CS4100: 計算機結構

Memory Hierarchy

國立清華大學資訊工程學系
九十三學年度第一學期

Adapted from Prof. D. Patterson's class notes

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy

Technology Trends

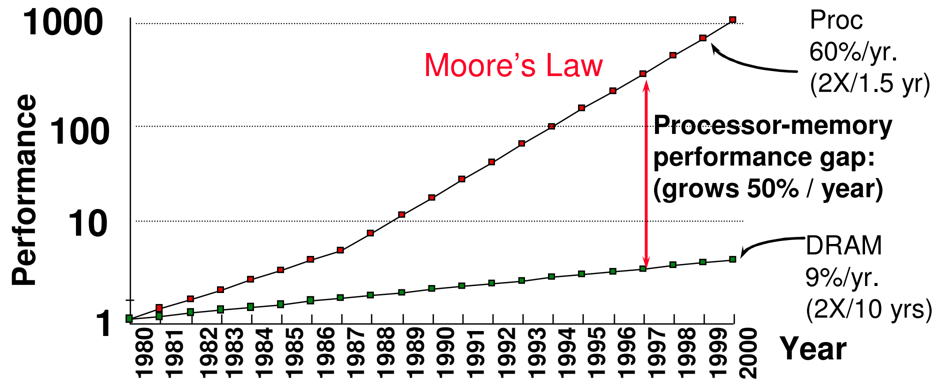
	Capacity	Speed (latency)
Logic:	4x in 1.5 years	4x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

DRAM

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

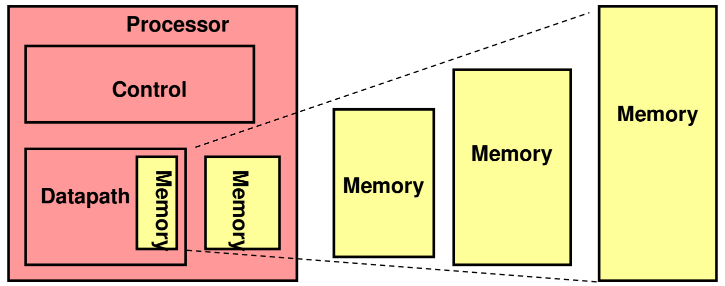
1000:1!
2:1!

Processor Memory Latency Gap



Solution: Memory Hierarchy

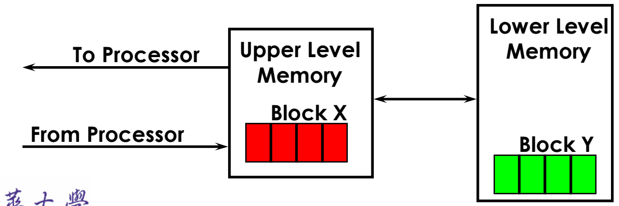
- An Illusion of a large, fast, cheap memory
  - Fact: Large memories slow, fast memories small
  - How to achieve: hierarchy, parallelism
- An expanded view of memory system:



Speed:	Fastest	Slowest
Size:	Smallest	Biggest
Cost:	Highest	Lowest

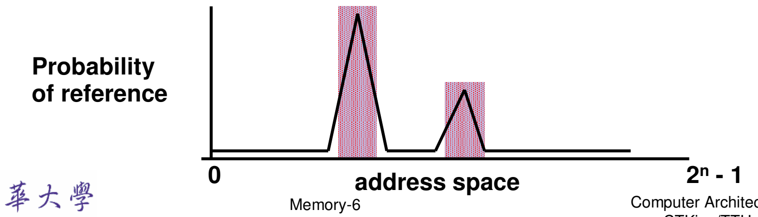
Memory Hierarchy: Principle

- At any given time, data is copied between only two adjacent levels:
  - Upper level: the one closer to the processor
    - Smaller, faster, uses more expensive technology
  - Lower level: the one away from the processor
    - Bigger, slower, uses less expensive technology
- *Block*: basic unit of information transfer
  - Minimum unit of information that can either be present or not present in a level of the hierarchy



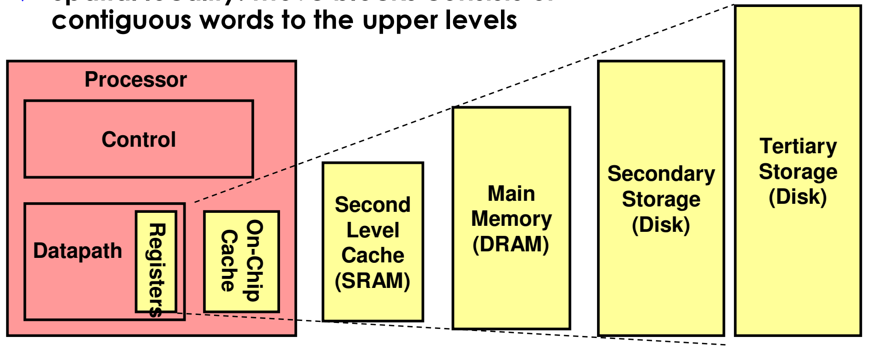
Why Hierarchy Works?

- *Principle of Locality*:
  - Program access a relatively small portion of the address space at any instant of time
  - 90/10 rule: 10% of code executed 90% of time
- Two types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close by tend to be referenced soon



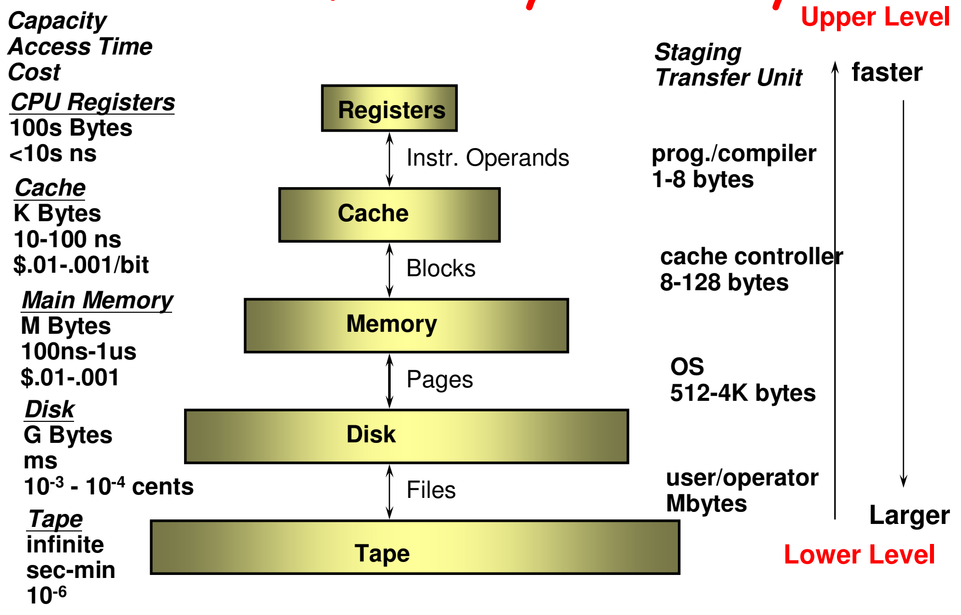
How Does It Work?

- Temporal locality: keep most recently accessed data items closer to the processor
- Spatial locality: move blocks consists of contiguous words to the upper levels



Speed (ns):	1's	10's		100's	10,000,000's (10's ms)	10,000,000,000's (10's sec)
Size (bytes):	100's	K's		M's	G's	T's

Levels of Memory Hierarchy



How Is the Hierarchy Managed?

