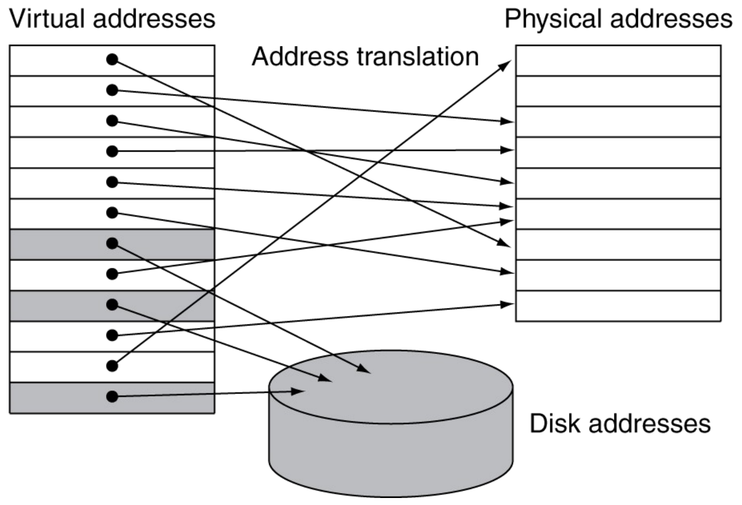
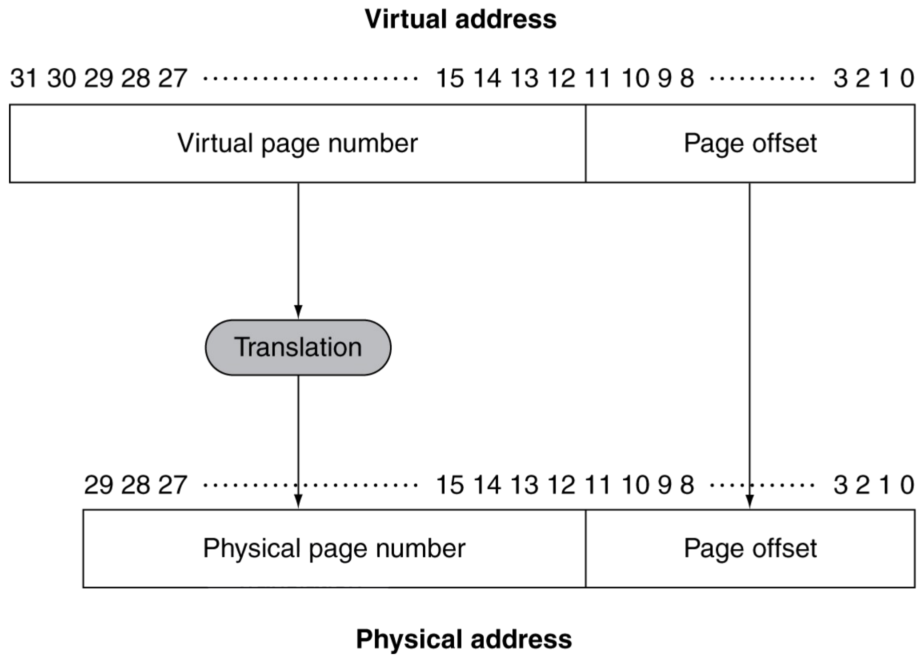
Virtual Memory

- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM "block" is called a page
  - VM translation "miss" is called a

Address Translation

- Fixed-size pages (e.g., 4K)





