

CS 152 Computer Architecture and Engineering

Final Exam **SOLUTIONS** May 12, 2020 Professor Krste Asanović

Name: _____

SID: _____

I am taking CS152 / CS252
(*circle one*)

180 Minutes, 27 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	Parallelism	32
2	Virtual Memory	21
3	Branch Prediction	28
4	Cache Coherence	24
5	Memory Consistency	30
6	Synchronization	25
TOTAL		160

Problem 1: Parallelism (32 points)

In this problem, we will explore how out-of-order processors, VLIW machines, and vector machines extract parallelism from the following code:

```
// Assume that N is large
for (i = 0; i < N; i++) {
    a = A[i];
    b = B[i];
    C[i] = (a * a) + (b * b);
}
```

Problem 1.A: Out-of-Order Execution (8 points)

This loop is translated into the following scalar code:

```
# a0 points to A
# a1 points to B
# a2 points to C
# a3 points to C+N
loop:
    fld f1, 0(a0)
    fld f2, 0(a1)
    fmul.d f1, f1, f1
    fmul.d f2, f2, f2
    fadd.d f1, f1, f2
    fsd f1, 0(a2)
    addi a0, a0, 8
    addi a1, a1, 8
    addi a2, a2, 8
    bltu a2, a3, loop
```

Consider an out-of-order processor with the following characteristics:

- Up to 6 instructions can be dispatched, issued, and committed per cycle
- 128-entry ROB
- 96-entry unified physical register file
- 2 integer ALUs, 1-cycle latency
- 2 load/store units, 2-cycle latency (assume all loads and stores hit in the cache)
- 1 floating-point adder, 2-cycle latency
- 1 floating-point multiplier, 3-cycle latency
- All functional units are fully pipelined
- Scheduler always selects the oldest ready instructions to issue
- Assume perfect branch prediction and memory disambiguation

What is the steady-state throughput in floating-point operations (FLOPs) per cycle? Count only arithmetic operations.

A common misconception is to divide by the minimum latency of one iteration ($fld \rightarrow fmul \rightarrow fadd \rightarrow fsd$), as if it were in-order issue. With dynamic scheduling, however, multiple iterations can be in flight simultaneously.

Since each loop iteration is independent of the others and all functional units are fully pipelined, the issue bandwidth is the true limiting factor. On average, a new loop iteration can be initiated at most every two cycles, due to these constraints:

- The four ALU operations must be issued over two cycles.
- The two fld and one fsd must be issued over two cycles as there are only two load/store units.
- The two $fmul$ must be issued over two cycles as there is only one multiplier.

The ROB size is sufficiently large compared to the latency through one iteration, so the oldest iteration should commit and have its ROB entries freed before the ROB is completely full.

Thus, the throughput is $3/2$ FLOPs per cycle.

Problem 1.B: VLIW (8 points)

Consider a VLIW machine with the following characteristics:

- 2 integer ALUs, 1-cycle latency
- 2 load/store units, 2-cycle latency
- 1 floating-point adder, 2-cycle latency
- 1 floating-point multiplier, 3-cycle latency
- All functional units are fully pipelined

Instructions are statically scheduled with no interlocks; all latencies are exposed in the ISA. All register operands are read before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

Schedule the operations from Part 1.A using software pipelining. You may rewrite the code to remove WAR/WAW hazards, but do not unroll the loop. Show only the software-pipelined loop; you **do not** need to include the prologue or epilogue. For each operation, also indicate which iteration that it belongs to ($i, i+1, i+2$, etc.).

What is the steady-state throughput in FLOPs per cycle?

Note: Not all rows may be needed.

ALU0	ALU1	MEM0	MEM1	FADD	FMUL
<code>addi a0,a0,8 [iter i+4]</code>	<code>addi a2,a2,8 [iter i+4]</code>	<code>fld f1,0(a0) [iter i+4]</code>	<code>fld f2,0(a1) [iter i+4]</code>	<code>fadd.d f5,f3,f4 [iter i+1]</code>	<code>fmul.d f3,f1,f1 [iter i+3]</code>
<code>addi a1,a1,8 [iter i+4]</code>	<code>bltu a2,a3,loop [iter i+4]</code>		<code>fsd f5,-40(15) [iter i]</code>		<code>fmul.d f4,f2,f2 [iter i+3]</code>

FLOPs per cycle:

$3/2$ FLOPs per cycle

The VLIW machine has the same mix of functional units as the OoO processor from Part 1.A, and so static scheduling should yield the same performance as dynamic scheduling under these conditions.

Problem 1.C: Vector Machines (8 points)

The loop is translated into the following vector assembly code:

```
# a0 points to A
# a1 points to B
# a2 points to C
# a3 holds N
loop:
    vsetvli t0, a3, e64
    vle.v v0, 0(a0)
    vfmul.vv v1, v0, v0
    vle.v v2, 0(a1)
    vfmul.vv v3, v2, v2
    vfadd.vv v4, v1, v3
    vse.v v4, 0(a2)
    sub a3, a3, t0
    slli t0, t0, 3
    add a0, a0, t0
    add a1, a1, t0
    add a2, a2, t0
    bnez a3, loop
```

For this question, consider a vector machine with the following characteristics:

- 16 elements per vector register
- 4 vector lanes
- 1 load/store unit per lane, 2-cycle latency
- 1 floating-point adder per lane, 2-cycle latency
- 1 floating-point multiplier per lane, 3-cycle latency
- All functional units are fully pipelined
- All functional units have dedicated read/write ports into the vector register file
- No dead time between vector instructions
- Vector instructions execute in order
- Scalar instructions execute separately on a decoupled control processor

First, we compare the performance of the vector processor with and without chaining. Vector chaining is performed through the vector register file. An element can be read on the same cycle that it is written back, or it can be read on any later cycle – the chaining is flexible.

