Project 4: Multicore Cache Coherence (10%)

ENEE 646: Digital Computer Design, Fall 2020

Assigned: Tuesday, Dec 1; Due: Saturday, Dec 19

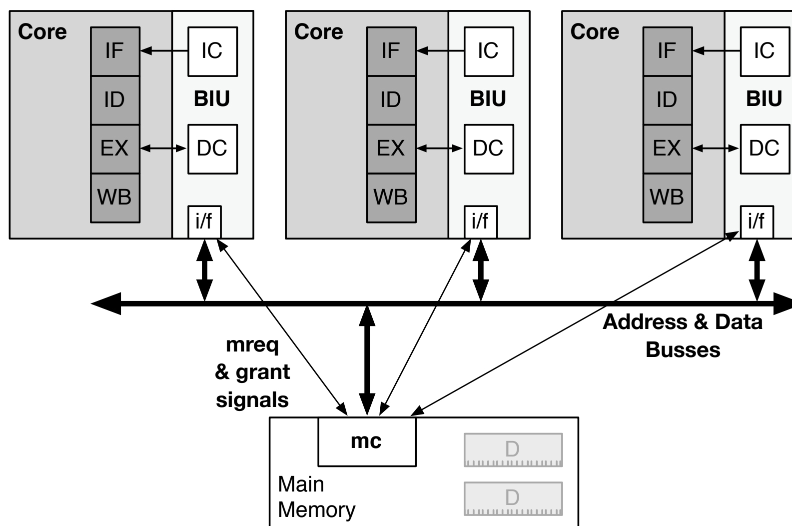
Purpose

This project is intended to help you understand the issues of writing software for multicore systems. You will find that having multiple cores (and therefore multiple program counters) is only half the problem; the real issue is the cache system. Your job is to get a multicore cache system up and running (the cache, the cache controller, the bus interface unit, and the memory controller have all been developed for you; you need only integrate them into your P2/P3 pipeline), and then add a cache-coherence engine of your choosing, so that the example multithreaded software given to you will run correctly.

Project Overview

The figure below indicates what you will be working with in this project. On the project website is code that will give your pipelined processors a split L1 Instruction/Data Cache (only one port to the Data Cache, as before, with the limitation that memory ops must be in the right-hand atom of an instruction), a separate Bus Interface Unit to the main memory, and a memory controller. This is similar to what you were given in Project 3, except that the internals of the Bus Interface Unit should be much easier to understand this time around. Each BIU is connected to the memory controller through two sets of wires: first, there is a shared ***address bus*** and ***data bus***, and the shared nature of the busses is important because it means that any core can see what every other core is requesting of main memory. Secondly, each core has a separate ***req/grant*** interface, so that when a core wants access to main memory, it sets its ***req*** bus to a valid signal, and at some point, the memory controller will ***grant*** that core permission to use the shared address & data busses.

For Project 4, your job is to build a cache-coherence engine for this system, which amounts to a modification of the BIU to implement some form of cache coherence. You can choose whatever type of coherence you want (e.g., SI, MSI, MESI, MOESI, etc.). You have been given two test harnesses to



experiment with. The first, ***test1.v***, instantiates one RiSC32 core and connects it to the main-memory system; the second, ***test3.v***, instantiates three RiSC32 core and connects all of them to the main-memory system. The ***test3.v*** test harness also sets the CPU-ID for each core in the system to a different value, so each core has a unique ID {1..3}.

Project Details

The following assembly code reverses an array in-place:

```
#
# reverse-1.s - array reverse for a single thread
#
# reverses a 2N-entry array in place
#
# register usage:
#       r1      - loop index
#       r2      - address of first item in array (base address)
#       r3      - address of last item in array
#       r4,r5   - temps for LW/SW (exchange)
#       r11     - address of i entry
#       r12     - address of n-i entry
#       r13     - N-1
#       r14     - N
#

        movi    r14, 6

        movi    r2, array
        addi    r13, r14, -1    | add   r3, r2, r14
        add     r1, r0, r13     | add   r3, r3, r13

loop:   add     r11, r2, r1     | sub   r12, r3, r1
        nop                     | lw    r4, r11, 0
        nop                     | lw    r5, r12, 0
        nop                     | sw    r4, r12, 0
        addi     r1, r1, -1     | sw    r5, r11, 0
        nop                     | blz   r1, next
        bne     r0, r14, loop

# at this point the array should be backwards in the cache
# now load a bunch of garbage into the cache to force writebacks

next:   movi    r2, pad
        movi    r3, 32
        movi    r5, 8
        add     r1, r0, r0
flush:  add     r4, r2, r1
        add     r1, r1, r5      | lw r0, r4, 0
        bne     r1, r3, flush   | lw r0, r4, 4

        halt
        nop
        nop
        nop
        nop

array:  .mfill  12, 1, 1

pad:    .mfill  32, 0xabcdef
```

