# Computer System Architecture
# 6.823 Quiz #3
# April 22nd, 2016
# Professors Daniel Sanchez and Joel Emer

Name: _____Solutions_____

This is a closed book, closed notes exam.
80 Minutes
18 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.
- 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 19 and 20 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

| | | |
|---|---|---|
| Part A | _____ | 24 Points |
| Part B | _____ | 25 Points |
| Part C | _____ | 26 Points |
| Part D | _____ | 25 Points |
| **TOTAL** | **_____** | **100 Points** |

# Part A: Cache Coherence (24 points)

We want to compare the standard directory-based MESI protocol with its simpler sibling, MEI.

The figure below shows their state-transition diagrams.

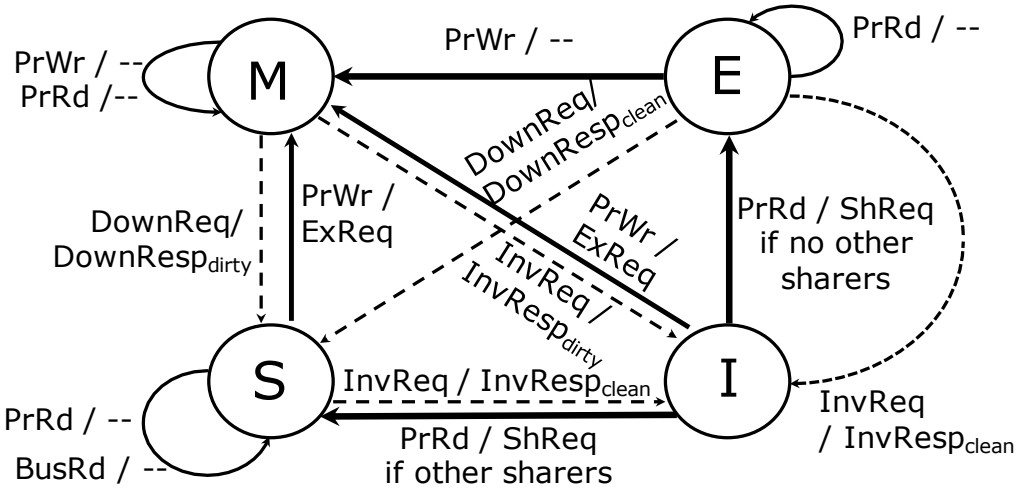



**Figure A-1: MESI protocol state transition diagram**

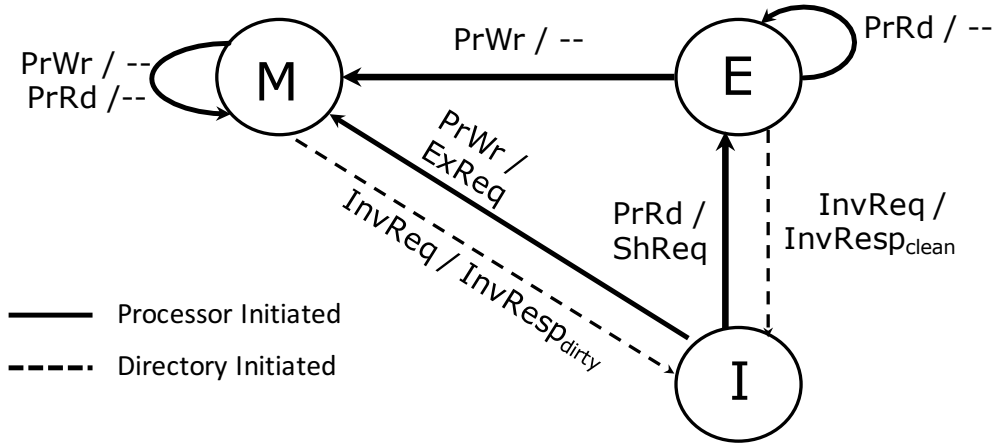



**Figure A-2: MEI protocol state transition diagram**

MEI has no shared state, and both loads and stores must request exclusive permission. Therefore, there is at most one valid copy of the line (in E state if clean, and in M state if dirty).

For this part, **assume that all integers are 32-bits, and cache lines are 32-bits** (i.e. each integer fits exactly in a single cache line). Also **assume that if you do not have the cache line with the required permissions, it counts as a miss** .i.e. ( I → S/M/E; S →M upgrades all count as misses).

