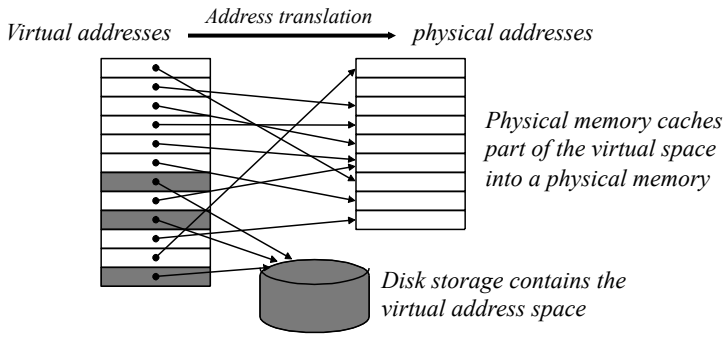
Virtual Memory

Virtual Memory

- Main memory is “cache” for secondary storage
- Secondary storage (disk) holds the complete “virtual address space”
- Only a portion of the virtual address space lives in the physical address space at any moment of time

Virtual Memory

- Main memory is a cache for secondary storage



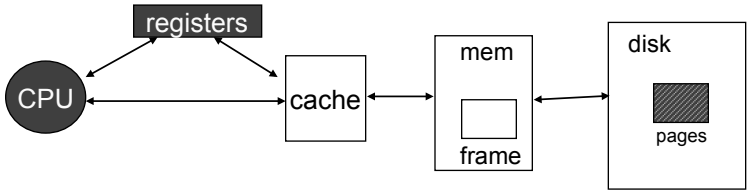
3

Advantages

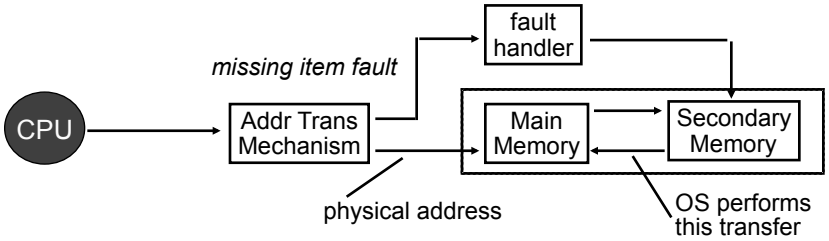
- Illusion of having more physical memory
  - Disk acts as the primary memory
  - Comes from the days of limited memory systems
- Multiple programs share the physical memory
  - Permit sharing without knowing other programs
  - Division of memory among programs is "automatic"
- Program relocation
  - Program addresses can be mapped to *any* physical location
  - Physical memory does *not* have to be contiguous
- Protection
  - Per process protection can be enforced on pages

4

Basic VM Issues



missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy*



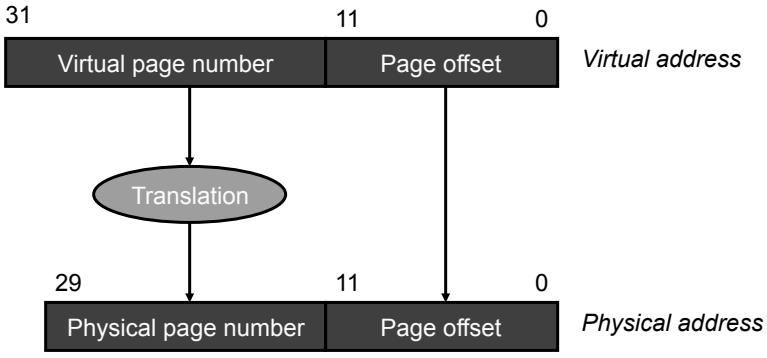
5

Pages: Virtual Memory Blocks

- Page faults: the data is not in memory, retrieve it from disk
  - Huge miss penalty (millions of cycles - disk access), thus pages should be fairly large (e.g., 4KB) to amoritize the high access time
  - Reducing page faults is important due to high access time
    - » LRU is worth the price, fully associative mapping
  - Can handle the faults in software instead of hardware
    - » the cost is in the disk access: so we have time to do more clever things in the OS
  - Use write-back because write-through is too expensive
    - » write-through not reasonable due to high cost of disk

