

18-447: Computer Architecture

Lecture 18: Caches and Main Memory

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2012, 3/28/2012

Reminder: Homeworks

- Homework 5
 - Due [April 2](#)
 - Topics: Out-of-order execution, dataflow, vector processing, memory, caches

Reminder: Lab Assignments

- Lab Assignment 5
 - Implementing caches and branch prediction in a high-level timing simulator of a pipelined processor
 - Due April 6
 - Extra credit: Cache exploration and high performance with optimized caches
- TAs will go over the baseline simulator in the discussion sessions

Review of Last Lecture

- Memory basics
 - DRAM, SRAM, bank organization
 - The memory hierarchy
 - Locality: temporal and spatial

- Caches
 - Basic operation
 - Set associativity
 - Replacement policies
 - Write handling

Today

- Wrap up basics of caches
- Main memory

What's In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
 - Write back vs. write through caches

Handling Writes (Stores)

- When do we write the modified data in a cache to the next level?
 - Write through: At the time the write happens
 - Write back: When the block is evicted
- Write-back
 - Need a bit in the tag store indicating the block is “modified”
 - + Can consolidate multiple writes to the same block before eviction
 - Potentially saves bandwidth between cache levels + saves energy
- Write-through
 - + 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 (Stores)

- 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
 - + 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
 - + Conserves cache space if locality of writes is low (potentially better cache hit rate)

Sectored Caches

- Divide a block into subblocks (or sectors)
 - Associate separate valid and dirty bits for each sector
 - When is this useful?
 - How many subblocks do you transfer

++ No need to transfer the entire cache block into the cache
++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)

-- More complex design

-- May not exploit spatial locality fully when used for reads



Instruction vs. Data Caches

- Unified:
 - + 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
 - for the last reason above
- Second and higher levels are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
 - Decisions very much affected by cycle time
 - Small, lower associativity
 - Tag store and data store accessed in parallel
- Second-level caches
 - Decisions need to balance hit rate and access latency
 - Usually large and highly associative; latency not as important
 - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
 - Serial: Second level cache accessed only if first-level misses
 - Second level does not see the same accesses as the first
 - First level acts as a filter. Can you exploit this fact to improve hit rate in the second level cache?

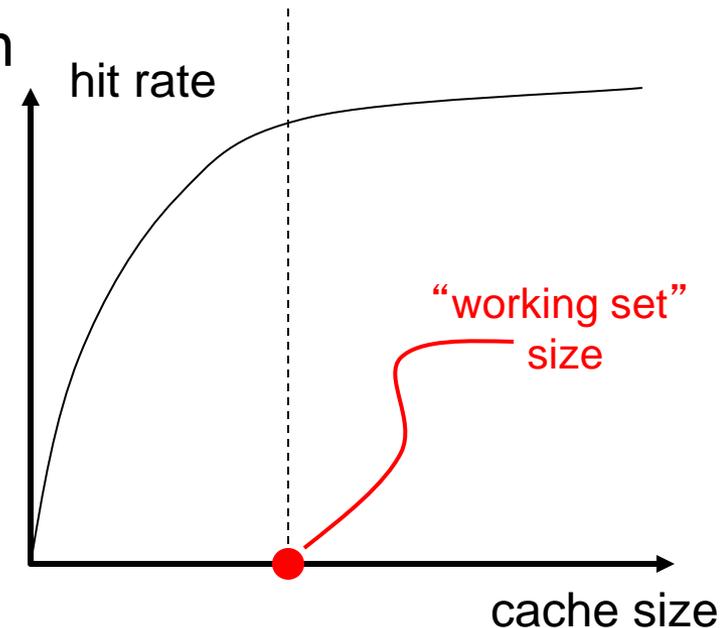
Cache Performance

Cache Parameters vs. Miss Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

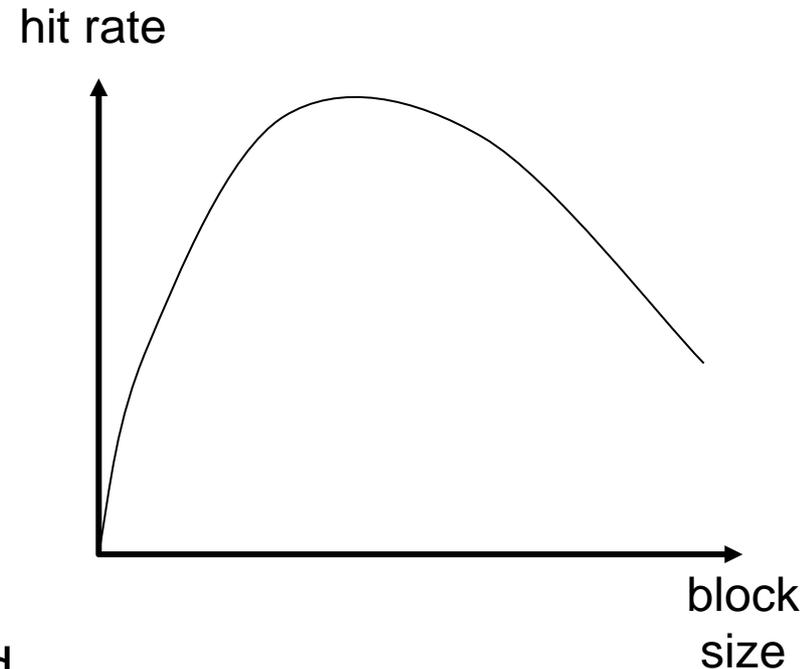
Cache Size

- Cache size in the total data (not including tag) capacity
 - bigger can exploit temporal locality better
 - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
 - smaller is faster => bigger is slower
 - access time may degrade critical path
- Too small a cache
 - doesn't exploit temporal locality well
 - useful data replaced often
- **Working set**: the whole set of data the executing application references
 - Within a time interval



Block Size

- Block size is the data that is associated with an address tag
 - not necessarily the unit of transfer between hierarchies
 - Sub-blocking: A block divided into multiple pieces (each with V bit)
 - Can improve “write” performance
- Too small blocks
 - don't exploit spatial locality well
 - have larger tag overhead
- Too large blocks
 - too few total # of blocks
 - likely-useless data transferred
 - Extra bandwidth/energy consumed



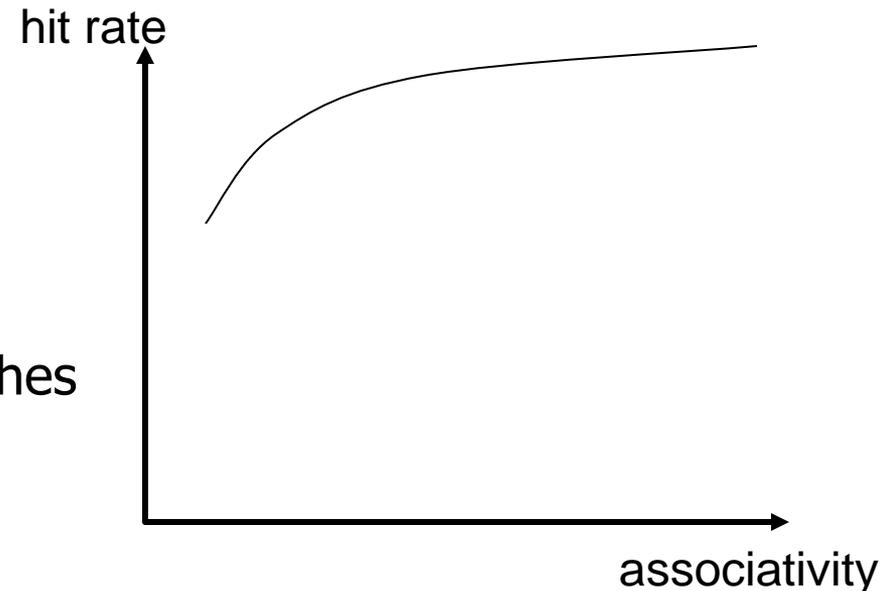
Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
 - fill cache line **critical word first**
 - restart cache access before complete fill
- Large cache blocks can waste bus bandwidth
 - divide a block into subblocks
 - associate separate valid bits for each subblock
 - **When is this useful?**



Associativity

- How many blocks can map to the same index (or set)?
- Larger associativity
 - lower miss rate, less variation among programs
 - diminishing returns
- Smaller associativity
 - lower cost
 - faster hit time
 - Especially important for L1 caches
- Power of 2 associativity?