- Registers <-> Memory
  - by compiler (programmer?)
- cache <-> memory
  - by the hardware
- memory <-> disks
  - by the hardware and operating system (virtual memory)
  - by the programmer (files)

Memory Hierarchy Technology

- Random access:
  - Access time same for all locations
  - DRAM: *Dynamic Random Access Memory*
    - High density, low power, cheap, slow
    - Dynamic: need to be refreshed regularly
    - Addresses in 2 halves (memory as a 2D matrix):
      - RAS/CAS (Row/Column Access Strobe)
    - Use for main memory
  - SRAM: *Static Random Access Memory*
    - Low density, high power, expensive, fast
    - Static: content will last (forever until lose power)
    - Address not divided
    - Use for caches

Comparisons of Various Technologies

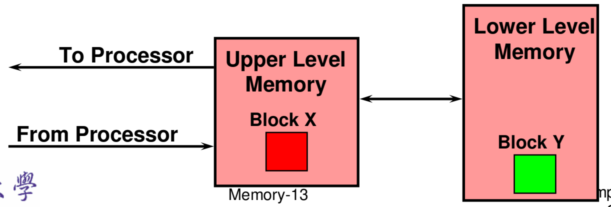
Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5 – 5 ns	$4000 – $10,000
DRAM	50 – 70 ns	$100 – $200
Magnetic disk	5,000,000 – 20,000,000 ns	$0.05 – $2

Memory Hierarchy Technology

- Performance of main memory:
  - Latency: related directly to *Cache Miss Penalty*
    - Access Time: time between request and word arrives
    - Cycle Time: time between requests
  - Bandwidth: Large Block Miss Penalty (interleaved memory, L2)
- Non-so-random access technology:
  - Access time varies from location to location and from time to time, e.g., disk, CDROM
- Sequential access technology: access time linear in location (e.g., tape)

Memory Hierarchy: Terminology

- Hit: data appears in upper level (Block X)
  - Hit rate: fraction of memory access found in the upper level
  - Hit time: time to access the upper level
    - RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: time to replace a block in the upper level + time to deliver the block to the processor (latency + transmit time)
- Hit Time << Miss Penalty



4 Questions for Hierarchy Design

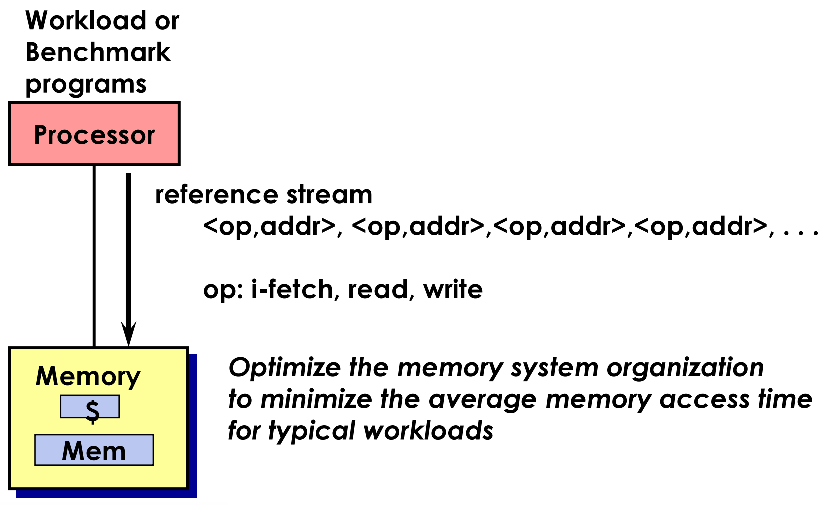
Q1: Where can a block be placed in the upper level?
=> *block placement*

Q2: How is a block found if it is in the upper level?
=> *block identification*

Q3: Which block should be replaced on a miss?
=> *block replacement*

Q4: What happens on a write?
=> *write strategy*

Memory System Design


Workload or Benchmark programs
Processor
reference stream
<op,addr>, <op,addr>,<op,addr>,<op,addr>, . . .
op: i-fetch, read, write
Memory
$
Mem

Optimize the memory system organization to minimize the average memory access time for typical workloads

Summary of Memory Hierarchy

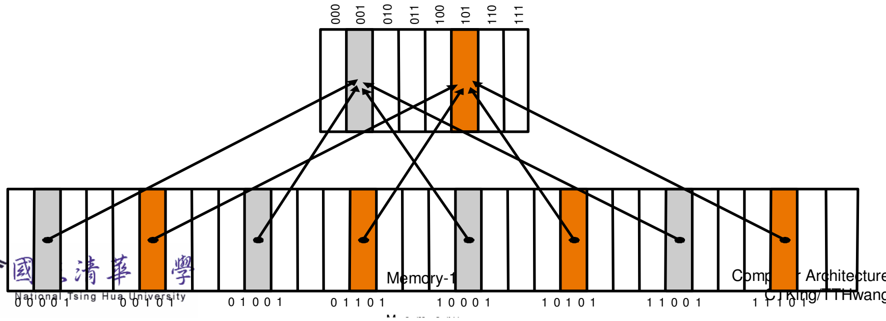
- Two different types of locality:
  - Temporal Locality (Locality in Time)
  - Spatial Locality (Locality in Space)
- Using the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.
- DRAM is slow but cheap and dense:
  - Good for presenting users with a BIG memory system
- SRAM is fast but expensive, not very dense:
  - Good choice for providing users FAST accesses

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy

Basics of Cache

- Our first example: *direct-mapped cache*
- Block Placement :
  - For each item of data at the lower level, there is exactly one location in cache where it might be
  - Address mapping: modulo number of blocks
- Block identification :
  - How to know if an item is in cache? Tag and valid bit
  - If it is, how do we find it?



Accessing a Cache

- 10K words, 1-word block:
  - Cache index: lower 10 bits
  - Cache tag: upper 20 bits
  - Valid bit (When start up, valid is 0)

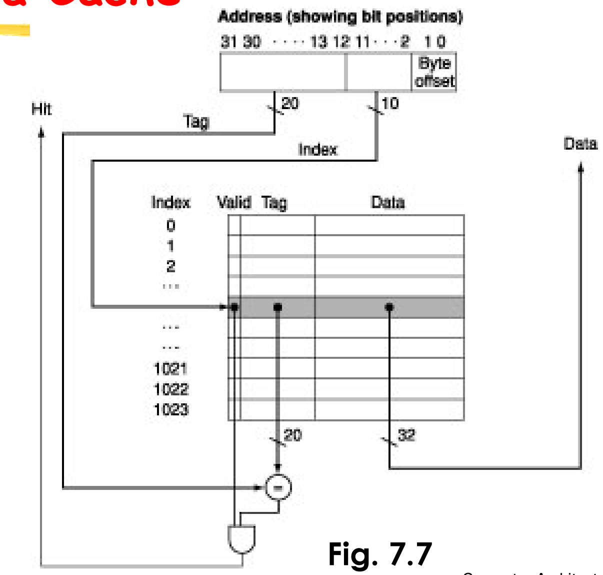


Fig. 7.7

Hits and Misses

- Read hits: this is what we want!
- Read misses
  - Block replacement ?
  - Stall CPU, freeze register contents, fetch block from memory, deliver to cache, restart
- Write hits: keep cache/memory consistent?
  - Write-through: write to cache and memory at same time => but memory is very slow!
  - Write-back: write to cache only (write to memory when that block is being replaced)
    - Need a *dirty bit* for each block
  - DECStation 3100 uses write-through, but no need to consider hit or miss on a write (one block has one word)
    - index the cache using bits 15-2 of the address
    - write bits 31-16 into tag, write data, set valid
    - write data into main memory

Hits and Misses

- Write misses:
  - Write-allocated: read block into cache, write the word
    - low miss rate, complex control, match with write-back
  - Write-non-allocate: write directly into memory
    - high miss rate, easy control, match with write-through

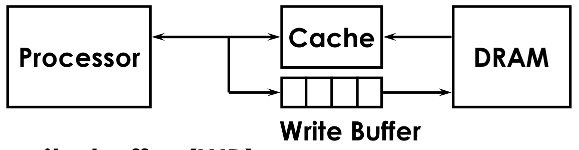
Miss Rate

- Miss rate of Instrinsity FastMATH for SPEC2000 Benchmark:

Intrinsity FastMATH	Instruction miss rate	Data miss rate	Effective combined miss rate
	0.4%	11.4%	3.2%