-

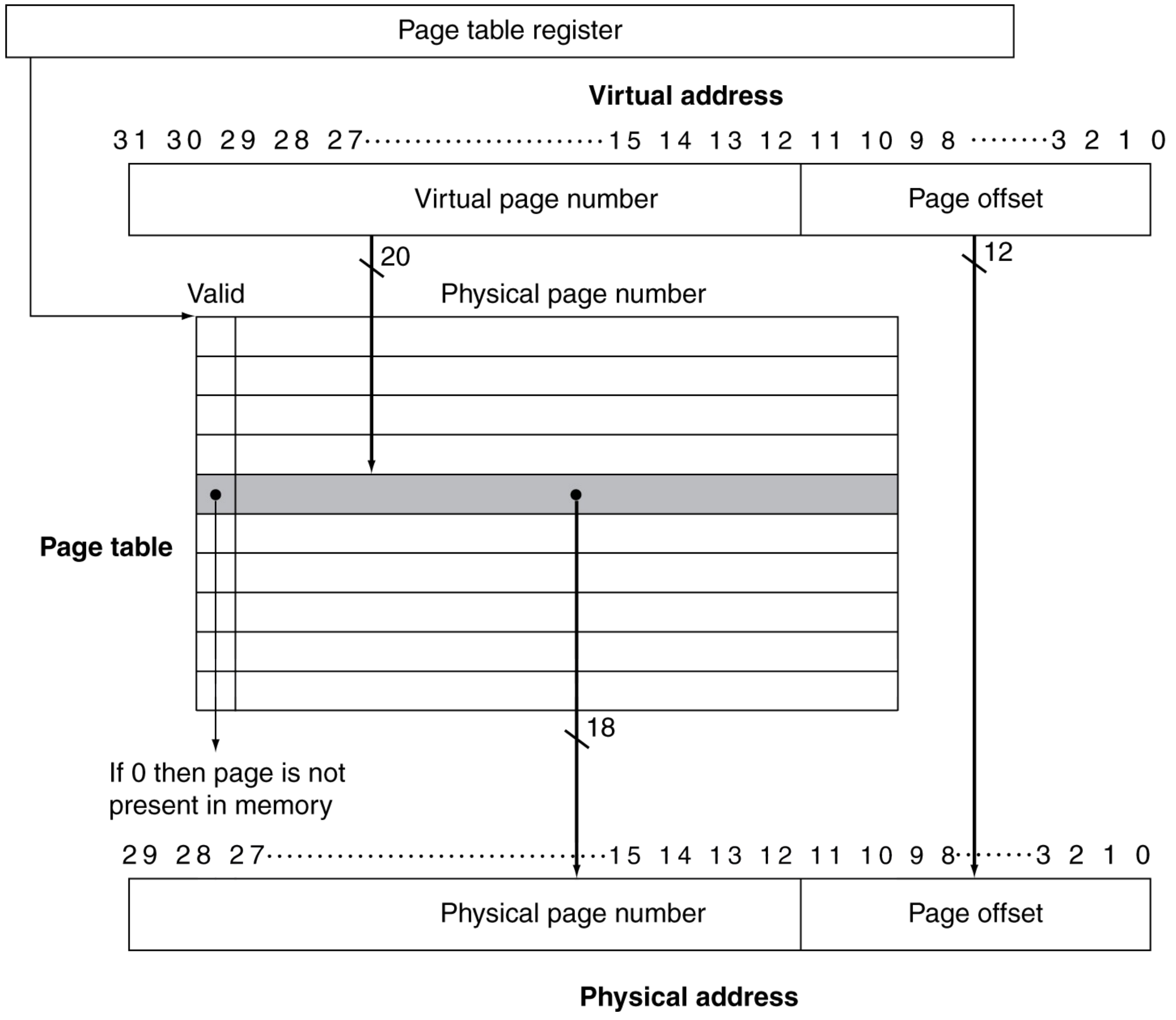
Page Tables

- Stores placement information
  - Array of page table entries (PTEs), 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
  -

