CS152
Computer Architecture and Engineering
Lecture 21

Virtual Memory and Buses

April 19, 1999

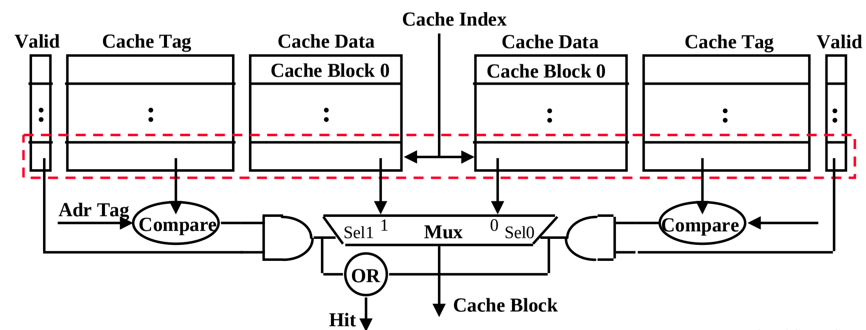
John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

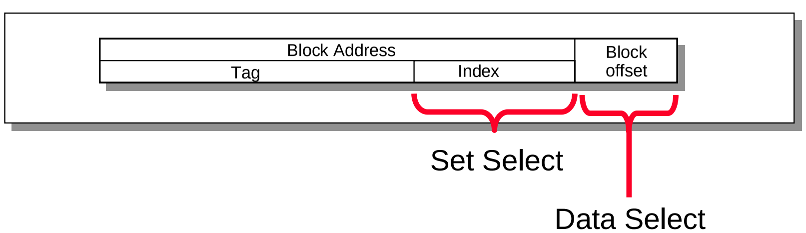
Recap: Set Associative Cache

° **N-way set associative: N entries for each Cache Index**
- N direct mapped caches operates in parallel

° **Example: Two-way set associative cache**
- Cache Index selects a "set" from the cache
- The two tags in the set are compared to the input in parallel
- Data is selected based on the tag result



Recap: How is a block found if it is in the upper level?



° **Direct indexing (using index and block offset), tag compares, or combination**

° **Increasing associativity shrinks index, expands tag**

Recap: Cache Performance

Execution_Time =
 Instruction_Count x Cycle_Time x
 (ideal CPI + Memory_Stalls/Inst + Other_Stalls/Inst)

Memory_Stalls/Inst =
 Instruction Miss Rate x Instruction Miss Penalty +
 Loads/Inst x Load Miss Rate x Load Miss Penalty +
 Stores/Inst x Store Miss Rate x Store Miss Penalty

Average Memory Access time (AMAT) =
 $\text{Hit Time}_{L1} + (\text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}) =$
 $(\text{Hit Rate}_{L1} \times \text{Hit Time}_{L1}) + (\text{Miss Rate}_{L1} \times \text{Miss Time}_{L1})$

L2 Equations:
 $\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$

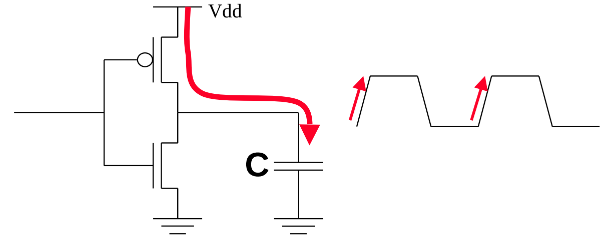
 $\text{AMAT} = \text{Hit Time}_{L1} +$
 $\text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$

Recap: Cache Optimization Summary

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

	Technique	MR	MP	HT	Complexity
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			2
miss penalty	Faster Memory System		+		3
	Priority to Read Misses		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2

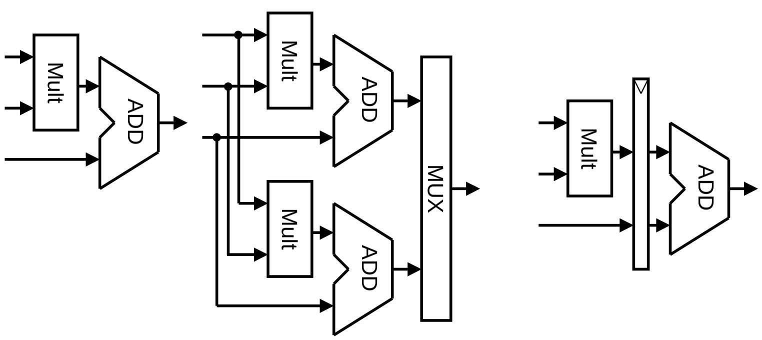
Recap: Power — One of the most important problems today



