中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Advanced Computer Architecture
高级计算机体系结构

第4讲： Memory (1)

张献伟

xianweiz.github.io

DCS5367, 10/19/2021

Review(1): Loop Unrolling & Branch

- Loop unrolling[循环展开]
  - Re-order instructions to transform
  - Loop unrolling to expose scheduling opportunities
  - Gains are limited by several factors

- Branch prediction[分支预测]
  - Predict how branches will behave to reduce stalls
  - Basic static predictor
  - Correlating predictors (a.k.a., two-level predictors)
    - *(m, n)*: last *m* branches, *n*-bit predictor for a single branch
  - Tournament predictors
    - Adaptively combining local and global predictors

Review(2): Dynamic Scheduling

- **Static** scheduling: in-order instruction issue and execution
  - If an inst is stalled in pipeline, no later insts can proceed
  - Loop unrolling: reduce stalls by separating dependent insts
    - Static pipeline scheduling by compiler
- **Dynamic** scheduling: in-order issue, OoO execution
  - Reorders the instruction execution to reduce the stalls while maintaining data dependence
  - OoO execution may introduce WAR and WAW hazards
    - Both can be avoided by register renaming
  - *ID* stage is split into two
    - Issue: decode insts, check for structural hazards
    - Read operands: wait until no data hazards, then read operands
  - Scoreboard: a technique for allowing insts to execute OoO when there are sufficient resources and no data dependences

Review(3): Tomasulo

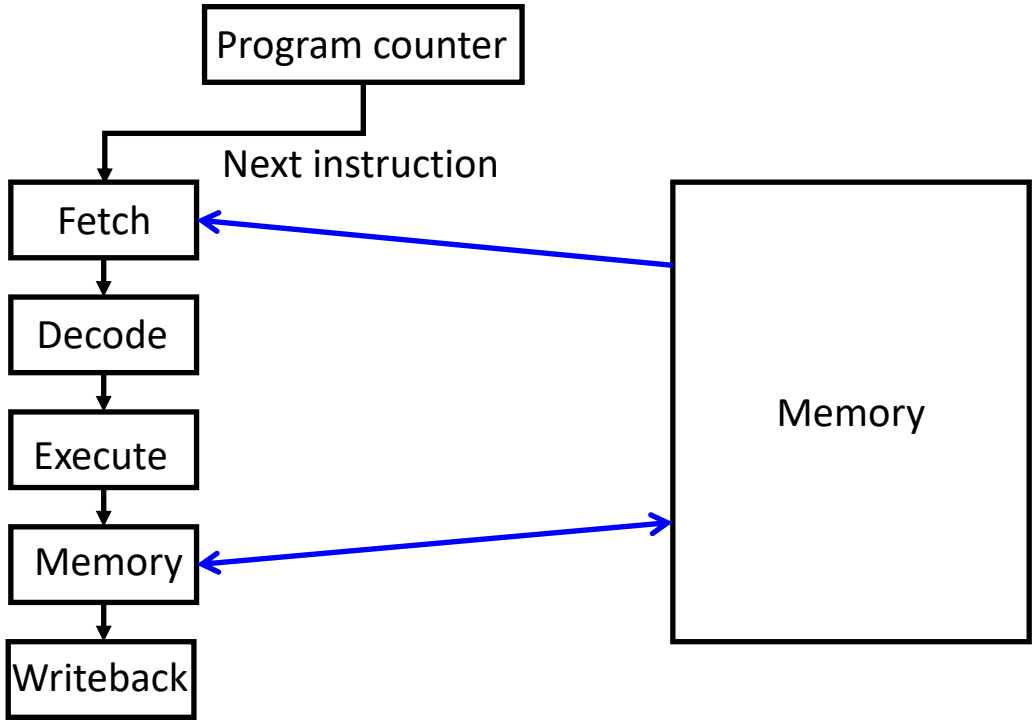
- To support dynamic scheduling
  - Dynamically determining when an inst is ready to execute
  - Avoid unnecessary hazards
    - RAW hazards: avoided by executing an inst only when its operands are available
    - WAR and WAW hazards: eliminated by register renaming
  - Register renaming is provided by reservation stations
- To support speculation
  - Speculate the branch outcome and execute as if guesses are correct
  - Allow insts execute OoO but to force them to commit in order
  - Reorder buffer: hold the results of insts that have finished execution but have not committed
    - Pass results among insts that may be speculated

Review(4): Multiple Issue

- Single issue: ideal CPI of one
  - Issue only one inst every clock cycle
  - Techniques to eliminate data, control stalls
- Multiple issue: ideal CPI less than one
  - Issue multiple insts in a clock cycle
  - **Statically scheduled superscalar** processors
    - Issue varying number of insts per clock, execute in-order
  - **VLIW** (very long inst word) processors
    - Issue a fixed number of insts formatted as one large inst
    - Inherently statically scheduled by the compiler
  - **Dynamically scheduled superscalar** processors
    - Issue varying number of insts per clock, execute OoO

Memory Access[存储访问]

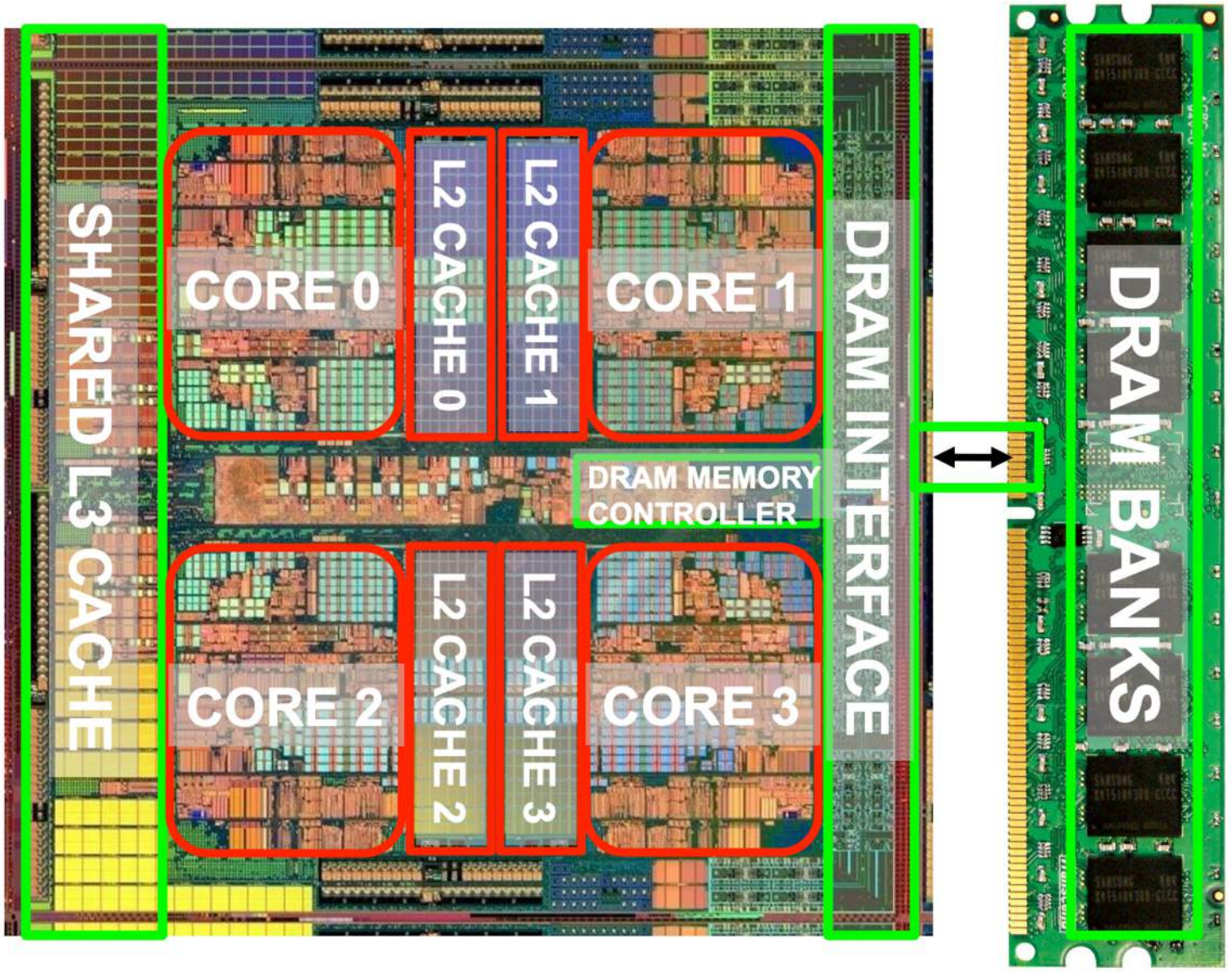
- Programmer's view: read/write (i.e., load/store)
  - Instruction[指令]
  - Data[数据]



