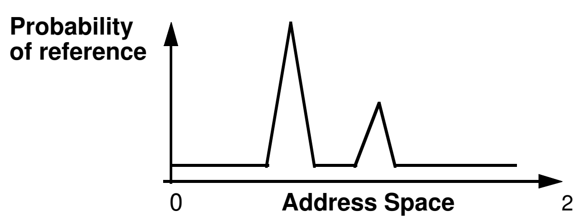
CS152
Computer Architecture and Engineering
Lecture 18: Virtual Memory

March 22, 1995

Dave Patterson (patterson@cs) and
Shing Kong (shing.kong@eng.sun.com)

Slides available on http://http.cs.berkeley.edu/~patterson

Review: The Principle of Locality



- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
  - Example: 90% of time in 10% of the code
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.

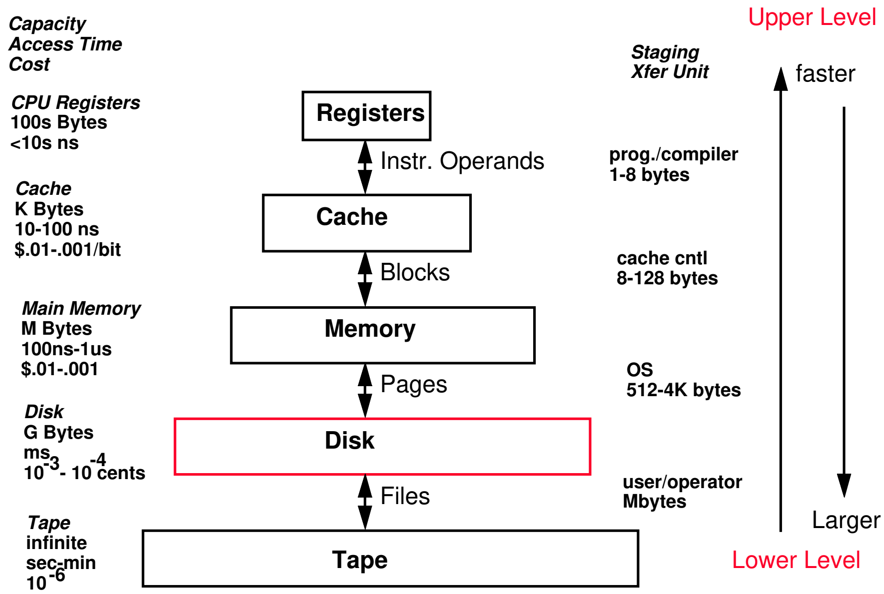
Review: The Need to Make a Decision!

- Direct Mapped Cache:
  - Each memory location can only mapped to 1 cache location
  - No need to make any decision :-)
    - Current item replaced the previous item in that cache location
- N-way Set Associative Cache:
  - Each memory location have a choice of N cache locations
- Fully Associative Cache:
  - Each memory location can be placed in ANY cache location
- Cache miss in a N-way Set Associative or Fully Associative Cache:
  - Bring in new block from memory
  - Throw out a cache block to make room for the new block
  - Damn! We need to make a decision which block to throw out!

Review Summary:

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  - Capacity Misses: increase cache size
- Write Policy:
  - Write Through: need a write buffer. Nightmare: WB saturation
  - Write Back: control can be complex

Review: Levels of the Memory Hierarchy



Outline of Today's Lecture

- **Recap of Memory Hierarchy & Introduction to Cache (5 min)**
- **Virtual Memory**
- **Questions and Administrative Matters (3 min)**
- **Page Tables and TLB (25 min)**
- **Break (5 minutes)**
- **Protection (20 min)**
- **Summary (5 min)**

Virtual Memory

Provides *illusion* of very large memory
– sum of the memory of many jobs greater than physical memory
– address space of each job larger than physical memory

Allows available (fast and expensive) physical memory to be very well utilized

Simplifies memory management (*main reason today***)**

Exploits memory hierarchy to keep average access time low.

