CS 152 Computer Architecture and Engineering

Final Exam
SOLUTIONS
May 10, 2021
Professor Krste Asanović

Name:________________________
SID:________________________

180 Minutes, 21 pages.

Notes:

- Not all questions are of equal difficulty, so look over the entire exam!
- Please carefully state any assumptions you make.
- Please write your name on every page in the exam.
- Do not discuss the exam with other students who haven't taken the exam.
- If you have inadvertently been exposed to an exam prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Question	Topic	Point Value
1	Iron Law	20
2	Virtual Memory	28
3	Pipelining and OoO	28
4	Vector Architectures	20
5	Cache Coherence	22
6	Memory Consistency	31
7	Synchronization	27
TOTAL		176

Problem 1: (20 points) Iron Law

Complete each of the following statements, referencing at least one component of the Iron Law in your answer.

Example: Adding caches might **decrease** time-per-program when ...

the code regularly hits in the cache, reducing the frequency of stalls and decreasing cycles-per-instruction.

A. **(4 pt)** Using interrupts for I/O instead of polling might **decrease** time-per-program when ...

I/O is very infrequent, so using interrupts instead of polling reduces instructions-per-program, as polling could execute many useless instructions

(ALT) Using interrupts for I/O instead of polling might **increase** time-per-program when ...

I/O is very frequent. any many cycles will be spent doing context switching, increasing cycles per instruction.

B. **(4 pt)** Applying trace scheduling might **decrease** time-per-program when ...

The code is statically scheduleable, so trace scheduling will increase ILP, decreasing cycles per instruction. Instructions per program will not decrease, because you have to execute at least the same amount of code as you do in the original program.

(ALT) Applying trace scheduling might **increase** time-per-program when ...

The code is inherently not statically scheduleable. Instructions per program would increase significantly due to much time spent fixing up work that shouldn't have been performed. CPI doesn't necessarily increase, because the trace-scheduled code could have been predicted well by a dynamic branch predictor.

C. **(4 pt)** Using coherent DMA to maintain I/O coherence instead of cache-flush instructions might **decrease** time-per-program when ...

The software is conservatively flushing every line in the DMA region with a non-coherent bus, greatly increasing instructions-per-program.

D. **(4 pt)** Using dynamic binary translation instead of a software interpreter for non-native code might **decrease** time-per-program when...

The program is used frequently, and the translated binary is kept cached, decreasing native instructions per program significantly compared to the software interpreted version.

E. **(4 pt)** A VLIW encoding scheme that compresses NOP fields in the instruction encoding might **decrease** time-per-program when...

NOPs are very common in the VLIW code, so the compressed form will take up less ICache space, reducing instruction cache miss rate, and reducing cycles per instruction.

Problem 2: (28 points) Virtual Memory

Consider a system which uses a two-level page-based virtual memory system.

- Pages are 16 bytes
- PTEs are 4 bytes
- Memory is byte-addressed
- The system is initialized with only the base page table allocated
- Physical pages are allocated from lower to higher PPNs incrementally
- The base page table is architecturally mandated to be at physical address 0x00, so a PTE containing value 0x00 is effectively an "invalid" PTE (no valid bit is necessary)
- The PTE is entirely reserved for a PPN (no valid, status, or permission bits)

2.A (12 pt) Paging Behavior

Fill out the contents of physical memory after value **0x6C** is written to virtual address **0x94**.
Fill out the contents of physical memory after value **0x94** is written to virtual address **0x6C**.
You only need to show the values of the memory locations that are written/changed.

Address	Value
0x00	
0x04	0x1
0x08	0x1
0x0c	
0x10	
0x14	0x2
0x18	0x2
0x1c	
0x20	
0x24	0x6C
0x28	
0x2c	0x94
0x30	
0x34	
0x38	
0x3c	
0x40	
0x44	
0x48	
0x4c	

2.B (4 pt) Virtual Address Space

What is the size of the virtual address space of this virtual memory system in bytes?

Virtual address is 8 bits (2 bits VPN0 + 2 bits VPN1 + 4 bits offset).
2^8 bytes = 256 bytes

2.C (4 pt) Physical Address Space

How much physical memory does this virtual memory system support?

32 bit PPN in PTE + 4 bit offset = 36 bits physical address
2^36 bytes = 64 GB

2.D (4 pt) VIPT L1

Explain briefly why **L1** caches are often designed to be **VIPT** (virtually indexed, physically tagged).

