SFU

CMPT 450/750: Computer Architecture
Fall 2021

Memory Consistency & DRAM

Alaa Alameldeen & Arrvindh Shriraman

Memory Consistency Models

- **Formal specification of how the memory system will appear to the programmer**
- **Places restrictions on the value that can be returned by a "read" operation in a shared memory program execution**
  - "Read" should return the value of the last "Write" to the same location
  - For uniprocessor, "last" is defined by program order
  - For a multiprocessor, not clear how to define "last write"
- **Example (textbook)**

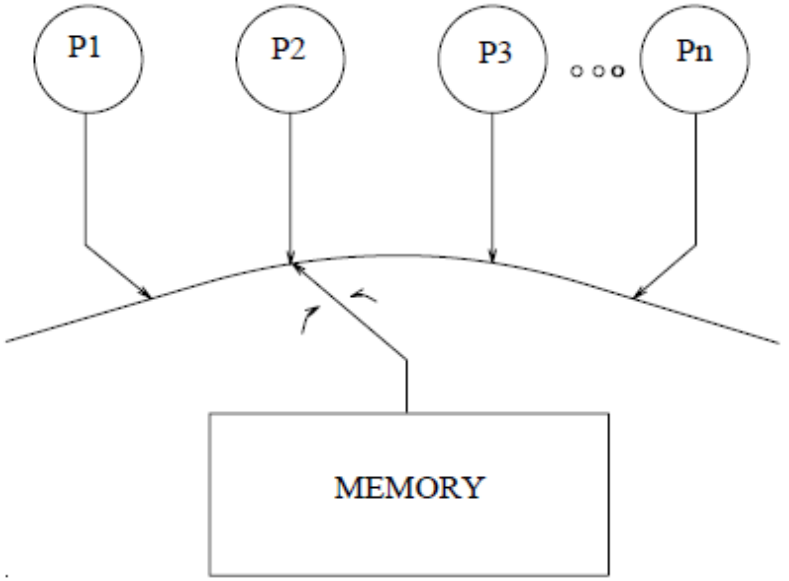
```
P1:     A = 0;              P2:     B = 0;
        .....                       .....
        A = 1;                      B = 1;
L1:     if (B == 0)...      L2:     if (A == 0)...
```

Why Memory Consistency Models are Important

- **Programmability**
  - Some memory consistency models are easier to reason about than others
  - With no clear definition of memory consistency, programmers have to be conservative with shared data
- **Performance**
  - Strict consistency models don't allow many performance optimizations in hardware and system software
  - Conservative programming strategies may inhibit parallelism
- **Portability**
  - Programs written for one architecture may not work on another architecture with a different memory consistency model

Sequential Consistency

- **An extension of uniprocessor "program order"**
- **A multiprocessor is sequentially consistent if**
  - Result of any execution is the same as if the operation of all processors were executed in some sequential order
  - Operation of each processor appear in this sequence in the order specified by its program
- **Two requirements for SC**
  - Program order
  - Atomicity for memory operations
- **Advantage**
  - Simple and intuitive programming model
- **Disadvantages**
  - Prevents many hardware optimizations (e.g., write buffers)
  - Prevents many compiler optimizations (e.g., code motion)



Adve and Gharachorloo, Figure 3: Programmer's view of SC

Sequential Consistency: Examples

Adve and Gharachorloo, Figure 4

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a)

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A ==1)	
	B = 1	
		if (B==1)
		register1 = A

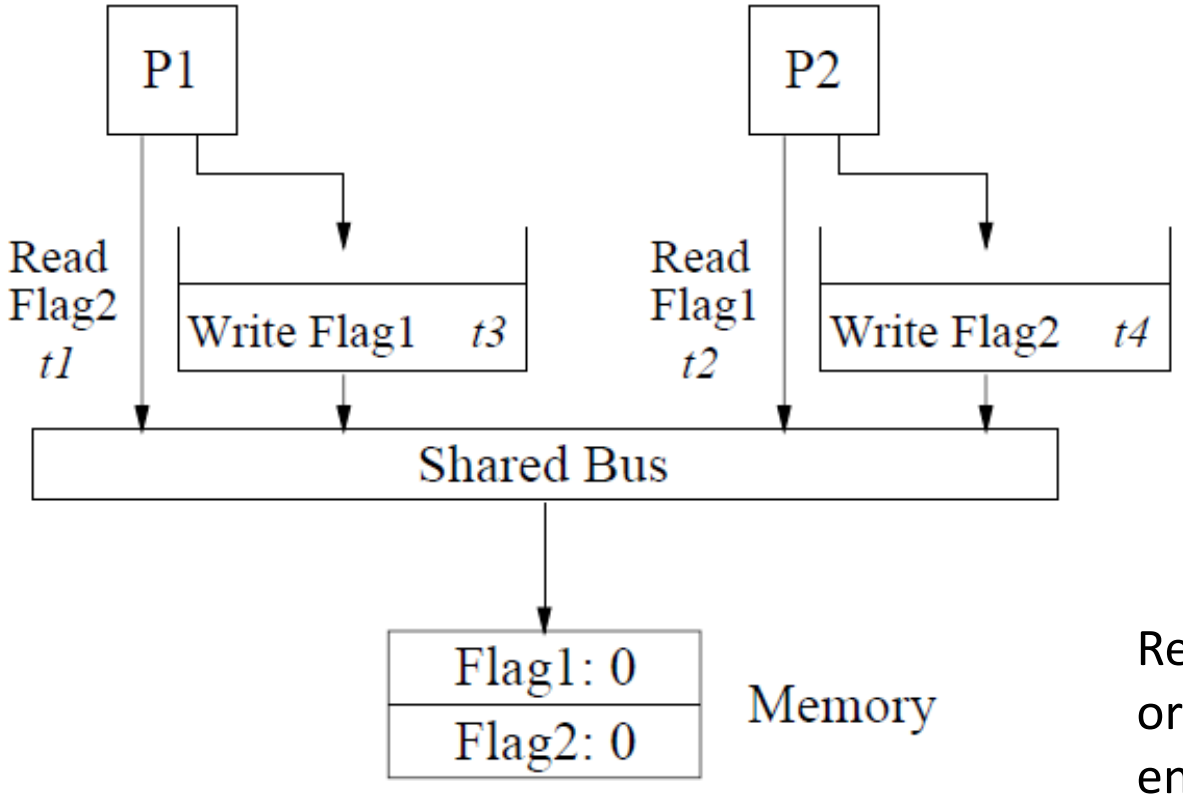
(b)

a) Dekker's Algorithm for mutual exclusion: Program order needed so both processors don't enter critical section simultaneously

b) Atomicity of memory operations needed so writes appear in the same order for all processors (otherwise P3 may return old value of A)

Implementing Sequential Consistency (1)

