Computer System Architecture
6.823 Quiz #3
April 30th, 2021

Name: ______________________________

90 Minutes
17 Pages

Notes:
- Not all questions are equally hard. Look over the whole quiz and budget your time carefully.
- Please state any assumptions you make, and show your work.
- Please write your answers by hand, on paper or a tablet.
- Please email all 17 pages of questions with your answers, including this cover page. Alternatively, you may email scans (or photographs) of separate sheets of paper. Emails should be sent to 6823-staff@csail.mit.edu
- Please ensure your name is written on every page you turn in.
- Do not discuss a quiz's contents with students who have not yet taken the quiz.
- Please sign the following statement before starting the quiz. If you are emailing separate sheets of paper, copy the statement onto the first page and sign it.

I certify that I will start and finish the quiz on time, and that I will not give or receive unauthorized help on this quiz.

Sign here: ______________________________

Part A	________	30 Points
Part B	________	20 Points
Part C	________	20 Points
Part D	________	30 Points
TOTAL	________	**100 Points**

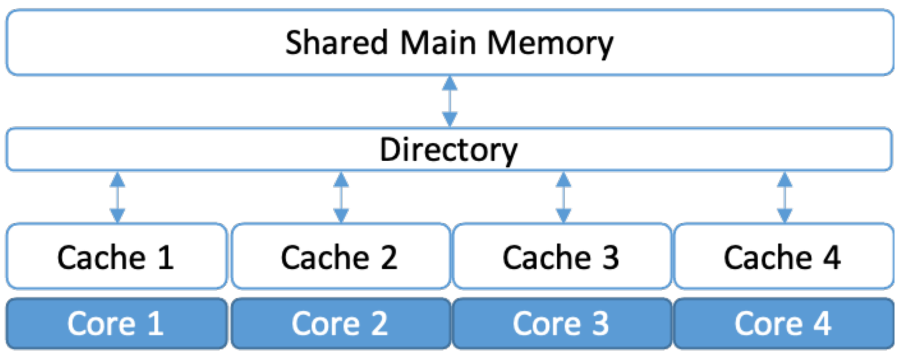
Name ______________________________

Part A: Cache Coherence (30 points)

Ben Bitdiddle is given a multicore processor that enforces cache coherence using a directory-based MESI protocol with silent evictions. The Quiz 3 handout details this coherence protocol.

Question 1 (5 points)

Consider the four-core system below. Each core has a *private cache that can only hold a single cache line*, and the caches start out empty. Each core runs a thread that performs the following set of reads and writes. The number in parenthesis indicates the global order of accesses (i.e., Core 1's LD A happens before ST A, which happens before Core 2's LD B, etc).



`(1) LD A` `(2) ST A`	`(3) LD B` `(6) ST B`	`(4) LD B`	`(5) LD B`

(a) What is the final coherence state for each of the four caches?

(b) How many of the following requests are sent with the MESI protocol?

- `ShReq`
- `ExReq`
- `InvReq`
- `DownReq`

In the standard MESI protocol, the directory serves data for `ShReq` requests from main memory, even when other caches have the line in Shared (S) or Exclusive (E) state. This is inefficient because caches in this machine can serve a (clean) copy of the data much faster than main memory.

Ben modifies the MESI protocol to serve data from caches whenever possible, as is done in 3-hop protocols. For simplicity, we focus on one particular case: modifying the protocol so that, when there are one or more read-only (S) sharers, an `ShReq` is served by one of the sharers.

To do this, the standard modification to a 4-hop MESI protocol to make into a 3-hop protocol is as follows: when the directory receives a `ShReq` and there are one or more sharers, it chooses one sharer and sends it a `FwdShReq` message. Once the sharer receives the message, it responds directly to the requesting cache with a `ShResp` message that has a clean copy of the data, and sends a `FwdShResp` to the directory to notify that it has forwarded the data.

Unfortunately, this change alone doesn't quite work, because *our protocol allows silent evictions*. We must also take care of the case when the cache silently dropped the line. To do this, a cache that receives a `FwdShReq` in I state responds to the directory with a new `IAck` message, notifying the directory that it does not hold the line. Upon receiving an `IAck,` the directory removes the cache from the sharer set, and sends a `FwdShReq` to another sharer. If all potential sharers reply with `IAcks`, the directory serves the line from main memory, using an `ShResp` message.

Question 2 (8 points)

(a) Assume that there are N caches participating in the coherence protocol. In the worst case, how many hops does it take for the data to arrive at the cache that sent the `ShReq`?