## *Question 1 (4 points)*

Two threads running on different cores execute the following code:

```
int A[1024];  // shared across threads

void worker() {
  int counter = 0;
  while (true) {
    for (int i = 0; i < 1024; i++) {
      counter = counter + A[i];
    }
  }
}
```

**Assume `counter`, `i`, and the starting address of `A` are stored in registers**, so each iteration of the loop performs a single load operation to fetch `A[i]`. Each core has a 2048-word cache, so the whole `A` array fits in the cache. Threads 0 and 1 execute at roughly the same speed, and stay about 512 elements apart over time (e.g., when thread 0 is reading element 0, thread 1 is reading element 512, and so on).

(a) In steady state, how many misses per iteration (of the `for` loop) does MESI incur?

0: Both caches can hold the line in the S state.

(b) In steady state, how many misses per iteration (of the `for` loop) does MEI incur?

1: Each cache keeps invalidating the other cache when it reads A[i]

## *Question 2 (6 points)*

Two threads running on different cores execute the following code:

```
int A[1024];  // shared across threads

void worker() {
  while (true) {
    for (int i = 0; i < 1024; i++) {
      A[i] = A[i] + 1;
    }
  }
}
```

As before, **assume that `counter`, `i`, and the starting address of `A` are stored in registers**, so each iteration of the loop performs one load operation to fetch `A[i]`, followed by one store operation to update its value. Each core has a 2048-word cache, so the whole `A` array fits in the cache. Threads 0 and 1 execute at roughly the same speed, and stay about 512 elements apart over time (e.g., when thread 0 is reading element 0, thread 1 is reading element 512, and so on).

(a) In steady state, how many misses per iteration (of the `for` loop) does MESI incur?

2: Each access to A[i] goes through the sequence I → S → M (and correspondingly gets downgraded → S → I, when the other core accesses the same A[i]). Hence two misses.

(b) In steady state, how many misses per iteration (of the `for` loop) does MEI incur?

1: Each access to A[i] brings the line involves a I → M transition.

## *Question 3 (4 points)*

Two threads running on different cores execute the following code:

```
int A[1024];
int B[1024];

void worker() {
  while (true) {
    for (int i = 0; i < 1024; i++) {
      A[i] = A[i] + B[i];
    }
  }
}
```

This code combines the access patterns from Questions 1 and 2: each iteration loads `B[i]`, and both loads and stores `A[i]`. As before, each private cache has 2048 words, and threads 0 and 1 execute at roughly the same speed and stay about 512 elements apart over time (e.g., when thread 0 is reading element 0, thread 1 is reading element 512, and so on). Also **assume `i` and starting address of `A`, `B` are stored in registers**.

(a) In steady state, how many misses per iteration (of the `for` loop) does MESI incur?

2: Accesses to A[i] result in 2 misses. Accesses to B[i] result in 0 misses. The reasoning is similar to that in questions 1 and 2.

(b) In steady state, how many misses per iteration (of the `for` loop) does MEI incur?

2: Accesses to A[i] and B[i] each result in 1 miss, following similar reasoning to questions 1 and 2.

## *Question 4 (10 points)*

To improve the performance of MESI, we modify the directory to predict whether it should give the line in E or S instead of using the current number of sharers. Each tag in the directory is extended with an additional bit, Exclusive-Predict (EP), which works as follows:

- If the line's EP bit is 0, the directory handles read requests like conventional MESI, granting E to the first sharer and S to successive sharers.
- If the line's EP bit is 1, the directory works like MEI: it always grants E to read requests, invalidating other sharers if needed.

Note that the prediction is done on a per-line basis, to accommodate mixed access patterns like the one in Question 3. Also the prediction changes when it is detected that the opposite prediction would have avoided a transaction to the directory (i.e. there is no hysteresis in the predictor).

Your goal is to derive the update rules for the EP bit (Hint: remember that, in MESI, private caches respond to invalidations/downgrades differently if they are in S or E than if they are in M: in S and E, the line is clean, so caches do not include the data in the invalidation/downgrade response; however, in M, the line is dirty, so private caches include the data in their invalidation/downgrade response).

(a) (5 points) Assume a given line's EP is initially 0. What event should cause the directory to set the line's EP to 1?

An ExReq from a processor cache for the line, when the current state of the directory for that line is S.

(b) (5 points) Assume a given line's EP is initially 1. What event should cause the directory to set the line's EP to 0?

A clean invalidation response from a processor cache for the line, when the current state of the directory for that line is E.