Memory[存储]

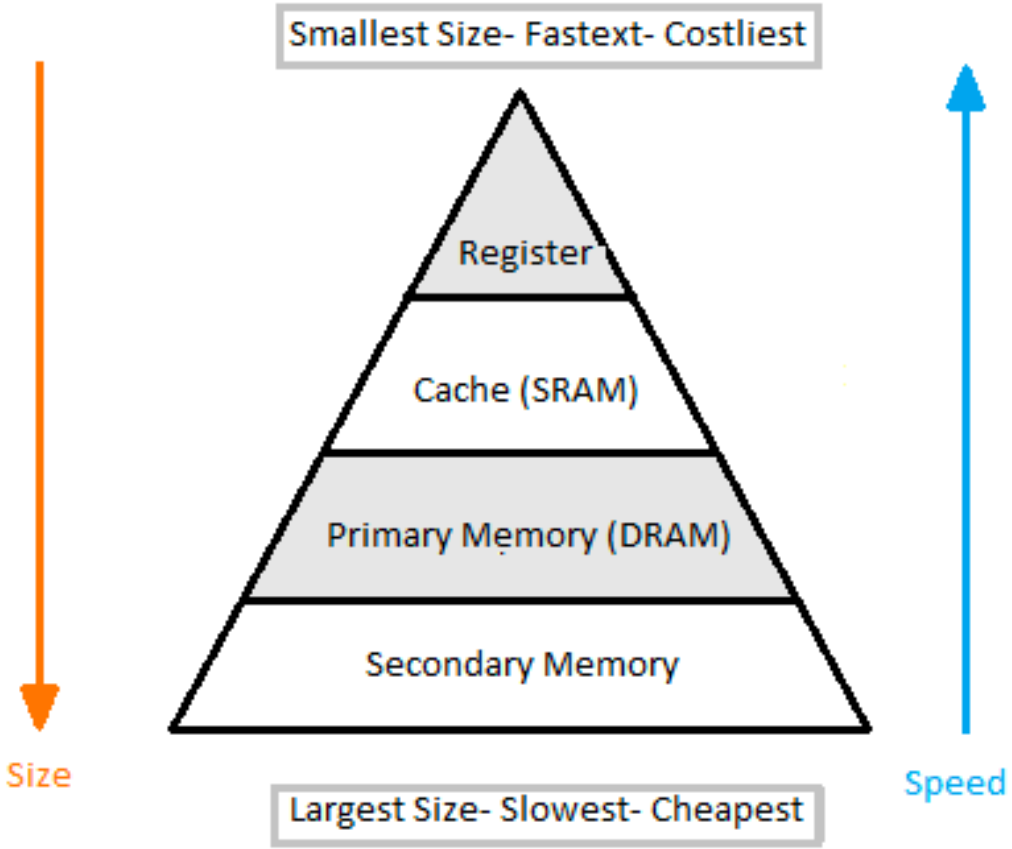
- Ideal memory[理想情况]
  - Zero access time (latency)[零时延]
  - Infinite capacity[无限容量]
  - Zero cost[零成本]
  - Infinite bandwidth (to support parallel accesses)[无限带宽]
- Problem: the requirements are conflicting[问题：需求互斥]
  - Bigger is slower[大容量→长时延]
    - Longer time to determine the location
  - Faster is more expensive[快访问→高成本]
    - More advanced technology
  - Higher bandwidth is more expensive[高带宽→高成本]
    - More access ports, higher frequency, …

Memory in Modern System

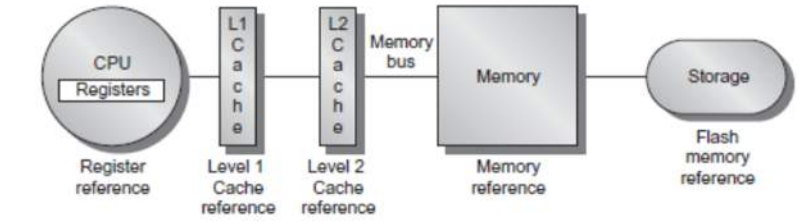


Memory Hierarchy[存储层级]

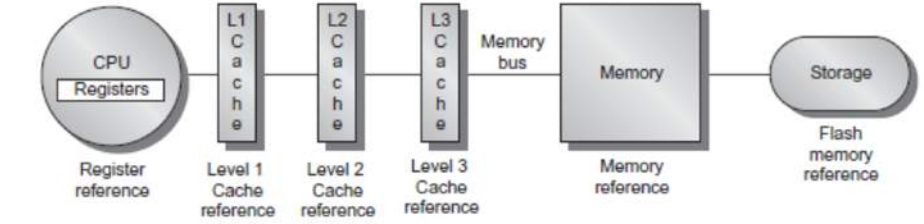
- Goal: provide a memory system with a cost per bit that is almost as low as the cheapest level of memory and a speed almost as fast as the fastest level



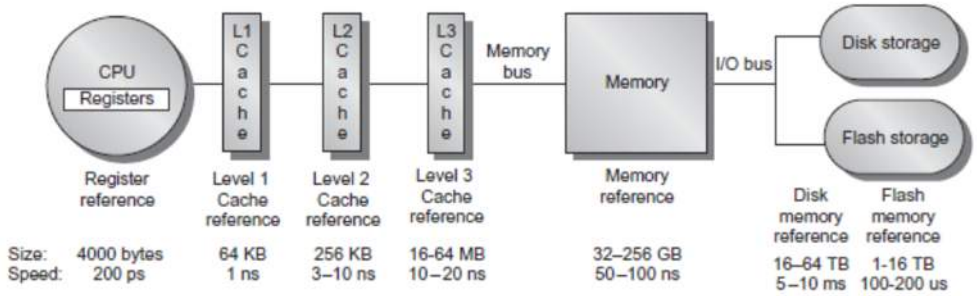
Memory Hierarchy (cont.)



(A) Memory hierarchy for a personal mobile device



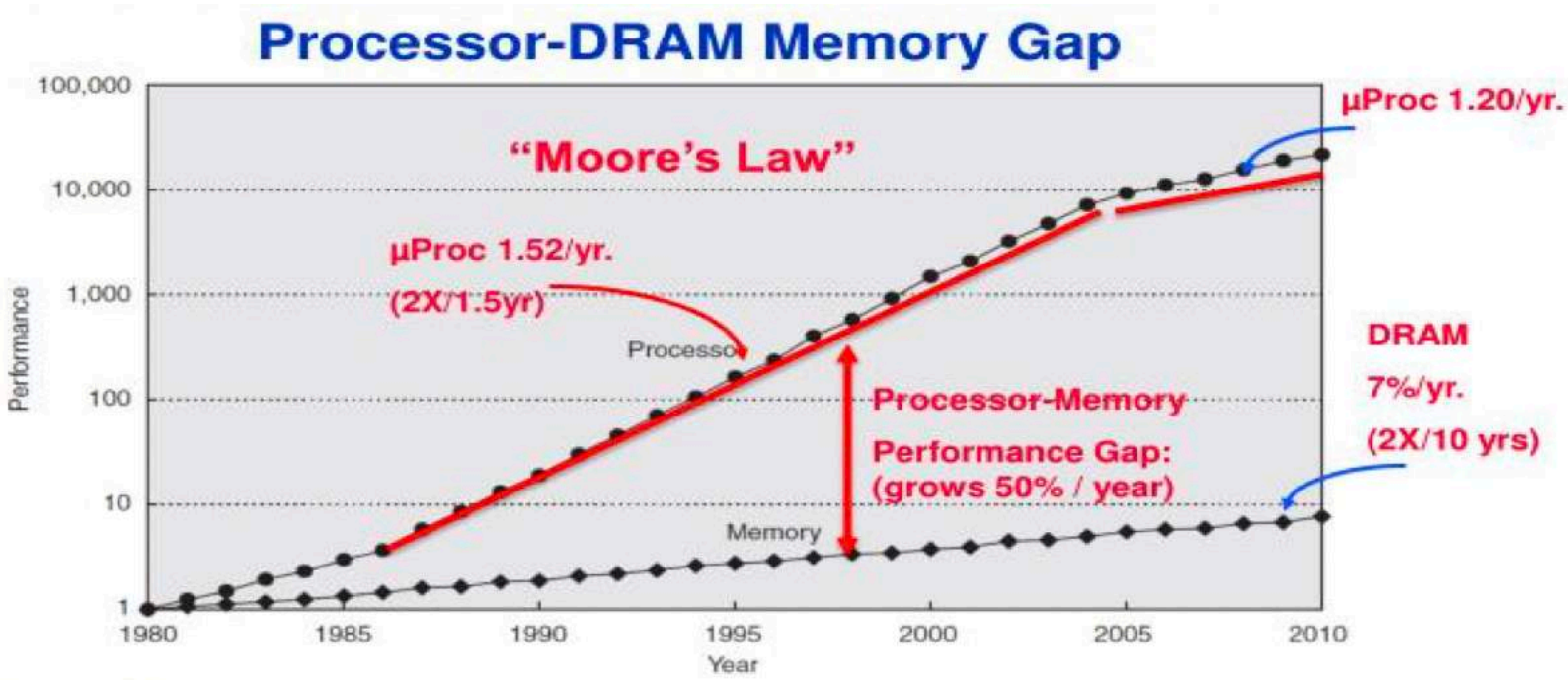
(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