° Energy drained from power-supply on rising edges
- Energy = CV^2
- Power = $CV^2 f$

° Important metric: total amount of energy expended
- Simply slowing down frequency doesn't help: only makes things slower (total energy for job same)
- Must either *lower voltage* or *lower capacitance*
- At lower voltage, CMOS runs slower

Recap: two important tricks for lowering energy/OP



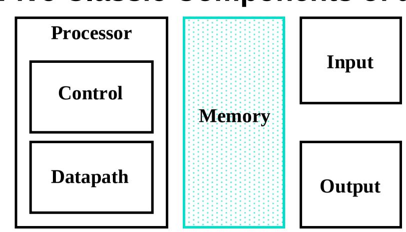
° Original

° Parallel
- Run at 1/2 clock rate
- Twice capacitance
- Lower voltage

° Pipelined
- Run at same clock rate
- Lower voltage, since each piece simpler

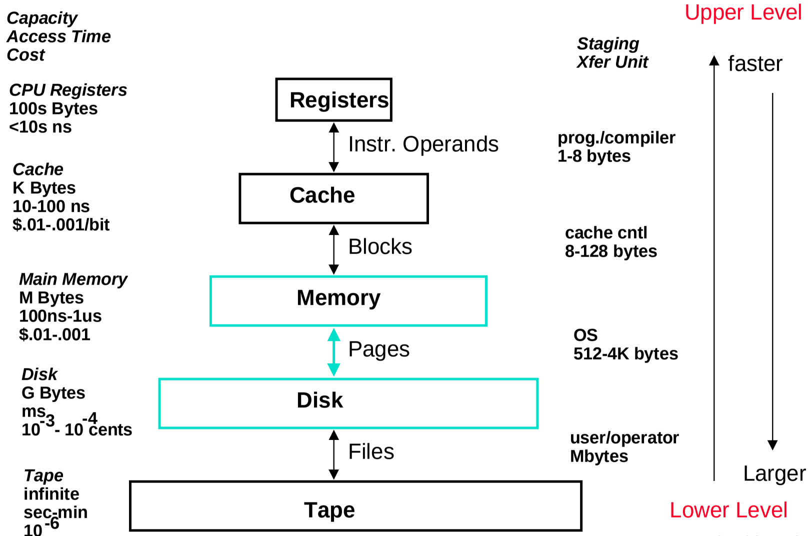
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer



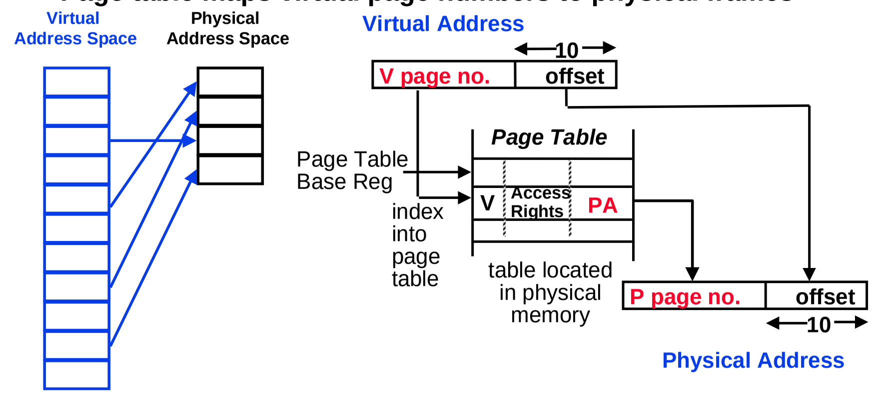
° Today's Topics:
- Recap last lecture
- Virtual Memory
- Protection
- TLB
- Buses

Recall: Levels of the Memory Hierarchy



What is virtual memory?

- Virtual memory => treat memory as a cache for the disk
- Terminology: blocks in this cache are called "Pages"
- Typical size of a page: 1K — 8K
- Page table maps virtual page numbers to physical frames



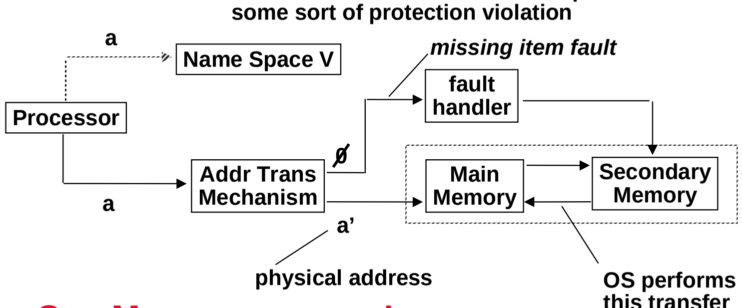
Address Map

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

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

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

= Ø if data at virtual address a is not present in M or some sort of protection violation



One Map per process!

