Computer System Architecture
6.823 Quiz #3
April 26th, 2019

Name: _____

This is a closed book, closed notes exam.
80 Minutes
15 Pages (+2 Scratch)

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.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 16 and 17 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	_____	38 Points
Part B	_____	35 Points
Part C	_____	27 Points
TOTAL	**_____**	**100 Points**

Part A: Cache Coherence (38 points)

Ben Bitdiddle writes a parallel program where two processors P1 and P2 increment a shared counter. The following is the memory access trace of the program, which shows that P1 and P2 alternate reading and writing the same shared counter at address `A`:

```
P1: LD A
P1: ST A
P2: LD A
P2: ST A
P1: LD A
P1: ST A
...
```

Question 1 (10 points)

Ben is building a bus-based multicore, and wants to evaluate the tradeoff between the MSI and MOSI coherence protocols (refer to the Quiz 3 Handout for details on the MSI and MOSI coherence protocols). Fill in the two tables below to show the states and bus messages for both protocols (for MSI on the first table, then for MOSI on the second table, in the next page). The top row shows the initial state of line A, which is Invalid in both P1 and P2's private caches. On each row, write the state of line A on each cache *after* the access is performed, and all the bus messages that the access causes.

	MSI		
	P1 State	**P2 State**	**Bus Messages**
`Initial State`	**I**	**I**	
`P1:LD A`	**S**	**I**	**BusRd**
`P1:ST A`	**M**	**I**	**BusRdX**
`P2:LD A`	**S**	**S**	**BusRd/BusWB**
`P2:ST A`	**I**	**M**	**BusRdX**
`P1:LD A`	**S**	**S**	**BusRd/BusWB**

	MOSI		
	P1 State	P2 State	Bus Messages
`Initial State`	I	I	
`P1:LD A`	S	I	BusRd
`P1:ST A`	M	I	BusRdX
`P2:LD A`	O	S	BusRd/BusFwd
`P2:ST A`	I	M	BusRdX/BusFwd
`P1:LD A`	S	O	BusRd/BusFwd

Question 2 (4 points)

In steady state, how many writebacks to memory are performed per memory access (load or store) with the MSI protocol for the above trace? What about with the MOSI protocol?

1/2 writeback per memory access for MSI
0 for MOSI

Question 3 (5 points)

Ben decides that forwarding the data upon receiving a `BusRd` request in `O` state is wasteful. Ben modifies the `MOSI` protocol so that, when a cache observes a `BusRd` request for a cache line it has in `O` state, the cache stays silent instead of replying with `BusFwd`, and the requester reads the data from memory. Is this a correct cache coherence protocol? If so, briefly explain why. If not, give a sequence of memory accesses where the protocol fails to maintain cache coherence.

No this is not a correct coherence protocol.
A simple example is the following:

P1: ST A
P2: LD A
P3: LD A

Now, P3 has the stale copy of the data that it grabbed from memory.

Question 4 (7 points)

The threads in Ben's program perform atomic Fetch-and-Add (XADD) operations to increment the shared counter. The XADD instruction is given below:

```
XADD rVal, Imm(rBase):
      old ← Memory[(rBase) + Imm]
      Memory[(rBase) + Imm] ← old + (rVal)
      rVal ← old
```

`XADD rVal, Imm(rBase)` *atomically* increments the value at the effective memory address by the value of register `rVal`, and writes the old value at the effective memory address to `rVal`. XADD is atomic, meaning that no intervening memory operations can occur between the read of `Memory[(rBase) + Imm]` and the subsequent write.

Ben wants to run his code on a MIPS processor that does not implement the XADD instruction, but has load-reserve (LR) and store-conditional (SC) instructions, given below:

```
LR rs, Imm(rt):
      rs ← Memory[(rt) + Imm]
      Track address (rt) + Imm

SC rs, Imm(rt):
      If (rt) + Imm modified:
            rs ← 0                        # Fail
      Else:
            Memory[(rt) + Imm] ← (rs)     # Succeed
            rs ← 1
```

Implement `XADD rVal, Imm(rBase)` using LR and SC. If needed, you can use registers `r1` through `r4` for temporary values.

```
XADD rVal, Imm(rBase):
      Loop:       LR r1, Imm(rBase)
                  ADD r2, r1, rVal
                  SC r2, Imm(rBase)
                  BEQZ r2, Loop
                  ADDI rVal, r1, 0
```

