Name: _____ Uniqname: _____

EECS 570 Winter 2022 Final Exam - **SOLUTIONS**

Total points: 100
Total time: 120 minutes
Closed book, closed notes.
Calculators are allowed, but no PDAs, Portables, Cell phones, etc.
For calculation questions, show your work and clearly indicate your final answers.
Use pen or a dark pencil.
Do not spend too much time on any one question.
State any reasonable assumptions that you need to make.
There are 12 pages in this exam. Please make sure that you have all pages.
Please try to write all your answers in the assigned spaces.
If you really need extra space, you may use the backs of pages (but please **clearly** indicate that you are doing so).

Please sign the Honor Code:

I have neither given nor received unauthorized aid on this examination, nor have I concealed any violations of the Honor Code.

_____

Q1	/10
Q2	/20
Q3	/20
Q4	/30
Q5	/20
Total	/100

1 Memory Consistency I (10 points)

(a) (5 points) Consider the following program.

Core 0	Core 1
(i1) x = 1	(i3) y = 1
(i2) r1 = y	(i4) r2 = x

Is the outcome `r1=0, r2=0` forbidden under SC? Circle Yes or No as appropriate.

Yes No

(b) (5 points) Consider the following program.

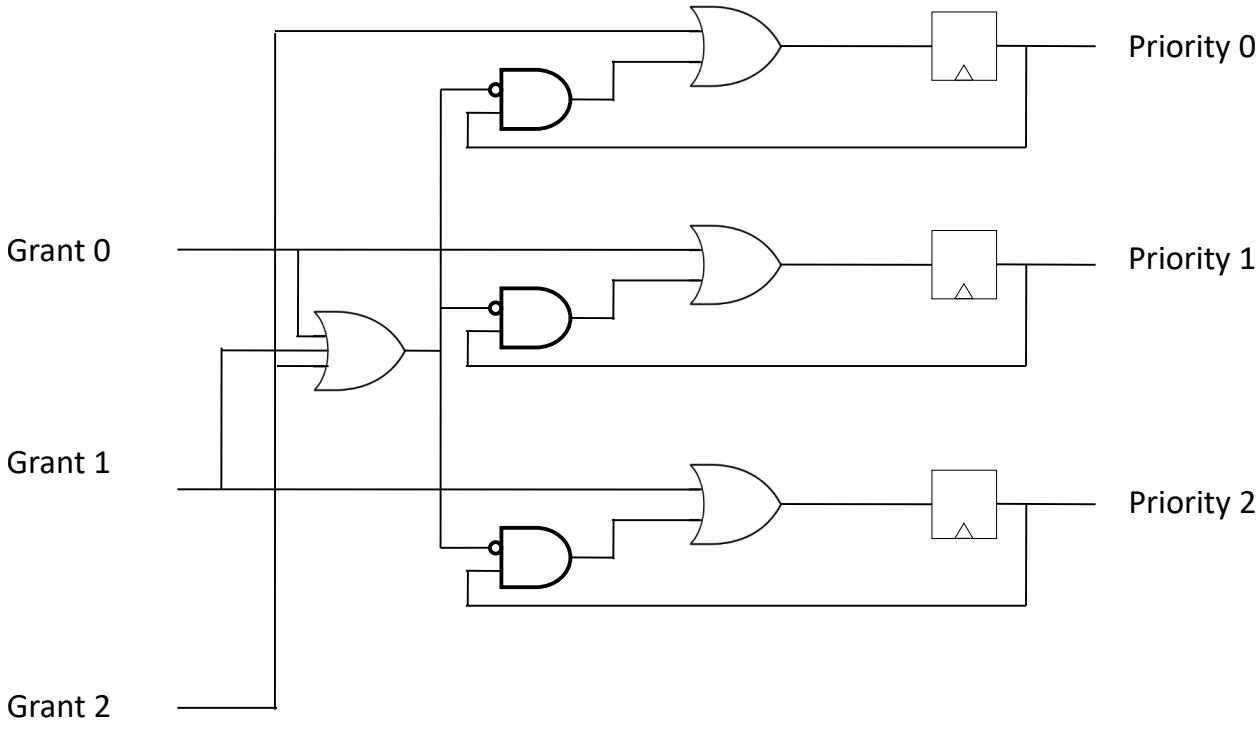
Core 0	Core 1
(i1) x = 1	(i3) r1 = y
(i2) y = 1	(i4) r2 = x

Is the outcome `r1=1, r2=0` forbidden under TSO? Circle Yes or No as appropriate.