Classification of Cache Misses

- Compulsory miss
 - ❑ first reference to an address (block) always results in a miss
 - ❑ subsequent references should hit unless the cache block is displaced for the reasons below
 - ❑ dominates when locality is poor
- Capacity miss
 - ❑ cache is too small to hold everything needed
 - ❑ defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Conflict miss
 - ❑ defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- Compulsory
 - Caching cannot help
 - Prefetching
- Conflict
 - More associativity
 - Other ways to get more associativity without making the cache associative
 - Victim cache
 - Hashing
 - Software hints?
- Capacity
 - Utilize cache space better: keep blocks that will be referenced
 - Software management: divide working set such that each “phase” fits in cache

Improving Cache “Performance”

- Reducing miss rate
 - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency
- Reducing hit latency
- Remember
 - Average memory access time (AMAT)
= (hit-rate * hit-latency) + (miss-rate * miss-latency)

Improving Basic Cache Performance

- Reducing miss rate
 - More associativity
 - Alternatives/enhancements to associativity
 - Victim caches, hashing, pseudo-associativity, skewed associativity
 - Software approaches

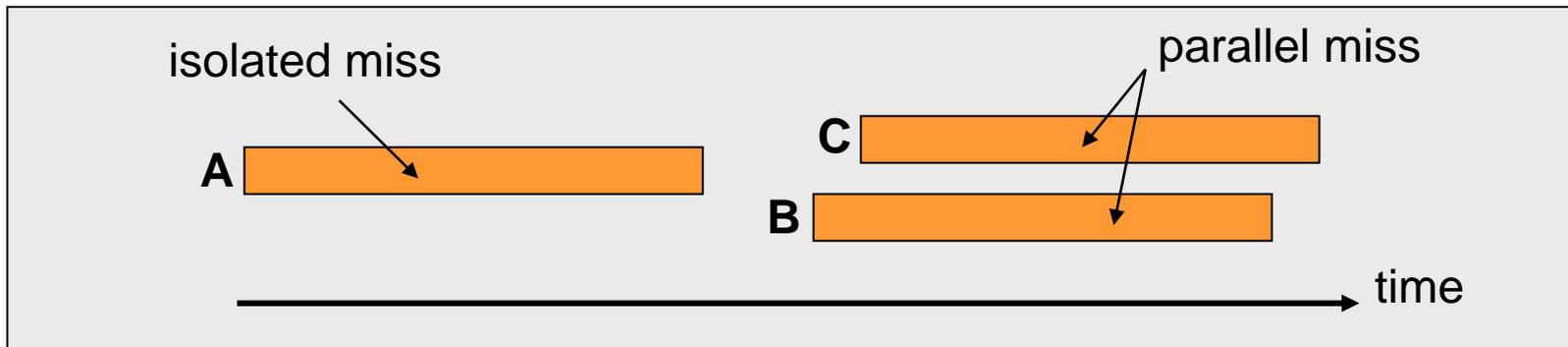
- Reducing miss latency/cost
 - Multi-level caches
 - Critical word first
 - Subblocking/sectoring
 - Non-blocking caches (multiple cache misses in parallel)
 - Multiple accesses per cycle
 - Software approaches

Handling Multiple Outstanding Accesses

- **Non-blocking** or **lockup-free** caches
 - Kroft, “**Lockup-Free Instruction Fetch/Prefetch Cache Organization**,” ISCA 1981.
- Question: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?
- Idea: **Keep track of the status/data of misses that are being handled in Miss Status Handling Registers (MSHRs)**
 - A cache access checks MSHRs to see if a miss to the same block is already *pending*.
 - If pending, a new request is not generated
 - If pending and the needed data available, data forwarded to later load
 - Requires buffering of outstanding miss requests

Non-Blocking Caches (and MLP)

- Enable cache access when there is a pending miss
- Enable multiple misses in parallel
 - **Memory-level parallelism (MLP)**
 - generating and servicing multiple memory accesses in parallel
 - Why generate multiple misses?



- Enables latency tolerance: **overlaps latency of different misses**
- How to generate multiple misses?
 - Out-of-order execution, multithreading, runahead, prefetching

Miss Status Handling Register

- Also called “miss buffer”
- Keeps track of
 - Outstanding cache misses
 - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR
 - Valid bit
 - Cache block address (to match incoming accesses)
 - Control/status bits (prefetch, issued to memory, which subblocks have arrived, etc)
 - Data for each subblock
 - For each pending load/store
 - Valid, type, data size, byte in block, destination register or store buffer entry address

Miss Status Handling Register

| | | |
|-------|---------------|--------|
| 1 | 27 | 1 |
| Valid | Block Address | Issued |

| | | | | |
|-------|------|--------------|-------------|--------------|
| 1 | 3 | 5 | 5 | |
| Valid | Type | Block Offset | Destination | Load/store 0 |
| Valid | Type | Block Offset | Destination | Load/store 1 |
| Valid | Type | Block Offset | Destination | Load/store 2 |
| Valid | Type | Block Offset | Destination | Load/store 3 |

MSHR Operation

- On a cache miss:
 - Search MSHR for a pending access to the same block
 - Found: Allocate a load/store entry in the same MSHR entry
 - Not found: Allocate a new MSHR
 - No free entry: stall
- When a subblock returns from the next level in memory
 - Check which loads/stores waiting for it
 - Forward data to the load/store unit
 - Deallocate load/store entry in the MSHR entry
 - Write subblock in cache or MSHR
 - If last subblock, deallocate MSHR (after writing the block in cache)

Non-Blocking Cache Implementation

- When to access the MSHRs?
 - In parallel with the cache?
 - After cache access is complete?
- MSHRs need not be on the critical path of hit requests
 - Which one below is the common case?
 - Cache miss, MSHR hit
 - Cache hit

Enabling High Bandwidth Caches (and Memories in General)

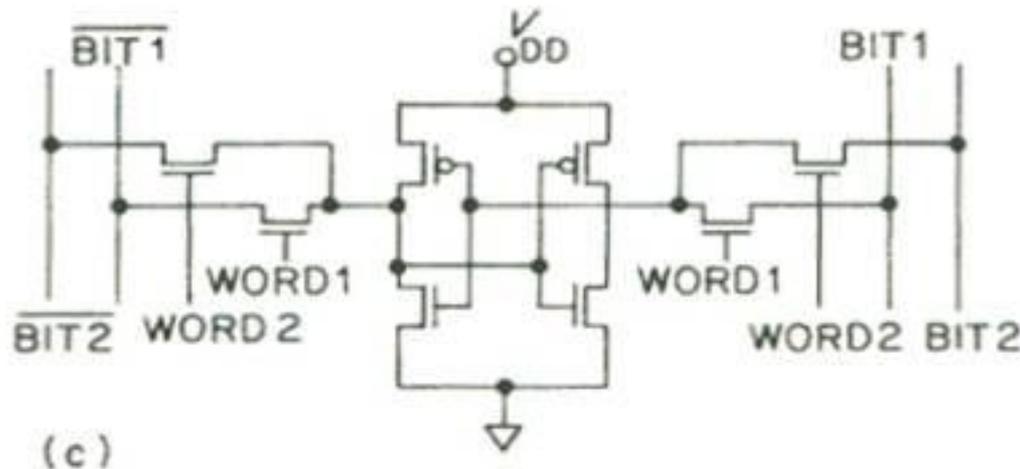
Multiple Instructions per Cycle

- Can generate multiple cache accesses per cycle
- How do we ensure the cache can handle multiple accesses in the same clock cycle?

