Virtual Memory

Muhamed Mudawar

CS 282 – KAUST – Spring 2010

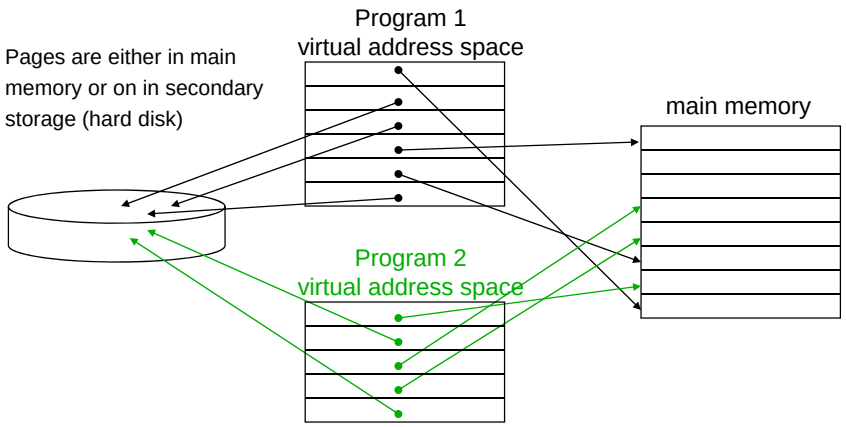
Computer Architecture & Organization

Virtual Memory Concepts

- What is Virtual Memory?
  - Uses disk as an extension to memory system
  - Main memory acts like a cache to hard disk
  - Each process has its own virtual address space
- Page: a virtual memory block
- Page fault: a memory miss
  - Page is not in main memory ⇒ transfer page from disk to memory
- Address translation:
  - CPU and OS translate virtual addresses to physical addresses
- Page Size:
  - Size of a page in memory and on disk
  - Typical page size = 4KB – 16KB

Virtual Memory Concepts

- A program's address space is divided into pages
  - All pages have the same fixed size (simplifies their allocation)



Issues in Virtual Memory

- Page Size
  - Small page sizes ranging from 4KB to 16KB are typical today
  - Large page size can be 1MB to 4MB (reduces page table size)
  - Recent processors support multiple page sizes
- Placement Policy and Locating Pages
  - Fully associative placement is typical to reduce page faults
  - Pages are located using a structure called a page table
  - Page table maps virtual pages onto physical page frames
- Handling Page Faults and Replacement Policy
  - Page faults are handled in software by the operating system
  - Replacement algorithm chooses which page to replace in memory
- Write Policy
  - Write-through will not work, since writes take too long
  - Instead, virtual memory systems use write-back

Three Advantages of Virtual Memory

- Memory Management:
  - Programs are given contiguous view of memory
  - Pages have the same size ➔ simplifies memory allocation
  - Physical page frames need not be contiguous
  - Only the "Working Set" of program must be in physical memory
  - Stacks and Heaps can grow
    - Use only as much physical memory as necessary
- Protection:
  - Different processes are protected from each other
  - Different pages can be given special behavior (read only, etc)
  - Kernel data protected from User programs
  - Protection against malicious programs
- Sharing:
  - Can map same physical page to multiple users "Shared memory"

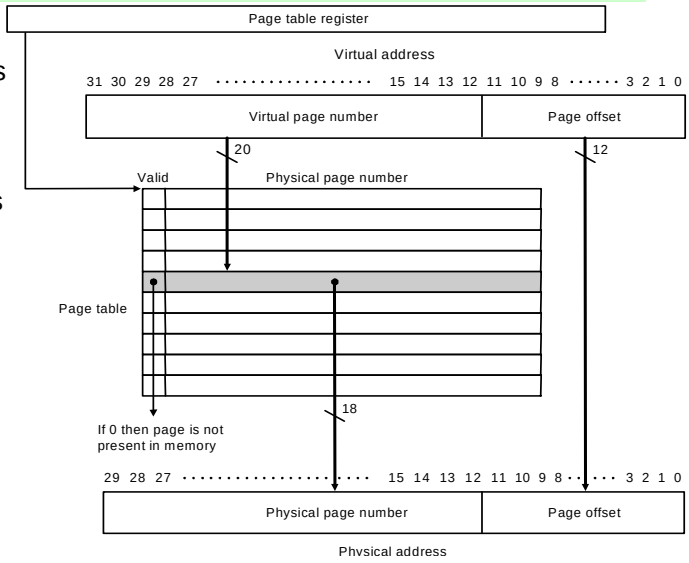
Page Table and Address Mapping

Page Table maps virtual page numbers to physical frames

Page Table Register contains the address of the page table

Virtual page number is used as an index into the page table

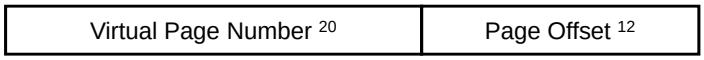
Page Table Entry (PTE): describes the page and its usage