# Part B: Memory Consistency Models (25 points)

Consider two processes P1 and P2 running on two different processors.
**Assume that memory locations X and Y contain initial value 0.**

| P1 | P2 |
|---|---|
| P1.1: LD R1 ← (Y)<br>P1.2: LD R2 ← (X) | P2.1: ST (X) ← 1<br>P2.2: ST (Y) ← 1 |

## *Question 1 (3 points)*

Out of the following possible final values of (X, Y, R1, R2), circle the ones that could occur if the system is **Sequentially Consistent (SC)**.

**(0,0,0,0)** **(0,0,0,1)** **(0,0,1,0)** **(0,0,1,1)**

**(1,1,0,0)** **(1,1,0,1)** **(1,1,1,0)** **(1,1,1,1)**

## *Question 2 (3 points)*

Out of the following possible final values of (X, Y, R1, R2), circle the ones that could occur if the system enforces **RMO**, a weak memory model where **loads and stores can be reordered after prior loads or stores.**

**(0,0,0,0)** **(0,0,0,1)** **(0,0,1,0)** **(0,0,1,1)**

**(1,1,0,0)** **(1,1,0,1)** **(1,1,1,0)** **(1,1,1,1)**

## *Question 3 (4 points)*

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

- **MEMBAR$_{RR}$** guarantees that all reads initiated before MEMBAR$_{RR}$ will be performed before any read initiated after it.
- **MEMBAR$_{RW}$** guarantees that all reads initiated before MEMBAR$_{RW}$ will be performed before any write initiated after it.
- **MEMBAR$_{WR}$** guarantees that all writes initiated before MEMBAR$_{WR}$ will be performed before any read initiated after it.
- **MEMBAR$_{WW}$** guarantees that all writes initiated before MEMBAR$_{WW}$ will be performed before any write initiated after it.

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

| **P1** | **P2** |
|---|---|
| P1.1: LD R1 ← (Y) | P2.1: ST (X) ← 1 |
| MEMBAR$_{RR}$ | MEMBAR$_{WW}$ |
| P1.2: LD R2 ← (X) | P2.2: ST (Y) ← 1 |

Again, consider two processes P1 and P2 running the code below on two different processors. **Assume that memory locations X, Y, and Z contain initial value 0.**

| P1 | P2 |
|---|---|
| P1.1: LD R1 ← (Z)<br>P1.2: ST (Y) ← 1<br>P1.3: LD R2 ← (X) | P2.1: ST (X) ← 1<br>P2.2: LD R3 ← (Y)<br>P2.3: ST (Z) ← 1 |

## *Question 4 (5 points)*

Out of the following possible final values of (R1, R2, R3), circle the ones that could occur if the system is **Sequentially Consistent (SC)**.

**(0,0,0)** **(0,1,0)** **(1,0,0)** **(1,1,0)**

**(0,0,1)** **(0,1,1)** **(1,0,1)** **(1,1,1)**

## *Question 5 (5 points)*

Out of the following possible final values of (R1, R2, R3), circle the ones that could occur if the system enforces **RMO** (**loads and stores can be reordered after prior loads or stores**).

**(0,0,0)** **(0,1,0)** **(1,0,0)** **(1,1,0)**

**(0,0,1)** **(0,1,1)** **(1,0,1)** **(1,1,1)**

## *Question 6 (5 points)*

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

| P1 | P2 |
|---|---|
| P1.1: LD R1 ← (Z)<br>MEMBAR$_{RW}$<br>P1.2: ST (Y) ← 1<br>MEMBAR$_{WR}$<br>P1.3: LD R2 ← (X) | P2.1: ST (X) ← 1<br>MEMBAR$_{WR}$<br>P2.2: LD R3 ← (Y)<br>MEMBAR$_{RW}$<br>P2.3: ST (Z) ← 1 |