6

Address Translation

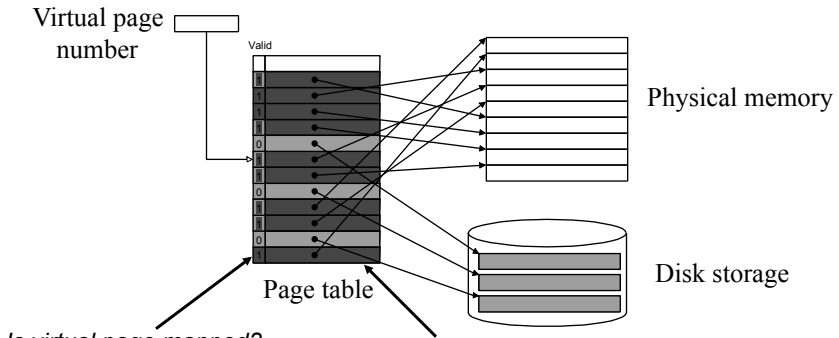


Full associativity (tag is the virtual page number)
Tag comparison is replaced by a table lookup
This example: 4GB virtual memory, 1GB physical memory, page size is 4KB (2^{12}), with 2^{18} physical pages.

7

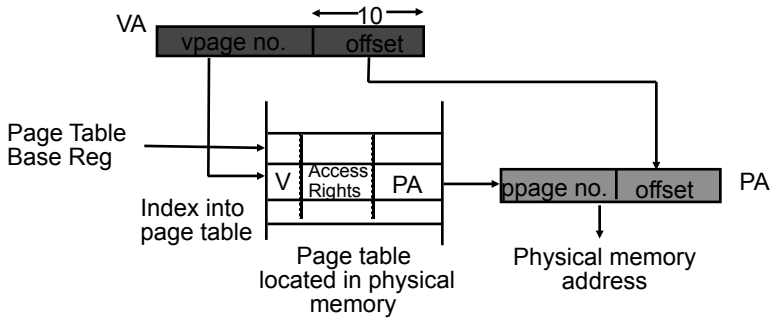
Page Tables

How do we know what's where? On disk? In memory?



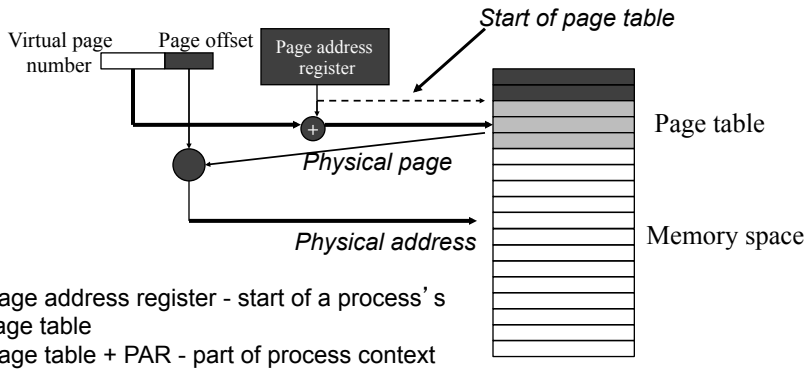
8

Page Tables for Address Translation



9

Page Tables



- Page address register - start of a process's page table
- Page table + PAR - part of process context
- Each memory reference requires two memory operations
- Page fault needs memory operation + disk access

11

Page Table Entries
(determined by architecture)

- Valid bit - has the page been loaded
- Read and write permissions - can the user program read and write to this page
- Dirty bit - has the physical page been written to and will need to be written back to disk when replaced
- Use bit - has the page been used recently
- Physical memory page - mapping of virtual page to physical page in memory
- Disk location - mapping of virtual page to virtual page on disk