Page Table – cont'd

- Each process has a page table
  - The page table defines the address space of a process
  - Address space: set of page frames that can be accessed
  - Page table is stored in main memory
  - Can be modified only by the Operating System
- Page table register
  - Contains the physical address of the page table in memory
  - Processor uses this register to locate page table
- Page table entry
  - Contains information about a single page
  - Valid bit specifies whether page is in physical memory
  - Physical page number specifies the physical page address
  - Additional bits are used to specify protection and page use

Size of the Page Table

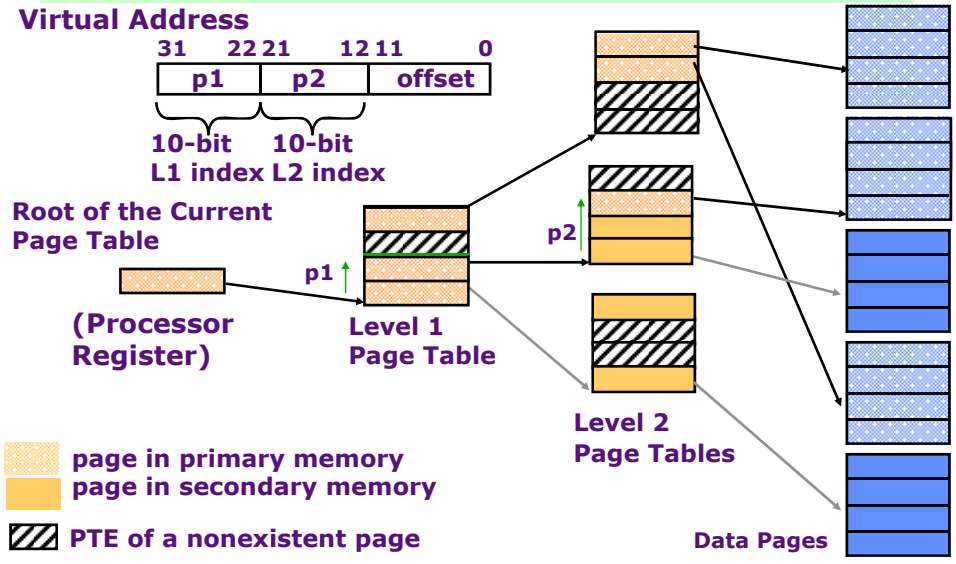


- One-level table is simplest to implement
- Each page table entry is typically 4 bytes
- With 4K pages and 32-bit virtual address space, we need:
  $2^{32}/2^{12} = 2^{20}$ entries × 4 bytes = 4 MB
- With 4K pages and 48-bit virtual address space, we need:
  $2^{48}/2^{12} = 2^{36}$ entries × 4 bytes = 2^{38} bytes = 256 GB !
- Cannot keep whole page table in memory!
- Most of the virtual address space is unused