L1 cache accesses must be fast, so it is desirable to perform translation in parallel with cache access. VIPT enables the L1 cache to be indexed with the virtual address, and then the tag can be compared with the physical address after translation and tag read are complete.

2.E (4 pts) PIPT L2

Explain briefly why **L2** caches are often designed to be **PIPT** (physically indexed, physically tagged).

Primary reason is because L2 is usually accessed only on L1 miss, so the physical address would be available at this time.

Alternative reason could be to provide anti-aliasing protection to a VIPT L1 may contain aliases, but PIPT is still desirable for L2 even if L1 cannot contain aliases. However, PIPT by itself is not sufficient for this, there needs to be an additional mechanism for purging L1 caches of the aliased line.

Problem 3: (28 points) Pipelining and Out-of-Order Execution

3.A (12 pt) ROB Behavior

In this question, we consider a data-in-ROB design of an out-of-order core. For the following instructions, fill out the contents of the ROB after a large amount of time has passed, but the first load has not yet retrieved a value from memory.

```
0x800: li   t0, 0x4
0x804: lw   t1, 0(t0)
0x808: addi t1, t1, 0x4
0x80c: lw   t0, 0(t1)
```

Address 0x4 contains value 0x4 initially. The first row is partially completed for you.

IDX	PC	issued	completed	p1	src1	pd	dest	wbdata
0	0x800	Y	Y	Y	N/A	Y	t0	0x4
1	0x804	Y	N	Y	0x4	N	t1	
2	0x808	N	N	N	idx1	N	t1	
3	0x80c	N	N	N	idx2	N	t0	

```
0x800: li   t0, 0x4
0x804: lw   t1, 0(t0)
0x808: addi t0, t0, 0x4
0x80c: lw   t0, 0(t1)
```

IDX	PC	issued	completed	p1	src1	pd	dest	wbdata
0	0x800	Y	Y	Y	N/A	Y	t0	0x4
1	0x804	Y	N	Y	0x4	N	t1	
2	0x808	Y	Y	Y	0x4	Y	t0	0x8
3	0x80c	N	N	N	idx1	N	t0	

3.B (2 pt) PC in ROB

The PC field in the ROB is not used when instructions issue. What is the PC field used for?
PC is used to determine address at which an precise exception is seen, so the exception handler knows where to go to resume execution. PC is **not** used for rollback.
Although PC could be used as a source of data for some instructions which need the PC, the prompt says that PC is not used when instructions issue.

3.C (14 pt) Hazard Identification

For each of the following microarchitectural optimizations, circle the types of hazards that it addresses. Some optimizations may address multiple hazards, and some hazards may be addressed by multiple optimizations.

i. **Register renaming:**

RAW WAR WAW RAR Control Structural

ii. **Bypass paths:**

RAW WAR WAW RAR Control Structural

iii. **Branch prediction:**

RAW WAR WAW RAR Control Structural

iv. **Non-blocking data cache:**

RAW WAR WAW RAR Control Structural

v. **Load forwarding out of speculative store buffer**

RAW WAR WAW RAR Control Structural

vi. **Out-of-order execution:**

RAW WAR WAW RAR Control Structural

vii. **Fully pipelined functional units:**

RAW WAR WAW RAR Control Structural

Problem 4: (20 points) Vector Architectures

In this problem, we consider an algorithm for transposing a square matrix in-place by swapping rows and columns. The C code is provided below. The matrix elements are 32-bit integers.

```
void transpose(size_t n, int *mat) {
  for (size_t i = 0; i < n; i++) {
    for (size_t j = i + 1; j < n; j++) {
      int t = mat[(i*n)+j];
      mat[(i*n)+j] = mat[(j*n)+i];
      mat[(j*n)+i] = t;
    }
}
```

An abbreviated listing of potentially relevant vector load/store instructions is provided below.