- SC restricts some common optimizations, even in the absence of caches

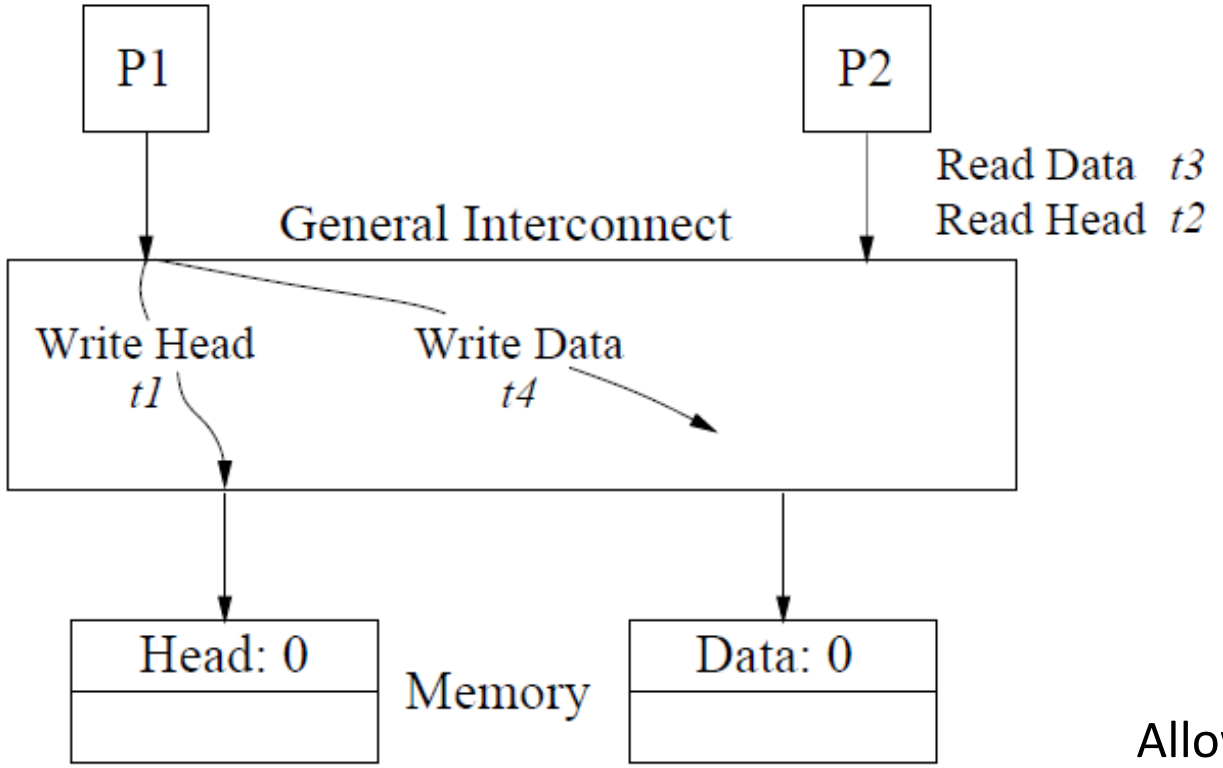


P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Retiring writes to a write buffer violates the program order requirement of SC, leading to both processors entering critical section simultaneously

Adve and Gharachorloo, Figure 5a: Write Buffers

Implementing Sequential Consistency (2)

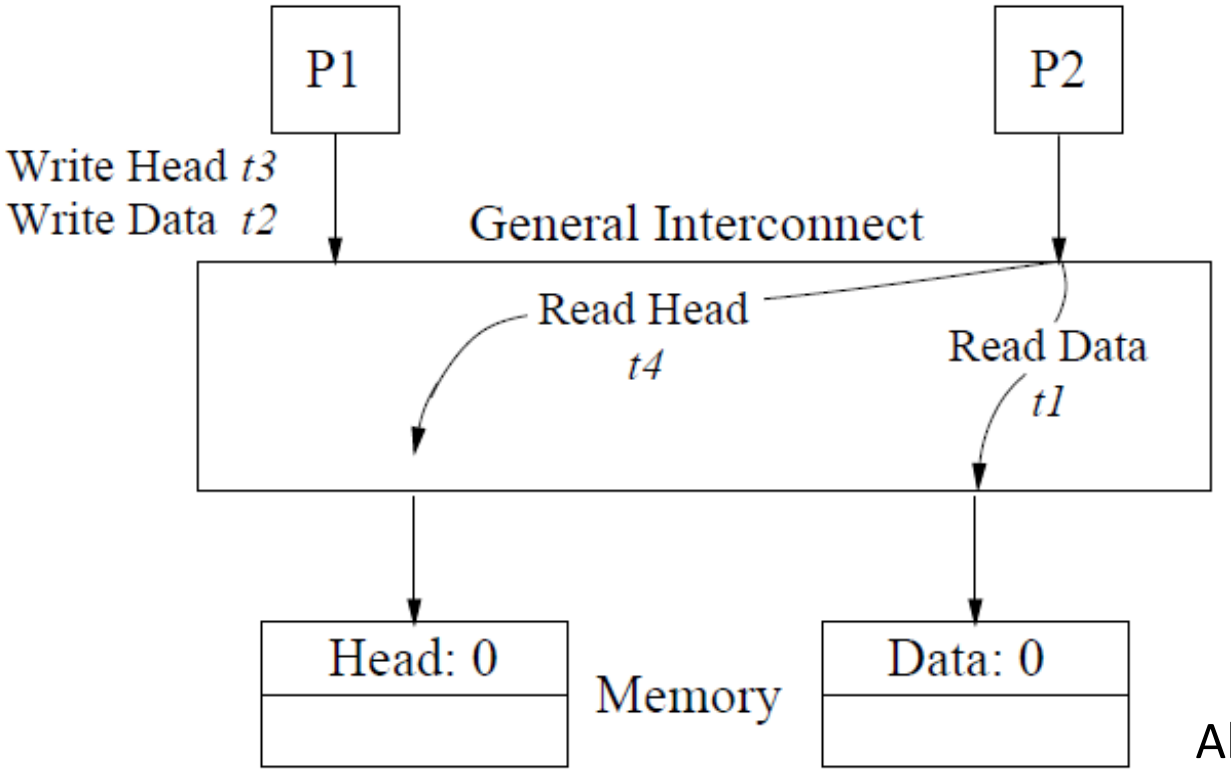


P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Allowing P1's writes to local memories to finish faster violates SC, leading to incorrect read order at P2

Adve and Gharachorloo, Figure 5b: Overlapping Writes

Implementing Sequential Consistency (3)



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Allowing P2's read from local memory to finish faster violates SC, leading to P2 reading new value of Head but old value of data

Adve and Gharachorloo, Figure 5c: Non-blocking Reads

Implementing Sequential Consistency (4)

- **Architectures with caches**
  - Cache coherence represents the mechanism that propagates a newly written value to the cached copies of the modified location
  - Memory consistency model is the policy that places an early and late bound on when a new value can be propagated to any given processor
  - How do we detect the completion of a write operation?