However, with no chaining, a dependent vector instruction must stall until the previous vector instruction finishes writing back all elements. As an example, the pipeline timing would proceed as follows for two dependent `vfadd` instructions if not using chaining:

Instruction	1	2	3	4	5	6	7	8	9
<code>vfadd v2, v0, v1</code>	R	X1	X2	W					
		R	X1	X2	W				
			R	X1	X2	W			
				R	X1	X2	W		
<code>vfadd v4, v2, v3</code>								R	X1

Complete the following table for one stripmine iteration, which shows the cycle numbers at which each vector instruction begins execution (starting from the vector register read). The first column corresponds to the baseline vector design with no chaining. The second column adds flexible chaining to the processor. Assume that `v1` is set to the maximum vector length, and the first vector instruction executes in cycle 1. Ignore scalar instructions.

Instruction	Cycle number	
	Without chaining	With chaining
<code>vle v0</code>	1	1
<code>vfmul v1</code>	8	4
<code>vle v2</code>	9	5
<code>vfmul v3</code>	16	8
<code>vfadd v4</code>	24	12
<code>vse v4</code>	31	15

Assuming that the scalar instruction overhead of the stripmine loop is entirely hidden by the control processor executing separately, what is the FLOPs per cycle per lane with and without chaining? Consider the loop in steady state.

Note that with only one load/store unit per lane, `vle .v` of the next iteration must wait three more cycles for the last vector element group of the previous `vse .v` to proceed far enough through the pipeline (avoiding a structural hazard).

With chaining: $(16 \times 3)/(31 + 3)/4 = 6/17$

Without chaining: $(16 \times 3)/(15 + 3)/4 = 6/9$

Problem 1.D: Impact on Performance (4+4 points)

For each of the processors (OoO, VLIW, vector) introduced in Parts 1.A to 1.C, discuss how the following hardware changes would impact performance on the given loop. Assume that all other design parameters remain unchanged.

- (i) Doubling the register file size (i.e., doubling the size of the unified physical register file for OoO, doubling the number of architectural registers for VLIW, doubling the length of the vector registers). Does taking advantage of the expanded register file capacity require changing the code?

OoO: FLOPs/cycle is unchanged; performance is bottlenecked by structural hazards with the functional units, not lack of registers. Increasing the number of physical registers beyond the number of ROB entries also serves no useful purpose. No code changes are needed since renaming automatically exploits the larger physical register file.

VLIW: FLOPs/cycle is unchanged for the same reasons as the OoO case. The code must be rewritten to use the extra architectural registers now provided by the ISA.

Vector: FLOPs/cycle improves if chaining is used. Chaining longer vectors increases the proportion of cycles in which the adder and multiplier are simultaneously utilized. No code changes are needed if the code is written to be vector-length-agnostic.

- (ii) Adding another floating-point multiplier. Does taking advantage of this new functional unit require changing the code?

OoO: FLOPs/cycle is unchanged. Performance is still bottlenecked to the same degree by structural hazards with the ALUs and load/store units. No code changes are needed since the issue logic can automatically exploit the extra multiplier.

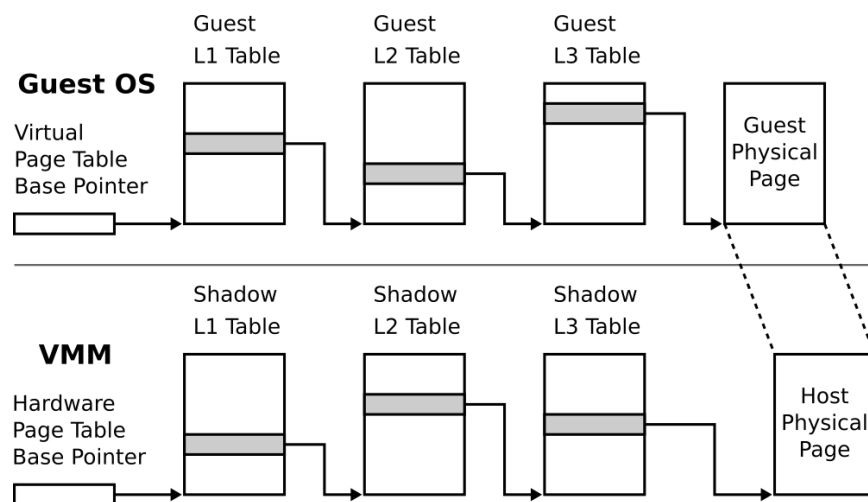
VLIW: FLOPs/cycle is unchanged for the same reasons as the OoO case. (Some improvement is possible if the instruction scheduling is initially suboptimal.) The code must be rewritten to use the extra VLIW slot.

Vector: FLOPs/cycle is unchanged. The lack of a second load unit in each lane prevents the two multipliers from being utilized simultaneously. No code changes are needed.