Involves at least two storage levels: *main* and *secondary*

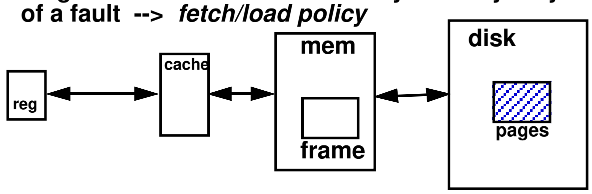
Virtual Address -- **address used by the programmer**

Virtual Address Space -- **collection of such addresses**

Memory Address -- **address of word in physical memory also known as "physical address" or "real address"**

Basic Issues in VM System Design

- **size of information blocks that are transferred from secondary to main storage**
- **block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy***
- **which region of M is to hold the new block --> *placement policy***
- **missing item fetched from secondary memory only on the occurrence of a fault --> *fetch/load policy***



Paging Organization

virtual and physical address space partitioned into blocks of equal size

page frames

pages

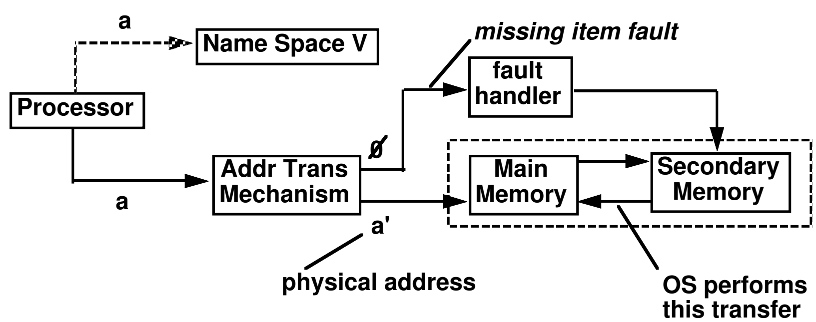
Address Map

V = {0, 1, . . . , n - 1} virtual address space
M = {0, 1, . . . , m - 1} physical address space n > m

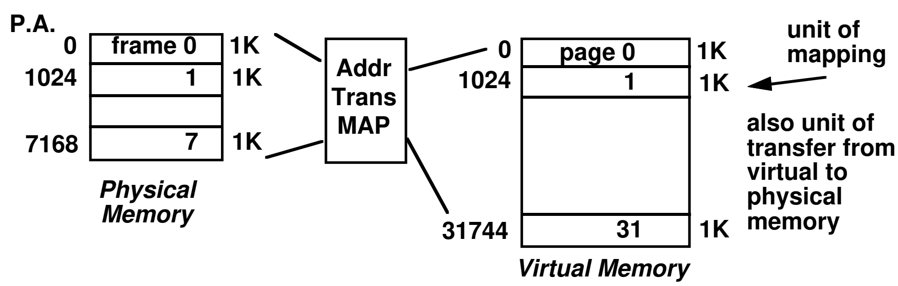
MAP: V --> M U {∅} address mapping function

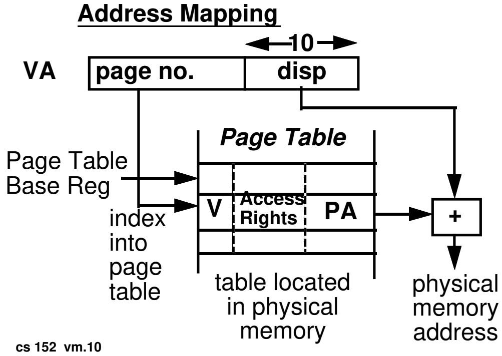
MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

= ∅ if data at virtual address a is not present in M



Paging Organization





actually, concatenation is more likely

Address Mapping Algorithm

If V = 1
then page is in main memory at frame address stored in table
else address located page in secondary memory