Yes No

2 Interconnects I (20 points)

(a) (10 points) What is this circuit? Label its inputs and outputs (using the blanks provided), and briefly explain its operation.



Answer: This circuit is the priority generation circuitry for a round-robin arbiter. It uses the grant vector from the current cycle to generate the priority vector for the next cycle. If $Grant_i$ is asserted, then $Priority_{i+1}$ will be asserted in the next cycle, and all other entries in the priority vector will be 0. Thus, the request currently granted this cycle will have the lowest priority next cycle. If no *Grant* signals are asserted, the priorities will remain unchanged.

(b) (5 points) The Occamy is a parallel processor with an on-chip network. The Occamy's network is a 2-D mesh, and uses Valiant's routing algorithm with X-Y routing. The routing is implemented using source routing. Tom is an architect working on the Occamy. He notices that the use of Valiant's routing algorithm requires that the packet exit the network at the intermediate node, only to be re-injected into the network from the intermediate node to proceed to its destination. This seems like extra overhead to Tom, so he optimizes the network implementation as follows: instead of exiting the network at the intermediate node, a packet is simply routed first from its source to an input of the intermediate node, and directly from there to its destination. Shortly after making this change, Tom observes that the Occamy starts hanging occasionally and needs to be rebooted whenever it hangs. What is going on? Be sure to explain your answer.

Answer: When using Valiant's routing algorithm in combination with X-Y routing, deadlock can result if the packet does not exit the network at the intermediate node. For Valiant's routing algorithm with X-Y routing, the overall route is an X-Y plus X-Y route, with the first X-Y being to go from the source to the intermediate node, and the second X-Y to go from the intermediate node to the destination. If a packet does not need to travel any hops in the X-direction to get to the intermediate node, and does not need any hops in the Y direction to get from the intermediate node to the destination, then the overall route becomes a Y plus X route (i.e., traverse the Y direction, exit, be re-injected, and traverse the X direction). If the packet doesn't exit the network at the intermediate node, this becomes a Y-X route. We then have a network that essentially allows both X-Y and Y-X routes, which is known to lead to deadlock. This is what is happening.

(c) (5 points) A network implements credit-based flow control, with a 5-cycle router pipeline, with a credit pipeline delay of 2 cycles. The network uses buffers of size 10, with a flit propagation delay of 2 cycles and a credit propagation delay of 1 cycle. What is the buffer turnaround time for this network?

Answer: Turnaround time = Credit prop. delay + Credit pipeline delay + Router pipeline delay + Flit propagation delay
= 1 + 2 + 5 + 2 = 10 cycles.

—END OF QUESTION 2—

3 Fantastic Bugs and Where to Find Them (20 points)

Newt is working on the Thunderbird, a server-class multicore processor which implements TSO. The Thunderbird is running a web server. The Thunderbird currently runs workloads correctly, but it's a bit slow.

Answer the following questions. **Be sure to explain your answers.**

(a) (5 points) Newt wishes to improve the performance of the Thunderbird, so he adds a next-line prefetcher (for both instructions and data) to the processor. Puzzlingly to Newt, the modified processor's performance when running the web server remains almost exactly the same as before the modification. What is likely going on? (You may assume that Newt has implemented the next-line prefetcher correctly.) **The prefetcher remains in the processor for parts (b) and (c) of this question.**

Answer: A web server is a scale-out workload. Next-line prefetchers have been shown to be incapable of handling the large working set sizes/instruction footprints and access patterns/control flow of scale-out workloads [Ferdman12]. Thus, it's not surprising that the next-line prefetcher doesn't improve performance for such a workload.

(b) (5 points) Newt notices that the coherence protocol of the Thunderbird is rather simplistic, and stalls in a number of different scenarios. Newt optimizes the coherence protocol to reduce stalls, including eliminating the stall (by adding transient states) that occurs when a core receives an invalidation for a line before it receives the data for that line. However, upon running some benchmarks with the new protocol, Newt notices that certain loads in the program end up staying in the pipeline forever and never complete, despite generating lots of memory requests and receiving data for those memory requests. What is the reason for this?

Answer: Invalidation-before-use in a non-stalling protocol (which is what the protocol essentially is after Newt's modifications) can lead to livelock if it occurs repeatedly. This is what is happening here. The loads in question issue read requests, but they are invalidated before their data arrives due to another core's store, forcing the loading core to discard the data when it arrives and re-issue its load request. If this happens repeatedly, the load will not be able to complete and will stay in the pipeline forever. This is what is happening to the loads in question.