Vector Load/Store Instructions	
`vle32.v vd, (rs1), vm`	`vd[i] = mem[(rs1) + i*4]`
`vse32.v vs3, (rs1), vm`	`mem[(rs1) + i*4] = vs3[i]`
`vlse32.v vd, (rs1), rs2, vm`	`vd[i] = mem[(rs1) + i*rs2]`
`vsse32.v vs3, (rs1), rs2, vm`	`mem[(rs1) + i*rs2] = vs3[i]`
`vluxei32.v vd, (rs1), vs2, vm`	`vd[i] = mem[(rs1) + vs2[i]] (unordered)`
`vsuxei32.v vs3, (rs1), vs2, vm`	`mem[(rs1) + vs2[i]] = vs3[i] (unordered)`
`vloxei32.v vd, (rs1), vs2, vm`	`vd[i] = mem[(rs1) + vs2[i]] (ordered)`
`vsoxei32.v vs3, (rs1), vs2, vm`	`mem[(rs1) + vs2[i]] = vs3[i] (ordered)`

4.A (12 pt) Vectorizing Transpose

Fill out the following vector code for vectorizing matrix transpose.

```
    # a0: n
    # a1: mat
transpose:
    li   t0, 1
    bleu a0, t0, end   # skip if n <= 1

    slli t0, a0, 2____ # initialize t0 with stride in bytes

    __________________ # optional line if needed

    addi a0, a0, -1    # decrement n

loop_i:
    mv   t2, a0        # number of elements to swap = n - (i+1)
    addi t3, a1, 4     # temporary pointer to row at mat[i][i+1]
    add  t4, a1, t0    # temporary pointer to column mat[i+1][i]
    addi a1, t4, 4     # bump mat pointer by (n + 1) elements

loop_j:
    vsetvli t5, t2, e32, m1, ta, ma

    vle32.v  v0, (t3)____ # vector load row mat[i][...]

    vlse32.v v1, (t4), t0 # vector load column mat[...][i]

    vsse32.v v0, (t4), t0 # vector store column mat[...][i]

    vse32.v  v1, (t3)____ # vector store row mat[i][...]

    sub  t2, t2, t5       # decrement vl
    mul  t6, t5, t0       # vl*stride in bytes
    slli t5, t5, 2        # vl in bytes
    add  t3, t3, t5       # bump row pointer
    add  t4, t4, t6       # bump column pointer
    bnez t2, loop_j

    addi a0, a0, -1       # decrement n
    bnez a0, loop_i

end:
    ret
```

4.B (4 pt) Vectors and Virtual Memory

Suppose n is very large (n > 1024). What is the minimum number of TLB entries as a function of the vector length (VL) that is necessary to avoid all non-compulsory TLB? Assume the page size is 4 KiB.

The original intent of this problem was to find the minimum size to avoid all non-compulsory TLB misses **within one iteration of the inner loop (loop_j)**. In this case, we need TLB entries for every entry in the column vector and every entry in the row vector, such that the vector loads and stores do not evict each other's entries. When n is large, each entry in a column vector will be on its own page. The number of pages for a row is at least ceil(VL / 1024) + 1. Thus, the minimum number of TLB entries with this interpretation is
VL + ceil(VL / 1024) + 1.

However, since the wording does not specify that compulsory misses across strip-mine iterations are allowed, an alternative interpretation is that there can be no compulsory misses across the entire algorithm. With this interpretation, we require that there can be no evictions across all iterations of loop_j. Following similar reasoning as before, the minimum number of TLB entries would be
n + ceil(n / 1024) + 1

4.C (4 pt) Reducing Cache Misses

Briefly explain how you could restructure the code to dramatically reduce the frequency of cache misses.

The ideal solution is to transpose small blocks of the matrix that fit in cache. In the native algorithm, the column-wise accesses touch only a single element of many cache lines, so cache misses will be frequent when n is large. The ideal blocking is to pick two blocks of the input array such that both blocks fit in cache.

Loop interchange by itself is not sufficient. No matter how you reorder the loops, one set of accesses will demonstrate a strided access pattern.

Problem 5: (22 points) Cache Coherence

5.A (6 pt) Out-of-order Coherence

Consider an out-of-order processor that implements conservative out-of-order load execution as discussed in lecture. A load is issued as soon as its address calculation is completed (potentially out of program order) and the following conditions are met:

- All addresses for older stores in the speculative store buffer are known.
- If the load address matches one of those entries in the speculative store buffer, the store data from the youngest store older than the load is available for bypassing.

i. **(4 pt)** This approach behaves correctly in a single-core system. How can this approach cause a coherence violation in a multi-core system?

Consider the following example:

Core 0	Core 1

`# Let a0 = a1 = A`
`3: lw t0, (a0)`
`1: lw t1, (a1)`
	`# Let a0 = A`
	`2: sw t0, (a0)`

Loads to the same address are not necessarily executed in program order within a core. It is possible for a younger load (label 1) to be issued first and observe an older value in memory, while the older load (label 3) is issued later and obtains a newer value written by a remote store (label 2).