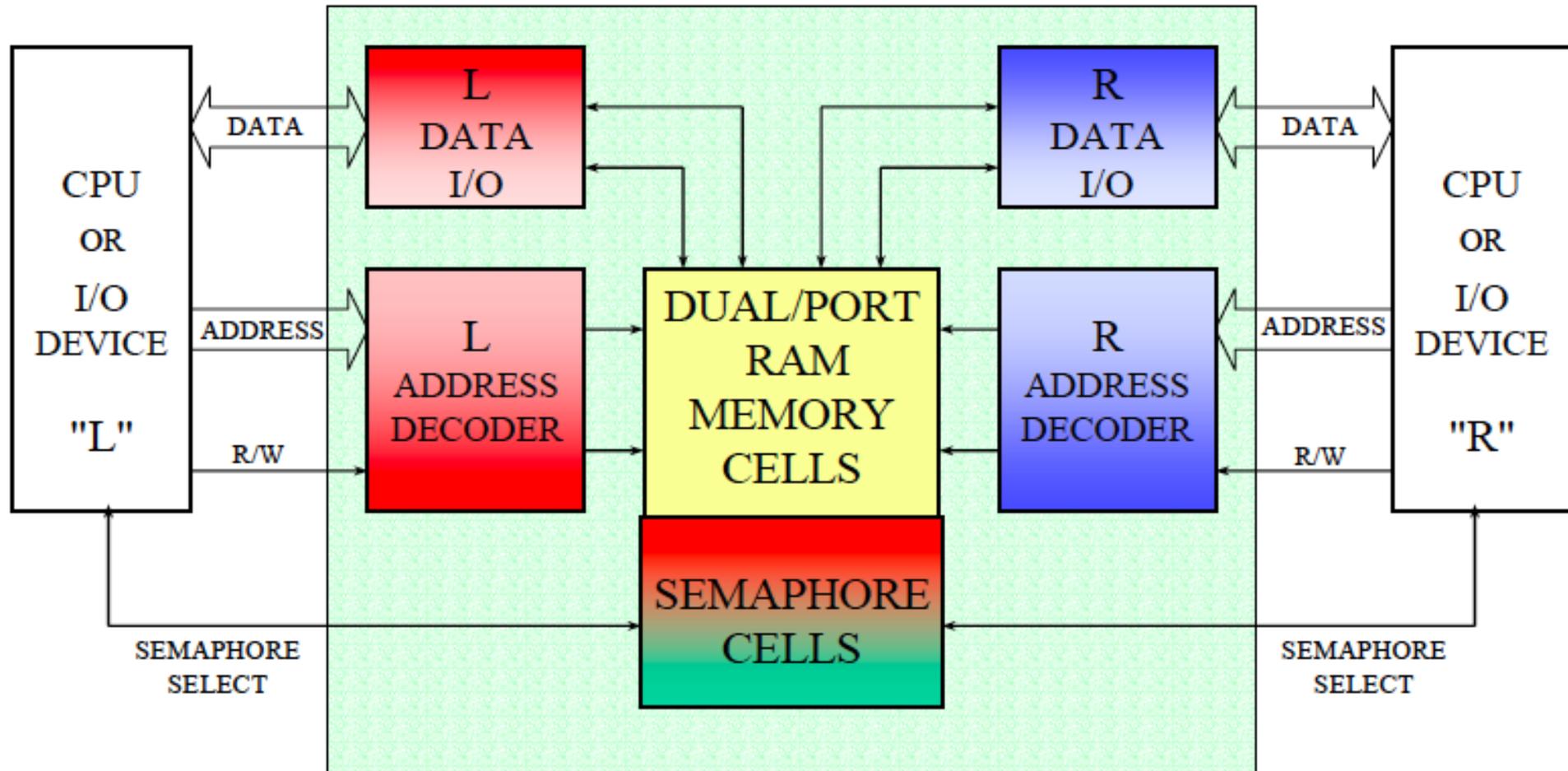
- Solutions:
 - true multi-porting
 - virtual multi-porting (time sharing a port)
 - multiple cache copies
 - banking (interleaving)

Handling Multiple Accesses per Cycle (I)

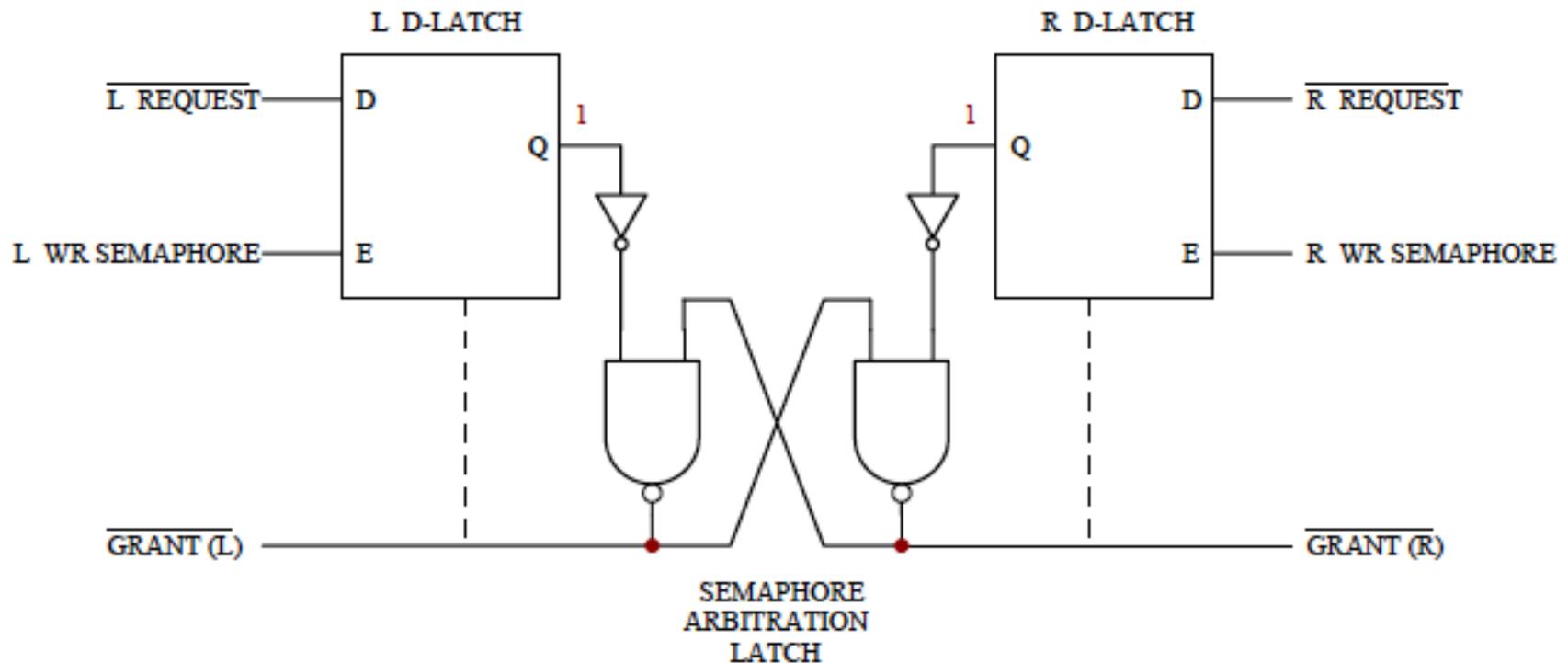
- True multiporting
 - Each memory cell has multiple read or write ports
 - + Truly concurrent accesses (no conflicts regardless of address)
 - Expensive in terms of latency, power, area
 - What about read and write to the same location at the same time?
 - Peripheral logic needs to handle this