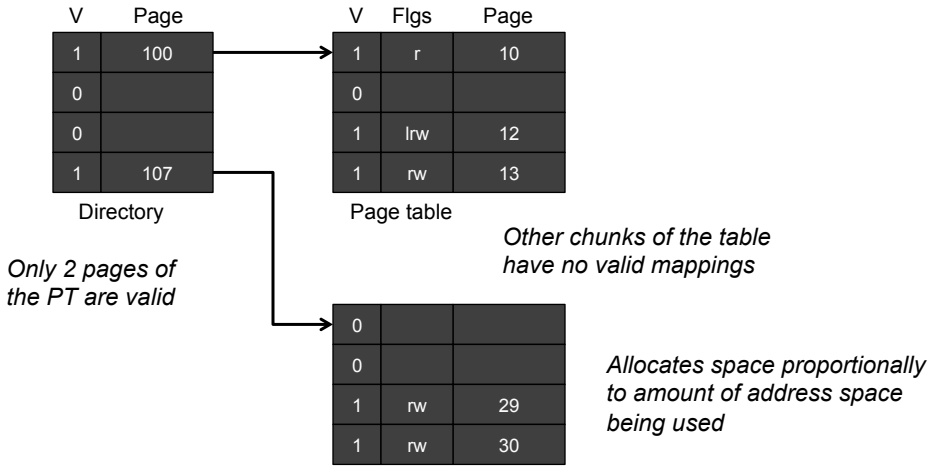
13

Multi-Level Page Tables

- PT (linear structure) can be very large!
  - 32-bit addr (2^{32} bytes), 4KB (2^{12} bytes) page, 4B PT entry
  - 1M entries, each 4 bytes = 4MB per page table
  - Hundreds of processes => Hundreds of MB for PT
- Turn PT into a tree (hierarchy) structure
  - Divide PT into page sized chunks
  - Hold only the part of PT where PT entries are valid
  - Directory points to portions of the PT
  - Directory says where to find PT, or that chunk is invalid

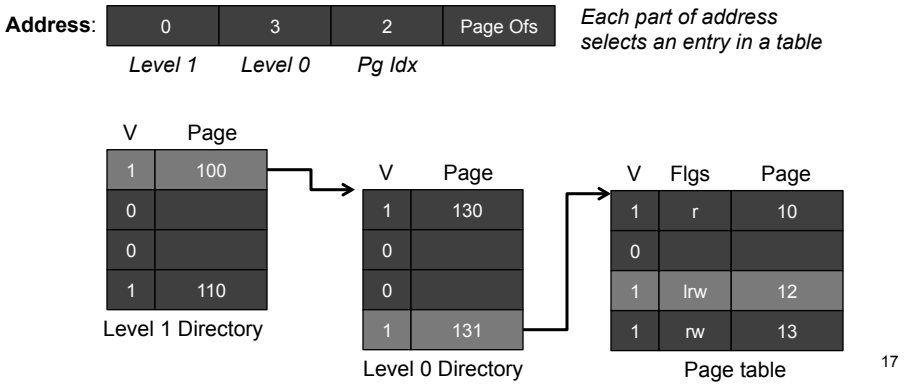
15

Multi-Level Page Tables

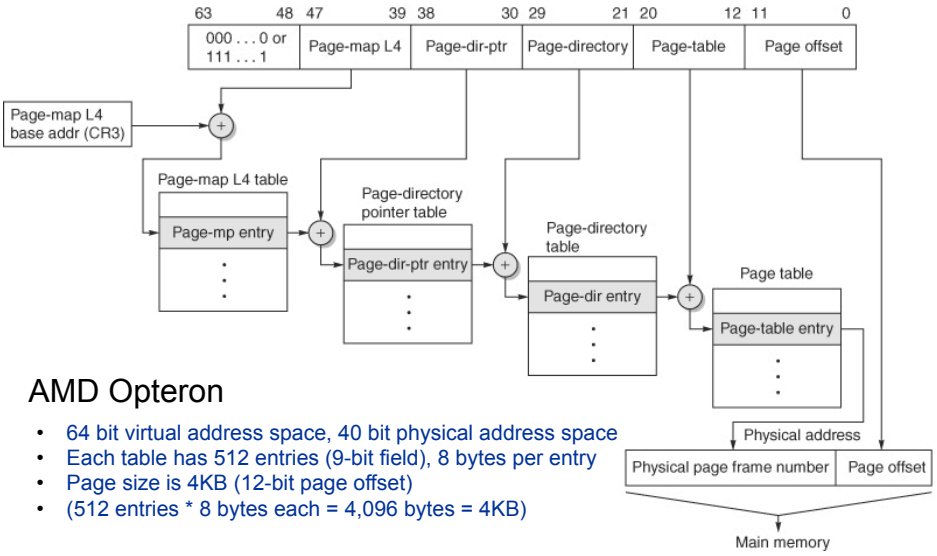


Multi-Level Page Tables

- What happens when we can't fit the page directory into a single page?
  - Divide up into a hierarchy (tree) of directories