Implementing SC: Write Atomicity (1)

- Writes to the same location need to be serialized

Initially A = B = C = 0

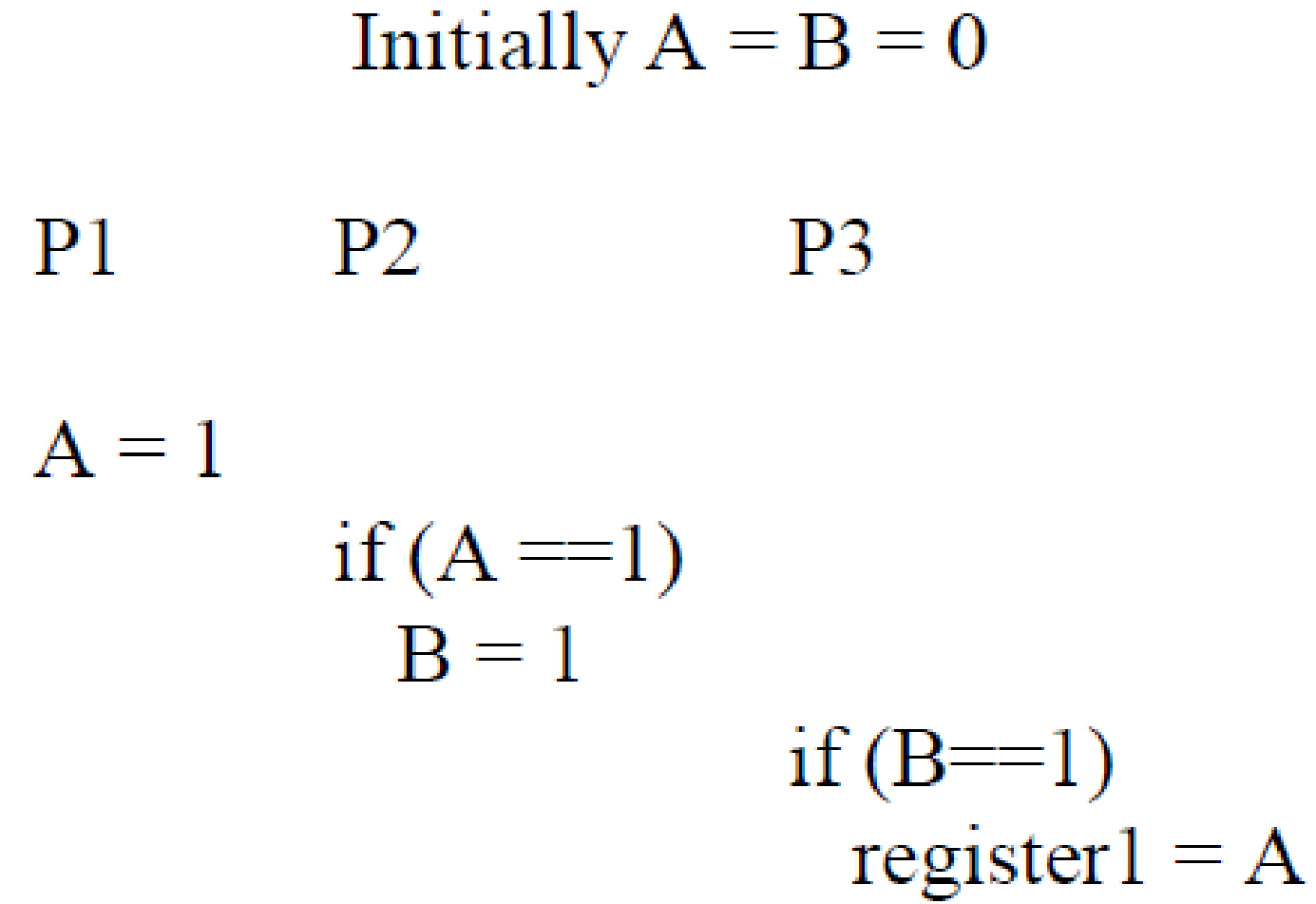
P1	P2	P3	P4
A = 1	A = 2	while (B != 1) {;}	while (B != 1) {;}
B = 1	C = 1	while (C != 1) {;}	while (C != 1) {;}
		register1 = A	register2 = A

Adve and Gharachorloo, Figure 6

- Assuming a write update protocol and a general interconnection network, writes to A by P1 and P2 can reach P3 and P4 in different orders, violating the write atomicity condition of SC
- Can be avoided if we guarantee writes to the same location are serialized (e.g,, using write-invalidate cache coherence protocol)

Implementing SC: Write Atomicity (2)

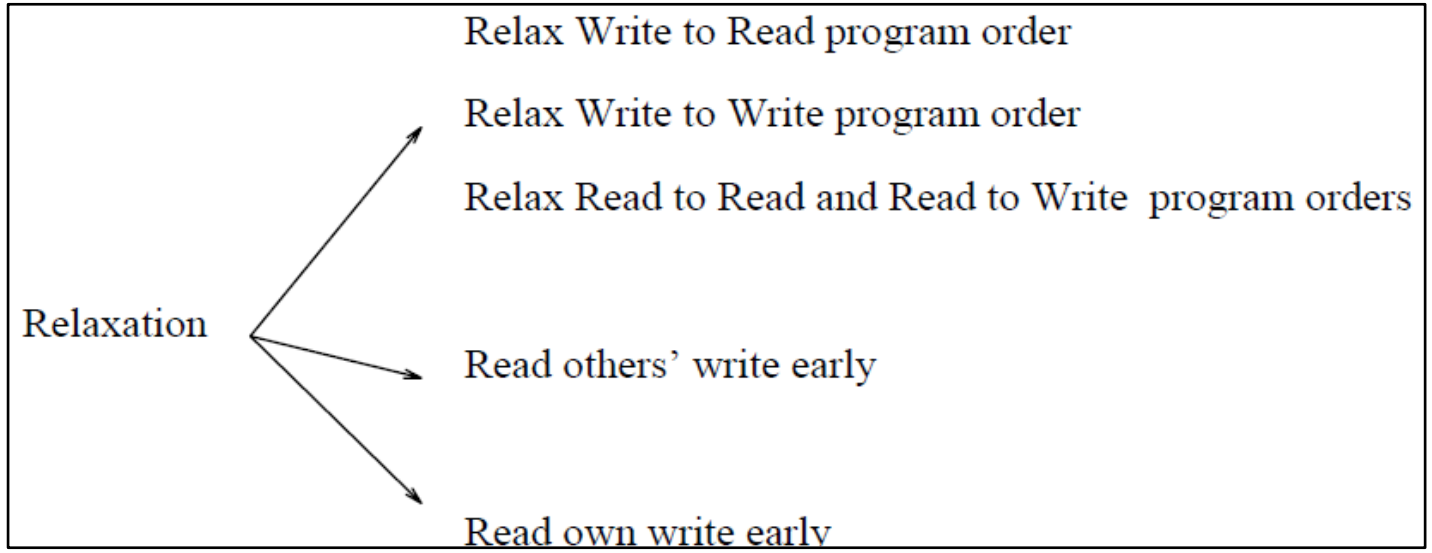
- With a general interconnection network, it is possible to violate SC if P2 gets P1's update to A before P3, and P3 gets P2's update to B before P1's update to A
- To avoid this, need to prevent read from returning new value until all acknowledgements for write are received
  - Again, this is simpler with write-invalidate coherence protocols