(c) (5 points) To fix the issue with the loads that don't finish, Newt decides to ensure that memory requests generated for loads can use the data provided to them (even if it's been invalidated) to service a load, as long as the cache discards the data immediately after the load operation completes. This does indeed fix the issue of loads never completing, but Newt now finds memory consistency violations (specifically, TSO violations) popping up in his benchmark runs, leaving him rather befuddled. What is the reason for these memory consistency violations?

Answer: This is an instance of the Peekaboo problem. Newt has added prefetching, a coherence protocol with invalidation-before-use, and a livelock-avoidance mechanism that allows the use of stale data to the Thunderbird. When combined, these three features can generate a consistency violation for programs like mp:

Core 0	**Core 1**
(i1) x = 1	(i3) r1 = y
(i2) y = 1	(i4) r2 = x

Here, if we prefetch x on core 1, and it's invalidated before use by core 0's store to x, and the stores on core 0 complete and the store of y is read by core 1 before the now-invalidated data for x reaches core 1 and is used for the load i4 (i.e., for one operation), then a consistency violation of TSO (the forbidden outcome of mp) can occur, even without OoO execution or pipeline reorderings.

(d) (5 points) What can Newt do to eliminate the memory consistency violations in (c) while also preventing the issue described in (b)?

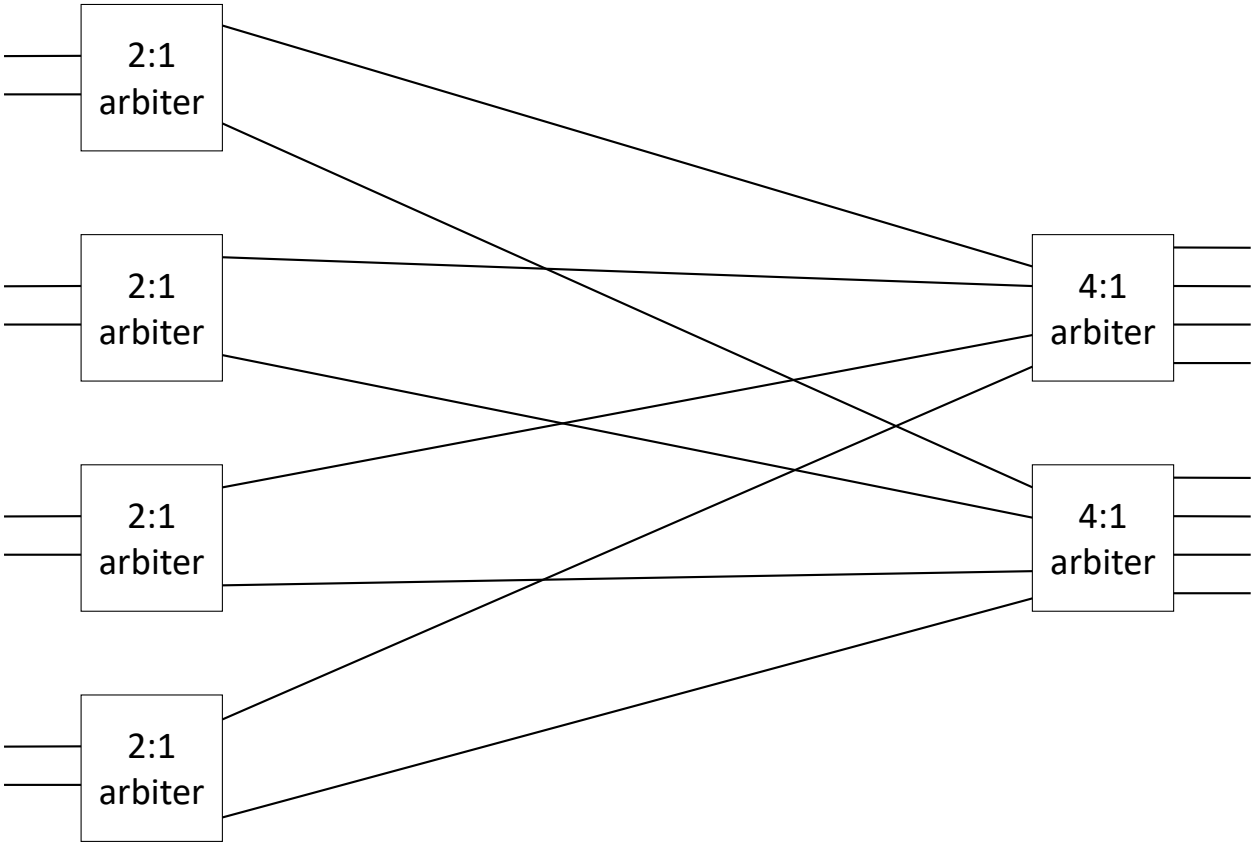
Answer: Newt can ensure that for any instruction like i4 in mp (henceforth called a 'Peekaboo instruction'), it is the oldest load or store in program order at the time the coherence request for it is issued. Prior loads must have performed and prior stores must have reached the memory hierarchy and become visible to all cores before the coherence request for the Peekaboo instruction is issued.