(b) Without adding more states or messages, can you propose a solution where we reduce the number of hops required in the above worst-case scenario? What is the downside of your solution?

Name ____________________________

As we saw in the previous question, silent evictions make forwarding tricky when there are multiple sharers. To solve this, Ben adds a Forward (F) state to his coherence protocol. The F state allows the same access permissions as S, i.e., read-only. In addition, an F-sharer is responsible for forwarding the data, so *it cannot silently drop the line*. When a line has multiple read-only copies, one of them holds it in F and the others in S. This way, most caches can enjoy the traffic savings of silent drops and, at the same time, the directory avoids asking multiple sharers for a forward.

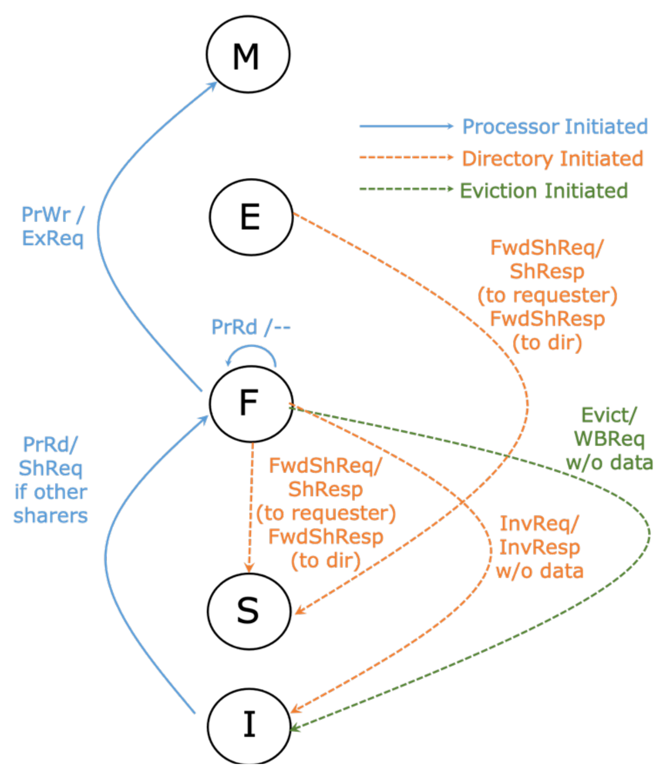
This protocol is called **MESIF**, and is one of the protocols used in Intel processors.

In more detail, the MESIF protocol has the following properties:

- Silent evictions are disallowed from the F state (as they are from E or M states).
- At most one cache holds a line in the F state, with other read-only sharers holding it in S.
- The directory explicitly tracks the cache that holds the line in F, and sends `FwdShReq`s only to that cache. If there is no F-state sharer, the directory serves data from main memory, and the new sharer becomes an F-state sharer.
- Forwarding operations transfer the F state along with the data: when a cache forwards read-only data from the F (or E) state, it transitions to the S state. Then, the cache receiving the `ShReq` transitions to the F state (not S).
- Given the previous two properties, the sharer that holds the line in F is *always* the one that requested the cache line most recently.

Note that while the E and F states have some similarities, a line in F state cannot silently transition to the M state since there can be other sharers with a read-only copy of the data.

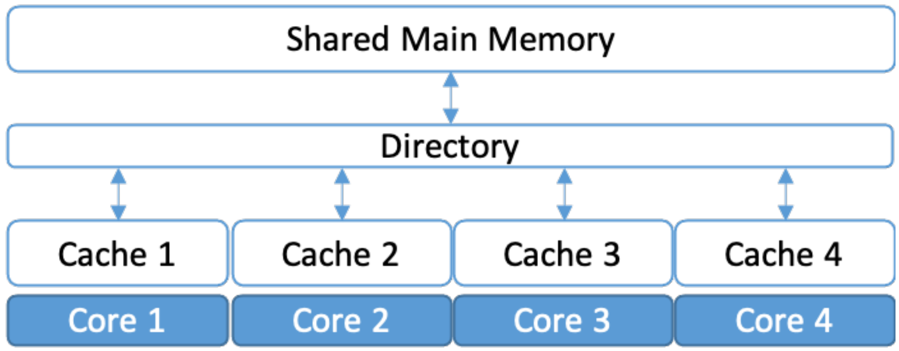
The state-transition diagram below shows the modifications to the protocol to implement MESIF. For clarity, the diagram only shows newly added transitions.



