SFU

CMPT 450/750: Computer Architecture
Fall 2024

Memory Consistency & DRAM

Alaa Alameldeen & Arrvindh Shriraman

SFU

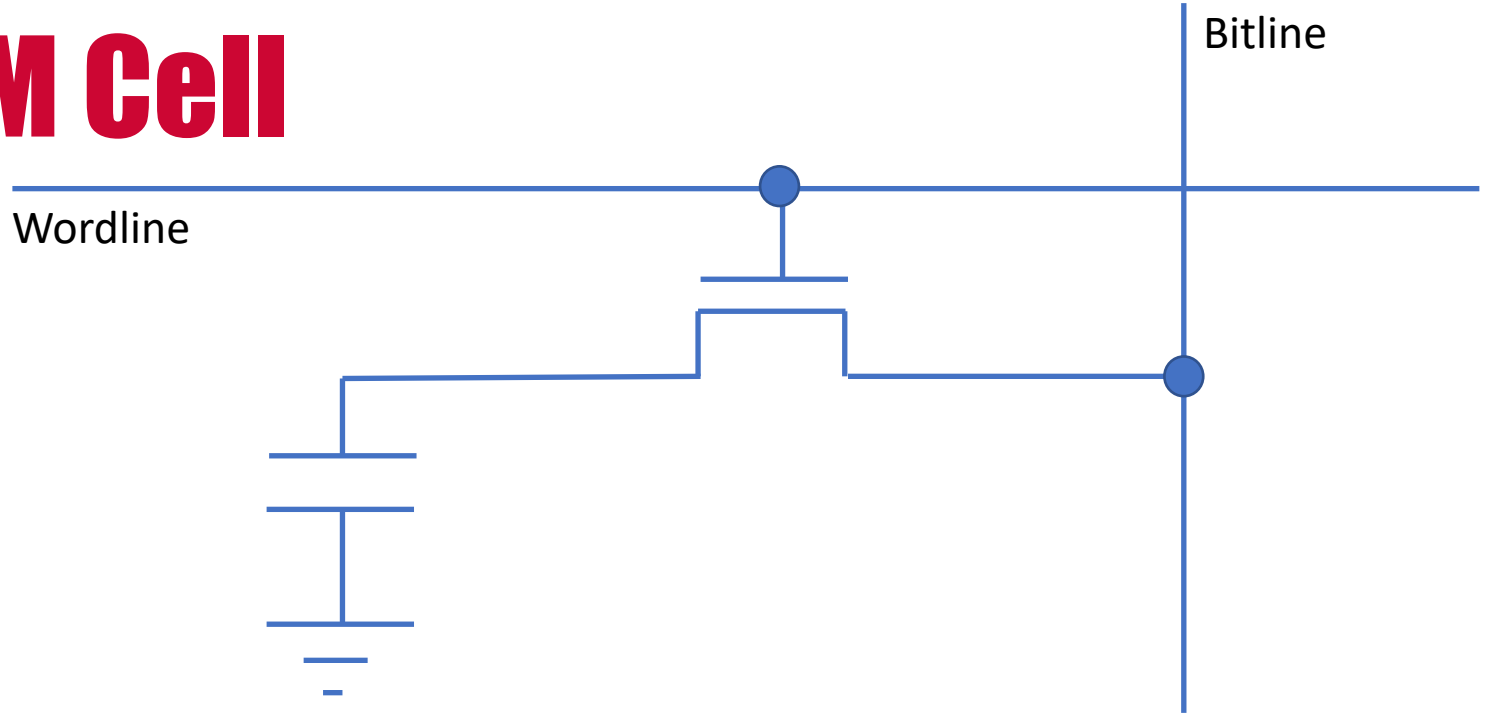
DRAM Basics

DRAM

SFU

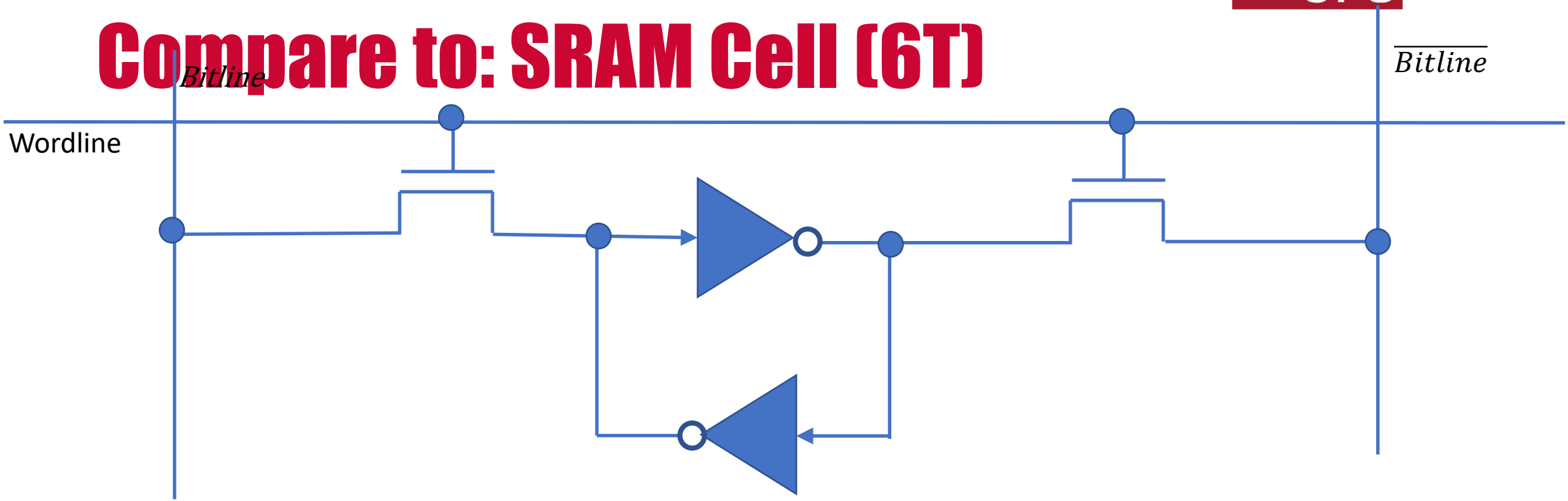
- Stands for “Dynamic Random Access Memory”
- Volatile memory, used as main memory in most computer systems
- DRAM cells are single-transistor, single-capacitor cells (1T1C)
  - Much higher density than 6T SRAM cells
- Data stored by charging or discharging capacitor
- Reads are destructive: Data needs 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 Types

SFU

- **DRAM has different types with different interfaces optimized for different purposes**
  - Commodity: DDR, DDR2, DDR3, DDR4, DDR5, …
  - Low power (for mobile): LPDDR1, …, LPDDR5, …
  - High bandwidth (for graphics): GDDR2, …, GDDR5, …
  - Low latency: eDRAM, RLDRAM, …
  - 3D stacked: WIO, HBM, HMC, HBM2.0, …
- **Underlying microarchitecture is fundamentally the same**
- **A flexible memory controller can support various DRAM types. This complicates the memory controller**
  - Difficult to support all types (and upgrades)
  - Analog interface is different for different DRAM types

Modern DRAM Types: Comparison to DDR3

SFU

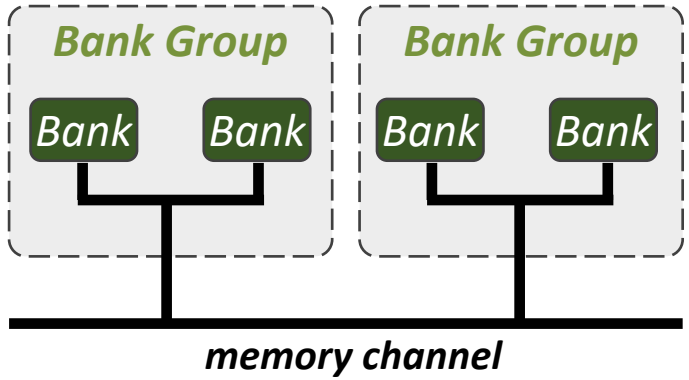
DRAM Type	Banks per Rank	Bank Groups	3D-Stacked	Low-Power
DDR3	8			
DDR4	16	✓		
GDDR5	16	✓		
HBM High-Bandwidth Memory	16		✓	
HMC Hybrid Memory Cube	256		✓	
Wide I/O	4		✓	✓
Wide I/O 2	8		✓	✓
LPDDR3	8			✓
LPDDR4	16			✓

increased latency

increased area/power

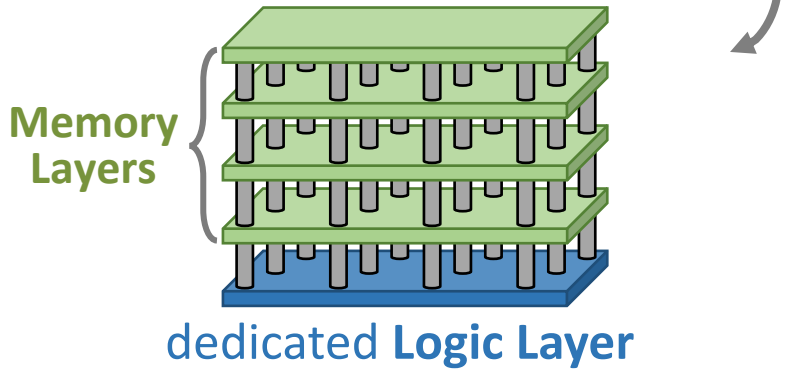
narrower rows, higher latency

- Bank groups



- 3D-stacked DRAM

high bandwidth with Through-Silicon Vias (TSVs)

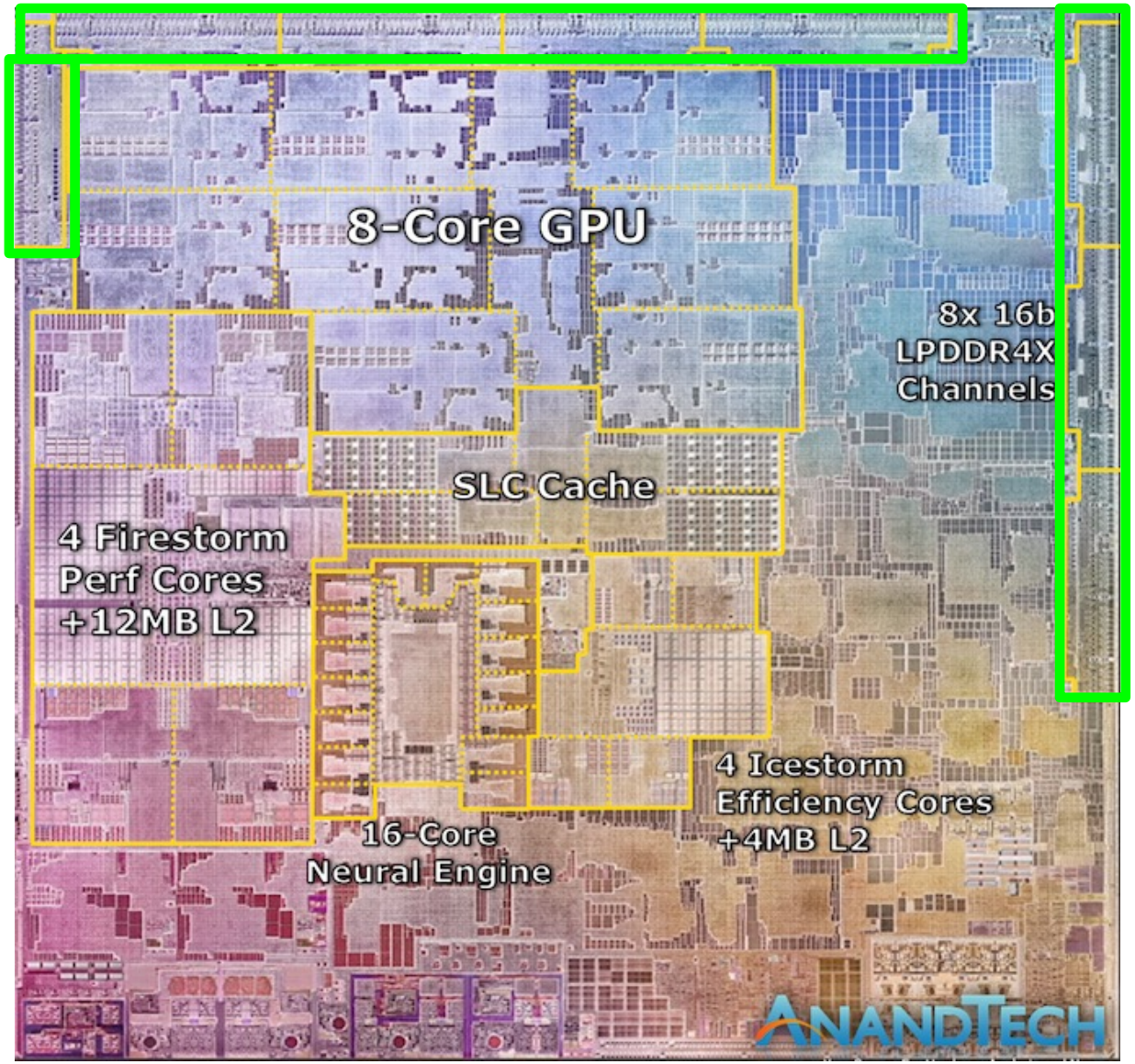


SFU

Memory Controllers

- **Controls Memory Channel**
  - Sends **bank**, **row**, and **column addresses** to memory via the **address bus**.
  - Sends **control signals,** Row Address Strobe (RAS), Column Address Strobe (CAS), Output enable, clock, and clock enable.
  - Sends/receives data to/from memory via the **data bus**.
- **Rank Selection**
  - Selects the active rank by enabling **Chip Select (CS)** signals on the **chip-select bus**.
- **Address Decoding:** Chip Select bits, Bank address bits, Row/column addresses (based on DRAM configuration).
- **Special Functions**
  - **Error detection and correction (ECC), Prefetching** for improved performance.
  - **Initiating parallel memory accesses**

