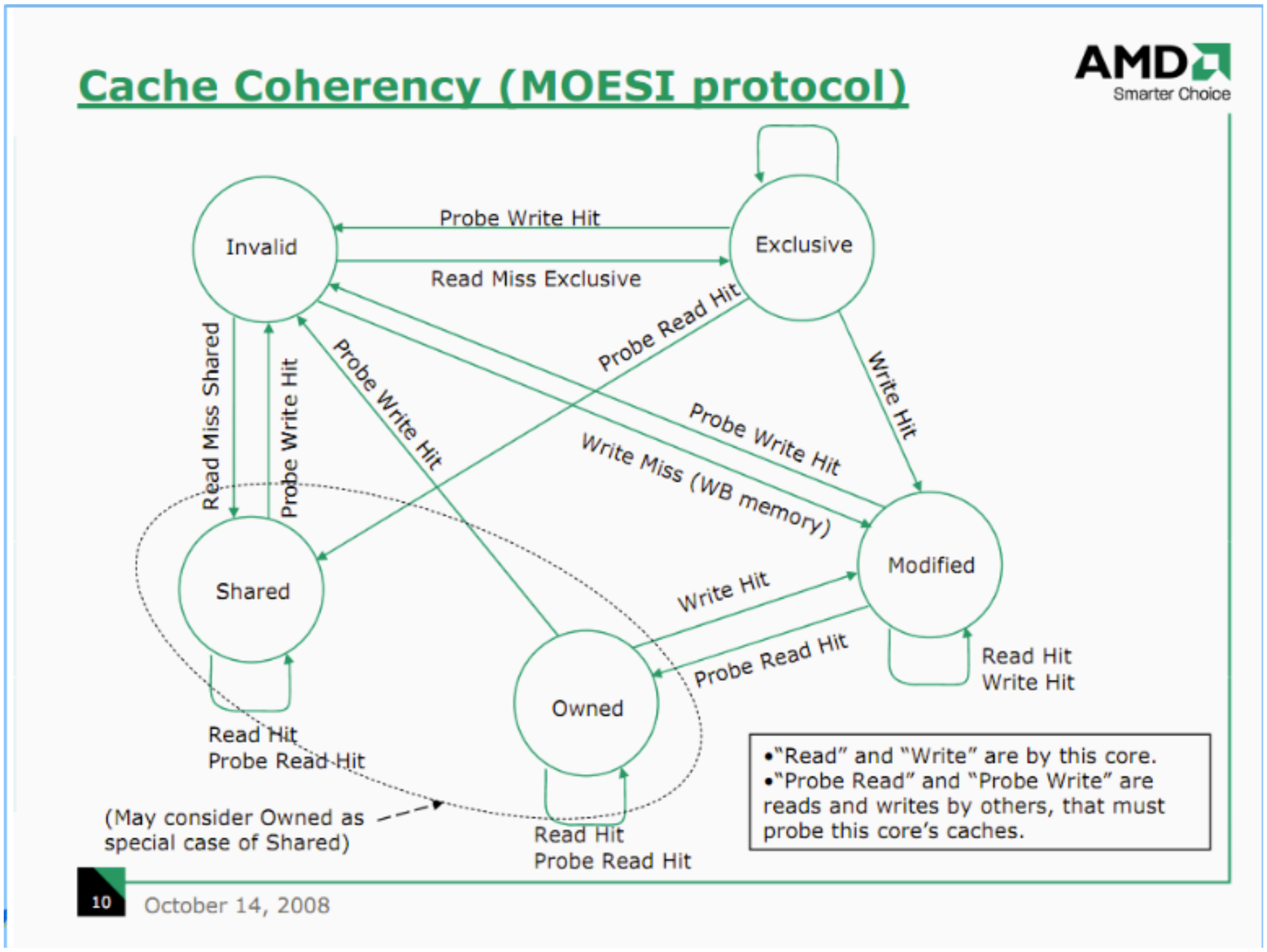
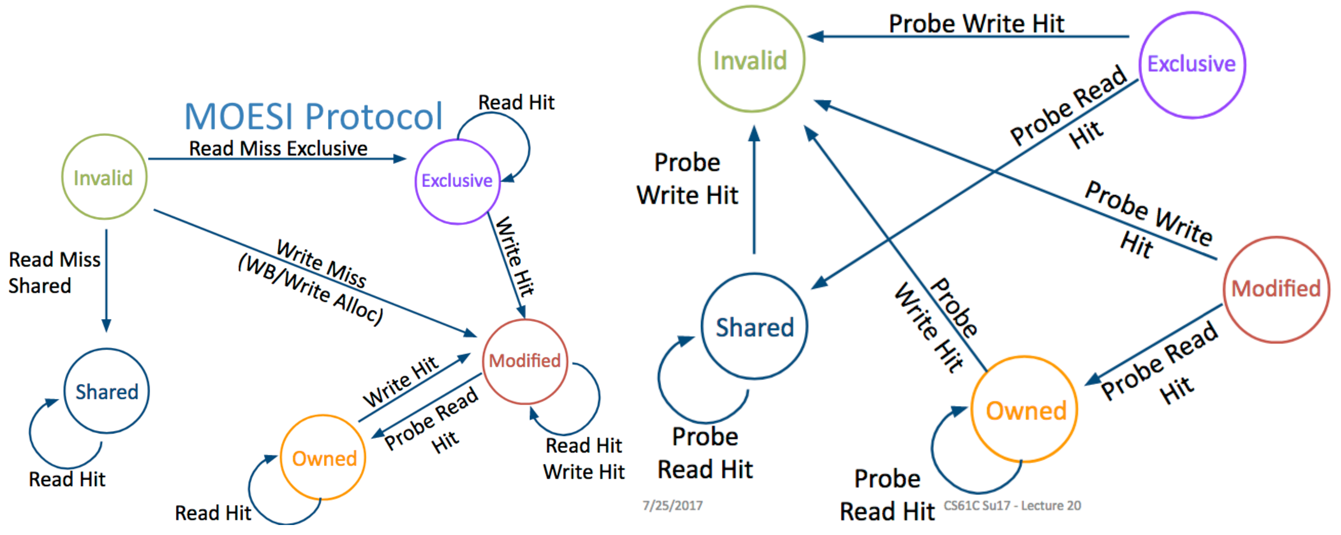
CS 61C Summer 2017– Cache Coherency

