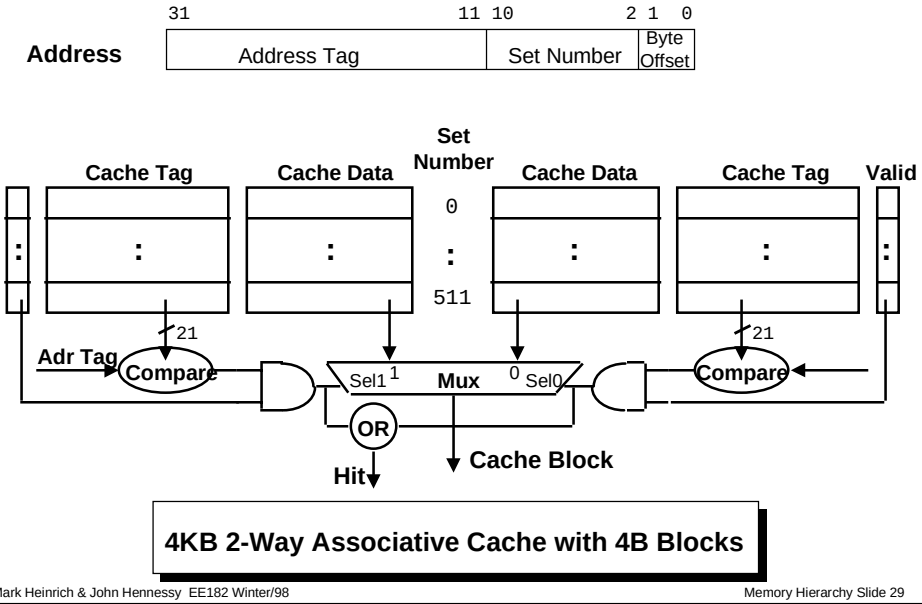
Set-Associative Caches

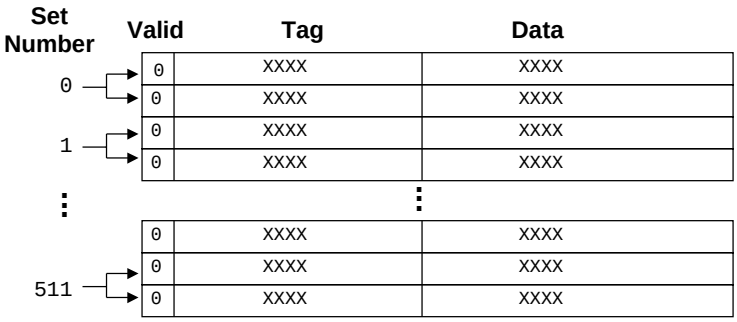
- **Improve cache hit ratio by allowing a memory location to be placed in more than one cache block**
  - — *N-Way* associative cache allows placement in any block of a *set* with N elements
    - N is the *set-size*
    - Number of blocks = N x number of sets
    - *Set number* is selected by a simple modulo function of the address bits (The set number is sometimes called the *index*.)
    - N comparators are needed to search all elements of the set in parallel
  - — *Fully-Associative* Cache
    - When there is a single set allowing a memory location to be placed in any cache block
  - — Direct-mapped organization can be considered a degenerate set-associative cache with set-size 1
- **For fixed cache capacity, higher associativity leads to higher hit rates**
  - — Because more combinations of cache lines can be present in the cache at the same time

2-Way Set-Associative Cache Example



Memory Reference Sequence

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - 0,4,8188,0,16384,0
- **This sequence had 5 misses and 1 hit for the direct-mapped cache with the same capacity and block size**