Fig. 7.10

Avoid Waiting for Memory in Write Through



- Use a *write buffer* (WB):
  - Processor: writes data into cache and WB
  - Memory controller: write WB data to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
- Memory system designer's nightmare:
  - Store frequency > 1 / DRAM write cycle
  - Write buffer saturation => CPU stalled

Exploiting Spatial Locality (I)

- Increase block size for spatial locality

Total no. of tags and valid bits reduced

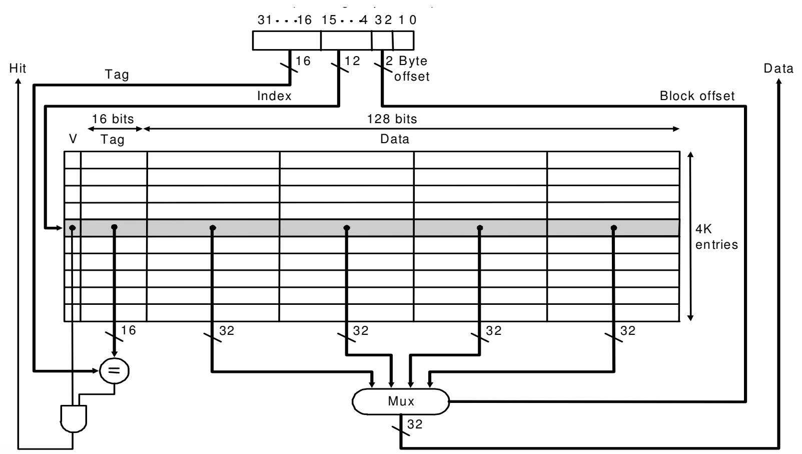


Fig. 7.9

Exploiting Spatial Locality (II)

- Increase block size for spatial locality
  - Read miss : bring back the whole block
  - Write: check tag and write at same time (WR-allocate)
    if tag match: OK; if not, read block and write again (more than one word in a block)

Block Size on Performance

- Increase block size tends to decrease miss rate

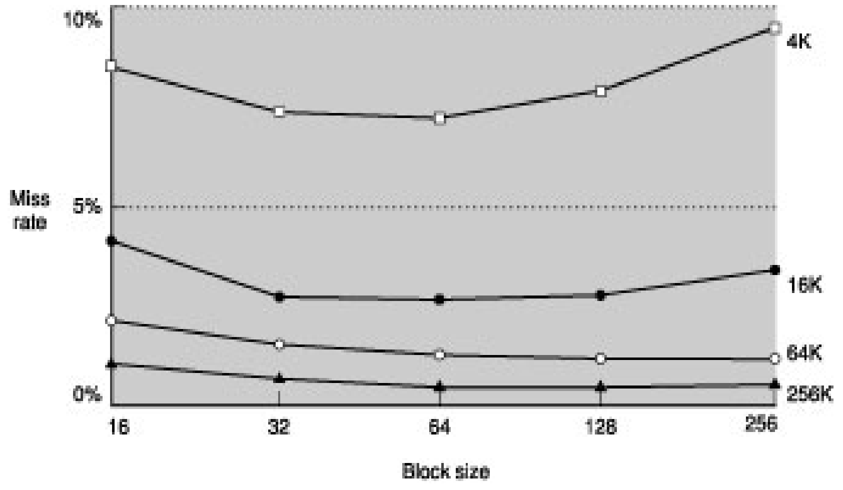
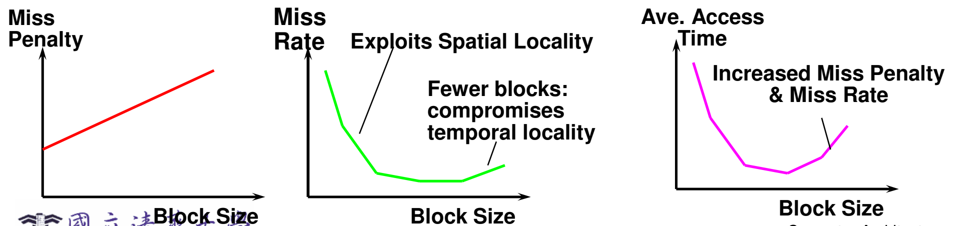


Fig. 7.8

Block Size Tradeoff

- Larger block size take advantage of spatial locality and improve miss ratio, BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size too big, miss rate goes up
    - Too few blocks in cache => high competition
- Average access time:
  = hit time x (1 - miss rate)+miss penalty x miss rate



Memory Design to Support Cache

- How to increase memory bandwidth to reduce miss penalty?

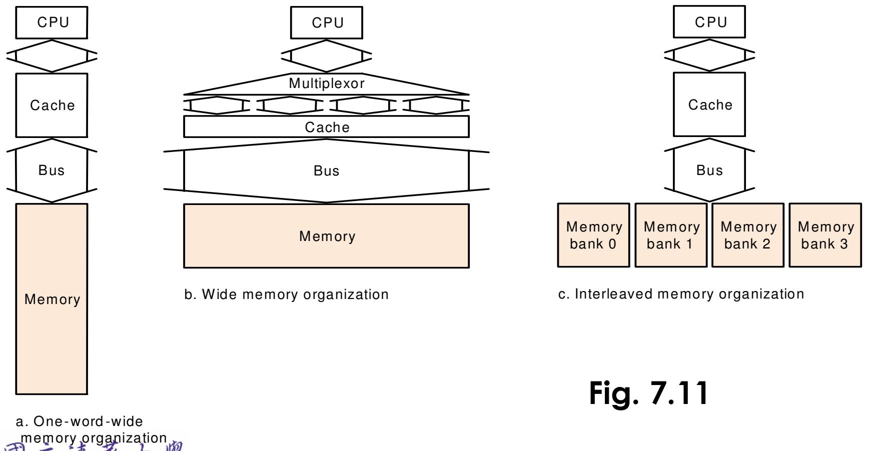
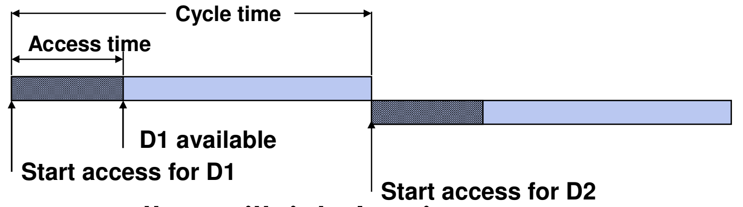


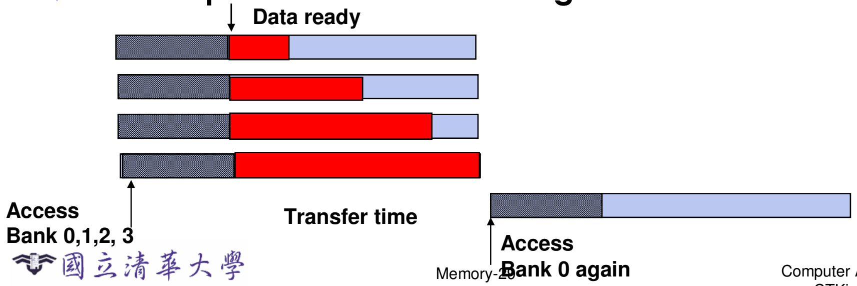
Fig. 7.11

Interleaving for Bandwidth

- Access pattern without interleaving:



- Access pattern with interleaving



Miss Penalty for Different Memory Organizations

Assume
- 1 memory bus clock to send the address
- 15 memory bus clocks for each DRAM access initiated
- 1 memory bus clock to send a word of data
- A cache block = 4 words
- Three memory organizations :
  - A one-word-wide bank of DRAMs
    Miss penalty = 1 + 4 x 15 + 4 x 1 = 65
  - A two-word-wide bank of DRAMs
    Miss penalty = 1 + 2 x 15 + 2 x 1 = 33
  - A four-bank, one-word-wide bank of DRAMs
    Miss penalty = 1 + 1 x 15 + 4 x 1 = 20