Three Advantages of Virtual Memory

- **Translation:**
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- **Protection:**
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs => Far more "viruses" under Microsoft Windows
- **Sharing:**
  - Can map same physical page to multiple users ("Shared memory")

Issues in Virtual Memory System Design

What is the size of information blocks that are transferred from secondary to main storage (M)? ⇒ *page size*
(Contrast with physical block size on disk, I.e. *sector size*)

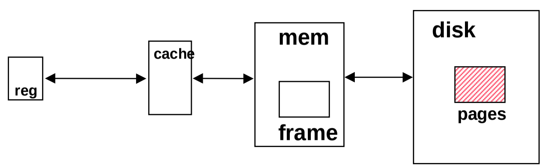
Which region of M is to hold the new block ⇒ *placement policy*

How do we find a page when we look for it? ⇒ *block identification*

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*

What do we do on a write? ⇒ *write policy*

Missing item fetched from secondary memory only on the occurrence of a fault ⇒ *demand load policy*



pages

How big is the translation (page) table?

Virtual Page Number	Page Offset

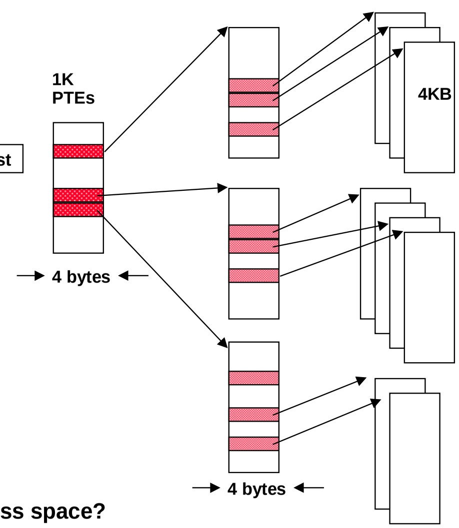
- Simplest way to implement "fully associative" lookup policy is with large lookup table.
- Each entry in table is some number of bytes, say 4
- With 4K pages, 32- bit address space, need:
  $2^{32}/4K = 2^{20}$ = 1 Meg entries x 4 bytes = 4MB
- With 4K pages, 64-bit address space, need:
  $2^{64}/4K = 2^{52}$ entries = BIG!
- Can't keep whole page table in memory!

Large Address Spaces

Two-level Page Tables

32-bit address:

10	10	12
P1 index	P2 index	page offest



- 2 GB virtual address space
- 4 MB of PTE2
  - paged, holes
- 4 KB of PTE1

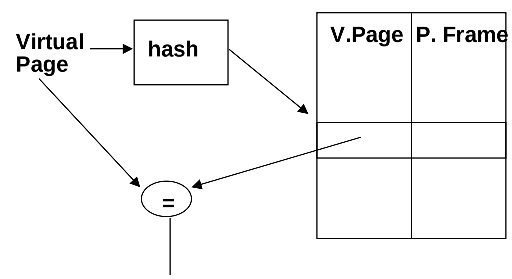
What about a 48-64 bit address space?

Inverted Page Tables

IBM System 38 (AS400) implements 64-bit addresses.

48 bits translated

start of object contains a 12-bit tag



=> TLBs or virtually addressed caches are critical

Administrivia

° Second midterm coming up (Wed, April 21)
Will be 5:30 - 8:30 in 277 Cory. LaVal's for pizza afterwards!

- Microcoding/implementation of complex instructions
- Pipelining
  - Hazards, branches, forwarding, CPI calculations
  - (may include something on dynamic scheduling)
- Memory Hierarchy (including Caches, TLBs, DRAM)
- Simple Power issues (if you understand the two slides in this lecture, you should be in good shape)
- No I/O will be on this exam.

Administrivia II

° Project evaluations (Lab 6) to TAs by midnight tonight.

° Final project has been posted!