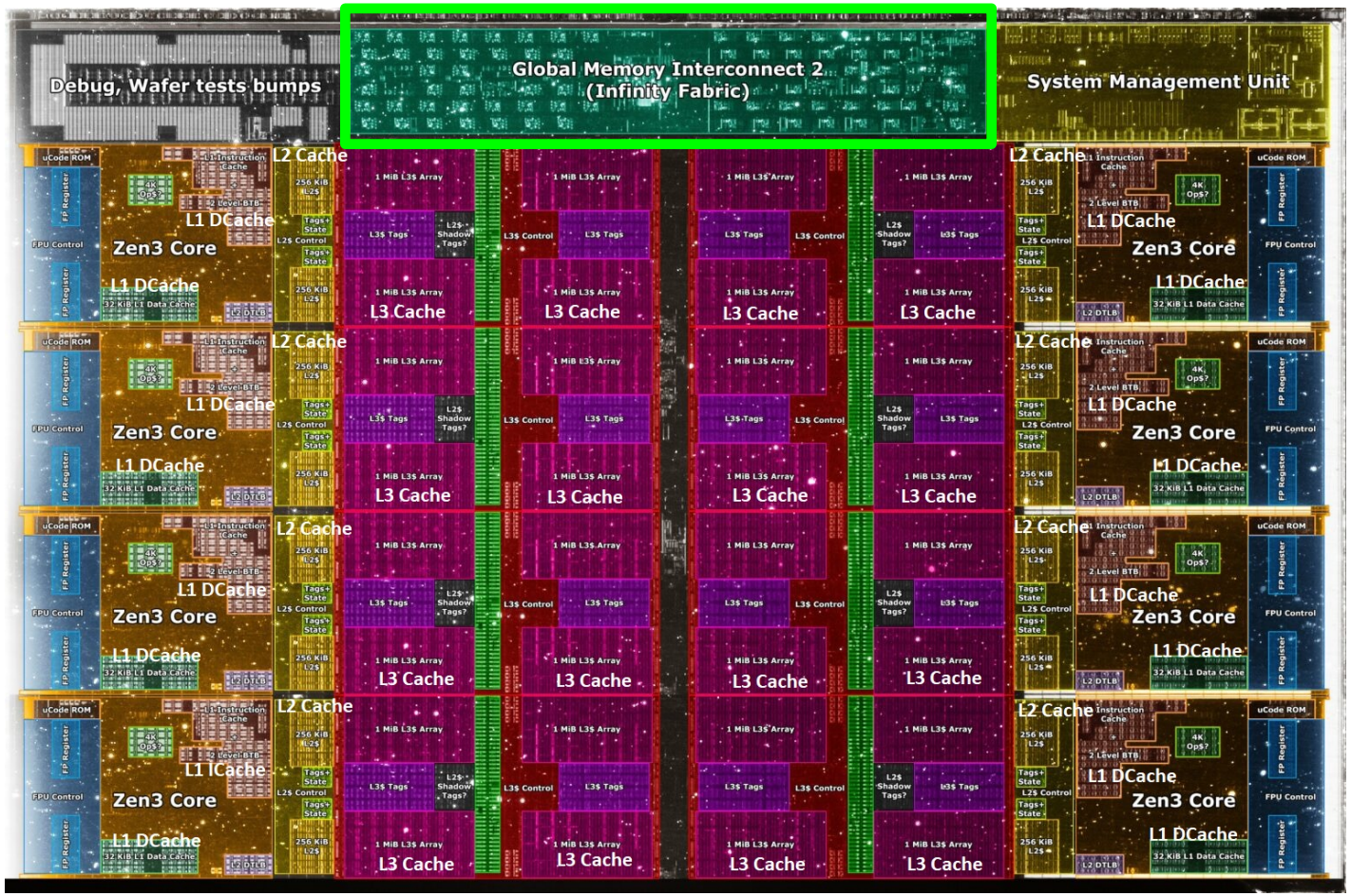
DRAM Control Logic Is Large



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

DRAM Control Logic Is Large



AMD Ryzen 5000, 2020

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

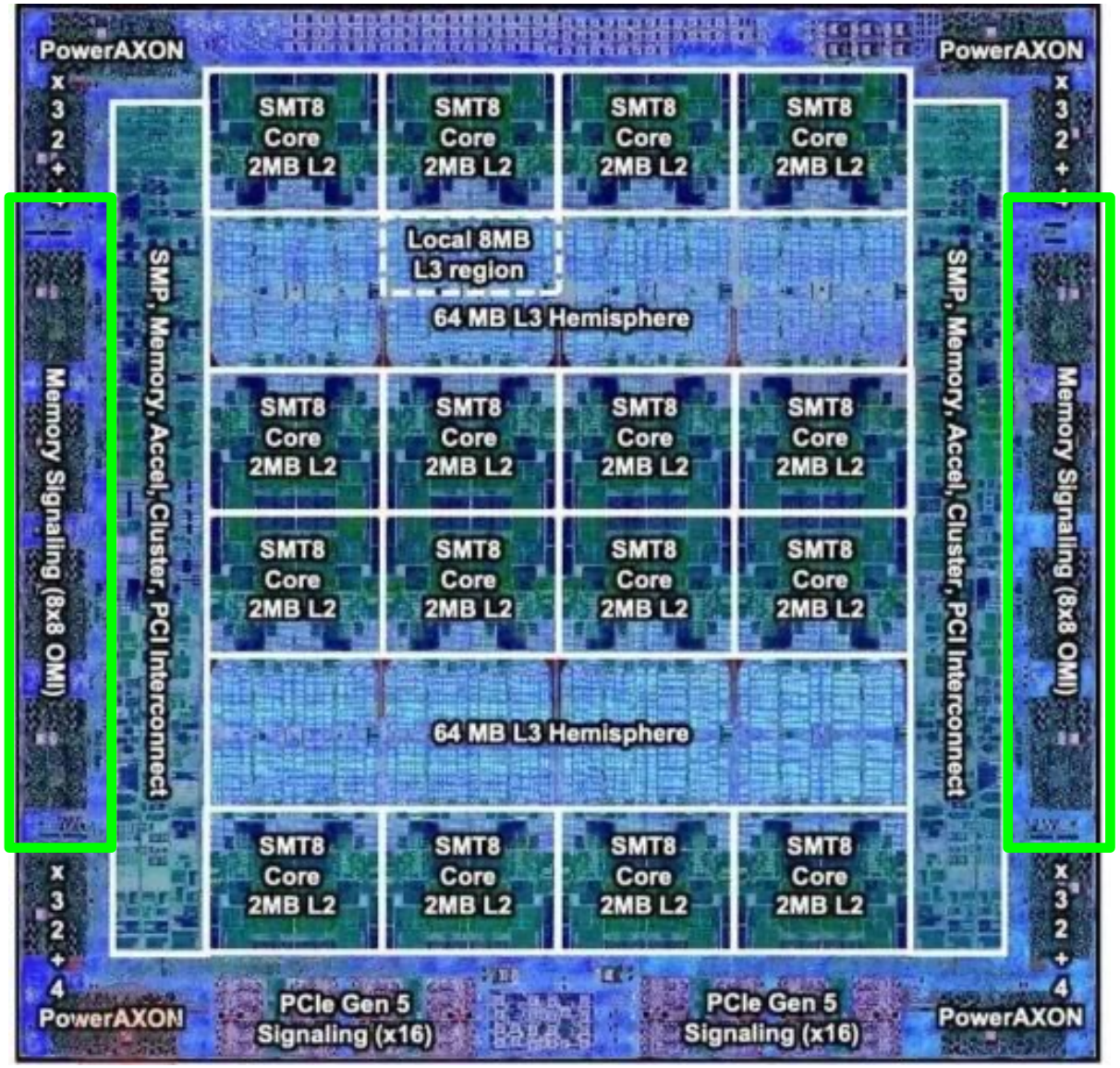
Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

DRAM Control Logic Is Large



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

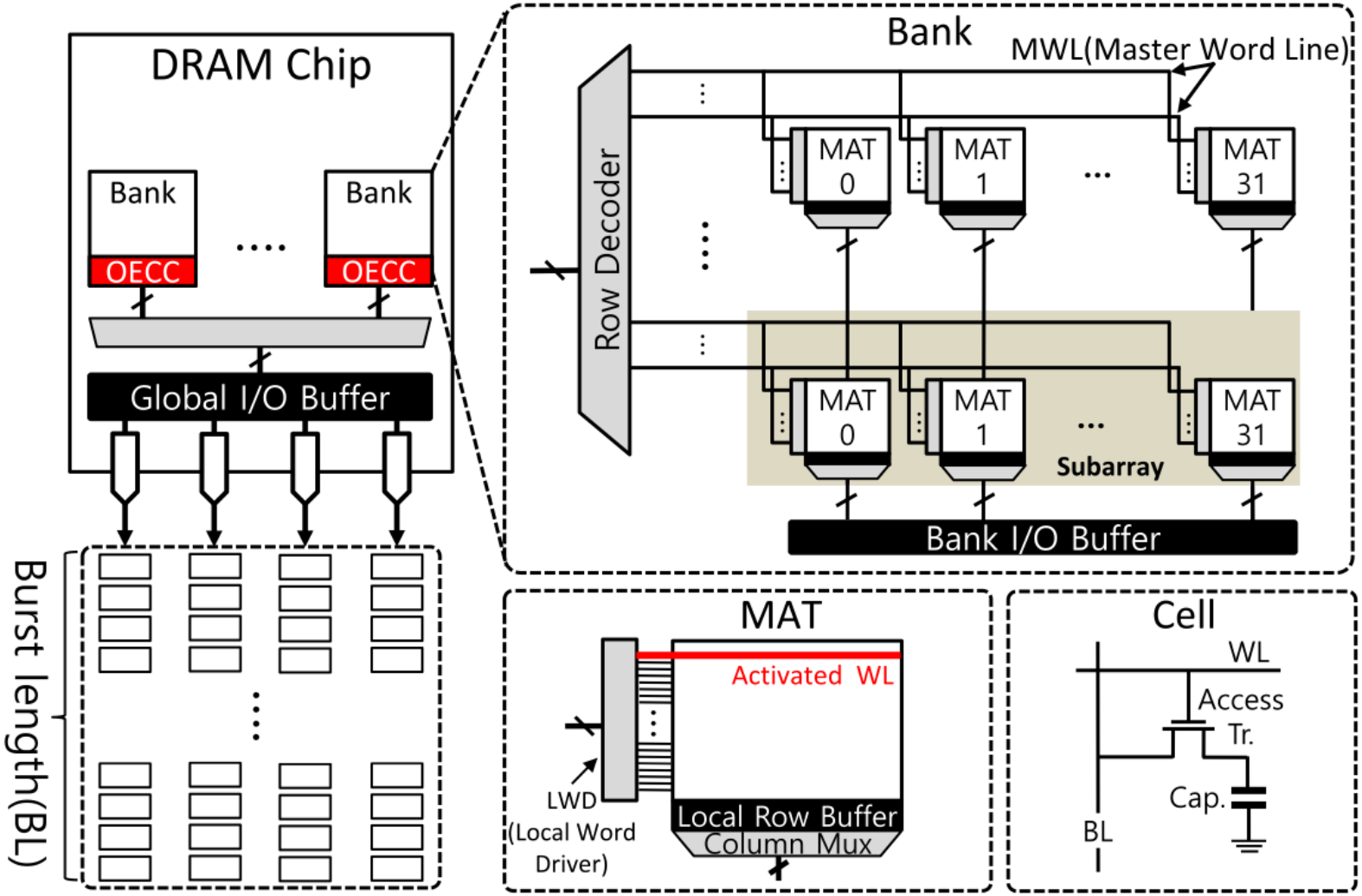
L2 Caches:

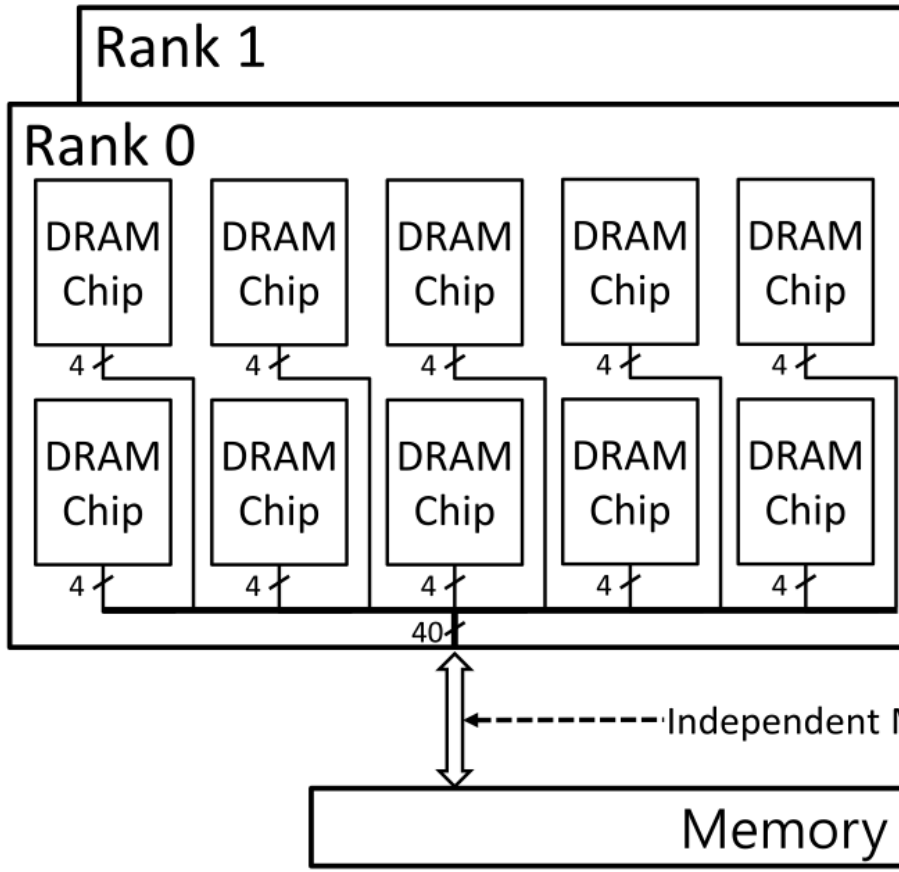
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

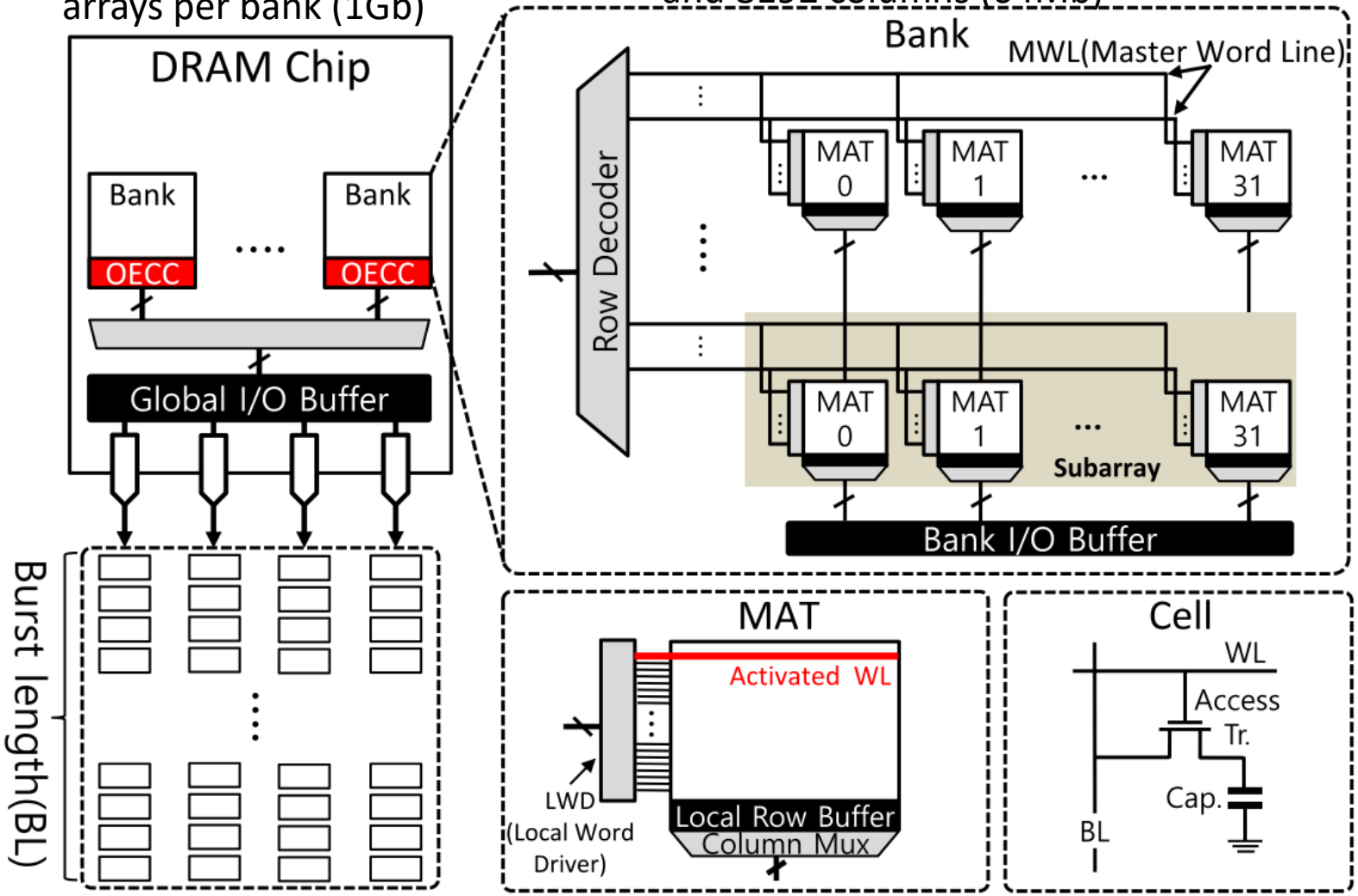

DRAM Chip
Bank
OECC
Bank
OECC
Global I/O Buffer
Burst length(BL)
Bank
MWL(Master Word Line)
Row Decoder
MAT 0
MAT 1
MAT 31
Subarray
Bank I/O Buffer
MAT
Activated WL
LWD (Local Word Driver)
Local Row Buffer
Column Mux
Cell
WL
Access Tr.
Cap.
BL

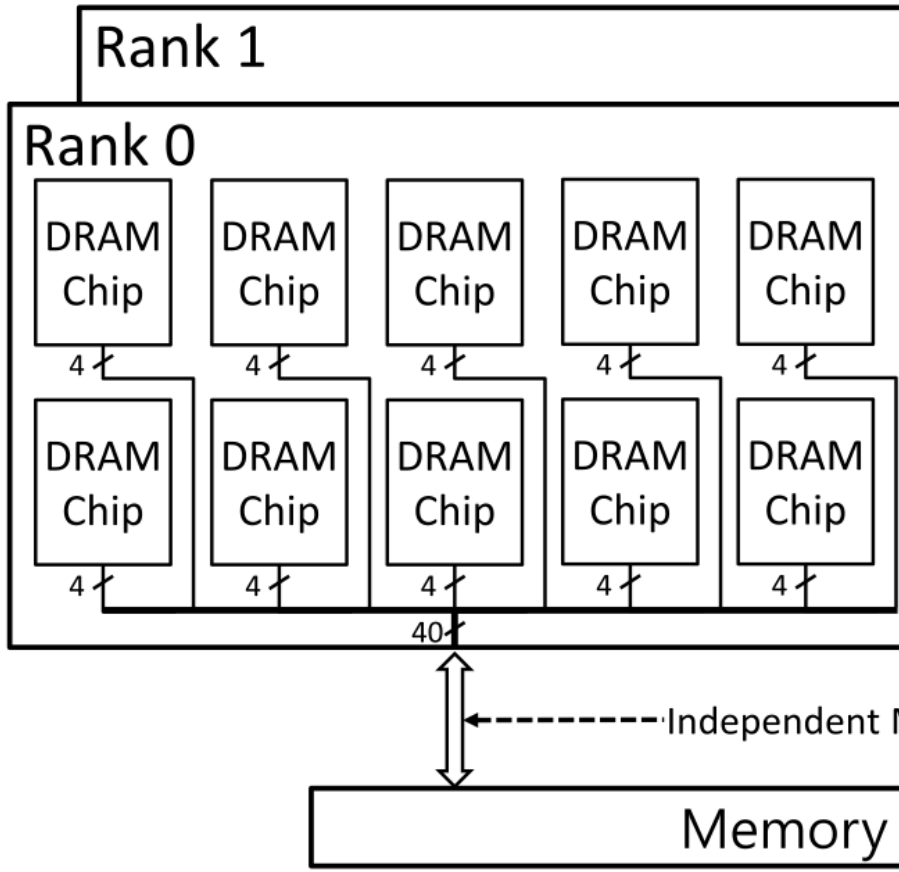

Rank 1
Rank 0
DRAM Chip
4
40
Independent
Memory

A x16 DIMM has 16 arrays per bank (1Gb)

A DRAM array contains 8192 rows and 8192 columns (64Mb)

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 dual-socket system has a maximum DRAM capacity of 2x64GB = 128 GB

A single socket may have 4 memory channels each with 2 DIMMs, so capacity per socket = 4x2x8GB = 64GB

If each DIMM has

SFU

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

SFU

Why?