Multi-Level Page Table



AMD Opteron

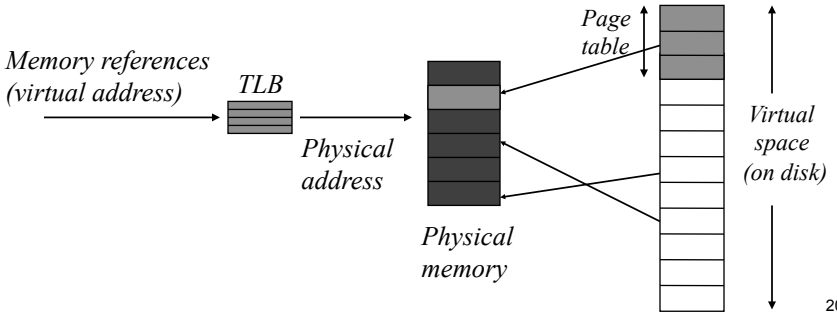
- 64 bit virtual address space, 40 bit physical address space
- Each table has 512 entries (9-bit field), 8 bytes per entry
- Page size is 4KB (12-bit page offset)
- (512 entries * 8 bytes each = 4,096 bytes = 4KB)

Page Size

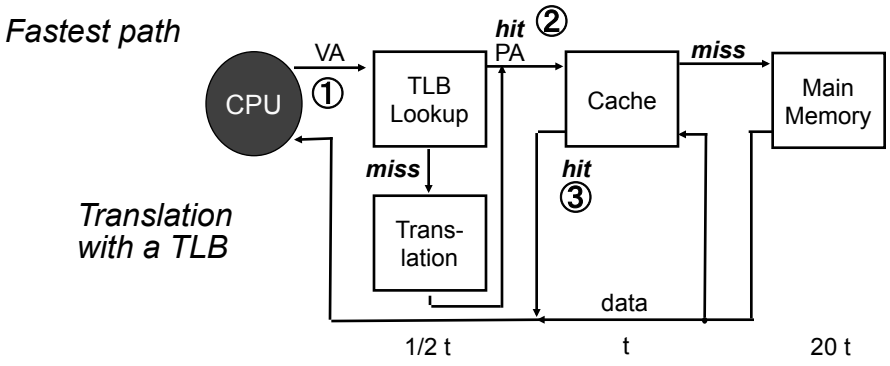
- Arguments for larger page size
  - Leads to a smaller page table
  - May be more efficient for disk access (block size of disk)
  - Larger page size - TLB entries capture more addresses per entry, so there are fewer misses, with the “right locality”
    - » TLB misses can be significant
  - x86 page sizes: 4KB, 2MB, 4MB, 1GB
- Arguments for smaller page size
  - Conserves storage space - less fragmentation

Translation Look-aside Buffer (TLB)

- Reduce memory reference time if we can store the page table in hardware
- Essentially, ***caching*** of the PT
  - **TLB Entry: Tag is virt. page and data is PTE for that tag**

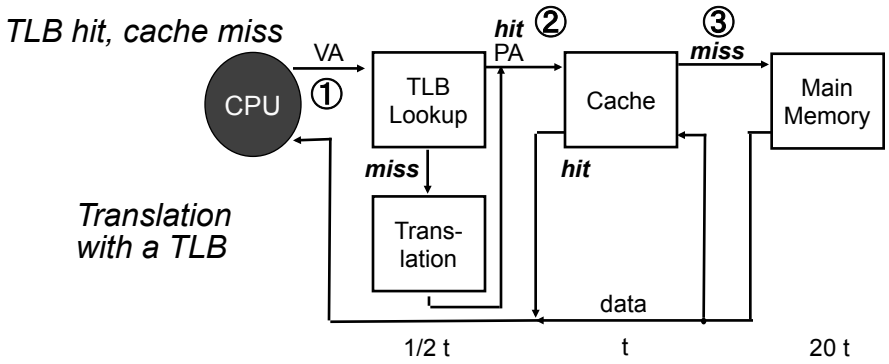


TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Overlap the cache access with the TLB access:high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