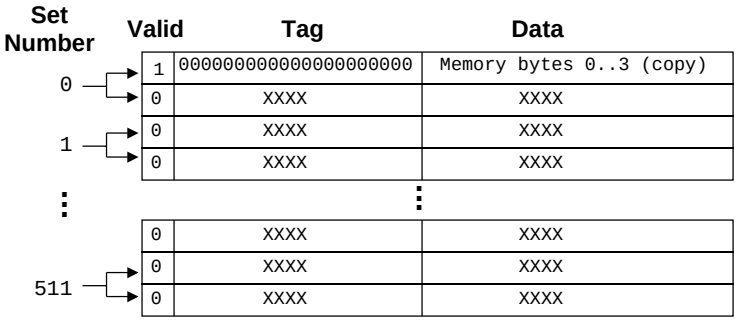


Cache Initially Empty

After Reference 1

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - 0,4,8188,0,16384,0

Address = 00000000000000000000 000000000 00

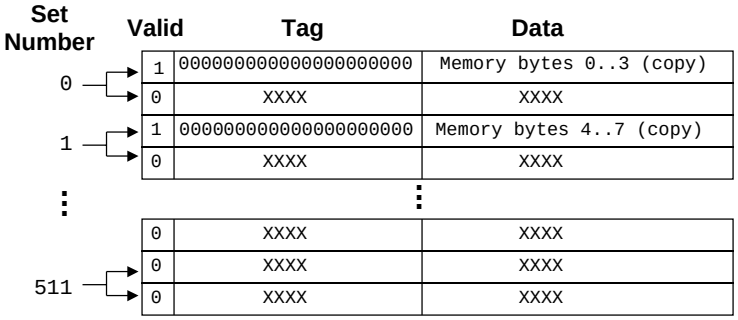


Cache Miss, Place in First Block of Set 0

After Reference 2

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - `0,4,8188,0,16384,0`

Address = 00000000000000000000 000000001 00

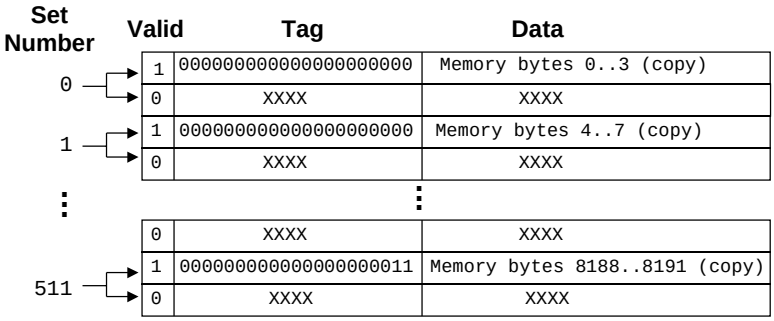


Cache Miss, Place in First Block of Set 1

After Reference 3

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - `0,4,8188,0,16384,0`

Address = 00000000000000000011 111111111 00

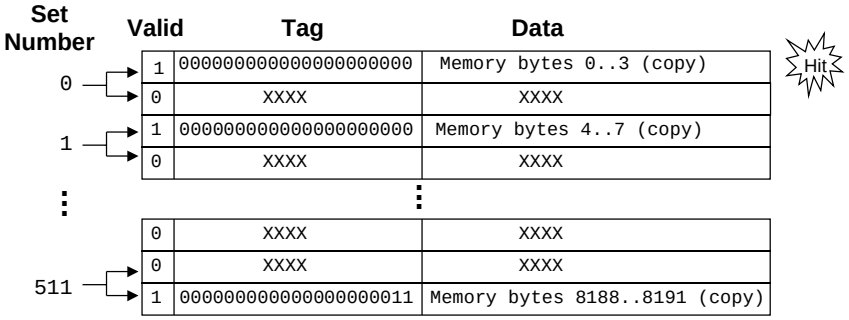


Cache Miss, Place in First Block of Set 511

After Reference 4

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - `0,4,8188,0,16384,0`

Address = 00000000000000000000 000000000 00

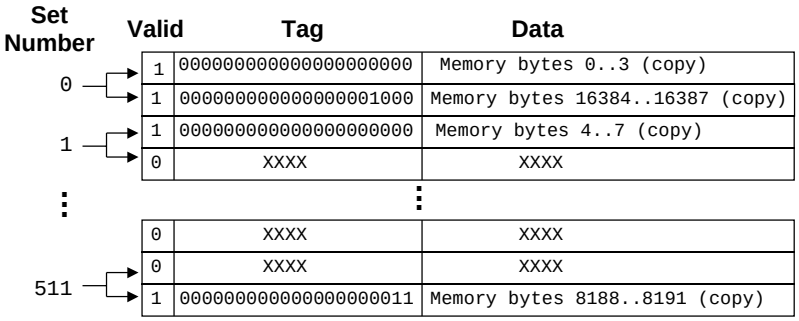


Cache Hit to First Block in Set 0

After Reference 5

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - `0,4,8188,0,16384,0`

Address = 00000000000000001000 000000000 00

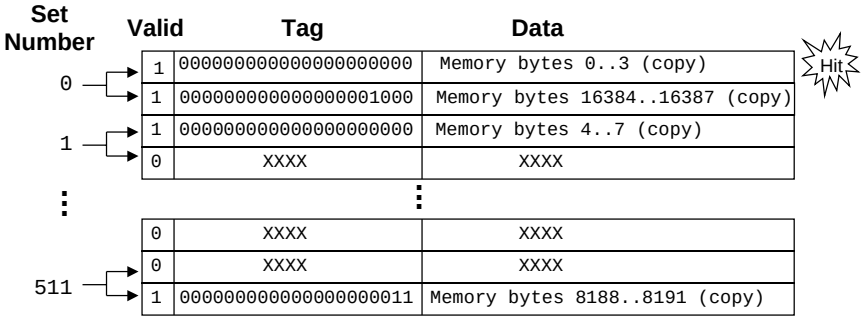


Cache Miss, Place in Second Block of Set 0

After Reference 6

- **Look again at the following sequence of memory references for the previous 2-way associative cache**
  - `0,4,8188,0,16384,0`

Address = 00000000000000000000 000000000 00



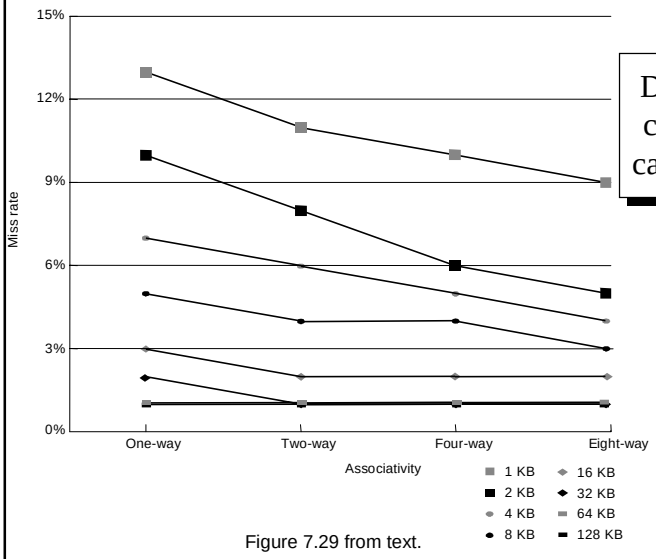
Cache Hit to First Block in Set 0
Total of 2 hits and 4 misses

Miss Rate vs. Set Size

Associativity	Instruction Miss Rate	Data Miss Rate
1	2.0%	1.7%
2	1.6%	1.4%
4	1.6%	1.4%