Peripheral Logic for True Multiporting



Peripheral Logic for True Multiporting

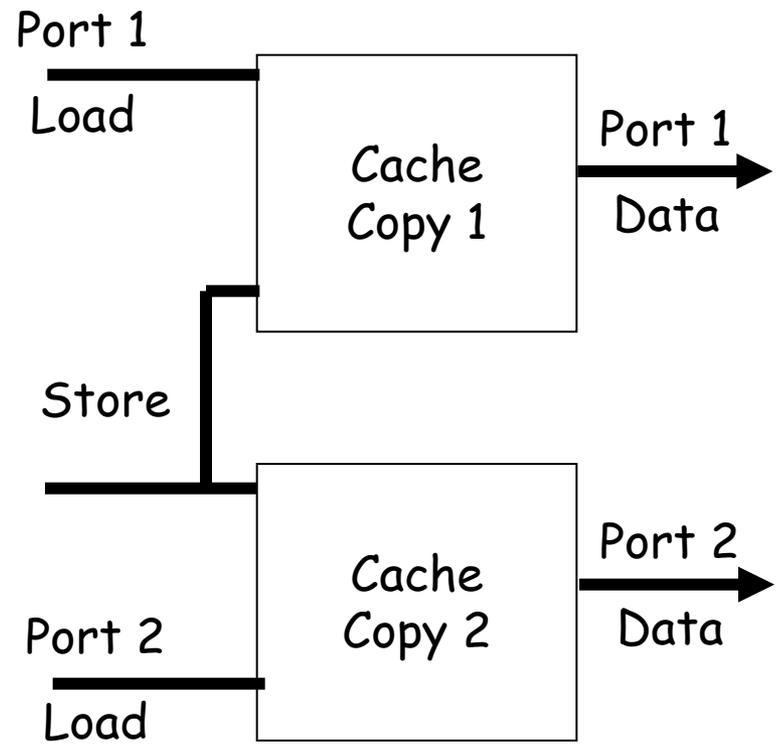


Handling Multiple Accesses per Cycle (I)

- Virtual multiplexing
 - Time-share a single port
 - Each access needs to be (significantly) shorter than clock cycle
 - Used in Alpha 21264
 - Is this scalable?

Handling Multiple Accesses per Cycle (II)

- **Multiple cache copies**
 - ❑ Stores update both caches
 - ❑ Loads proceed in parallel
- Used in Alpha 21164
- Scalability?
 - ❑ Store operations form a bottleneck
 - ❑ Area proportional to “ports”



Handling Multiple Accesses per Cycle (III)

■ Banking (Interleaving)

- Bits in address determines which bank an address maps to
 - Address space partitioned into separate banks
 - Which bits to use for “bank address”?

+ No increase in data store area

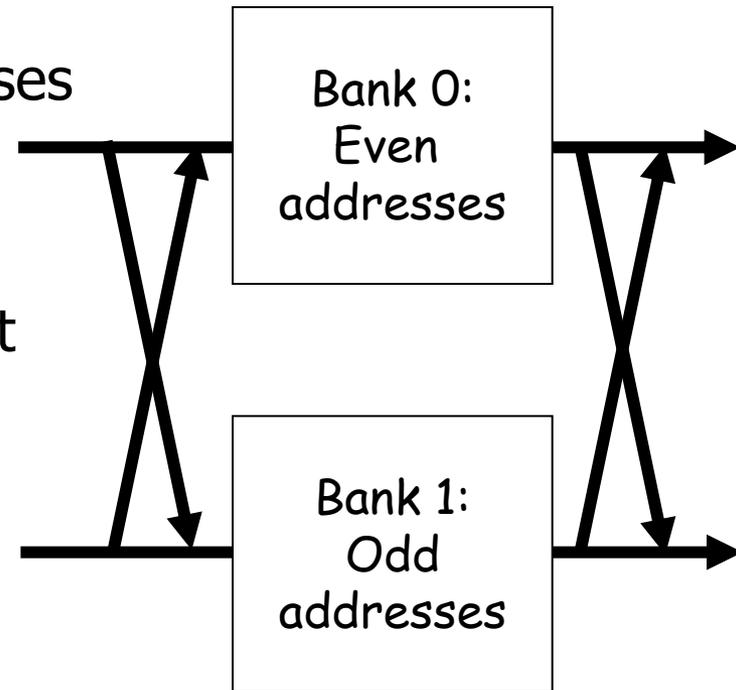
-- Cannot always satisfy multiple accesses to the same bank

Why?

-- Crossbar interconnect in input/output

■ Bank conflicts

- Two accesses are to the same bank
- How can these be reduced?
 - Hardware? Software?