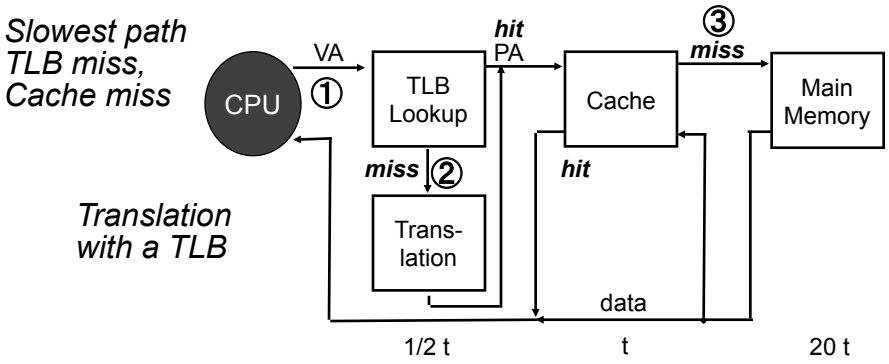
TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Overlap the cache access with the TLB access:high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

22

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Overlap the cache access with the TLB access:high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

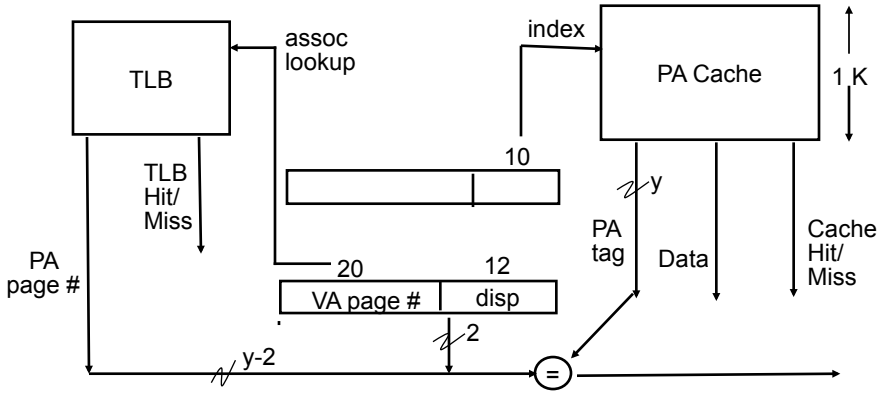
23

Translation Look-aside Buffers

- Relies on locality
  - If access has locality, then address translation has locality
  - The address translations are cached by the TLB
- One address translation maps a page worth of memory addresses, so the TLB can be small
  - From 32-256 entries & Usually fully associative
- Separate instruction and data TLBs
- Multi-level TLBs (I-TLB, D-TLB, L2-TLB)
- TLB Miss handling in HW or SW (PT walk)
- Entries may be tagged with process identifier to avoid flushing whole TLB on process switch

24

Overlapped Cache & TLB Access



IF *TLB hit* and *cache hit* and (*cache tag* = *PA*) then deliver data to CPU
ELSE IF *TLB hit* and (*cache miss* or *cache tag != PA*) THEN
access memory with the PA from the TLB
ELSE do standard VA translation

Limited to small caches, large page sizes, or high n-way set associative caches if you want a large cache

25

Protection

- Context switch
  - Save state needed to restart process when switched out for another process
- Process state needs to be protected from different processes
  - Can t write to disk: Too expensive
  - Keep state in memory for multiple processes at one time
- Protection needed so one process can't overwrite or access another process' state
  - Also, sharing code (libraries), data, interprocess communication, etc.