Many incorrect answers claimed that, since the speculative store buffers are not shared between cores, a load from Core 1 to the same address would observe a different value than a load from Core 0 which simultaneously bypasses from a store in its local store buffer.

While possible, this is not problematic for coherence. Stores in the speculative store buffer might not actually commit and therefore should not be visible to other cores. If Core 0's store eventually does commit, then Core 1's load would logically precede that store and Core 0's load in the global memory order.

If software requires that Core 1 reads the value written by Core 0, the ordering must be enforced by some other mechanism, and this becomes a matter of memory consistency and synchronization rather than coherence.

Similarly, other incorrect answers claimed that bypassing by both cores would cause them to see a different order of writes to the same location, typically using the example:

Core 0	Core 1

`    li t0, 1`
`1: sw t0, (A)`
`3: lw t1, (A) # bypass 1`
	`    li t0, 2`
	`2: sw t0, (A)`
	`4: lw t1, (A) # bypass 2`

However, each core's load would in fact be ordered logically after the local store from which it bypassed but before the next remote store to the same address. Suppose Core 1's store (label 2) becomes visible before Core 0's store (label 1). From the perspective of the system, the global memory order by label would be 2 4 1 3, so there is no contradiction between what each core individually observes and the total order of writes.

ii. **(2 pt)** Propose a simple solution for the coherence problem discussed above.

Multiple feasible solutions could work:

- Do not execute loads out of program order
- Replay a load whenever its cache line becomes invalidated, either due to a remote store or voluntary eviction
- Replay all younger loads to the same address after issuing an older load
- Replay all loads in program order at commit time

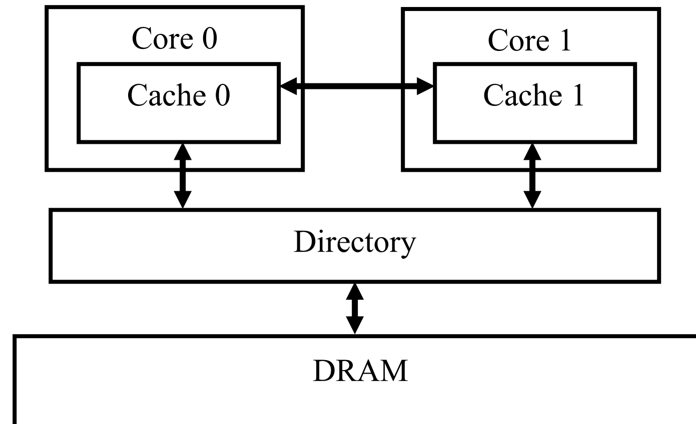
Loads should still bypass from the speculative store buffer on a match even if another core commits a store to the same location in the meantime, since the uncommitted local store represents a future write earlier in program order.

Note that dependent instructions do not strictly need to be re-executed after replaying a load if the newer load result is identical to the previous speculative value.

5.A (16 pt) Directory-based MOSI Coherence

Consider the baseline directory-based cache-coherence protocol discussed in Handout 6 (distributed with exam), which implements an MSI protocol. We consider extending that protocol to support MOSI coherence in a system which implements cache-to-cache links.

In the diagram of the adjusted system below, notice that DRAM is distinct from the directory.



To support the MOSI protocol in the directory-based system, we make the following modifications:

- New cache state **C-owned** for the O state in MOSI
  - If a cache line is in this state, the line is **dirty** and **read-only**, and the owning cache is responsible for providing data to other caches.
  - The C-owned state can only be entered from the C-modified state.
  - A single cache may have the line in C-owned while multiple other caches have the same line in C-shared.
- New directory state **O(id, dir)**
  - Cache <id> is the owner of the line, and all caches in *dir* are sharers.
- New message type **FwdShReq(Home, id, id`, a)**
  - This is sent from the directory to cache <id> when the directory is in the W or O state and has received a ShReq from cache <id`>.
  - When cache <id> receives this message, it moves the line to the C-owned state and sends ShRep directly to cache <id`>.
  - Note that FwdShReq subsumes WbReq/WbRep in the original MSI protocol.
- New message type **FwdExReq(Home, id, id`, a)**
  - This is sent from the directory to cache <id> when the directory is in the W or O state and has received an ExReq from cache <id`>.
  - When cache <id> receives this message, it invalidates its copy of the line and sends ExRep directly to cache <id`>.
  - Note that FwdExReq subsumes FlushReq/FlushRep in the original MSI protocol.
- Caches can send **ShRep(id, id`, data(a))** and **ExRep(id, id`, data(a))**.
  - These messages go through the cache-to-cache links, bypassing the directory.

