Review

- Last Class:
  - Synchronization in multi-core.
- Today's class:
  - Memory consistency
  - Data parallelism
- Announcement and reminder

Shared Memory Example #1

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
`store 1 → y`	`store 1 → x`
`load x`	`load y`

- What value pairs can be read by the two loads?

Shared Memory Example #1: "Answer"

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)?

Shared Memory Example #2

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
store 1 → a store 1 → flag	loop: if (flag == 0) goto loop load a

- What value can be read by "load a"?

Shared Memory Example #2: "Answer"

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- What value can be read by "load a"?
- Can "load a" read the value zero?
  - Unfortunately, yes.

What is Going On?

- Reordering of memory operations to different addresses!
- **In the hardware**
  1. To tolerate write latency
     - Cores don't wait for writes to complete (via store buffers)
     - And why should they? No reason to wait with single-thread code
  2. To simplify out-of-order execution
- **In the compiler**
  3. Compilers are generally allowed to re-order memory operations to different addresses
  - Many compiler optimizations reorder memory operations.

Memory Consistency

- **Cache coherence**
  - Creates globally uniform (consistent) view of a single cache block
  - Not enough on its own:
    - What about accesses to different cache blocks?
    - Some optimizations skip coherence(!)
- **Memory consistency model**
  - Specifies the semantics of shared memory operations
  - i.e., what value(s) a load may return
- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

3 Classes of Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Typically what programmers expect**
  - 1. Processors see their own loads and stores in program order
  - 2. Processors see others' loads and stores in program order
  - 3. All processors see same global load/store ordering
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **Indistinguishable from multi-programmed uni-processor**
- **Total Store Order (TSO)** (**x86**, SPARC)
  - Allows an in-order (FIFO) store buffer
    - Load can skip the stores if accessing different addresses.
    - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (**ARM**, Itanium, **PowerPC**)
  - Allows an un-ordered coalescing store buffer
    - Stores can be put into cache in any order
  - Loads re-ordered, too.

Axiomatic vs Operational Semantics

- Two ways to understand consistency models
  - **Reorderings** allowed by the model (axiomatic)
  - **Hardware optimizations** allowed by the model (operational)
- Both understandings are correct and equivalent

TABLE 3.4: SC Ordering Rules. An "X" Denotes an Enforced Ordering.

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

from "A Primer on Memory Consistency and Cache Coherence" by Sorin, Hill and Wood

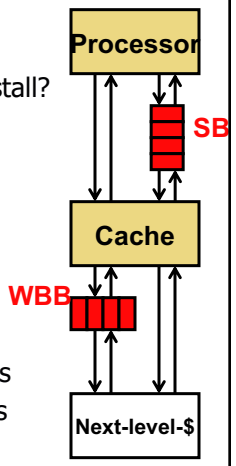
TSO (x86) Axiomatic Semantics

TABLE 4.4: TSO Ordering Rules. An "X" Denotes an Enforced Ordering. A "B" Denotes that Bypassing is Required if the Operations are to the Same Address. Entries that are Different from the SC Ordering Rules are Shaded and Shown in Bold.

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	FENCE	**X**	**X**	**X**	**X**

Write Misses and Store Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?
- Store buffer: a small buffer for store misses
  - Stores put address/value into SB, keep going
  - SB writes to D$ in the background
  - Loads must search SB (in addition to D$)
  - (mostly) eliminates stalls on write misses
- Store buffer vs. writeback buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Why? To Hide Store Miss Latency

- Why? Why Allow Such Odd Behavior?
  - Reason #1: hiding store miss latency
- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Store buffer
- It complicates multiprocessors
  - By allowing reordering of store and load (to different addresses)
- Example:

thread 1	thread 2
`store 1 → y`	`store 1 → x`
`load x`	`load y`

  - Both stores miss cache, are put in store buffer
  - Loads hit, receive value before store completes, see "old" values

Shared Memory Example #1: Answer

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0,y=0)? Yes! (for x86, SPARC, ARM, PowerPC)

Release Consistency Axiomatic Semantics

TABLE 5.5: XC Ordering Rules. An "X" Denotes an Enforced Ordering. An "A" Denotes an Ordering that is Enforced Only if the Operations are to the Same Address. A "B" Denotes that Bypassing is Required if the Operations are to the Same Address. Entries Different from TSO are Shaded and Indicated in Bold Font.

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	FENCE	X	X	X	X

Why? Simplify Out-of-Order Execution

- Why? Why Allow Such Odd Behavior?
  - simplifying out-of-order execution
- One key benefit of out-of-order execution:
  - Out-of-order execution of loads to (same or different) addresses

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- No problem if cache is coherent and SC or TSO is maintained
- Aside: some store buffers reorder stores by same thread to different addresses (as in thread 1 above) in RC model.

Why? Allow Compiler Optimizations

- Why? Why Allow Such Odd Behavior?
  - allow compiler optimizations
- Compiler optimizations are important
  - Consider a case of loop-invariant code motion:

original code

```
for (i=0; i<10; i++)
  array[i] = array2[i] + x^2;
```

optimized code

```
tmp1 = x^2;
for (i=0; i<10; i++)
  array[i] = array2[i] + tmp1;
```

- Optimized code is much faster, but loads of x have been reordered.

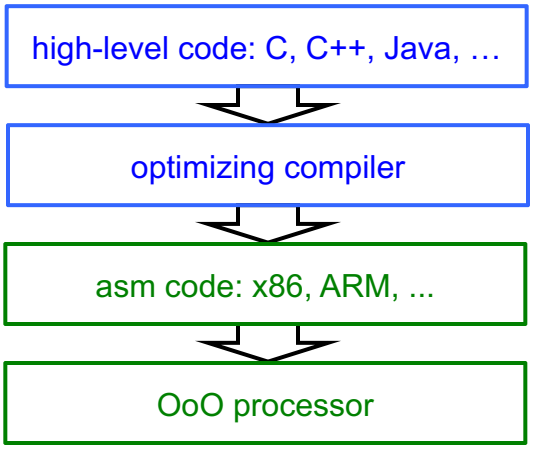
Shared Memory Example #2: Answer

- **Initially: all variables zero** (flag == a == 0)

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- What value can be read by "load a"?
  - "load a" can see the value "1"
- Can "load a" read the value zero? (same as last slide)
  - **Yes! (for ARM, PowerPC, Itanium, and Alpha)**
  - **No! (for Intel/AMD x86, Sun SPARC, IBM 370)**
    - Assuming the compiler didn't reorder anything…

Consistency Models: A Layered Cake



- How do we prevent our code from getting screwed up?
  - by the compiler and/or the hw?
- We adhere to the **language's consistency model**
  - compiler writer ensures code is correct on each hw architecture

Recap: Four Shared Memory Issues

1. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache "show up" in others?
2. **Synchronization**
   - How to regulate access to shared data?
   - How to implement "locks"?
3. **Memory consistency models**
   - How to keep programmer sane while letting hw/compiler optimize?
4. **Parallel programming**
   - How does the programmer express the parallelism?
   - To be continued