- **Why multiple ranks?**
  - Increases capacity per module
  - Enables rank interleaving for higher performance.
  - Better power management by activating only necessary ranks.

- **Why multiple banks?**
  - Improves parallelism through bank interleaving.
  - Reduces latency by hiding internal operations.
  - Maximizes memory bus utilization.

SFU

Why are memory controllers difficult to design?

- **Need to obey DRAM timing constraints for correctness**
  - There are many (50+) timing constraints in DRAM
  - tWTR: Minimum number of cycles to wait before issuing a read command after a write command is issued
  - tRC: Minimum number of cycles between the issuing of two consecutive activate commands to the same bank
  - …
- **Need to keep track of many resources to prevent conflicts**
  - Channels, banks, ranks, data bus, address bus, row buffers
- **Need to handle DRAM refresh**
- **Need to manage power consumption**
- **Need to optimize performance & QoS (in the presence of constraints)**
  - Reordering is not simple
  - Fairness and QoS needs complicates the scheduling problem

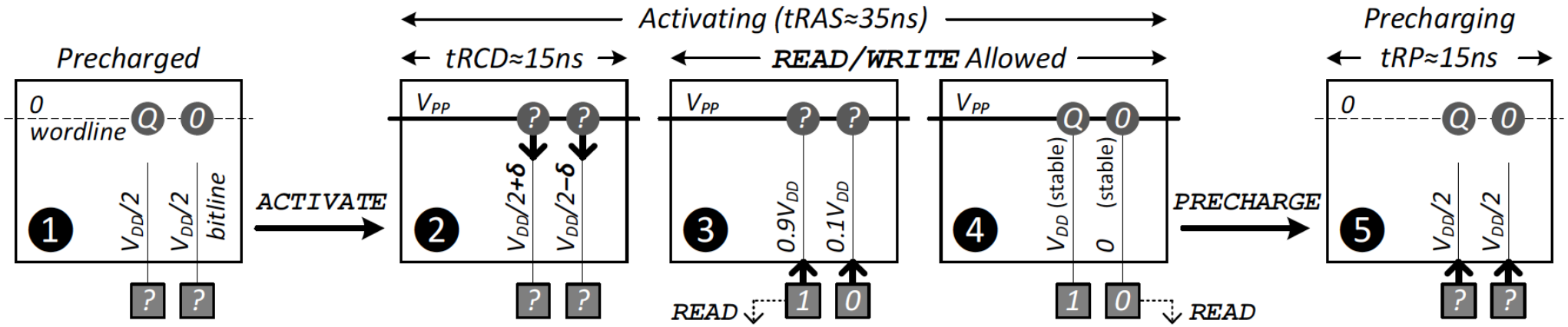


Figure 4. DRAM bank operation: Steps involved in serving a memory request [17] ($V_{PP} > V_{DD}$)

Category	RowCmd↔RowCmd			RowCmd↔ColCmd			ColCmd↔ColCmd			ColCmd→DATA	
Name	*tRC*	*tRAS*	*tRP*	*tRCD*	*tRTP*	*tWR**	*tCCD*	*tRTW*†	*tWTR**	*CL*	*CWL*
Commands	A→A	A→P	P→A	A→R/W	R→P	W*→P	R(W)→R(W)	R→W	W*→R	R→DATA	W→DATA
Scope	Bank	Bank	Bank	Bank	Bank	Bank	Channel	Rank	Rank	Bank	Bank
Value (ns)	**~50**	~35	13-15	13-15	~7.5	**15**	5-7.5	11-15	~7.5	13-15	10-15

A: ACTIVATE– P: PRECHARGE– R: READ– W: WRITE

∗ Goes into effect after the last write *data*, not from the WRITE command

† Not explicitly specified by the JEDEC DDR3 standard [18]. Defined as a function of other timing constraints.

Table 1. Summary of DDR3-SDRAM timing constraints (derived from Micron's 2Gb DDR3-SDRAM datasheet [33])

Credit: Prof. Onur Mutlu

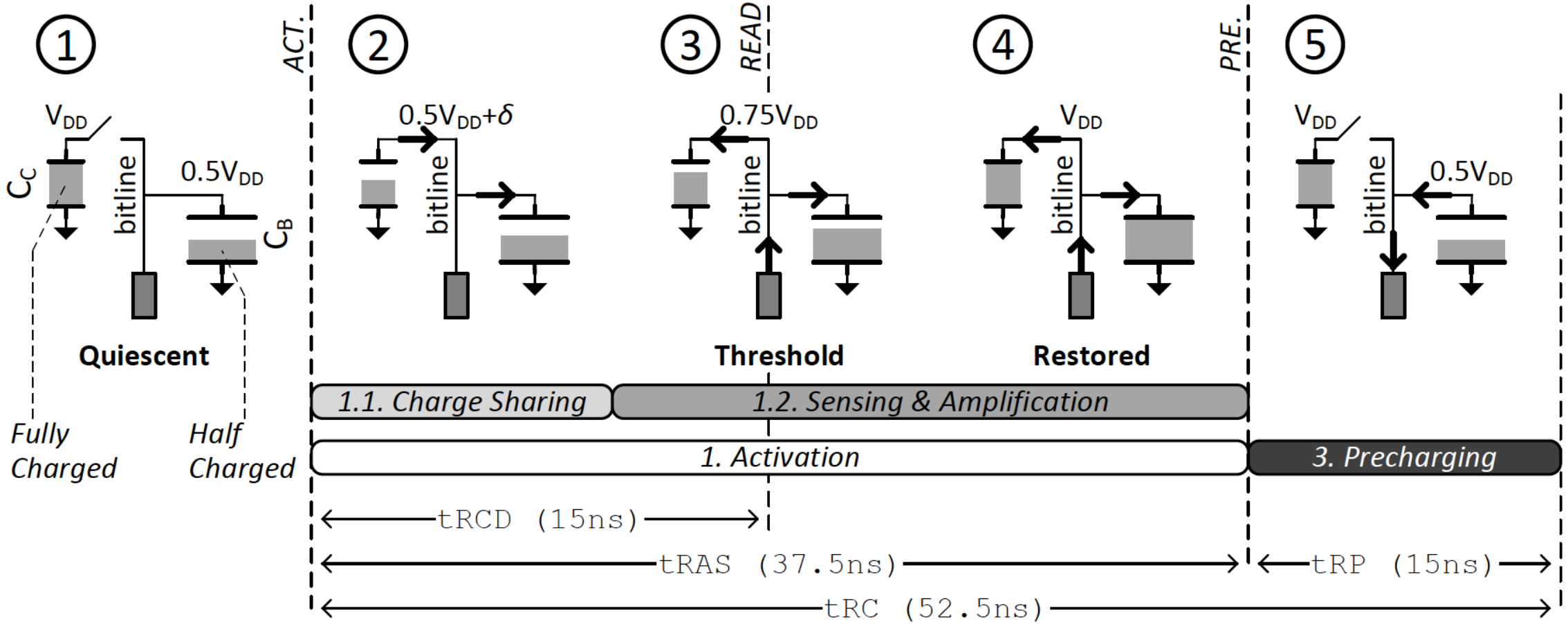
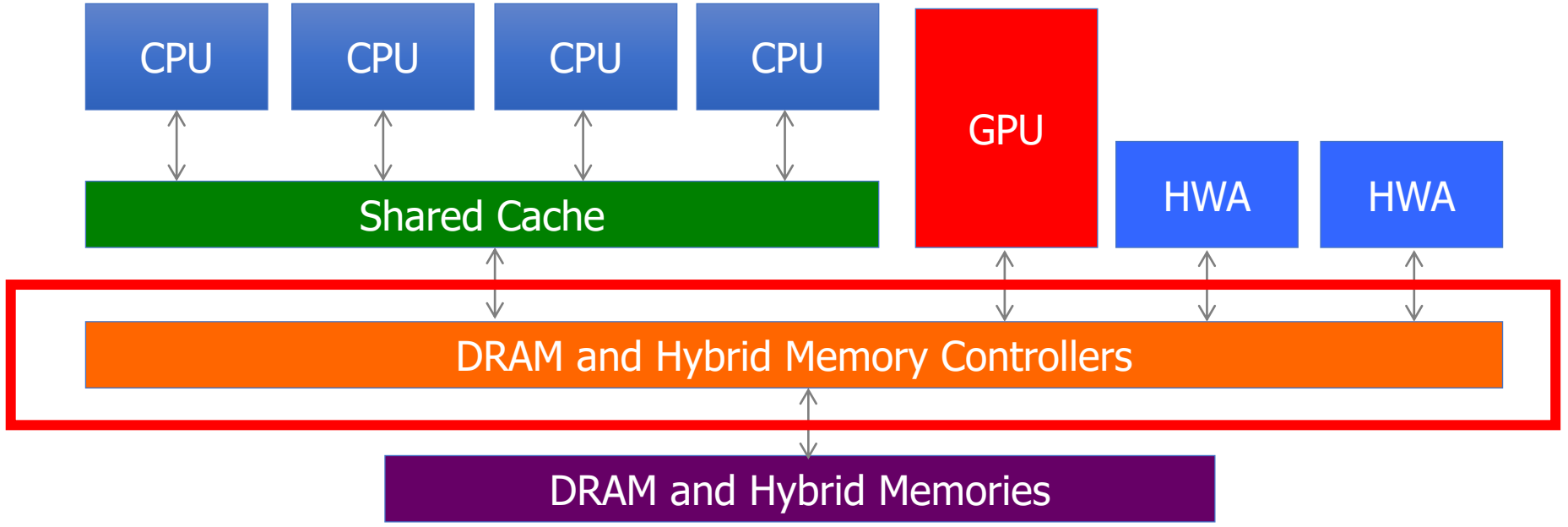


Figure 6. Charge Flow Between the Cell Capacitor (C_C), Bitline Parasitic Capacitor (C_B), and the Sense-Amplifier ($C_B \approx 3.5C_C$ [39])

Credit: Prof. Onur Mutlu

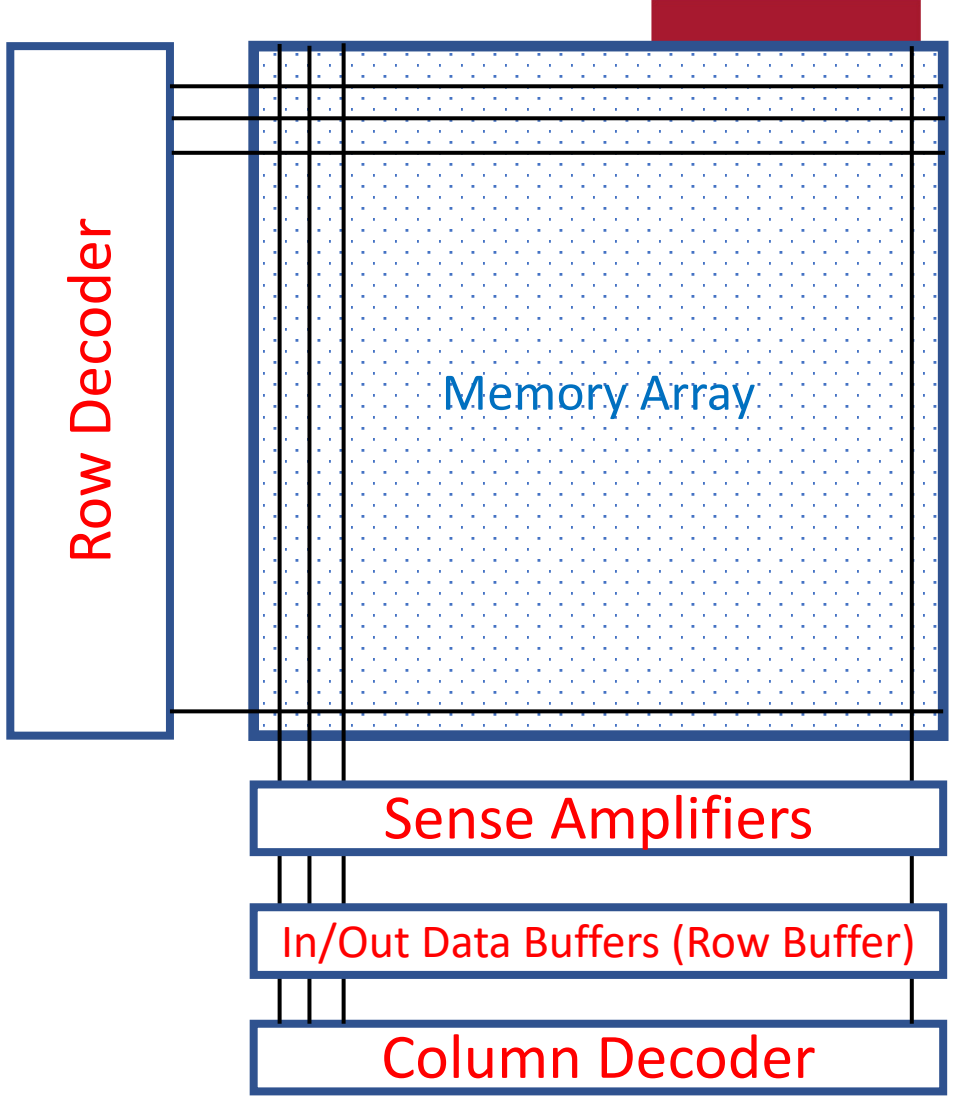
Memory Controller Design Is Becoming More Difficult