Cache Performance

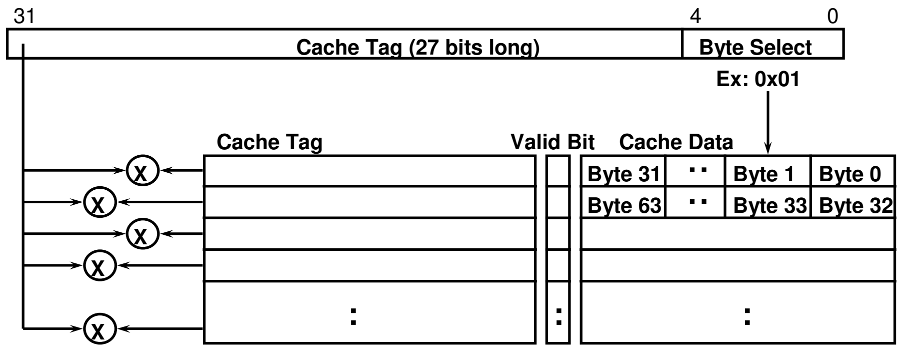
- Simplified model: (instruction misses)
  CPU time = (CPU execution cycles + memory stall cycles) x cycle time
  Memory stall cycles = instruction count x miss ratio x miss penalty
- Impact on performance: (data misses)
  - Suppose CPU executes at clock rate = 200MHz, CPI=1.1, 50% arith/logic, 30% ld/st, 20% control
  - 10% memory op. get 50-cycle miss penalty
  - CPI = ideal CPI + average stalls per instruction
    = 1.1+(0.30 mops/ins x 0.10 miss/mop x 50 cycle/miss)
    = 1.1 cycle + 1.5 cycle = 2. 6
  - 58 % of the time CPU stalled waiting for memory!
  - 1% inst. miss rate adds extra 0.5 cycles to CPI!

Improving Cache Performance

- Decreasing the miss ratio
- Reduce the time to hit in the cache
- Decreasing the miss penalty

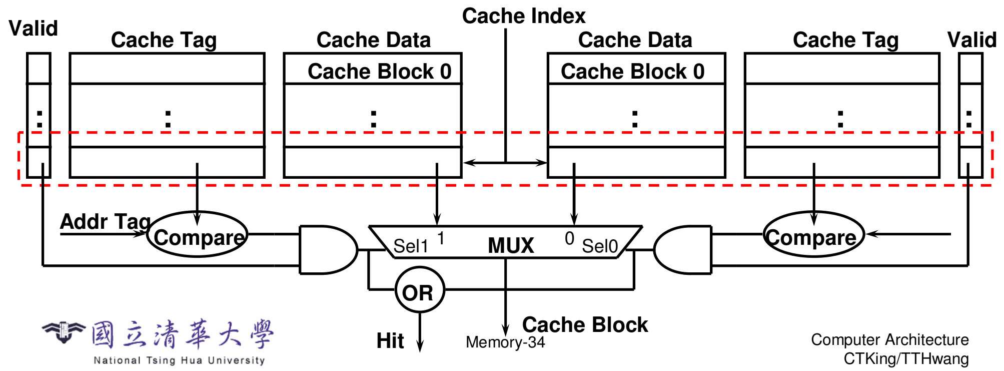
Reduce Miss Ratio with Associativity

- A fully associative cache:
  - Compare cache tags of all cache entries in parallel
  - Ex.: Block Size = 8 words, N 27-bit comparators

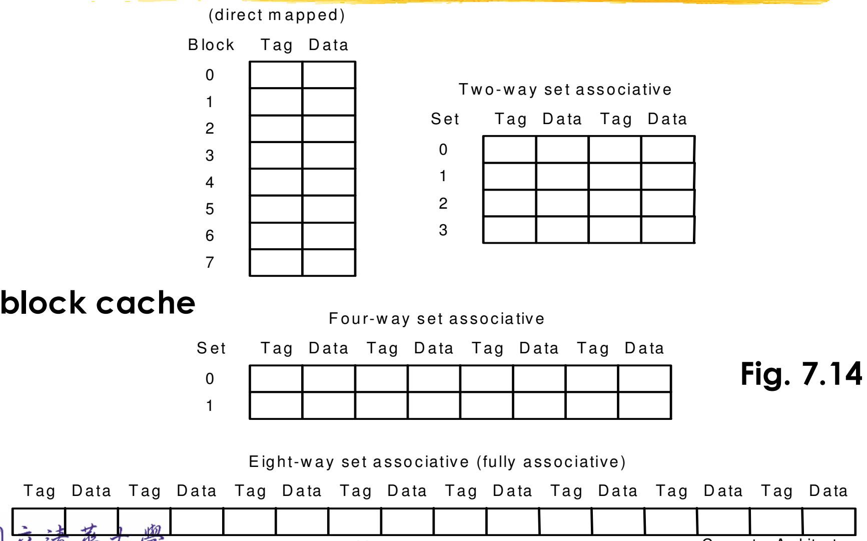


Set-Associative Cache

- N-way: 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 in parallel
  - Data is selected based on the tag result



Possible Associativity Structures



An 8-block cache

Fig. 7.14

Block Placement

- Placement of a block whose address is 12:

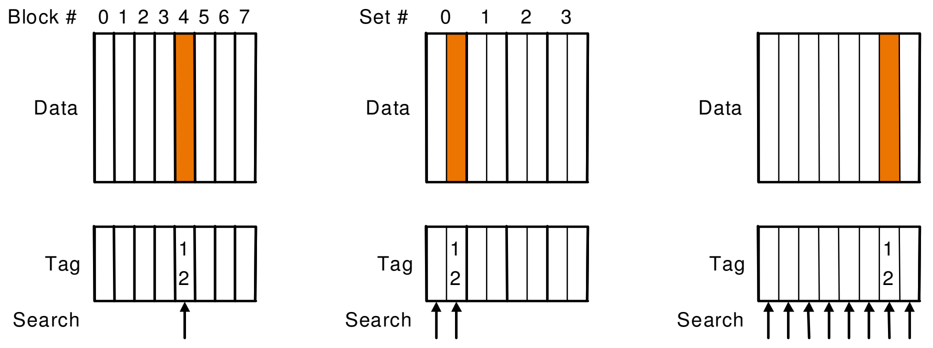


Fig. 7.13

Data Placement Policy

- Direct mapped cache:
  - Each memory block mapped to one location
  - No need to make any decision
  - Current item replaces previous one in location
- N-way set associative cache:
  - Each memory block has choice of N locations
- Fully associative cache:
  - Each memory block can be placed in ANY cache location
- Misses in N-way set-associative or fully associative cache:
  - Bring in new block from memory
  - Throw out a block to make room for new block
  - Need to decide on which block to throw out

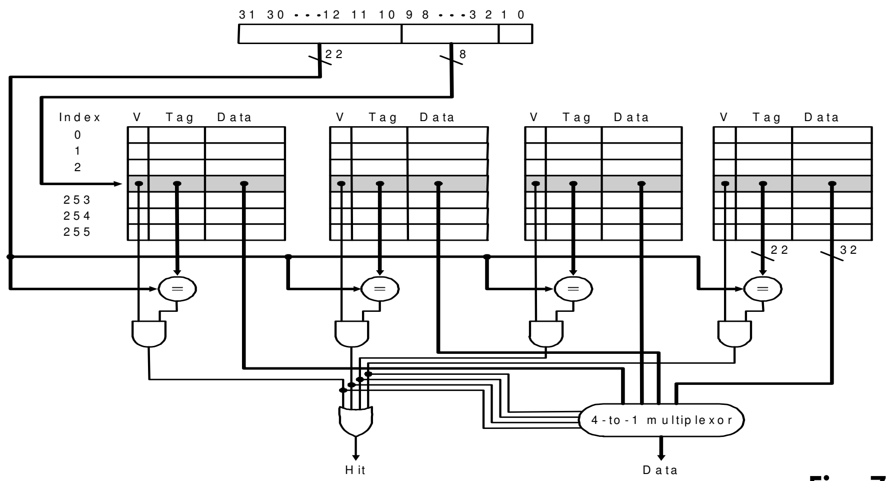
Cache Block Replacement

- Easy for direct mapped
- Set associative or fully associative:
  - Random
  - LRU (Least Recently Used):
    - Hardware keeps track of the access history and replace the block that has not been used for the longest time
  - An example of a pseudo LRU:
    - use a pointer pointing at each block in turn
    - whenever an access to the block the pointer is pointing at, move the pointer to the next block
    - when need to replace, replace the block currently pointed at