Alternate answer: remove the prefetcher, and issue memory requests in program order and one-at-a-time.

Another alternate answer: remove the prefetcher, the coherence optimizations, and the livelock-avoidance mechanism. (Not the nicest solution, but it works.)

4 Interconnects II (30 points)

(a) (10 points) The following diagram gives a skeleton of a 2:4 separable allocator (i.e., 2 requestors, 4 resources). Label the boxes and connect the missing wires to complete the diagram of the allocator. You do not need to label the wires.



(b) (5 points) SuperNet industries are developing a processor with an on-chip network, where routers have 4 ports. The network has good path diversity, and so multiple paths exist to every destination. Engineers at SuperNet want to develop a new feature called "turbo mode". In turbo mode, instead of transmitting at most one flit from one input to one output every cycle, the router attempts to send two flits from a given input to two different outputs in a single cycle (where each output corresponds to one of the paths to the packet's destination). You may assume that the flits are set up so that each flit contains the information necessary for its routing. You may also assume that flits are only taken from a maximum of 2 input ports every cycle (since up to 4 output ports will be needed to support turbo mode for flits from the 2 input ports).

Can the separable allocator from (a) be used by routers for switch allocation in such a network? Why or why not? Explain your answer.

Answer: No, the separable allocator from (a) cannot be used for this network. The separable allocator in (a) only allows each requestor to gain access to one resource each cycle. Thus, each input can only be granted at most one output by it, but turbo mode requires an input to be granted access to two outputs in the same cycle. This is not possible with the allocator from (a).

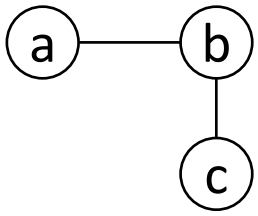
(c) (5 points) Consider an on-chip network with 3 virtual channels (VC0, VC1, and VC2). The cores (and directory) connected by the network use an MSI coherence protocol that does not expect point-to-point ordering—in other words, a protocol like your base MSI protocol for PA2. Requests travel on VC0, forwarded requests on VC1, and replies on VC2.

The network's routing algorithm is not deadlock-free. To alleviate the deadlock, an engineer proposes making VC1 an escape VC. Will this make the system deadlock-free (as far as the processor and network are concerned)? Why or why not? Explain your answer.

Answer: No, making VC1 an escape VC will just swap a routing deadlock for a protocol-level deadlock. In escape VCs, a packet always has a chance of moving to the escape VC. However, this would enable e.g., Data messages from VC2 to jump to VC1, and potentially be blocked by forwarded requests, e.g., a Fwd-GetM blocking a Data message, while the Data is needed to service the Fwd-GetM. Thus, the system will still deadlock.

(d) (5 points) A RISC-V startup, Fiver, is developing an MPSoC that has an on-chip network. They decide to use an irregular topology for their network. They begin with a 2D mesh that uses turn-based routing, and proceed by successively merging adjacent routers until doing so would violate performance or bandwidth constraints. Once the switch merging is complete, Fiver engineers run benchmarks on their model of the processor. To their surprise, they find that certain nodes are unable to send messages to each other despite a network path existing between the nodes. What is likely going on?

Answer: Turn-based routing will not work in a network if there is no path from a source to a destination that only uses the turns allowed by the routing algorithm. This can happen in an irregular topology. Consider this topology and the West-first routing algorithm:



Here, node c cannot reach node a under the West-first algorithm. This is likely the sort of phenomenon that is happening to Fiver's chip.

(e) (5 points) What is a solution to the issue that Fiver engineers are facing in (d)?

Answer: Fiver engineers can use a custom routing algorithm for their irregular topology, as long as they can prove it is deadlock-free for that topology.

Alternate answer: Fiver engineers could elect to not merge switches in their custom topology creation if doing so would make certain nodes unreachable from others under the turn-based routing algorithm they are using.

5 Memory Consistency II (20 points)

(a) (10 points) Engineers at Chips-R-Us are creating a multicore processor. Its pipelines are in-order and don't conduct speculative execution. Each core has its own FIFO store buffer from which it can read its own writes early. Only one store is allowed to be in-flight from a given core's store buffer to the memory hierarchy at any time. The cache hierarchy is private L1s and a shared L2.