- Print out again, if you printed already. I added an option to do a TLBs.
- Lots of options. Must do an amount of work that is proportional to number of people in your group (this comes out to 6 "points" per person)

° Major organizational options:

- 2-way superscalar (18 points)
- 2-way multithreading (20 points)
- 2-way multiprocessor (18 points)
- out-of-order execution (22 points)

° Fill out with other options => Can't get more points than allowed by group, but can "hedge" with more points.

Administrivia III

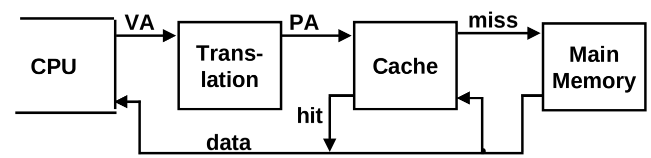
° Important: Design for Test

- You should be testing from the very start of your design
- Consider adding special monitor modules at various points in design => I have asked you to label trace output from these modules with the current clock cycle #
- The time to understand how components of your design should work is while you are designing!

° Pending schedule:

- Friday 4/23: lab organizational message to TAs
- Monday 4/26 and Wednesday 4/28: I/O
- Friday 4/30: Update on design
- Monday 5/3: no class
- Tuesday 5/4 and Wednesday 5/5: Oral reports
- Monday 5/10 Last class (wrap up, evaluations, etc)
- Tuesday 5/11 by 5pm: final project reports due.
- Friday 5/14 grades should be posted.

Virtual Address and a Cache: Step backward???



° Virtual memory seems to be really slow:

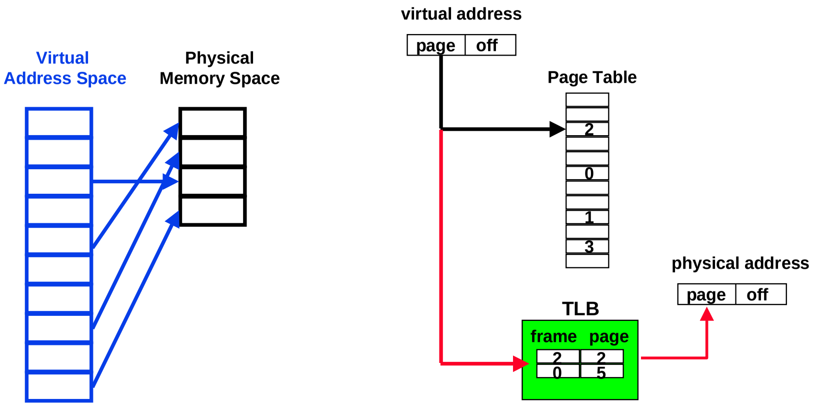
- we have to access memory on every access -- even cache hits!
- Worse, if translation not completely in memory, may need to go to disk before hitting in cache!

° Solution: Caching! (surprise!)

- Keep track of most common translations and place them in a "Translation Lookaside Buffer" (TLB)

Making address translation practical: TLB

- Virtual memory => memory acts like a cache for the disk
- Page table maps virtual page numbers to physical frames
- Translation Look-aside Buffer (TLB) is a cache of recent translations

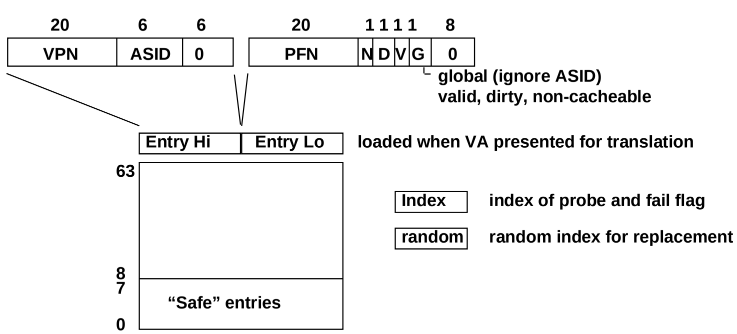


TLB organization: include protection

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

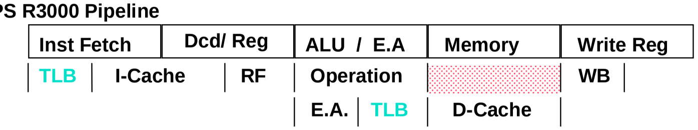
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- Dirty => Page modified (Y/N)?
  Ref => Page touched (Y/N)?
  Valid => TLB entry valid (Y/N)?
  Access => Read? Write?
  ASID => Which User?