Adve and Gharachorloo, Figure 4b

Relaxed Memory Models (1)

Adve and Gharachorloo, Figure 7



- **Can relax either:**
  - Program order requirement
  - Write atomicity requirement
- **Relaxing program order requirement**
  - Write to a following read
  - Two writes
  - Read to a following read or write
- **Relaxing write atomicity requirement**
  - Can a read return the value of another processor's write before the write is visible to all processors?
- **Relaxing both requirements**
  - Can a processor read the value of its own previous write before it is made visible to all other processors?

Relaxed Memory Models (2)

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Adve and Gharachorloo, Figure 8

Examples of Systems with Relaxed Models

Relaxation	Example Commercial Systems Providing the Relaxation
W → R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W → W Order	AlphaServer 8200/8400, Cray T3D
R → RW Order	AlphaServer 8200/8400, Cray T3D
Read Others' Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Adve and Gharachorloo, Figure 9

Relaxing Write to Read Program Order

- **IBM 370**
  - Does not enable a processor to read its own write early (SC enables this optimization)
- **Total Store Order (TSO), implemented in SPARC V8 and many x86 processors**
  - Relaxes Write to read order and enables a processor to read its own write early
  - Enforces order between writes (W -> W)
  - Enforces order between a read and subsequent reads/writes (R -> RW)
- **Processor Consistency (PC) (e.g, DASH)**
  - Similar to TSO except that it enables reading other writes early
- **All techniques allow a read to be reordered wrt previous writes from the same processor**
  - Enable write buffers
- **Techniques differ on when to allow a read to return the value of a write**

Relaxing Write to Read Program Order: Examples

Initially A = Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
A = 1	A = 2
register1 = A	register3 = A
register2 = Flag2	register4 = Flag1

Result: register1 = 1, register3 = 2,
register2 = register4 = 0

(a)

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

Result: B = 1, register1 = 0

(b)

Adve and Gharachorloo, Figure 10

a) Is possible with TSO and PC but not with IBM 370
b) Is possible with PC but not with TSO or IBM 370

Relaxing Write to Read & Write to Write Program Orders

- **Example: Partial Store Order (PSO), implemented in SPARC V8**
- **Writes to different locations from the same processor can be pipelined or overlapped**
  - ➢Writes allowed to reach memory or other caches out of program order
- **A processor can read the value of its own write early**
- **A processor is prohibited from reading another processor's write until it is visible to all other processors**

Relaxing All Program Orders

- **Relax program order between all operations to a different location**
- **A read or write may be reordered wrt a following read or write to a different location**
- **Allows non-blocking reads (lockup-free caches, speculative execution)**
- **Allows almost all compiler optimizations**
- **Examples**
  - Weak Ordering (WO)
  - Release Consistency (RCpc, PCsc)
  - DEC Alpha
  - PowerPC
  - Relaxed Memory Order (RMO) in SPARC V9

Weak Ordering (WO)

- **Classifies memory operations into two categories**
  - Data operations
  - Synchronization operations
- **To enforce program order between two operations, programmer needs to specify synchronization operation**
- **Intuition: reordering data operations in between synchronization operations would not affect correctness**
- **Writes appear atomic to programmer**

Release Consistency (RC)

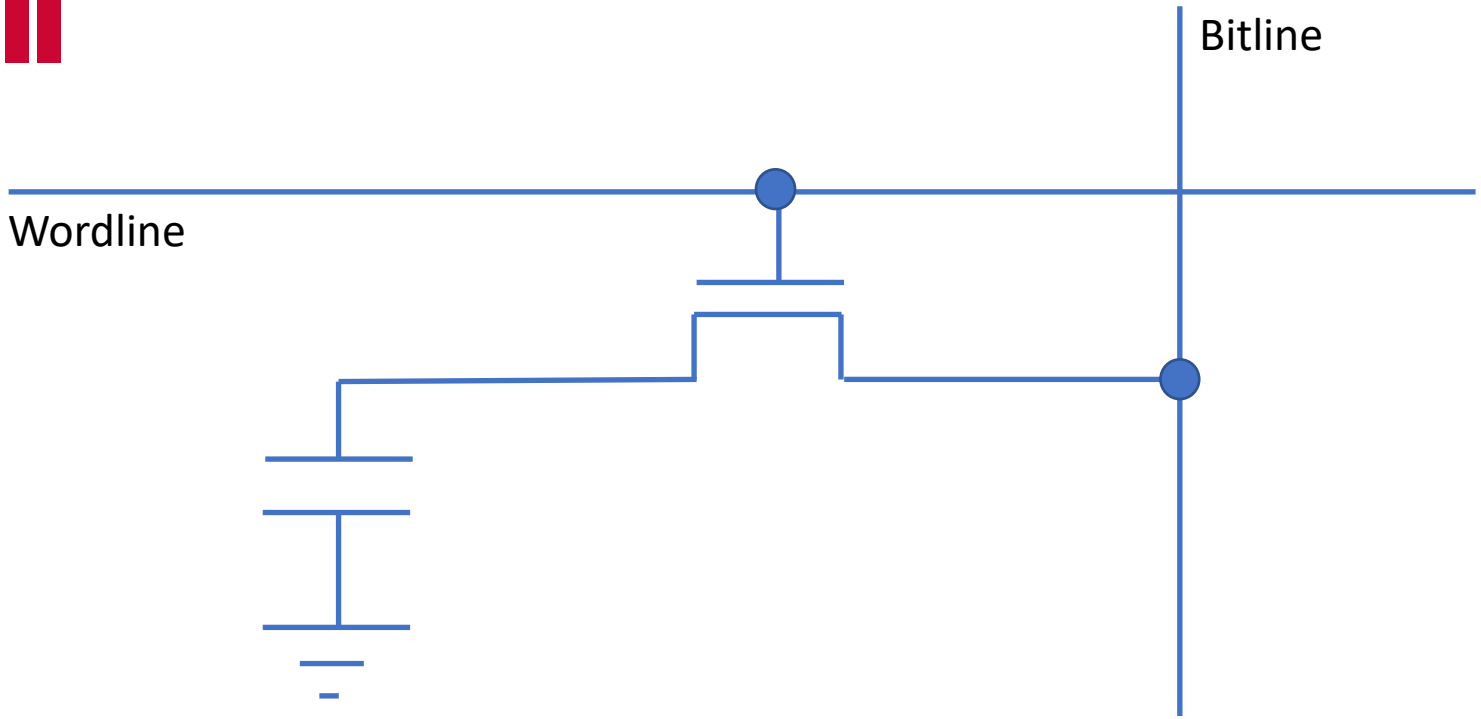
- **Classifies memory operations into:**
  - Ordinary operations
  - Special operations
    - Sync: Synchronization operations
    - Nsync: asynchronous data operations, not used for synchronization
- **Sync operations are either**
  - Acquire: read operation to gain access to a set of shared locations (e.g., lock, spin for a flag to be set)
  - Release: write operation to grant permission for accessing set of shared location (e.g., unlock, set flag)
- **Different RC Models provide different program orders among special operations**
  - RCsc: acquire → all, all → release, special → special
  - RCpc: RCsc: acquire → all, all → release, special → special except for special write followed by a special read