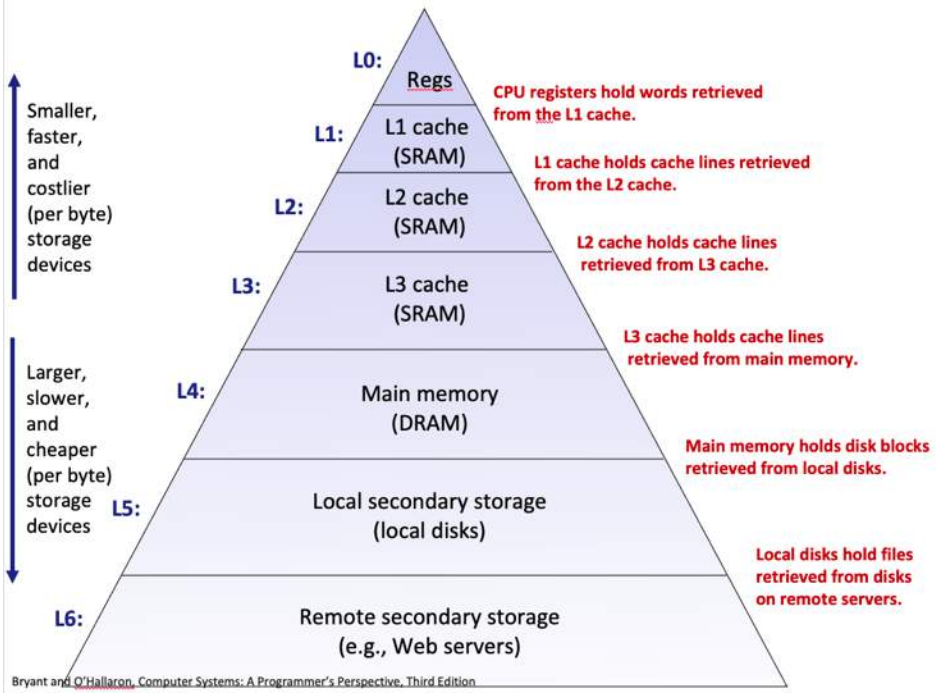
Memory Wall[存储墙]

- On modern machines, most programs that access a lot of data are memory bound
  - Latency of DRAM access is roughly 100-1000 cycles
  - Involves both the limited capacity and the bandwidth of memory transfer



Deeper Hierarchy[更深层级]

- 1980: no cache in micro-processor
- 1989: Intel 486 processor with 8KB on-chip L1 cache
- 1995: Intel Pentium Proc with 256KB on-chip L2 cache
- 2003: Intel Itanium 2 with 6MB on-chip L3 cache
- 2010: 3-level cache on chip, 4th-level cache off chip



Memory Locality[局部性]

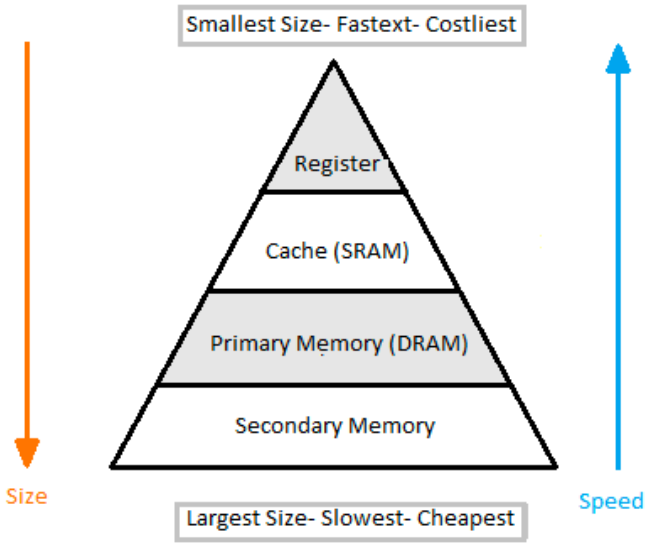
- A “typical” program has a lot of locality in memory references
  - Typical programs are composed of “loops”
- **Temporal**[时间]: a program tends to reference the same memory location many times and all within a small window of time
- **Spatial**[空间]: a program tends to reference a cluster of memory locations at a time
  - Most notable examples:
    - Instruction memory references (sequential execution)
    - Array/data structure references (array traversal)

Caching: Exploit Locality[利用局部性]

- **Temporal**[时间]: recently accessed data will be again accessed in the near future
  - Idea: store recently accessed data in automatically managed fast memory (called cache)
  - Anticipation: the data will be accessed again soon
- **Spatial**[空间]: nearby data in memory will be accessed in the near future (e.g., sequential instruction access, array traversal)
  - Idea: store addresses adjacent to the recently accessed one in automatically managed fast memory
    - logically divide memory into equal size blocks
    - Fetch to cache the accessed block in its entirety
  - Anticipation: nearby data will be accessed soon

Management[管理]

- 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)*



Management Policies[策略]

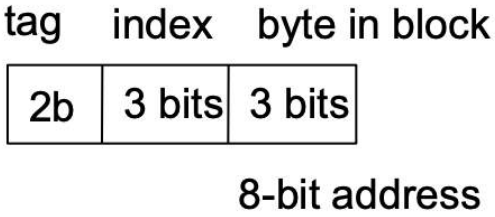
- Manual[手动]: programmer manages data movement across levels
- -- too painful for programmers on substantial programs
  - "core" vs "drum" memory in the 50's
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)
- Automatic[自动]: hardware manages data movement across levels, transparently to the programmer
- ++ programmer's life is easier
  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

Cache Basics[缓存基础]

- **Block** (line): unit of storage in the cache[缓存单位]
  - Memory is logically divided into cache blocks that map to locations in the cache
- When data referenced[使用]
  - HIT: if in cache, use cached data instead of accessing memory
  - MISS: if not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement[放置]: where and how to place/find a block in cache?
  - Replacement[替换]: what data to remove to make room in cache?
  - Granularity of management[粒度]: large, small, uniform blocks?
  - Write policy[写策略]: what do we do about writes?
  - Instructions/data[指令/数据]: do we treat them separately?

Cache Basics (cont.)

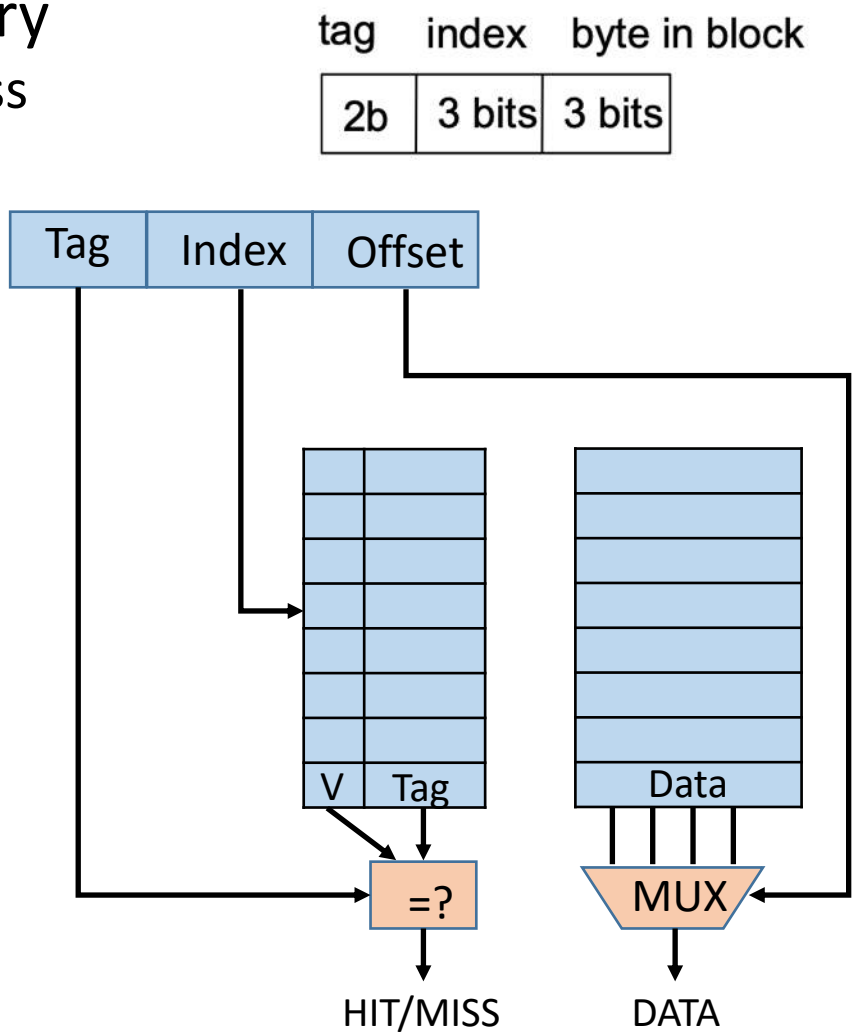
- Memory is logically divided into fixed-size blocks
- Each block maps to a location in the cache, determined by the index bits in the address
  - Used to index into the **tag and data stores**



