# Computer System Architecture
# 6.5900 Quiz #2
# November 8th, 2024

Name: Solutions

This is a closed book, closed notes exam.
80 Minutes
14 Pages (+1 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.
- Page 15 is a scratch page. Use it if you need more space to answer one of the questions, or for rough work.

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

Name _____

# Part A: Broken Sequence Locks (25 Points)

A *sequence lock* is a synchronization primitive that allows *one* writer thread to "publish" data that can be consumed by *many* readers. Crucially, it has the property that the writer *should not block*, but it's alright if the readers block temporarily. The code below is *correct* if run on a system with **sequential** memory consistency. However, it is *not* guaranteed correct under looser consistency models. Code shown below:

```
typedef struct {
  volatile uint64_t seq;
  volatile object_t object;
} seqlock_t;

// only ever called from a single thread (the writer)
void publish(seqlock_t* seqlock, object_t* src) {
  atomic_add(&seqlock->seq, 1);
  memcpy(&seqlock->object, src, sizeof(object_t));
  atomic_add(&seqlock->seq, 1);
}

// can be called from any thread
void get_obj(seqlock_t* seqlock, object_t* dest) {
  while (1) {
    // store a copy of the sequence number
    uint64_t seq = seqlock->seq;

    // if it's odd, a write is in progress
    if (seq % 2 == 1) continue;
    memcpy(dest, seqlock->object, sizeof(object_t));

    // no intervening writes have occurred
    if (seqlock->seq == seq) break;
  }
}
```

Writers increment the sequence number *twice*. Once before the memcpy and once after. An odd sequence number implies that a write is in-progress. For this question, assume that object_t is some opaque type of size >16 bytes. Importantly, the goal of a sequence lock is that each reader gets the *exact bits* of an object that was previously published. Note: memcpy is not atomic!

**Note: on this system, atomic instructions *do not* include a memory barrier.**

Name _____

## *Question 1 (10 points)*

Suppose we use this sequence lock on a multicore system with **TSO** consistency. Would the code still be correct? If yes, please briefly justify why. If no, *precisely* describe what memory barriers would need to be inserted to fix it.

The code would still be correct under TSO. TSO only allows stores to be re-ordered after loads. get_obj() does not have any stores to shared data, so it is clearly correct.

publish() is also correct, but it requires more justification. Let's reason through the loads and stores it does to shared data (with atomics broken down into LD/ST pairs):
- LD seq <- cannot be reordered later due to RAW dep.
- ST seq <- cannot be reordered later because TSO doesn't allow ST-ST reordering
- ST object[0] ... object[n] <- cannot be reodered later because TSO doesn't allow ST-LD reordering
- LD seq <- cannot be reordered later due to RAW dep.
- ST seq

## *Question 2 (10 points)*

Suppose we use this sequence lock on a multicore system with **RMO** consistency. Would the code still be correct? If yes, please briefly justify why. If no, *precisely* describe what memory barriers would need to be inserted to fix it.

With RMO, we need to add fences in both publish() and get_obj(). Specifically, we need RR fences both before and after the memcpy() in get_obj() and WW fences before and after the memcpy() in publish.

Name _____

## *Question 3 (5 points)*

The code above is implemented with two atomic adds in the publish function. Assume a *correct* implementation on either TSO or RMO (whatever barriers are needed have been inserted). Are the atomic instructions necessary? Would using ordinary non-atomic increments instead be sufficient? Why or why not?

Non-atomic instructions are fine.

The problem states that only a single thread will call publish(). So, there will never be data races on &seqlock->seq. We can just regard the store to &seqlock->seq as the serialization point.

Name _____

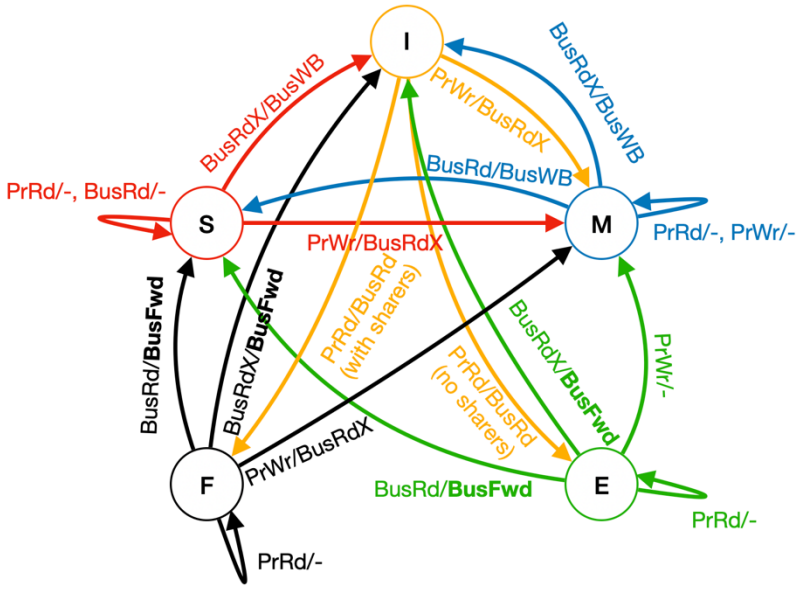
# Part B: MESIF (25 Points)