Problem 2: Virtual Memory and Virtualization (21 points)

A virtual machine monitor (VMM) runs several guest OSs on a single host machine. The guest OSs run in user (unprivileged) mode, whereas the VMM runs in supervisor (privileged) mode. The OS in each guest virtual machine manages its own set of page tables, which reflect the mapping of the guest virtual address space to the guest physical address space (“virtual-to-real”). The guest physical addresses must then be mapped to host physical addresses.

To reuse the hardware TLB, the VMM maintains a set of *shadow page tables* that map directly from the guest virtual address space to the host physical address space (“virtual-to-physical”). When running the guest OS in user mode, the VMM sets the hardware page table base pointer to point to the shadow page table. The TLB works as if there were no virtualization.



Problem 2.A: TLB Miss Latency (3 points)

Suppose that the guest and host machines both use three-level page tables. The host has a hardware-refilled TLB. When running the guest OS, what is the TLB miss latency if the TLB access takes 1 cycle and the memory latency is 50 cycles per access?

$$1 + 50 + 50 + 50 = 151 \text{ cycles}$$

A hardware TLB refill involves a page table walk of the shadow page tables in memory, same as if there were no virtualization in effect. Note that the guest page tables are not directly involved.

The TLB miss latency must also include the cost of the initial TLB lookup (1 cycle).

Problem 2.B: TLB Miss Latency Revisited (3 points)

Now suppose that the guest machine uses two-level page tables instead, while the host continues to use three-level page tables. When running the guest OS, what is the TLB miss latency if the TLB access takes 1 cycle and the memory latency is 50 cycles per access?

$1 + 50 + 50 + 50 = 151$ cycles

The shadow page table walk is agnostic to the structure of the guest page tables.

Problem 2.C: Page Table Base Pointers (5 points)

When the guest OS begins to run a guest OS user process, it attempts to change the page table base pointer to point to the guest page table of the process. Since the guest OS itself is running in unprivileged mode, this causes a trap into the VMM. What action does the VMM take when it encounters this trap?

On a write attempt to the page table base register, the VMM sets the hardware page table pointer to the corresponding L1 shadow page table and saves the guest version of the page table pointer elsewhere. If the VMM needs to allocate a new L1 shadow page table for a new process, it does not need to immediately initialize all shadow entries to mirror the guest page tables; the mappings can be populated lazily as page faults arise.

On a read attempt, the VMM returns the guest page table base pointer that was last recorded.

Problem 2.D: Page Table Updates (5 points)

From the perspective of the guest OS, the guest page tables live in guest physical memory. How does the VMM ensure that the shadow page table is updated when the corresponding guest page table is modified by the guest OS?

In the shadow page table, the VMM write-protects the pages holding the guest page tables so that any attempt to modify the guest page tables causes a trap into the VMM. On a trap, the VMM mirrors the changes to the guest page table entry in the shadow page table. The VMM must translate the guest physical addresses presented by the guest OS to host physical addresses; this typically requires the VMM to maintain a separate table with the real-to-physical mappings.

Problem 2.E: Different Page Sizes (5 points)

Describe how it is possible to support a guest virtual machine with an 8 KiB page size on a host machine with a 4 KiB page size.

Each guest page is mapped to two host pages; a leaf entry in the guest page table corresponds to two consecutive entries in the shadow page table with the same permissions. To properly support I/O (i.e., DMA requests), the two host pages should be physically contiguous.

Problem 3: Branch Prediction (28 points)

The following loop iterates through two arrays of integers and compares their elements. The code contains four branches labeled **B1**, **B2**, **B3**, and **B4**. Assume that the arrays X and Y are populated with uniformly random values.

<pre>c = 0; for (i = 0; i < N; i++) { // B4 x = X[i]; y = Y[i]; if (x == 0) // B1 c++; if (y == 0) // B2 c--; if (x != y) // B3 c += (x - y); }</pre>	<pre>la x1, X la x2, Y li x3, N li x4, 0 # c loop: lw x5, (x1) # x lw x6, (x2) # y bnez x5, skip1 # B1 addi x4, x4, 1 skip1: bnez x6, skip2 # B2 addi x4, x4, -1 skip2: beq x5, x6, skip3 # B3 sub x5, x5, x6 add x4, x4, x5 skip3: addi x1, x1, 4 addi x2, x2, 4 addi x3, x3, -1 bnez x3, loop # B4</pre>
---	---

Problem 3.A: Branch Correlation (2+2 points)

In contrast to *spatial correlation*, a branch may also demonstrate *temporal correlation* such that the present outcome of the branch is related to the previous outcomes of the same branch.

- (i) For the code above, briefly explain which branches exhibit spatial correlation, if any.