DRAM Basics

DRAM

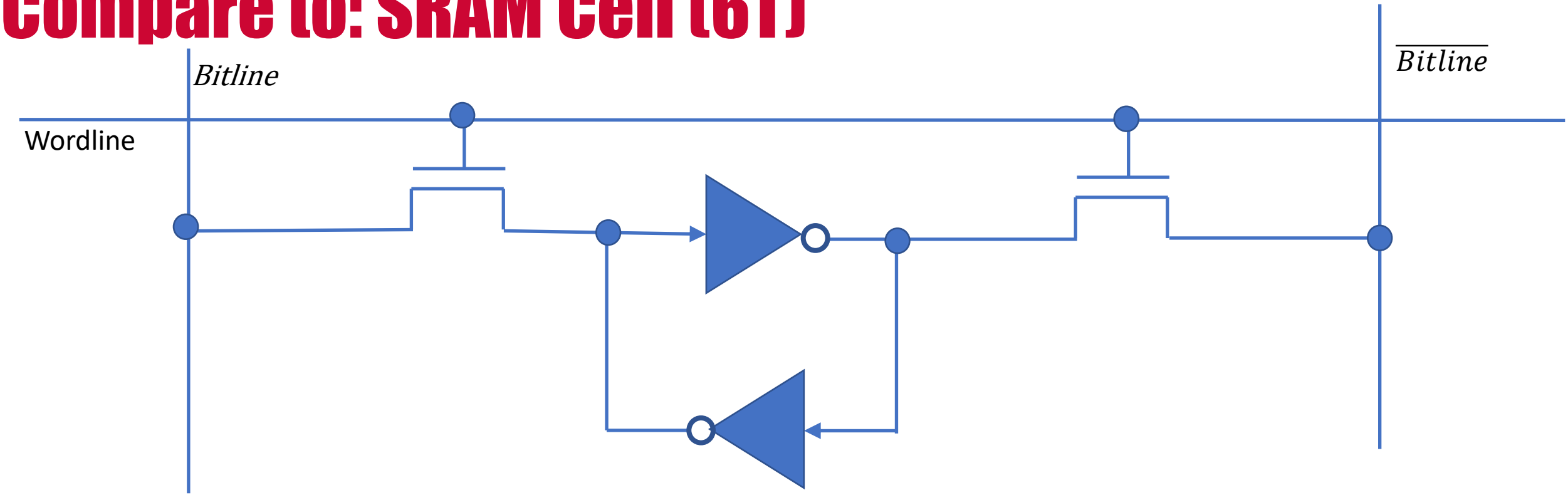
- **Stands for “Dynamic Random Access Memory”**
- **Volatile memory, used as main memory in most computer systems**
- **DRAM cells are single-transistor, single-capacitor cells (1T1C)**
  - Much higher density than 6T SRAM cells
- **Data stored by charging or discharging capacitor**
- **Reads are destructive: Data needs to be written back to cell after read**
- **As capacitors lose charge over time, DRAM cells need to be “refreshed” to restore charge**
  - “Dynamic” RAM requires periodic refreshing while “Static” RAM doesn’t
- **Power consumption is mostly from leakage and refresh power**

DRAM Cell



- **Storing “1”: Set Wordline (WL) to high to turn on transistor, set Bitline (BL) to high to charge capacitor**
- **Storing “0”: Set Wordline to high to turn on transistor, set Bitline to low to discharge capacitor**
- **Read cell: Set Wordline to high to turn on transistor, value is read on the Bitline (sensed using a sense amplifier to amplify change)**
  - Reading disturbs charge stored on capacitor so old value needs to be restored

Compare to: SRAM Cell (6T)



- **Storing "1": Set Wordline to high to turn on access transistors, set Bitline to high and $\overline{Bitline}$ to low to store "1" at lower inverter output, "0" at upper inverter output**
- **Storing "0": Set Wordline to high to turn on access transistors, set Bitline to low and $\overline{Bitline}$ to high to store "0" at lower inverter output, "1" at upper inverter output**
- **Read cell: Set Wordline to high to turn on access transistors, value from lower transistor is read on Bitline and upper inverter is read on $\overline{Bitline}$ (Read is not destructive)**

DRAM Terminology

- **A system has multiple sockets each containing a chip multiprocessor (i.e., a multicore processor) that interfaces to DRAM using one or more memory channels each controlled by a memory controller**
  - Single socket systems are common in client systems, multi-socket systems are common for servers
- **Each memory channel can interface with one or more DIMMs "Dual Inline Memory Module"**
  - A DIMM is a circuit board with chips on both sides
- **Each DIMM is divided into ranks (typically 1 or 2)**
- **Each Rank has multiple DRAM chips which are further divided into banks. Each bank is addressable independently of other banks**
- **Each bank is divided into one or more memory arrays (also called subarrays or tiles). Each array contains rows and columns**
  - A "xN" DIMM has N memory arrays per bank and can access data from N columns simultaneously
  - For example, a "x4" DIMM has 4 memory arrays per bank, and can read 4 bits from each bank simultaneously