Question 5 (7 points)

Ben now uses a MIPS processor that does not have `LR`/`SC`, but implements the atomic compare-and-swap (`CAS`) instruction instead, which is given below:

```
CAS rOld, rNew, Imm(rBase):
     old ← Memory[(rBase) + Imm]
     If old == (rOld):
          Memory[(rBase) + Imm] ← (rNew)
     else:
          rOld ← old
```

`CAS rOld, rNew, Imm(rBase)` *atomically* loads the value at the effective memory address and compares it with the value stored in register `rOld`. If both values are equal, it updates the memory location with the value stored in register `rNew`. If both values are not equal, it updates the value in `rOld` with the value loaded from memory. `CAS` is atomic, meaning that no intervening memory operation can occur between the read of `Memory[(rBase) + Imm]` and the subsequent write.

Implement `XADD rVal, Imm(rBase)` using `CAS`. If needed, you can use registers `r1` through `r4` for temporary values.

```
XADD rVal, Imm(rBase):
     Loop:     LD r1, Imm(rBase)
               ADD r2, r1, rVal
               ADDI r3, r1, 0
               CAS r1, r2, Imm(rBase)
               SUB r1, r1, r3
               BNEZ r1, Loop
               ADDI rVal, r3, 0
```

Question 6 (5 points)

Ben writes a new program where two processors now update shared counters within an array of 256 elements. Each counter is a 32-bit integer (`int32_t`):

```
int32_t counters[256];
```

A trace of Ben's program shows that processors P1 and P2 increment nearby counters in the `counters` array. The following trace shows the update pattern of the two processors, where `R1` holds the base address of the `counters` array.

```
Processor 1:      Processor 2:

XADD R2, 0(R1)    XADD R2, 4(R1)
XADD R3, 8(R1)    XADD R3, 20(R1)
XADD R4, 12(R1)   XADD R4, 16(R1)
...               ...
```

Ben's processor has 64-byte cache lines. Is there any unnecessary communication between caches in this scenario? If so, what is a potential **software solution** to mitigate the problem?

The potential problem here is false sharing between the two processor's accesses. Since most of the accesses from both processors go to the same cache line, the line will ping-pong between the two.

There are several potential solutions, notably:

- Padding the counters such that each counter occupies an entire cache line
- Having the compiler rearrange accesses

Part B: Memory Consistency (35 points)

Consider two processors that each read one shared counter and update the other.
Assume that memory locations A and B contain initial value 0.

P1	P2
P1.1: LD Ra ← (A) P1.2: ST (B) ← 1	P2.1: LD Rb ← (B) P2.2: ST (A) ← 1

Question 1 (10 points)

Fill in the table below to indicate which set of final values (Ra, Rb) are possible under the following memory consistency models:

- Sequential Consistency (**SC**)
- Total Store Order (**TSO**): **stores can be reordered after later loads**
- Relaxed Memory Order (**RMO**): **loads and stores can be reordered after later loads and stores**

For each entry, write a check mark if the memory consistency model of the corresponding column can produce the final set of values on the leftmost column of the same row.

(Ra, Rb)	SC	TSO	RMO
(0, 0)	√	√	√
(0, 1)	√	√	√
(1, 0)	√	√	√
(1, 1)			√

Question 2 (6 points)

The RMO machine has the following fine-grained barrier instructions:

- **MEMBAR$_{RR}$** guarantees that all reads that precede MEMBAR$_{RR}$ in program order will be performed before any read that follows the barrier.
- **MEMBAR$_{RW}$** guarantees that all reads that precede MEMBAR$_{RW}$ in program order will be performed before any write that follows the barrier.
- **MEMBAR$_{WR}$** guarantees that all writes that precede MEMBAR$_{WR}$ in program order will be performed before any read that follows the barrier.
- **MEMBAR$_{WW}$** guarantees that all writes that precede MEMBAR$_{WW}$ in program order will be performed before any write that follows the barrier.