The first loop (in the middle of the code) does the work of reversing the array, and then the loop at the bottom forces writebacks of all the data produced by the first loop, by loading a bunch of data that conflicts in the data cache. You can run this program on any of your processor designs, and it should work. Moreover, you should be able to see in the main-memory output that it works. The code starts in the middle of the array. On each iteration, the loop loads the two values to be swapped, stores them back in opposite places, and then decrements the loop index by one and repeats.

Here is the problem. The following is a multi-threaded version of the exact same code, but it fails to work correctly, and it is not the fault of the software. Try running it on your P1 or P2 solutions (which do not have caches).

```
#
# reverse-3.s - array reverse for three threads
#
# reverses a 2N-entry array in place (N divisible by # threads)
#
# register usage:
#       r1      - loop index
#       r2      - address of first item in array (base address)
#       r3      - address of last item in array
#       r4,r5   - temps for LW/SW (exchange)
#       r11     - address of i entry
#       r12     - address of n-i entry
#       r13     - N-1
#       r14     - N
#       r15     - cpuID (1..3)
#

        movi    r14, 6

        movi    r2, array
        addi    r13, r14, -1    | add  r3, r2, r14
        add     r1, r0, r14     | add  r3, r3, r13
        sub     r1, r1, r15

loop:   add     r11, r2, r1     | sub  r12, r3, r1
        nop                     | lw   r4, r11, 0
        nop                     | lw   r5, r12, 0
        nop                     | sw   r4, r12, 0
        addi     r1, r1, -3     | sw   r5, r11, 0
        nop                     | blz  r1, next
        bne     r0, r14, loop

# at this point the array should be backwards in the cache
# now load a bunch of garbage into the cache to force writebacks

next:   movi    r2, pad
        movi    r3, 32
        movi    r5, 8
        add     r1, r0, r0
flush:  add     r4, r2, r1
        add     r1, r1, r5      | lw r0, r4, 0
        bne     r1, r3, flush   | lw r0, r4, 4

        halt
        nop
        nop
        nop

array:  .mfill  12, 1, 1

pad:    .mfill  32, 0xabcdef
```

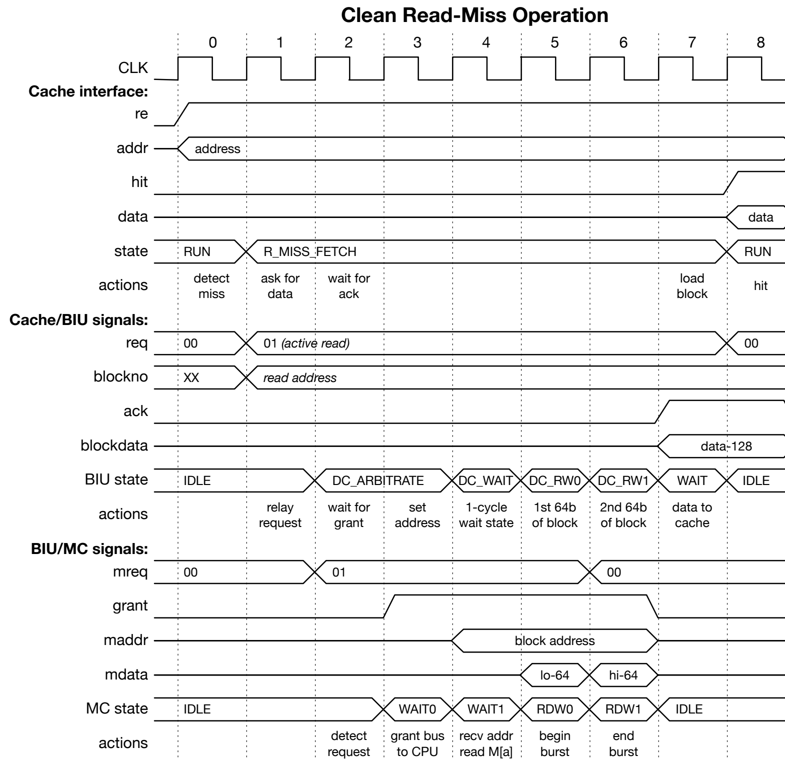
The main difference is that the array is divided into thirds, and each CPU uses its CPU-ID to determine which third of the array it gets. The array values are interleaved such that a particular thread on a CPU swaps two values and then decrements by *three* places instead of *one*, as in the previous code. When this runs, the main memory values get all messed up because there is false sharing going on; by design, none of the threads will overwrite each other's data, but because the data is closely packed, two different threads will write to different addresses that happen to lie in the same cache block, and this is what causes main memory to get out of sync.