DRAM Hierarchy Example

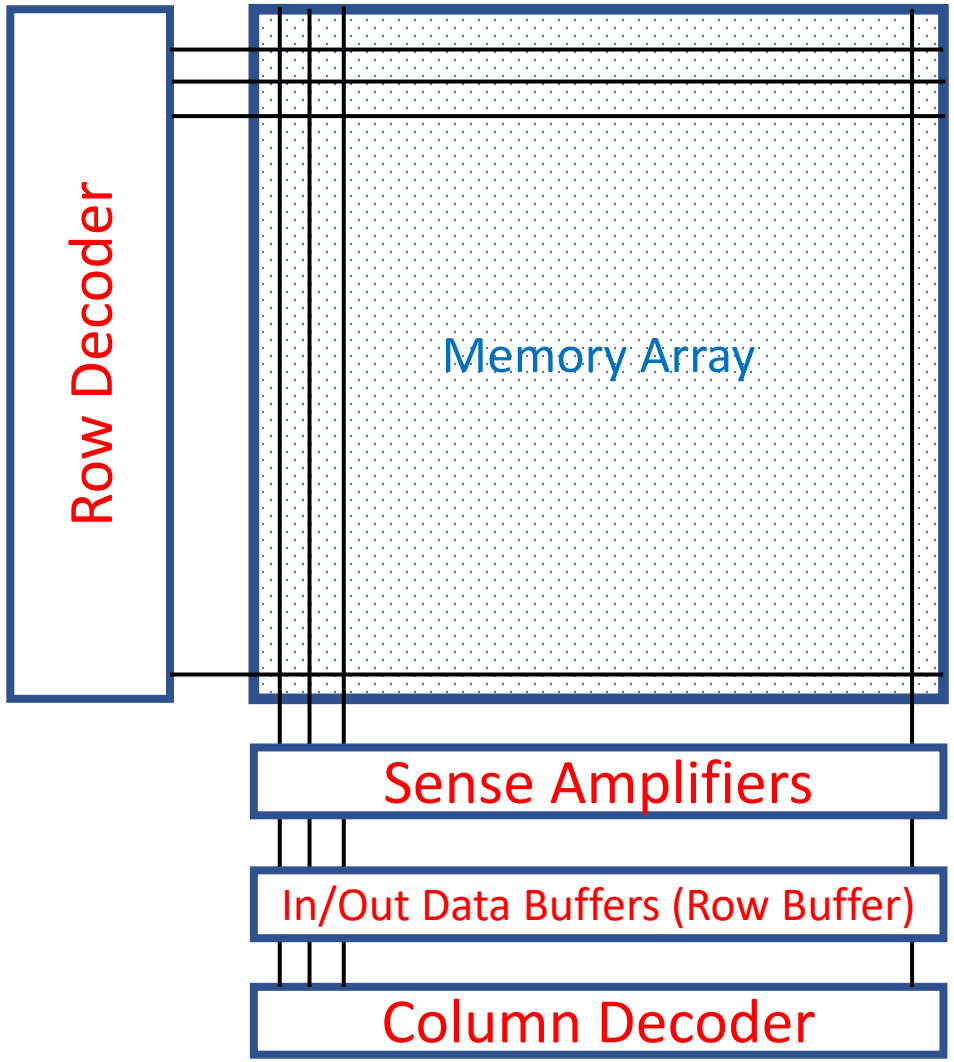
- **Hierarchy: Socket → Channel → DIMM → Rank → Chip → Bank → Array → Rows and Columns**
- **Example:**
  - A DRAM array contains 8192 rows and 8192 columns (64Mb)
  - A x16 DIMM has 16 arrays per bank (1Gb)
  - If each DIMM has 2 ranks, each rank has 8 chips, and each chip has 4 banks, then the total memory capacity per DIMM is 2x8x4x1Gb = 64Gb (8GB)
  - A single socket may have 4 memory channels each interfacing with 2 DIMMs, so total memory capacity per socket = 4x2x8GB = 64GB
  - A dual-socket system has a maximum DRAM capacity of 2x64GB = 128 GB

Memory Controllers

- **A memory controller controls a memory channel**
  - Sends bank, row & column address to memory on the **address bus**
  - Sends control bits on the **control bus**. E.g., Row Address Strobe (RAS), Column Address Strobe (CAS), output enable, clock and clock enable etc.
  - Sends/receives data on the **data bus**
  - Selects which ranks need to respond by enabling “Chip Select” bits on the **chip-select bus**
  - Sends/receives data to/from processor
- **Memory controllers decode a data address into chip select bits, bank address bits and row/column addresses depending on DRAM configuration**
- **Memory controllers may also handle special functions (e.g., error detection and correction, prefetching, initiating parallel memory accesses)**