The Chips-R-Us coherence protocol is based on MSI, but does not track sharers. On writes, the writing core obtains write permissions from the directory (ensuring a total order on all writes to a given address), but invalidations are not sent to current sharers. Instead, on any miss in its private L1, each core self-invalidates all Shared lines in its L1.Such self-invalidations also invalidate lines in transient states on the way to Shared (i.e., lines that are waiting for data responses to service loads). Cores do not use invalidated data. A fence results in a self-invalidation identical to that which occurs on an L1 miss.

Also note the following points:

- If a core has a copy of a line in Shared and wants to write to the line, it experiences a store miss. In response to its corresponding write request, the latest copy of the line is sent to the core along with write permissions.
- After a bounded number of reads to a Shared line in an L1, the line is evicted from that L1.
- You may ignore read-modify-write instructions for the purposes of this question.

Is Chips-R-Us's processor a valid implementation of TSO? If so, explain how the implementation maintains TSO's required orderings. If this implementation of TSO is buggy, describe a bug that could arise in it. (You may assume that the pipeline and store buffer are implemented correctly.)

Answer: Yes, Chips-R-Us's processor is a valid implementation of TSO. (It is a slightly simplified version of TSO-CC [Elver and Nagarjan HPCA 2014].) It ensures St-St ordering through its FIFO store buffer which sends stores to the memory hierarchy one at a time. Ld-St ordering is ensured by in-order execution of the pipeline and the FIFO-ness of the store buffer. To ensure Ld-Ld ordering, if a load observes a given value, then subsequent loads cannot observe any values that were stale (i.e., no longer the latest value) at the time the first load's value was written, otherwise the loads will appear to have been reordered. Self-invalidation on a miss ensures that all values that are stale with respect to that miss are eliminated from the private L1 which missed.

A fence must ensure St-Ld ordering. It does so by draining the store buffer and ensuring that subsequent loads after the fence cannot read stale values by self-invalidating all shared lines before those loads can execute.

(b) (10 points) **This question is not related to part (a).** Consider the following litmus test.

Core 0	Core 1
(i1) x = 1	(i4) r1 = y
(i2) FENCE	(i5) y = 2
(i3) y = 1	(i6) r2 = y
	<addr>
	(i7) r3 = x
Can r1=1, r2=2, r3=0?	

A microarchitecture maintains the following orderings:

- The `FENCE` (instruction `i2`) ensures that all cores observe `i1` before `i3`.
- The set of memory operations to a given address in the test obey SC with respect to each other. In other words, the outcomes of memory operations obey SC per location.
- As the test depicts, there is an address dependency between `i6` and `i7`. This dependency is preserved by the microarchitecture, i.e., `i6` and `i7` are executed in order.
- The microarchitecture implements rMCA write atomicity.
- The microarchitecture does not implement address prediction or value prediction.

Given these orderings, can the outcome `r1=1, r2=2, r3=0` be visible on this microarchitecture? If the outcome cannot occur, clearly explain why it cannot occur. If the outcome can occur, describe clearly how it could occur. (If you contend that the outcome can occur, you may not use obviously incorrect functionality—e.g., a buggy microarchitectural implementation of the FENCE operation—to cause it to occur.)

Answer: Yes, this outcome can occur on the microarchitecture. This test is essentially mp+dmb+fri-rfi-ctrlisb from [Alglave et al. TOPLAS 2014], which was found to be visible on certain ARM processors at the time. Here, i6 can execute and read its value (early) from i5 in the store buffer before i5 is made visible to all cores. i7 can then execute after i6, and both i6 and i7 can "commit early" before i5 is made visible to all cores and before i4 executes. This does not break per-location SC, as an "old" value of y will still be in the memory hierarchy. As long as i4 executes and reads from the memory hierarchy before i5 leaves the store buffer and is made visible to all other cores, i4 will get an "old" value of y and per-location SC will be maintained.

Thus, we can have the following execution:

i. i5 puts y = 2 in the store buffer.
ii. i6 reads r2 = 2 from the store buffer.
iii. i7 executes and reads r3 = 0.
iv. i6 and i7 commit.
v. i1 executes.
vi. The fence i2 ensures that i1 is made visible to all cores.
vii. i3 executes and is made visible to core 1.
viii. i4 reads r1 = 1 from the memory hierarchy and commits.
ix. i5 exits the store buffer and is made visible to all cores. i5 commits.