- **Data is for gcc (compiler execution) for DECStation 3100 with separate code/data 64KB caches using 16B blocks**
- **In general, the benefit increasing associativity beyond 2-4 has minimal impact on miss ratio**
  - 4-way associativity shows more benefit for combined code/data caches

Miss Rate vs. Set Size

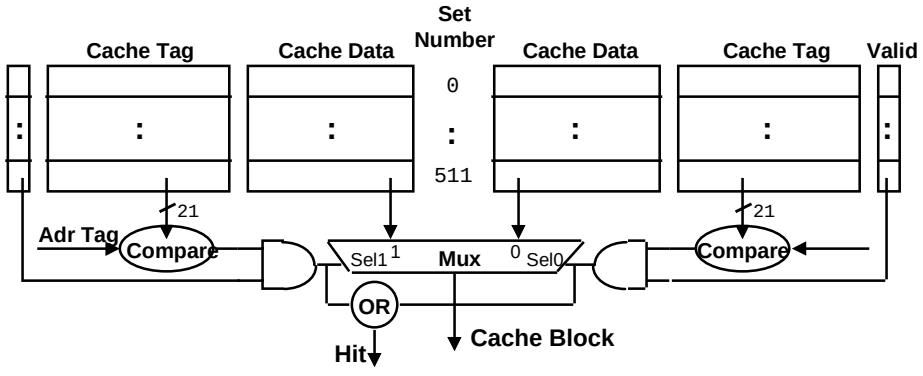


Data for SPEC92 on combined code/data cache with 32B block

Figure 7.29 from text.

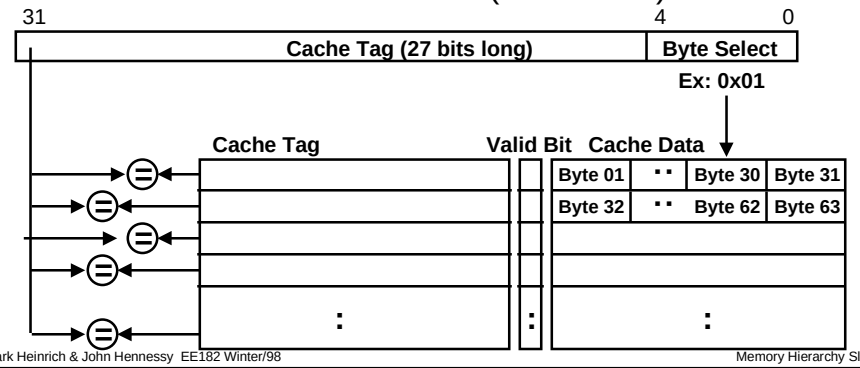
Set-Associative Cache Disadvantages

- **N-way Set Associative vs. Direct Mapped Cache**
  - N comparators vs. 1
  - Extra mux delay for data
  - Data available *after* Hit/Miss
- **Direct mapped cache:**
  - Data available *before* Hit/Miss
    - Assume hit and continue
    - Recover later if miss



Another Extreme: Fully Associative

- **Fully Associative Cache**
  - — push set associative to its limit: only one set!
    - => no set number (or Index)
  - — Compare the Cache Tags of *all* cache entries in parallel
  - — Example: Block Size = 32B blocks => N 27-bit comparators
  - — Generally not used for caches because of cost, but fully-associative translation buffers (cover soon) are common



Cache Miss Classification

- **Start by measuring miss rate with an idealized cache**
  - — Ideal is fully associative and infinite capacity
  - — Then reduce capacity to size of interest
  - — Then reduce associativity to degree of interest
- **Compulsory**
  - — First access to a block => *cold start*
  - — Helps to increase block size and can use *prefetching*
- **Capacity**
  - — Cache cannot contain all blocks accessed by program
  - — Helps to increase cache size
- **Conflict**
  - — Number of memory locations mapped to a set exceeds the set size
  - — Helps to
    - increase cache size because there are more sets
    - increase associativity
- **Invalidation**
  - — Another processor or I/O invalidates the line
  - — Helps to tune allocation and usage of shared data

Cache Block Replacement Policies (Part I)

- **Direct Mapped Cache**
  - — Each memory location mapped to a single cache location
  - — No replacement policy is necessary
    - New item replaces previous item in that cache location
- **Set-Associative Caches**
  - — N-way Set Associative Cache
    - Each memory location has a choice of N cache locations
  - — Fully Associative Cache
    - Each memory location can be placed in any cache location
  - — Cache miss handling for set-associative caches
    - Bring in new block from memory
    - Identify a block in the selected set to replace
    - Need to decide which block to replace

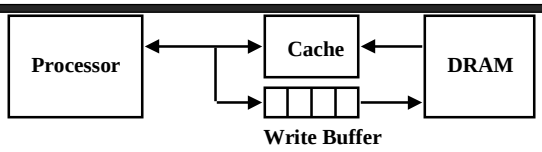
Cache Block Replacement Policies (Part II)

- **Random Replacement**
  - — Hardware randomly selects a cache block to replace
  - — Implementation is actually *pseudo-random*
    - typically selected by counter or pointer
- **Least Recently Used (LRU)**
  - — Hardware keeps track of access history
    - Maintaining complete usage order for set size N requires $\lceil \log_2(N!) \rceil$ bits
    - Replace entry that has not been used for the longest time
  - — Simple for 2-way associative
    - single bit per set records block in the set that was more/less recently used
  - — Generally use *pseudo-LRU* for higher degrees of associativity
    - e.g., 4-way associative uses 3 bits for 2 pairs of entries
      - — 1 bit for each pair tracks LRU within the pair, another bit tracks LRU between the pairs
- **Optimal Replacement**
  - — **Replace the block that will be used latest in the future**
- **In practice replacement policy has minor impact on miss rate**
  - — Especially for high associativity
  - — Surprises possible

Cache Write Policy

- **Cache read much easier to handle than cache write**
  - — Read does not change value of data
- **Cache write**
  - — Need to keep data in the cache and memory consistent
- **Two options**
  - — *Write-Back*: write to cache only.
    - Write the cache block to memory when that cache block is being replaced on a cache miss.
    - Reduces the memory bandwidth required
    - Keep a bit (called the *dirty* bit) per cache block to track whether the block has been modified
      - — Only need to write back modified lines
    - Control can be complex
  - — *Write-Through*: write to both cache and memory
    - What!!! How can this be? Isn't memory too slow for this?