- Cache access steps
  - 1) index into the tag and data stores with index bits in address
  - 2) check valid bit in tag store
  - 3) compare tag bits in address with the stored tag in tag store
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Cache Basics (cont.)

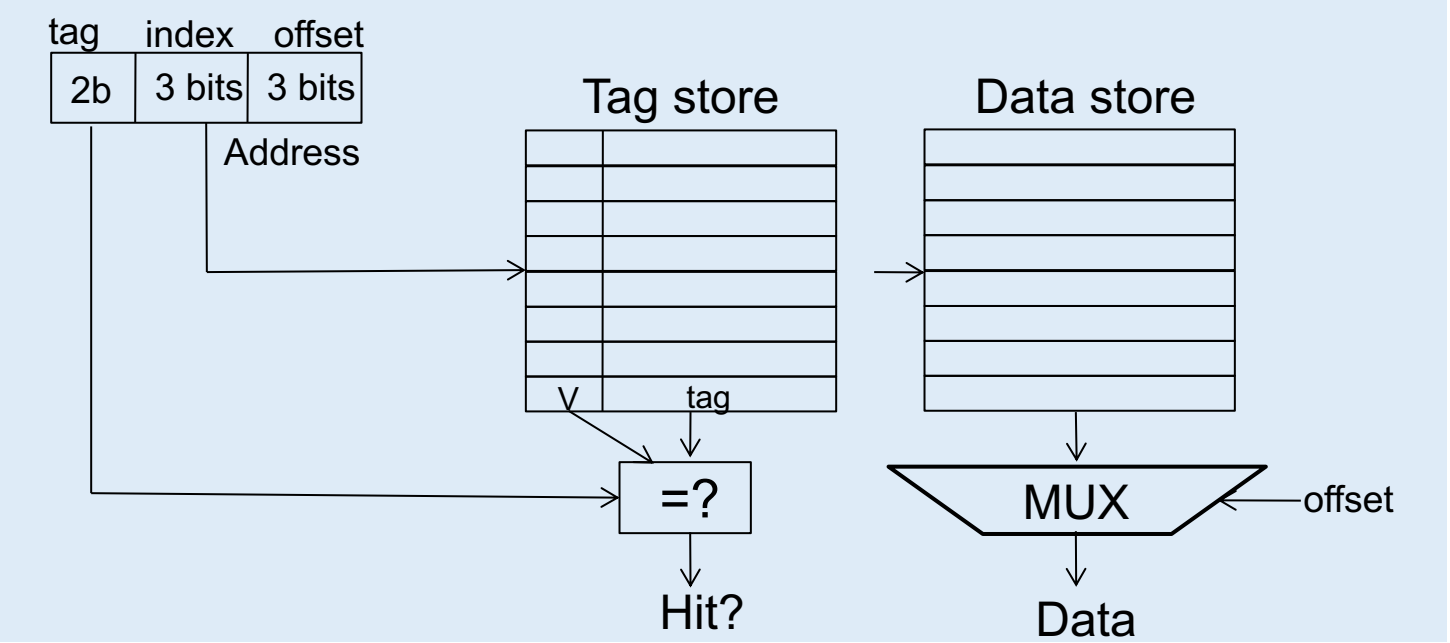
- Assume byte-addressable memory
  - Capacity: 256 bytes → 8-bit address
  - Block: 8 bytes → 3-bit offset
  - #blocks: 32 (256/8)
- Assume cache
  - Capacity: 64 bytes → 3-bit index
    - Holding 8 blocks (64/8)
- What is a tag store?
  - Tag
  - Metadata
    - Valid bit
    - Replacement policy bits
    - Dirty bit
    - ECC



MUX: multiplexer (数据选择器)

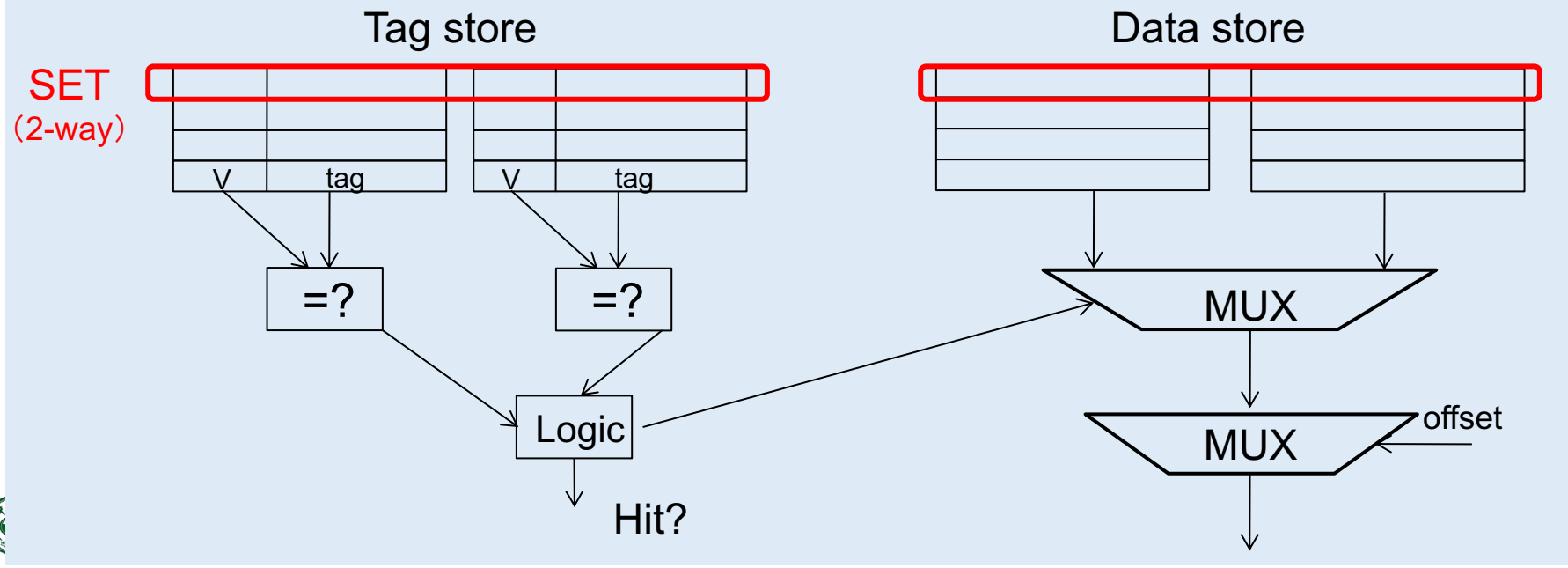
Direct Mapped[直接映射]

- For each item (block) of data in memory, there is **exactly one** location in the cache where it might be
- Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - Addresses A/B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → all misses