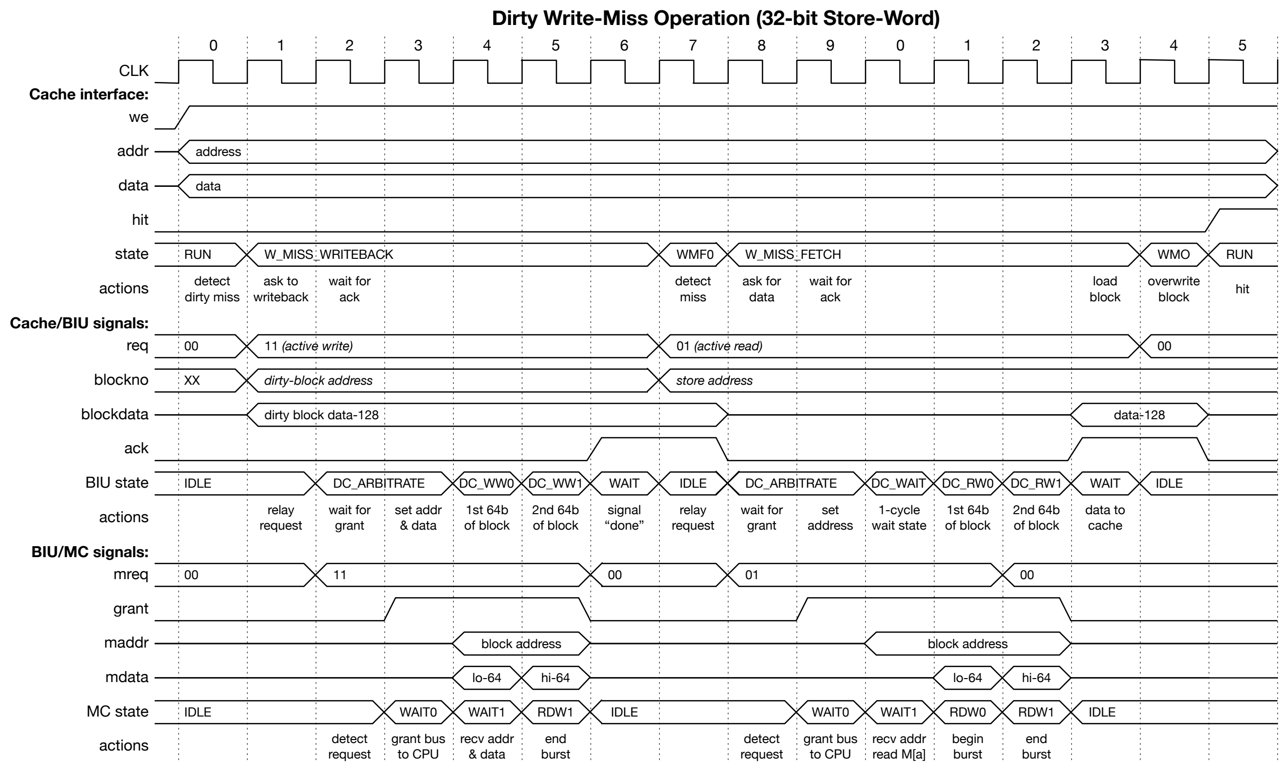
Yes, the simplest solution is not to write code this way. However, this code presents a very simple, if contrived, multicore/multi-threaded example that fits on a single page and is thus easily analyzed, and that can be solved with a cache-coherence engine.