Comparing the Structures

- N-way set-associative cache
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
- Direct mapped cache
  - Cache block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue, recover later if miss

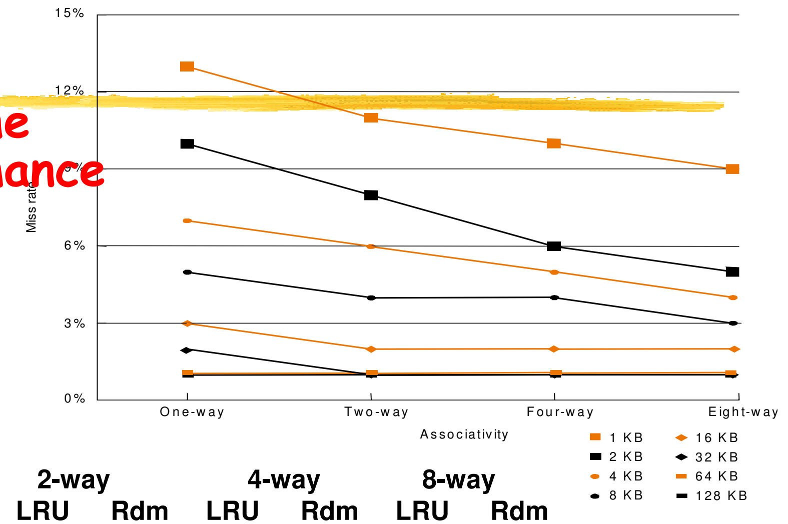
A 4-Way Set-Associative Cache



♦ Increasing associativity shrinks index, expands tag

Fig. 7.17

Cache Performance



Asso.	2-way		4-way		8-way	
Size	LRU	Rdm	LRU	Rdm	LRU	Rdm
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Reduce Miss Penalty with Multilevel Caches

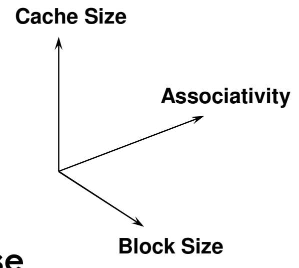
- ♦ Add a second level cache:
  - Often primary cache is on same chip as CPU
  - L1 focuses on minimizing hit time to reduce effective CPU cycle => faster, higher miss rate
  - L2 focuses on miss rate to reduce miss penalty => miss penalty goes down if data is in L2 cache
  - Average access time = L1 hit time + L1 miss rate $\times$ L1 miss penalty
  - L1 miss penalty = L2 hit time + L2 miss rate $\times$ L2 miss penalty
- ♦ Example:
  - CPI of 1.0 on a 500Mhz machine with a 5% miss rate, 200ns DRAM access
  - Adding a L2 cache with 20ns access time decreases overall miss rate to 2%, what miss penalty reduced?

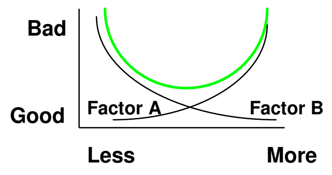
Sources of Cache Misses

- ♦ Compulsory (cold start, process migration):
  - First access to a block, not much we can do
  - Note: If you are going to run billions of instruction, compulsory misses are insignificant
- ♦ Conflict (collision):
  - >1 memory blocks mapped to same location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- ♦ Capacity:
  - Cache cannot contain all blocks by program
  - Solution: increase cache size
- ♦ Invalidation:
  - Block invalidated by other process (e.g., I/O) that updates the memory

Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Cache Summary

- Principle of Locality:
  - Program likely to access a relatively small portion of address space at any instant of time
    - Temporal locality: locality in time
    - Spatial locality: locality in space
- Three major categories of cache misses:
  - Compulsory: e.g., cold start misses.
  - Conflict: increase cache size or associativity
  - Capacity: increase cache size
- Cache design space
  - total size, block size, associativity
  - replacement policy
  - write-hit policy (write-through, write-back)
  - write-miss policy

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy

Virtual Memory

- Provide illusion of a large single-level store
  - Every program has its own address space, starting at address 0, only accessible to itself
    - yet, any can run anywhere in physical memory
    - executed in a name space (virtual address space) different from memory space (physical address space)
    - virtual memory implements the translation from virtual space to physical space
  - Every program has lots of memory (> physical memory)
- Many programs run at once with protection and sharing
- OS runs all the time and allocates physical resources

Virtual Memory

- View main memory as a cache for disk

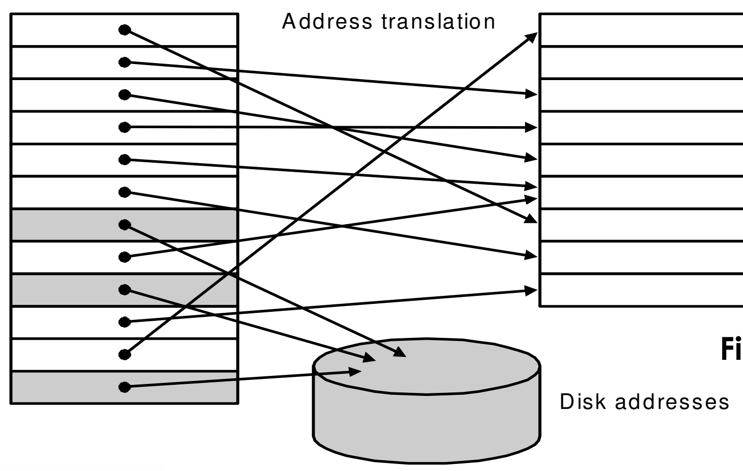


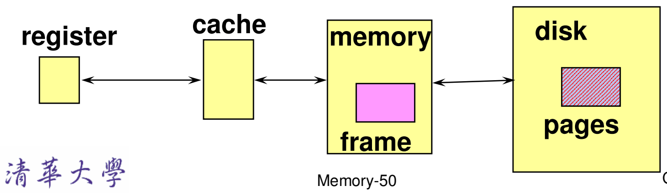
Fig. 7.19

Why Virtual Memory?

- Efficient and safe sharing of main memory among multiple programs
  - Map multiple virtual addresses to same physical addr.
- Remove prog. burden of a small physical memory
- Generality: run programs larger than size of physical memory
- Protection: regions of address space can be read-only, exclusive, ...
- Flexibility: portions of a program can be placed anywhere, without relocation
- Storage efficiency: retain only most important portions of program in memory
- Concurrent programming and I/O: execute other processes while loading/dumping page

Basic Issues in Virtual Memory

- Size of data blocks that are transferred from disk to main memory
- When memory is full, then some region of memory must be released to make room for the new block => replacement policy
- Which region of memory to hold new block => placement policy
- When to fetch missing items from disk?
  - Fetch only on a fault => demand load policy



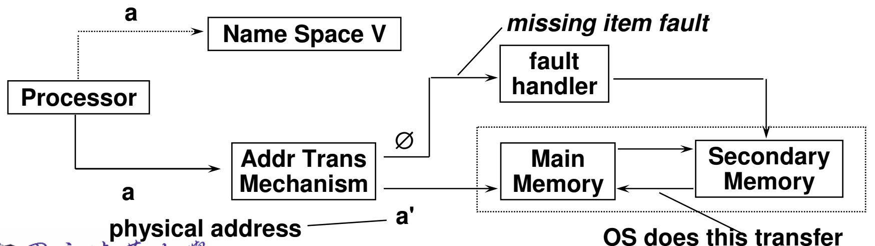
Paging

- Virtual and physical address space partitioned into blocks of equal size
  - *pages* *page frames*
- Key operation: address mapping