SFU



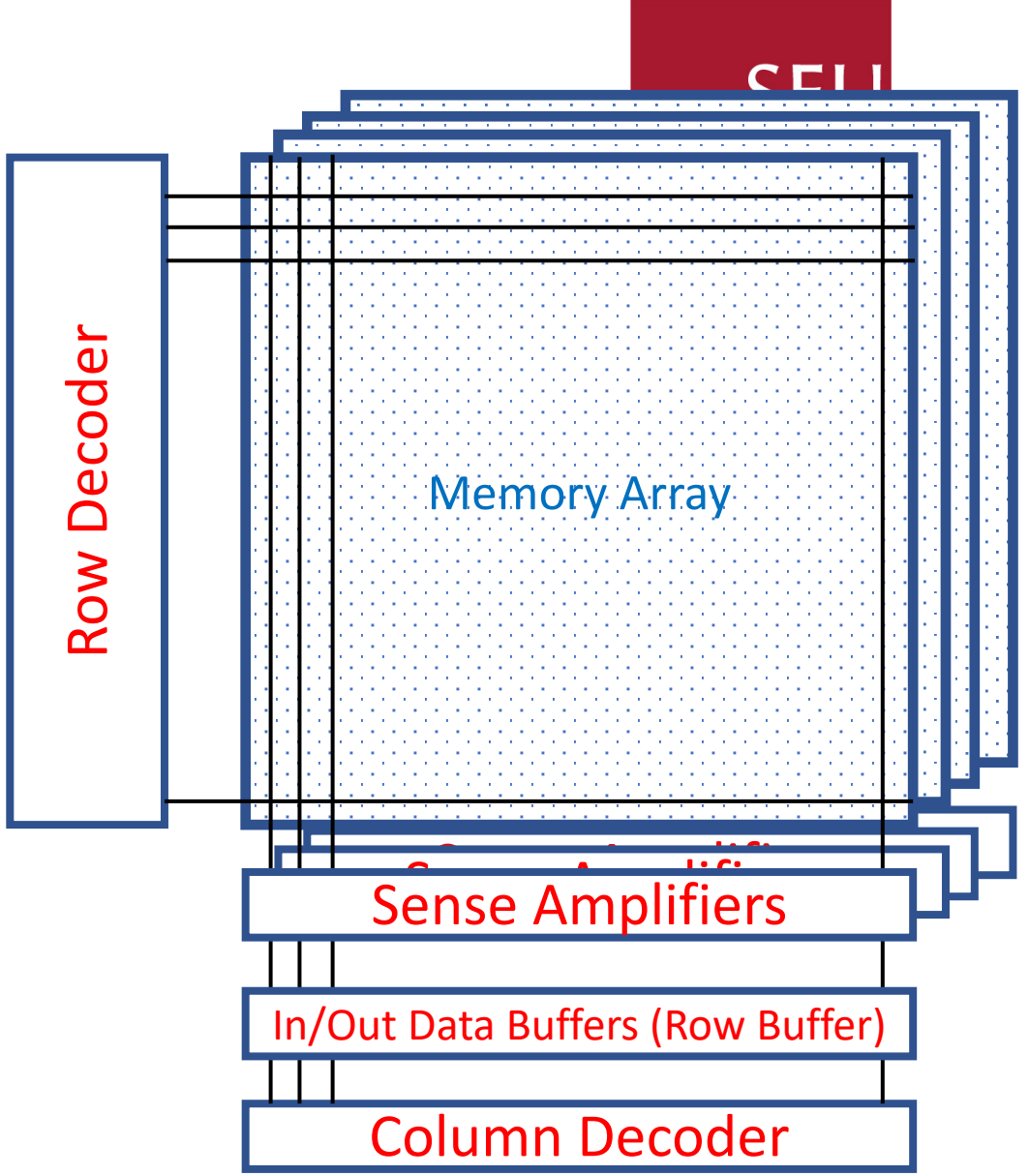
Memory Array

- 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



Row Buffer

SFU

- 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

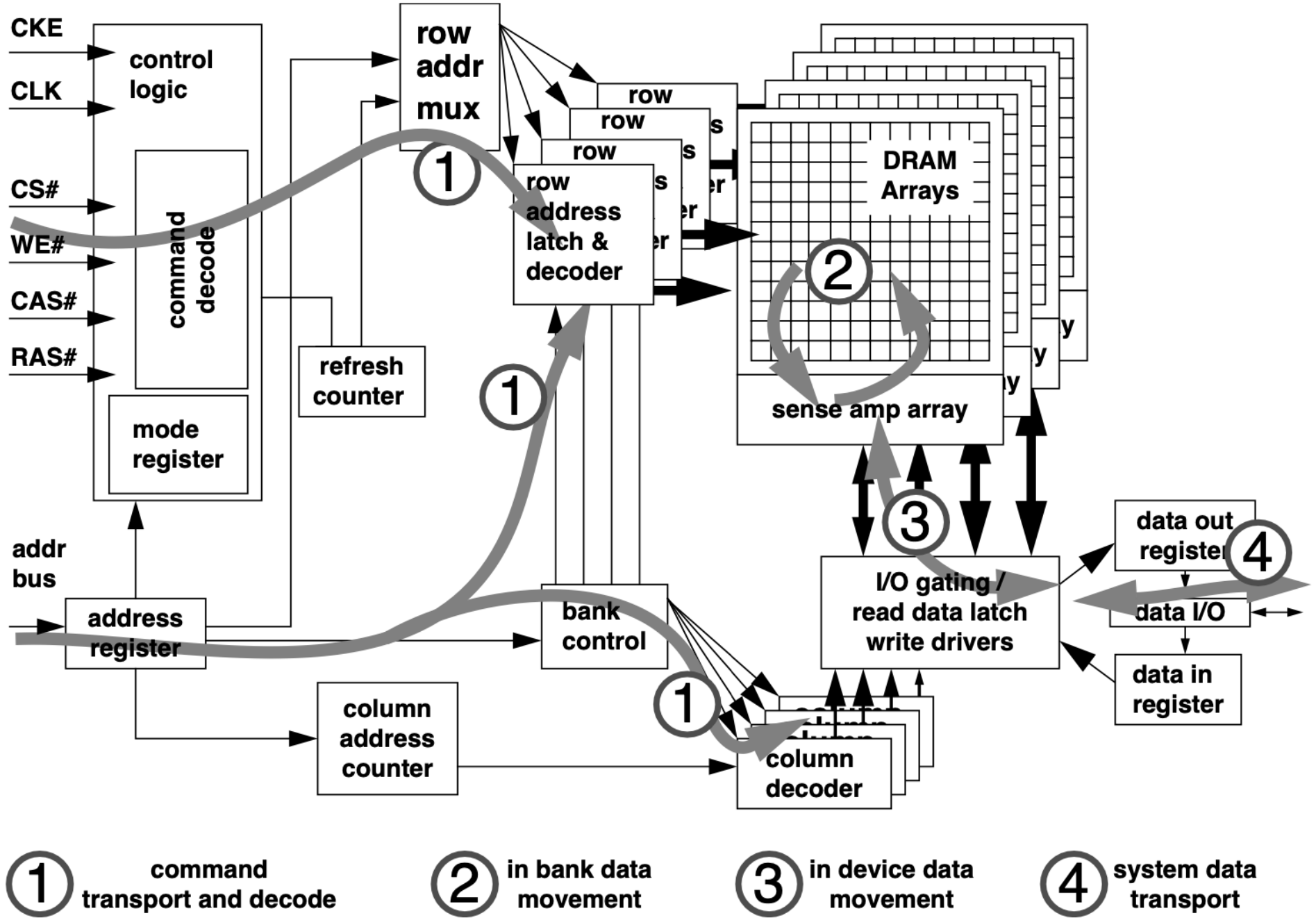
SFU

Row-buffer management policies

- **Open row**
  - Keep the row open after an access
  + Next access might need the same row → row hit
  -- Next access might need a different row → row conflict, wasted energy

- **Closed row**
  - Close the row after an access (if no other requests already in the request buffer need the same row)
  + Next access might need a different row → avoid a row conflict
  -- Next access might need the same row → extra activate latency

- **Adaptive policies**
  - Predict whether or not the next access to the bank will be to the same row and act accordingly


CKE
CLK
CS#
WE#
CAS#
RAS#
control logic
command decode
mode register
refresh counter
row addr mux
row address latch & decoder
row
DRAM Arrays
sense amp array
addr bus
address register
bank control
column address counter
column decoder
I/O gating / read data latch write drivers
data out register
data I/O
data in register
1
2
3
4
1 command transport and decode
2 in bank data movement
3 in device data movement
4 system data transport

Steps for a Memory Read Operation

SFU

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

Refresh Operations

SFU

- 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

SFU

DRAM Power Modes

- **DRAM chips have power modes**
- **Idea: When not accessing a chip power it down**

- **Power states**
  - Active (highest power)
  - All banks idle
  - Power-down
  - Self-refresh (lowest power)

- **Tradeoff: State transitions incur latency during which the chip cannot be accessed**

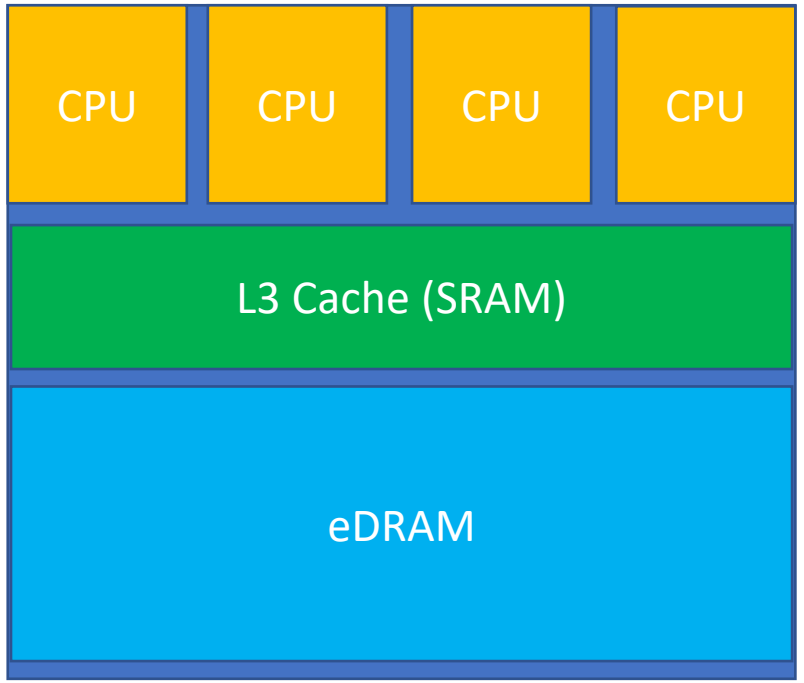
Other Memory Technologies: Embedded DRAM

- 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



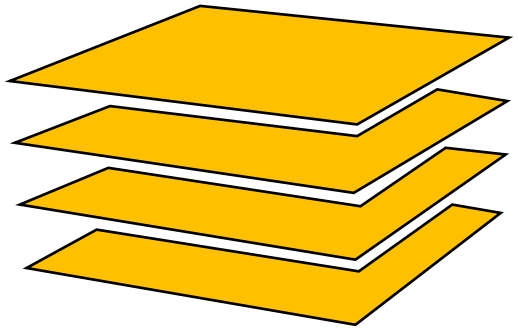
Other Memory Technologies: Stacked DRAM

- 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: Non-Volatile Memory

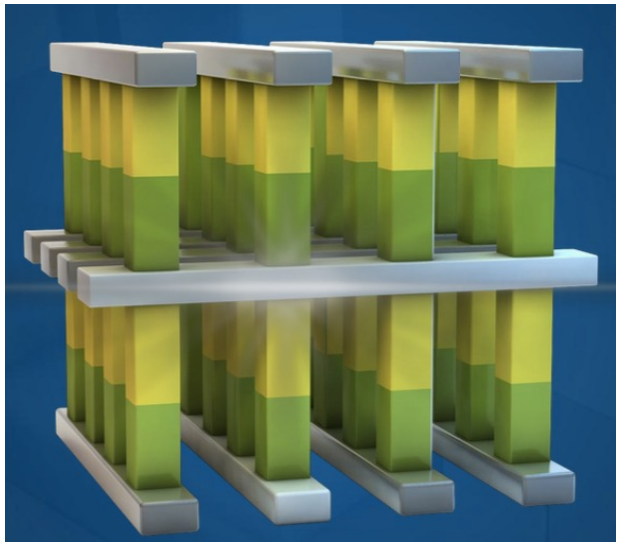
SFU

- 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

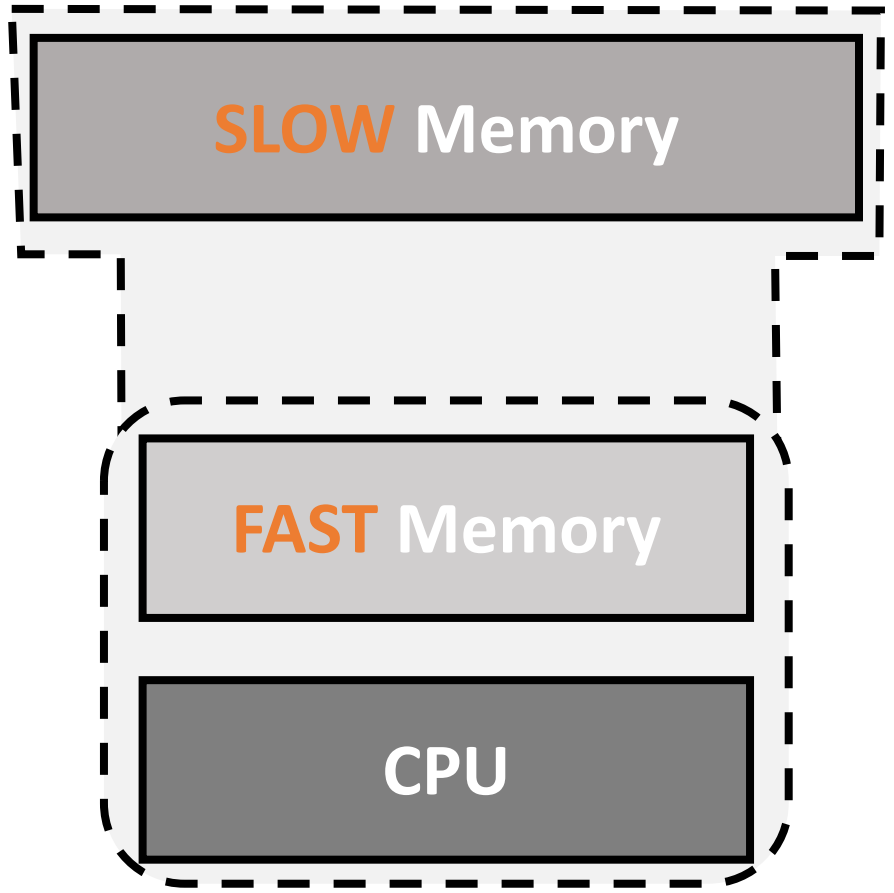


3D Xpoint
(Intel, Micron)

Heterogeneous Memory Systems

SFU

- **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

SFU

- 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.

SFU

Memory Consistency

SFU

Memory consistency is Hard

- Memory consistency models are difficult to understand
  - Knowing when and how to use memory barriers in your programs takes a long time to master

- I read the long version of this paper about once a year – Started in graduate architecture, still mastering this

- Even if you can't master this material, it is worth conveying some intuitions and getting you started on the path
  - Multi-core programming is increasingly common

Multithreaded programs

- Initially A =B =0

- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

- What can be printed?
- “Hello”?
- “World”?
- Nothing?
- “Hello World”?

Multithreaded programs

- Initially A =B =0

- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

- What can be printed?
- “Hello”?
- “World”?
- Nothing?
- “Hello World”?

Multithreaded programs

- Initially A =B =0
- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;
- What can be printed?
- “Hello”?

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

```
A = 1
if (B == 0)
    print “Hello”;
B = 1
if (A == 0)
    print “World”;
```

SFU

Multithreaded programs

- Initially A =B =0
- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;
- What can be printed?
- “Hello”?
- ”World”?

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

```
B = 1
if (A == 0)
    print “World”;
A = 1
if (B == 0)
    print “Hello”;
```

SFU

Multithreaded programs

- Initially A =B =0
- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;
- What can be printed?
- “Hello”?
- ”World”?
- Nothing

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

```
B = 1
A = 1
if (A == 0)
    print “World”;
if (B == 0)
    print “Hello”;
```

Multithreaded programs

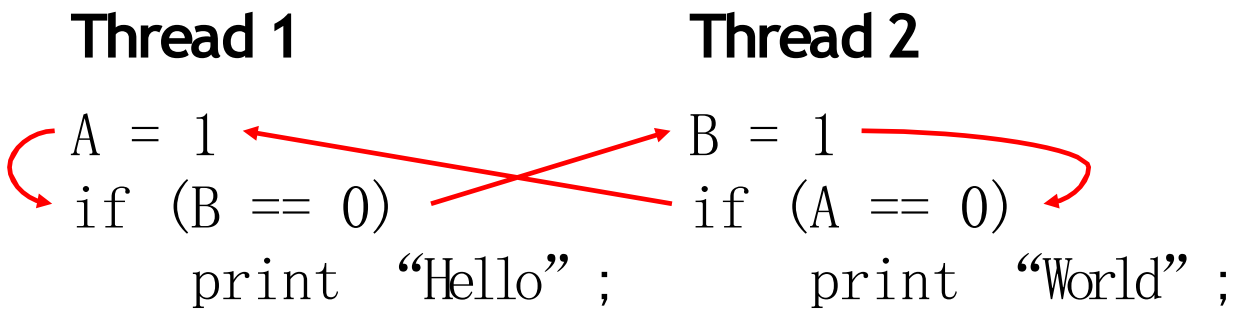
- Initially A =B =0
- Thread 1
- A = 1
- if (B == 0)
  - print “Hello”;
- What can be printed?
- “Hello”?
- ”World”?
- Hello World

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

Things that shouldn't happen

This program should never print "Hello World".



A "happens-before" graph shows the order in which events must execute to get a desired outcome.

- If there's a cycle in the graph, an outcome is impossible—an event must happen before itself!