Cache Implementation & Limitations

The cache that has been provided to you is extremely simple. It has 8 blocks and a single port. Each block is the size of one 128-bit (4-word) vectors. It is *write-back* and *allocate-on-write-miss*.

The *Bus Interface Unit (cache_and_BIU)* instantiates both an Instruction Cache and a Data Cache, so it can satisfy two simultaneous requests from the pipeline: one from instruction fetch, and one from a data operation in the execute stage.





Additional limitation:

1. The *Cache* cannot handle two data requests at the same time, thus instructions of the form

```
lw r2, r3, 4    | lw r5, r6, 7
```

are not allowed, unless you modify your system to handle it. Additionally, the memory operations must be on the right-hand side, e.g. like this:

```
nop             | lw r5, r6, 7
```

You should check out the example code above.

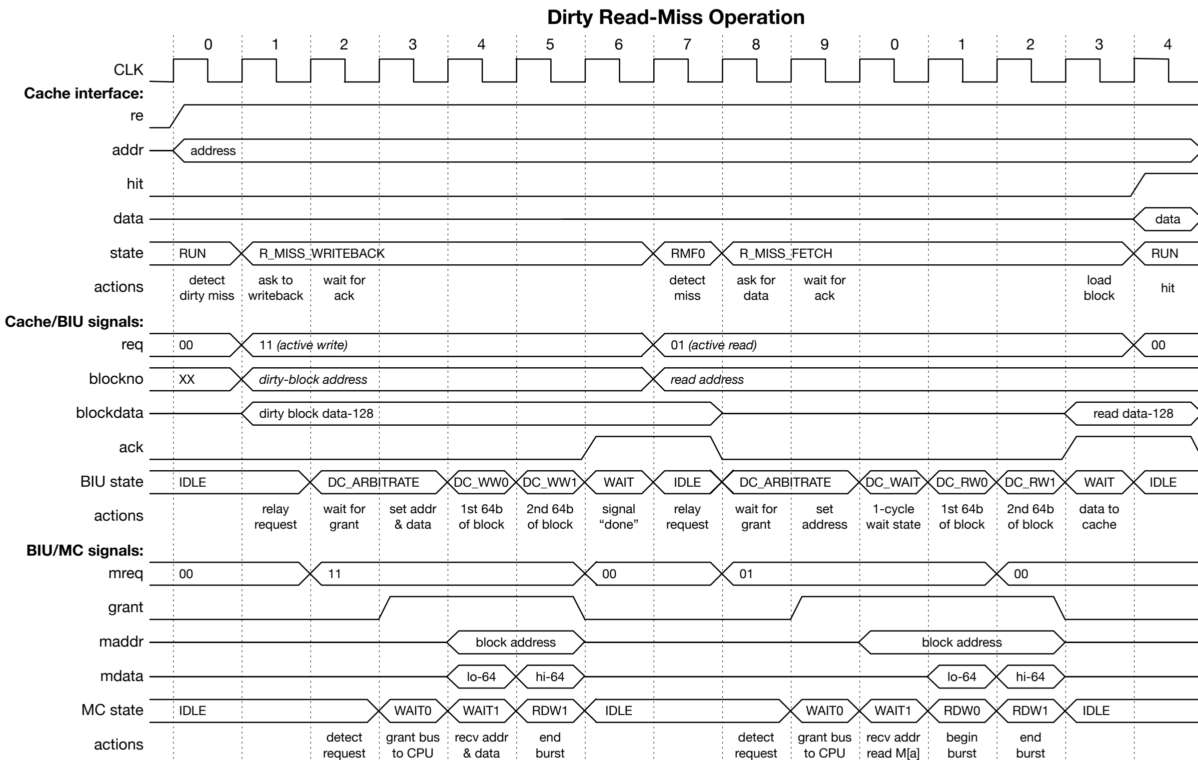
The cache & BIU do handle store operations. The Caches were integrated into the BIU to facilitate the implementation of your cache-coherence engine: it will make it much easier for your BIU to make updates to the cache contents based on events observed on the common address/data bus to main memory.