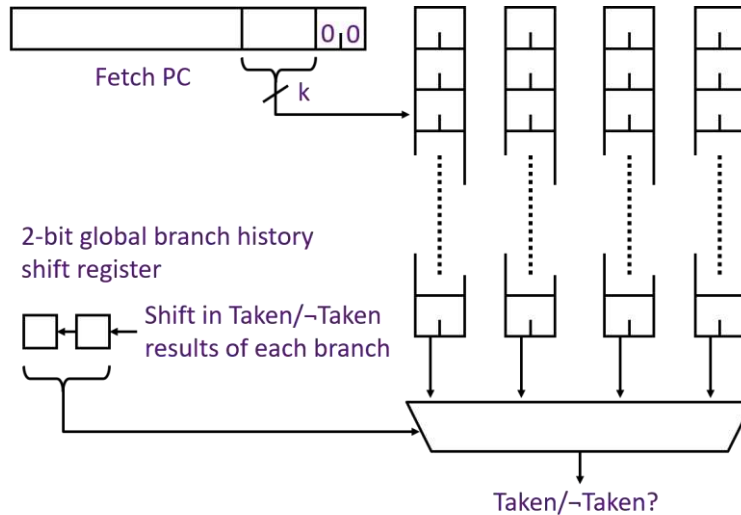
B3 is correlated with B1 and B2. For example, if B1 and B2 are both not taken, then B3 will always be taken.

- (ii) For the code above, briefly explain which branches exhibit temporal correlation, if any.

Only B4. B1, B2, and B3 are not temporally correlated since the value of each element is independent of the others.

Problem 3.B: Two-Level Predictors (8 points)

Consider the two-level branch predictor presented in lecture, which consists of four branch history tables (BHTs) indexed by PC. Each BHT entry contains a 2-bit saturating counter. The prediction is selected from one of the four ways based on two bits of global branch history. The outcome of the most recent branch is shifted into the global history register from right to left (1 for “taken” and 0 for “not-taken”).



The most significant bit of the bimodal counter provides the prediction: 1 for “taken” and 0 for “not-taken”. The counter that was used for the prediction is incremented if the branch is taken and decremented if not taken.

Complete the table on the following page for the first three iterations of the loop. The predictor columns show the state of the counters and the history register before the branch is resolved. For each row, only the counter value that was last updated needs to be written. Assume that:

- $N > 3$, and the contents of the arrays are $X = \{ 0, 1, 2, \dots \}$ and $Y = \{ 2, 0, +2, \dots \}$.
- All counters are initialized to the weakly “not-taken” state (01), and the global history is initialized to 00.
- Each branch is resolved before the next branch is predicted.
- The BHTs are large enough to avoid aliasing of PCs.

The first iteration has been done as an example.

Let $X = \{ 0, 1, 2, \dots \}$ and $Y = \{ 2, 0, 2, \dots \}$.

Loop		Branch Predictor					Branch Behavior	
Iteration	Branch	History	Way 00	Way 01	Way 10	Way 11	Predicted	Actual
0	B1	00	01	01	01	01	NT	NT
	B2	00	01	01	01	01	NT	T
	B3	01	01	01	01	01	NT	NT
	B4	10	01	01	01	01	NT	T
1	B1	01	00				NT	T
	B2	11	10				NT	NT
	B3	10		00			NT	NT
	B4	00			10		NT	T
2	B1	01		10			T	T
	B2	11				00	NT	T
	B3	11			00		NT	T
	B4	11	10				NT	T

Problem 3.C: Expected Accuracy (6 points)

Suppose that the elements in arrays X and Y are randomly and uniformly distributed over the set of integers 0, 1, and 2 (each possibility is equally likely). With the two-level predictor, what is the **expected accuracy** in predicting branch B3 correctly (`beq x5, x6, skip3`) as the loop approaches an infinite number of iterations? Show your work.

Hint: Consider the combined outcomes of branches B1 and B2, their probabilities, and how they contribute to the bimodal counters for B3. It may be helpful to separate the cases as such:

Conditions		Possibilities of (x, y)
$x \neq 0$	$y \neq 0$	(1, 1), (1, 2), (2, 1), (2, 2)
$x = 0$	$y \neq 0$	(0, 1), (0, 2)
$x \neq 0$	$y = 0$	(1, 0), (2, 0)
$x = 0$	$y = 0$	(0, 0)

For $\frac{3}{4}$ of the cases, the global history perfectly determines the outcome of B3 with 100% accuracy. In the remaining case ($x \neq 0$ and $y \neq 0$), both outcomes are equally likely, so the counter increments and decrements cancel out on average. Thus, the predictor does not learn anything useful and performs no better than a random guess, resulting in only 50% accuracy.

B1	B2	P(B1, B2)	P(B3 = T)	Prediction
T	T	4/9	0.5	50% accurate on average
NT	T	2/9	0.0	Always NT; counter converges to 00
T	NT	2/9	0.0	Always NT; counter converges to 00
NT	NT	1/9	1.0	Always T; counter converges to 11

The expected prediction accuracy is therefore:

$$(1/9 + 2/9 + 2/9)(100\%) + (4/9)(50\%) = 7/9 \approx 77.7\%$$

Problem 3.D: Trace Scheduling (5 points)

Now consider a different microarchitecture without a dynamic branch predictor. The processor statically predicts that branches are never taken, and taken branches incur a multi-cycle penalty.