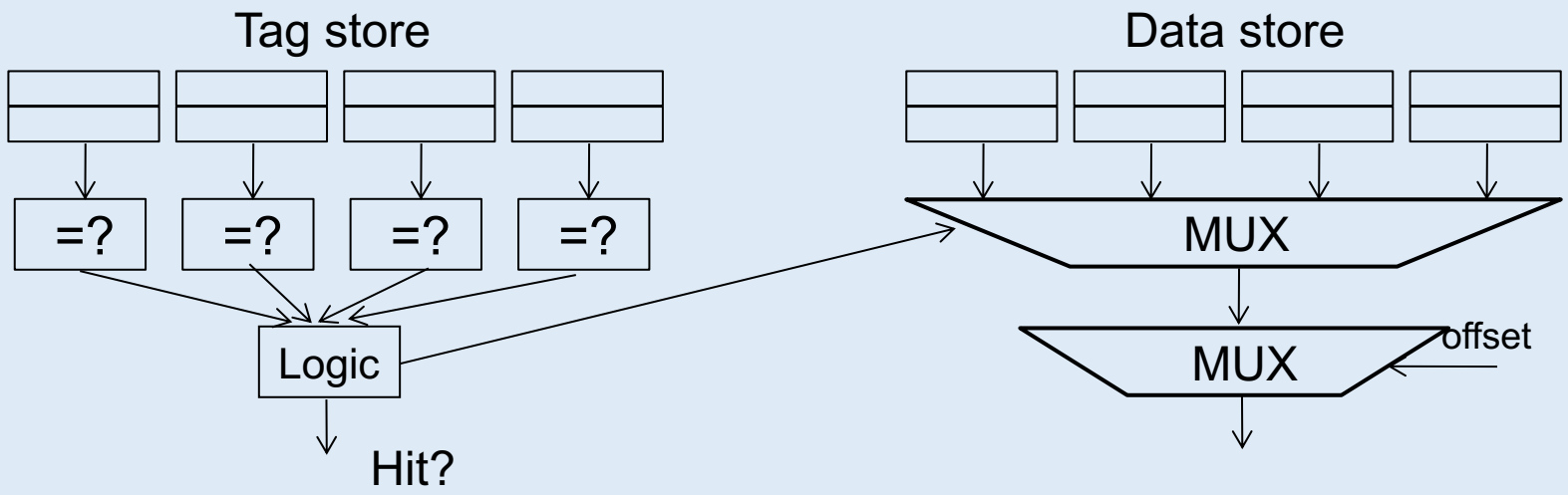
Set-Associative[组相连]

- For the direct mapped
  - Addresses *0* and *8* always conflict in direct mapped cache
  - Instead of having one column of *8* blocks, have *2* columns of *4*
- Key idea: associative memory within the set
  - + Accommodates conflicts better (fewer conflict misses)
  - -- More complex, slower access, larger tag store



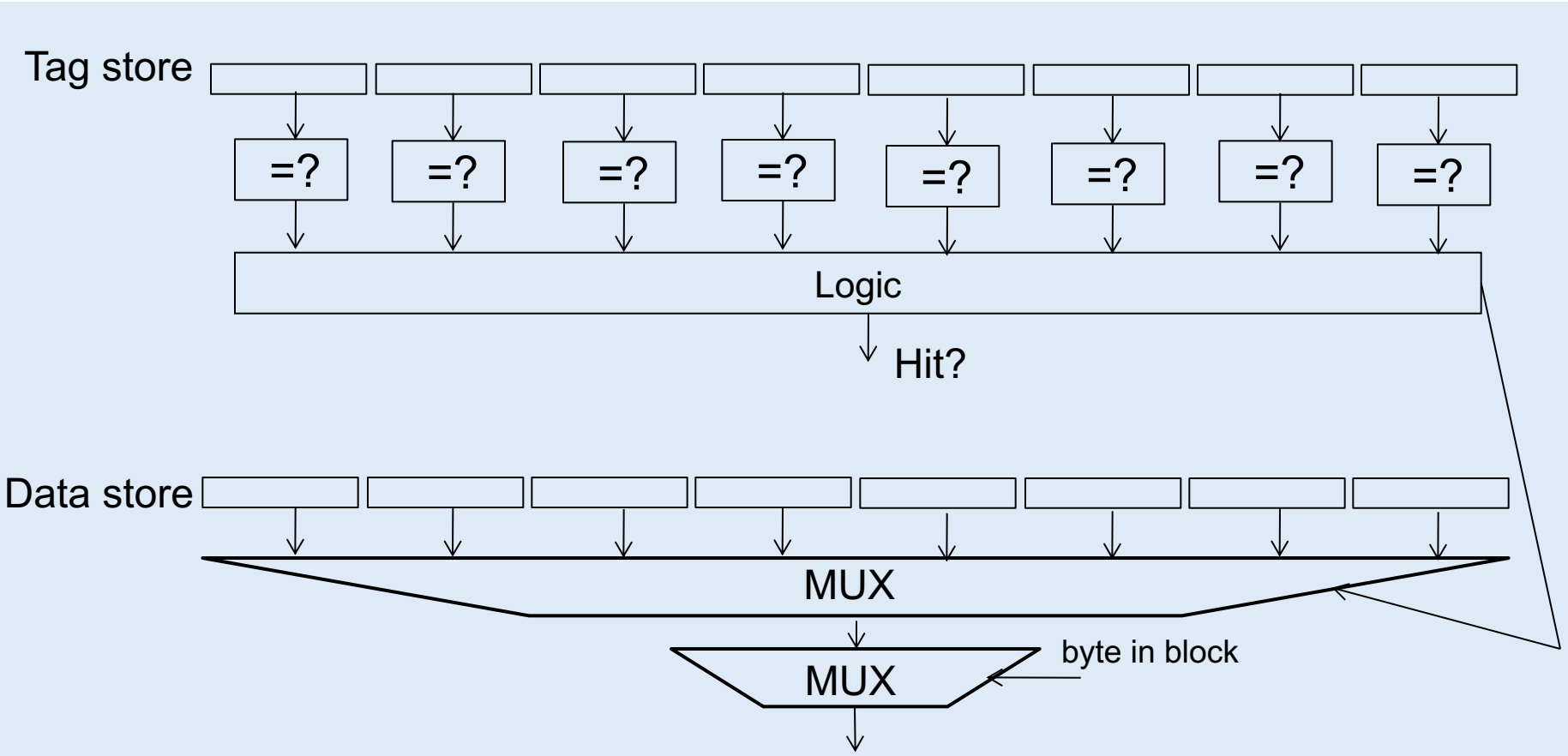
Higher Associativity[高相连度]

- 2-way → 4-way
  - + Likelihood of conflict misses even lower
  - -- More tag comparators and wider data mux
  - -- larger tags



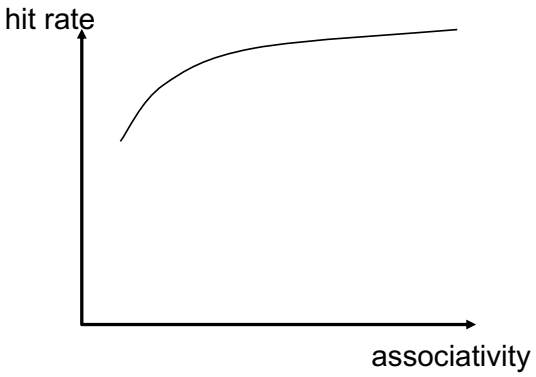
Fully-Associative[全相联]

- A block can map *anywhere* in the cache
  - Most efficient use of space
  - Least efficient to check



Issues of Set-Associative[一些问题]

- Degree of associativity[相连度]: how many blocks can map to the same index (or set)?
- Higher associativity
  - ++ Higher hit rate
  - -- Slower cache access time (hit latency and data access latency)
  - -- More expensive hardware (more comparators)
- Diminishing returns from higher associativity



- Block replacement[块替换]
  - Not an issue for Direct-Mapped
  - Set Associative or Fully Associative
    - Random, LRU (Least Recently Used), FIFO

Handling Writes[写]

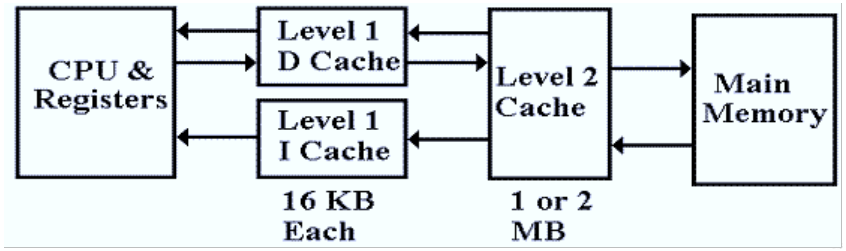
- When do we write the modified data in a cache to next level?
  - Write back: when the block is evicted
  - Write through: at the time the write happens
- Write-back[写回]
  - + Can consolidate multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy
  - -- Need a bit in the tag store indicating the block is "dirty/modified"
- Write-through[写通]
  - + Simpler
  - + All levels are up to date. Consistency: simpler cache coherence because no need to check lower-level caches
  - -- More bandwidth intensive; no coalescing of writes

Handling Writes (cont.)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No
- Allocate on write miss[写分配]
  - + Can consolidate writes instead of writing each of them individually to next level
  - + Simpler because write misses can be treated the same way as read misses
  - -- Requires (?) transfer of the whole cache block
- No-allocate[写不分配]
  - + Conserves cache space if locality of writes is low (potentially better cache hit rate)

Instruction vs. Data Caches

- Separate or Unified?