Cache-Miss Operations & Timing

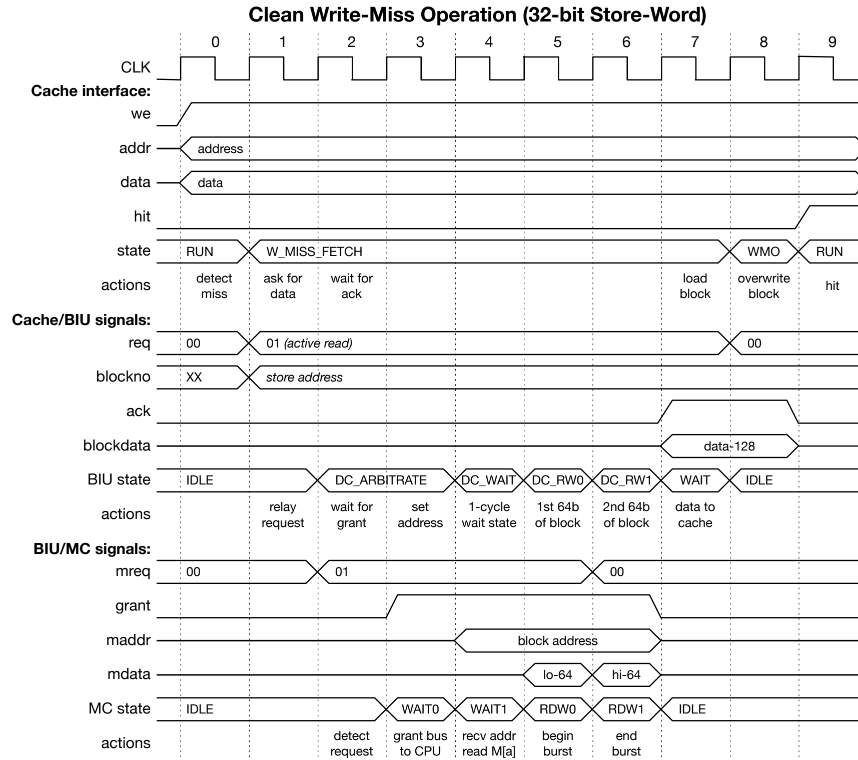
The protocol between our cache and main memory is shown in the following timing diagrams. There are four different scenarios:

1. Clean Read-Miss
2. Dirty Read-Miss
3. Clean Write-Miss
4. Dirty Write-Miss

There are two main possibilities: reads (LW operations) and writes (SW operations), and for each of



these operations, the desired clock in the cache can either be present (a hit, in which case the request is handled immediately and main-memory is not involved), or not (a miss).



In the miss case, we will always fetch from main memory; this is because the cache implements an *allocate-on-write-miss* policy as mentioned above.

The main issue that comes up is whether the block that is being displaced to make room for the incoming block has been previously written: i.e., whether the block is *dirty* or not. If dirty, i.e., if it has been modified since fetching the data from main memory, then the block must be first written back to main memory before any new data can be fetched and brought into the cache. Thus, the protocols for the "dirty" scenarios are much longer and much more complex.

You will see that the protocols are relatively straightforward but take a bit longer than the timing in Project 3, and this was done intentionally to make it *much* simpler to extend and modify.

Project Submission & Grading

Your job is the develop a system that runs the ***reverse-3.s*** code correctly. You can choose whatever type of coherence engine you want (e.g., SI, MSI, MESI, MOESI, etc.). You *do not* have to implement a memory-consistency scheme—i.e., do not worry about sequential consistency, processor consistency, total store order, etc.

You have two deliverables for this project:

1. Submit your working project as a tarball of all your files.
2. Include a README file in PDF that includes diagrams of what you have done. Your documentation need not be elaborate, but it must cover everything you did.

Use the on-line *submit* facility and number this as assignment #4.