Access Rights
R = Read-only, R/W = read/write, X = execute only

If kind of access not compatible with specified access rights,
then *protection_violation_fault*

If valid bit not set then *page fault*

Protection Fault: access rights violation; causes trap to hardware, microcode, or software fault handler

Page Fault: page not resident in physical memory, also causes a trap; usually accompanied by a *context switch:* current process suspended while page is fetched from secondary storage

e.g., VAX 11/780
each process sees a 4 gigabyte (2^{32} bytes) virtual addr space
1/2 for user regions, 1/2 for a system wide name space shared by all processes

page size is 512 bytes

Optimal Page Size

Choose page that minimizes fragmentation

large page size => internal fragmentation more severe
BUT increases the # of pages / name space => larger page tables

In general, the trend is towards larger page sizes because

-- memories get larger as the price of RAM drops

-- the gap between processor speed and disk speed grow wider

-- programmers desire larger virtual address spaces

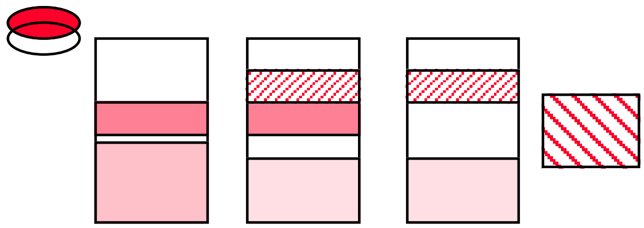
Most machines at 4K byte pages today, with page sizes likely to increase

Fragmentation & Relocation

Fragmentation is when areas of memory space become unavailable for some reason

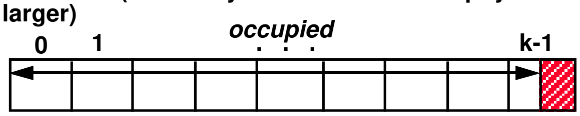
***Relocation*:** move program or data to a new region of the address space (possibly fixing all the pointers)

***External Fragmentation*: Space left between blocks.**



Internal Fragmentation:

program is not an integral # of pages, part of the last page frame is "wasted" (obviously less of an issue as physical memories get larger)



Fragmentation (cont.)

Table Fragmentation **occurs when page tables become very large because of large virtual address spaces; direct mapped page tables could take up sizable chunk of memory**

EX: VAX Architecture

	21	9
XX	**Page Number**	**Disp**

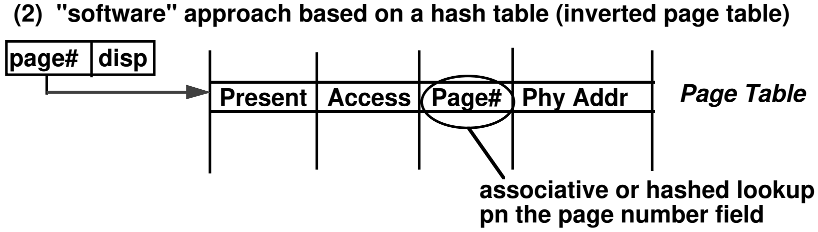
00 P0 region of user process
01 P1 region of user process
10 system name space

NOTE: this implies that page table could require up to 2 ^21 entries, each on the order of 4 bytes long (8 M Bytes)

Alternatives:

(1) Hardware associative mapping:
requires one entry per page frame (O(|M|)) rather than per page (O(|N|))

(2) "software" approach based on a hash table (inverted page table)



Questions and Administrative Matters (5 Minutes)

- No lecture next Friday March 24 (belated President's Day)
- Exercises due by Friday vs. Tuesday

Page Replacement Algorithms

Just like cache block replacement!

First-in/First-Out:

-- in response to page fault, replace the page that has been in memory for the longest period of time

-- does not make use of the principle of locality: an old but frequently referenced page could be replaced

-- easy to implement: maintain history thread thru page table entries, no need to track past reference history

-- usually exhibits the worst behavior!