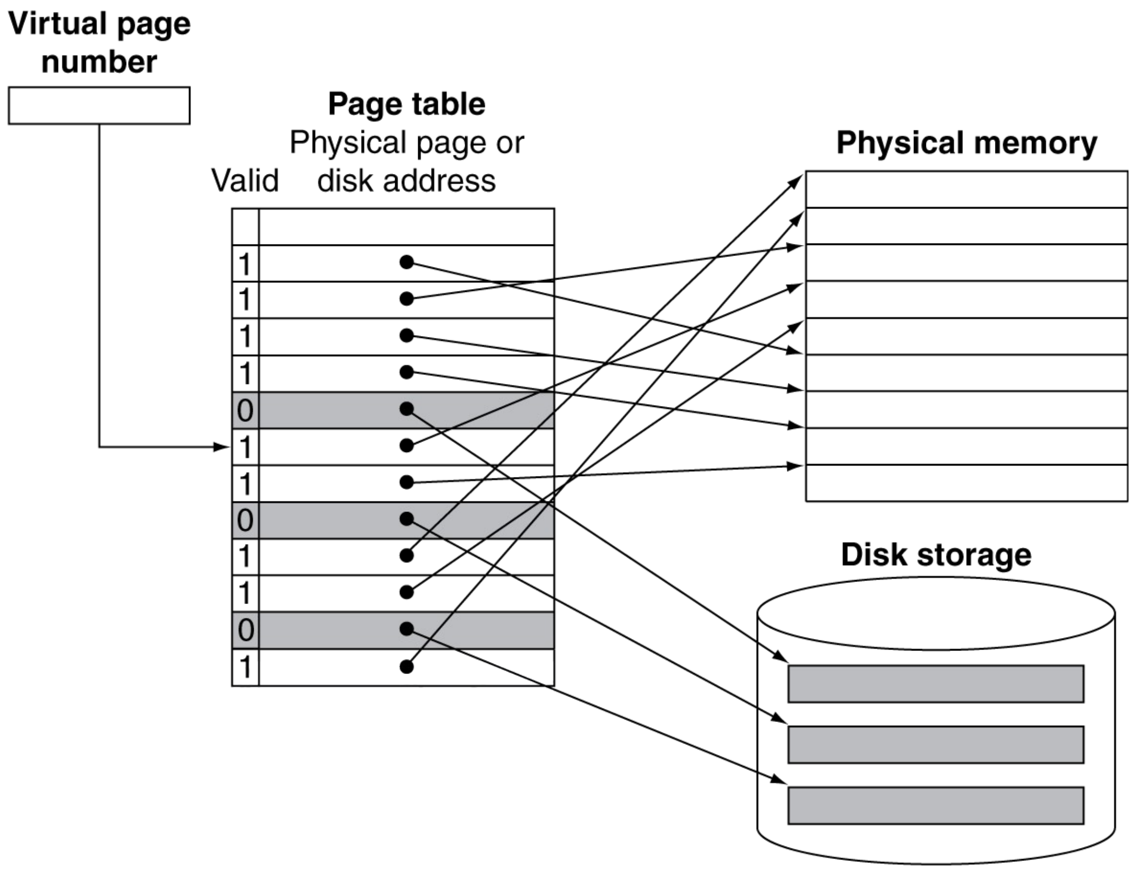
Page Fault Penalty

- 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
  -

Translation Using a Page Table



Mapping Pages to Storage



Segmentation

- Paging: fixed size blocks
- Segmentation: variable size blocks
  - Address consists of two parts: segment address (mapped to physical address) and a segment offset
  - Bounds check required to ensure offset within segment
  - (+) Enables more powerful methods of protection, sharing
  - (-) Splits address space into logically separate pieces that must be manipulated as a 2-part address
    - Paging makes boundary between page number and offset invisible to programmers and compilers

Page Table Size Calculation

- 32-bit virtual address space
- 4 KB pages, 4 bytes/PTE
- What is the total page table size?

- No. of page table entries = $2^{32}/2^{12} = 2^{20}$
- Size of page table = 2^{20} PTE x 2^{2} bytes/PTE = 4 MB per program in execution at any given time
- What if there are hundreds of programs in execution?
- How can we handle 64-bit addresses that would need 2^{52}

Techniques to Minimize Page Table Size

- Limit register: restricts size of page table for a process
  - If virtual page number becomes larger than contents of limit register, add entries to page table (i.e., page tables grows)
- Use hashing function for virtual address to limit page table size to number of physical pages in main memory
  - Inverted page table; more complex lookup procedure
- Multiple levels of page tables (typically 2-level)
  - First level maps large fixed size blocks of virtual address space 64 to 256 pages in total (segment table); Second level contains page tables; More complex address translation
- Paging the page tables
  - Page tables reside in virtual address space
  - Most modern systems allow this to reduce the actual main memory tied up in page tables