R3000 TLB & CP0 (MMU)



Constraints on TLB organization

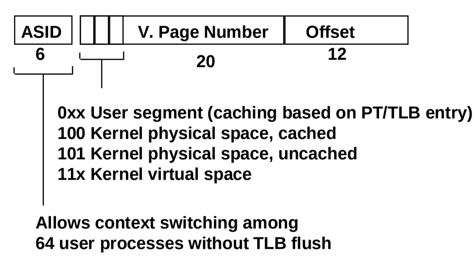
MIPS R3000 Pipeline



TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space



What is the replacement policy for TLBs?

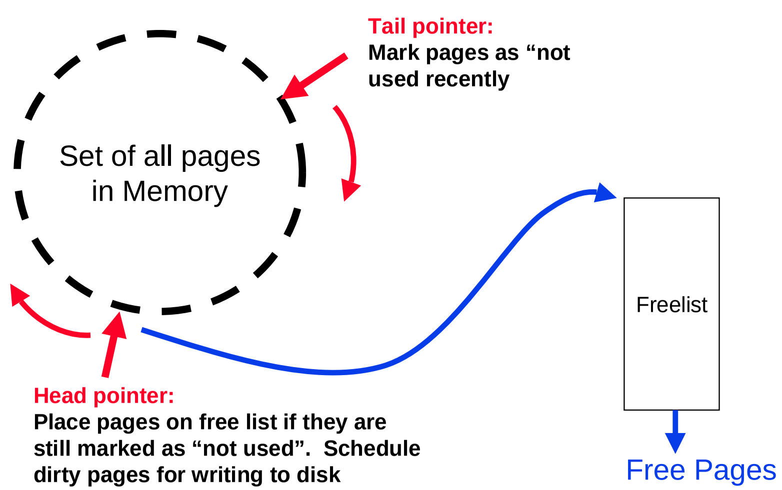
- On a TLB miss, we check the page table for an entry.
- Two architectural possibilities:
  - Hardware "table-walk" (Sparc, among others)
    - Structure of page table must be known to hardware
  - Software "table-walk" (MIPS was one of the first)
    - Lots of flexibility
    - Can be expensive with modern operating systems.
- What if missing virtual entry is not in page table?
  - This is called a "Page Fault"
  - A "Page Fault" means that requested virtual page is not in memory.
  - Operating system must take over.
- Note: possible that parts of page table are not even in memory (I.e. paged out!)
  - The root of the page table always "pegged" in memory

Page Fault: What happens when you miss?

- Page fault means that page is not resident in memory
- Hardware must detect situation
- Hardware cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
  - pick a page to discard (possibly writing it to disk)
  - start loading the page in from disk
  - schedule some other process to run

  Later (when page has come back from disk):
  - update the page table
  - resume to program so HW will retry and succeed!
- What is in the page fault handler?
  - see CS162
- What can HW do to help it do a good job?

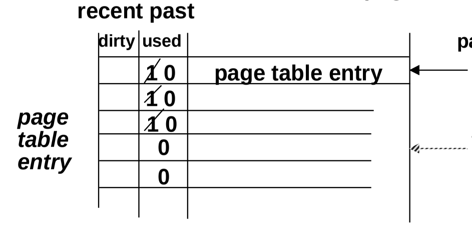
Page Replacement: Not Recently Used (1-bit LRU, Clock)



Page Replacement: Not Recently Used (1-bit LRU, Clock)

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



page fault handler:

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.

Or search for the a page that is both not recently referenced AND not dirty.

Architecture part: support dirty and used bits in the page table
=> may need to update PTE on any instruction fetch, load, store
How does TLB affect this design problem? Software TLB miss?

Reducing Translation Time

Still have TLB translation time in serial with cache lookup!

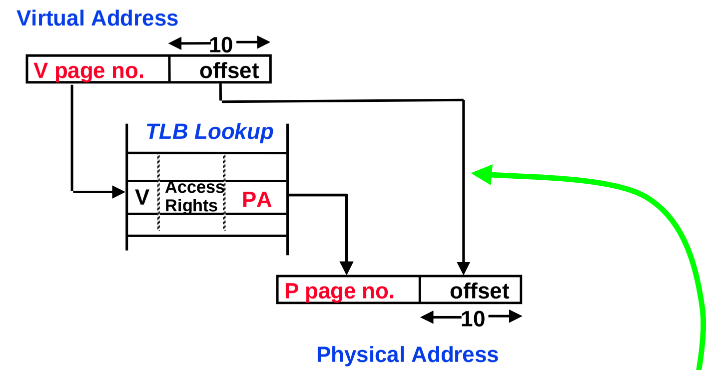
Machines with TLBs go one step further to reduce # cycles/cache access