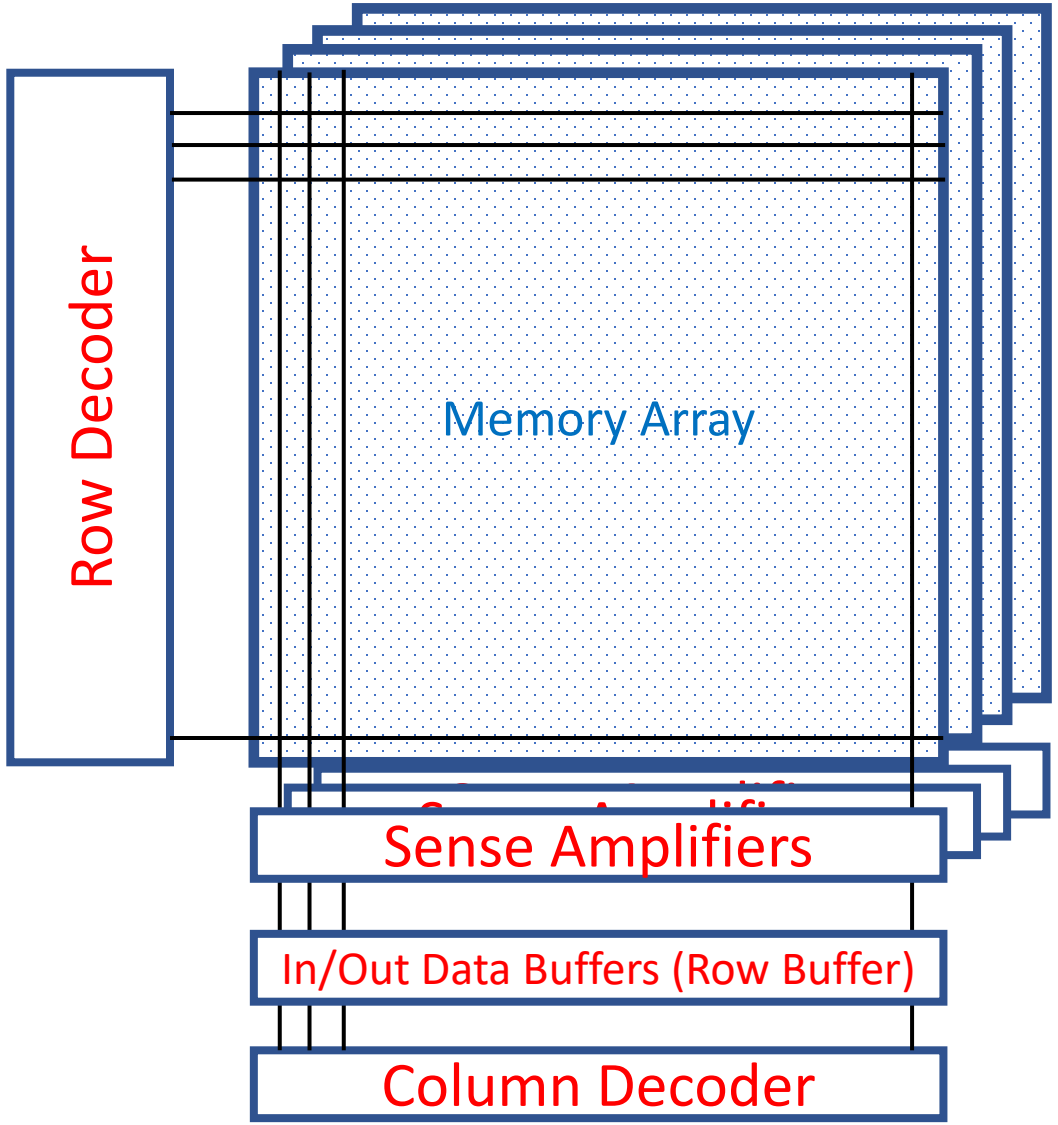
Memory Array

- A memory cell is at the intersection of wordline (horizontal) and bitline (vertical)
- Row decoder decodes row address bits to enable a single row wordline
- Sense amplifiers amplify signals on bitlines
- Column decoder decodes column address bits to select a few bits from the data buffer



Memory Bank

- Contains multiple arrays
- Figure shows a x4 bank which contains 4 memory arrays, each with its own set of sense amplifiers
- For memory reads, data from all arrays are read into a row buffer, then column decoder selects which bits to send out. Row buffer data needs to be written back after reads to restore original bit values
- For memory writes, the whole row is read first to row buffer, selected bits (based on column address) are modified, then whole row is written back



Steps for a Memory Read Operation

1. CPU request misses all cache levels, is sent to memory controller (MC)
2. Request is queued at MC until all prior and higher priority requests are handled
3. MC decodes address into chip select, bank, row and column address bits and sent over to DRAM
4. All bitlines in a bank are ***precharged*** (i.e., set to a level in the middle between logic 0 and 1). Row Precharge Time is referred to as ***tRP***.
5. Appropriate row is ***activated***: Chip select and bank address bits enable bank, RAS signals row address bits are ready, row decoder enables wordline for selected row. This switches on transistors so stored value in capacitors can alter charge of precharged bitlines. Row needs to be active for at least ***tRAS*** to ensure data is read and restored before precharging another row.
6. Data from all bitlines within selected row are sent to sense amplifiers which amplify the signal read from memory cells and store data into row buffer. Column address can be sent to row buffer after time ***tRCD*** (row-address to column-address delay)
7. MC ***reads*** column: Chip select and bank address bits enable bank, CAS signals column address bits are ready, column decoder selects appropriate column, all bits from that column across different arrays are connected to output drivers which drive data bus. ***CL*** is CAS latency: time between sending a column address and receiving data

Row Buffer

- **Each bank has a row buffer where the active row is cached following a read**
  - DRAM row also called a “page” (different from the page concept in virtual memory)
- **Subsequent reads from the same row get data from row buffer (saving RAS and cell access time). This is called a row buffer hit**
- **Since reads are destructive, row buffer data needs to be written back to row before accessing a different row**
- **Row buffer size depends on number of columns and number of arrays per bank**
  - Example: A x8 chip with 8192 rows x 8192 columns has a row buffer size of 8x8192 bits = 8KB
- **Open page vs. Closed page policy**
  - **Open page:** Keep data of active row in row buffer. Works well for high spatial locality (multiple row buffer hits before a row buffer miss)
  - **Closed page:** Write back row buffer data to row and precharge bitelines to save tRP time when accessing a different row. Works well for low spatial locality

Refresh Operations

- **DRAM cells lose charge over time (leakage) so they need to be recharged before bits flip**
- **If a row is read or written via normal memory controller requests, then it is automatically refreshed**
  - However, no guarantee that a specific row will be read/written before bits flip
- **To avoid errors due to charge loss, rows are periodically refreshed**
  - All rows have to be refreshed within a refresh cycle
  - Time between refreshes (tREFI) is based on time to flip weakest cells
- **Refreshes usually initiated by the memory controller**
- **Refresh overhead:**
  - Latency: During refresh operations to a bank, no reads or writes can be performed to that bank. tRFC is delay between a REFRESH command and next valid command to same bank.
  - Power and Energy: Refresh power is a significant fraction of DRAM power/energy consumption
- **Self-refresh mode is performed by DRAM in low power mode when CPU and memory controller are turned off**

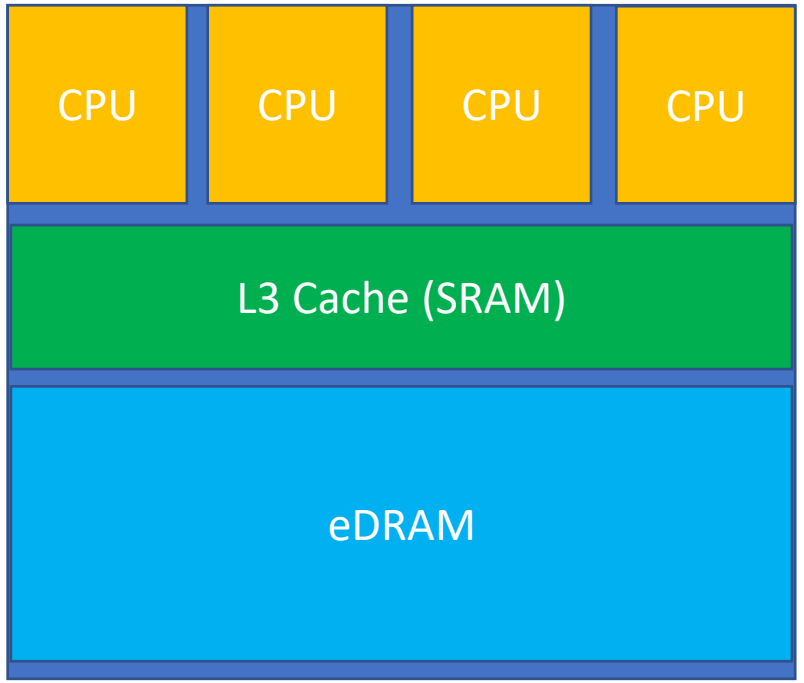
Other Memory Technologies: Embedded DRAM