Write Buffer for Write Through



- **Use Write Buffer between the Cache and Memory**
  - — Processor: writes data into the cache and the write buffer
  - — Memory controller: write buffer contents to memory
- **Write buffer is just a FIFO (First-In, First-Out) queue**
  - — Typically small number of entries (4-8)
  - — Works fine if: Rate of stores < 1 / DRAM write cycle
  - — As Rate of stores approaches 1 / DRAM write cycle
    - write buffer fills up
    - stall processor to allow memory to catch up
    - happens due to bursts in write rate even if average rate is OK

Cache Performance

- **CPU Time = (CPU Cycles + Memory Stall Cycles) x Clock cycle time**
- **Memory system affects**
  - — Memory stall cycles
    - cache miss stalls + write buffer stalls
  - — Clock cycle time
    - since cache access often determines clock speed for a processor
- **Memory stall cycles = Read stall cycles + Write stall cycles**
- **Read stall cycles = Read Miss Rate x #Reads x Read Miss Penalty**
- **For write-back cache**
  Write stall cycles = Write Miss Rate x #Writes x Write Miss Penalty
  - — Or can combine read and write components
    - Memory stall cycles = Miss Rate x #Accesses x Miss Penalty
    - Memory stall CPI = Miss Rate/instruction x Miss Penalty
- **For write-through caches**
  - — Add write buffer stalls

Cache Performance: Example

- **Assume:**
  - — Miss rate for instructions = 5%
  - — Miss rate for data = 8%
  - — Data references per instruction = 0.4
  - — CPI with perfect cache = 1.4
  - — Miss penalty = 20 cycles
- **Find performance relative to perfect cache with no misses (same clock rate)**

  Misses/instruction = 0.05 + 0.08 x 0.4 = 0.08

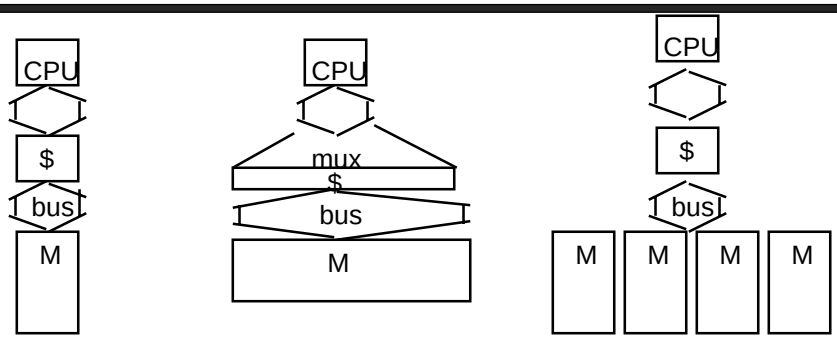
  Miss stall CPI = 0.08 x 20 = 1.6
  - — Performance is ratio of CPIs

$$\frac{\text{Performance no misses}}{\text{Performance with misses}} = \frac{\text{CPI w. misses}}{\text{CPI no misses}} = \frac{1.4 + 1.6}{1.4} = 2.1$$

Improving Cache Performance

- **Cache Performance is determined by**
  Average Memory Access Time = Hit Time + Miss Rate x Miss Penalty
- **Use better technology:**
  - — Use faster RAMs
  - — Cost and availability are limitations
- **Decrease Hit Time**
  - — Make cache smaller, but miss rate increases
  - — Use direct mapped, but miss rate increases
- **Decrease Miss Rate**
  - — Make cache larger, but can increases hit time
  - — Add associativity, but can increase hit time
  - — Increase block size, but increases miss penalty
- **Decrease Miss Penalty**
  - — Reduce transfer time component of miss penalty
  - — Add another level of cache