# Part C: Synchronization (26 points)

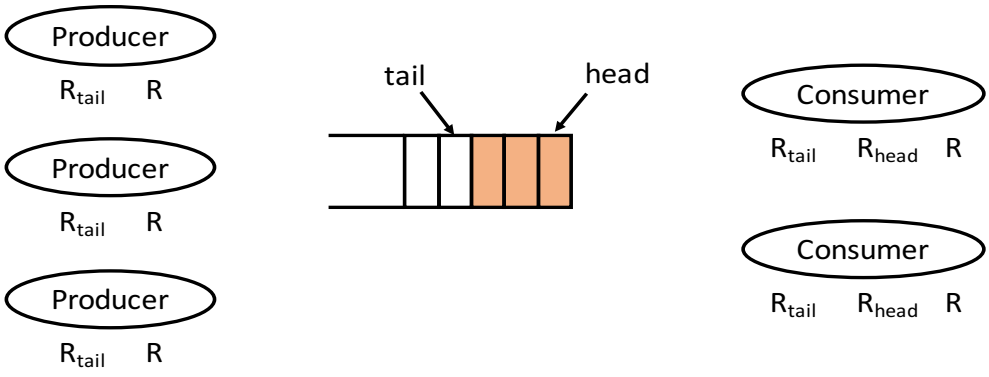
For this question, we will use the load-reserve (LR) and store-conditional (SC) instructions discussed in Lecture 16. Consider the following implementation, in which SC leverages the coherence protocol to cancel reservations from other cores:

- Assume a multi-core system where each core has a private cache. Private caches are kept coherent with a directory-based MSI protocol.
- Each core features a reservation flag, an address, and a status bit for the outcome of SC.
- LR loads a value from memory (fetching the line with shared read-only permission if needed, as a normal load), and sets the reservation flag and address.
- Each core listens for invalidations to the address its hold the reservation for. On a matching invalidation, the core clears the reservation flag.
- SC first checks whether the flag is set and the address matches its own. If so, it requests exclusive permission for the line (potentially triggering an upgrade request and invalidating all other copies). It then checks whether the reservation flag is still set. If set, it performs the store and sets the status flag to 1, denoting success. Otherwise, it does not perform the store and sets the status flag to 0, denoting failure.

| Load-Reserve (LR) | Store-Conditional (SC) |
|---|---|
| BL1: **LR** rs, (rt):<br>BL2: <flag, addr> ← <1, rt><br>BL3: rs ← Mem[rt] | BS1: **SC** (rt), rs:<br>BS2: if <flag, addr> == <1, rt>:<br>BS3: ExclReq(addr)<br>BS4: if <flag, addr> == <1, rt>:<br>BS5: Mem[addr] ← rs<br>BS6: status ← 1<br>BS7: else:<br>BS8: status ← 0<br>BS9: else:<br>BS10: status ← 0 |

**Listing B-1: Load-Reserve, Store-Conditional Implementation**

We will first use LR/SC to implement a multi-producer, multi-consumer FIFO queue (i.e., producer threads push elements to the tail of the queue, and consumer threads pop elements from the head of the queue). Assume an unbounded queue for all questions below.



## *Question 1 (6 points)*

The listing below shows the code for the pop operation, executed by consumer threads.

```
1.1: Try:  LR R_head, (head)
1.2: Spin: Load R_tail, (tail)
1.3:       if (R_head == R_tail): goto Spin  // Queue is empty
1.4:       Load R, R_head                    // Read element at head of queue
1.5:       R_head = R_head + 1               // New value of head pointer
1.6:       SC (head), R_head                 // Update head pointer
1.7:       if (status == 0): goto Try
1.8:       Process(R)
```

Suppose we have several consumer threads popping elements using the code above.

a) (3 points) Assume the queue is never empty. Can **all** consumer threads be stuck in the `Try` loop for an unbounded amount of time (i.e., can the consumers livelock)? Why or why not?
No. For a SC to fail, some other thread must have issued an invalidation for the line i.e. reached its SC first. Even if multiple threads request exclusive access, one thread's request is guaranteed to succeed, in turn invalidating other threads' flags. Thus at least, one consumer thread makes forward progress.

b) (3 points) Assume the queue is never empty. Can **one or more** consumer threads spend an unbounded amount of time in the `Try` loop (i.e., can starvation occur)? Why or why not?
Yes. It is possible that a particular thread continuously gets invalidated between its LR and SC, as other threads' requests succeed.

## *Question 2 (4 points)*

The listing below shows an implementation of the push operation, executed by producer threads.

```
2.1: // Assume R holds the element to push
2.2: Try:  LR R_tail, (tail)
2.3:       Store (R_tail), R
2.4:       R_tail = R_tail + 1
2.5:       SC (tail), R_tail
2.6:       If (status == 0): goto Try
```

Is this code correct? If not, describe what might go wrong.

No. Multiple producers may load(-reserve) the same value of the tail pointer. This may cause the winner thread's (i.e. the one that succeeds with the SC) element to be over-written.