MESIF is a coherence protocol that extends MESI. In addition to the standard Modified (M), Exclusive (E), Shared (S), and Invalid (I) states, it adds a Forward (F) state.

*MESI* is inefficient when we have widely read-shared data. When a new processor requests a read-only copy of the data, we either need to serve it from the next level of the memory hierarchy or *all* the existing sharers would need to stampede to reply. *MESIF* addresses this by introducing an F state that can be viewed as S state with the additional responsibility of forwarding the data to the next read-only sharer.

For example, suppose that caches 0 and 1 currently hold clean copies of cache line 0x1000. Cache 0 holds it in F and cache 1 holds it in S. If cache 2 issues a BusRd request for 0x1000, then cache 0 is responsible for *forwarding* the (clean) data to cache 2 via a BusFwd message.

The state transition diagram for the MESIF protocol is shown here:




| Actions |
|---|
| Processor Read (PrRd) |
| Processor Write (PrWr) |
| Bus Read (Bus Rd) |
| Bus Read Exclusive (BusRdX) |
| Bus Writeback (BusWB) |
| Bus Forward Data (BusFwd) |

Name _____

## *Question 1 (5 points)*

Briefly explain the justification behind cache coherence protocols having an E state.

Programs often have thread-private read-modify-write data. Without a read state, a thread would need to issue a BusRdX request to upgrade read-only data (in the S state) to read/write data (in the M state). This creates unnecessary bus traffic in the common case that this thread is the only one that ever reads or writes this data. Instead, with an E state, we can silently upgrade to M without generating any additional bus traffic and incurring no additional latency.

## *Question 2 (10 points)*

Does the F state primarily benefit snooping coherence protocols or directory-based coherence protocols? Justify your answer.

The F state primarily benefits snooping protocols. Without an F state, data could either be forwarded from ALL sharers (causing a stampede with lots of bus traffic) or from NO sharers (and instead be served by the next level of the memory hierarchy, creating a lot of congestion there). Instead, with the F state, a single cache can serve sharing requests with BusFwd messages, avoiding both problems.

## *Question 3 (10 points)*

In the MESIF state-transition diagram shown above, the most *recent* sharer becomes the new forwarder. This is shown by the I→F and F→S arrows in the state transition diagram. Why might this be a desirable implementation choice?

Two main reasons (we accepted either):
- Load balance: switching the forwarder is desirable for load balance reasons. If the forwarder always remained the same, then for widely-shared data, one core would be very busy forwarding data, stopping it from doing other productive data. Changing the sharer spreads the load around more fairly.
- Minimizing F-state evictions: evicting a line in the F state is complicated. It can either be done silently (no forwarders, causes all the problems detailed in Q2) or noisily (requires an expensive protocol to select a new forwarder). Given the same principles as LRU caches, it is unlikely that the most recent sharer will evict anytime soon.

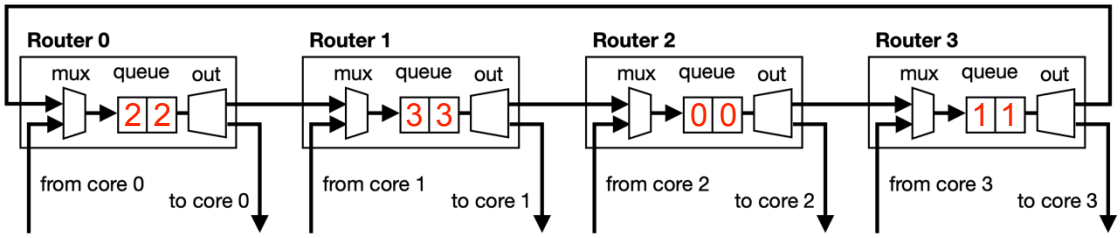
Name _____

# Part C: Ring NoC (25 Points)

## *Question 1 (8 points)*

Consider the ring network drawn below. Assume that all packets are 1 flit. As drawn, each router has a queue of size 2 flits. Each router works as follows. At each cycle:

- If there is an empty space in the queue, the mux will enqueue a new packet to the queue *either* from the previous router on the ring *or* from the core, based on logic you'll be asked to describe below.
- If the queue is not empty, the "out" stage inspects the destination of the packet at the head of the queue. If the destination matches the current router's ID, it dequeues the packet and pushes it out to the core. If it does not match *and* the next router in the ring can accept a new packet, then it dequeues it and sends it to the next router.