i. **(12 pt)** Complete the table showing the sequence of transactions in this MOSI system. In each line, show the state of the caches and directories after the entire load/store has been completed. (Ignore transient states. Assume that every message is atomic).

Note: Handout 6 inconsistently uses both "C-modified" and "C-exclusive" to refer to the M cache state. Either is acceptable.

	Cache 0 State	**Cache 1 State**	**Directory State**	**Message(s) sent**
C0: read a	C-shared	C-nothing	R({0})	ShReq(0, Home, a) ShRep(Home, 0, data(a))
C0: write a	C-modified	C-nothing	W(0)	ExReq(0, Home, a) ExRep(Home, 0, data(a))
C1: read a	C-owned	C-shared	O(0, {1})	ShReq(1, Home, a) FwdShReq(Home, 0, 1, a) ShRep(0, 1, data(a))
C1: write a	C-nothing	C-modified	W(1)	ExReq(1, Home, a) FwdExReq(Home, 0, 1, a) ExRep(0, 1, data(a))
C1: evict a	C-nothing	C-nothing	R({})	FlushRep(1, Home, data(a))

Alternate:

	Cache 0 State	**Cache 1 State**	**Directory State**	**Message(s) sent**
C0: read a	C-shared	C-nothing	R({0})	ShReq(0, Home, a) ShRep(Home, 0, data(a))
C1: write a	C-nothing	C-modified	W(1)	ExReq(1, Home, a) InvReq(Home, 0, a) InvRep(0, Home, a) ExRep(Home, 1, data(a))
C0: write a	C-modified	C-nothing	W(0)	ExReq(0, Home, a) FwdExReq(Home, 1, 0, a) ExRep(1, 0, data(a))
C1: read a	C-owned	C-shared	O(0, {1})	ShReq(1, Home, a) FwdShReq(Home, 0, 1, a) ShRep(0, 1, data(a))
C0: evict a	C-nothing	C-shared	R({1})	FlushRep(0, Home, data(a))

ii. **(4 pt)** Describe a system in which this directory-based MOSI protocol would provide significant advantages compared to the baseline MSI protocol.

This protocol benefits a system in which DRAM latency is high and bandwidth is low.

- More data requests are satisfied by on-chip caches instead of far-slower DRAM.
- It eliminates the writeback message when another cache sends an ExReq. The dirty line can be passed among successive writers without updating memory.
- It eliminates a potentially unnecessary writeback when a line is written again by the owner after being shared.

Using a one-writer many-readers scenario as an example of an advantage is not completely sufficient without mentioning the cost of DRAM accesses. For subsequent ShReqs after the first ShReq following a write, forwarding a ShReq to the owner introduces one additional hop in the data response compared to the directory itself sending the ShRep in the R state.

Also note that this particular version of a MOSI protocol is *not* an update protocol which propagates writes automatically to existing sharers. As the C-Owned state is read-only, the cache must still transition to C-modified with an ExReq transaction and invalidate all other sharers before performing a write.

Problem 6: (31 points) Memory Consistency

6.A (16 pt) True/False

Indicate whether each statement is true or false and briefly explain your reasoning:

i. **(4 pt)** Sequential consistency is guaranteed if all processors have in-order pipelines.

False – The memory system is also responsible for enforcing sequential consistency, not only the core pipelines. Even with in-order issue and cache coherence, reordering of memory operations can arise from write buffers with bypassing, write-through and non-blocking caches, and a non-shared-bus interconnect fabric to separate memory banks.

ii. **(4 pt)** A high-level language with a sequentially consistent memory model can be implemented on an ISA with a weaker memory model if fence instructions are provided.

True – In the extreme case, the compiler can conservatively insert full fences between all memory operations to enforce strict program order within each thread. A more practical approach is to emit fences only as necessary or create locks implicitly around accesses to variables that the programmer identifies as being shared, typically with a keyword or another language feature. This guarantees SC executions for certain programs classified as data-race-free (all accesses to shared locations are properly synchronized).