Although originally conceived in a VLIW context, trace scheduling is a general compiler technique for removing control hazards that can also be applied to conventional scalar architectures. Assuming the contents of arrays X and Y follow the same uniform distribution as Part 3.C (all elements are equally likely to be either 0, 1, or 2), reschedule the assembly code to minimize the branch penalty along the most frequently executed code path.

```

    la x1, X
    la x2, Y
    li x3, N
    li x4, 0          # c
loop:
    lw x5, (x1)      # x
    lw x6, (x2)      # y
    beqz x5, B1      # B1
cont1:
    beqz x6, B2      # B2
cont2:
    beq x5, x6, skip3 # B3
    sub x5, x5, x6
    add x4, x4, x5
skip3:
    addi x1, x1, 4
    addi x2, x2, 4
    addi x3, x3, -1
    bnez x3, loop    # B4
B1:
    addi x4, x4, 1
    j cont1
B2:
    addi x4, x4, -1
    j cont2

```

Problem 3.E: Predication (5 points)

In high-performance processor implementations, one hardware technique to reduce the impact of frequent branch mispredictions is to internally convert short forward branches into sequences of predicated operations. This can be performed at the microarchitectural level without requiring any software modifications. For example, the original instruction sequence on the left can be executed as the predicated form on the right, where p1 represents an internal predicate register.

<pre>bnez x5, skip addi x4, x4, 1 skip:</pre>	<pre>sneq p1, x5, x0 addi x4, x4, 1, p1.t</pre>
---	---

However, this scheme nonetheless incurs some overhead in that the predicated operation would still be executed as a NOP even when the predicate is false, whereas a correctly predicted taken branch would avoid fetching and executing that instruction.

For the above sequence, how accurate does the branch predictor have to be in the non-predicated version to attain the same performance (latency) as the predicated version? Assume that the branch is taken 50% of the time, bnez takes 1 cycle to execute, and the misprediction penalty is 3 cycles. All other operations (addi, sneq) also each take 1 cycle to execute.

Hint: The average latency of this sequence would be $1+(1)(0.5) = 1.5$ cycles in the cases where the branch is correctly predicted and $3+(1)(0.5) = 3.5$ cycles if mispredicted.

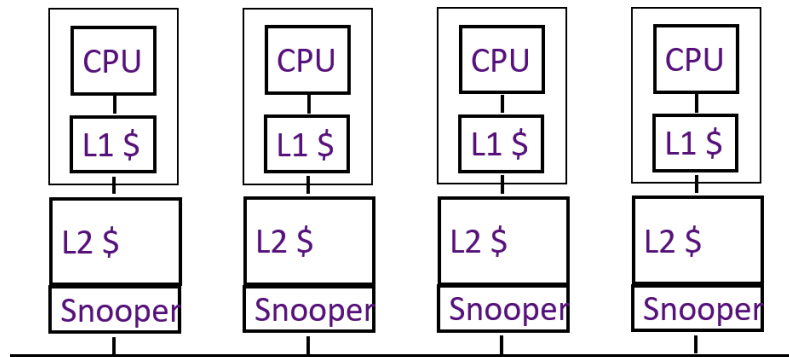
Let x represent the branch prediction accuracy.

$$\begin{aligned}1.5x + (3.5)(1 - x) &= 2 \\1.5x + 3.5 - 3.5x &= 2 \\-2x &= -1.5 \\x &= 3/4 = 75\%\end{aligned}$$

Problem 4: Cache Coherence (24 points)

Problem 4.A: Inclusion Policy (4+4 points)

In lecture, it was mentioned that an inclusive L2 cache can act as a filter to reduce the amount of L1 coherence traffic in a snoopy cache-coherence protocol. If a coherence request misses in the L2 cache, there is no need to probe the L1 cache for the given line.



- (i) Explain how a strictly exclusive L2 cache can also be used to optimize snooping by the L1 cache.

If a coherence request hits in the strictly exclusive L2 cache, the given line cannot be in the L1 cache, so there is no need to probe the L1.

Another acceptable answer is that coherence requests to upgrade permission for lines already present in the L1 (e.g., S to M) do not need to be broadcasted to the L2.

- (ii) Could a non-inclusive, non-exclusive L2 cache (i.e., neither strictly inclusive nor strictly exclusive) be similarly used to optimize snooping by the L1 cache? Explain.

No, hits and misses at the L2 cache reveal no information about what the L1 cache contains, so the L1 must snoop every coherence request.

A “yes” answer is also acceptable if it sufficiently explains how the L2 can track inclusivity with the L1 (e.g., an extra bit per L2 line to indicate it is shared with the L1).

Problem 4.B: False Sharing (4 points)

In the following table, indicate which memory operations experience a hit, true sharing miss, or false sharing miss under an MSI coherence protocol. Assume that x1 and x2 reside in the same cache line, and both words are read by both processors P1 and P2 before this sequence. The first row has been completed for you.

Time	P1	P2	Hit	True Sharing Miss	False Sharing Miss
1		write x1		X	
2	write x2			X	
3		read x1			X
4	read x1		X		
5	write x2				X

Problem 4.C: Directory-Based Coherence (6+6 points)

The following questions explore the directory-based coherence protocol described in Appendix A (same as Handout #6 from Problem Set 5) in more detail.

As before, assume that message passing maintains FIFO order: All messages between the same source and destination are always received in the same order that they were sent. Also assume that each site has sufficient queuing capacity to buffer all incoming messages without drops.

- (i) Consider the situation where a cache is sent an InvReq message for a given cache line. This occurs only if the directory state indicates that the site is a current sharer of the memory block, and the directory intends to invalidate the copy in the cache before granting exclusive access to another cache.