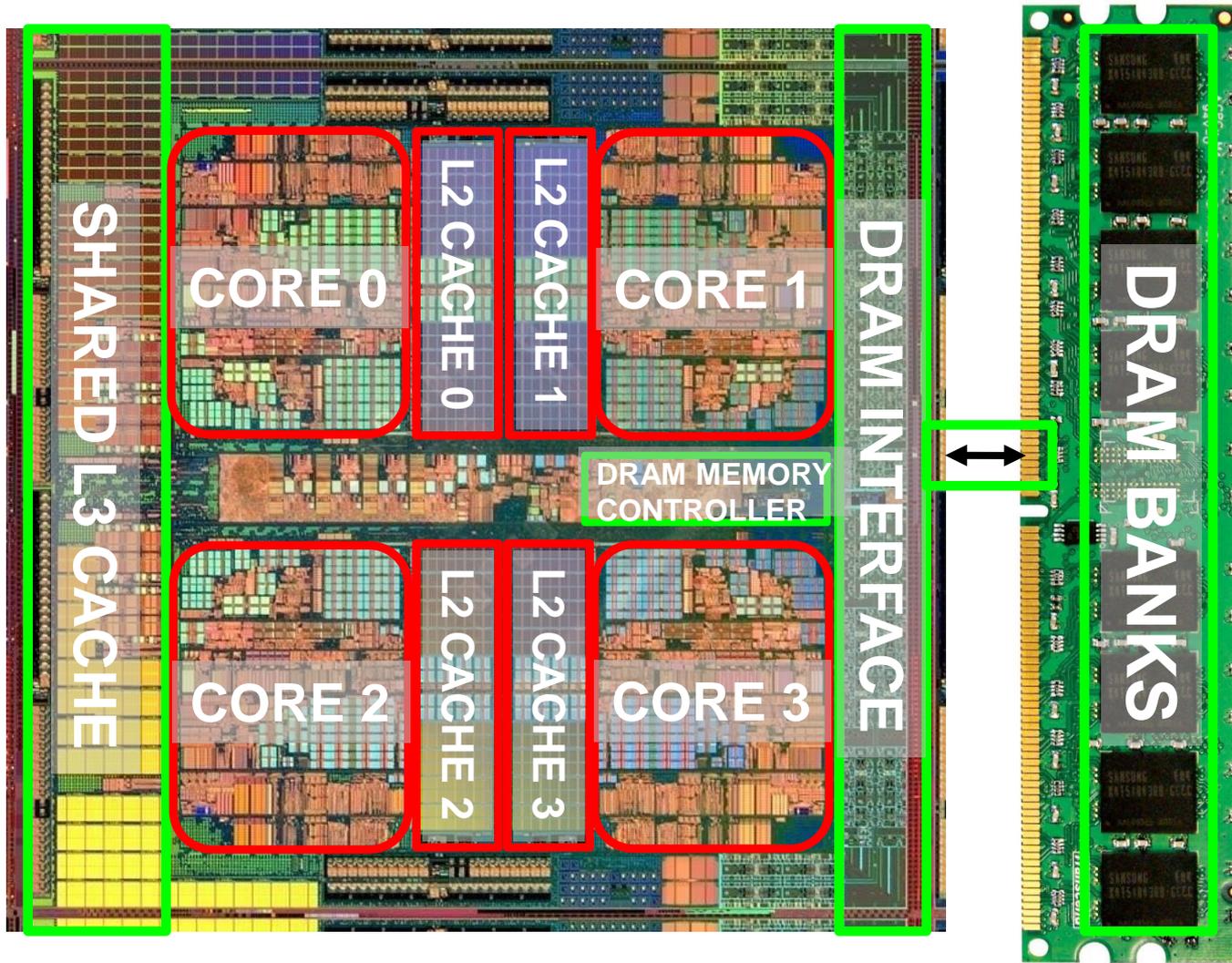


Evaluation of Design Options

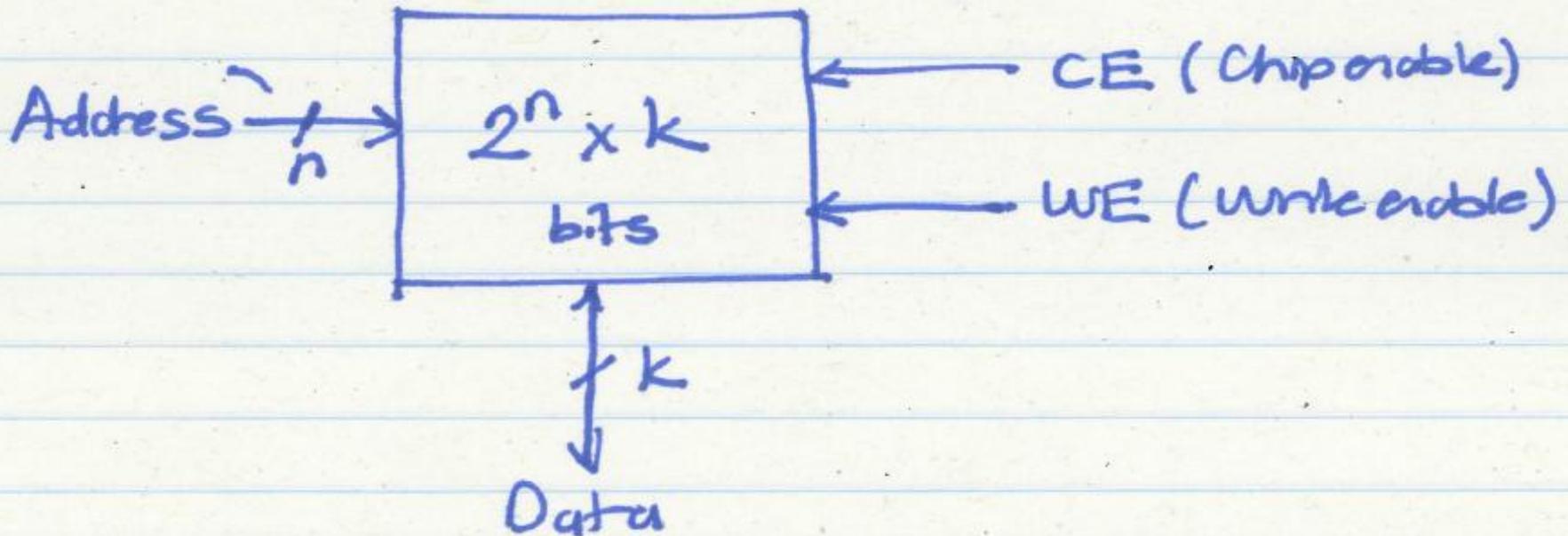
- Which alternative is better?
 - true multi-porting
 - virtual multi-porting (time sharing a port)
 - multiple cache copies
 - banking (interleaving)
 - How do we answer this question?
- Simulation
 - See Juan et al.'s evaluation of above options: “Data caches for superscalar processors,” ICS 1997.
 - What are the shortcomings of their evaluation?
 - Can one do better with sole simulation?

