Name _____

Computer Architecture and Engineering
CS152 Quiz #5
May 2th, 2013
Professor Krste Asanović

Name: _____<ANSWER KEY>_____

This is a closed book, closed notes exam.
80 Minutes
15 pages

Notes:

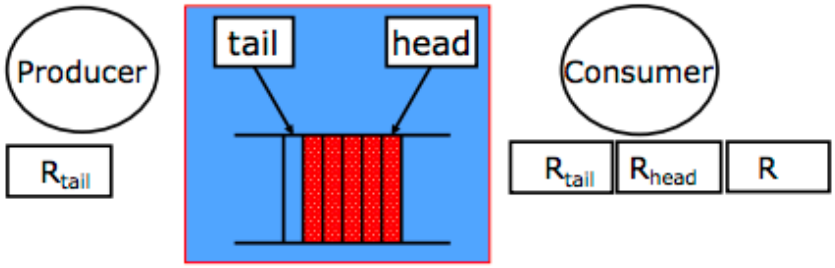
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get no credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	_____	1 Point
Question 1	_____	24 Points
Question 2	_____	27 Points
Question 3	_____	28 Points
TOTAL	_____	80 Points

Name _____

Question 1: Relaxed Memory Models [24 points]

The following code implements a simple producer-consumer code. The producer writes data to the tail pointer of the queue (that has no bounds), while the consumer reads data from the head pointer of the queue (see the figure below). The code is similar to the one used in Lecture 2, however, note that the producer writes 2 elements into the queue every time. The consumer still processes only 1 element at a time. Assume an element is 8 bytes. There is one producer and one consumer.



Producer:

```
Load R_tail, 0(tail)
Store 0(R_tail), x
Store 8(R_tail), y
R_tail = R_tail + 16
Store 0(tail), R_tail
```

Consumer:

```
        Load R_head, 0(head)
spin:   Load R_tail, 0(tail)
        if R_head == R_tail goto spin
        Load R, 0(R_head)
        R_head = R_head + 8
        Store 0(head), R_head
        process(R)
```

This code is correct on a sequentially consistent machine, but may not be on system with a relaxed memory model. With that in mind, answer the following questions on page 3 and 4.

Name _____

Q1.A Memory Ordering Constraints without Fences [6 points]

If a system has a fully relaxed memory model and no fences are inserted in the code, what memory ordering constraints are present? Clearly draw an arrow between *memory instructions* (i.e., **Load/Store** instructions) that have an ordering constraint. For example, **Load** → **Store** means that the Load is ordered to the Store (i.e., **Load** should happen before the **Store**).

Producer:

```
Load Rtail, 0(tail)
Store 0(Rtail), x
Store 8(Rtail), y
Rtail = Rtail + 16
Store 0(tail), Rtail
```

-2 if missed important edges
-0.5 for any wrong edge

Consumer:

```
      Load Rhead, 0(head)
spin: Load Rtail, 0(tail)
      if Rhead == Rtail goto spin
      Load R, 0(Rhead)
      Rhead = Rhead + 8
      Store 0(head), Rhead
      process(R)
```

Q1.B Add Global Memory Fences [6 points]

The code is correct on a sequentially consistent system, but on a system with a fully relaxed memory model it may not be. Insert the *minimum number* of *global memory fences* (**Membar**$_{all}$ instruction) to make the code correct on a system with a relaxed memory model. Note that a *global memory fence* orders all loads and stores preceding the fence before all loads and stores following the fence. Also, draw arrows for additional memory ordering constraints that the **Membar**$_{all}$ instructions introduced.

