Cache Coherency

MOESI:

State	*Cache up to date?*	*Memory up to date?*	*Others have copy?*	*Can respond to other's reads?*	*Can write without changing state?*
Modified	YES	NO	NO	YES, REQUIRED	YES
Owned	YES	NO	MAYBE	YES, REQUIRED	YES
Exclusive	YES	YES	NO	YES, OPTIONAL	NO
Shared	YES	MAYBE	MAYBE	NO	NO
Invalid	NO	MAYBE	MAYBE	NO	NO

With the MOESI coherency protocol implemented, accesses to cache accesses have sequential access consistency. This means that the result of parallel cache accesses appear as if they were done in serial from one processor in some ordering and this ordering is consistent with the ordering to which they are issued to each cache.

1. Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache. Assume the MOESI protocol is used, with write-back caches and invalidation of other caches on write (instead of updating the value in the other caches). Assume all reads and writes are for entire cache blocks.

Time	*After Operation*	*P1 cache state @ 0*	*P2 cache state @ 0*	*Memory @ 0 up to date?*
0	P2: read block 0	Shared	Shared	YES
1	P1: write block 0	Modified	Invalid	NO
2	P2: read block 0	Owned	Shared	NO
3	P2: write block 0	Invalid	Modified	NO
4	P1: read block 0	Shared	Owned	NO
5	P2: read block 0	Shared	Owned	NO

Synchronization

Consider the following function:

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

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

The from->cents -= cents is going to have to load the old value of from->cents first, then compute the new value, then store the new value. After the load, some other thread could load the old value again and compute a different new value. One of the two new values will be the last value written to from->cents, so one of the updates will be lost.

Let's look at two approachs to avoiding these races. Here are MIPS instructions for synchronization:

- `ll rt, immed(rs)` ("load linked") — rt ← Memory[rs+immed]
- `sc rt, immed(rs)` ("store conditional") —

  if no writes to Memory[rs+immed] since ll:

  Memory[rs+immed] ← rt; rt ← 1

  otherwise:

  rt ← 0

1. Write an ***atomic*** increment function in MIPS assembly which increments the value at a memory address by a given amount.

```
# $a0 = memory location to increment
# $a1 = how much to increment by
atomic_increment:
        ll   $t0 0($a0)
        addu $t0 $t0 $a1
        sc   $t0 0($a0)
        beq  $t0 0 atomic_increment
        jr   $ra
```

2. Finish this implementation of a ***spinlock,*** which can be used to exclude multiple threads from running a section of code at once. The lock value is 0 when unlocked and 1 if locked.

```
spin_unlock:
        sw   $0 0($a0)        # reset to 0
        jr   $ra
spin_lock:
        ll   $t0 0($a0)
        beq  $t0 1 spin_lock
        li   $t0 1
        sc   $t0 0($a0)
        beq  $t0 0 spin_lock
        jr   $ra
```