Alternatively, false – It could be argued that local fences are *not* sufficient without assuming multi-copy atomicity. Whether SC can be implemented on top of a non-multi-copy-atomic memory model depends on the ISA providing global barriers to enforce ordering with respect to accesses in threads other than the thread issuing the barrier.

iii. **(4 pt)** Suppose an ISA specifies a non-multi-copy-atomic memory model, but a particular hardware implementation provides sequential consistency. Will software written for this ISA execute correctly on this machine?

Yes/True – As sequential consistency mandates multi-copy atomicity and is therefore a stronger form of memory consistency, SC executions are a proper subset of the execution outcomes permitted by the less restrictive non-multi-copy-atomic model. Software that is correctly written with the proper fences necessary to avoid data races under the weaker model would continue to execute correctly on this hardware implementation. The hardware implicitly enforces the same ordering constraints that fences would; the fence instructions would be treated by this implementation as NOPs.

iv. **(4 pt)** Suppose we have a sequentially consistent multi-core processor with a cache-coherent memory system. If we add a hardware prefetcher that prefetches directly into the L1 data caches, does the implementation still preserve sequential consistency?

Yes/True – If there is an intervening write by another core between the prefetch and the load, the prefetched line will be invalidated by the coherence protocol before the write, and the load will still observe the same value as if no prefetching occurred. (It is not sufficient to merely assert that coherence maintains consistency without explaining how stale prefetches are erased and that the cores continue to issue loads/stores in the same order, as coherence alone does not imply any particular memory consistency model.)

6.B (15 pt) Comparing Memory Models

For each of the following pairs of memory models, describe a hardware optimization that would be difficult to implement under the stricter model but easier to implement under the weaker model, and explain why.

Additionally, for the following code sequences, provide an example final result that would not be legal in the stricter model but would be legal in the weaker model. Variables A, B, and C are non-overlapping in memory and are initialized to 0.

Core 0	Core 1	Core 2

`li t3, 1`
`lw t1, (A)`
`sw t3, (B)`
`fence r, r`
`lw t2, (C)`
	`li t3, 2`
	`lw t1, (A)`
	`sw t3, (B)`
	`fence r, r`
	`lw t2, (C)`
		`li t2, 3`
		`sw t2, (C)`
		`lw t1, (B)`
		`sw t2, (A)`

All sequentially consistent results are listed in this table, which are not valid answers for any of the subsequent parts.

C0.t1	C0.t2	C1.t1	C1.t2	C2.t1
0	0	0	0	1
0	0	0	0	2
0	0	0	3	1
0	0	0	3	2
0	0	3	3	1
0	3	0	0	1
0	3	0	0	2
0	3	0	3	0
0	3	0	3	1
0	3	0	3	2

0	3	3	3	0
0	3	3	3	1
3	3	0	0	2
3	3	0	3	0
3	3	0	3	2
3	3	3	3	0

i. **(5 pt)** SC → TSO

Optimization: Private write buffers
TSO enables local buffering of stores with bypassing, since W→R ordering is relaxed.

C0.t1	C0.t2	C1.t1	C1.t2	C2.t1
0	0	0	0	0
0	0	0	3	0
0	0	3	3	0
0	3	0	0	0
3	3	0	0	0

ii. **(5 pt)** TSO → Weak multi-copy-atomic

Optimization: Out-of-order execution of loads and stores
Weak memory models relax all orderings between reads and writes, enabling aggressive out-of-order execution of loads and stores.

C0.t1	C0.t2	C1.t1	C1.t2	C2.t1
0	0	3	0	0
0	0	3	0	1
0	0	3	0	2
0	0	3	3	2
0	3	3	0	0
0	3	3	0	1
0	3	3	0	2
0	3	3	3	2
3	0	0	0	0
3	0	0	0	1
3	0	0	0	2
3	0	0	3	0
3	0	0	3	1
3	0	0	3	2

3	0	3	0	0
3	0	3	0	1
3	0	3	0	2
3	0	3	3	0
3	0	3	3	1
3	0	3	3	2
3	3	0	0	1
3	3	0	3	1
3	3	3	0	0
3	3	3	0	1
3	3	3	0	2
3	3	3	3	1
3	3	3	3	2

iii. **(~~5~~ 3 pt)** Weak multi-copy-atomic → Weak non-multi-copy-atomic

Optimization: Shared hierarchical buffers
Without multi-copy atomicity, a write can be partially visible to some cores without being visible to all cores.