Name ______________________________

Question 3 (5 points)

Ben implements the MESIF protocol for his 4-core processor and runs the same set of memory accesses as in Question 1, shown below:

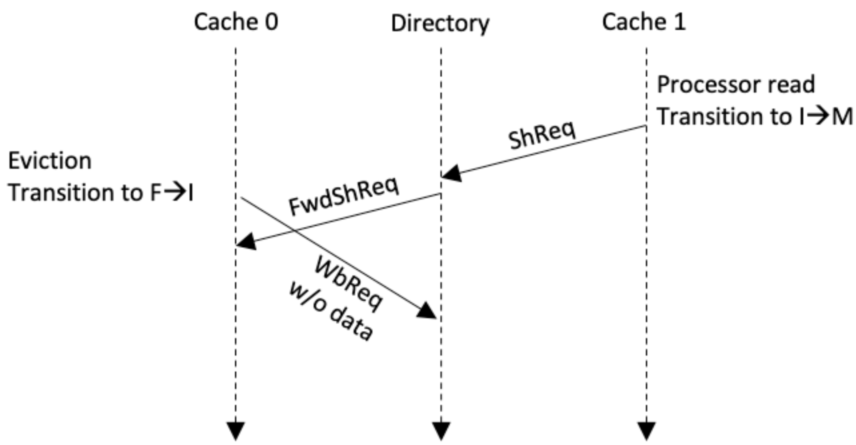


`(1) LD A`
`(2) ST A`
	`(3) LD B`
	
	`(6) ST B`
		`(4) LD B`	`(5) LD B`

How many `ShResp` messages are sent out with the MESIF protocol?

Question 4 (6 points)

So far, we assumed each coherence transaction completes before the next transaction begins. Alyssa P. Hacker thinks that Ben's MESIF protocol has additional races he should consider. A possible race scenario is as follows. The directory receives a `ShReq` from Core 1, and sends a `FwdShReq` to Core 0, which was holding the line in F state. Concurrently, Core 0 attempts to evict the line by sending a `WbReq` to the directory. Note that the network guarantees that messages arrive in the order they were sent between a source and destination pair.



To maintain coherence, what action should Cache 0 take in response to the `FwdShReq` while in the F→I transient state? Choose one of the three following answers (explain your choice briefly for partial credit):

A: Acknowledge the `FwdShReq` by sending a `ShResp` to Cache 1 and `FwdShResp` to the directory, transitioning to the S→I transient state. When the directory receives Cache 0's `WbReq`, it will infer that the `WbReq` and `FwdShResp` raced, and send a `WbResp` to Cache 0 so that it can transition to the I state.

B: Do nothing, since once the directory receives the `WbReq`, it will know that Cache 0 has evicted its own copy. The directory will then reply to Cache 0 with a `WbResp` (so it can transition to the I state) and serve the line to Cache 1 from memory.

C: Performing either of A or B will result in correct behavior.

Question 5 (6 points)

Our MESIF protocol implementation transfers the F state with forwarding responses, so that the most recent requester is always the F-state sharer. Ben thinks that this is not necessary: the cache performing the forwarding can simply stay in F state, responding to all future `FwdShReq` messages. What is a potential **downside** of this alternative implementation in terms of performance? Do not worry about the complexity of implementing transient states for this question.

Part B: Memory Consistency (20 points)

Consider a shared-memory machine that executes the following two threads on two different cores. Assume that memory locations `a`, `b`, and `c` contain initial value `0`.

T1	T2
T1.1: Store (a) ← 1 T1.2: Store (c) ← 1 T1.3: Load r2 ← (b)	T2.1: Store (b) ← 1 T2.2: Load r3 ← (c) T2.3: Load r1 ← (a)

Question 1 (5 points)

State all values of r1, r2, and r3 that can occur if the machine implements sequential consistency. *Note:* You can but *do not have to* express the result as (r1, r2, r3) tuples.

Question 2 (5 points)

Now assume the machine implements the Total Store Order (TSO) consistency model. Recall that TSO allows stores to be ordered after later loads. What execution outcomes can this code produce?

Question 3 (5 points)

Now assume the machine implements a relaxed consistency model (RMO), which allows loads and stores to be reordered after later loads and stores. What execution outcomes can this code produce?

Question 4 (5 points)

The relaxed consistency model (RMO) has the following fine-grained barrier instructions:

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

Add barrier instructions to T1 and T2 so that the RMO machine produces the same outputs as the SC machine for this code. Use the ***minimum*** number of memory barrier instructions. List the locations of each barrier below (e.g., "Add **$\texttt{MEMBAR}_{RR}$** after `T1.1`").