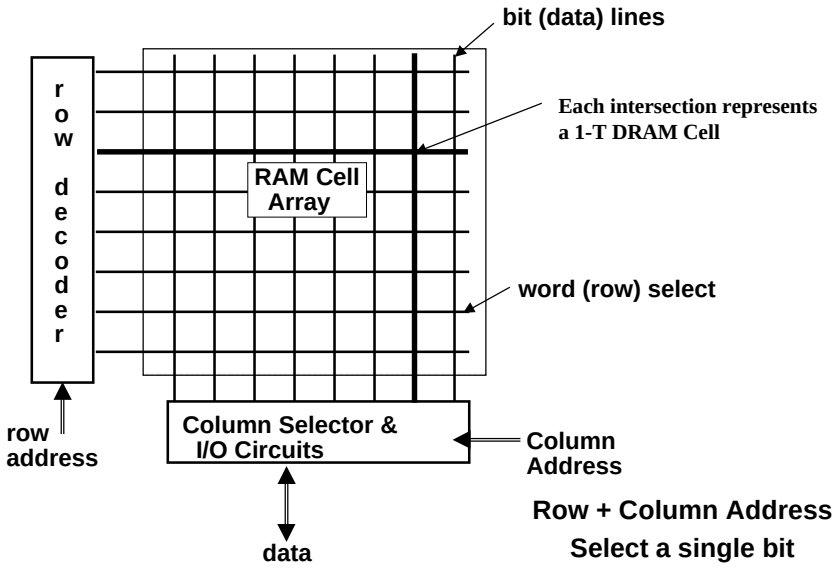
Reducing Memory Transfer Time



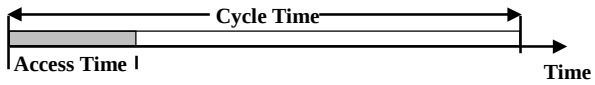
Solution 1 **Solution 2** **Solution 3**
High BW DRAM **Wide Path Between Memory & Cache** **Memory Interleaving**

Examples:
Page Mode DRAM
SDRAM
RAMbus

Classical DRAM Organization



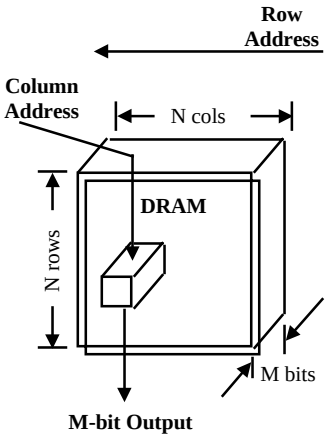
Cycle Time versus Access Time



- **DRAM Cycle Time >> DRAM Access Time**
- **DRAM (Read/Write) Cycle Time :**
  —How frequently can you initiate an access?
- **DRAM (Read/Write) Access Time:**
  —How quickly will you get what you want once you initiate an access?

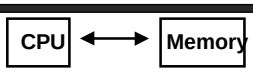
Fast Page Mode DRAM

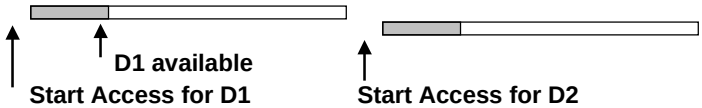
- **Regular DRAM Organization**
  - — N rows x N column x M-bit wide
  - — Read & Write M-bit at a time
  - — Each M-bit access requires RAS / CAS cycle
- **Fast Page Mode DRAM**
  - — N x M “register” to save a row
  - — only CAS is needed



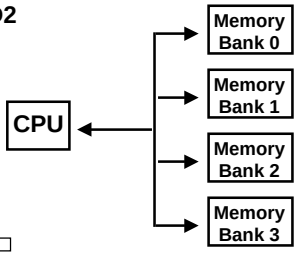
Increasing Bandwidth By Interleaving

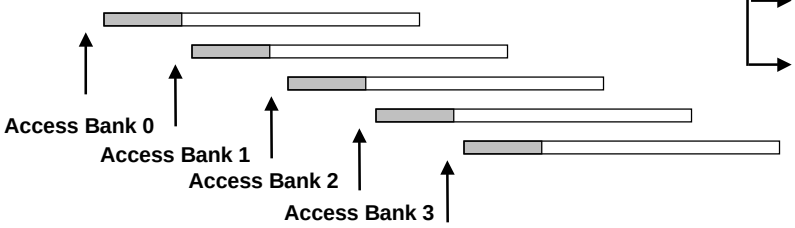
Access Pattern without Interleaving:





Access Pattern with 4-way Interleaving:





Cache Summary

- **Principle of Locality:**
  - — Temporal Locality: Locality in Time
  - — Spatial Locality: Locality in Space
- **Cost/performance tradeoff in memory technologies**
- **Three Major Categories of Cache Misses**
  - — Compulsory Misses: first-time access
  - — Capacity Misses: increase cache size
  - — Conflict Misses: increase associativiy or size
- **Write Policy**
  - — Write-Through: use a write buffer, high memory BW
  - — Write-Back: control can be complex, but low memory BW

Virtual Memory

- **Provides appearance of very large memory**
  - — Total memory for all jobs >> physical memory
  - — Address space of each job > physical memory
- **Allows available (fast and expensive) physical memory to be utilized effectively**
- **Simplifies memory management**
- **Exploits hierarchy to reduce average access time**
- **Uses 2 storage levels**
  - — primary (main) and secondary
- **Distinguish between virtual and physical addresses**
  - — Virtual address is used by the programmer to address memory within a process's address space
  - — Physical address is used by the hardware to access a physical memory location

Virtual Memory (VM) Basics

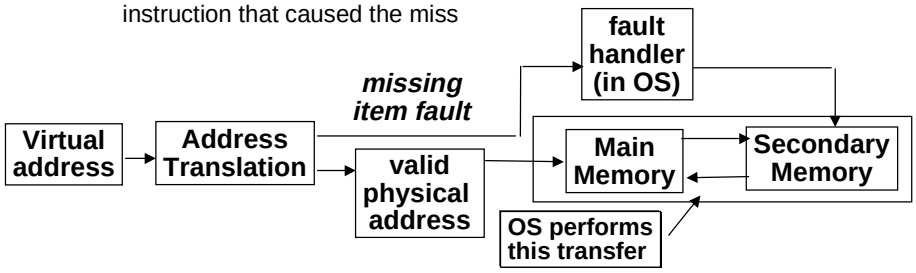
- **Maps virtual addresses to physical addresses**
  - — called (dynamic) address translation
  - — provides protection and sharing between processes
- **Data missing from main memory must be transferred from secondary memory (disk)**
  - — Misses handled by operating system
    - miss time very large so OS manages the hierarchy and schedules another process instead of stalling (context switch)
  - — Architectures support either variable or fixed-size data transfers
    - fixed-size is most common now
  - — Placement of blocks in memory depends on choice of fixed vs. variable
  - — Blocks can be fetched when needed or prefetched in advance