Typically, one expects the line to be in the *C-shared* state when the InvReq arrives. How is it also possible for the cache to receive the InvReq message while it has the line in the *C-pending* state (row #22 in Table H12-1 of Appendix A) – in other words, when the line is not actually present? Why does ignoring InvReq work out correctly in this case?

Assume that the home directory state is initially $R(id')$, indicating that the block is shared by the cache at site id' . Consider the following scenario:

1. The directory receives an ExReq from a site other than id' . The directory sends an InvReq to site id' . The home directory state becomes $Tw(id')$.
2. Before the InvReq arrives at site id' , the cache performs a voluntary invalidation to evict the cache line. The cache line state moves from *C-shared* to *C-nothing*.
3. The processor then issues a load or store to the evicted line, causing the cache to send a ShReq or ExReq. The cache line state becomes *C-pending*.
4. The InvReq associated with the first ExReq eventually arrives at site id' .

Note that the directory is waiting for an invalidation acknowledgement from site id' to complete the first ExReq; unless received, the directory will remain indefinitely in the Tw(id') state. However, the cache at site id' can safely ignore the InvReq here since it already sent an InvRep for the line as part of the voluntary invalidation (step 2), so deadlock is avoided.

- (ii) Consider the case where a cache requests a line in the *C-exclusive* state (ExReq) when the line is clean and shared by other caches. To reduce the response latency, it is a tempting idea to send the ExRep message with the data to the requestor in parallel to sending InvReq to the other caches. Does this optimization work correctly?

A convincing argument for either side could be made depending on the assumptions.

Since the interconnect guarantees FIFO delivery, the proposed optimization is safe from a **coherence** perspective. Recall that coherence can be defined with two invariants:

1. Writes to the same memory location are serialized (i.e., observed in the same order by all processors).
2. All writes eventually become visible to all processors.

Most responses asserting that this optimization breaks coherence do so primarily based on these common objections:

- *The protocol misbehaves if the directory receives another ExReq before the first round of InvReqs are acknowledged (e.g., more than one site ends up in the exclusive state).*

This situation of overlapping ExReqs is handled in the same manner without the optimization, as this optimization applies only if the line is clean and in the R(dir) state. The second ExReq will be queued until the directory receives all InvReps for the first, and then the directory will send a FlushReq to the current owner. Since messages are received in FIFO order, the FlushReq will arrive after the first ExRep, and so the FlushReq will contain the recently written data. The other site will not transition to the C-exclusive state until it receives ExRep, which the directory will not send until it obtains FlushRep. Thus, writes are serialized in the order by which their respective ExReqs arrive at the directory, upholding the first coherence invariant.

- *There is a window during which other caches can still read the “stale” data if the new owner receives the ExRep early and modifies its copy before all InvReqs are received elsewhere.*

While possible, this is not necessarily problematic. Although these operations can overlap in real time, the reads are treated as logically preceding the write in the global memory order. Coherence does not require that writes be “immediately” visible. All other caches will eventually receive an InvReq, and subsequent reads will trigger a ShReq that returns the updated data, so the second coherence invariant continues to be upheld.

Conversely, it can be argued that this proposed optimization is unsafe from a **consistency** perspective since the store-load reordering described above may lead to a violation of a stricter memory consistency model. This is potentially the case when multiple directory sites are involved (multi-bank last-level caches).

Problem 5: Memory Consistency (30 points)

Problem 5.A: Load/Store Queues (4+4+4 points)

Consider a multiprocessor with out-of-order cores that implement conservative out-of-order load/store execution (loads wait for memory addresses to be fully checked/disambiguated).

Table 2.1 shows the current state of the store queue in one of the cores. Stores are kept in the store queue until they commit. The instruction number indicates the order of the instructions in the program, with lower numbers being earlier in program order.

Table 2.2 shows the values present in the non-blocking data cache of the same core. Loads following cache misses can read from the data cache on a hit if the memory consistency model is not violated.

Tables 2.3, 2.4, and 2.5 show the current state of the load queue. Assume that all loads and stores access the full 32-bit word.

Table 2.1: Store Queue

Instruction #	Address	Value
5	0x100	0x12345678
7	0x200	<i>unknown</i>
11	0x300	0xABCDABCD
13	0x200	0x11001100
17	<i>unknown</i>	<i>unknown</i>

Table 2.2: Data Cache

Valid?	Address	Value
Y	0x100	0xFFFFFFFF
Y	0x200	0x1234ABCD
Y	0x300	0x87654321
N	0x400	<i>unknown</i>

- (i) Under *sequential consistency* (SC), if the stores make no progress, can each load in Table 2.3 complete? If so, what value is read?

Table 2.3: Load Queue (SC)

Instruction #	Address	Can Complete?	Value
2	0x300	Y	0x87654321
6	0x100	N	
12	0x400	N	
16	0x200	N	
18	0x300	N	

- (ii) Under *total store order* (TSO), if the stores make no progress, can each load in Table 2.4 complete? If so, what value is read?

Table 2.4: Load Queue (TSO)

Instruction #	Address	Can Complete?	Value
2	0x300	Y	0x87654321
6	0x100	Y	0x12345678
12	0x400	N	
16	0x200	N	
18	0x300	N	

- (iii) Under a *fully relaxed multi-copy-atomic memory model*, if the stores make no progress, can each load in Table 2.5 complete? If so, what value is read?