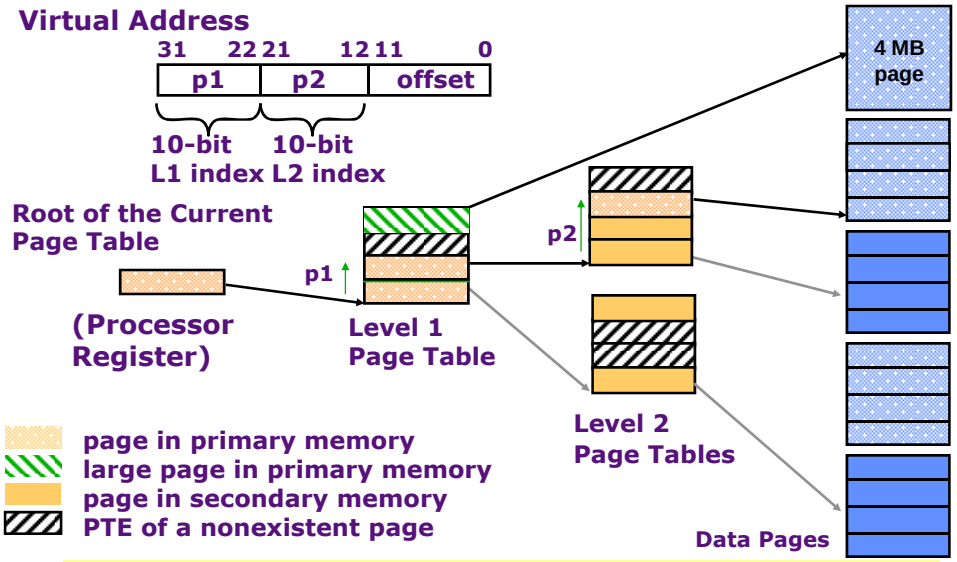
Reducing the Page Table Size

- Use a limit register to restrict the size of the page table
  - If virtual page number > limit register, then page is not allocated
  - Requires that the address space expand in only one direction
- Divide the page table into two tables with two limits
  - One table grows from lowest address up and used for the heap
  - One table grows from highest address down and used for stack
  - Does not work well when the address space is sparse
- Use a Multiple-Level (Hierarchical) Page Table
  - Allows the address space to be used in a sparse fashion
  - Sparse allocation without the need to allocate the entire page table
  - Primary disadvantage is multiple level address translation

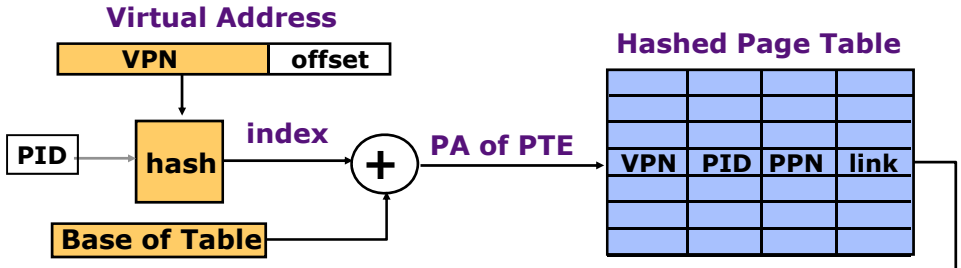
Multi-Level Page Table



Variable-Sized Page Support



Hashed Page Table



- One table for all processes
  - Table is only small fraction of memory
  - Number of entries is 2 to 3 times number of page frames to reduce collision probability
- Hash function for address translation
  - Search through a chain of page table entries

Handling a Page Fault

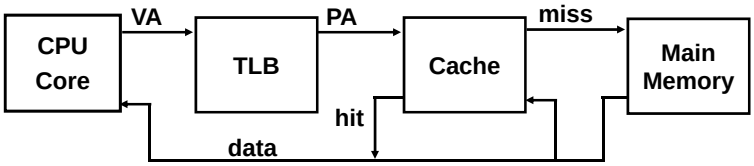
- Page fault: requested page is not in memory
  - The missing page is located on disk or created
  - Page is brought from disk and Page table is updated
  - Another process may be run on the CPU while the first process waits for the requested page to be read from disk
- If no free pages are left, a page is swapped out
  - Pseudo-LRU replacement policy
  - Reference bit for each page each page table entry
  - Each time a page is accessed, set reference bit =1
  - OS periodically clears the reference bits
- Page faults are handled completely in software by the OS
  - It takes milliseconds to transfer a page from disk to memory

Write Policy

- Write through does not work
  - Takes millions of processor cycles to write disk
- Write back
  - Individual writes are accumulated into a page
  - The page is copied back to disk only when the page is replaced
- Dirty bit
  - 1 if the page has been written
  - 0 if the page never changed

TLB = Translation Lookaside Buffer

- Address translation is very expensive
  - Must translate virtual memory address on every memory access
  - Multilevel page table, each translation is several memory accesses
- Solution: TLB for address translation
  - Keep track of most common translations in the TLB
- TLB = Cache for address translation