MAP: $V \rightarrow M \cup \{\varnothing\}$ address mapping function

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

= $\varnothing$ if data at virtual address a is not present in M



Key Decisions in Paging

- Huge miss penalty: a page fault may take millions of cycles to process
  - Pages should be fairly large (e.g., 4KB) to amortize the high access time
  - Reducing page faults is important
    - LRU replacement is worth the price
    - fully associative placement
      => use page table (in memory) to locate pages
  - Can handle the faults in software instead of hardware, because handling time is small compared to disk access
    - the software can be very smart or complex
    - the faulting process can be context-switched
  - Using write-through is too expensive, so we use write back <= write policy

Choosing the Page Size

- Minimize wasted storage:
  - small page minimizes internal fragmentation
  - small page increase size of page table
- Minimize transfer time:
  - large pages (multiple disk sectors) amortize access cost
  - sometimes transfer unnecessary info
  - sometimes prefetch useful data
  - sometimes discards useless data early
- A trend toward larger pages because
  - big cheap RAM
  - increasing memory/disk performance gap
  - larger address spaces

Page Tables

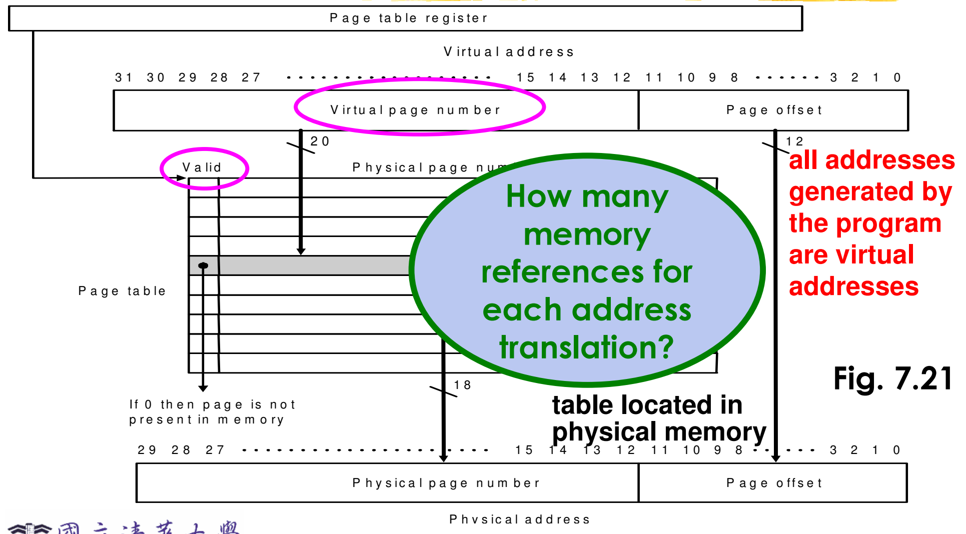


Fig. 7.21

Page Fault: What Happens When You Miss?

- Page fault means that page is not resident in memory
- Hardware must detect situation (why? how?), but it cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
  - Pick a page to discard (may write it to disk)
  - Load the page in from disk
  - Update the page table
  - Resume to program so HW will retry and succeed!

What can HW do to help the OS?

Handling Page Faults

- OS must know where to find the page
  - Create space on disk for all pages of process
  - Use a data structure to record where each valid page is on disk (may be part of page table)
  - Use another data structure to track which process and virtual addresses use each physical page => for replacement purpose

How to determine which frame to replace?
=> LRU policy
How to keep track of LRU?

Handling Page Faults

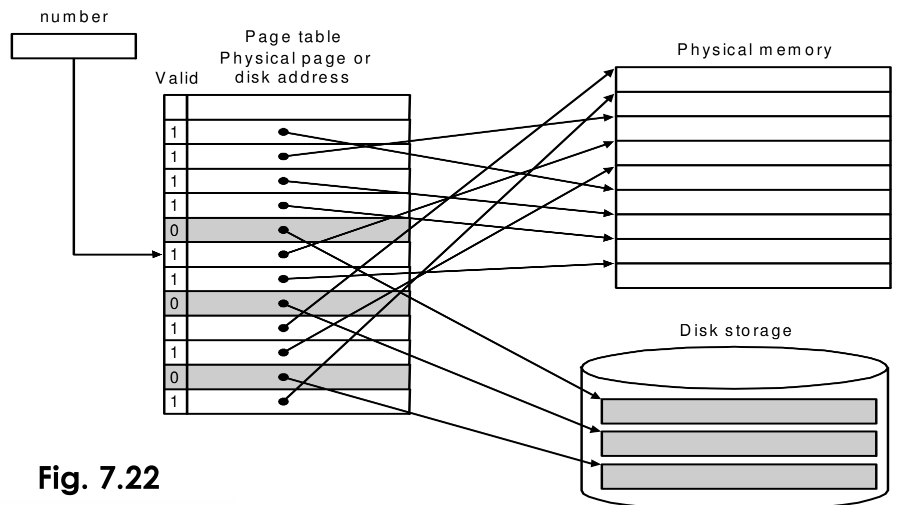
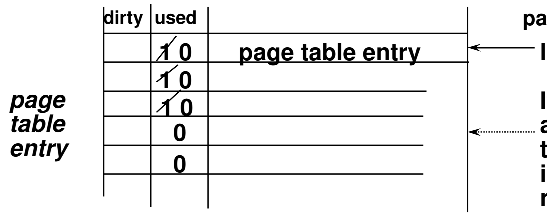


Fig. 7.22

Page Replacement: 1-bit LRU

- Associated with each page is a *reference flag*:
  ref flag = 1 if 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 a page that is both not recently referenced AND not dirty

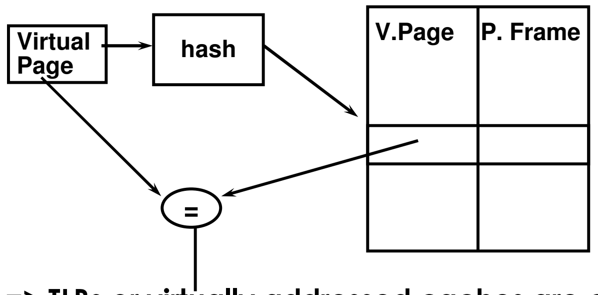
Architecture part: support dirty and used bits in the page table (how?)
=> may need to update PTE on any instruction fetch, load, store

Impact of Paging (I)

- Page table occupies storage
  32-bit VA, 4KB page, 4bytes/entry
  => 2^{20} PTE, 4MB table
- Possible solutions:
  - Use bounds register to limit table size; add more if exceed
  - Let pages to grow in both directions => 2 tables, 2 limit registers, one for hash, one for stack
  - Use hashing => page table same size as physical pages
  - Multiple levels of page tables
  - Paged page table (page table resides in virtual space)

Hashing: Inverted Page Tables

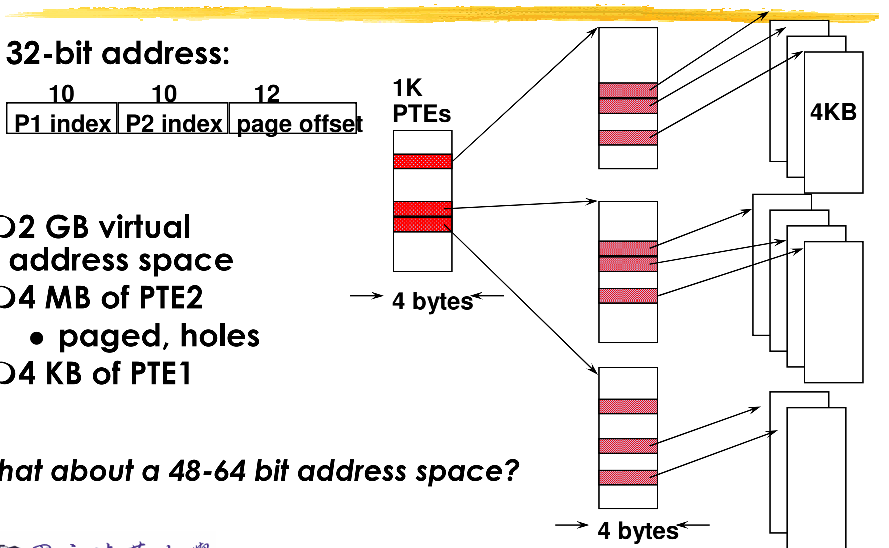
- IBM AS400 implements 64-bit addresses
  - 48 bits translated
  - start of object contains a 12-bit tag



=> TLBs or virtually addressed caches are critical

Two-level Page Tables

32-bit address:



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

What about a 48-64 bit address space?