T1	**T2**

`T1.1: Store (a) ← 1`
`T1.2: Store (c) ← 1`
`T1.3: Load r2 ← (b)`
	`T2.1: Store (b) ← 1`
	`T2.2: Load r3 ← (c)`
	`T2.3: Load r1 ← (a)`

Name ______________________________

Part C: Synchronization (20 points)

Question 1 (10 points)

Ben wants to use atomic compare-and-swap (`CAS`) instructions. The code below describes the behavior of `CAS`:

```
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`. Otherwise, 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. (If you've seen other variants of `CAS`, note that this is an implementation of strong `CAS`.)

Ben wants to run his code on a MIPS processor that does not implement the `CAS` 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
```

Ben implements `CAS` with `LR` and `SC` as follows (Instructions are labeled `I1` through `I6` for your convenience)

```
     CAS rOld, rNew, Imm(rBase):
I1:           LR r1, Imm(rBase)     # r1 ← Memory[Imm + (rBase)]
I2:           BNE r1, rOld, _fail   # if (r1) != (rOld), goto _fail
I3:           ADDI r2, rNew, 0
I4:           SC r2, Imm(rBase)     # Memory[(rt) + Imm] ← (r2)
I5:           BNEZ r2, _success     # if (r2) == 1, goto _success
I6:  _fail:   ADDI rOld, r1, 0
     _success:
```

Alyssa points out that Ben's implementation has a bug. Describe how Ben's implementation can violate the semantics of `CAS`, and fix Ben's implementation such that it correctly implements `CAS`. You only need to write down the changes to the code you would make (e.g., replace `I3` with `ADDI, r2, rNew, 0`).

Question 2 (10 points)

Ben now wants to implement double compare-and-swap (DCAS), an atomic primitive that writes new values to two distinct (and not necessarily contiguous) memory locations if their old values match expected values:

```
DCAS rOld1, rOld2, rNew1, rNew2, Imm1(rBase1), Imm2(rBase2):
     old1 ← Memory[(rBase1) + Imm1]
     old2 ← Memory[(rBase2) + Imm2]
     If old1 == (rOld1) and old2 == (rOld2):
          Memory[(rBase1) + Imm1] ← (rNew1)
          Memory[(rBase2) + Imm2] ← (rNew2)
     else:
          rOld1 ← old1
          rOld2 ← old2
```

Do you think DCAS can be implemented with LR and SC? If so, write down your implementation below. If not, briefly explain why it is impossible.

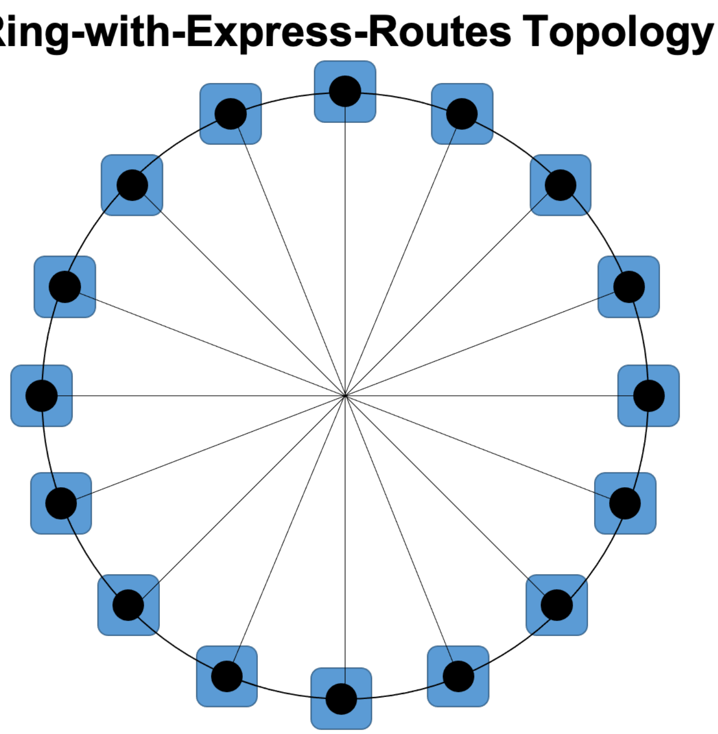
Name ______________________________

Part D: Networks (30 points)

Question 1 (10 points)

Consider the following Ring-with-Express-Routes Topology. On top of the usual ring topology, we add express routes that connect each node to the node farthest from it on the ring. Assume that the network has N nodes, where **N is a multiple of 4**.

Answer the questions below for the allowed values of N, i.e., multiples of 4, and not only for the case shown in the figure.



(a) How many total links does this network have?