- Unified[一体]
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  - -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?
- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

Evaluation Metrics[评价指标]

- Cache hit ratio= (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT) = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Cache hit rate: number of misses per kilo instructions

Example: Assume that
Processor speed = 1 GHz (1 n.sec. clock cycle)
Cache access time = 1 clock cycle
Miss penalty = 100 n.sec (100 clock cycles)
I-cache miss ratio = 1%, and D-cache miss ratio = 3%
74% of memory references are for instructions and 26% for data

Effective cache miss ratio = 0.01 * 0.74 + 0.03 * 0.26 = 0.0152
Av. (effective) memory access time = 1 + 0.0152 * 100 = 2.52 cycles = 2.52 n.sec

Improve Cache Performance[性能提升]

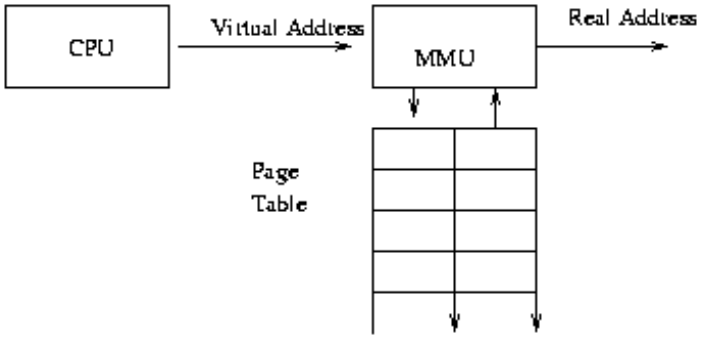
- Reduce the **miss ratio**
  - Larger block size
  - Larger caches
  - Higher associativity
    - But increase hit time and power consumption
- Reduce the **miss penalty**
  - Multi-level caches
  - Read priority over write on miss
    - Serve reads before writes have completed
- Reduce **hit time**
  - Avoiding address translation
    - Just use virtual address

Virtual Memory[虚拟内存]

- Idea: give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory
  - Virtual memory enables each process to have its own unique view of a computer's memory
- **Physical memory** is a storage hardware, made up of physical memory devices, which is organized as an array of *M* contiguous byte-sized cells
  - Each byte has a unique physical address
- Physical vs. virtual address
  - Physical addresses are unique in the system, only used by kernel
  - Virtual memory addresses are unique per-process, used by userspace programs

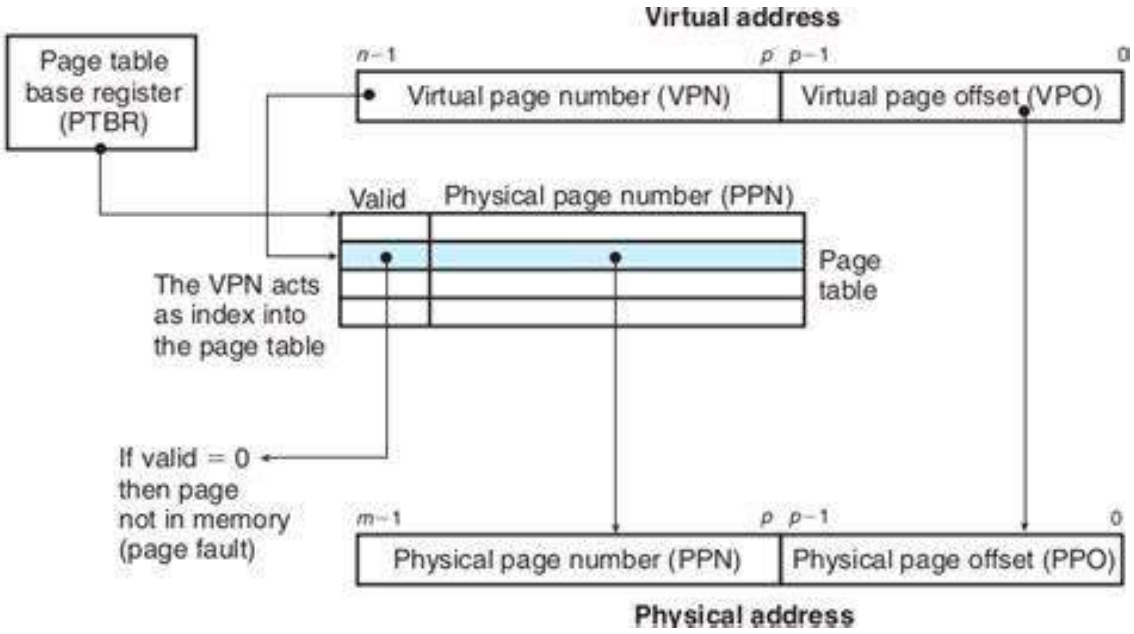
Address Translation[地址转换]

- Address Translation: the hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)
- HW and SW cooperatively manage the translation
  - OS software
  - Address translation hardware in MMU
  - Pages table stored in physical memory or disk
- Memory management unit[内存管理单元]
  - Includes Page Table Base Register(s), TLBs, page walkers



Address Translation (cont.)

- A virtual page is mapped to
  - A physical frame, if the page is in physical memory
  - A location in disk, otherwise
- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical frame and adjusts the mapping → this is called *demand paging*

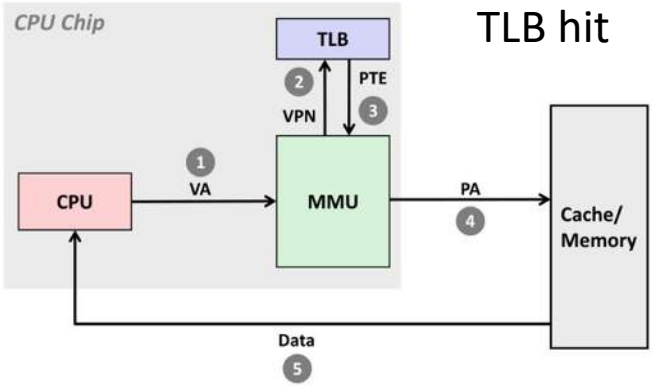


Page Table and TLB[页表]

- Page table is the table that stores the mapping of virtual pages to physical frames
- Page table is just a data structure to map VA (or really VPN) to PA (PFN)
  - Each process has its own set of page tables
  - Page table size for a process is roughly 4MB for 32-bit address space with 4-byte page table entry (PTE)
    - Can be 400MB for 100 processes
- TLB: part of chip's MMU to speed address translation
  - Cache the popular virtual-to-physical address translations
  - Upon each virtual memory reference, the hw first checks the TLB to see if the desired translation is held there
    - If so, the translation is performed without having to consult the page table (which has all translations)

TLB (cont.)

- A typical TLB might have 32, 64, 128 entries, which are *fully associative*
- TLB contains v2p translations that are only valid for the currently running process
  - Those translations are not meaningful for other processes
  - Flush is needed when switching from one process to another
- Accesses to virtual addresses not listed in TLB (a “TLB miss”) trigger a page table lookup
  - Performed either by hw or the page fault handler