Least Recently Used:

-- selects the least recently used page for replacement

-- requires knowledge about past references, more difficult to implement (thread thru page table entries from most recently referenced to least recently referenced; when a page is referenced it is placed at the head of the list; the end of the list is the page to replace)

-- good performance, recognizes principle of locality

Page Replacement (Continued)

Not Recently Used:

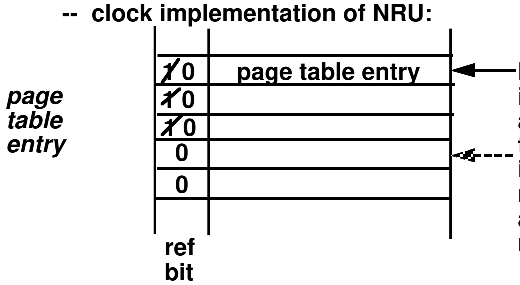
Associated with each page is a reference flag such that

ref flag = 1 if the page has been referenced in recent past

= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

-- clock implementation of NRU:



last replaced pointer (lrp)
if replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE's have their reference bits set to zero.

An optimization is to search for the a page that is both not recently referenced AND not dirty.

Demand Paging and Prepaging

Fetch Policy

when is the page brought into memory?

if pages are loaded solely in response to page faults, then the policy is *demand paging*

An alternative is *prepaging:*

anticipate future references and load such pages before their actual use

+ reduces page transfer overhead

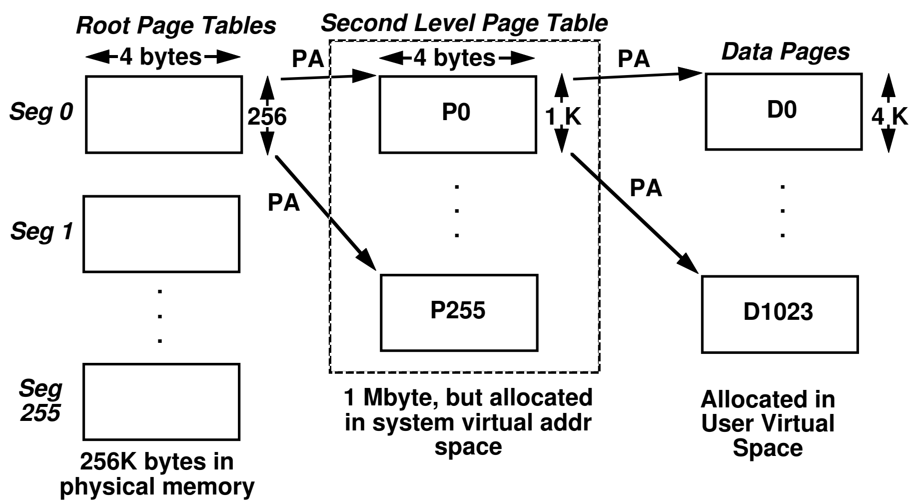
- removes pages already in page frames, which could adversely affect the page fault rate

- predicting future references usually difficult

Most systems implement demand paging without prepaging

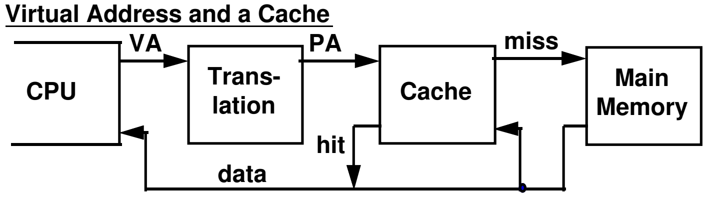
(One way to obtain effect of prepaging behavior is increasing the page size

2-level page table



$2^8 \times 2^8 \times 2^{10} \times 2^{12} = 2^{38}$

Virtual Address and a Cache



In Segment + Page approach OR 2-level page table approach, it takes two memory accesses to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