Using the **minimum number of memory barrier instructions**, rewrite **P1** and **P2** from Question 1 such that the **RMO machine produces the same outputs as the SC machine for the given code.**

P1	**P2**
P1.1: LD Ra ← (A) MEMBAR$_{RW}$ P1.2: ST (B) ← 1	P2.1: LD Rb ← (B) MEMBAR$_{RW}$ P2.2: ST (A) ← 1

Now consider two processors P1 and P2 running the following code.
Assume that memory locations A, B, and C contain initial value 0. Note that instruction `P1.3` writes the result of `Ra - Ra + 1` to temporary register `t1`, and similarly `P2.2` writes the result of `Rc - Rc + 1` to temporary register `t2`.

P1	P2
P1.1: ST (B) ← 1 P1.2: LD Ra ← (A) P1.3: t1 ← Ra - Ra + 1 P1.4: ST (C) ← t1	P2.1: LD Rc ← (C) P2.2: t2 ← Rc - Rc + 1 P2.3: ST (A) ← t2 P2.4: LD Rb ← (B)

Question 3 (12 points)

Fill in the table below to indicate which sets of final values `(Ra, Rb, Rc)` are possible under the three memory consistency models SC, TSO, and RMO (as in Question 1). For each entry, write a check mark if the memory consistency model of the corresponding column can produce the final set of values on the leftmost column of the same row. Note that in all three consistency models, data-dependent memory operations are performed in program order to preserve correctness.

(Ra, Rb, Rc)	SC	TSO	RMO
(0, 0, 0)		√	√
(0, 0, 1)			√
(0, 1, 0)	√	√	√
(0, 1, 1)	√	√	√
(1, 0, 0)	√	√	√
(1, 0, 1)			
(1, 1, 0)	√	√	√
(1, 1, 1)			

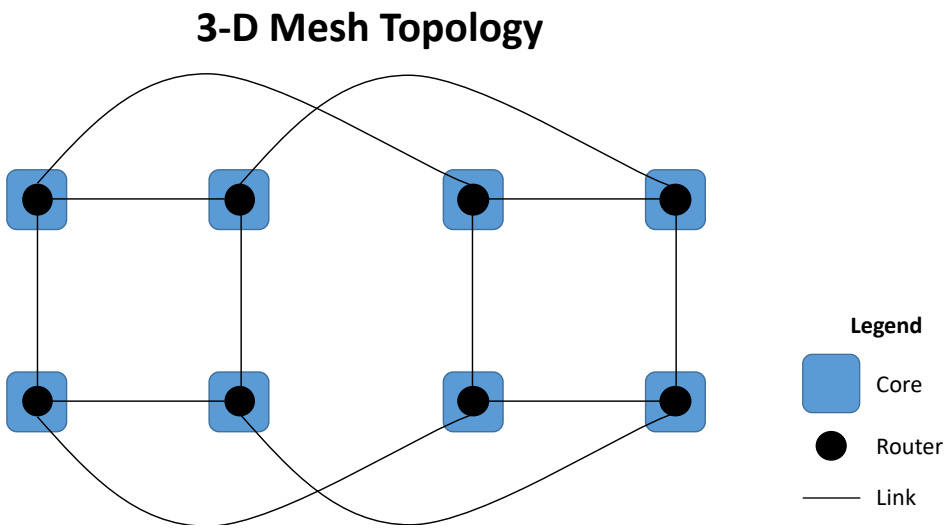
Question 4 (7 points)

Using the **minimum number of memory barrier instructions** (provided in Question 2), rewrite **P1** and **P2** from Question 3 such that the **RMO machine produces the same outputs as the SC machine for the given code.**

P1	**P2**
P1.1: ST (B) ← 1 MEMBAR$_{WR}$ P1.2: LD Ra ← (A)	P2.1: LD Rc ← (C) P2.2: t2 ← Rc - Rc + 1
P1.3: t1 ← Ra - Ra + 1	P2.3: ST (A) ← t2 MEMBAR$_{WR}$
P1.4: ST (C) ← t1	P2.4: LD Rb ← (B)

Part C: Networks (27 points)

Consider the 3-D mesh topology:



The above diagram shows a 3-D mesh with N=8 nodes. For a system of N nodes, the 3-D mesh topology consists of $k = N^{1/3}$ nodes in each dimension.

Question 1 (4 points)

Calculate the **average distance** for the 3-D mesh with N=8 shown above, in number of hops. Assume uniform random traffic, where each node sends $1/N^{th}$ of the traffic to each destination, including itself.