MOESI Cache Coherency





With the MOESI concurrency protocol implemented, accesses to cache accesses appear *serializiable*. This means that the result of the parallel cache accesses appear the same as if there were done in serial from one processor in some ordering.

State	Cache up to date?	Memory up to date?	Others have a copy?	Can write without changing state?
Modified	Yes	No	No	Yes
Owned	Yes	No	Yes	No
Exclusive	Yes	Yes	No	No
Shared	Yes	Maybe	Yes	No
Invalid	No	Maybe	Maybe	No

1. Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with one cache block and a two cache block memory. Assume the MOESI protocol is used, with write- back caches, write-allocate, and invalidation of other caches on write (instead of updating the value in the other caches).

Time	After Operation	P1 cache state	P2 cache state	Memory @ 0 up to date?	Memory @ 1 up to date?
0	P1: read block 1	Exclusive (1)	Invalid	YES	YES
1	P2: read block 1	Shared (1)	Shared (1)	YES	YES
2	P1: write block 1	Modified (1)	Invalid	YES	NO
3	P2: write block 1	Invalid	Modified (1)	YES	NO
4	P1: read block 0	Exclusive (0)	Modified (1)	YES	NO
5	P2: read block 0	Shared (0)	Shared (0)	YES	YES
6	P1: write block 0	Modified (0)	Invalid	NO	YES
7	P2: read block 0	Owned (0)	Shared (0)	NO	YES
8	P2: write block 0	Invalid	Modified (0)	NO	YES
9	P1: read block 0	Shared (0)	Owned (0)	NO	YES