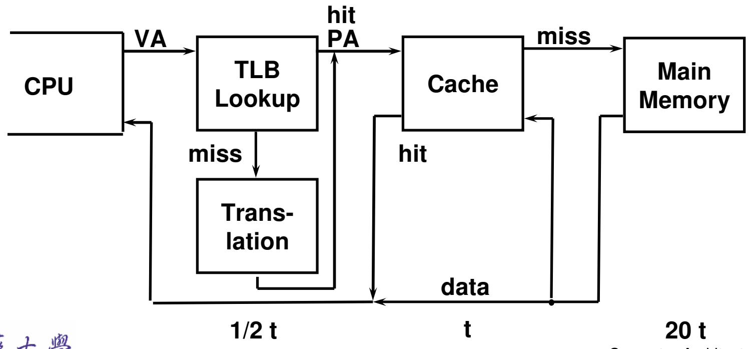
Impact of Paging (II)

- Each memory operation (instruction fetch, load, store) requires a page-table access!
  - Basically double number of memory operations
- Internal fragmentation: minimum a page
- Page fault may occur in the middle of an instruction and OS must be invoked to serve it

Making Address Translation Practical

- In VM, memory acts like a cache for disk
  - Page table maps virtual page numbers to physical frames
  - Use a page table cache for recent translation => *Translation Lookaside Buffer* (TLB)

Translation with a TLB



Translation Lookaside Buffer

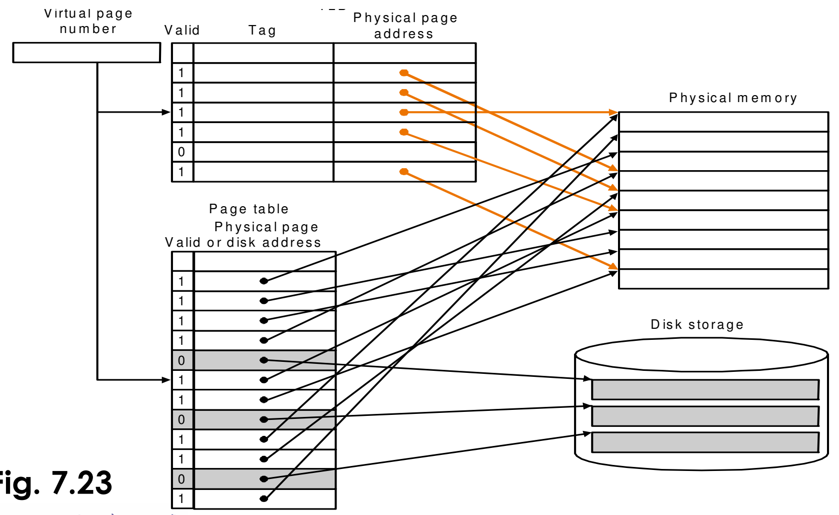


Fig. 7.23

Translation Lookaside Buffer

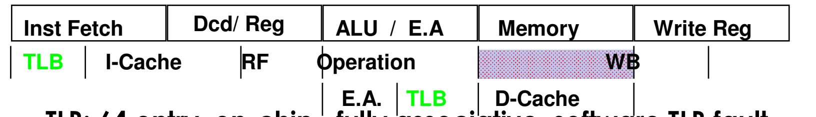
- Typical RISC processors have *memory management unit* (MMU) which includes TLB and does page table lookup
  - TLB can be organized as fully associative, set associative, or direct mapped
  - TLBs are small, typically < 128 - 256 entries
    - Fully associative on high-end machines, small n-way set associative on mid-range machines
- TLB hit on write:
  - Toggle dirty bit (write back to page table on replacement)
- TLB miss:
  - If page fault also => OS exception
  - If only TLB miss => load PTE into TLB (SW or HW?)

TLB of MIPS R2000

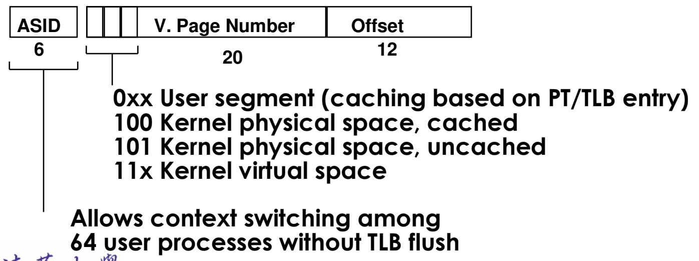
- 4KB pages, 32-bit VA
  => virtual page number: 20 bits
- TLB organization:
  - 64 entries, fully assoc., serve address and data
  - 64-bit/entry (20-bit tag, 20-bit physical page number, valid, dirty)
- On TLB miss:
  - Hardware saves page number to a special register and generates an exception
  - TLB miss routine finds PTE, uses a special set of system instructions to load physical addr into TLB
- Write requests must check a write access bit in TLB to see if it has permit to write
  => if not, an exception occurs

TLB in Pipeline

- MIPS R3000 Pipeline:



  - TLB: 64 entry, on-chip, fully associative, software TLB fault handler
  - Virtual address space:



ASID
V. Page Number
Offset
6
20
12
0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space
Allows context switching among
64 user processes without TLB flush

Integrating TLB and Cache

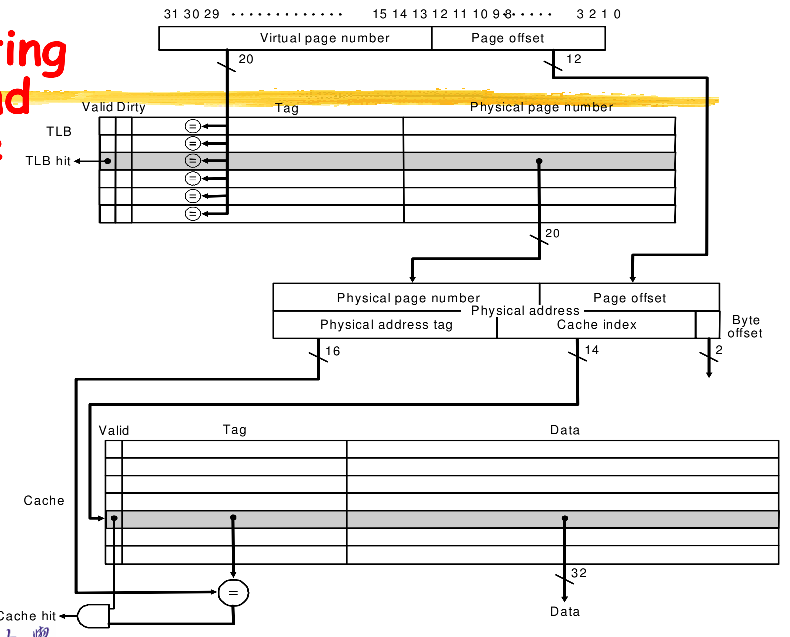
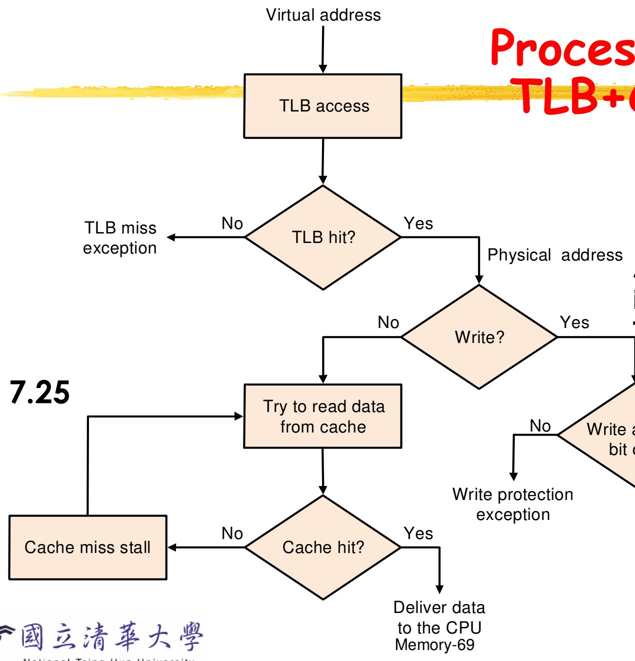