Fixed vs. Variable Blocks

- **Allocation**
  - — Easier to allocate fixed-size blocks because a block from secondary storage can be placed in any free main memory location
  - — Variable-size blocks require search for free main memory location of adequate size
- **Fragmentation => free memory that cannot be allocated**
  - — Variable=Size blocks exhibit *external fragmentation*
    - There may be sufficient free main memory to hold a block, but the free locations are scattered among variable size blocks
  - — Fixed-Size blocks exhibit *internal fragmentation*
    - program is allocated an integral no. of fixed sized blocks, so part of the last block is "wasted"
    - The impact can be minimized by selecting small block sizes relative to program requirements and main memory capacity

Address Translation

- **Program uses virtual addresses**
  - — Relocation => a program can be loaded anywhere in physical memory without recompiling or re-linking
- **Memory accessed with physical addresses**
- **ISA defines the interface between OS and Hardware to specify the virtual-to-physical translation**
  - — When a virtual address is missing from main memory, the OS handles the miss
    - read the missing data, create the translation, return to re-execute the instruction that caused the miss



Exceptions

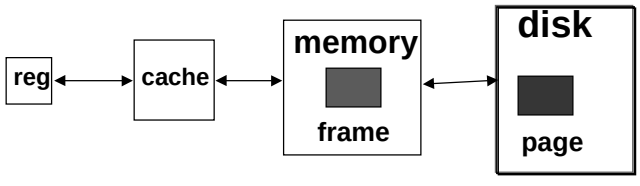
- **Conditions and events that cause transfer to the OS**
  - — Examples
    - integer overflow, translation fault, HW failure, I/O interrupt
- **OS handles exception and may restart execution**
  - — Saves state of executing process, including address of the instruction that caused the exception
  - — Determine cause of exception
    - exception vector address: address where OS is initiated
    - register that records reasons (MIPS Cause register)
  - — May need to restart instruction as if nothing happened
    - Hardest type of exceptions
    - Page faults are like these–transparent to the user
    - Hardware may implement precise exceptions
      - — "Hardware stops on a dime"– no extra instructions executed
    - Or SW may need to do extra work to recover state changes

Implementing Protection

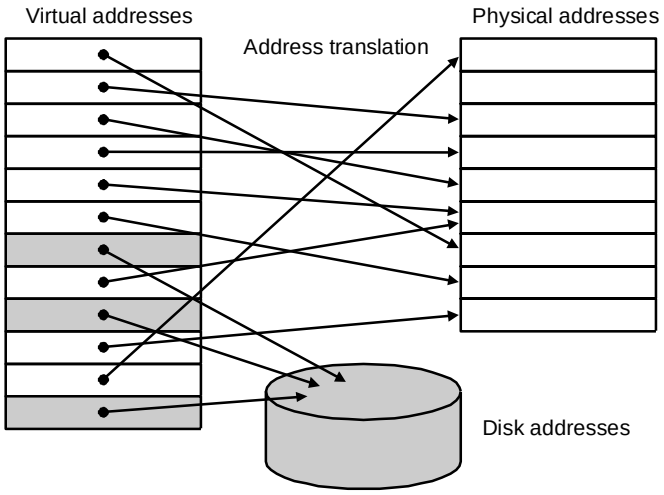
- **The address translation mechanism allows multiple users (and the OS) to use portions of main memory while maintaining protection from other users**
- **Each process has its own address space**
  - process = address space + processor state (registers + PC)
    - address space contains code + data
    - active when the process is running on CPU
      - processor state loaded (including PC), address space mapped
    - process state = everything that needs to be saved and restored to stop and restart process
  - Compiler, linker, and loader are simplified because they see only the virtual address space abstracted from physical memory allocation
- **How does this happen?**
  - Each process has its own translation table set up by the OS
  - Therefore, a process can only access only what the OS allocates and permits

Paged Virtual Memory

- **Most common form of address translation**
  - Virtual and physical address space partitioned into blocks of equal size
  - Virtual address space blocks are called *pages*
  - Physical address space blocks are called *frames* (or *page frames*)
- **Placement**
  - Any page can be placed in any frame (fully-associative)
- **Pages are fetched on demand**



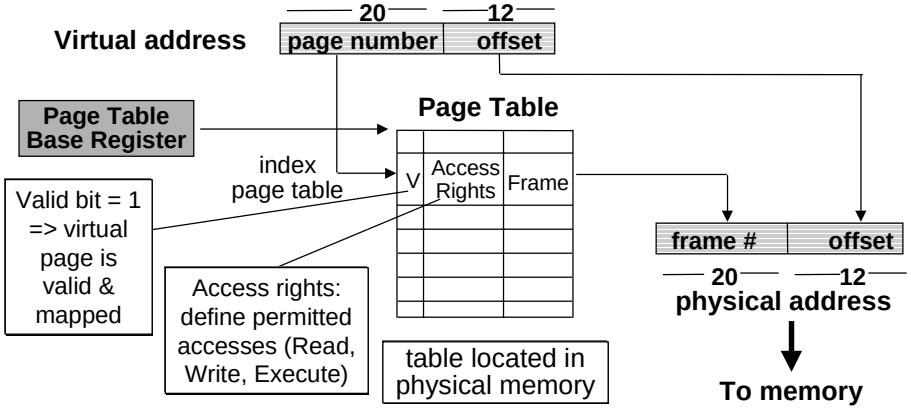
Paging Organization



- **Paging can map any virtual page to any physical frame**

Mapping Process

- **Mapping is done by a page table kept in memory**
- **Page size is power of two:**
  - physical frame address replaces virtual page address
  - lower portion (page offset or displacement) is unchanged