ASIDE: Why access cache with PA at all? VA caches have a problem!
synonym problem: two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is ***Translation Lookaside Buffer*** or ***TLB***

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

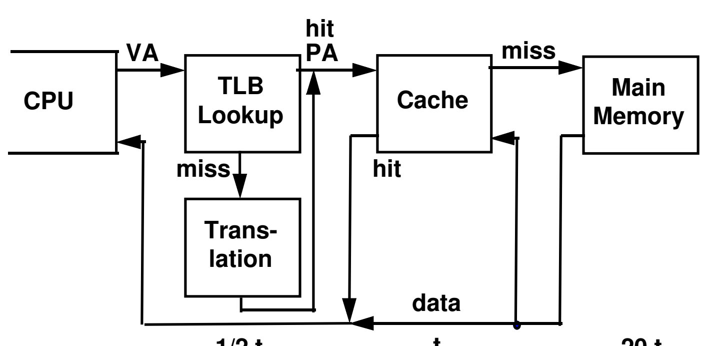
TLB access time comparable to, though shorter than, cache access time (still much less than main memory access time)

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

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.

Translation with a TLB



Reducing Translation Time

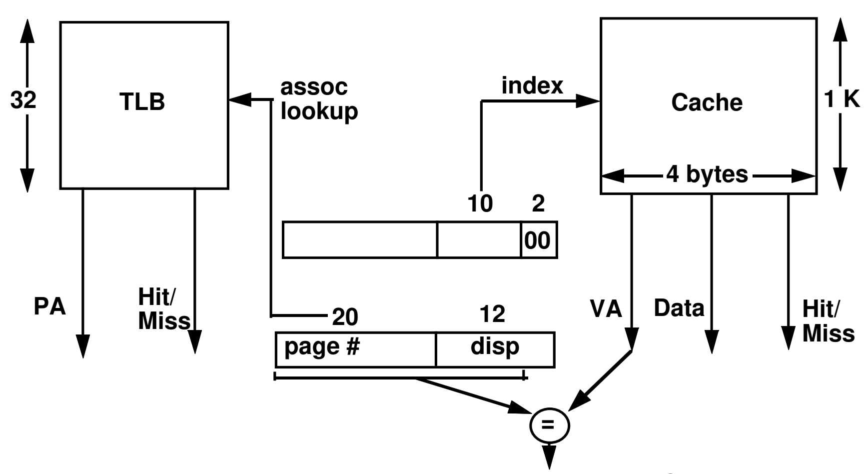
Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

DS3100: 32bit VA, 4KB page, 20bit V Page tag, 32Bit PA
TLB fully associative, 64 entries, (tag, page, dirty, valid)
Hardware support for table walk on TLB miss (rather than full hardware replacement)

Overlapped Cache & TLB Access



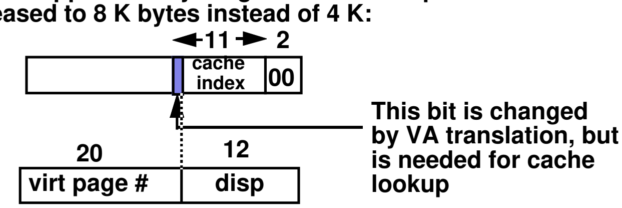
IF cache hit AND (cache tag = VA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = VA)] and TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

Problems With Overlapped TLB Access

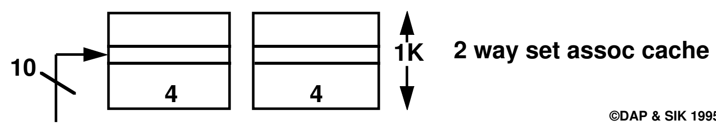
Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



Solutions:
- go to 8K byte page sizes
- go to 2 way set associative cache (would allow you to continue to use a 10 bit index)