Fig. 7.24

Processing in TLB+Cache



A reference may miss in all 3 components: TLB, VM, cache

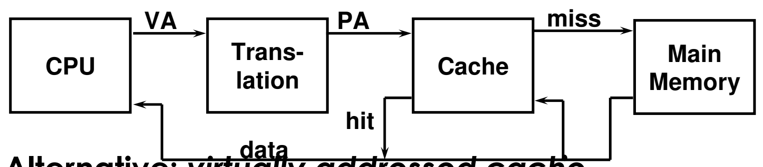
Fig. 7.25

Possible Combinations of Events

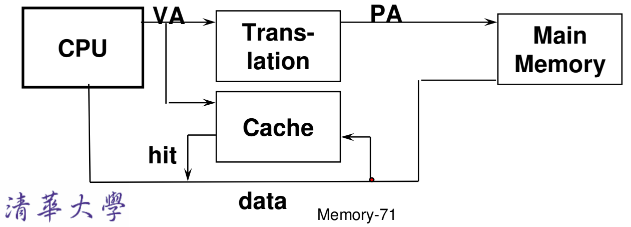
Cache	TLB	Virtual Memory	Possible? Conditions?
Miss	Hit	Hit	Yes; but page table never checked if TLB hits
Hit	Miss	Hit	TLB miss, but entry found in page table; after retry, data in cache
Miss	Miss	Hit	TLB miss, but entry found in page table; after retry, data miss in cache
Miss	Miss	Miss	TLB miss and is followed by a page fault; after retry, data miss in cache
Miss	Hit	Miss	No; not in TLB if page not in memory
Hit	Hit	Miss	No; not in TLB if page not in memory
Hit	Miss	Miss	No; not in cache if page not in memory

Virtual Address and Cache

- TLB access is serial with cache access
  - Cache is physically indexed and tagged



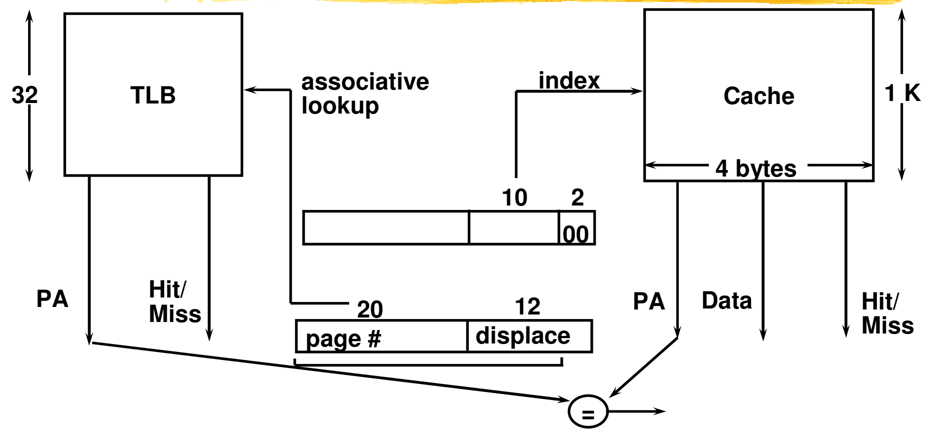
- Alternative: *virtually addressed cache*
  - Cache is virtually indexed and virtually tagged



Virtually Addressed Cache

- Require address translation only on miss!
- Problem:
  - *Synonym/alias 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;
  - Or software enforced alias boundary: same least-significant bits of VA &PA > cache size

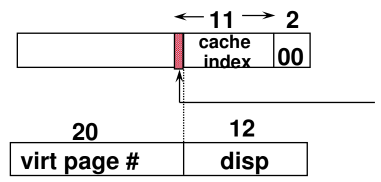
An Alternative: Overlapped TLB and Cache Access



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

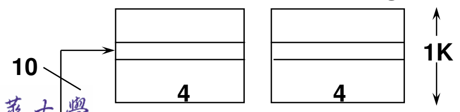
Problem with Overlapped Access

- Address bits to index into cache must not change as a result of VA translation
  - Limits to small caches, large page sizes, or high n-way set associativity if want a large cache
  - Ex.: cache is 8K bytes instead of 4K:



This bit is changed by VA translation, but is needed for cache lookup

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



2-way set associative cache

Protection with Virtual Memory

- Protection with VM:
  - Must protect data of a process from being read or written by another process
- Supports for protection:
  - Put page tables in the addressing space of OS
    => user process cannot modify its own PT and can only use the storage given by OS
  - Hardware supports: (2 modes: kernel, user)
    - Portion of CPU state can be read but not written by a user process, e.g., mode bit, PT pointer
      - These can be changed in kernel with special instr.
    - CPU to go from user to kernel: system calls
      From kernel to user: return from exception (RFE)
- Sharing: P2 asks OS to create a PTE for a virtual page in P1's space, pointing to page to be shared

A Common Framework for Memory Hierarchies

- Policies and features that determine how hierarchy functions are similar qualitatively
- Four questions for memory hierarchy:
  - Where can a block be placed in upper level?
    - Block placement: one place (direct mapped), a few places (set associative), or any place (fully associative)
  - How is a block found if it is in the upper level?
    - Block identification: indexing, limited search, full search, lookup table
  - Which block should be replaced on a miss?
    - Block replacement: LRU, random
  - What happens on a write?
    - Write strategy: write through or write back

Modern Systems

Characteristic	Intel Pentium Pro	PowerPC 604
Virtual address	32 bits	52 bits
Physical address	32 bits	32 bits
Page size	4 KB, 4 MB	4 KB, selectable, and 256 MB
TLB organization	A TLB for instructions and a TLB for data Both four-way set associative Pseudo-LRU replacement Instruction TLB: 32 entries Data TLB: 64 entries TLB misses handled in hardware	A TLB for instructions and a TLB for data Both two-way set associative LRU replacement Instruction TLB: 128 entries Data TLB: 128 entries TLB misses handled in hardware

Characteristic	Intel Pentium Pro	PowerPC 604
Cache organization	Split instruction and data caches	Split intruction and data caches
Cache size	8 KB each for instructions/data	16 KB each for instructions/data
Cache associativity	Four-way set associative	Four-way set associative
Replacement	Approximated LRU replacement	LRU replacement
Block size	32 bytes	32 bytes
Write policy	Write-back	Write-back or write-through

Challenge in Memory Hierarchy

- Every change that potentially improves miss rate can negatively affect overall performance

Design change	Effects on miss rate	Possible effects
size ↑	capacity miss ↓	access time ↑
associativity ↑	conflict miss ↓	access time ↑
block size ↑	spatial locality ↑	miss penalty ↑

- Trends:
  - Synchronous SRAMs (provide a burst of data)
  - Redesign DRAM chips to provide higher bandwidth or processing
  - Restructure code to increase locality
  - Use prefetching (make cache visible to ISA)

Summary

- Caches, TLBs, Virtual Memory all understood by examining how they deal with four questions:
  1) Where can block be placed?
  2) How is block found?
  3) What block is replaced on miss?
  4) How are writes handled?
- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance

Summary (cont.)

- Virtual memory was controversial:
  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 protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops):
  What does this mean to compilers, data structures, algorithms?