2. What is the advantage of MOESI over MESI? (Hint: notice a key difference between MOESI and MESI, what state does MOESI have that MESI doesn't and how might that state be advantageous?)
The MOESI protocol has two main advantages over MESI:
1. The Owned state allows for snooping, that is, directly sharing its content (which is not up to date with main memory) with other caches. For other caches reading from an Owned state is oftern faster than reading from main memory.
2. In theory, we can implement MOESI such that if a cache block has an Owner and multiple Shared states, a write to the Owned state and be reflected in other caches by having them update. In MESI, there is no Owned state and a Modified state cannot be Shared, so a write to an Shared state would invalidate all Shared caches, leading to further cache misses. (A Shared state need not be up-to-date with memory in MOESI, but it must be in MESI).

Concurrency

1. Consider the following function:

```
void transferFunds(struct account *from,
                   struct account *to,
                   long cents)
  {
    from->cents -= cents;
    to->cents += cents;
  }
```

   a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: if the problem isn't obvious, translate the function into MIPS first)

      Each thread needs to read the "current" value, perform an add/sub, and store a value for from->cents and to->cents. Two threads could read the same "current" value and the later store essentially erases the other transaction at either line.

   b. How could you fix or avoid these races? Can you do this without hardware support?

      Wrap transferFunds in a critical section, or divide up the accounts array and for loop in a way that you can have separate threads work on different accounts. You can also create an atomic section for any parts of the code that may have data races.

3. Data race and Atomic operations.

The benefits of multi-threading programming come only after you understand concurrency. Here are two most common concurrency issues:

- **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. The can often be solved by bypassing cache and writing directly to memory, i.e. using volatile keyword in many languages.
- The famous **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-thread programming, the **interleaving** of R,M,W stages often produces a lot of issues.

To solve problem with Read-modify-write, we have to rely on the idea of **undisrupted execution**.

In RISC-V, we have two categories of atomic instructions:
- Load-reserve, store-conditional (undisrupted execution across multiple instructions)
- Amo.swap (single, undisrupted memory operation) and other amo operations.

Both can be used to achieve atomic primitives, here are two examples.

Test-and-set

```
Start: addi           t0 x0 1 #locked state is 1
       amoswap.w.aq t1 t0 (a0)
       bne            t1 x0 start #if the lock is not
                                  free, retry

       … #critical section

       amoswap.w.rl  x0 x0 a0#release lock
```

Compare-and-swap

```
#expect old value in a1, desired new value in a2
Start: lr.w    a3 (a0)
       bne     a3 a1  fail #CAS fail
       sc.w    a3 a2 (a0)
       bnez    a3 error #store unsuccessful
        … #critical section
        amoswap.w.rl  x0 x0 a0
fail: #failed CAS
```

Instruction semantics:

- lr: Loads the four bytes from memory at address x[rs1], writes them to x[rd], sign-extending the result, and registers a reservation on that memory word.
- sc: Stores the four bytes in register x[rs2] to memory at address x[rs1], provided there exists a load reservation on that memory address. Writes 0 to x[rd] if the store succeeded, or a nonzero error code otherwise.
- Amoswap: Atomically, let t be the value of the memory word at address x[rs1], then set that memory word to x[rs2]. Set x[rd] to the sign extension of t.

Question: why do we need special instructions for these operations? Why can’t we use normal load and store for lr and sc? Why can’t we expand amoswap to a normal load and store?

Answer: For lr and sc, after lr, other threads cannot write to the location marked reserve, hence the value loaded from memory (a3 in the above example) will be unchanged between lr and sc.
For amoswap, it does load and store in one single CPU cycle, hence the operation is atomic and undisruptable.