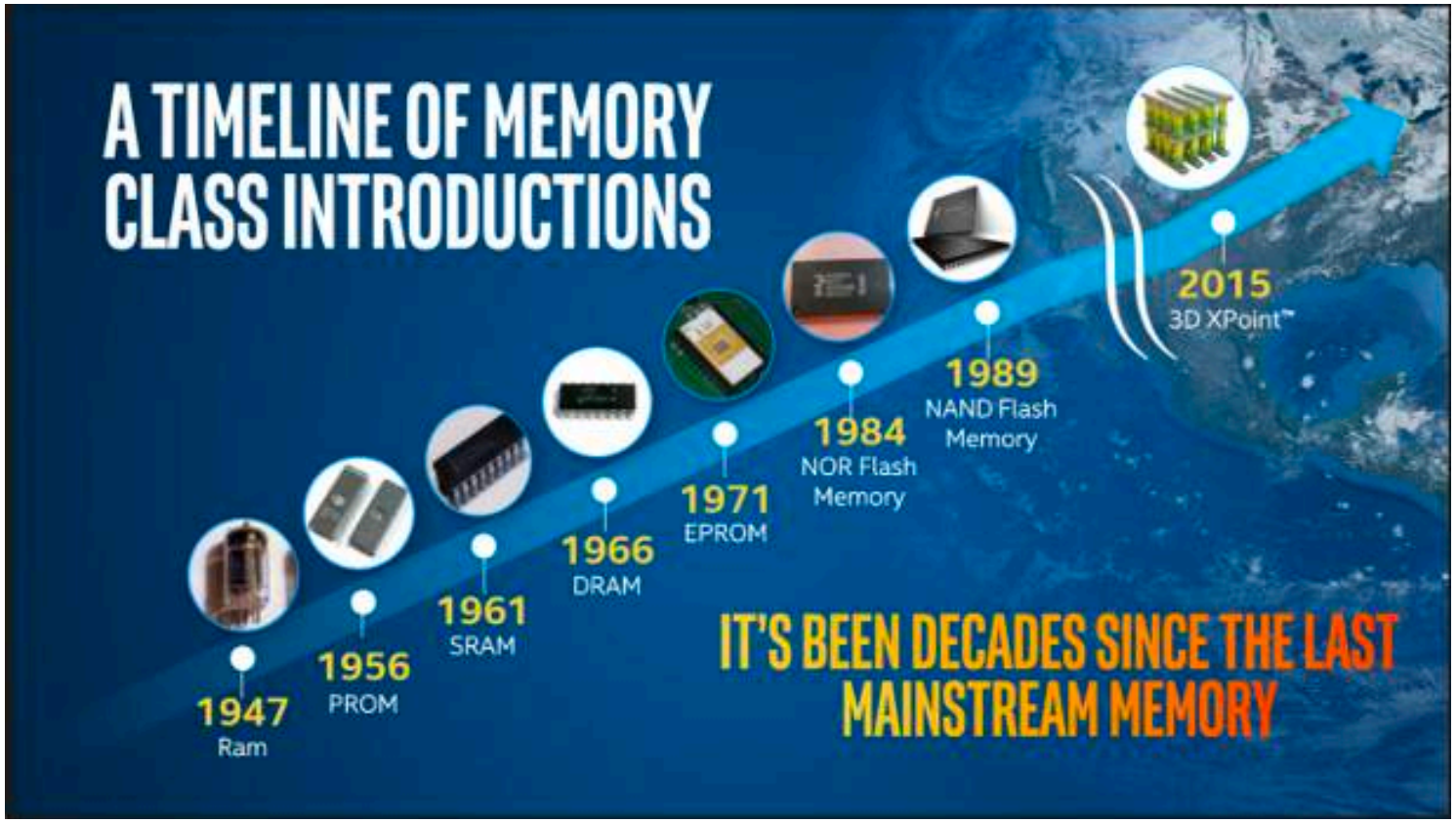


Page Fault[页缺失]

- Physical memory is a cache for pages stored on disk
  - In fact, it is a *fully associative* cache in modern systems (a virtual page can be mapped to any physical frame)
- Page fault: a DRAM cache miss
  - Find out where the contents of the page are stored on disk
  - Possible that this page isn't anywhere at all
    - The memory reference is buggy and thus the process will be killed
- Suppose page fault happens on page *p1*, which is on disk
  - Find page *p2* mapped to some frame *f* that is not used much
  - Copy the contents of frame *f* out to disk
  - Clear page *p2*'s valid bit (subsequent refs to *p2* will cause page faults)
  - Update the MMU's table so that *p1* is mapped to frame *f*
  - Return from the interrupt, allowing the CPU to retry the inst that caused the interrupt

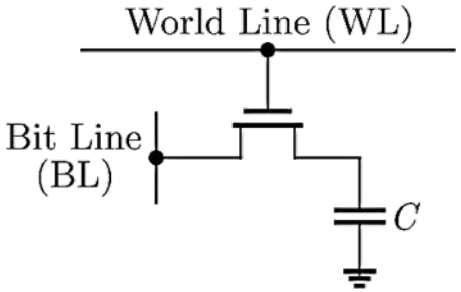
Memory Technology[存储技术]

- Performance of main memory
  - Latency: affects Cache Miss Penalty
  - Bandwidth: affects I/O & Large Block Miss Penalty

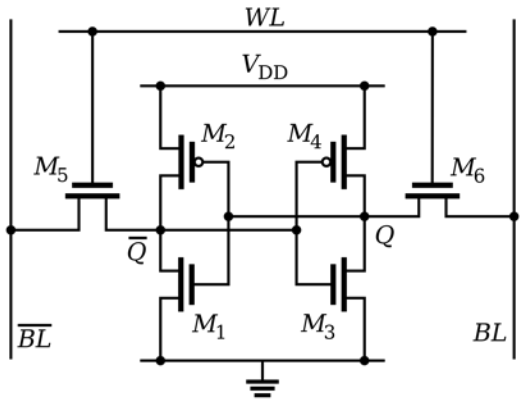


DRAM vs. SRAM

- Main Memory uses *DRAM*: Dynamic Random Access Memory
  - Needs to be refreshed periodically (one row at a time)
  - Addresses divided into 2 halves (memory as a 2D matrix):
    - *RAS* or *Row Access Strobe*
    - *CAS* or *Column Access Strobe*

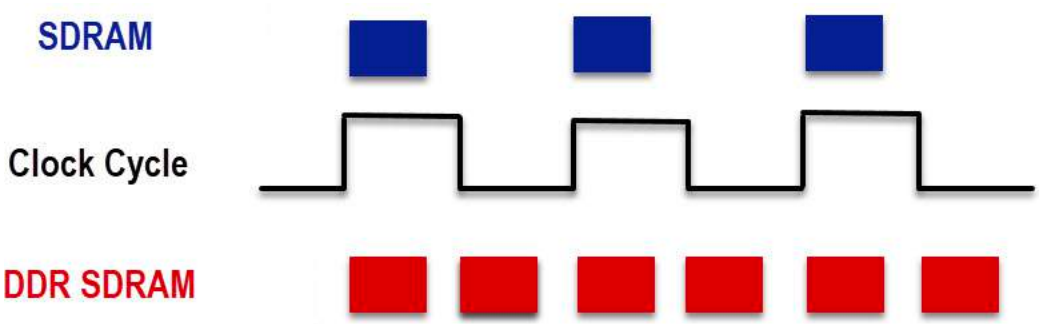


- Cache uses *SRAM*: Static RAM
  - No refresh (6 transistors/bit vs. 1)
    - *Size*: DRAM/SRAM *4-8*
    - *Cost/Cycle time*: SRAM/DRAM *8-16*



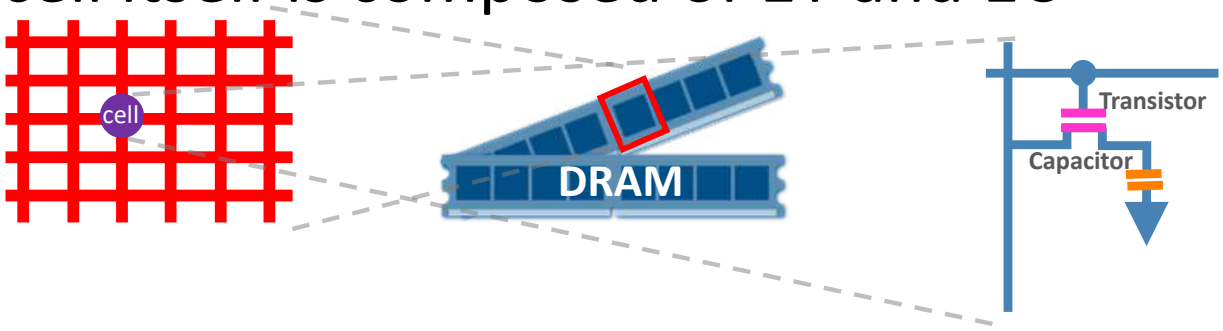
DRAM

- SDRAM = DRAM with a clocked interface
- DDR SDRAM = double data rate, transfer data at both clock edges
  - DDR2 (1.8 V, 266-400 MHz)
  - DDR3 (1.5 V, 800 MHz)
  - DDR4 (1-1.2 V, 1600 MHz)

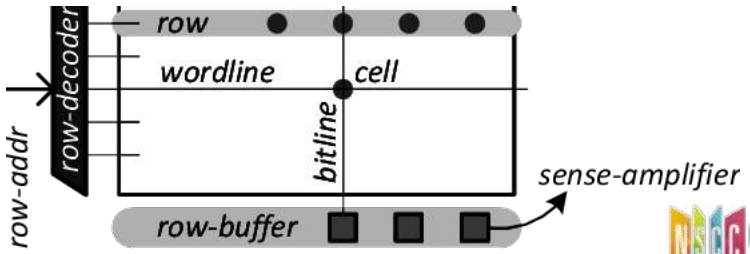


DRAM Structure[结构]

- DRAM is provided as DIMMs, which contain a bunch of chips on each side
- DRAM chip can be thought of as 2D array
- Each intersection in the array is one cell
- The cell itself is composed of 1T and 1C