As originally presented, this problem actually contains a subtle bug which prevents any new results from appearing in the weak non-multi-copy-atomic model, since all 48 possible outcomes already manifest under the weak multi-copy-atomic model. For non-multi-copy atomicity to produce an observable effect, the code must be amended slightly so that Cores 0 and 1 do not execute an identical sequence of memory operations.

The fixed code is a variation of the Independent Read Independent Write (IRIW) litmus test:

Core 0	Core 1	Core 2

`li t3, 1`
`lw t1, (A)`
`sw t3, (B)`
`fence r, r`
`lw t2, (C)`
	`li t3, 2`
	`lw t1, (C) # changed`
	`sw t3, (B)`
	`fence r, r`
	`lw t2, (A) # changed`
		`li t2, 3`
		`sw t2, (C)`
		`lw t1, (B)`
		`sw t2, (A)`

Then this result implies that Cores 0 and 1 do not observe the same order of stores to A and C:

C0.t1	C0.t2	C1.t1	C1.t2	C2.t1
3	0	3	0	0

Full credit was given for effort, and additional credit was given for realizing that no new results were possible compared to the weak multi-copy-atomic model.

Problem 7: (27 points) Synchronization

7.A (3 pt) Uniprocessor Atomics

Why would atomic synchronization instructions (AMO, CAS, LR/SC, etc.) be used in a single-core processor with no hardware multithreading?

Atomic primitives still benefit a system that uses preemptive multitasking to context switch between multiple processes on the same core. Processes, software threads, and interrupt handlers accessing shared memory can be synchronized without disabling interrupts in critical sections.

Unrelated to synchronization, vector AMOs can be used to vectorize some read-modify-write sequences with RAW memory dependencies, such as parallel reductions or histograms.

7.B (4 pt) Emulating LR/SC

Suppose we are attempting to perform binary translation of a program compiled for an ISA that provides only load-reserved/store-conditional (LR/SC) instructions into another ISA that provides only a compare-and-swap (CAS) instruction. Is the behavior of the translated instruction sequence on the right equivalent to the original instruction sequence on the left?

Assume that `x1` points to an aligned word and that `lw` reads the value atomically.

Original:

```
# x1: pointer
# x2: old value
# x3: new value
# x4: 0=success, 1=failure
retry:
  lr.w x2, (x1)
  ...    # compute x3 from x2
  sc.w x4, x3, (x1)
  bnez x4, retry
```

Translated:

```
# x1: pointer
# x2: old value
# x3: new value
# x4: 0=success, 1=failure
retry:
  lw x2, (x1)             # emulate lr.w
  ...                     # compute x3 from x2
  cas x4, (x1), x2, x3 # emulate sc.w
  bnez x4, retry
```

No, CAS cannot detect whether the memory location `x1` has been overwritten with the same value read by `lw`. Thus, there are situations where LR/SC fails since the intervening write clears the reservation, but CAS succeeds (i.e., the ABA problem).

There is another subtler difference in that the SC may still succeed even if the same thread writes a different value to `x1` with a regular store between the LR and SC (depending on the semantics of SC and how reservations are implemented), whereas CAS would fail the comparison if the value in memory is mutated.

7.C (11 pt) Optimizing LR/SC

Suppose we are looking to optimize LR/SC for a situation in which there are many readers and a few writers of a shared variable. We propose the following modifications to the cache-based implementation described in lecture under an MSI coherence protocol:

- The LR initially obtains the cache line in the shared state instead of the modified state.
- The reservation is cleared if the line is invalidated or if another store from the local core modifies the same address before the SC.
- If the reservation is intact, the SC attempts to upgrade the line from the shared state to the modified state. If the coherence transaction completes without an intervening invalidation, the SC succeeds.

i. **(4 pt)** How does this modification improve forward progress?

In the original implementation, the reservation is cleared whenever the line moves out of the modified state, which can happen if another sharer reads the line with a regular load. This alternative scheme allows readers to avoid disturbing the reservation before the SC is attempted, which improves the probability of success.

ii. **(4 pt)** For a constrained LR/SC instruction sequence in which no other loads and stores appear between the LR and SC and the shared variable is placed in its own cache line with no other variables, does this modification ensure that livelock cannot occur?