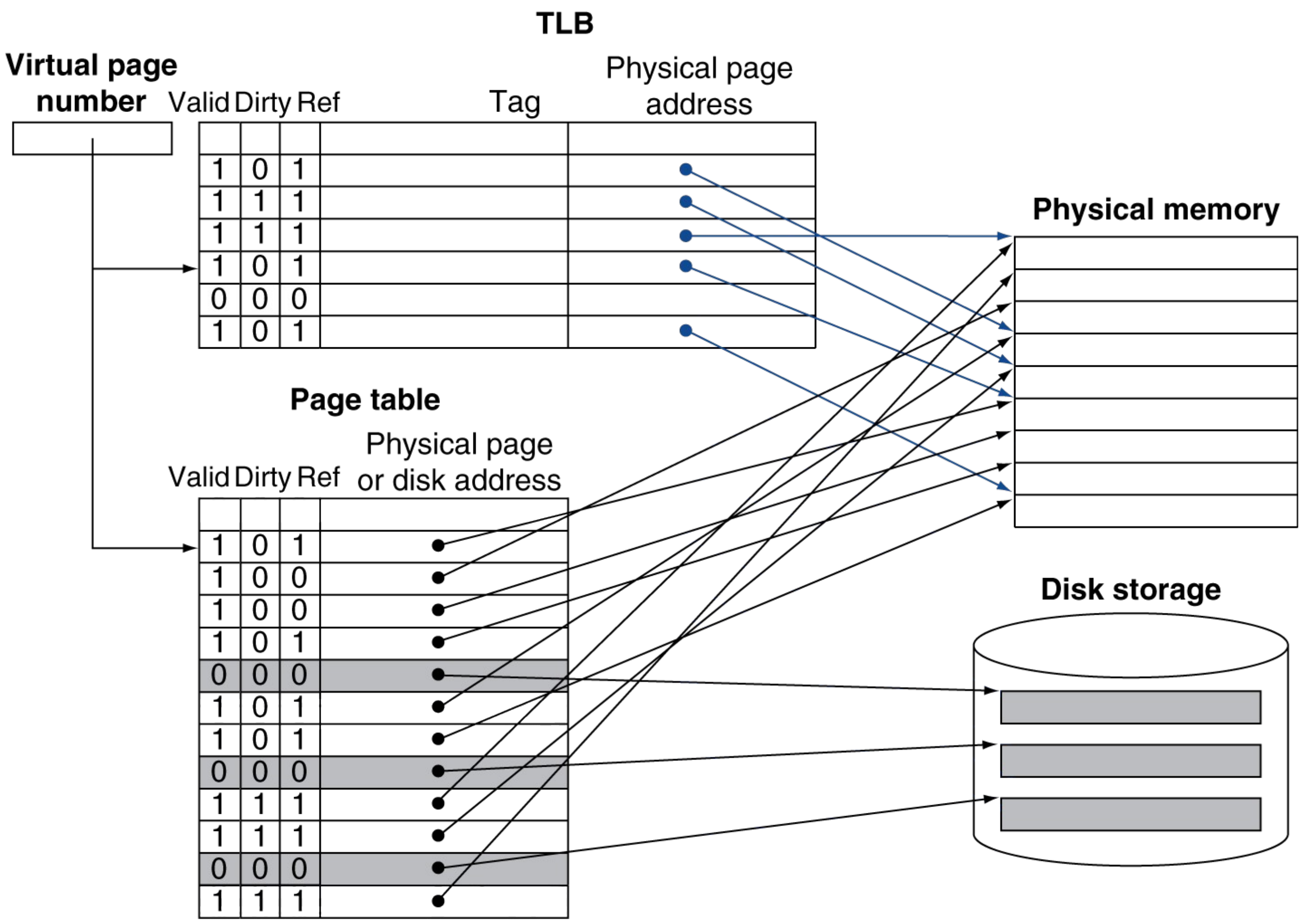
Replacement and Writes

- 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
  -

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
  - Small TLBs are typically fully associative, large TLBs have small associativity (due to cost issues)
  - Misses could be handled by hardware or software