Please describe the input selection logic that will cause this network to deadlock, and explain how.

Please provide convincing evidence of deadlock as follows:

- List a set of packets (source, destination) that would cause deadlock. Assume the network starts from empty and each core can send 1 packet per cycle in the order you specify. There should be ≤ 12 total packets.
- Describe (preferably accompanied by a diagram) the state of the network when the deadlock occurs.

Selection logic: prioritize packets from the core over packets from the ring

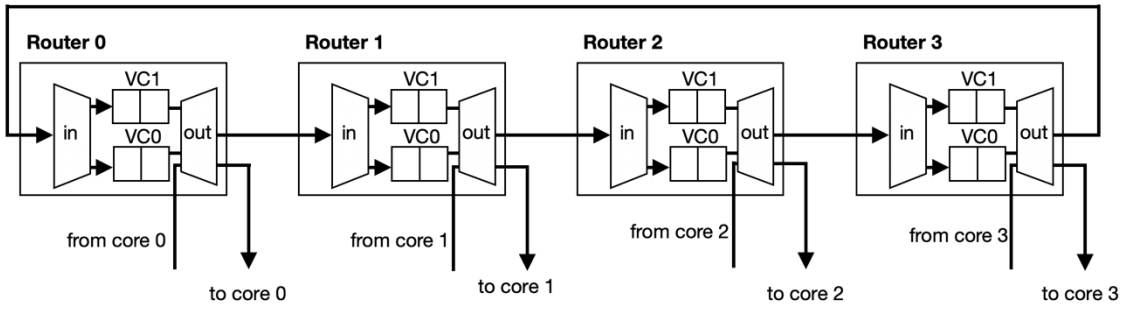
Deadlocking packets (many options work):
- Core 0: (0, 2), (0, 2)
- Core 1: (1, 3), (1, 3)
- Core 2: (2, 0), (2, 0)
- Core 3: (3, 1), (3, 1)

Explanation: see diagram above. In this state, no packet can make progress on the ring because each packet wants to hop to the next router, but the next router is blocked by its next router (circularly).

Name _____

## *Question 2 (8 points)*

We can try to fix this deadlock by introducing *2 virtual channels*, as shown below:



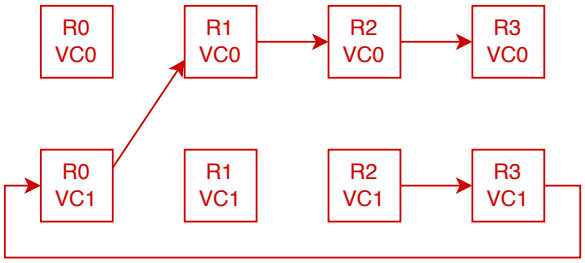

The system now has several modifications from the first part:

- Packets that are sent between routers are now tagged with a single bit to indicate which virtual channel they belong to.
- The "in" module *randomly* decides at every cycle whether to prioritize receiving a packet from the core or the ring.
- The "out" module *randomly* selects a packet from VC0, VC1, or the core, and routes it using the pseudocode shown below:

```
def route(cur_router, message):
  if cur_router.id == message.dest:
    return to_pe
  if cur_router.id > message.dest:
    return VC1
  if cur_router.id < message.dest:
    return VC0
```

Can this design deadlock or not? If it does, please list a sequence of packets; otherwise, prove why not. Drawing a channel dependency graph may be helpful. *Do not make assumptions about how the random decisions are made.*