Yes, if a SC fails due to invalidation, then some other SC or store must have succeeded. The restrictions on the LR/SC sequence and variable placement prevent voluntary eviction and false sharing from invalidating the line and clearing the reservation.

Note that this question is specifically concerned about *livelock freedom*, not *starvation freedom* – an individual thread might never succeed at SC due to contention, but the system as a whole always makes progress somewhere.

iii. **(3 pt)** We would like to avoid the extra coherence traffic needed to upgrade the line from the shared to the modified state when no other readers access the line between the LR and SC. Could a similar optimization be applied to a MESI coherence protocol?

Yes: The LR acquires the line in the exclusive state. Downgrading the line from exclusive to shared does not cause the reservation to be cleared. The SC succeeds only if the line has either continuously existed in the exclusive state, which is then silently upgraded to modified, or if it is upgraded from shared to modified without an intervening invalidation.

This does reintroduce livelock as a possibility, but for the given use case, forward progress is still improved on average compared to the original MESI-based scheme from lecture. The relatively fewer writers (using LR) are less of a concern than the many readers (using regular loads), which now do not interfere with reservations.

7.D (9 pt) Parallel Reduction

Consider the following code which finds the index of the maximum element in an array of 32-bit integers. The work is split between multiple threads. Each thread first searches the array between the start and end indices uniquely assigned to it, and then conditionally updates the global index variable with its result.

```
    # a0: start index
    # a1: end index (non-inclusive)
    # a2: base address of array
    # t0: index of local maximum element
    # t1: value of local maximum element

    slli t2, a0, 2      # scale start index by element size
    add t2, a2, t2      # compute pointer to array[start]
    mv t0, a0           # initialize t0 to start index
    lw t1, 0(t2)        # initialize t1 to array[start]
    addi a0, a0, 1      # increment current index
    addi t2, t2, 4      # bump pointer

loop:
    lw t3, 0(t2)        # load current element
    addi t2, t2, 4      # bump pointer
    bge t1, t3, skip    # compare current element to local maximum
    mv t0, a0           # update local maximum index
    mv t1, t3           # update local maximum value
skip:
    addi a0, a0, 1      # update current index
    bltu a0, a1, loop

reduce:
    # TODO
```

i. **(4 pt)** The code is run on a single-issue vertically threaded processor. If threads are switched every cycle in a fixed round-robin schedule, what is the minimum number of threads required to avoid stalls for any input array? Ignore the prologue and final reduction, and consider only the steady-state execution of the loop over many iterations. Assume that loads have a 50-cycle latency, arithmetic instructions have a 1-cycle latency, and the latency for a taken branch is two cycles.

The longest stall is the load-use dependency from `lw` to `bge` in the same loop iteration. The `addi` between `lw` and `bge` hides one cycle of latency per thread. If thread 0 issues `lw` on cycle 0, `bge` will issue on cycle 2N, where N is the number of threads.

$2N \leq 50$
$N = \text{ceil}(50 / 2) = 25$ threads
25 threads is also sufficient to hide the overhead of taken branches.

ii. **(5 pt)** Write code to perform the final reduction atomically using LR/SC. If the local maximum element is greater than the current global maximum element, the thread updates the global index variable. You may use any available temporary registers.

- `t0` holds the index of the local maximum element found by the thread after the loop
- `t1` holds the value of the local maximum element found by the thread
- `a2` points to the base of the array
- `a3` points to the global index variable in memory

Assume that each thread maintains a separate reservation for LR/SC.

The first few instructions are provided for you.

```
reduce:
    lw t2, (a3)         # load global index
    slli t3, t2, 2      # scale global index by element size
    add t3, a2, t3      # compute pointer to global maximum
    lw t3, (t3)         # load value of global maximum

    # TODO: Finish reduction code

    bge t3, t1, done     # skip if global max >= local max
    lr.w t4, (a3)        # acquire reservation on global index
    bne t2, t4, reduce   # retry if global index has changed
    sc.w t2, t0, (a3)    # attempt to update global index
    bnez t2, reduce      # retry if update failed
done:
```

- The code is structured like a CAS operation to avoid placing the second `lw` in the middle of the LR/SC sequence, which could otherwise prevent forward progress. (Consider the possibility of a cache conflict that evicts the line corresponding to the reserved address.)
- Loading the global maximum value does not require an LR. Although shared, the input array is not modified by any thread.