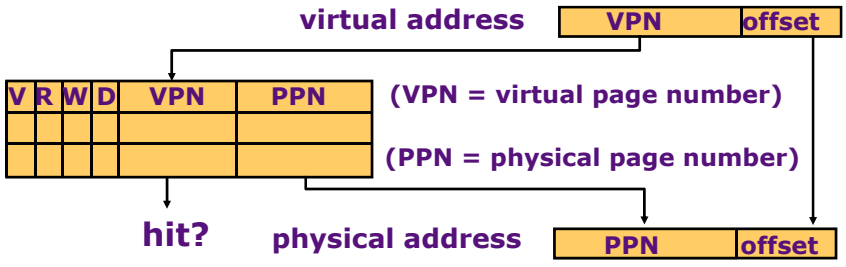


Translation Lookaside Buffer

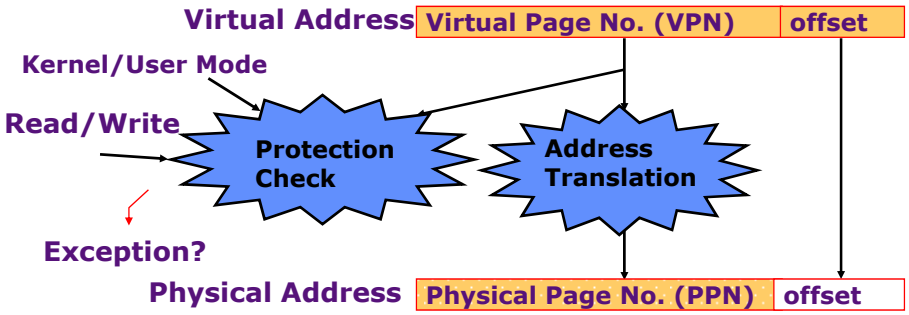
TLB hit: Fast single cycle translation

TLB miss: Slow page table translation

Must update TLB on a TLB miss



Address Translation & Protection



Every instruction and data access needs address translation and protection checks

Check whether page is read only, writable, or executable

Check whether it can be accessed by the user or kernel only

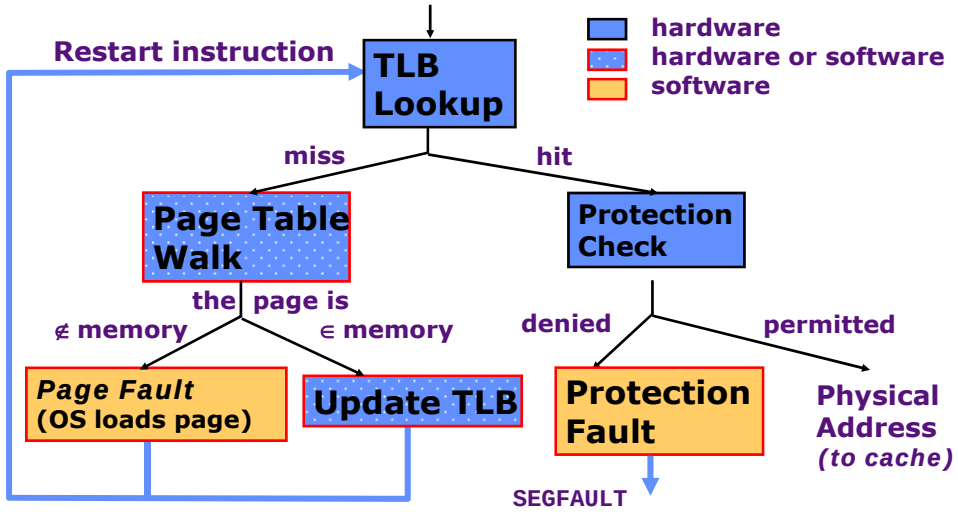
Handling TLB Misses and Page Faults

- TLB miss: No entry in the TLB matches a virtual address
- TLB miss can be handled in software or in hardware
  - Lookup page table entry in memory to bring into the TLB
  - If page table entry is valid then retrieve entry into the TLB
  - If page table entry is invalid then page fault (page is not in memory)
- Handling a Page Fault
  - Interrupt the active process that caused the page fault
    - Program counter of instruction that caused page fault must be saved
    - Instruction causing page fault must not modify registers or memory
  - Transfer control to the operating system to transfer the page
  - Restart later the instruction that caused the page fault

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 addressing mode is used to access page tables
- 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. The page fault is handled by the OS software.