Producer:

```
Load Rtail, 0(tail)
Store 0(Rtail), x
Store 8(Rtail), y
Membarall
Rtail = Rtail + 16
Store 0(tail), Rtail
```

-3 if missed Membar
-1 for any additional Membar
-0.5 for any wrong edge

Consumer:

```
      Load Rhead, 0(head)
spin: Load Rtail, 0(tail)
      if Rhead == Rtail goto spin
      Membarall
      Load R, 0(Rhead)
      Rhead = Rhead + 8
      Store 0(head), Rhead
      process(R)
```

Name _____

Q1.C Add Fine-Grain Memory Fences [6 points]

Now assume that you have fine-grain memory fences (**Membar$_{LL}$**, **Membar$_{LS}$**, **Membar$_{SL}$**, **Membar$_{SS}$**). The suffix after the **Membar** specifies the type of memory instructions that are ordered (e.g., **Membar$_{LL}$** orders all loads before the fence to all loads after the fence, **Membar$_{LS}$** orders all loads before the fence to all stores after the fence). Insert the *minimum number* of fences to make the code correct on a system with a fully relaxed memory model. Also draw arrows for additional memory ordering constraints that the **Membar** instructions introduced.

Producer:

```
Load Rtail, 0(tail)

Store 0(Rtail), x

Store 8(Rtail), y
Membar SS
Rtail = Rtail + 16

Store 0(tail), Rtail
```

-3 if missed Membar
-1 for any additional Membar
-0.5 for any wrong edge

Consumer:

```
      Load Rhead, 0(head)

spin: Load Rtail, 0(tail)

      if Rhead == Rtail goto spin
      Membar LL
      Load R, 0(Rhead)

      Rhead = Rhead + 8

      Store 0(head), Rhead

      process(R)
```

Q1.D Impact on Performance [6 points]

The global memory fence can add more memory ordering constraints than are necessary for correct execution of a code sequence. Is it possible to have an interleaving of memory instructions that is valid in Q1.C but invalid in Q1.B? If so, show the ordering by writing numbers next to the memory instructions, starting from 1 as the first executed instruction. If not, check here _____.

Producer:

```
_1_ Load Rtail, 0(tail)

_2_ Store 0(Rtail), x

_3_ Store 8(Rtail), y

    Rtail = Rtail + 16

_4_ Store 0(tail), Rtail
```

The point of the question was with fine-grain fences, there is no ordering constraint for the load->store instruction on the consumer side.

Consumer:

```
      _5_ Load Rhead, 0(head)

spin: _7_ Load Rtail, 0(tail)

        if Rhead == Rtail goto spin

      _8_ Load R, 0(Rhead)

        Rhead = Rhead + 8

      _6_ Store 0(head), Rhead

        process(R)
```

Name _____

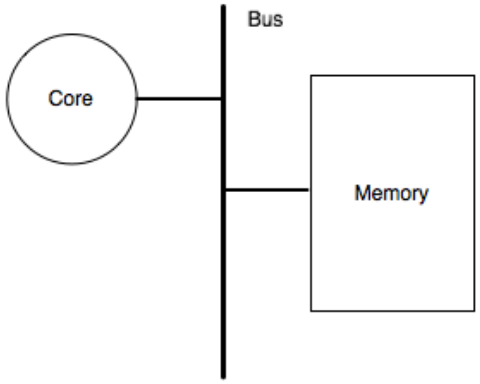
Question 2: Parallel Histograms [27 points]