28

Protection

- Address ranges
  - *Base address* register
  - *Bound address* register
  - Valid address: Base register <= Address <= Bound register
- User processes can't change base or bound registers
  - OS changes registers on a context switch
- Requires distinguishing between user and OS code - *user* and *kernel* modes

29

Protection CPU Mechanisms

- Kernel and user mode to indicate what is running
- CPU state that can be read by user but not written; e.g., user and kernel mode bit, base/bound registers, exception enable/disable
- Mechanisms to go between modes
  - System call: TRAP or similar causes transfer to kernel mode and a call into the OS
  - System return: when returning from TRAP, transfer back to user mode

30

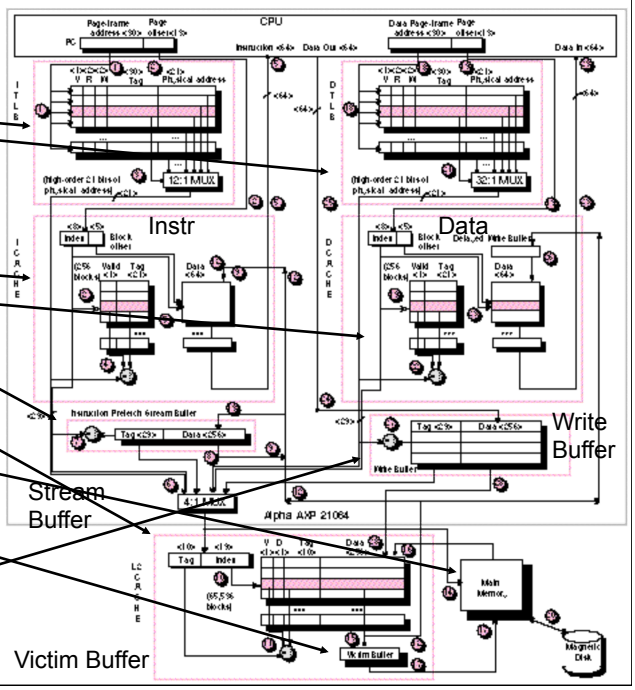
Protection with VM

- Virtual memory - protection on a per page basis
- Read/write permissions - text pages may be marked read-only
- User/kernel permissions - pages can be written only by the kernel (e.g., page table!)
  - Page tables are protected and can t be overwritten by other processes (OS ensures)
- Requires read/write and user/kernel bits maintained by the CPU

31

Alpha 21064

- Separate Instr & Data TLB
- TLBs fully associative
- TLB updates in SW ("Priv Arch Libr")
- Caches 8KB direct mapped, write thru
- Critical 8 bytes first
- Prefetch instr. stream buffer
- 2 MB L2 cache, direct mapped, WB (off-chip)
- 256 bit path to main memory, 4 x 64-bit modules
- Victim Buffer: to give read priority over write
- 4 entry write buffer between D$ & L2$