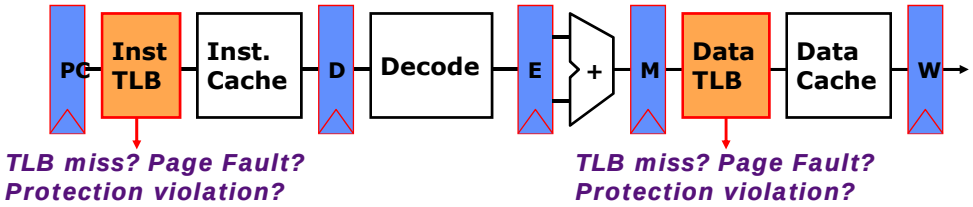
Address Translation Summary



TLB, Page Table, Cache Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data is not in cache
Miss	Miss	Miss	Yes – page fault (page is on disk)
Hit	Miss	Hit/Miss	Impossible – TLB translation is not possible if page is not in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

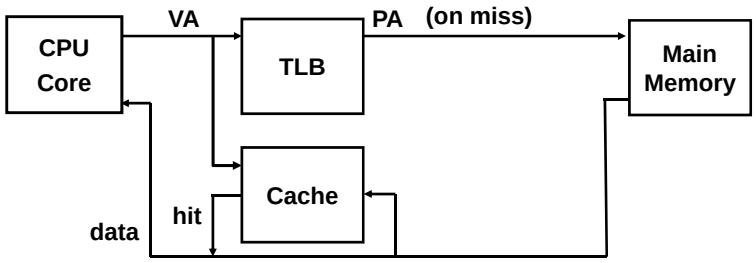
Address Translation in CPU Pipeline



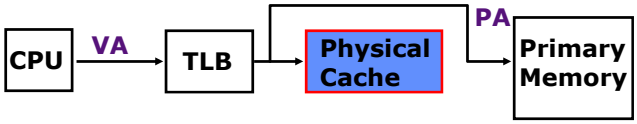
- Software handlers need restartable exception on page fault
- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB
  - Slow down the clock
  - Pipeline the TLB and cache access
  - Virtual address caches
  - Parallel TLB/cache access

Physical versus Virtual Caches

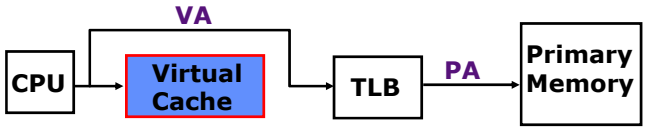
- Physical caches are addressed with physical addresses
  - Virtual addresses are generated by the CPU
  - Address translation is required, which may increase the hit time
- Virtual caches are addressed with virtual addresses
  - Address translation is not required for a hit (only for a miss)



Physical versus Virtual Caches



Alternative: place the cache before the TLB

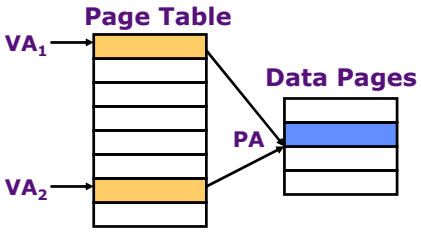


- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless process identifiers (PIDs) included in tags (-)
- Aliasing problems due to the sharing of pages (-)
- maintaining cache coherence (-)

Drawbacks of a Virtual Cache

- Protection bits must be associated with each cache block
  - Whether it is read-only or read-write
- Flushing the virtual cache on a context switch
  - To avoid mixing between virtual addresses of different processes
  - Can be avoided or reduced using a process identifier tag (PID)
- Aliases
  - Different virtual addresses map to same physical address
  - Sharing code (shared libraries) and data between processes
  - Copies of same block in a virtual cache
    - Updates makes duplicate blocks inconsistent
  - Can't happen in a physical cache

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: Disallow aliases to coexist in cache

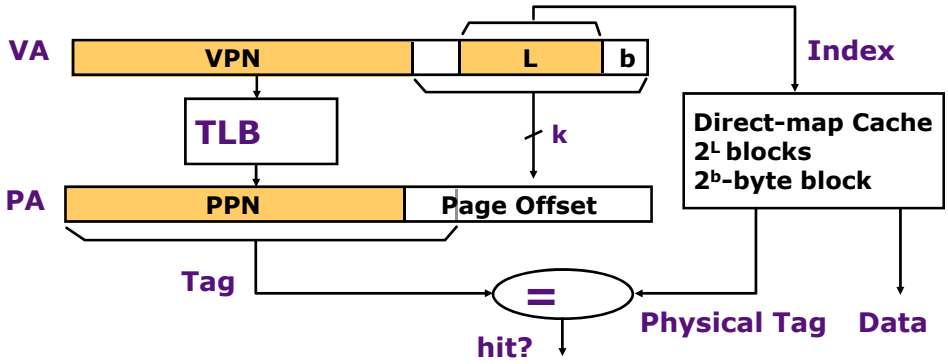
OS Software solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache

Address Translation during Indexing

- To lookup a cache, we can distinguish between two tasks
  - Indexing the cache – Physical or virtual address can be used
  - Comparing tags – Physical or virtual address can be used
- Virtual caches eliminate address translation for a hit
  - However, cause many problems (protection, flushing, and aliasing)
- Best combination for an L1 cache
  - Index the cache using virtual address
    - Address translation can start concurrently with indexing
    - The page offset is same in both virtual and physical address
    - Part of page offset can be used for indexing ⇒ limits cache size
  - Compare tags using physical address
    - Ensure that each cache block is given a unique physical address

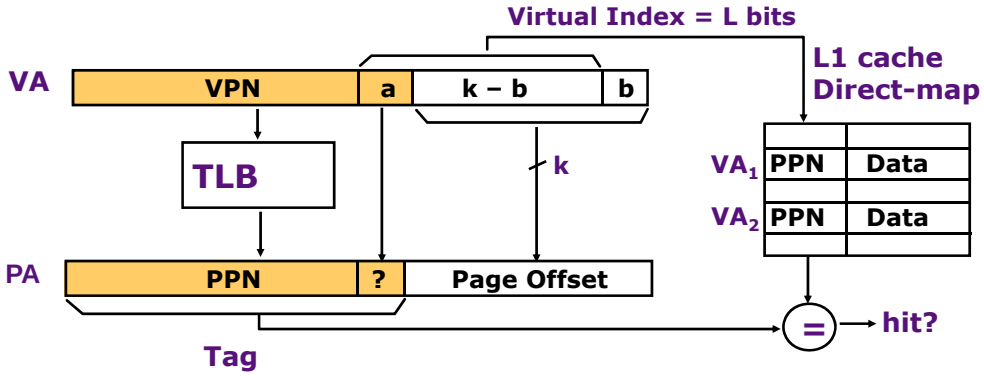
Concurrent Access to TLB & Cache



Index L is available without consulting the TLB
⇒ cache and TLB accesses can begin simultaneously
Tag comparison is made after both accesses are completed

Cases: L ≤ k-b, L > k-b (aliasing problem)

Problem with L1 Cache size > Page size



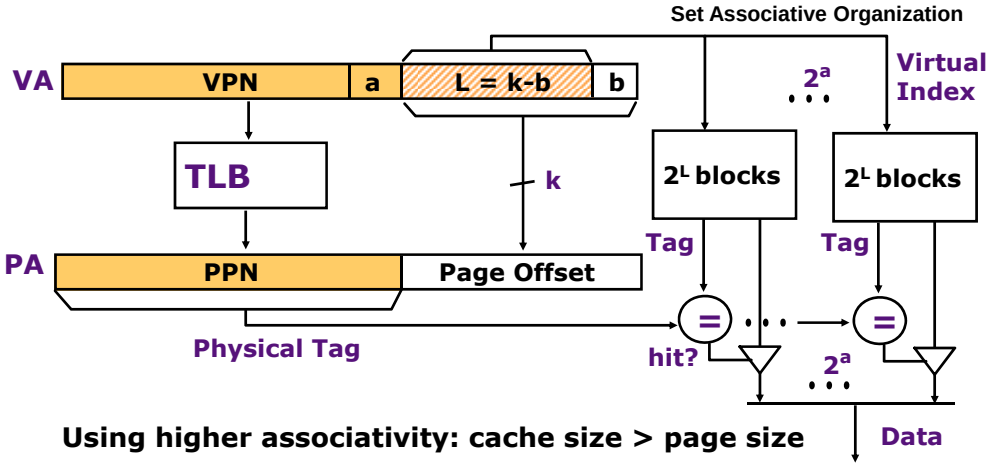
Virtual Index now uses the lower a bits of VPN

VA_1 and VA_2 can map to same PPN

Aliasing Problem: Index bits = L > k-b

Can the OS ensure that lower a bits of VPN are same in PPN?

Anti-Aliasing with Higher Associativity

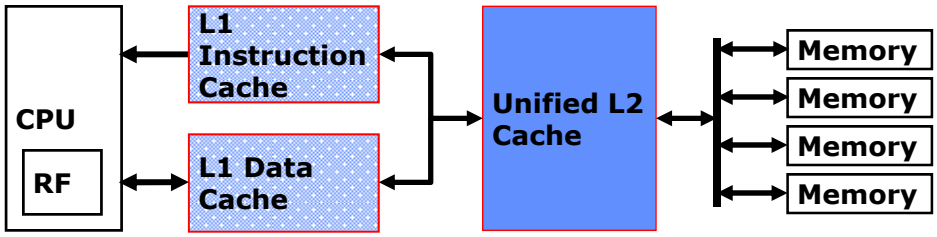


Using higher associativity: cache size > page size

2^a physical tags are compared in parallel

Cache size = 2^a x 2^L x 2^b > page size (2^k bytes)