Fast Translation Using a TLB



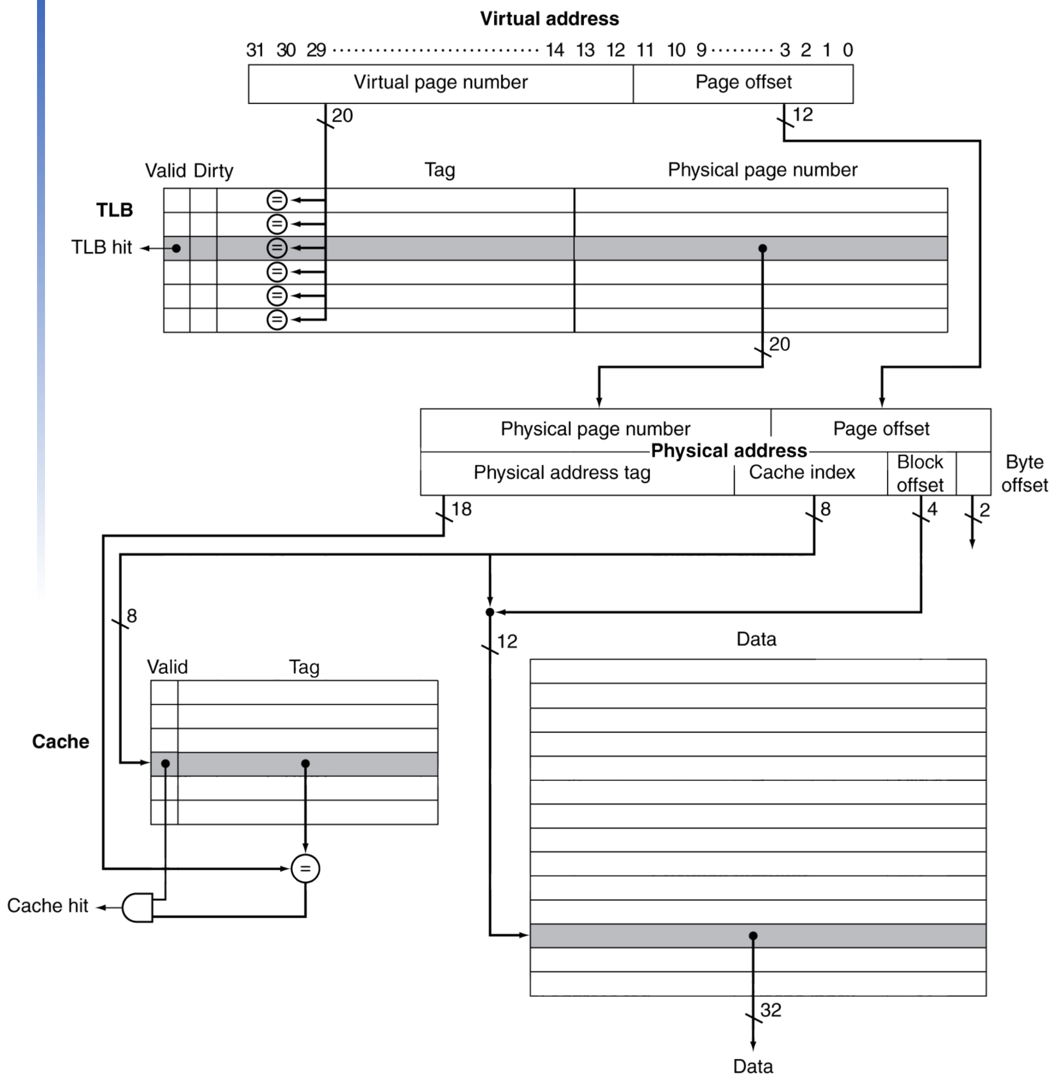
TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - 10's of cycles
  - Could be handled in hardware
    - Can get complex for more complex 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 page table
  - Then restart the faulting instruction
  - 1,000,000's of cycles

Page Fault Handler

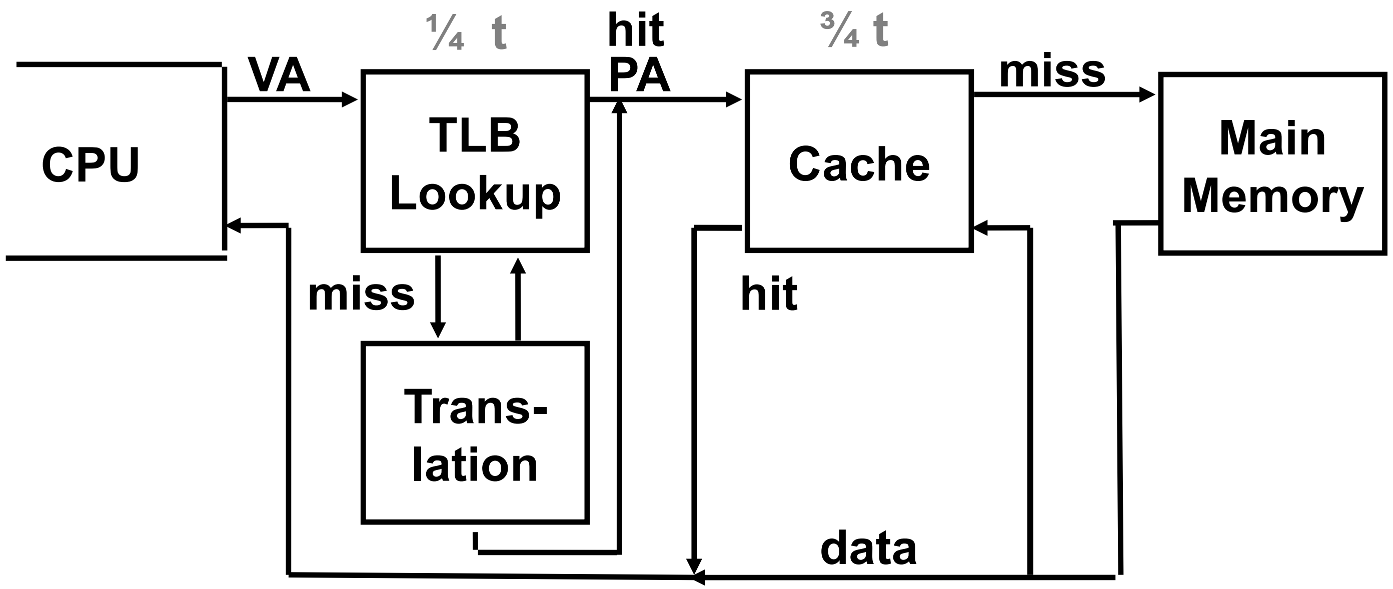
- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction Example