Instruction access

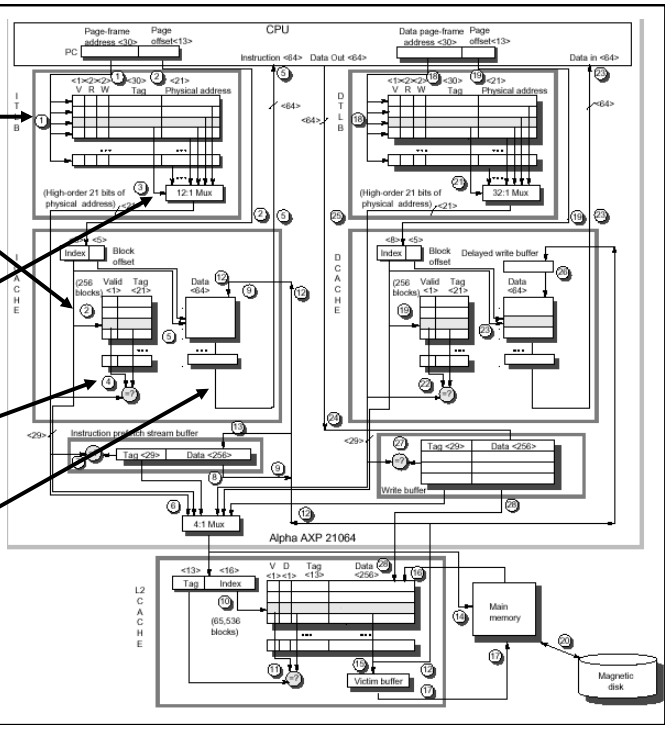
Step 1:
Virtual page sent to TLB

Step 2:
Page offset to L0 cache

Step 3:
TLB searched (12)
Translate address

Step 4:
Translated address matches cache tag

Step 5:
Send 8 bytes to CPU



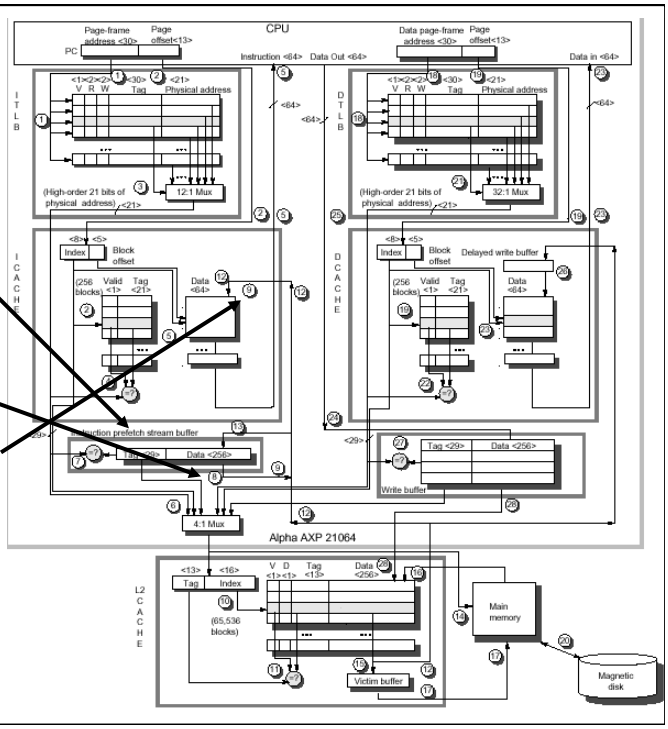
Instruction access

L0 cache miss
Step 6:
L2 accessed

Step 7:
Check prefetch buffer

Step 8:
Prefetch buffer hit, send 8 bytes to CPU

Step 9:
Full buffer written to cache



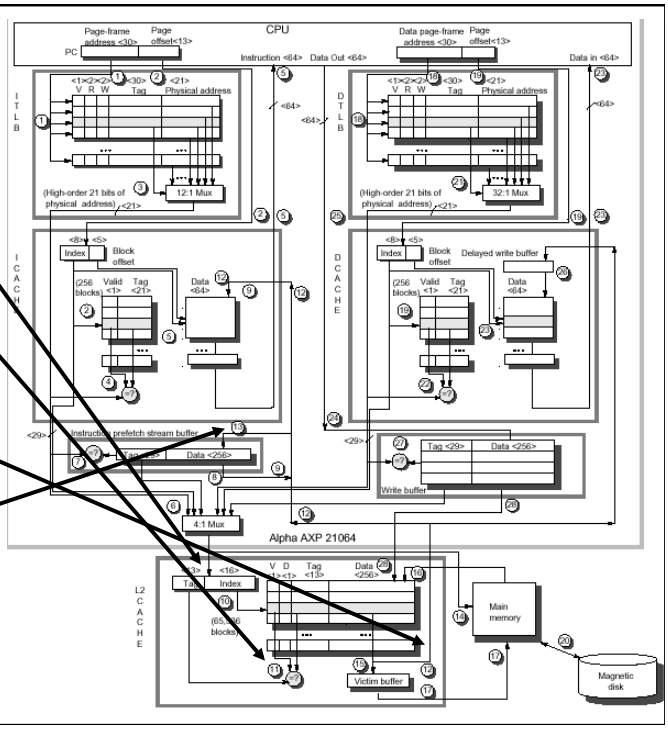
Instruction access

L0 cache miss - no hit in prefetch buffer
Step 10:
Index and tag for L2

Step 11:
Check cache hit

Step 12:
Return critical 16 bytes to CPU first , followed by the next 16 bytes

Step 13:
Next sequential line is requested and loaded into the stream buffer



Instruction access

L0 cache miss - no hit in prefetch buffer - L2 miss

Step 14:
Send address to memory

Step 15:
When replacing a dirty line, put it into the victim buffer so it can be written later

Step 16:
New data loaded into the cache

Step 17:
Old data written from victim buffer