Main Memory

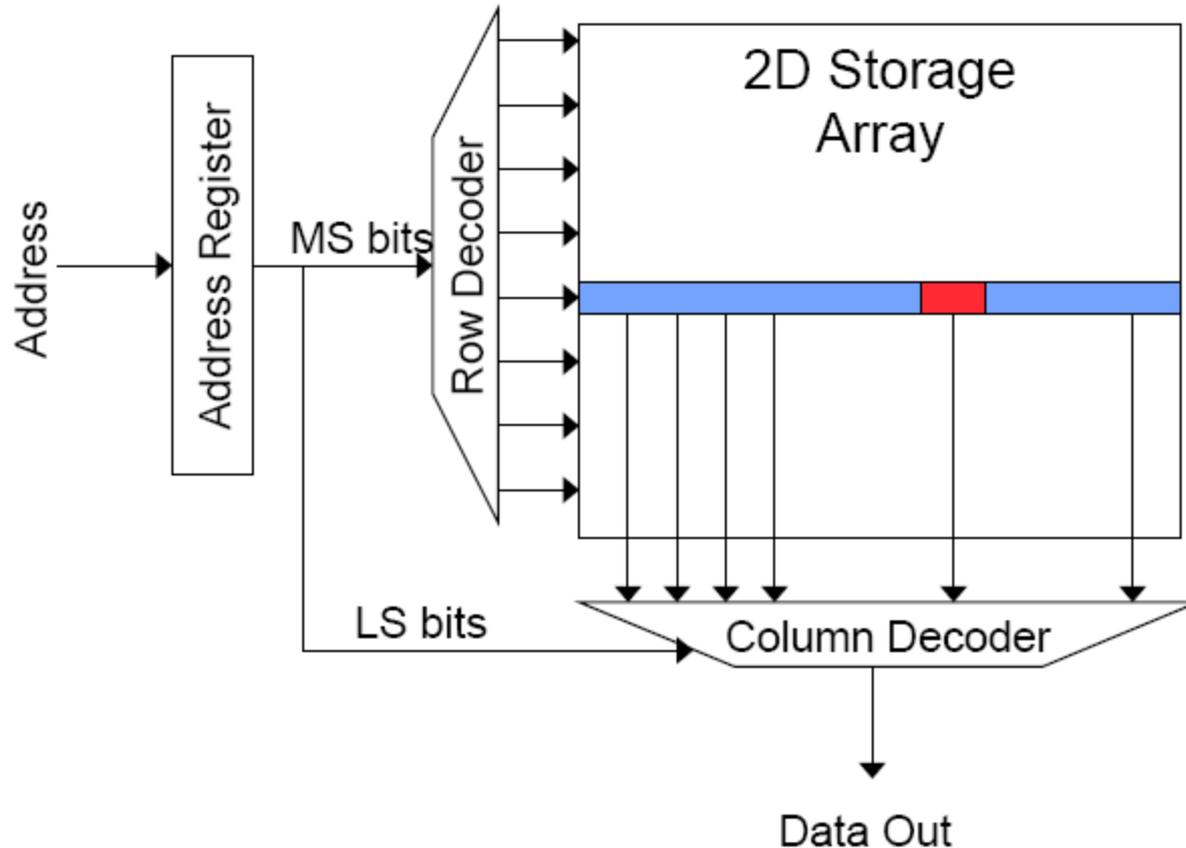
Main Memory in the System



The Memory Chip/System Abstraction



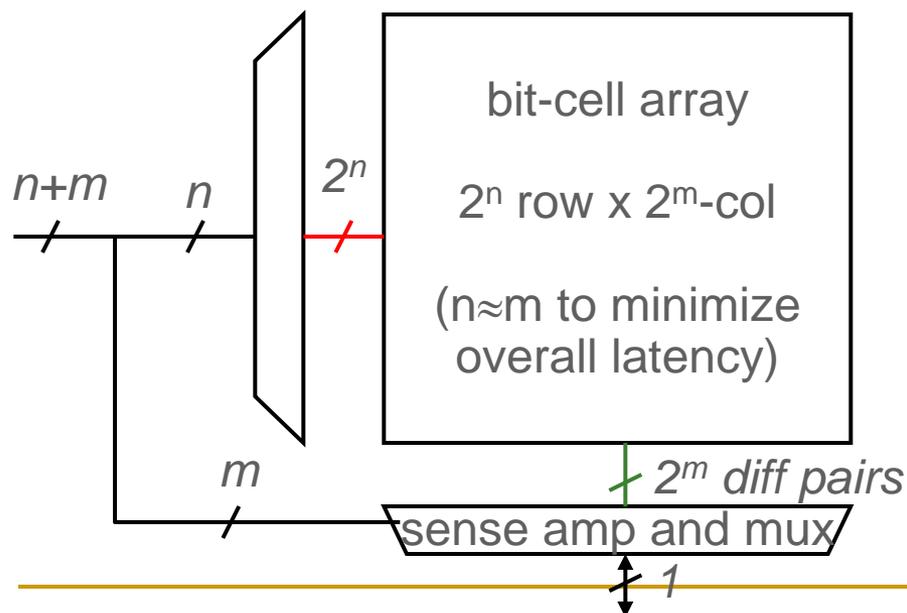
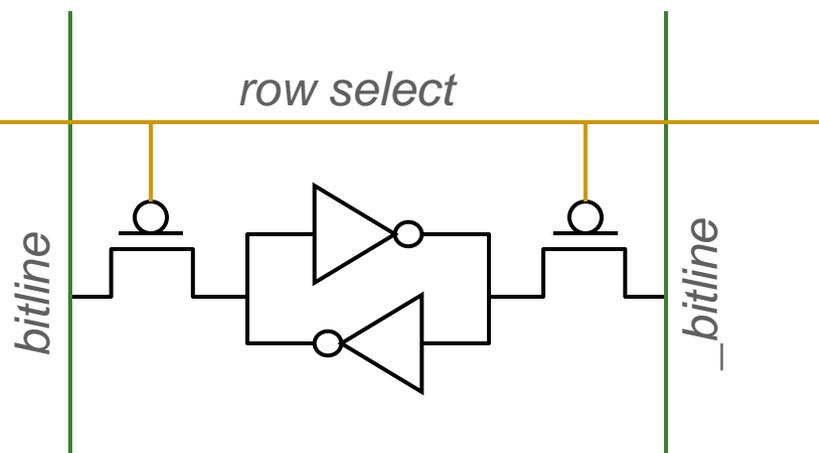
Memory Bank Organization



■ Read access sequence:

1. Decode row address & drive word-lines
2. Selected bits drive bit-lines
 - Entire row read
3. Amplify row data
4. Decode column address & select subset of row
 - Send to output
5. Precharge bit-lines
 - For next access

Review: SRAM (Static Random Access Memory)



Read Sequence

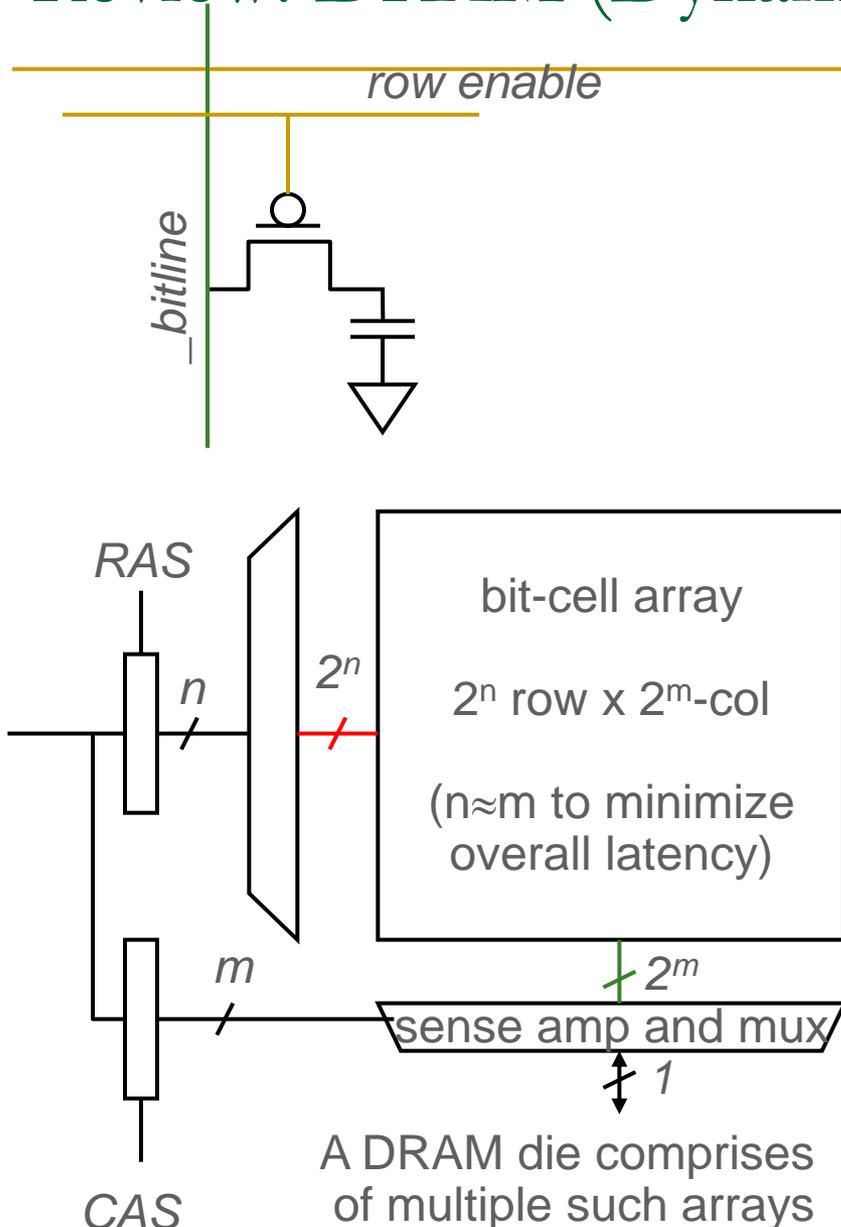
1. address decode
2. drive row select
3. selected bit-cells drive bitlines
(entire row is read together)
4. diff. sensing and col. select
(data is ready)
5. precharge all bitlines
(for next read or write)

Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

Review: DRAM (Dynamic Random Access Memory)



Bits stored as charges on node capacitance (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a “flip-flopping” sense amp amplifies and regenerates the bitline, data bit is mux’ed out

5. precharge all bitlines

Refresh: A DRAM controller must periodically read all rows within the allowed refresh time (10s of ms) such that charge is restored in cells

Review: DRAM vs. SRAM

■ DRAM

- ❑ Slower access (capacitor)
- ❑ Higher density (1T 1C cell)
- ❑ Lower cost
- ❑ Requires refresh (power, performance, circuitry)
- ❑ Manufacturing requires putting capacitor and logic together

■ SRAM

- ❑ Faster access (no capacitor)
- ❑ Lower density (6T cell)
- ❑ Higher cost
- ❑ No need for refresh
- ❑ Manufacturing compatible with logic process (no capacitor)

Some Fundamental Concepts (I)

■ Physical address space

- Maximum size of main memory: total number of uniquely identifiable locations

■ Physical addressability

- Minimum size of data in memory can be addressed
- Byte-addressable, word-addressable, 64-bit-addressable
- Addressability depends on the abstraction level of the implementation

■ Alignment

- Does the hardware support unaligned access transparently to software?