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)**

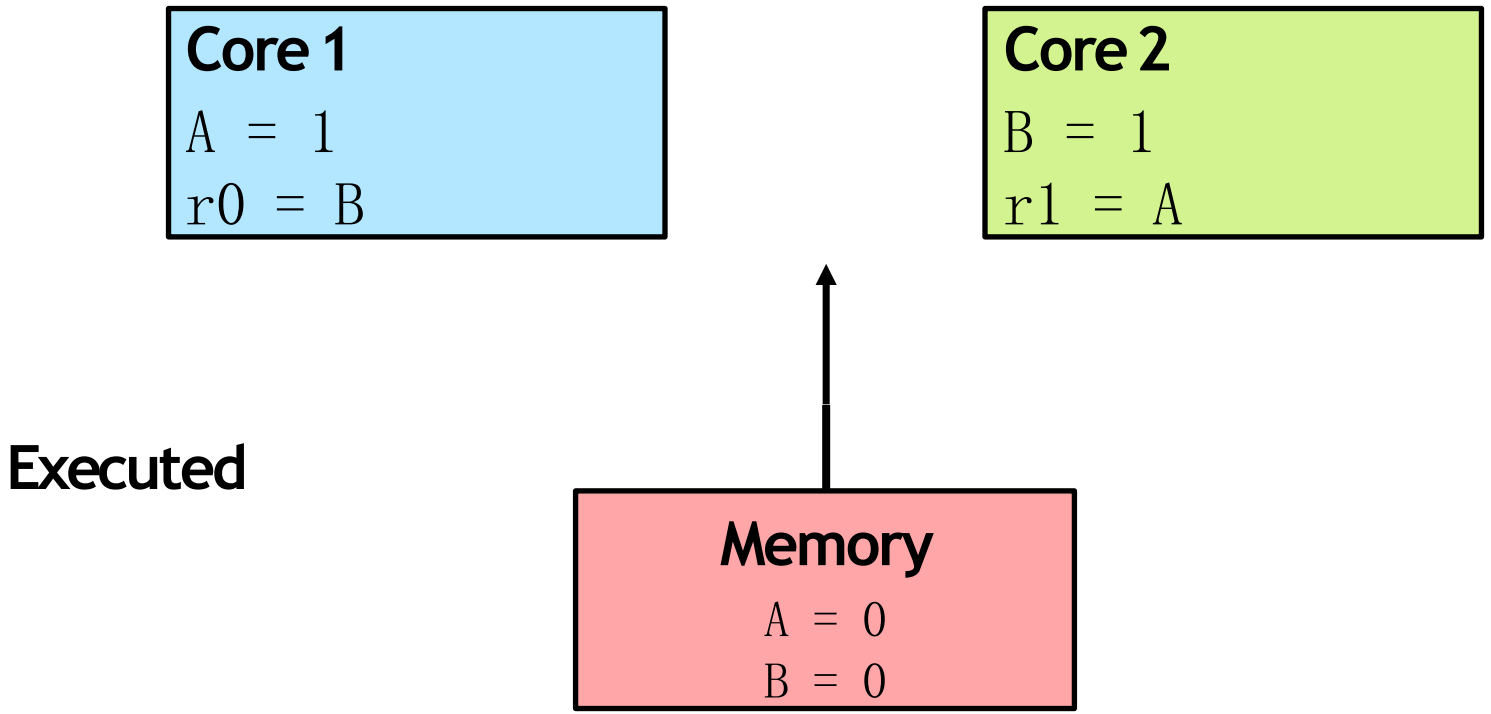
```
P1:     A = 0;              P2:     B = 0;
        .....                       .....
        A = 1;                      B = 1;
L1:     if (B == 0)...      L2:     if (A == 0)...
```