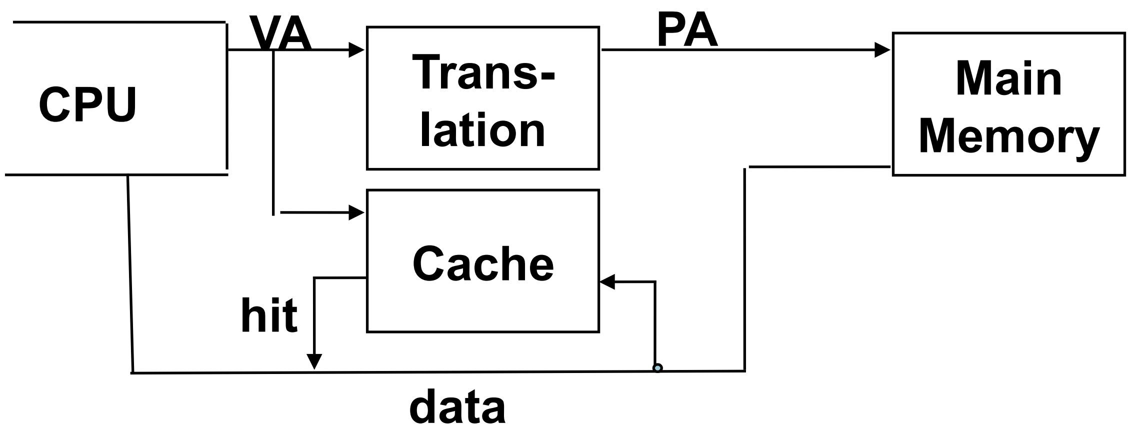
- Intrinsity FastMATH
- Fully associative TLB
- Concatenation avoids 16:1 multiplexor
- Direct mapped cache
  - Physically tagged and physically indexed