Table 2.5 Load Queue (weak ordering)

Instruction #	Address	Can Complete?	Value
2	0x300	Y	0x87654321
6	0x100	Y	0x12345678
12	0x400	N	
16	0x200	Y	0x11001100
18	0x300	N	

Problem 5.B: TSO and Optimizations (4+4+4 points)

Explain whether the following microarchitectural optimizations are permitted under the TSO consistency model.

- (i) Can a load from one thread on a multithreaded core bypass a value from the write buffer stored by a different thread on the same core? (Recall that a write buffer holds data from committed stores waiting to be written to the cache.)

No – TSO requires store atomicity (a global memory order for stores), so a thread cannot see the result of another thread's write early.

- (ii) Can a write buffer coalesce writes to the same word from the same thread? (Writes from two stores, not necessarily consecutive in program order, are merged into the same write buffer entry.)

No – TSO requires that stores appear in program order, so the write buffer must remain FIFO. Coalescing two non-consecutive stores creates a store-store reordering.

- (iii) Can hardware prefetching be used with the L1 cache for both loads and stores?

Yes – fetching a cache line earlier does not affect consistency so as the loads and stores are performed in the same order without prefetching. The coherence protocol will ensure that any intervening modifications to the line will erase an incorrect prefetch.

Problem 5.C: Sequential Consistency and OoO Scheduling (6 points)

Suppose we design an out-of-order multicore processor that implements sequential consistency within a cache-coherent memory system, using *speculation* to improve performance. Consider two independent load instructions, ld1 and ld2, where ld1 precedes ld2 in program order. In the situation where the address of ld2 is computed before ld1, the core chooses to dynamically reorder the execution of ld2 before ld1.

After the core speculatively executes ld2, but before it can commit ld2, a coherence transaction from another core invalidates the cache line accessed by ld2. What actions, if any, should the core take to maintain sequential consistency? Explain your reasoning.

The core should treat this as a case of misspeculation, as allowing ld2 to retire would create a load-load reordering that is prohibited under sequential consistency. Thus, it is necessary to squash ld2 and all subsequent instructions, and then re-execute ld2.

There are two ways to detect this situation: Either the core compares cache evictions to the load address (the technique employed by the MIPS R10000), or the load is replayed before commit to verify that the speculated load value is the same as the value at commit time.

Problem 6: Synchronization (25 points)

Consider a stack data structure implemented as a singly linked list which uses non-blocking synchronization to support concurrent access by multiple threads.

Each stack entry contains a pointer to the next entry further down the stack. The shared `stack` variable points to the entry at the top of the stack. The `push()` function adds a new entry onto the stack, and the `pop()` function removes and returns the topmost entry. For thread safety, the `stack` pointer must be updated atomically. The high-level pseudocode is shown below:

```
struct entry {
    struct entry *next;
    ...
};

struct entry *stack;

void push(struct entry *new) {
    ATOMIC_BLOCK {
        new->next = stack;
        stack = new;
    }
}

struct entry *pop() {
    struct entry *old, *top;
    ATOMIC_BLOCK {
        top = old = stack;
        if (top != NULL)
            top = top->next;
        stack = top;
    }
    return old;
}
```


Problem 6.A: LR/SC Deadlock (5 points)

Suppose that the load-reserved/store conditional (LR/SC) instruction pair is the only atomic read-modify-write operation provided by our 32-bit architecture. Our initial attempt at implementing the `push()` and `pop()` functions directly using LR/SC yields the following assembly code:

```
# a0 holds pointer to new entry
push:
    la a1, stack           # get address of stack variable
    lr.w t0, (a1)         # load pointer to top entry
    sw t0, 0(a0)          # new->next = stack
    sc.w t0, a0, (a1)     # stack = new
    bnez t0, push         # retry if SC failed
    ret

# a0 returns pointer to previous top entry
pop:
    la a1, stack           # get address of stack variable
    lr.w t0, (a1)         # load pointer to top entry
    mv a0, t0              # old = stack
    beqz t0, update       # skip dereference if (top == NULL)
    lw t0, 0(t0)          # top = top->next
update:
    sc.w t0, t0, (a1)     # stack = top
    bnez t0, pop          # retry if SC failed
    ret
```

Each core has a private write-back/write-allocate L1 data cache, and coherence is maintained through a MESI protocol. The processor implements LR/SC based on the simple approach described in lecture:

- `lr.w` ensures that the line is present in the local cache in the Modified or Exclusive state.
- `sc.w` succeeds if and only if the line has continually remained in the Modified or Exclusive state. On failure, it writes a non-zero code to the destination register.

It turns out that our code contains a major flaw!

We notice that `push()` and `pop()` become stuck in an infinite loop even when there is no contention from other cores. What caused this to happen?

If the stack pointer and the entry being pushed or popped map to the same cache set, the intervening `sw` and `lw` between `lr.w` and `sc.w` can evict the line containing the stack pointer and thus clear the reservation. Retrying the LR/SC sequence does not resolve the cache conflict.

Problem 6.B: Emulating CAS (5 points)

We attempt to fix the code by rewriting it in terms of compare-and-swap (CAS). CAS compares a word in memory to an expected value and, if equal, updates the memory location to a desired value. It returns a Boolean value indicating whether the substitution was successfully performed.