SS-20 Review

- 4-way associative 16 KB Dcache = 4 * 1 page (4 KB)
- 5-way associative 20 KB Icache = 5 * 1 page (4 KB)

Break (5 Minutes)

Virtual Memory

- Virtual address (2^{32}, 2^{64}) to Physical Address mapping (2^{28})
- Virtual memory terms of cache terms:
  - Cache block?
  - Cache Miss?
- How is Virtual Memory different from caches?
  - What Controls Replacement
  - Size
  - Lower level use
- 4Qs for VM?
  - Q1: Where can a block be placed in the upper level?
    Fully Associative, Set Associative, Direct Mapped
  - Q2: How is a block found if it is in the upper level?
    Cache was Tag/Block
  - Q3: Which block should be replaced on a miss?
    Cache was Random, LRU
  - Q4: What happens on a write?
    Write Back or Write Through (with Write Buffer)

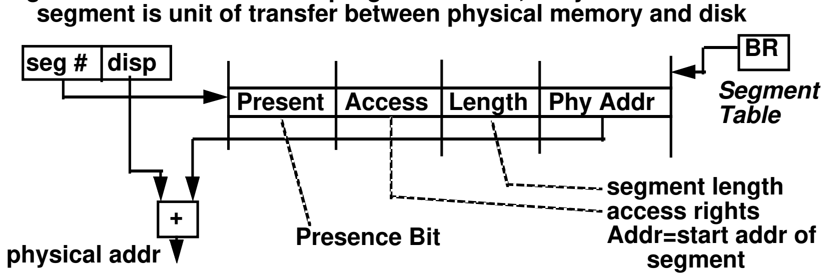
More on Selecting a Page Size

- Reasons for larger page size
  - Page table size is inversely proportional to the page size; therefore memory saved .
  - fast cache hit time easy when cache ≤ page size; bigger page makes it feasible to have 1 page cache
  - Transferring larger pages to or from secondary storage, possibly over a network, is more efficient
  - Number of TLB entries are restricted by clock cycle time, so a larger page size maps more memory thereby reducing TLB misses.
- Reasons for a smaller page size
  - don't waste storage; data must be contiguous within page
  - quicker process start for small processes?
- Hybrid solution: multiple page sizes
  Alpha: 8KB, 64KB, 512 KB, 4 MB pages

Segmentation (see x86)

Alternative to paging (often combined with paging)

Segments allocated for each program module; may be different sizes
segment is unit of transfer between physical memory and disk



Faults:
missing segment (Present = 0)
overflow (Displacement exceeds segment length)
protection violation (access incompatible with segment protection)

Segment-based addressing sometimes used to implement *capabilities*, i.e., hardware support for sophisticated protection mechanisms

Segment Based Addressing

Two Serious Drawbacks:

(1) storage allocation with variable sized blocks (best fit vs. fit fit vs. buddy system)

(2) external fragmentation: physical memory allocated in such a fashion that all remaining pieces are too small to be allocated to any segment. Solved be expensive run-time memory compaction.

The best of both worlds: paged segmentation schemes

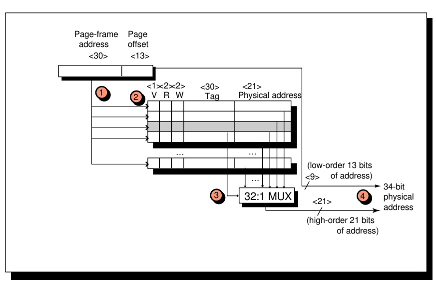
virtual address

seg #	page #	displacement

used by IBM: 4K byte pages, 16 x 1 Mbyte or 64 x 64 Kbyte segments

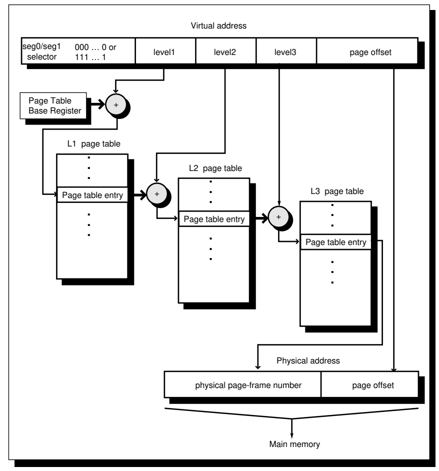
Example of Fast translation: Translation Buffer

