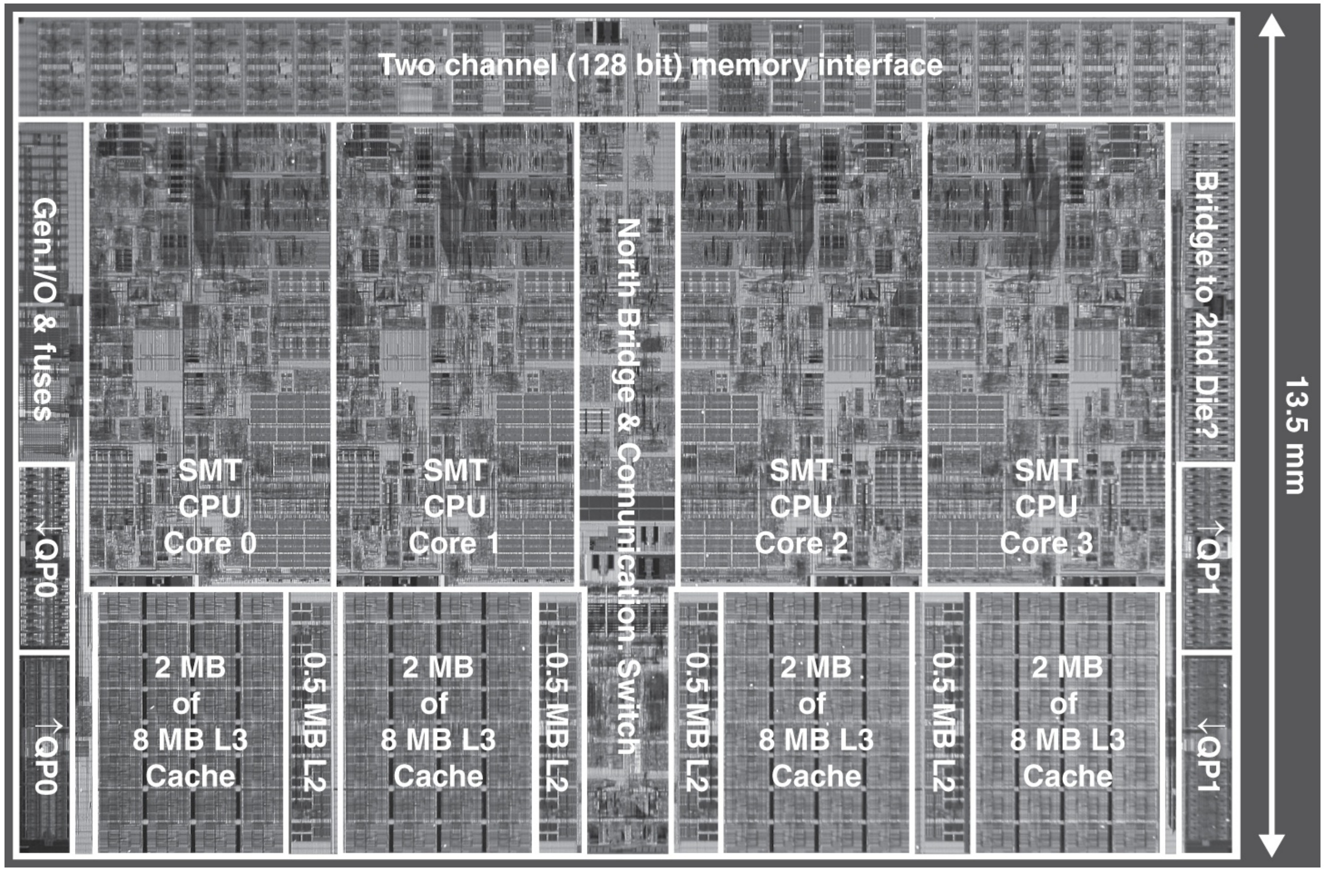




- Relies on maintaining program order only at the synchronization points in the program
- Rs and Ws: read and write operations identified as synchronization
- double lines: Each operation consists of multiple sub-operations. All sub-operations of the first must complete before any sub-operation of the second
- triple lines (after a read): the read sub-operation and the sub-operations of write (possibly form a different processor) whose value is returned by the read must complete before any sub-operation of the second.

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



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

2-Level TLB Organization

	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 for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both TLB: 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
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

Miss Penalty Reduction

- Return requested word first
  - Then back-fill rest of block
- Non-blocking cache
  - Used by designers who are attempting to hide the cache miss latency by using out-of-order processors
  - Hit under miss: allow hits to proceed
  - Miss under miss: allow multiple outstanding misses
- Hardware prefetch: instructions and data
- Opteron X4: 8-bank interleaved L1 D-cache
  - Two concurrent accesses per cycle
  - Two-ported cache: too expensive

Inclusion

- Cache coherence protocol
  - Inclusion, non-inclusion, exclusion
- Inclusion policy: (Nehalem or most other processors)
  - A copy of all data in higher level caches can also be found in lower-level cache.
  - i.e. a copy of all data in L1 cache can also be in L2 cache
  - If a block is replaced in the L2 cache due to a conflict or capacity miss, that same block must be evicted in all of the L1’s in which it is present.

Exclusion or Non-Inclusion

- Exclusion policy: AMD processors follows the policy of exclusion in L1 and L2 cache.
  - A cache block can only be found in L1 or L2 caches, bit not both.
  - Shared L3 cache doe not always follow exclusion
- Non-Inclusion lies in between the inclusion and exclusion
  - A block can be present in both the L1 and L2 or one or the other.