Address Translation Algorithm

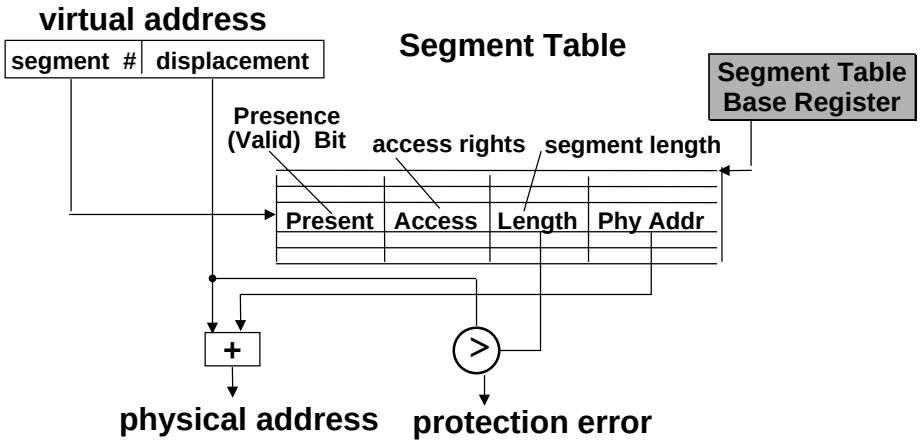
- **If V=1, the mapping is valid**
  - — CPU checks permissions (R,R/W,X) against access type
    - if access is permitted, generates physical address and proceeds
  - — If access is not permitted, generates a protection fault.
- **If V != 1, the mapping is invalid**
  - — CPU generates a page fault
- **Faults are exceptions handled by the OS**
  - — Page faults
    - The OS fetches the missing page, creates a map entry, and restarts the process
    - another user process is switched in to execute while the page is brought from disk
  - — Protection faults
    - Checks whether it is a programming error or permission needs to be changed

Page Replacement and Write Policies

- **When a page fault occurs, choose a page to replace**
  - — Fully associative so any frame/page is a candidate
  - — Choose empty one if it exists.
  - — Choose either (just as we did for cache):
    - LRU: approximated with *Reference* or *Use* bit in page table
      - — true LRU is too hard, since we would need to track the time of use or every page frame
      - — Commonly used *clock algorithms*: resets Use bits periodically
    - Random: approximate as we did for cache
- **Write policy: Always write-back**
  - — Keep a dirty bit
    - Set to 1 if page is modified
    - When a modified page is replaced, the OS writes it back to disk
  - — Many systems use SW page cleaner to write-back modified pages when the disk would be idle
    - Makes selection of replacement more efficient

Segmentation

- **Virtual memory with variable sized *segments***
- **Provides relocation as well as virtualization**
- **Can be applied instead of or combined with paging**



Segmented Address Translation

- **Three major drawbacks:**
  - — Segment register is separate part of the address
    - done to expand address space beyond word length
    - segmentation yields a nonlinear address space
    - segments are visible to programmer, since must specify a segment number and segments are not adjacent
    - unless segment is large enough for all code or data, it creates programming difficulties or inefficiencies
  - — Placing variable sized objects is hard
    - not simple like fixed-sized pages
  - — External fragmentation wastes memory, especially if wide variation in block size
- **Another kind of segmentation**
  - — Fixed-size segments on top of pages (MSBs = segment #)
  - — Goal: to conserve and manage page table space
  - — Called “segments”, but different: none of above drawbacks

Choosing a Page Size

- **Factors favoring large pages**
  - — smaller page tables
  - — more efficient for large programs and large memories
    - fewer page faults, more efficient disk transfer
- **Factors favoring smaller pages**
  - — time to start-up small processes (only a few pages)
  - — internal fragmentation (significant only for small programs)
- **General trend is towards large pages**
  - — 1978: 512 B
  - — 1984: 4 KB
  - — 1990: 16 KB
  - — 199x: 64 KB
- **Many recent machines support multiple page sizes to balance competing factors**

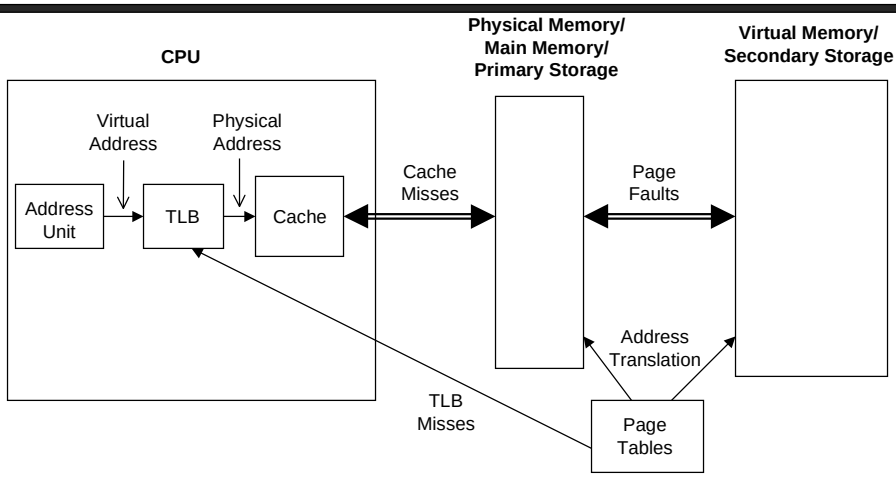
Making VM Fast: TLBs

- **If page table is kept in memory**
  - — all memory references require two accesses
    - one for page table entry and one to get the actual data.
- **Translation Lookaside Buffer (TLB)**
  - — Hardware maintains a cache of recently-used page table translations
  - — Look up all accesses up in TLB
    - Hit in TLB gives the physical page number
    - Miss in TLB => get translation from the page table and reload
  - — TLB usually smaller than cache (each entry maps a full page)
    - more associativity possible and common
    - similar speed to cache access
    - contains all bits needed to translate address, implement VM