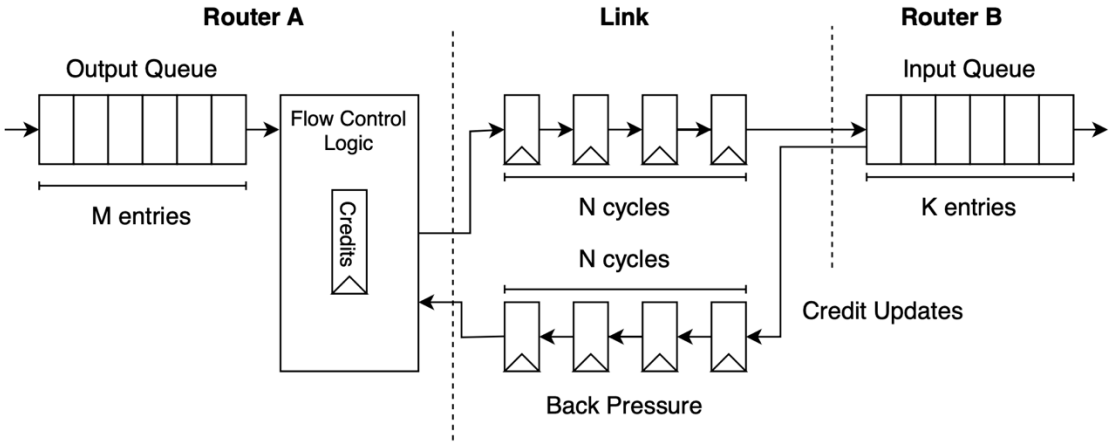
No, this cannot deadlock. The CDG is acyclic (and shown on the next page).


R0
VC0
R1
VC0
R2
VC0
R3
VC0
R0
VC1
R1
VC1
R2
VC1
R3
VC1

Name _____

## *Question 3 (9 points)*

The diagram below shows two routers, Router A and Router B, connected with an N-cycle fully pipelined link. Router A has an M-entry output queue, and Router B has a K-entry input queue. that receives a new flit every cycle whenever it is not full.




The routers use *credit-based* flow control, a technique that was discussed in lecture. Specifically, Router A sends packets over the link as long as its number of credits is >0. Whenever it sends a packet, its credit count is decremented. Whenever a new packet is *dequeued* from Router B, a credit becomes available, and Router B sends a *credit update* to Router A over an N-cycle fully-pipelined 1-bit backpressure link. When the credit update is received at Router A (N cycles later), the credit count is incremented. Note that Router B *does not* necessarily dequeue a flit every cycle.

Assume that Router A's output queue is never empty, i.e., it always has a flit available to send. Your task is to pick the **minimum** value of K (as a function of M and N) that ensures Router B *always has a flit ready to dequeue.* Please explain your reasoning.

K must be at least 2N.

Once a packet is dequeued from Router B's output queue, it takes 2N cycles to get a new packet from Router A. To cover up this latency, we need at least 2N entries in Router B's output queue.

Name _____

# Part D: Multithreaded Processor (25 Points)

Suppose that we have the following two functions, each being executed in their own thread:

```
#define SIZE (1 << 30)
char array[SIZE];
void thread1() {
  while (true) {
    for (int i = 0; i < SIZE; i += 64)
      array[i] += 1;
  }
}

void thread2() {
  int i = 1;
  while (true) {        // easy to predict! Loops forever
    int x = i++;
    while (x != 1) {    // medium hard to predict
      if (x & 0x1 == 0) // hard to predict!
        x = x >> 1;     // note: mathematically this is x/2
      else
        x = 3 * x + 1;
    }
  }
}
```

Assembly code for thread1:

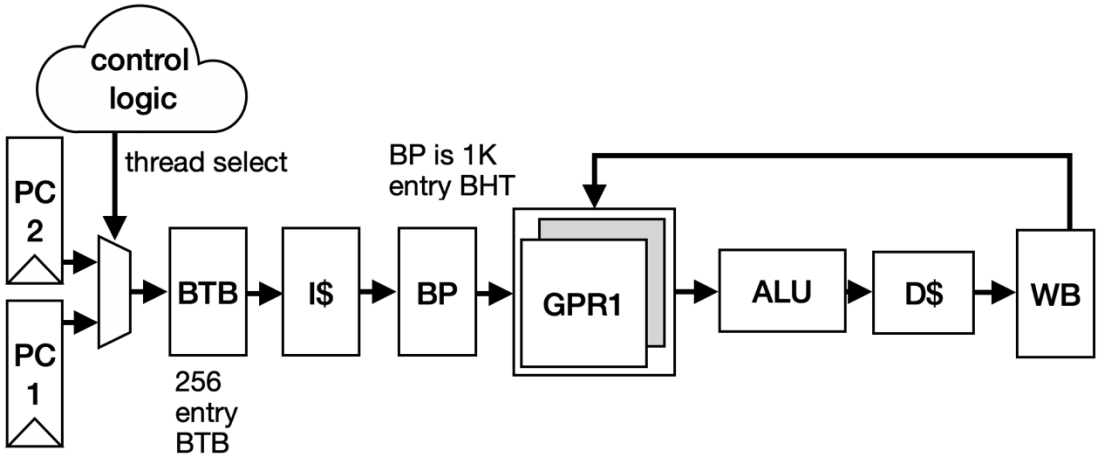
```
thread1():
// assume a1 holds array and a5 holds SIZE
.outer:
        mv      a2, a1
        li      a3, 1
.inner:
        lbu     a4, 0(a2)
        addi    a2, a2, 64
        addi    a3, a3, 1
        addi    a4, a4, 1
        sb      a4, -64(a2)
        bne     a3, a5, .inner
        j       .outer
```