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

Reducing translation time further

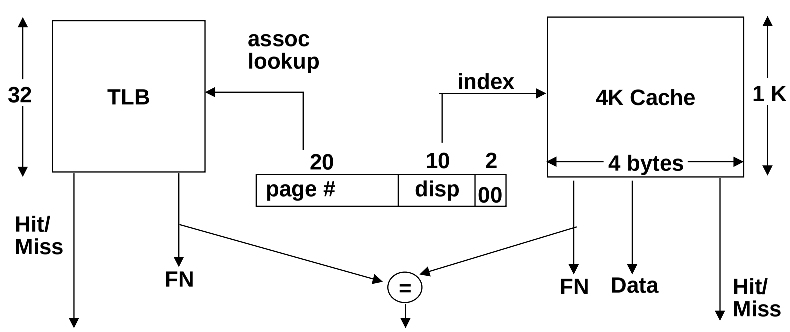
° As described, TLB lookup is in serial with cache lookup:



° Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  • Works because lower bits of result (offset) available early

Overlapped TLB & Cache Access

° If we do this in parallel, we have to be careful, however:



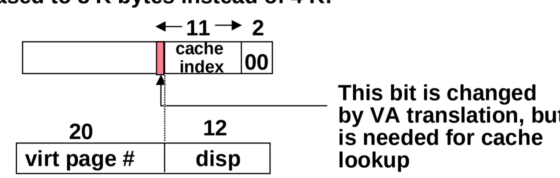
What if cache size is increased to 8KB?

Problems With Overlapped TLB Access

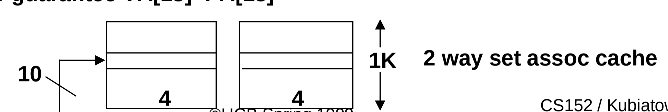
Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

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

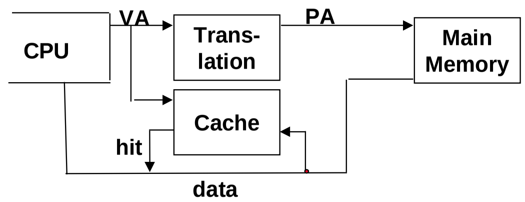
Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



Solutions:
go to 8K byte page sizes;
go to 2 way set associative cache; or
SW guarantee VA[13]=PA[13]



Another option: Virtually Addressed Cache



Only require address translation on cache miss!

synonym problem: two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

nightmare 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.

(usually disallowed by fiat)

Optimal Page Size

° Mimimize wasted storage

— small page minimizes internal fragmentation

— small page increase size of page table

° Minmize transfer time

— large pages (multiple disk sectors) amortize access cost

— sometimes transfer unnecessary info

— sometimes prefetch useful data

— sometimes discards useless data early

General trend toward larger pages because

— big cheap RAM

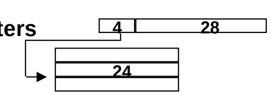
— increasing mem / disk performance gap

— larger address spaces

Survey

° **R4000**
- 32 bit virtual, 36 bit physical
- variable page size (4KB to 16 MB)
- 48 entries mapping page pairs (128 bit)

° **MPC601 (32 bit implementation of 64 bit PowerPC arch)**
- 52 bit virtual, 32 bit physical, 16 segment registers
- 4KB page, 256MB segment
- 4 entry instruction TLB
- 256 entry, 2-way TLB (and variable sized block xlate)
- overlapped lookup into 8-way 32KB L1 cache
- hardware table search through hashed page tables



° **Alpha 21064**
- arch is 64 bit virtual, implementation subset: 43, 47,51,55 bit
- 8,16,32, or 64KB pages (3 level page table)
- 12 entry ITLB, 32 entry DTLB
- 43 bit virtual, 28 bit physical octword address

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)**

Summary #1/2 : TLB, Virtual Memory

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
  1) Where can block be placed?
  2) How is block found?
  3) What block is replaced on miss?
  4) How are writes handled?
  More cynical version of this:
  Everything in computer architecture is a cache!
- Page tables map virtual address to physical address
- TLBs are a cache on translation and are extremely important for good performance
- Special tricks necessary to keep TLB out of critical cache-access path
- TLB misses are significant in processor performance:
  - These are funny times: most systems can't access all of 2nd level cache without TLB misses!

Summary #2 / 2: Memory Hierachy

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; VM *translation, protection, and sharing* are more important than memory hierarchy