## *Question 3 (4 points)*

The listing below shows an alternative implementation of the push operation.

```
3.1: // Assume R holds the element to push
3.2: Try:  LR R_tail, (tail)
3.3:       R_tail = R_tail + 1
3.4:       SC (tail), R_tail
3.5:       If (status == 0): goto Try
3.6:       Store (R_tail), R
```

Is this code correct? If not, describe what might go wrong.

No. A consumer thread may incorrectly read (or pop) an element from the queue before the producer has successfully written (pushed) the element into the queue. Note that the update to the tail pointer succeeds before the element has been written (pushed) to the queue.

## *Question 4 (6 points)*

Ben Bitdiddle is unhappy with the performance of our LR/SC implementation: when multiple threads take turns updating a shared variable (e.g., the head pointer in Question 1), each LR/SC incurs two roundtrips to the directory: first, LR fetches the variable's line in the S state, and then SC causes an upgrade to M state.

To improve performance, Ben argues that LR should fetch the variable's line in M state directly. This way, SC will not incur a second roundtrip in the common case, improving performance.

Does Ben's change affect the correctness or forward-progress guarantees of the implementation? Explain why or why not.

Ben's optimization no longer guarantees forward-progress. Since each thread's LR fetches the line in M state, it is possible that before reaching a SC, an intervening LR from another thread invalidates the line, causing the SC to fail. It is possible that multiple threads' LR interleave before any thread has the opportunity to complete a SC successfully.

## *Question 5 (6 points)*

Ben Bitdiddle wants to make our implementation of LR/SC more efficient (consider again our standard implementation from Question 1, not the one from Question 4). Ben argues that, because we are using the coherence protocol, the reservation flag and address register are not needed: LR works just like a normal load, and SC fails if the line is invalidated from the core's private cache anytime before SC acquires exclusive permission to the line, as shown below.

| **Load-Reserve (LR)** | **Store-Conditional (SC)** |
|---|---|
| 5L1: **LR** rs, (addr):<br>5L2: rs ← Mem[addr] | 5S1: **SC** (addr), rs:<br>5S2: if addr in I:<br>5S3: status ← 0<br>5S4: else:<br>5S5: ExclReq(addr)<br>5S6: if addr was invalidated<br>5S7: before we acquired M:<br>5S8: status ← 0<br>5S9: else:<br>5S10: Mem[addr] ← rs<br>5S11: status ← 1 |

This way, if there are multiple racing SCs, one of them will reach the directory first, causing all other copies to be invalidated, and all losing SCs will fail. In other words, **the line's state is used to fulfill the same role as the reservation flag in the original implementation**.

Does removing the reservation flag affect the correctness or forward-progress guarantees of the implementation? If so, describe a concrete scenario where LR/SC behaves incorrectly.

In the absence of the reservation flag, suppose a thread (T1) initiates a SC and issues an ExclReq. But before the request is complete, T1 is switched out (say by the OS), and a new thread (T2) begins execution on the core. The ExclReq initiated by T1 completes, and the line is now present in cache in the M state. T2 performs a LR-SC on the same address, and is then switched out. T1 is now switched in by the OS, and sees the line in M state in its cache, and successfully completes its SC. But this is incorrect behavior.

In essence, in the presence of multiple threads, it appears as if no other cache has performed a LR-SC on this address. A similar situation can arise in the context of multi-threading without reservation flags. In the presence of reservation flags, the flag registers are saved as part of the thread context and uniquely associated with each thread preventing this problem.

# Part D: On-chip Networks (25 points)

## *Question 1 (6 points)*

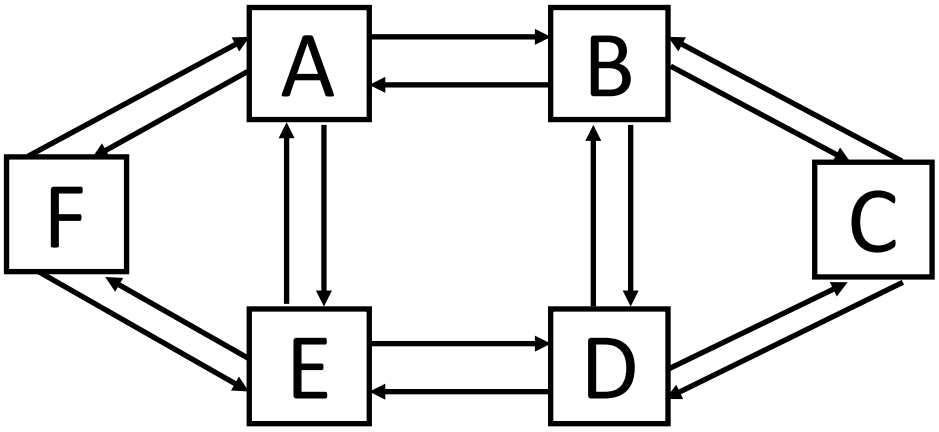
Determine whether the following routing algorithms are deadlock-free for a 2D-mesh. State your reasoning.