■ Interleaving

Some Fundamental Concepts (II)

■ Interleaving (banking)

- **Problem:** a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
- **Goal:** Reduce the latency of memory array access and enable multiple accesses in parallel
- **Idea:** Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
 - Each bank is smaller than the entire memory storage
 - Accesses to different banks can be overlapped
- **Issue:** How do you map data to different banks? (i.e., how do you interleave data across banks?)

Interleaving

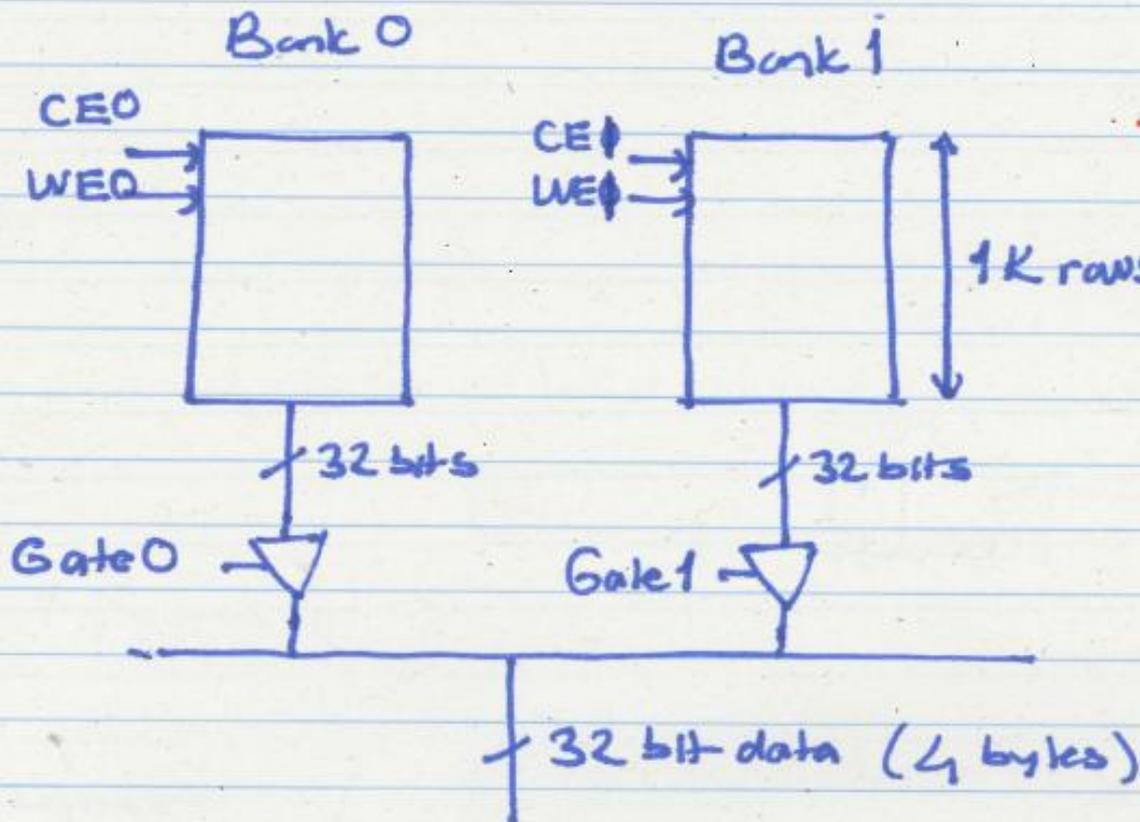
Interleaving (Example)

Assume each bank supplies a word.

Which banks do consecutive words in memory are mapped to?



i.e. how do we interleave the words across the banks?

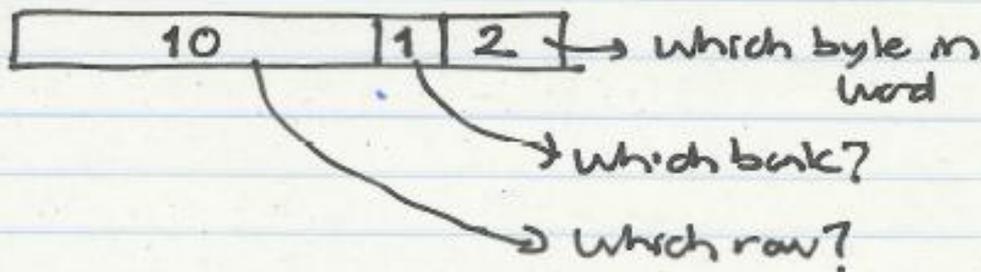


Interleaving Options

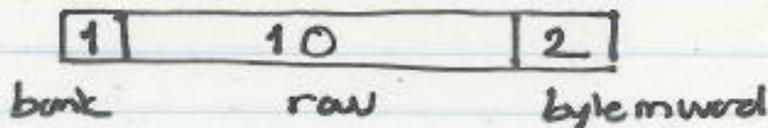
Physical address



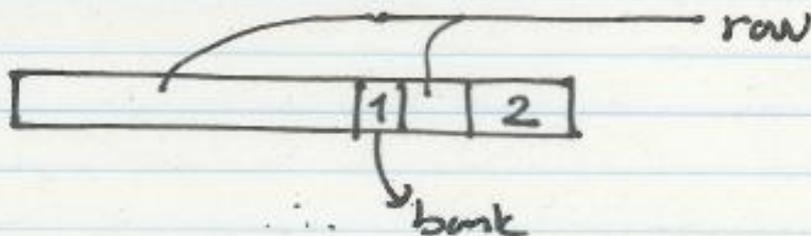
Interleaving scheme 1



Interleaving scheme 2



Interleaving scheme 3

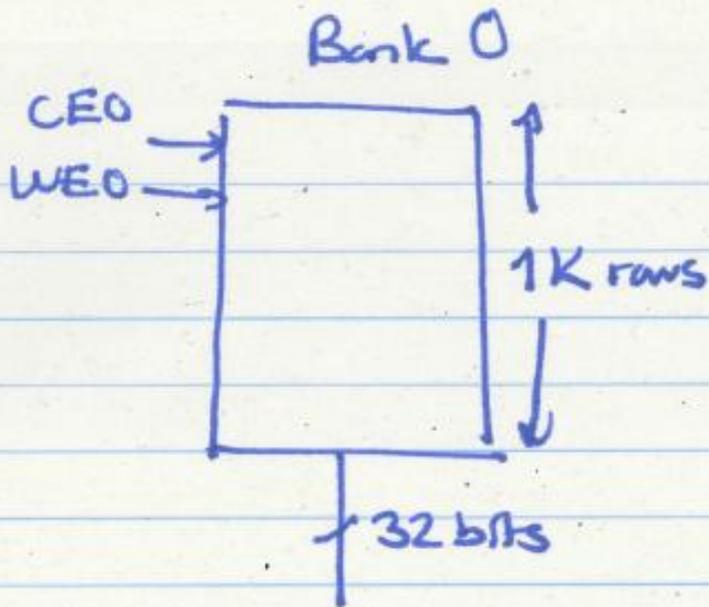


Where (which bank) do consecutive words in memory are mapped to?