- **Cache of translated addresses**
- **Alpha 21064 TLB: 32 entry fully associative**



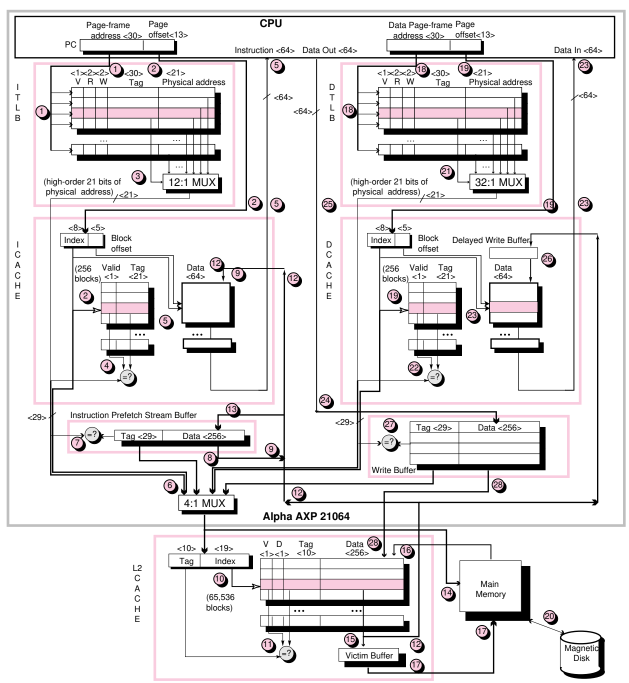
Alpha VM Mapping

- "64-bit" address divided into 3 segments
  - seg0 (bit 63=0) user code/heap
  - seg1 (bit 63 = 1, 62 = 1) user stack
  - kseg (bit 63 = 1, 62 = 0) kernel segment for OS
- 3 level page table, each one page
  - Alpha only 43 unique bits of VA
  - (future min page size up to 64KB = > 55 bits of VA)
- PTE bits; valid, kernel & user read & write enable (No reference, use, or dirty bit)
  - What do you do?



Alpha 21064

- Separate Instr & Data TLB & Caches
- TLBs fully associative
- Caches 8KB direct mapped
- Critical 8 bytes first
- 2 MB L2 cache, direct mapped
- 256 bit path to main memory, 4 64-bit modules



Virtual Memory in Historical Perspective

° Since VM invented, DRAMs now 64,000 times larger

° Today systems rarely have many page faults

° Should we drop VM then?

Conclusion #1

° Virtual Memory invented as another level of the hierarchy

° Controversial at the time: can SW automatically manage 64KB across many programs?

° DRAM growth removed the controversy

° Today VM allows many processes to share single memory without having to swap all processes to disk, protection more important

° (Multi-level) page tables to map virtual address to physical address

° TLBs are important for fast translation

° TLB misses are significant in performance

Conclusion #2

- Theory of Algorithms & Compilers based on number of operations
- Compiler remove operations and “simplify” ops:
  Integer adds << Integer multiplies << FP adds << FP multiplies
  - Advanced pipelines => these operations take similar time
- As Clock rates get higher and pipelines are longer, instructions take less time but DRAMs only slightly faster (although much larger)
- Today time is a function of (ops, cache misses)
- Given importance of caches, what does this mean to:
  - Compilers?
  - Data structures?
  - Algorithms?