Assembly code for thread2:

```
thread2():
        li      a4, 0
        li      a6, 1
        li      a3, 3
.outer:
        addi    a4, a4, 1
        mv      a1, a4
.inner:
        be      a1, a6, .outer
        andi    a2, a1, 1
        bne     a2, zero, .odd
        srai    a1, a1, 1
        j       .inner
.odd:
        mul     a1, a1, a3
        addi    a1, a1, 1
        j       .inner
```

We execute this code on the following **in-order, two-way coarse grain multithreaded** (CGMT) processor design shown below:

- Instruction and data cache hits take a single cycle
- Assume all code fits in the instruction cache and there are no instruction cache misses
- Data cache misses have 100-cycle miss latency
- Data cache is 32KB with 64B lines
- The ALU takes a single cycle
- The pipeline has full bypassing
- The branch misprediction penalty is 5 cycles. This means that the time from the fetch of the instruction following the mispredicted branch until the time of the fetch of the correct target is 5 cycles.



Name _____

## *Question 1 (5 points)*

If we were just executing each of these threads on its own *on this processor, without multithreading*, which thread would have higher IPC? Why?

thread2 would have higher IPC.

Cache misses (100 cycle penalty) are much more expensive than branch mispredictions. So, even though thread2 can mispredict ~2 branches per iteration, the penalty is much smaller, resulting in higher IPC.

## *Question 2 (10 points)*

Suppose that the core switches between threads on a cache miss. On a data cache miss, the processor flushes the pipeline and switches to the other thread, then continues executing the switched-to thread until *it* misses. The pipeline flush flushes both the load and all the instructions following the load, which can be up to 6 total instructions (including the load).

Would this system allow **both** threads to execute close to their peak single-threaded IPC in steady state? If not, how would you change the switching policy to have both threads achieve close to their peak single-threaded IPC?

No. thread2 has no memory instructions that could miss in cache. Therefore, once we switch to thread2, we would never have another cache miss again, starving thread1 entirely.

To fix this, we could switch on:
- both branch mispredicts and cache misses
- both cache misses and responses from memory
- cache misses and then switch back 100 cycles later

Name _____

## *Question 3 (5 points)*

We will now consider running multiple copies of thread1 on this CPU. To maximize IPC, we will need *more* than 2-way multithreading. Specifically, what is the minimum number of copies of thread1 that we need to run concurrently on this pipeline to maximize IPC?

Every time we have a cache miss, we will flush 6 instructions. These instructions need to be re-executed when we switch back to the thread (now the load will hopefully hit in cache!)

Each inner-loop iteration has 6 instructions. So, per loop iteration we have 6 instructions + 6 flushed instructions = 12 instructions per loop iteration. So, we need at least ceil(100 / 12) = 9 threads to maximize IPC.

## *Question 4 (5 points)*

Instead, suppose that we implement a policy of switching on branch instructions. There are several points in the pipeline where it may be reasonable to generate a "switch" signal for the multithreading control logic. Note that this is a CGMT implementation, so if we decide to switch at stage N in the pipeline, we *need* to flush stages 0, … N-1.

Suppose we now run multiple two copies of *thread2.* Which of these switching points would achieve the highest overall IPC?

- Switch on every branch (as predicted by BTB)
- Switch on every branch mispredict (detected in ALU)
- Switch on every branch (as determined in decode)
- Switch on every branch predicted taken (as determined by BP)

Explain your choice.

We should switch on every branch as predicted by the BTB.

Every time we switch, we flush all previous stages. So, switching as early in the pipeline as possible minimizes the amount of flushed work and maximizes IPC.

Name _____

## *Scratch Space*