Some Questions/Concepts

- Remember CRAY-1 with 16 banks
 - 11 cycle bank latency
 - Consecutive words in memory in consecutive banks (word interleaving)
 - 1 access can be started (and finished) per cycle
- Can banks be operated fully in parallel?
 - Multiple accesses started per cycle?
- What is the cost of this?
 - We have seen it earlier (today)
- Modern superscalar processors have L1 data caches with multiple, fully-independent banks

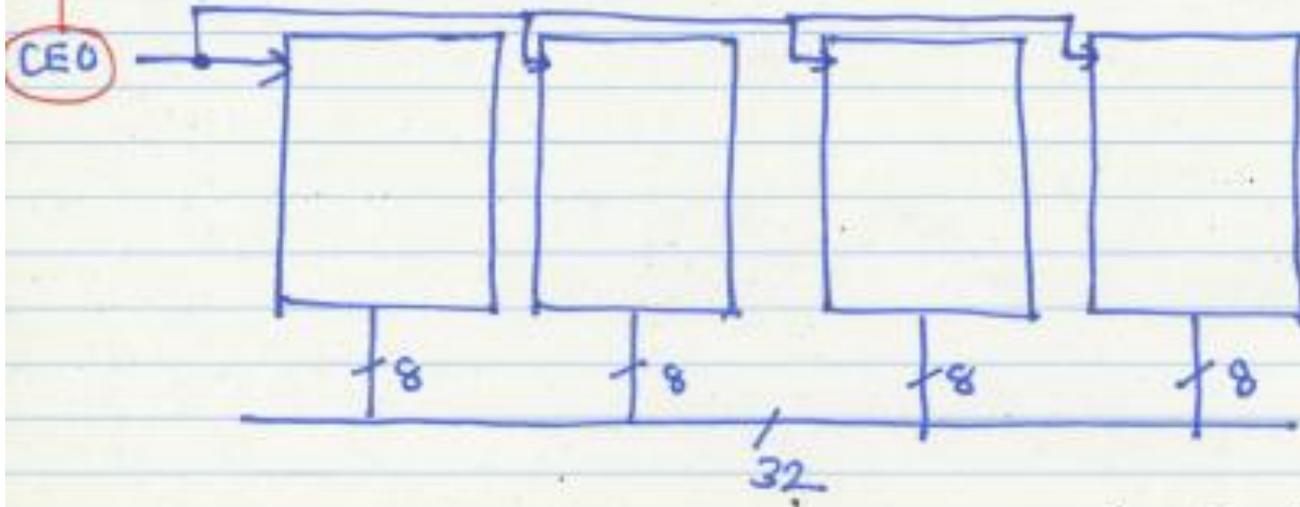
The Bank Abstraction



← Even this is an abstraction
The 32-bits can come from multiple chips, each of which can supply $32/N$ bits.

Rank

call this bank enable (BEO)



This is called a "rank." (only bank 0 shown here)
of the rank

Rank: A set of chips that respond to the same command & same address at the same time with different pieces of the requested data

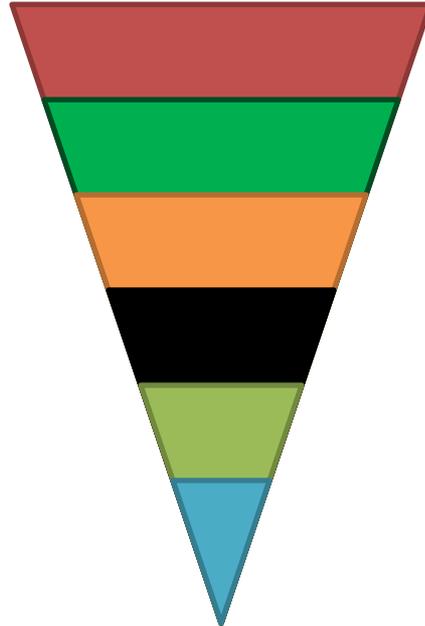
Why? Producing an 8-bit/pm chip cheaper than producing a 32-bit/pm chip

Idea: Produce an 8-bit/pm chip, but control/operate them as a rank so that we can get 32 bits in a single read.

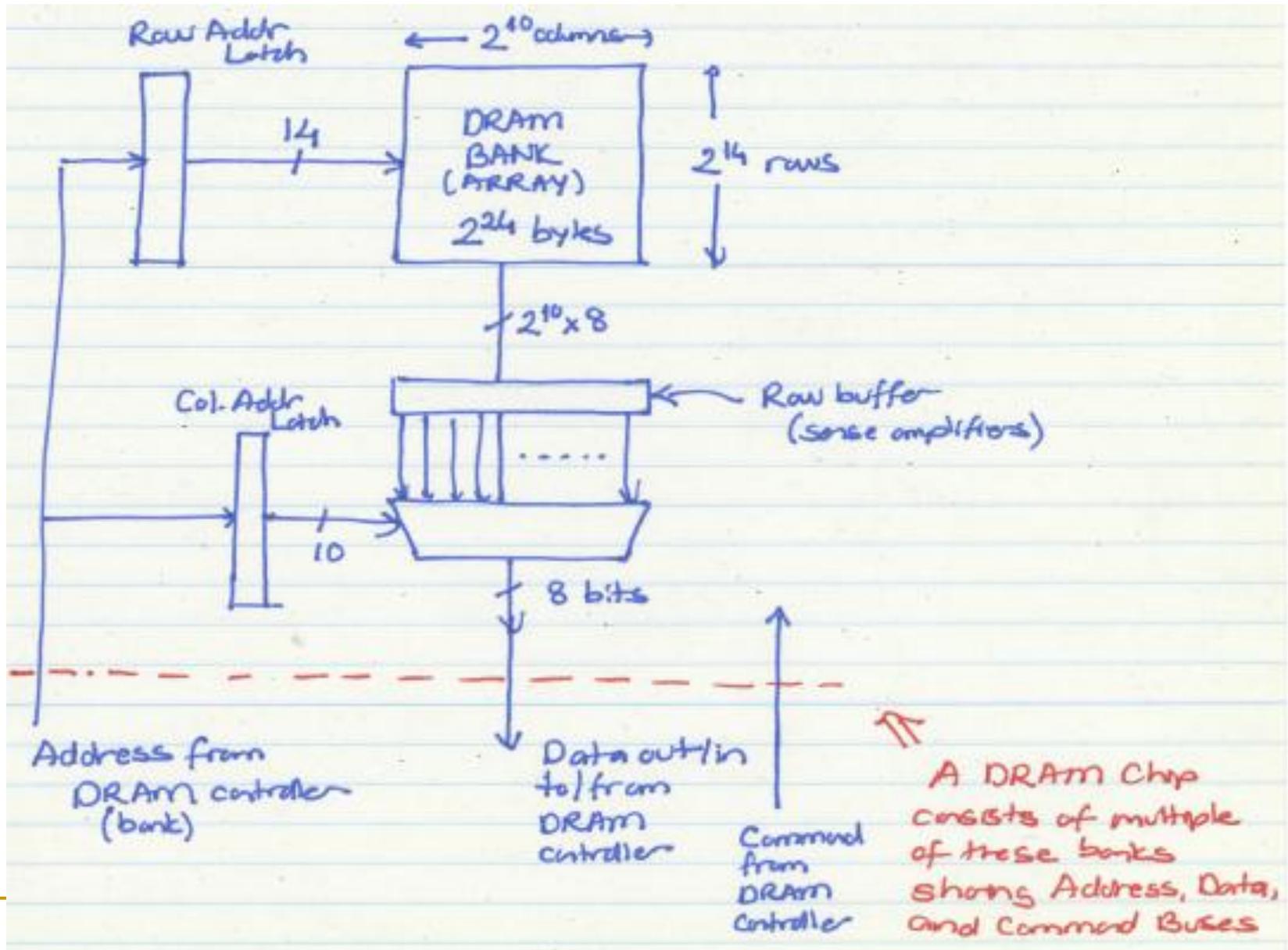
The DRAM Subsystem

DRAM Subsystem Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column



The DRAM Bank Structure



Page Mode DRAM

- A DRAM bank is a 2D array of cells: rows x columns
- A “DRAM row” is also called a “DRAM page”
- “Sense amplifiers” also called “row buffer”

- Each address is a <row,column> pair
- Access to a “closed row”
 - **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
- Access to an “open row”
 - No need for activate command

DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