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

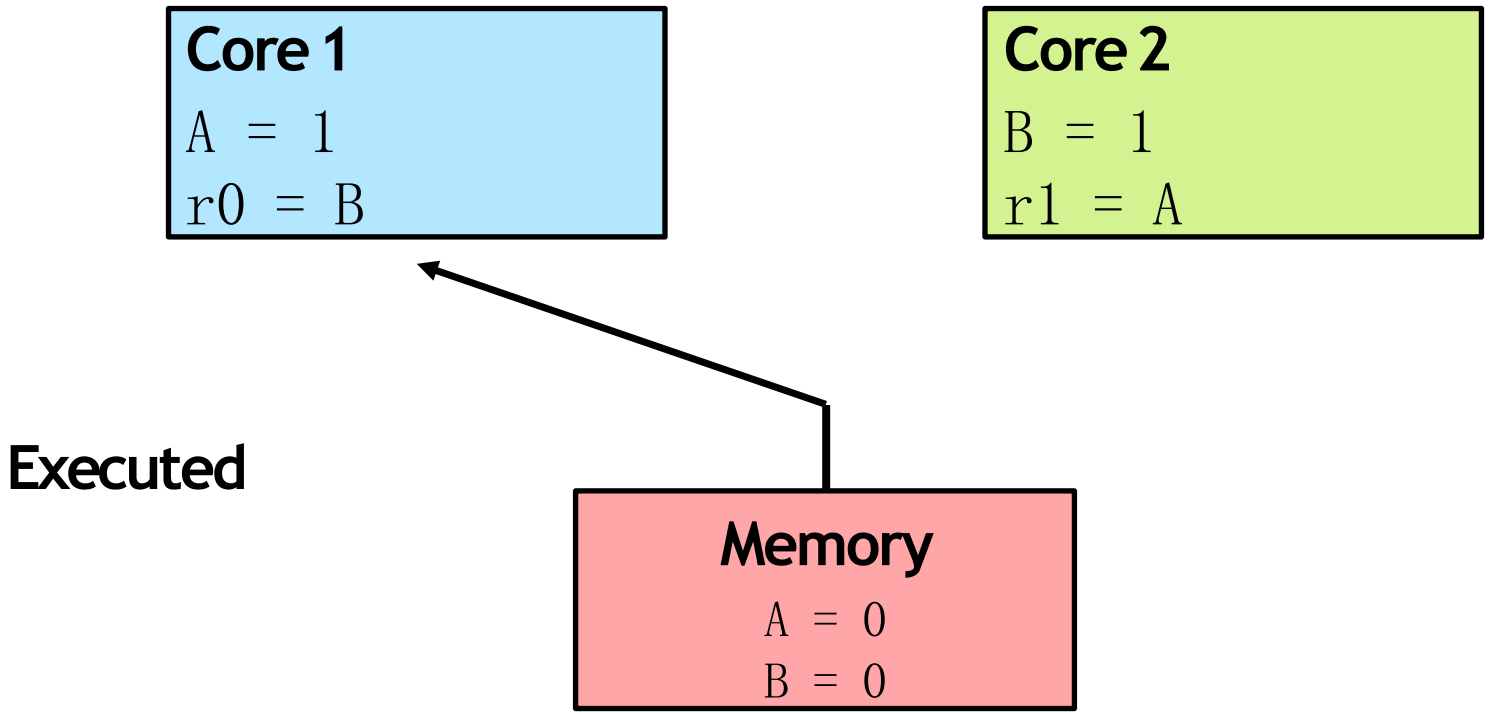
Sequential consistency

Can be seen as a "switch" running one instruction at a time



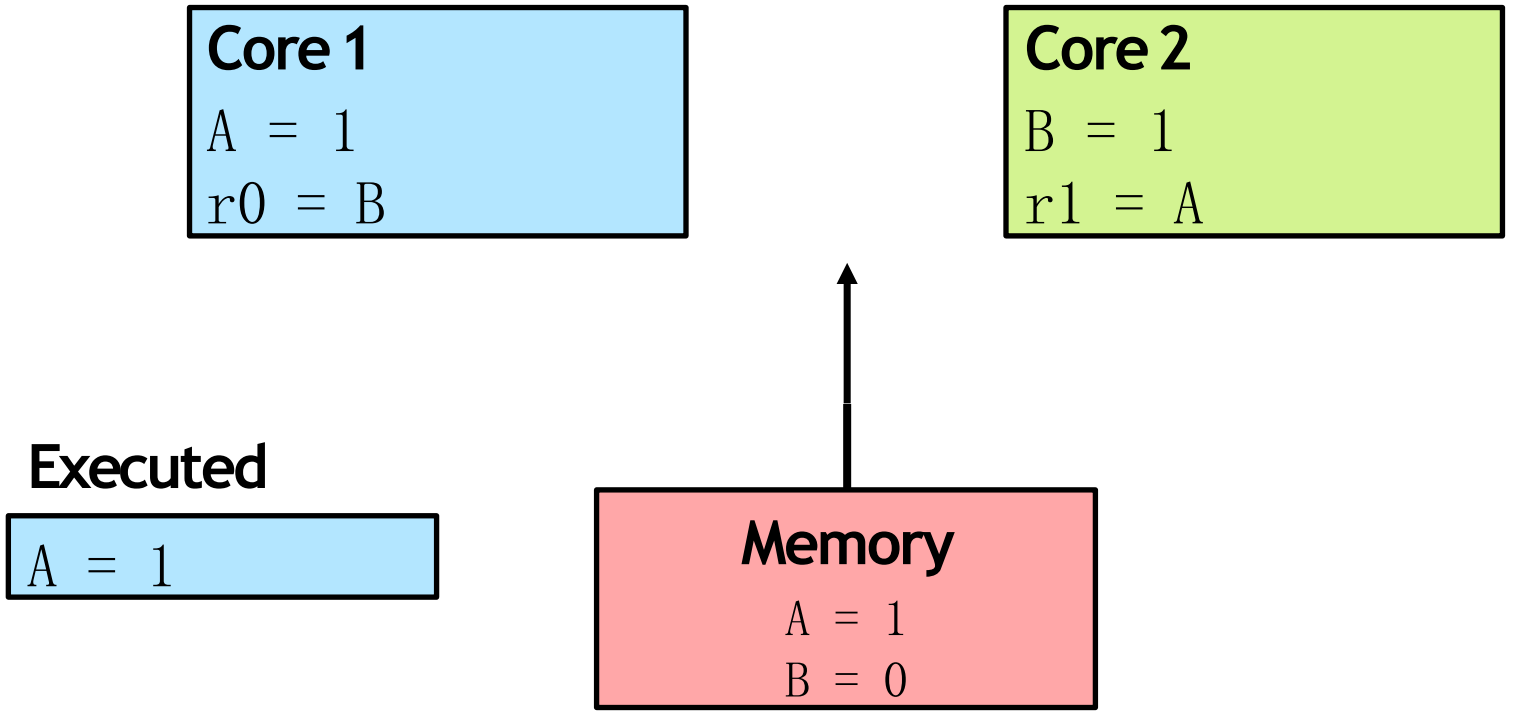
Sequential consistency

Can be seen as a "switch" running one instruction at a time



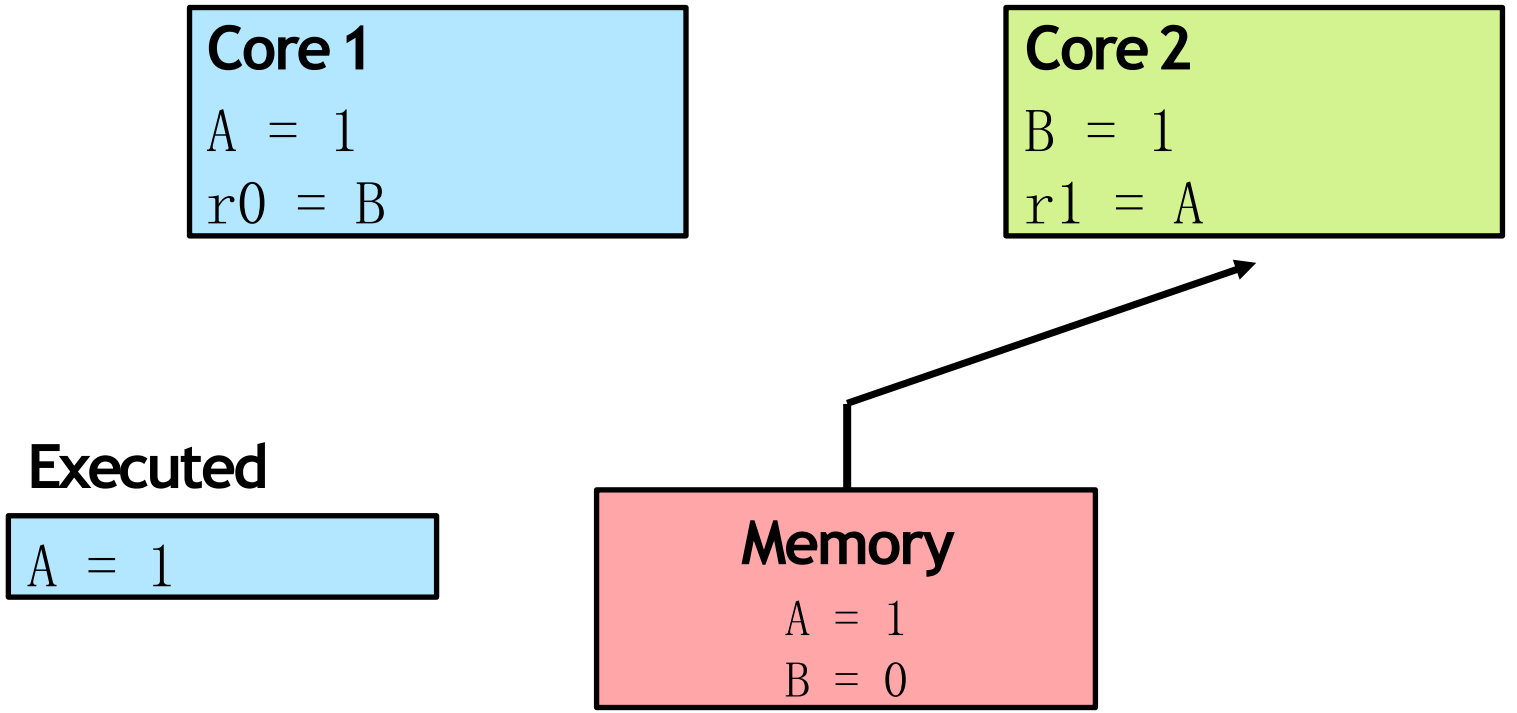
Sequential consistency

Can be seen as a "switch" running one instruction at a time



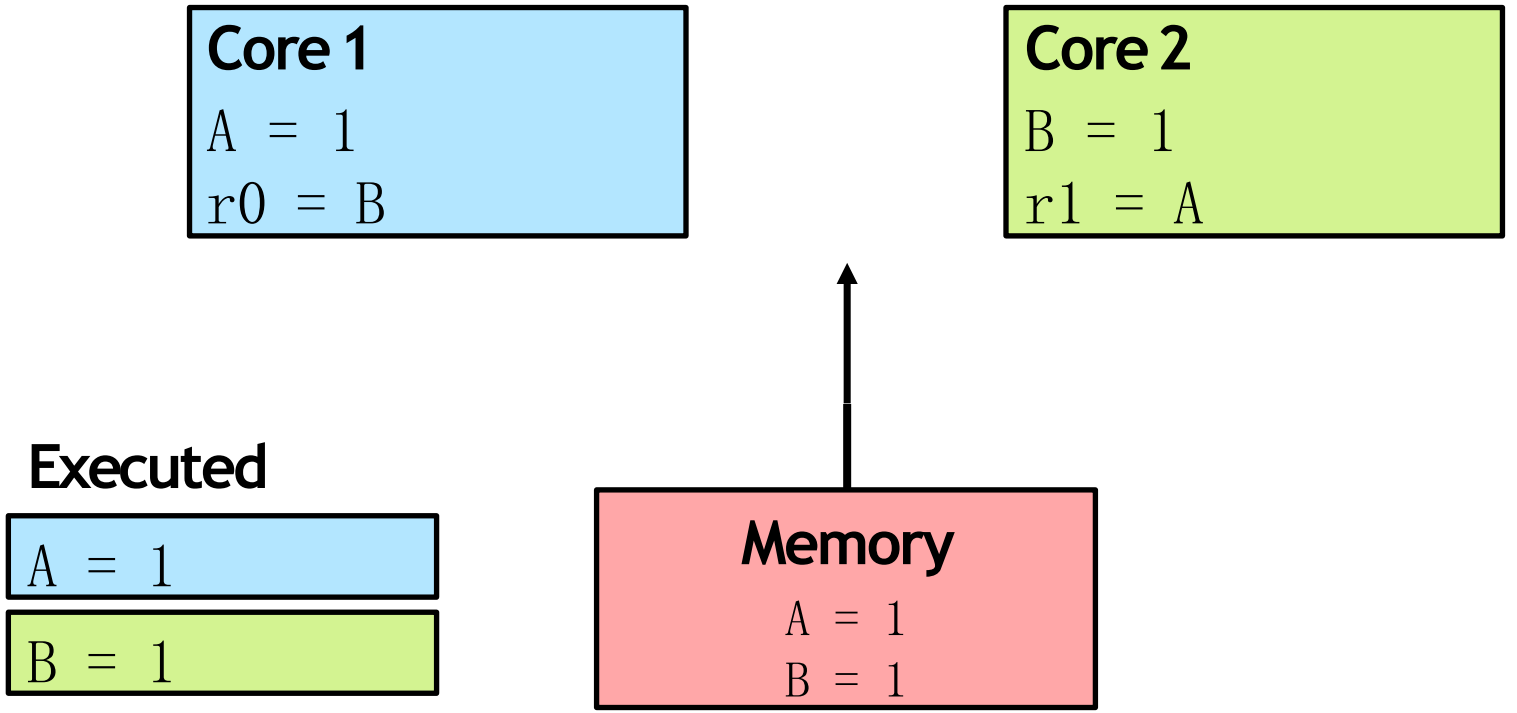
Sequential consistency

Can be seen as a "switch" running one instruction at a time



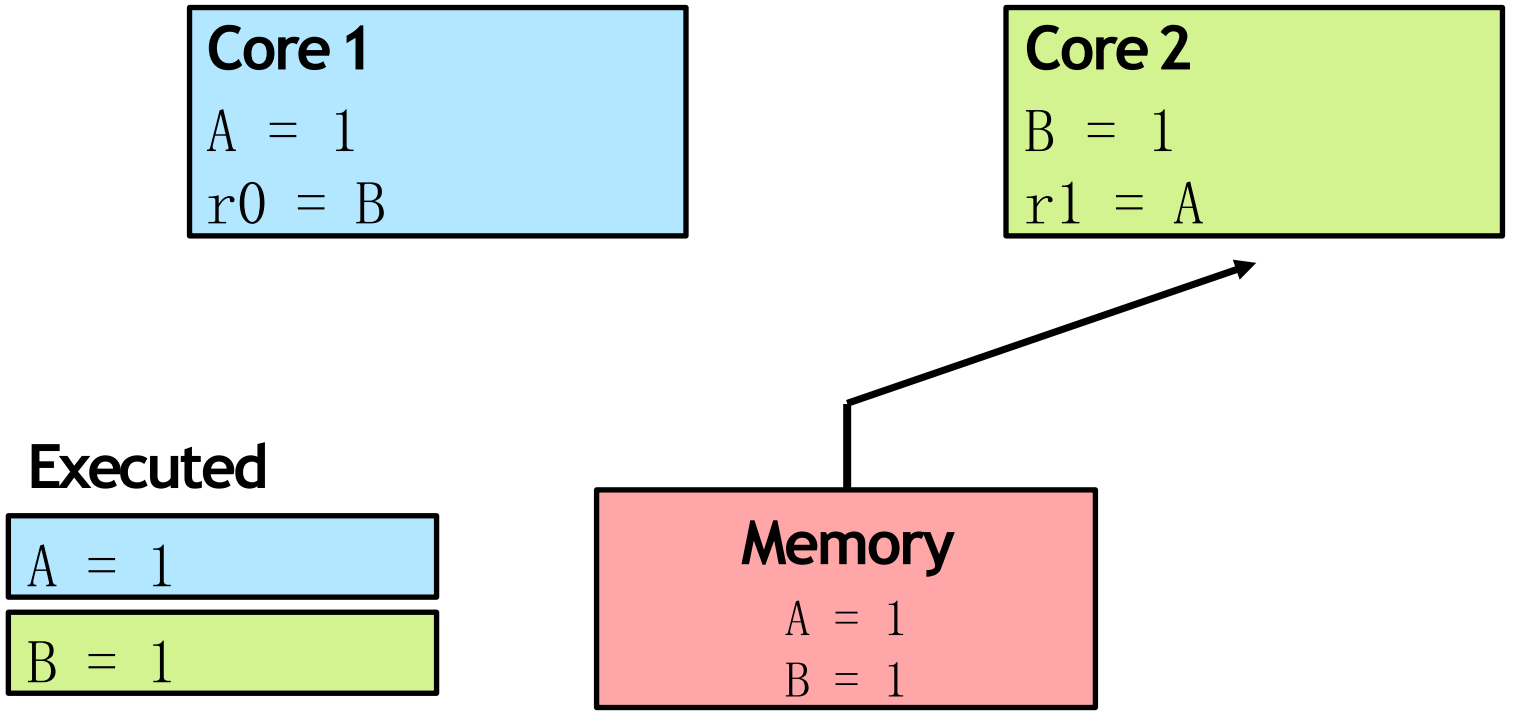
Sequential consistency

Can be seen as a "switch" running one instruction at a time



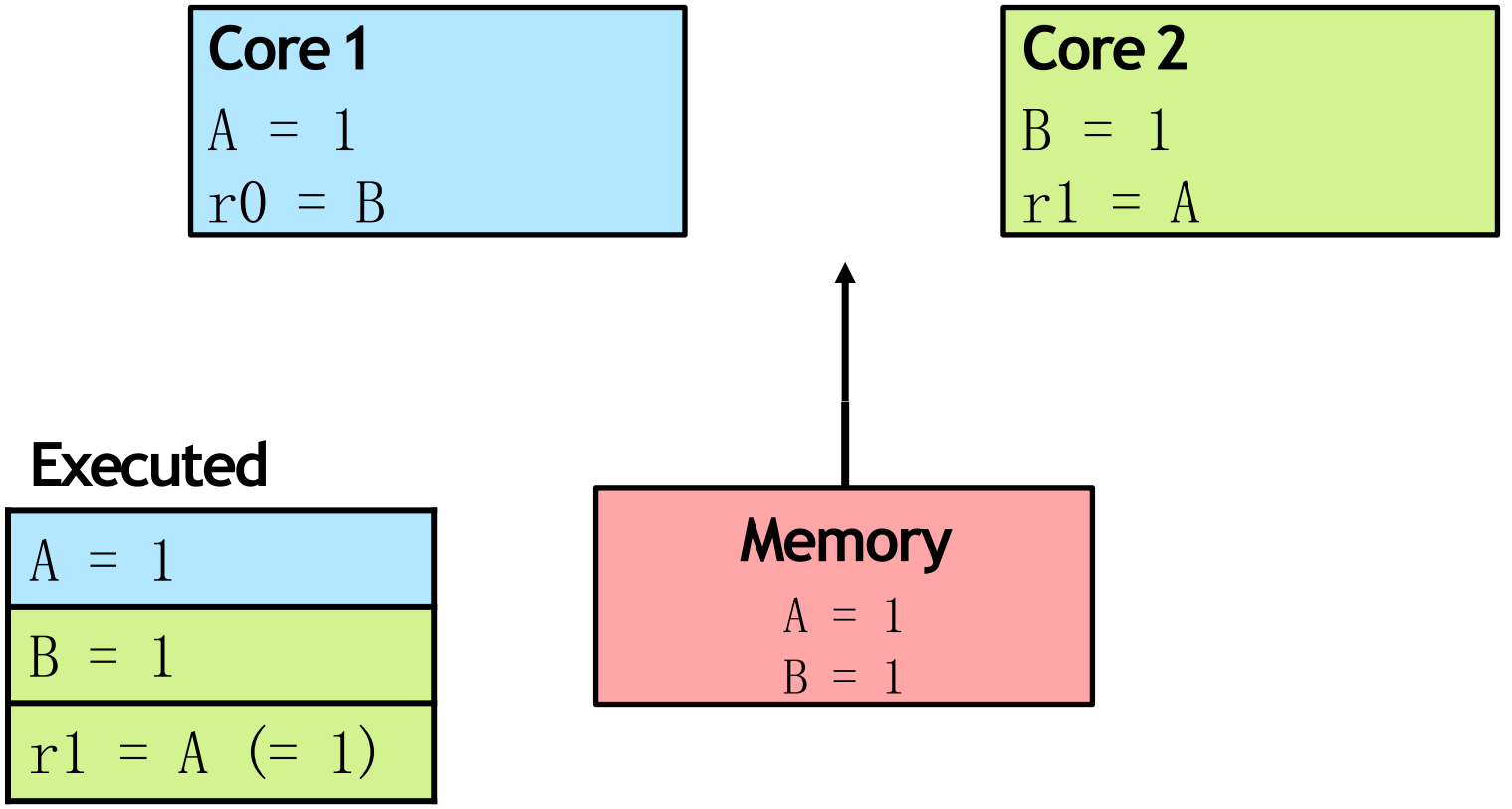
Sequential consistency

Can be seen as a "switch" running one instruction at a time



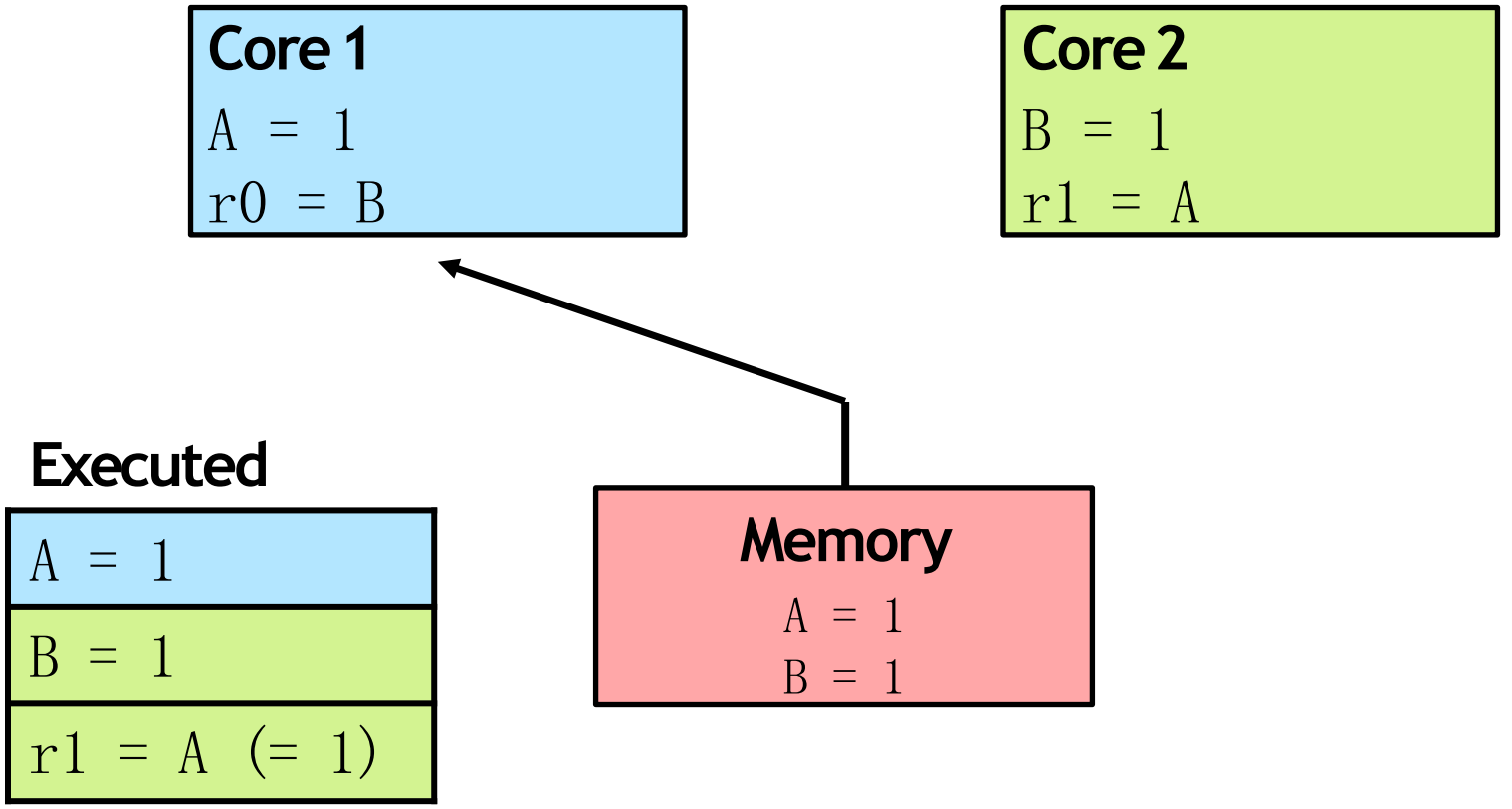
Sequential consistency

Can be seen as a "switch" running one instruction at a time



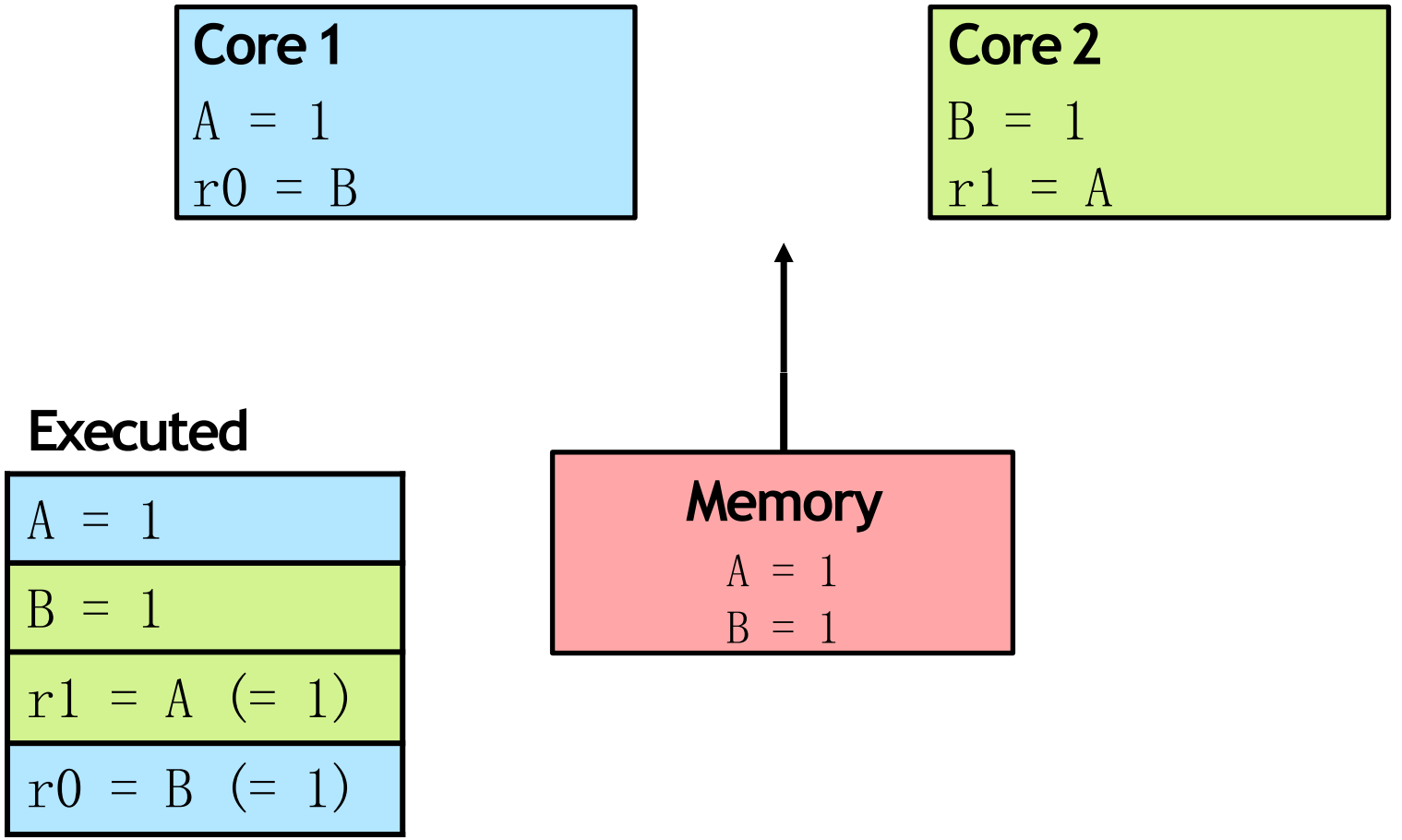
Sequential consistency

Can be seen as a "switch" running one instruction at a time



Sequential consistency

Can be seen as a "switch" running one instruction at a time



Sequential consistency

Two invariants:

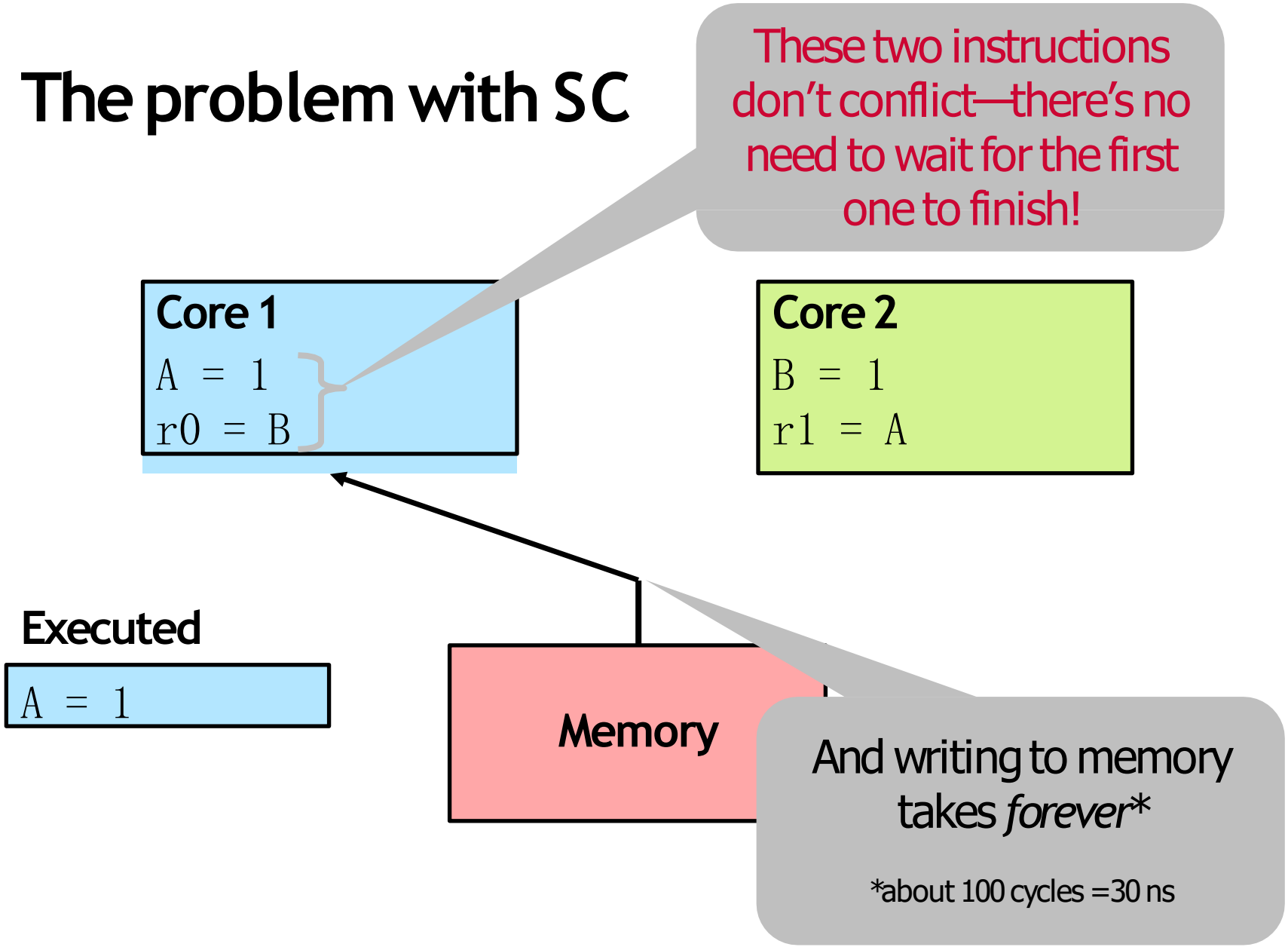
- All operations executed in some sequential order
- Each thread's operations happen in program order

Says nothing about which order all operations happen in

- Any interleaving of threads is allowed

- Due to Leslie Lamport in 1979

The problem with SC

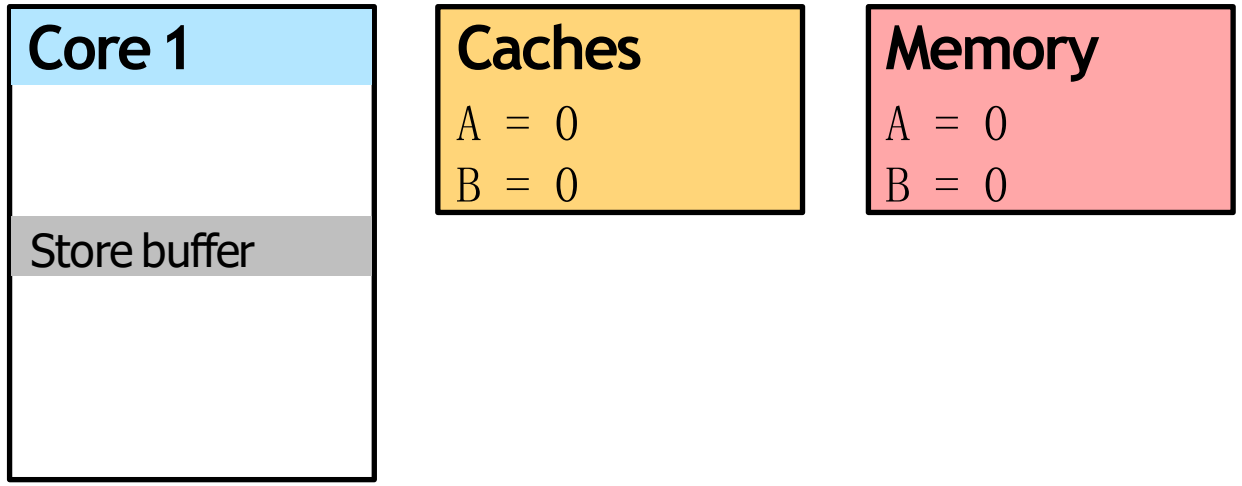


Optimization: Store buffers

- Store writes in a local buffer and then proceed to next instruction immediately
- The cache will pull writes out of the store buffer when it's ready

Thread 1

```
A = 1
r0 = B
```

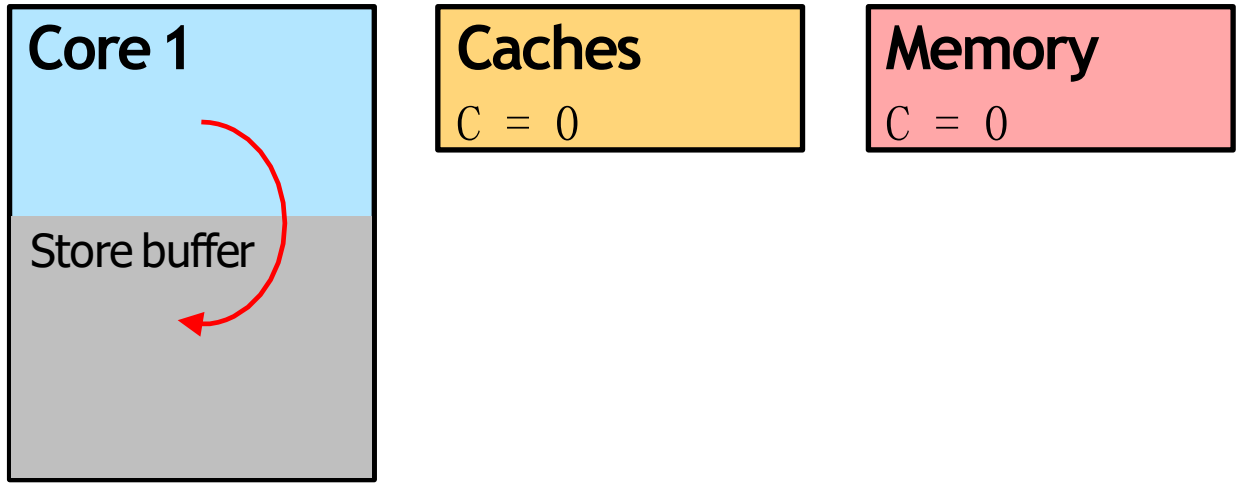


Optimization: Store buffers

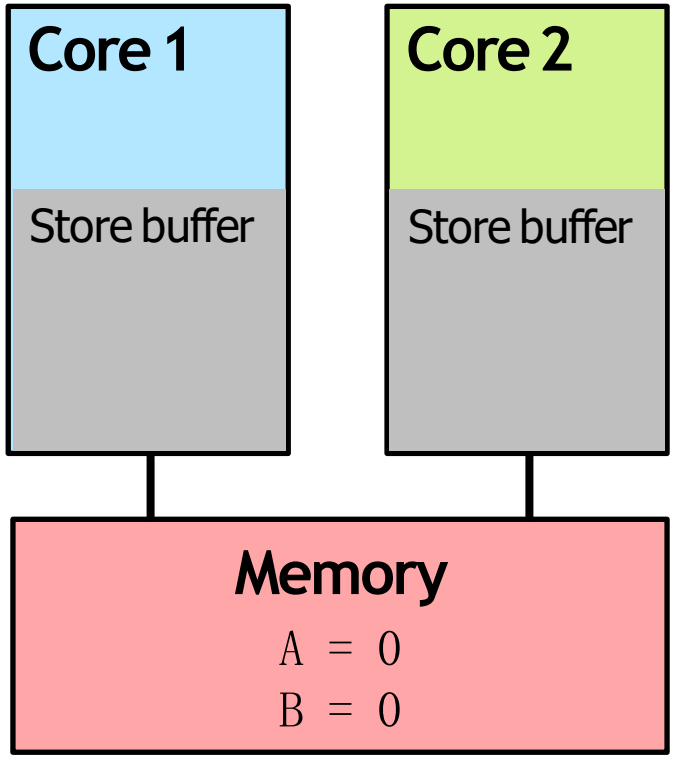
- Store writes in a local buffer and then proceed to next instruction immediately
- The cache will pull writes out of the store buffer when it's ready

Thread 1

```
C = 1
r0 = C
```



Store buffers change memory behavior

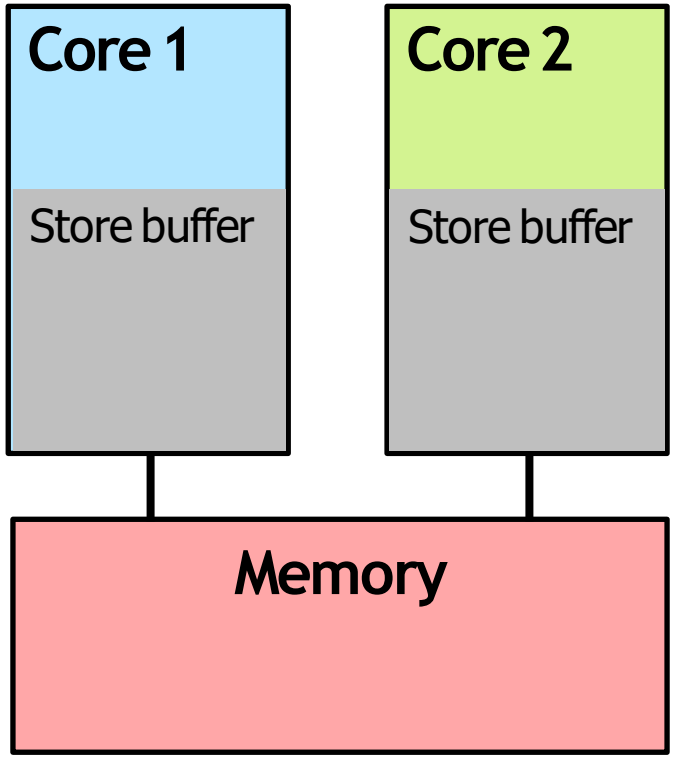


Thread 1	Thread 2
(1) A = 1	(3) B = 1
(2) r0 = B	(4) r1 = A

Can `r0 = 0` and `r1 = 0`?

SC: No!

Store buffers change memory behavior



Thread 1	Thread 2
(1) A = 1	(3) B = 1
(2) r0 = B	(4) r1 = A

Can r0 = 0 and r1 = 0?

SC: No! Store buffers: Yes!

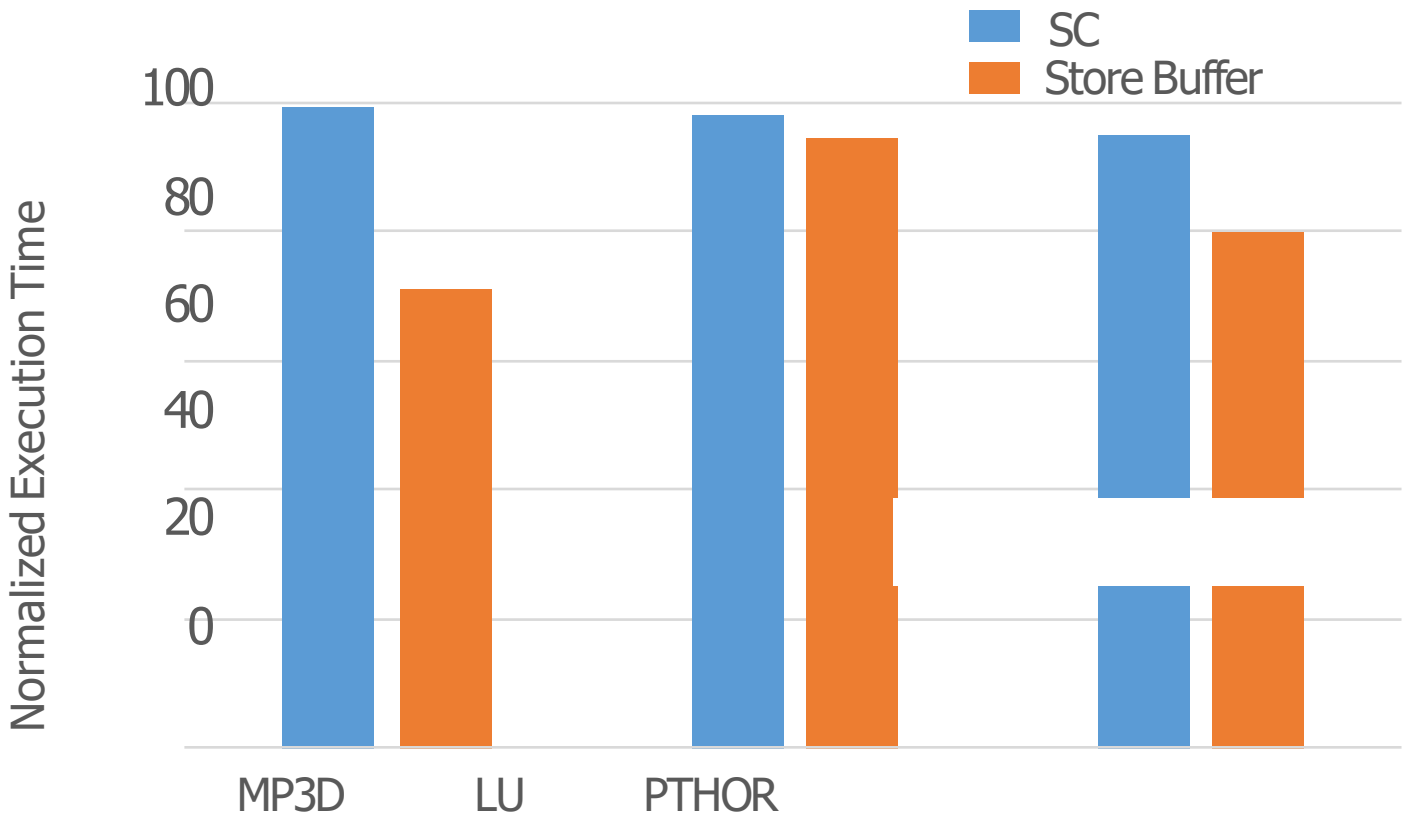
Executed

r0 = B (= 0)
r1 = A (= 0)
A = 1
B = 1

So, who uses store buffers?

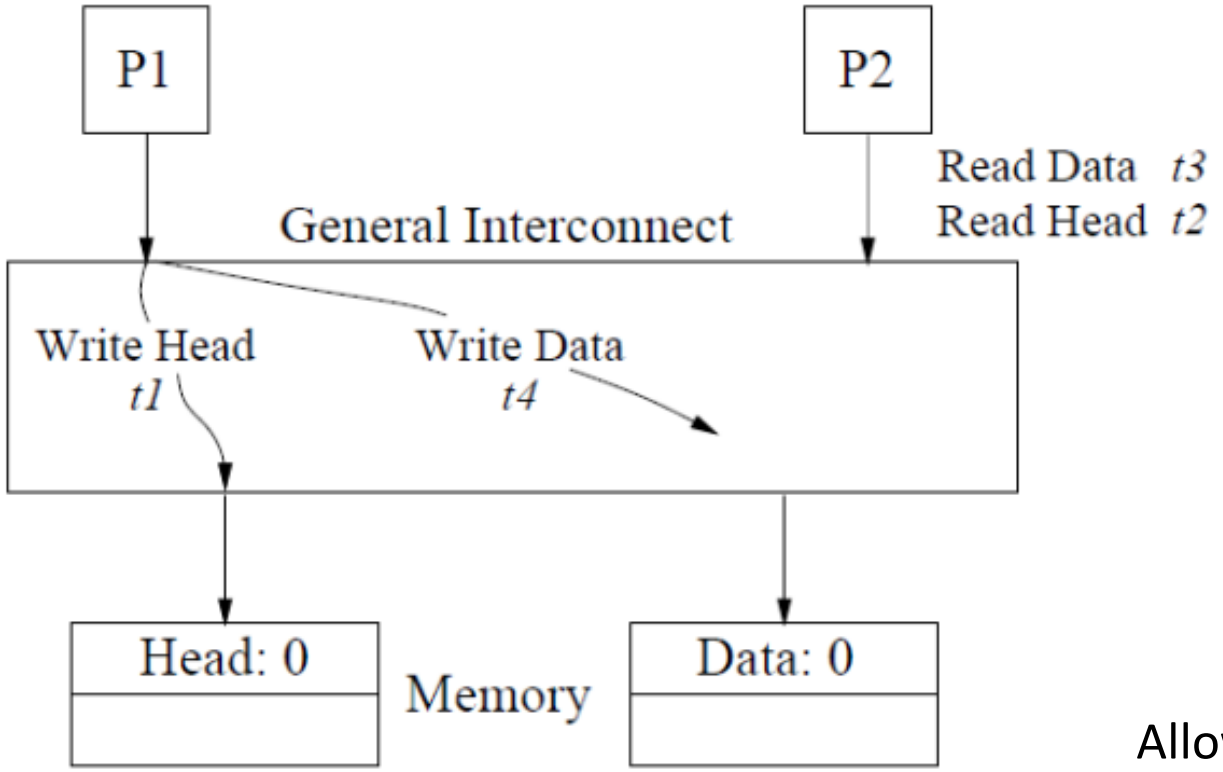
Every modern CPU!