Pitfalls

- Forgetting to account for byte addressing or the cache block size in simulating a cache.
- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1 : 36/4 mod 8 = 1
    - Word 36 maps to block 4: 36 mod 8 = 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

- Ignoring memory system behavior when writing programs or when generating code in a compiler
- matrix multiply

  -
    ```
    for ( i=0; i !=500; i =i+1)
          for ( j=0; j !=500; j =j+1)
                for ( k=0; k !=500; k =k+1)
                      x[i][j] = x[i][j] + y[i][k] * z[k][j];
    ```
  - 1MB secondary cache
  - Changing the loop order to k, j, i speed up twice
  - Blocking can make it 4x faster

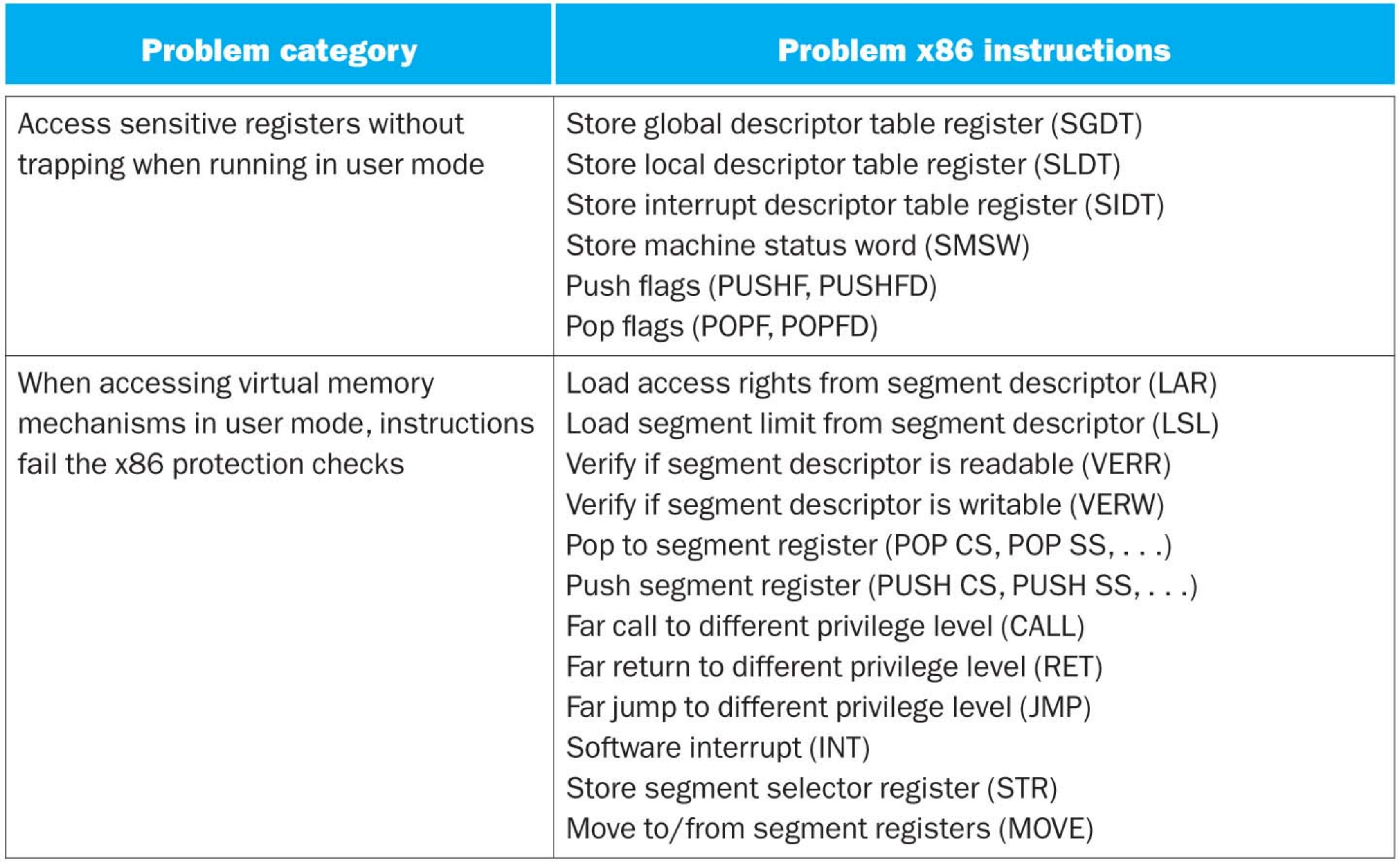
Pitfalls

- Having less set associativity for a shared cache less than the number of threads sharing cache
- 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
- Separately calculate the memory-stall time and the processor execution time
  - Ignores effect of non-blocked cache accesses
  - Instead, evaluate performance by simulation

Pitfalls

- Extending an address space by adding segments on top of an unsegmented address space
  - 32-bit address space: 16-bit segment + 16-bit address
  - 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

X86 ISA and Virtualization



Problem category	Problem x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- Principle of temporal and spatial localities
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- Virtual memory gives illusion of using the whole address space
- Memory system design is critical for multiprocessors

Another trend

- seek software help to efficiently manage the memory hierarchy.
- Reorganize the program to enhance its spatial and temporal locality.
  - This approach focus on loop-oriented programs with large arrays
  - Restructuring the loops
- Prefetching: a technique in which data blocks needed in the future are brought into the cache early by the use of special instructions that specify the address of the block

Homework: chapter 5

- Due before starting the class on Nov. 15
- Exercise 5.10
- Exercise 5.14
- Exercise 5.16