For this question, we consider executing a histogram kernel on systems that have one or more cores on a shared memory bus. Each core is a simple single-issue, in-order core. The following histogram kernel pseudo-code reads in elements, calculates the bin for every element, and increments a counter for that bin.

```
loop:
  ld element, 0(ptr)
  bin = calculate_bin(element)
  ld count, 0(bin)
  addi count, count, 1
  sd count, 0(bin)
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

The calculate_bin part is abstracted away as in a function, but in reality, it will be mapped down to machine instructions. Through this question, assume that the input elements are uniformly randomly distributed across the histogram bins, and the histogram bins are correctly zeroed out initially.

First consider running the code on the following system that has one core connected to the memory bus. Note the core doesn't have a cache.



We use the following simple performance model to calculate how many cycles it will take to bin an element.

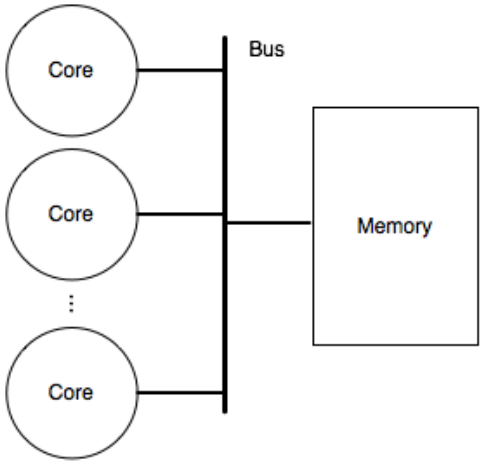
Load: 1 cycle + 1 bus access (1 bus access = 10 cycles, load will occupy the bus for 10 cycles)
Store: 1 cycle + 1 bus access (store also occupies the bus for 10 cycles)
Calculate_bin: 5 cycles
Integer instructions: 1 cycle

The given code will take 11 (load) + 5 (calculate_bin) + 11 (load) + 1 (addi) + 11 (store) + 1 (addi) + 1 (bne) = 41 cycles/element. Out of the 41 cycles, the core will occupy the bus for 30 cycles.

Name _____

Q2.A Histograms on Multi-core Systems [4 points]

Consider the following multi-core.



Andrew points out that the code in the previous page will not work properly on the system above. What is the problem with the original code when running on a system with many cores? Come up with an interleaving of instructions that would exercise the problem.

If instructions from core 0 and core 1 interleaves the following way to update the same bin:

Core 0	Core1
ld	
	ld
addi	
	addi
sd	
	sd

One update will be dropped.

Andrew revises the code:

```
loop:
  ld element, 0(ptr)
  bin = calculate_bin(element)
  fetch_and_add bin, 1
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

Why would using **`fetch_and_add`** fix the problem?

fetch_and_add provides atomicity.

Name _____

Q2.B Performance with Infinite Number of Cores without Caches [5 points]

Assume we have many elements (long input vector) for the histogram, and the elements are distributed among cores evenly.

`fetch_and_add` takes 1 cycle + 1 bus access for the memory read, + 1 cycle for the add, + 1 bus access for the memory write, for 22 cycles total. The fetch_and_add instruction occupies the bus for 21 cycles, as the bus is locked to provide atomicity during the read/add/write sequence. While the bus is locked, other cores cannot put their requests on the memory bus.

What would the performance be in terms of cycles/element in the steady state running the code in Q2.A (the code is also in the Appendix, which you may detach) with an infinite number of cores without caches? Justify your work.

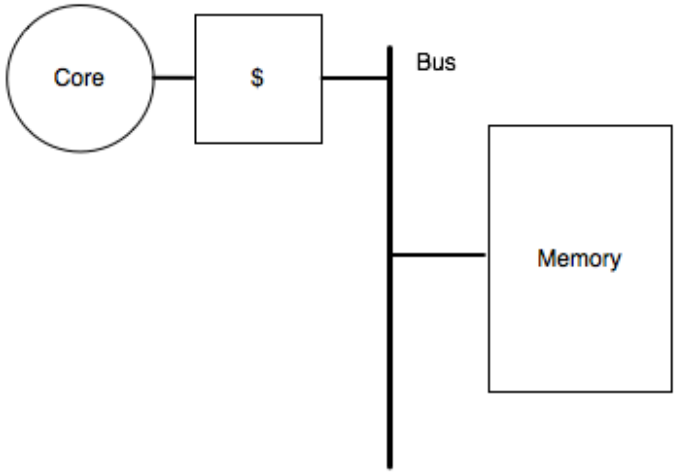
Bus occupancy / element = 10 + 21 = 31 cycles/element
(Assuming all non-bus activity is overlapped)

21 cycles/element = 2 points
-1 for not setting occupancy correct

Name _____

Q2.C Single Core with a Single-Word-Line Cache [5 points]

Now consider the following single-core system with single-word-line write-back caches.



Assume we have many elements (long input vector) to be histogrammed. The caches are big enough to hold all the bin data while we are running the histogram kernel. The characteristics of the cache are:

Load/Store Hit in the cache: 1 cycle
Load/Store Miss with a clean victim: 1 cycle + 1 bus access
Load/Store Miss with a dirty victim: 1 cycle + 2 bus accesses (1 to write back, 1 to pull data in)
Fetch_and_add hit in the cache: 3 cycles
Fetch_and_add miss with a clean victim: 3 cycles + 1 bus access
Fetch_and_add miss with a dirty victim: 3 cycles + 2 bus access

The fetch_and_add is performed atomically on the data in the local cache, and does not impact other accesses across the bus.