Cache Addressing



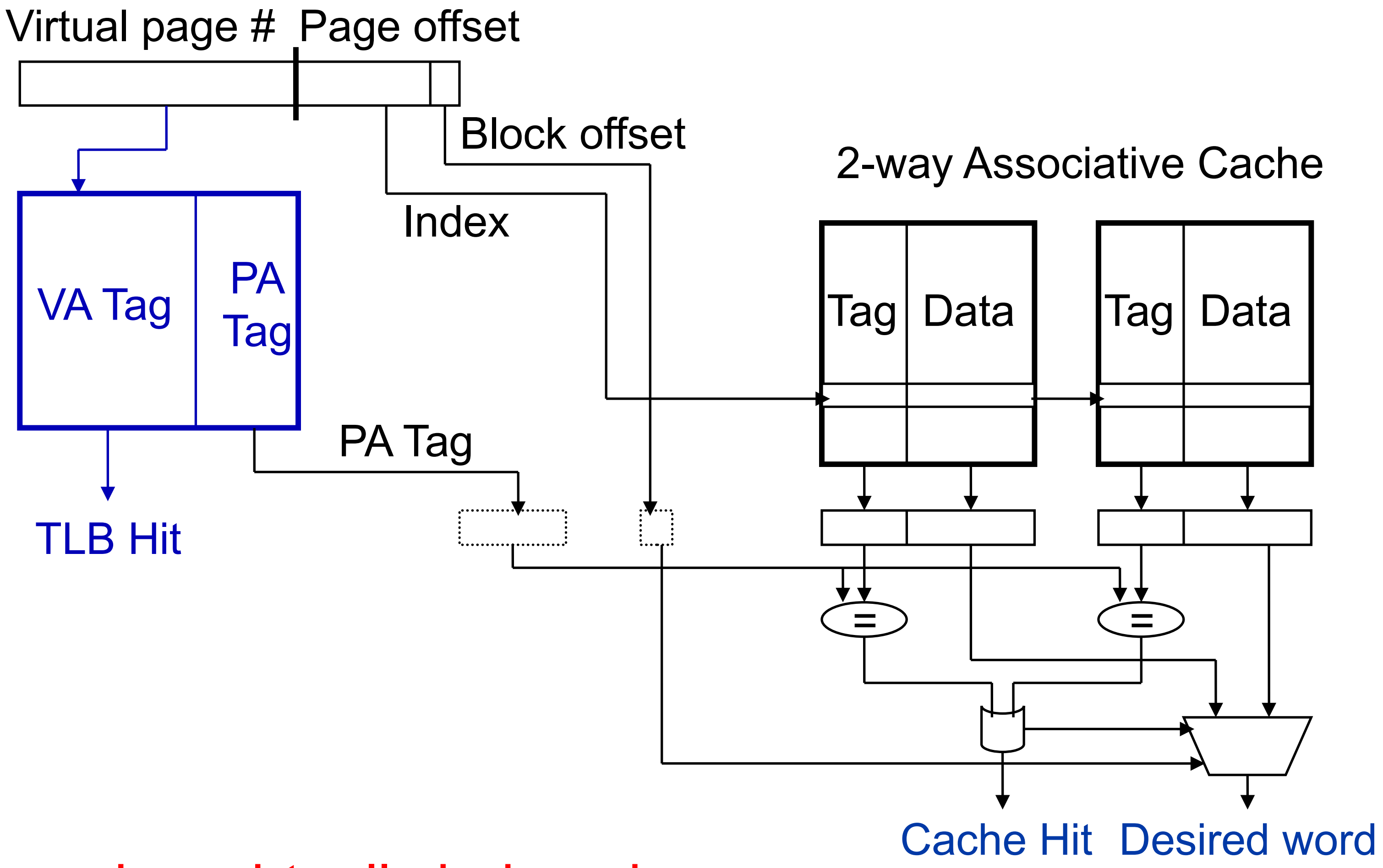
- Physically tagged and physically indexed caches
  - Need to translate address before cache lookup

Cache Addressing



- Virtually tagged and virtually indexed caches
  - TLB not in critical path and accessed only on cache miss to translate virtual address to physical main memory address
  - Complications due to aliasing
    - Different virtual addresses for shared physical address
      - stored in separate locations in cache
    - One program can write data without other program being aware that the data has changed – ambiguous!

Cache Addressing



- Compromise: virtually indexed physically tagged caches
  - Virtual page offset for cache indexing
    - Really a physical address as it is not translated

TLB, VM, Cache Event 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 not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if 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
  - Supervisor mode (aka kernel mode)
    - Runs privileged instructions not available in user mode
  - Page tables, TLBs and other state information only writeable in supervisor mode (read-only in user mode)
  - System call exception (e.g., syscall

TLB Miss Handling in MIPS

- Consider a TLB miss for a page that is present in memory (i.e., the Valid bit in page table is set)
  - MIPS handles TLB misses in software
  - A TLB miss (or a page fault exception) must be asserted by the end of the same clock cycle that the memory access occurs so that the next clock cycle will begin exception processing

MIPS Software TLB Miss Handler

- When a TLB miss occurs, the hardware saves the address that caused the miss in `BadVAddr` and transfers control to 8000 0000_{hex}, the location of the TLB miss handler

```
TLBmiss:
 mfc0   $k1, Context #copy addr of PTE into $k1
 lw     $k1, 0($k1)  #put PTE into $k1
 mtc0   $k1, EntryLo #put PTE into EntryLo
 tlbwr           #put EntryLo into TLB
                 #  at Random
 eret               #return from exception
```

- MIPS hardware places address of missing page in `Context`
- `tlbwr` copies from `EntryLo` into the TLB entry selected by the control register `Random`
- A TLB miss takes about a dozen clock cycles to handle