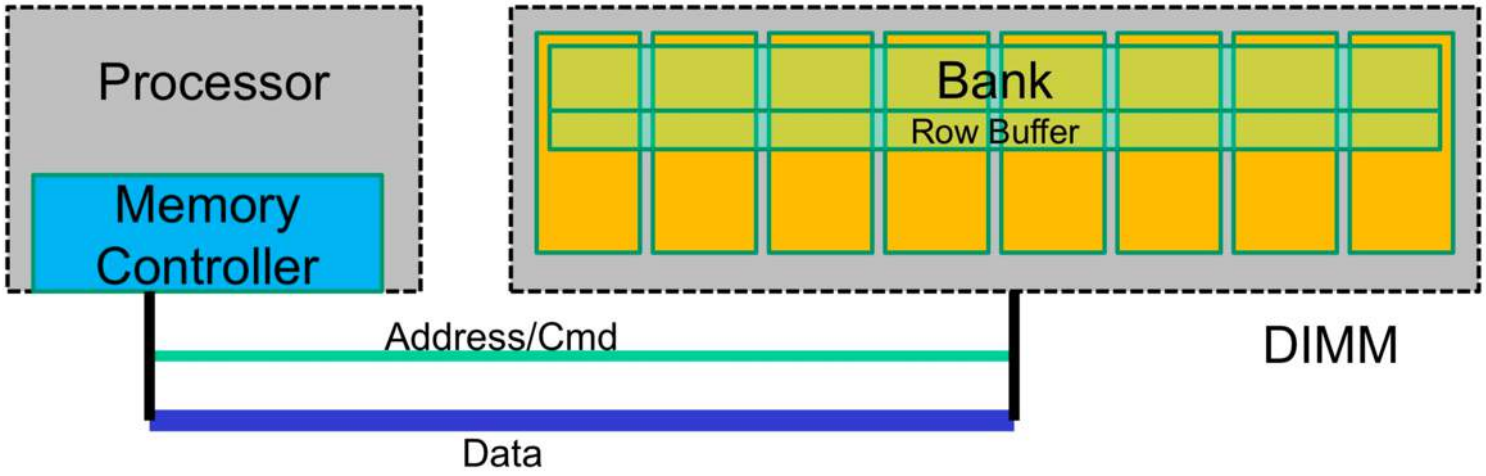


2D Array DIMM/Chip DRAM Cell



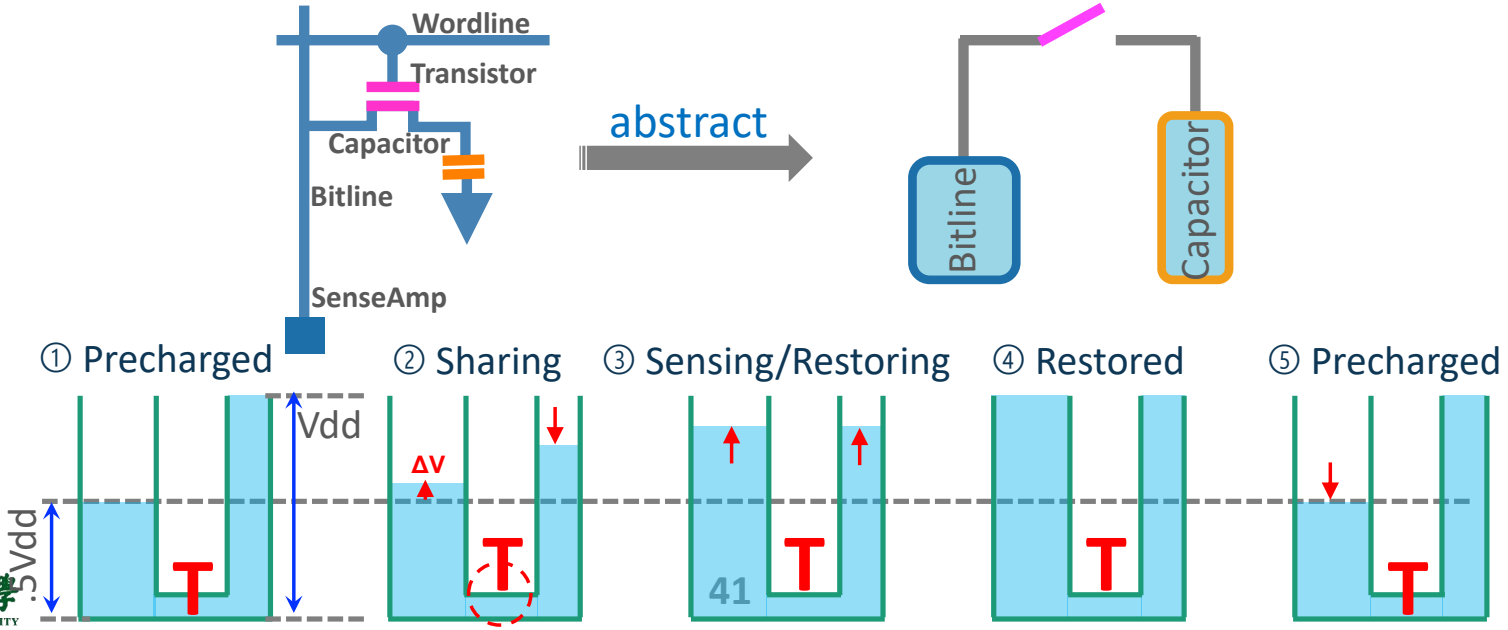
DRAM Structure (cont.)

- A rank consists of multiple (parallel) chips contributing to the same transaction
- A memory chip is organized internally as a number of banks (1-8 usually)
  - Physical bank: chip level, a portion of memory arrays
  - Logical bank: rank level, one physical bank from each chip
- Each memory bank has a "row buffer", which is non-volatile (SRAM registers)



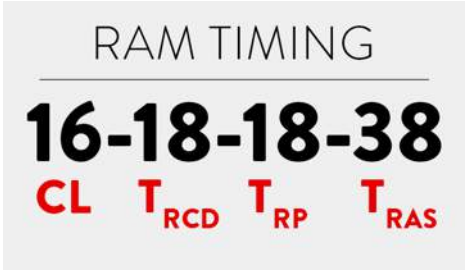
DRAM Operations[操作]

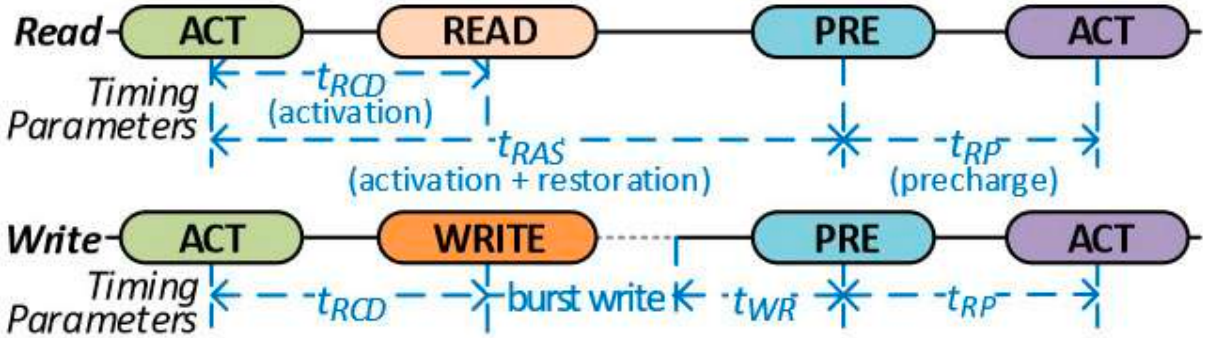
- To read a byte (a similar process applies for writing):
  - The MC sends the row address of the byte
  - The entire row is read into the row buffer (the row is opened)
  - The MC sends the column address of the byte
  - The memory returns the byte to the controller (from the row buffer)
  - The MC sends a Pre-charge signal (close the open row)



Timing Constraints[时序参数]

- Key timings
  - *tRCD*: the minimum number of clock cycles required to open a row and access a column
  - *tCAS*: number of cycles between sending a column address to the memory and the beginning of the data in response
  - *tRAS*: the minimum number of clock cycles required between a row active command and issuing the precharge command
  - *tRP*: number of clock cycles taken between the issuing of the precharge command and the active command
  - *tWR*: write recovery time





Page Mode[页模式]

- A “DRAM row” is also called a “DRAM page”
  - Usually larger than the OS page, e.g., 8KB vs. 4KB
- Row buffers act as a cache within DRAM
- Open page
  - Row buffer hit: ~20 ns access time (must only move data from row buffer to pins)
  - Row buffer conflict: ~60 ns (must first precharge the bitlines, then read new row, then move data to pins)
- Closed page
  - Empty row buffer access: ~40 ns (must first read arrays, then move data from row buffer to pins)
  - Steps
    - Activate command opens row (placed into row buffer)
    - Read/write command reads/writes column in the row buffer
    - Precharge command closes the row and prepares the bank for next access