a) (3 points) Randomized dimension-order: All packets are routed minimally. Half of the packets are routed completely in the X dimension before the Y dimension, and the other packets are routed in the Y dimension before the X dimension.
Not deadlock-free. All turns in the turn model are allowed, and hence not deadlock-free. Alternatively, you may argue that the CDG has a cycle.

b) (3 points) Less randomized dimension-order: All packets are routed minimally. Packets whose minimal direction is increasing in both X and Y always route **X before Y**. Packets whose minimal direction is decreasing in both X and Y always route **Y before X**. All other packets choose randomly between X before Y and vice-versa.
Deadlock-free. In essence, this prevents north-to-east and west-to-south turns in the turn model which is sufficient to prevent deadlock.

## *Question 2 (19 points)*

Consider the following topology:




(a) (2 points) What is the diameter of this topology?

3

(b) (2 points) What is the bisection bandwidth (in flits/cycle) of this topology?

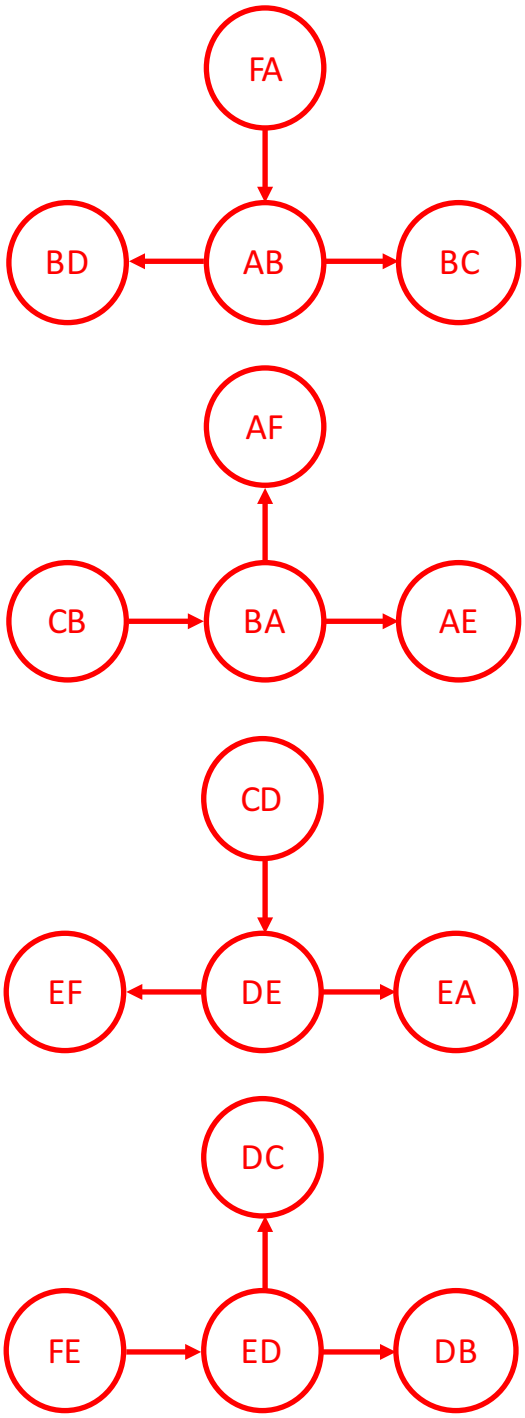
4

(c) (5 points) Assume that 180-degree turns are prohibited. No other turns are prohibited. Show how deadlock could arise in the given topology.

AB → BD → DE → EA → AB is a cycle in the CDG

(d) (10 points) We now restrict all routes to be minimal and disallow the following turns on the mesh (among the nodes A, B, E, D): north-to-east, north-to-west, south-to-east, south-to-west. Is the routing strategy deadlock-free? Draw the CDG to justify your answer.

Deadlock-free as shown by the CDG below.



## *Scratch Space*

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

## *Scratch Space*