- x86
- ARM
- PowerPC
- …



Performance evaluation of memory consistency models for shared-memory multiprocessors. Gharachorloo, Gupta, Hennessy. ASPLOS 1991.

Implementing Sequential Consistency (2)

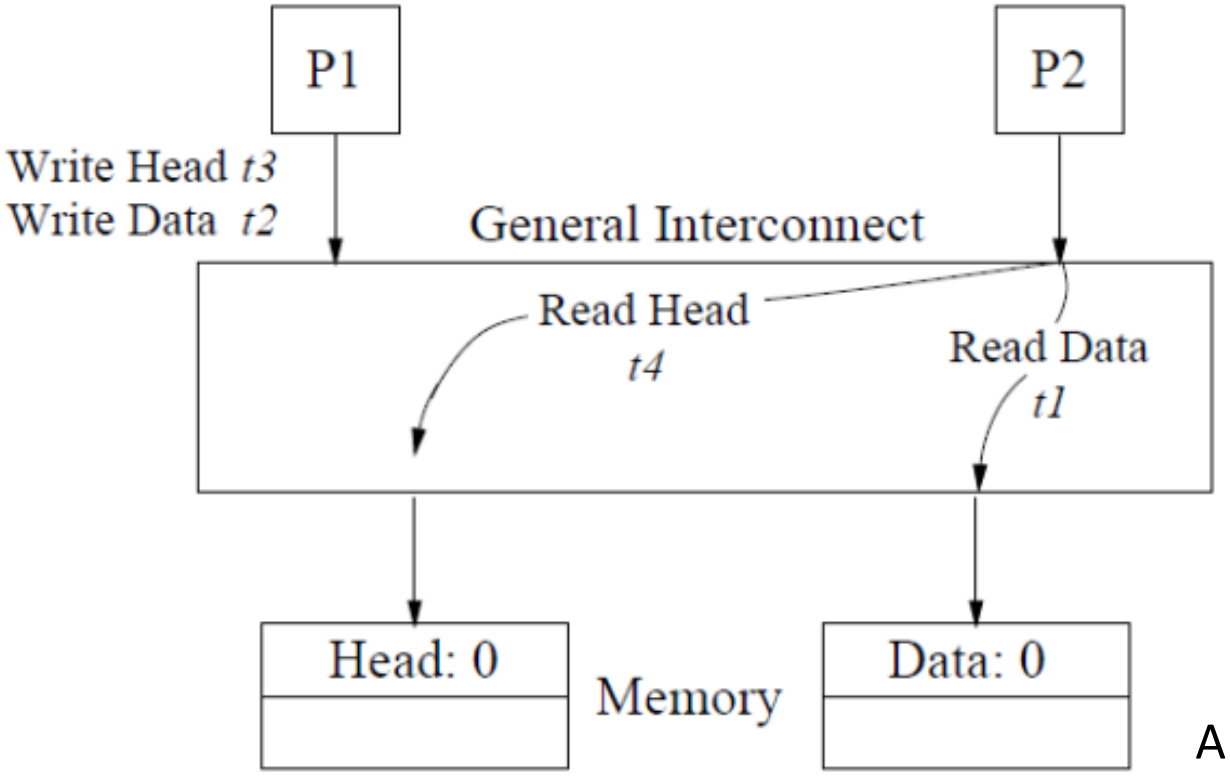


P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

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

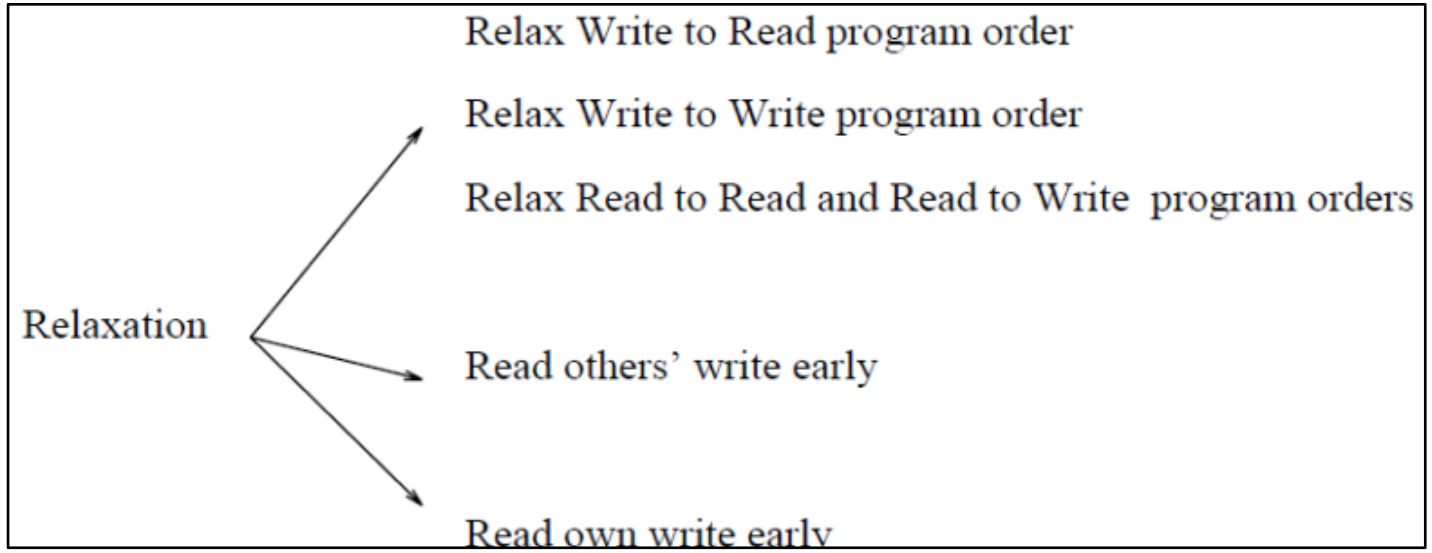
P1	P2	P3	P4
A = 1	A = 2	while (B != 1) {;}	while (B != 1) {;}
B = 1	C = 1	while (C != 1) {;}	while (C != 1) {;}
		register1 = A	register2 = A

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)

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

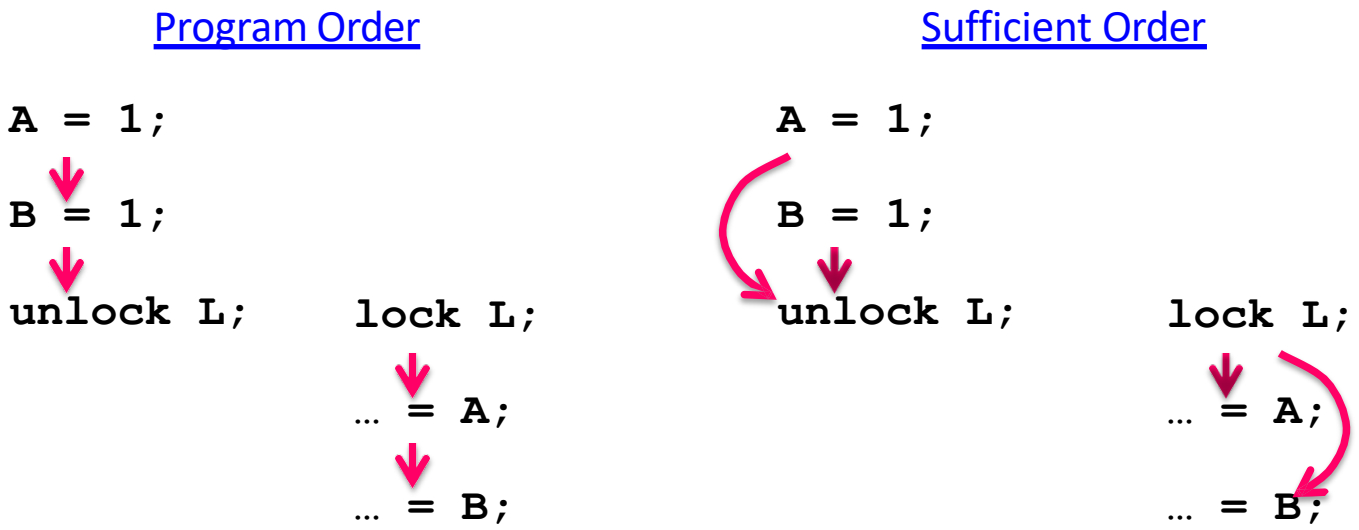
Adve and Gharachorloo, Figure 8

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**

But Can Programs Live with Weaker Memory?

- "Correctness": same results as sequential consistency
- Most programs don't require strict ordering (all of the time) for "correctness"



- But how do we know when a program will behave correctly?

Identifying Races and Synchronization

SFU

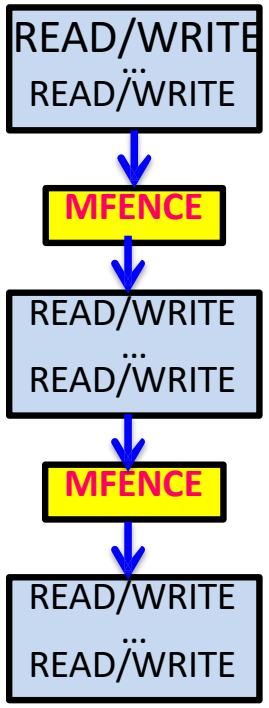
- Two accesses *conflict* if:
  - (i) access same location, and (ii) at least one is a wrrite
- Order accesses by:
  - program order (po)
  - dependence order (do): op1 --> op2
    if op2 reads op1

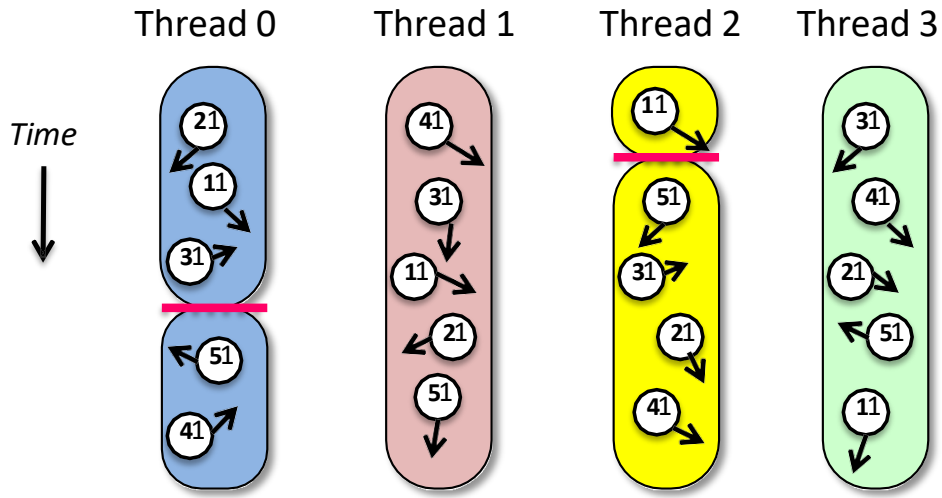
P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

- Data Race:
  - two conflicting accesses on different processors
  - not ordered by intervening accesses
- Properly Synchronized Programs:
  - all synchronizations are explicitly identified
  - all data accesses are ordered through synchroniz

MFENCE

- An **MFENCE** operation enforces the ordering seen on the previous slide:
  - does not begin until all prior reads & writes from that thread have completed
  - no subsequent read or write from that thread can start until after it finishes
- It simply stalls the thread that performs the MFENCE until write buffer empty



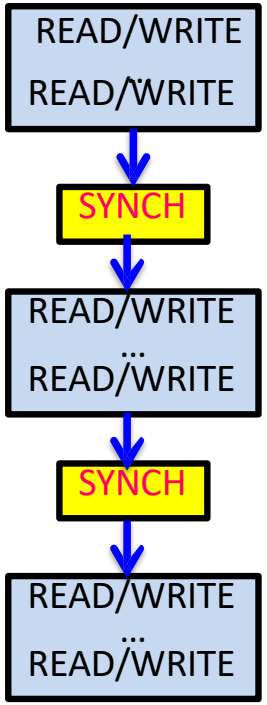


MFENCE operations create *partial orderings*

- that are observable across threads

Optimizations for Synchronized Programs

- Intuition: many parallel programs have mixtures of "private" and "public" parts*
  - the "private" parts must be protected by synchronization (e.g., locks)
  - can we take advantage of synchronization to improve performance?



Example:

Grab a lock

Insert node into data structure

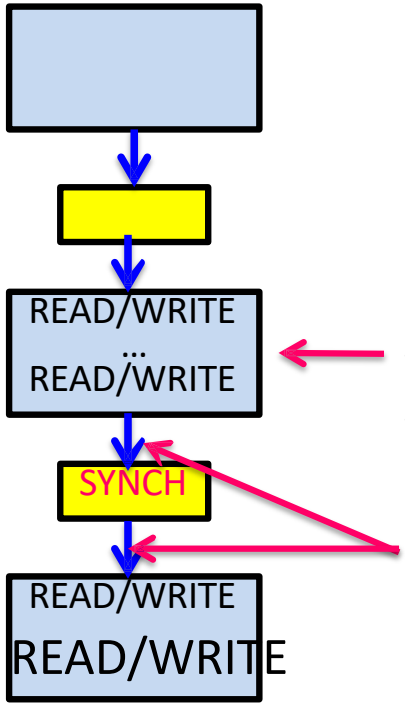
- Essentially a "private" activity; reordering is ok

Release the lock

- Now we make it "public" to the other nodes

**Caveat: shared data is in fact always visible to other threads.*

Optimizations for Synchronized Programs



Between synchronization operations:

- we can allow reordering of memory operations
- *(as long as intra-thread dependences are preserved)*

Just before and just after synchronization operations:

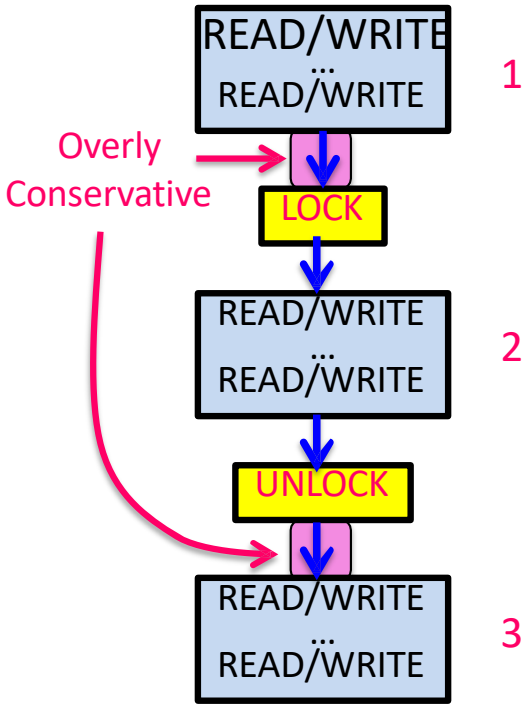
- thread must wait for all prior operations to complete

"Weak Ordering" (WO)

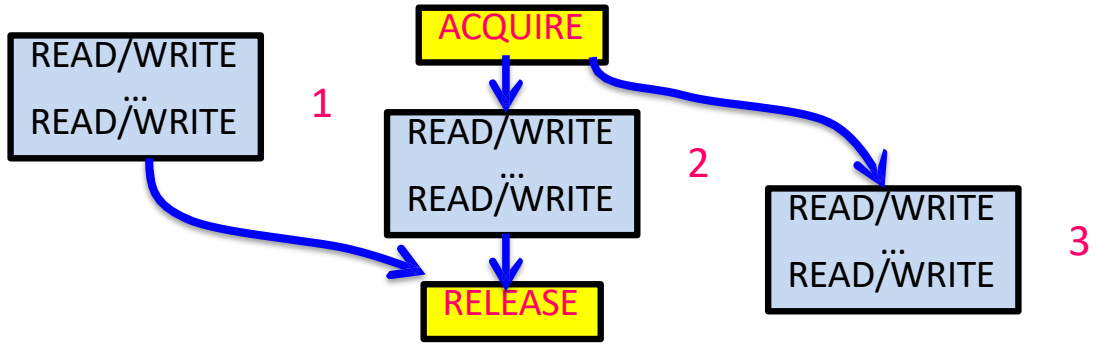
- properly synchronized programs should yield the same result as on an SC machine

Exploiting Asymmetry in Synchronization

- Lock operation: only gains ("acquires") permission to access data
- Unlock operation: only gives away ("releases") permission to access data



Weak Ordering (WO)



Release Consistency (RC)

Make sure writes completed before exit critical section

Make sure don't read/write shared state until lock acquired

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