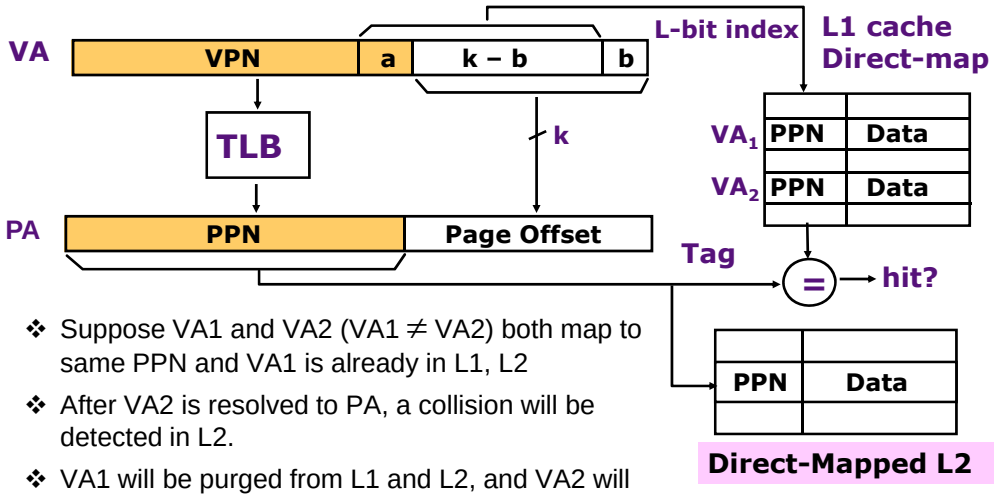
Anti-Aliasing via Second Level Cache



Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is typically "inclusive" of both Instruction and Data caches

Anti-Aliasing Using L2: MIPS R10000



- Suppose VA1 and VA2 (VA1 ≠ VA2) both map to same PPN and VA1 is already in L1, L2
- After VA2 is resolved to PA, a collision will be detected in L2.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ no aliasing !

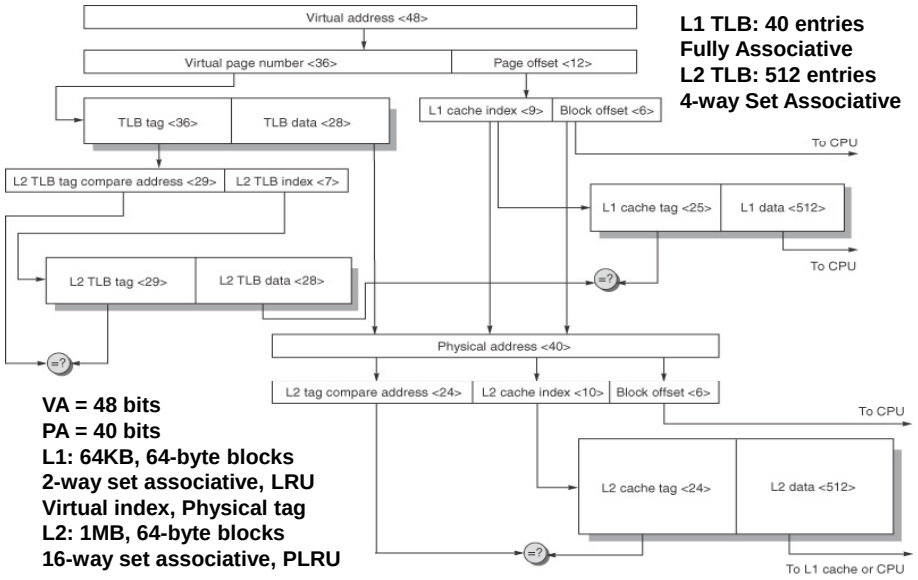
TLB Organization

- TLB keeps track of recently accessed pages
  - Virtual and Physical page numbers
  - Valid, Dirty, and Reference bits
  - Access bits: whether page is read-only or read-write
  - PID: process ID – which process is currently using TLB entry
- Some Typical Values for a TLB
  - TLB size = 16 – 512 entries
  - Small TLBs are fully associative, while big ones are set-associative
  - Hit time = 0.5 – 1 clock cycle
  - TLB Miss Penalty = 10 – 100s clock cycles
  - Miss Rate = 0.01% – 1%