All nodes are symmetric, so only need to calculate average distance from one node.
Avg. distance = (0 + 1 + 1 + 1 + 2 + 2 + 2 + 3) / 8 = 12/8 = 1.5 hops

Question 2 (8 points)

(a) Derive the diameter and bisection bandwidth of an *N*-node 3-D mesh. Assume that *N* is the cubic power of an even number, i.e., $k = N^{1/3}$ is an even integer. You can write your results in terms of *N* and/or *k*. *For partial credit, give the asymptotic growth instead.*

Bisection BW is simply a cut across the cube, which is k^2.
Diameter is hops to get from one corner to another, which is 3(k-1).

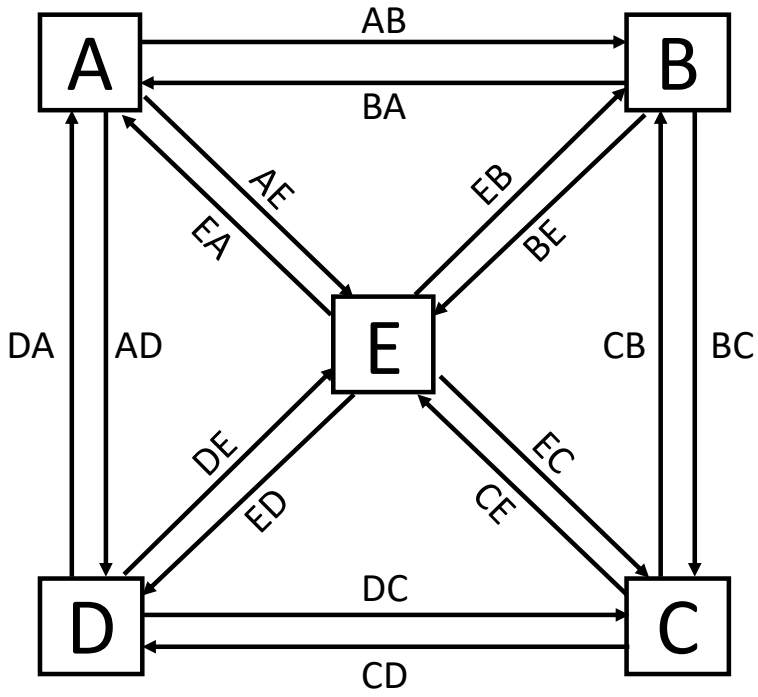
	3-D Mesh
Diameter	*3(k-1)*
Bisection bandwidth	k^2

(b) Derive the **asymptotic growth** of the number of links and average distance with respect to *N* for uniform random traffic. For example, if the average distance grows linearly with *N*, state your solution as O(*N*). Assume that *N* is the cubic power of an even number, i.e., $k = N^{1/3}$ is an even integer. You can write your results in terms of *N* and/or *k*.

Number of links grows linearly with N, since number of links per node stays constant.
Average distance will asymptotically grow linearly with k, same as with diameter.

	3-D Mesh
Number of Links	*O (__N__)*
Average Distance	*O (__k__)*

For Questions 3 and 4, consider the following network (channels are labeled by their source and destination; for example, the channel from A to B is labeled AB).



Question 3 (5 points)

Assume that 180-degree turns are prohibited in this network. Show how a deadlock can occur in this network with no other turns prohibited.

A simple CDG to show the deadlock:

AE->ED->DA->AE

Question 4 (10 points)

We try to eliminate deadlocks by disallowing the following turns:

- Among nodes A, B, C, and D: We allow X-Y routing only (i.e., we disallow north-to-east, north-to-west, south-to-east, and south-to-west turns)
- For all routes that go through node E: We disallow all 90-degree turns from node E. For example, if a packet is routed from D to E, then it can only be routed from E to B (routing from E to C or A are forbidden).

Does this result in a deadlock-free network? If so, draw an acyclic Channel Dependency Graph (CDG) to prove that there is no deadlock. If not, demonstrate the deadlock by showing a cycle from the CDG (in this case you do not need to draw the full CDG).

No – Notably, the turns from routes leaving E and entering the ABCD ring will cause a deadlock.

In CDG:

AE->EC->CD->DA->AE

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade these unless you tell us explicitly in the earlier pages.