While our architecture lacks a CAS instruction, its functionality can be emulated with an LR/SC sequence of four instructions:

```
int CAS(int *addr, int old, int new) {
    int status;
    ATOMIC_BLOCK {
        if (*addr == old) {
            *addr = new;
            status = 0; // success
        } else {
            status = 1; // failure
        }
    }
    return status;
}

# a0 holds addr
# a1 holds old
# a2 holds new

cas:
    lr.w t0, (a0)           # load original value
    bne a1, t0, fail        # fail if not equal
    sc.w t0, a2, (a0)       # attempt to update
    bnez t0, cas            # retry if SC failed
    ...                     # success

fail:
    ...                     # failure
```

The pseudocode for the stack data structure becomes:

```
void push(struct entry *new) {
    do {
        struct entry *old = stack;
        new->next = old;
    } while (CAS(&stack, old, new));
}

struct entry *pop() {
    struct entry *old, *top;
    do {
        top = old = stack;
        if (top != NULL)
            top = top->next;
    } while (CAS(&stack, old, top));
    return old;
}
```

For simplicity, we will consider only the `push()` function for the remainder of this problem. We rewrite the assembly code for `push()` with the CAS sequence inlined:

```
# a0 holds pointer to new entry
push:
    la a1, stack           # get address of stack variable
    lw t0, (a1)           # load pointer to top entry
    sw t0, 0(a0)          # new->next = stack
    lr.w t1, (a1)         # load pointer to top entry again
    bne t0, t1, push      # retry if not the same
    sc.w t0, a0, (a1)     # stack = new
    bnez t0, push         # retry if SC failed
    ret
```

Recall from lecture that CAS nominally guarantees forward progress. Does that remain true when CAS is emulated using LR/SC, if LR/SC is implemented as described in Part 6.A? If not, give a scenario where forward progress is obstructed.

No, a livelock situation is still possible where all threads repeatedly attempt to acquire exclusive access to the shared stack pointer, thereby clearing each other's reservations and preventing any store-conditional from ever succeeding.

Thread 1: `lr.w`

Thread 2: `lr.w`

Thread 1: `sc.w` (fails)

Thread 1: `lr.w`

Thread 2: `sc.w` (fails)

Problem 6.C: ABA Problem and CAS (5 points)

Is the revised LR/SC version from Part 6.B susceptible to the ABA problem like a CAS instruction would be? If so, describe an interleaved sequence of `push()` and `pop()` operations by two threads that results in a stack entry being permanently lost.

Yes. Let the initial state be `stack → A → C`.

Thread 1: Begin `pop()`; load `A->next`

Thread 2: `A = pop()`

Thread 2: `push(B)`

Thread 2: `push(A)`

Thread 1: Finish `pop()`; the outdated `A->next` becomes the new stack pointer after CAS

The final state is `stack → C`. Entry B has been lost by thread 1.

Problem 6.D: ABA Problem and DW-CAS (5 points)

To avoid the ABA problem, we switch to using double-width compare-and-swap (DW-CAS), a variant of CAS that supports atomic access to two contiguous words in memory.

```
int DWCAS(int *addr, int old1, int new1, int old2, int new2)
{
    int status;
    ATOMIC_BLOCK {
        if ((addr[0] == old1) && (addr[1] == old2)) {
            addr[0] = new1;
            addr[1] = new2;
            status = 0; // success
        } else {
            status = 1; // failure
        }
    }
    return status;
}
```

```
// Assume variables are allocated contiguously
struct entry *stack;
int count;

void push(struct entry *new) {
    do {
        struct entry *old = stack;
        new->next = old;
    } while (DWCAS(&stack, old, new, count, count+1));
}

struct entry *pop() {
    struct entry *old, *top;
    do {
        top = old = stack;
        if (top != NULL)
            top = top->next;
    } while (DWCAS(&stack, old, top, count, count+1));
    return old;
}
```

Explain how DW-CAS overcomes the ABA problem.

The count variable monotonically increases with each push () or pop () call. Thus, even if the stack pointer is modified and then restored to its original value, the double-width comparison fails since the count values will differ.

Problem 6.E: Emulating DW-CAS (5 points)

Consider the following naïve attempt to emulate DW-CAS with single-word LR/SC:

```
# a0 holds addr
# a1 holds old1
# a2 holds old2
# a3 holds new1
# a4 holds new2
dwcas:
    lr.w t0, 0(a0)           # load original value of word 1
    lr.w t1, 4(a0)           # load original value of word 2
    bne a1, t0, fail         # fail if word 1 not equal
    bne a2, t1, fail         # fail if word 2 not equal
    sc.w t0, a3, 0(a0)       # attempt to update word 1
    sc.w t1, a4, 4(a0)       # attempt to update word 2
    or t0, t0, t1
    bnez t0, dwcas           # retry if either SC failed
    ...                       # success
fail:
    ...                       # failure
```

Explain why this simple approach does not work, even if both words are located within the same cache line.

Both words are not updated atomically, exposing a race condition: If the first `sc.w` succeeds while the second `sc.w` fails, memory is left in a partially modified state.

(Note that RISC-V canonically specifies that `sc.w` pairs only with the most recent `lr.w`, but here we assume that the implementation can track multiple reservations independently. It is not too difficult to extend the simple approach based on cache line states to enable this.)