- **eDRAM is a DRAM integrated on the same die or multi-chip module (MCM) as the CPU**
- **Uses a logic process instead of a DRAM process**
- **Typically used as a large L4 cache**
- **Compared to DRAM:**
  - Pros: Faster, higher bandwidth
  - Cons: More expensive, has lower capacity and requires more frequent refreshes
- **Compared to SRAM:**
  - Pros: Denser (larger capacity) and lower cost/bit
  - Cost: Slower and requires additional cost to manufacture

eDRAM:

- **DRAM on a logic process**

Faster, higher bandwidth than DRAM
Denser than SRAM



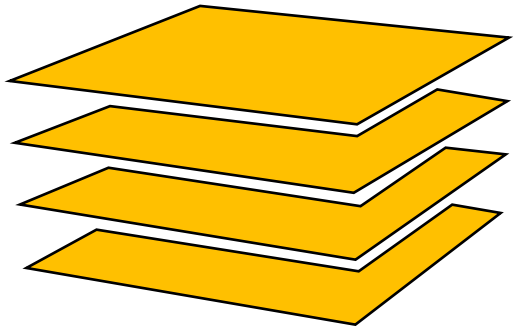
Other Memory Technologies: Stacked DRAM

- Multi-layer Stack of DRAM chips
- Could be above/below CPU/GPU die or stacked on a separate die
- Example: High-Bandwidth Memory (HBM)
- Pros: higher bandwidth, potentially faster than DRAM
  - Single HBM3 stack can have bandwidth higher than 800GB/sec
- Cons: More expensive (higher cost/bit), smaller capacity
- Could be used with DRAM or other technologies as part of a multi-level memory system

Stacked DRAM:

- **High-Bandwidth Memory (HBM)**
- **Hybrid Memory Cube (HMC)**

Faster, higher bandwidth than DRAM



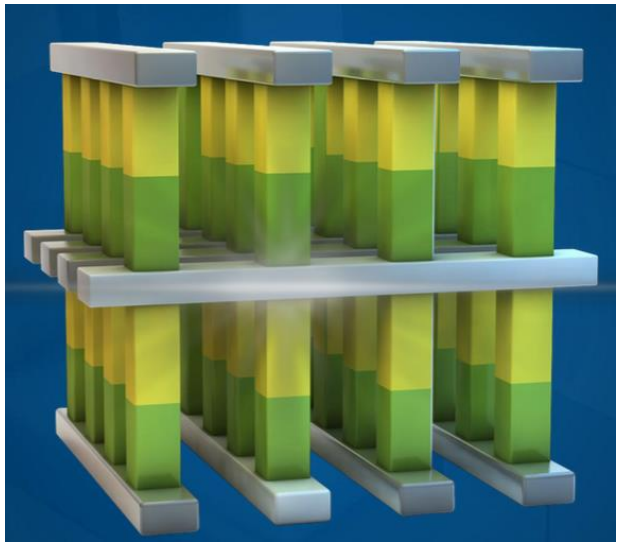
Other Memory Technologies: Non-Volatile Memory

- Typically denser than DRAM
- Non-volatile technology: Stored values persist past power shutdown
- Example: 3D Xpoint (3DXP), Phase Change Memory (PCM)
- Pros: higher capacity vs. DRAM, non-volatile
  - ➢ Can be used as a persistent memory (PMEM)
- Cons: Slower than DRAM, limited bandwidth, limited write endurance
- Could also be used with DRAM or other technologies as part of a multi-level memory system

Non-Volatile Memory (NVM):

- **3D Xpoint**
- **Phase-Change Memory (PCM)**

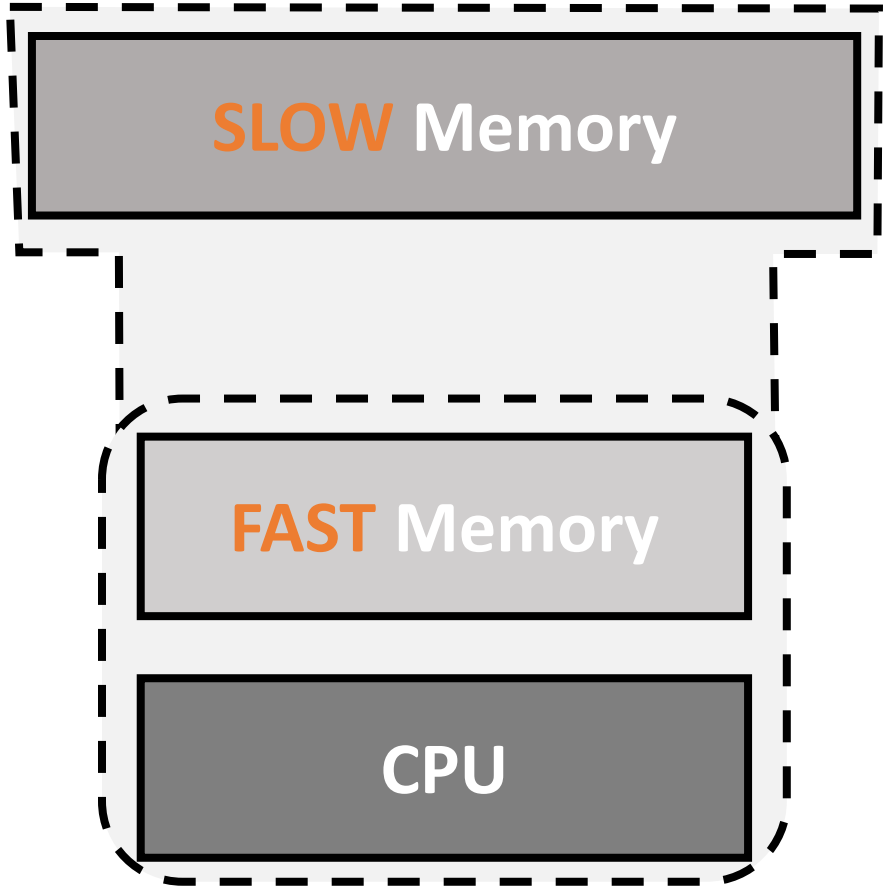
Large capacity, non-volatile but slower



3D Xpoint
(Intel, Micron)

Heterogeneous Memory Systems

- **Memory outside CPU/GPU die can have multiple levels of heterogeneous memory technologies**
- **Different types of memory co-exist:**
  - Fast (higher BW and/or lower latency)
  - Slow (lower BW and/or higher latency)
- **Systems perform better with higher hit rates in fast memory**
- **Fast memory could be organized as:**
  - A cache for slow memory; OR
  - Part of a flat memory address space
- **When does cache make sense vs. flat address space?**



Reading Assignments

- ARCH Chapter 5.6, 5.7 (Read)
- S. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” Technical Report, 1995 (Read)
- Jacob, Ng and Wang, “Memory Systems: Cache, DRAM, Disk” Chapter 7 (Read), Chapter 8 (Skim). Access from SFU Library.