Examples on TLB Parameters

Intel P4	AMD Opteron
1 TLB for instructions	2 TLBs for instructions (L1 and L2)
1 TLB for data	2 TLBs for data (L1 and L2)
Both TLBs are 4-way set associative	Both L1 TLBs are fully associative
Both use ~LRU replacement	Both L1 TLBs have 40 entries
Both have 128 entries	Both L1 TLBs use ~LRU replacement
TLB misses are handled in hardware	Both L2 TLBs are 4-way set associative
	Both L2 TLBs have 512 entries
	Both L2 LTBs use round-robin LRU
	TBL misses are handled in hardware

Putting It All Together: AMD Opteron

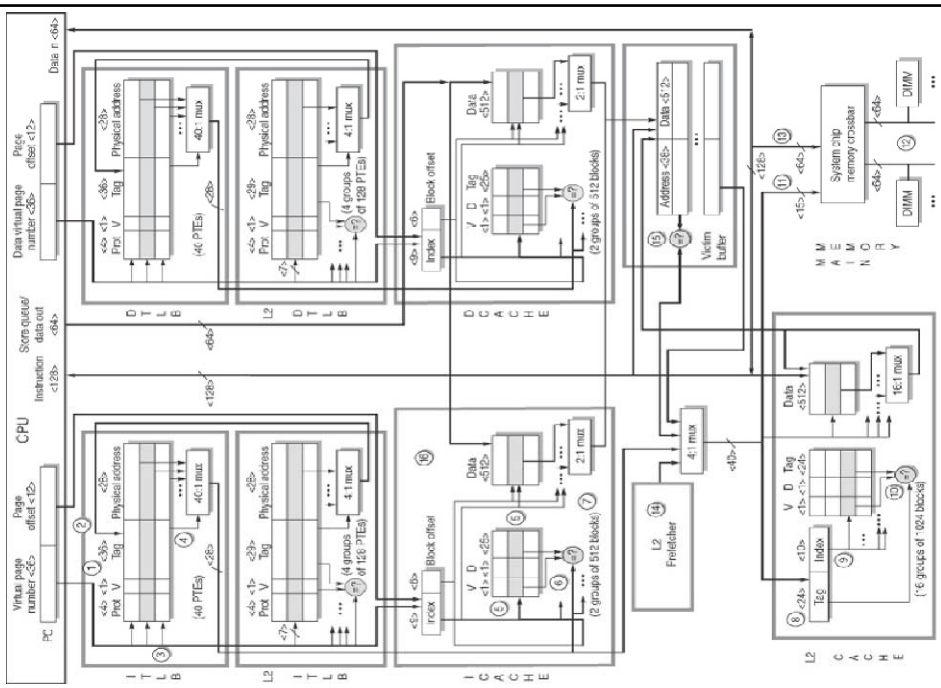


L1 TLB: 40 entries
Fully Associative
L2 TLB: 512 entries
4-way Set Associative

VA = 48 bits
PA = 40 bits
L1: 64KB, 64-byte blocks
2-way set associative, LRU
Virtual index, Physical tag
L2: 1MB, 64-byte blocks
16-way set associative, PLRU

AMD Opteron Memory Hierarchy

- AMD Opteron has an exclusion policy between L1 and L2
  - Block can exist in L1 or L2 but not in both
  - Better use of cache resources
  - Both the D-cache and L2 use write-back with write-miss allocation
- L1 cache is pipelined, latency of hit is 2 clock cycles
- Miss in L1 goes to L2 and to memory controller
  - Lower the miss penalty in case the L2 cache misses
- L1 cache is virtually indexed and physically tagged
- On a miss, cache controller must check for aliases in L1
  - $2^3 = 8$ L1 cache tags per way are examined for aliases in parallel during an L2 cache lookup.
  - If it finds an alias, the offending block is invalidated.
- Victim Buffer: used when replacing modified blocks


CPU
Virtual page number <36>
Page offset <12>
PC
Instruction <128>
Store queue/ data out <64>
Data virtual page number <36>
Page offset <12>
Data in <64>
Physical address
40:1 mux
4:1 mux
2:1 mux
Block offset
Index
Tag
Data <512>
Address <36>
Victim buffer
L2 Prefetcher
16:1 mux
System chip memory crossbar
DIMM
ITLB
DTLB
L2 ITLB
L2 DTLB
ICACHE
DCACHE
L2 CACHE
MEMORY