Typical TLB Entry

Valid	Virtual Address	Physical Address	Dirty	Access Rights

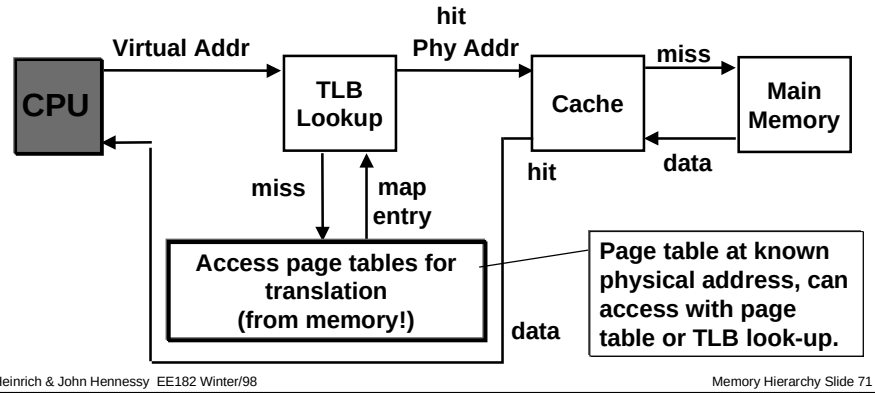
Virtual Memory and Cache



- **OS manages memory hierarchy between secondary storage and main memory**
  - Allocates physical memory to virtual memory and specifies the mapping to hardware through page tables
    - Hardware caches recently used page table entries in the Translation Lookaside Buffer

Memory access with a TLB

- **Access TLB to get physical address**
  - —Miss: access memory to get translation
- **Access cache to get data**
  - —Miss: access memory to get data
- **Optimization: access TLB and cache in parallel**



Common Memory Hierarchy Framework

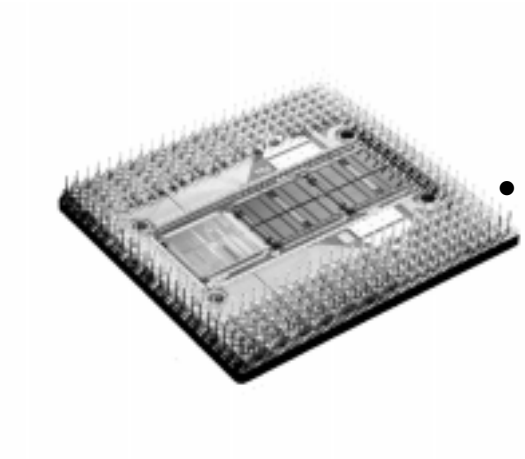
- **Block placement**
  - — Direct-Mapped
  - — Set-Associative
  - — Fully-Associative
- **Replacement policy**
  - — Random
  - — LRU
- **Write policy**
  - — Write-Back
  - — Write-Through

Memory Hierarchy Characteristics

- **Performance and cost drive different characteristics through the hierarchy**
  - — Access times to levels
  - — Size and cost
  - — Function and program behavior

Hierarchy	Blocks	Block Size	Miss Handling
Caches	Fixed	16-128 B	Hardware
Paging	Fixed	4KB-64KB	SW
Segments	Variable		SW
TLBs	Fixed	1-2 PTEs	SW or HW

Pentium Pro Processor Memory Hierarchy



Photograph from Intel.

- **Virtual Memory**
  - — 32-bit virtual address
    32-bit physical address
  - — 4KB page size
  - — TLB organization
    - split code/data TLBs
    - 32-entry code; 64-entry data
    - 4-way associative
    - pseudo-LRU replacement
- **Cache**
  - — Level-1 on CPU-chip
    - split code/data
    - 8KB capacity
    - 32B blocks
    - 4-way associative
    - pseudo-LRU replacement
  - — Level-2 cache off CPU-chip
    - combined code/data
    - 1MB capacity
    - 32B blocks
    - 4-way associative
    - pseudo-LRU replacement

Summary

- **Two Different Types of Locality**
  - — Temporal Locality (Locality in Time
    - The location of a memory reference is likely to be the same as another recent reference
  - — Spatial Locality (Locality in Space)
    - The location of a memory reference is likely to be near another recent reference
- **Memory hierarchies exploit locality**
  - — Present the user with the memory capacity of the least expensive technology at a speed near that of the fastest technology
- **DRAM is inexpensive, but slow**
  - — Good choice for presenting the user with a BIG memory system
- **SRAM is fast, but expensive**
  - — Good choice for providing the user FAST access time.

The Future

- **Memory Hierarchies will continue to be an area of great importance**
  - — Growing gap between CPU performance and disk/DRAM access time
  - — Innovative approaches needed to
    - Avoid degrading performance by the slowest technology
    - Keep costs and power consistent with system requirements
- **There are many issues in memory hierarchies we have not had time to discuss**
  - — multilevel caches
  - — sub-blocking
  - — MP consistency
  - — TLB design
  - — virtual caches
  - — write combining