What would the performance be in terms of cycles/element running the code in Q2.A (the code is also in the Appendix, which you may detach) in the steady state? Justify your work.

1 cycle + 2 bus access	ld – miss with dirty victim
+ 1 bus access	later causes miss with clean victim
5 cycles	calculate_bin
3 cycles	fetch_and_add hit
1 cycle	addi
1 cycle	bne

11 cycles + 3 bus accesses = 41 cycles / element (assuming cache capacity is just big enough to hold bins)
3 points for 31 cycles / element (if load misses with clean victim).
5 points for some number between 31-41 cycles / element.
1 point if you got 41 cycles / element assuming fetch_and_add missing in the cache

Q2.D Load No-Allocate [4 points]

We quickly realize that we are streaming through the elements (long input vector), and polluting the cache that holds useful data (bin data of the histogram). We extend our ISA to have a "load no-allocate instruction", which does not allocate space for the data in cache if the load is a miss, but only loads the value from memory into the destination register. Here's the new code:

```
loop:
  ld.noallocate element, 0(ptr)
  bin = calculate_bin(element)
  fetch_and_add bin, 1
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

Assume we have many elements (long input vector) for the histogram, and the caches are big enough to hold all the bin data of the histogram. The characteristics of the cache are the same as Q2.C. What would the performance be in terms of cycles/element running the code above in the steady state? Justify your work.

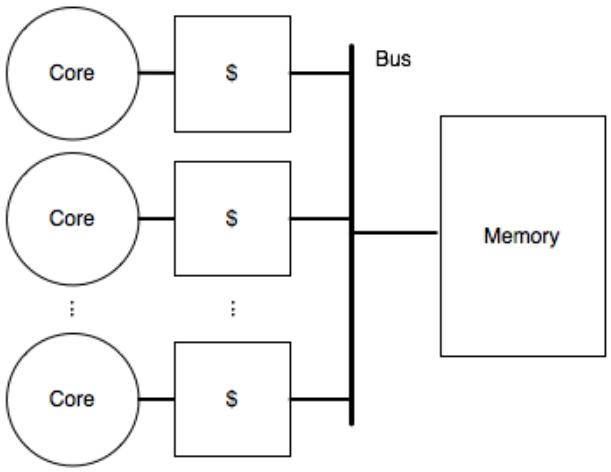
1 cycle + 1 bus access	ld.noallocate
5 cycles	calculate_bin
3 cycles	fetch_and_add
1 cycle	addi
1 cycle	bne

11 cycles + 1 bus access = 21 cycles / element

Name _____

Q2.E Optimal Number of Cores [4 points]

Consider the following multi-core system with single-word-line write-back caches.



Assume we have many elements (long input vector) for the histogram, and the input data is uniform random. The caches are big enough to hold all the bin data of the histogram. The characteristics of the cache are the same as Q2.C. To keep the caches coherent, we implement a snoopy cache coherence protocol on the bus. We can snoop, invalidate, and transfer the data all in 1 bus access of 10 cycles.

Determine the optimal number of cores to give the highest performance running the code in Q2.D (the code is also in the Appendix, which you may detach). Justify your work.

1 cycle + 1 bus access	ld.noallocate
5 cycles	calculate_bin
3 + (N-1)/N bus access	fetch_and_add
1 cycle	addi
1 cycle	bne

11 cycles + [1 + (N-1)/N] bus accesses

N = 1, 11 cycles + 1 bus access = 21 cycles / element
N = 2, 15 cycles / element (bottlenecked by bus occupancy)
N = 3, 10*(1+2/3) cycles / element (bottlenecked by bus occupancy)

As N gets bigger, the bus occupancy goes up. Therefore the optimal number of cores is 2.

Name _____

Q2.F Optimal Number of Cores with a New Algorithm [5 points]

Chris points out that rather than using fetch_and_add to atomically increment the bin count, each core could generate a local histogram (increment local bin counters) for its portion of the input vector. Once all cores are finished with their share of the input vector, adding together the local histograms will produce the global histogram. Assume that the last part (adding together the local histograms) doesn't impact performance, since the input vector is large. Because we are doing a local histogram, we don't need to use an atomic fetch_and_add. Here's the revised code:

```
loop:
  ld.noallocate element, 0(ptr)
  bin = calculate_bin(element)
  ld count, 0(bin)
  addi count, count, 1
  sd count, 0(bin)
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

We run this code on the multi-core system shown in Q2.E. Use the same performance model assumed in Q2.E.

Determine the optimal number of cores that will result in best performance. Justify your work.

1 cycle + 1 bus access	ld.noallocate
5 cycles	calculate_bin
1 cycle	ld hit
1 cycle	addi
1 cycle	sd hit
1 cycle	addi
1 cycle	bne

11 cycles + 1 bus access

N = 1, 11 cycle + 1 bus access = 21 cycles / element
N = 2, 10.5 cycles / element (can do 2 elements every 21 cycles, because limited by 11 cycle + 1 bus access)
N = 3, 10 cycles / element (limited by bus occupancy)
N = 4, 10 cycles / element (limited by bus occupancy)

As N gets bigger you are still limited by bus occupancy. Therefore the optimal number of cores is 3.

Question 3: Directory Protocols [28 points]

Your TA Yunsup is planning to build a 1024-core system that is cache coherent. Each core will have an 8-way set-associative 8KB cache with 64-byte cache lines. The system has 256 GB of DRAM. Recalling that directories are more scalable than snoopy buses, Yunsup decides to implement a full-map directory to maintain coherence.

Yunsup's first directory cache coherence protocol maintains a bit vector for each cache-line-sized block of main memory. For a given memory block, each bit in the vector represents whether or not one core in the machine has that line cached. To implement this scheme, Yunsup figures out he needs 512 GB to store only the directory bits (larger than the total DRAM in the system!).

Q3.A Compressed Directory [3 points]

Quickly realizing that the full-map directory scheme might be impractical to implement, Yunsup considers a compressed directory scheme where each bit in the vector now represents that one or more cores in a group of 32 cores has that line cached. For example, if bit 1 of the vector is valid, it means that at least one of cores between 32-63 has a cached copy. How large is the compressed directory?

256GB / 64 byte lines * 1024/32 bits/line = 16 GB

Q3.B 1-bit Directory [3 points]

Yunsup realizes that the compressed directory is still too big, and considers having only one bit for each memory line. The 1-bit is used to represent whether or not any core on the chip has that line cached. Yunsup calls this scheme the 1-bit directory. Now, how large is this 1-bit directory?

16GB / 32 = 512MB

Name _____

Q3.C Reverse Directory [6 points]

Yunsup finally figures out that the problem of directory schemes proposed in Q3.A-Q3.C is that the machine is allocating bits per all memory lines that exist in the system; even for the lines that are not cached by any core in the system. Yunsup does some research and realizes that he could build a reverse directory.

In a reverse directory scheme, tags of all caches in the system are duplicated in the directory controllers. Each tag in the directory controller has an associated valid bit. The duplicated tags are organized in a different way; tags from the same set across all caches live in the same directory controller. All cache requests go to the directory controller that has the duplicated tags of the corresponding set to see if any cache has a copy of the same line. The directory protocol keeps the duplicated tags coherent with the tags in the caches. For example, once a line in the cache is evicted, the directory controller is notified so that it will invalidate the duplicated tag.

Given that virtual addresses are 64 bits, how large is the reverse directory?

8KB / 64 bytes/line = 128 lines
8-way set-associative, 128 / 8 = 16 sets / cache

tag size = 64 – 4 (index) – 6 (offset) = 54 bits
valid bit = 1 bit

55 bits/line * 128 lines/cache * 1024 caches = 880 KB

-1 for error in tag calculation
-1 for missing the valid bit
-2 for missing one part of the multiplication
-3 for not multiplying at all

Name _____

Q3.D Invalidations for a Load [4 points]
Assume that core #250 has written to address X. Right after that, core #999 wants to read from address X. How many invalidation messages must be sent in total?