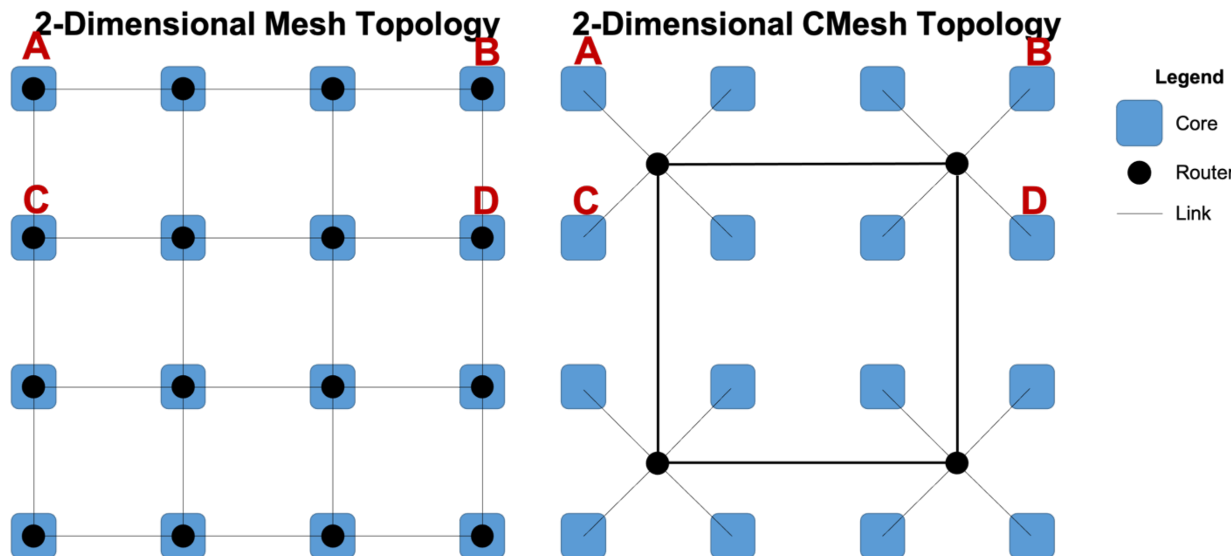
(b) What is the diameter of the network?

(c) What is the bisection bandwidth of the network?

(d) Express the average distance of the network in terms of Big-O notation with respect to N. For example, if you think that the average distance scales quadratically with respect to N, you can write $O(N^2)$. Assume uniform random traffic.

Name ___________________________

For the following questions, we will explore the design tradeoffs between two different network topologies: a 2-dimensional mesh and a 2-dimensional concentrated mesh (cmesh). The following figure shows the 16-core mesh and cmesh topologies:



Instead of assigning a single core per router, the cmesh topology allows four cores to share one router. Thus, there is a factor of 4 reduction in the number of routers needed.

Assume the following characteristics about the mesh and cmesh networks when under zero load:

- All links in the diagram represent 2 channels in opposite directions. Each channel has a throughput of 1 flit/cycle.
- Traversing the core-to-router or router-to-core channel takes 1 cycle.
- Traversing one router-to-router channel takes 1 cycle in the mesh topology, and 2 cycles in the cmesh topology.
- Traversing one router requires 2 cycles in the mesh topology, and 3 cycles in the cmesh topology.
- Packets are routed via XY-order routing.
- Both networks use virtual cut-through flow control. Recall that virtual cut-through allocates buffers and channels in packet granularities, but allows flits from a single packet to proceed immediately to the next channel without waiting for the rest of the packet's flits. This contrasts with store-and-forward, where the entire packet has to arrive at each intermediate router before proceeding.

Question 2 (6 points)

What is the minimum latency of sending a 4-flit packet from Core A to Core B for (a) the mesh topology, and (b) the cmesh topology? Assume that there is zero load, meaning that there are no other messages in the network. Note that you must include the cycles for all of the flits in a packet to arrive, not just the head flit.

Question 3 (7 points)

Now consider the scenario where Cores A and C both periodically send 4-flit packets to Cores B and D respectively. This can cause contention on the router-to-router channel for the cmesh network. By how much do you expect the latency of sending a packet from Core A to B to increase in the worst-case scenario compared to Question 2? Assume that routers arbitrate between packets from different input ports in a round-robin fashion.

Question 4 (7 points)

We observe increased latency for the cmesh topology in Question 3 due to both packets sharing a router-to-router link. How would you modify the hardware characteristics of the cmesh design without changing the topology such that we eliminate the latency increase?