(1) Compressed Directory

32 invalidation messages go out to the 32 cores that include #250

(2) 1-bit Directory

1023 invalidation messages go out
I didn't penalize people who said 1024, since it wasn't clear in the question whether you send an invalidation message to yourself

(3) Reverse Directory

1 invalidation message, as reverse directory has precise state

-1.5 if wrong answer, -0.5 for trivial mistake

Q3.E Invalidations for a Test&Set [5 points]
Consider the case where we use the atomic instruction test&set to implement a lock. For the case where all 1024 cores execute one test&set instruction to the same memory address (initially the memory address isn't present in any cache in the system), how many invalidation messages must be sent to complete all test&set instructions?

(1) Compressed Directory

[31/32] * 1023
The important thing here is that the first test&set doesn't send out any invalidation message, as the question says initially the memory address isn't present in any cache in the system. The reason why I said 31/32 is again, you might optimize the cache coherence protocol to not send an invalidation message to yourself.

(2) 1-bit Directory

[1023/1024] * 1023
Again, the first test&set doesn't send out any invalidation message. 1023/1024 depends on whether you send an invalidation message to yourself.

(3) Reverse Directory

1023
For the same reason, the first test&set doesn't send out an invalidation message.

-2 for wrong answer, -1 if you didn't omit the first invalidation message, -0.5 for trivial mistake

Name _____

Q3.F Invalidations for Test&Test&Set [7 points]
Realizing that using a Test&Set instruction to implement a lock generates a lot of invalidation traffic while waiting for the lock to be freed, we instead implement Test&Test&Set as the following:

```
lock(addr):
    while ((*addr == 1) || test&set(addr) == 1) ;
```

At the start of the program, the lock is not held and is not in any cache. Core #1, #2, #3, #258, #931 are competing for a lock. Every thread performs the first Test, which evaluates to false since the lock was not held. Each thread then executes the atomic Test&Set. Assume core #931 wins the lock. How many invalidation messages must be sent in this case?

After the first test, every core will have the line cached. The first test&set will have to invalidate all copies, and the following 4 test&set instructions will only have to invalidate one exclusive copy.

(1) Compressed Directory

(32 + 32 + [31/32]) + 32 + [31/32] + [31/32] + 32

The first clause is the number of invalidation messages you need to send in order to invalidate all shared copies. The following terms are invalidating exclusive copies.
[31/32] depending on whether you send an invalidation message to yourself.

(2) 1-bit Directory

[1023/1024] * 5

With a 1-bit directory scheme, the same amount of invalidation messages are sent out to invalidate many shared copies or one exclusive copy.

(3) Reverse Directory

4 + 1 + 1 + 1 + 1

Again, the first clause is the number of invalidation messages you need to send in order to invalidate 4 shared copies. The following terms are invalidating exclusive copies.

- Many students didn't count invalidation messages for the following 4 test&set instructions. I took a point off for that.
- Many students said for the compressed directory scheme, you send 64 invalidation messages to invalidate all shared copies. However, since the compressed directory scheme only uses 1 bit for 32 cores, you still need to send [31/32] invalidation messages to insure exclusivity. I took a point off for this case.
- -2 for wrong answer

Appendix

Code used for Q2.A, Q2.B, Q2.C

```
loop:
  ld element, 0(ptr)
  bin = calculate_bin(element)
  fetch_and_add bin, 1
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

Code used for Q2.D, Q2.E

```
loop:
  ld.noallocate element, 0(ptr)
  bin = calculate_bin(element)
  fetch_and_add bin, 1
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

Code used for Q2.F

```
loop:
  ld.noallocate element, 0(ptr)
  bin = calculate_bin(element)
  ld count, 0(bin)
  addi count, count, 1
  sd count, 0(bin)
  addi ptr, ptr, 8
  bne ptr, bounds, loop
```

YOU MAY DETACH THIS PAGE