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 **increase** time-per-program when ...

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

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

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

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

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**. You only need to show the values of the memory locations that are written/changed.

Address	Value
0x00	
0x04	
0x08	
0x0c	
0x10	
0x14	
0x18	
0x1c	
0x20	
0x24	
0x28	
0x2c	
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?

2.C (4 pt) Physical Address Space

How much physical memory does this virtual memory system support?

2.D (4 pt) VIPT L1

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

2.E (4 pts) PIPT L2

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

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			
1	0x804							
2	0x808							
3	0x80c							

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?

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

    __________________ # 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

    __________________    # vector load row mat[i][...]

    __________________    # vector load column mat[...][i]

    __________________    # vector store column mat[...][i]

    __________________    # 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.

4.C (4 pt) Reducing Cache Misses

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

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.

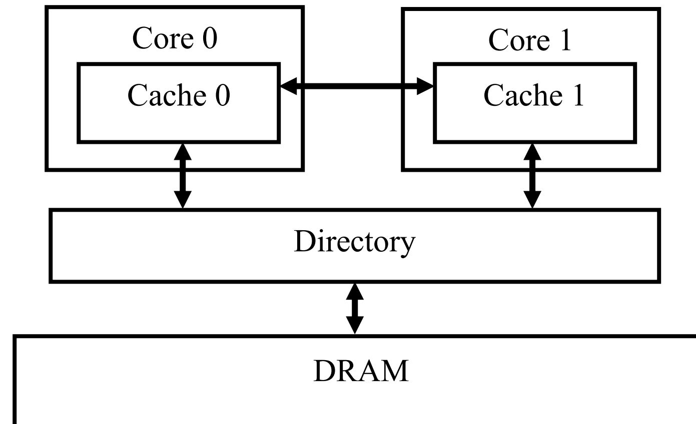
i. **(4 pt)** This approach behaves correctly in a single-core system. How can this approach cause a coherence violation in a multi-core system?

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

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-exclusive 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).

	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				
C1: read a				
C1: write a				
C1: evict a				

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

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.

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

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?

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?

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)`

i. **(5 pt)** SC → TSO

C0.t1	C0.t2	C1.t1	C1.t2	C1.t1

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

C0.t1	C0.t2	C1.t1	C1.t2	C1.t1

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

C0.t1	C0.t2	C1.t1	C1.t2	C1.t1

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?

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.

<table>
<tr><th>Original</th><th>Translated</th></tr>
<tr><td><pre>
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
</pre></td><td><pre>
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
</pre></td></tr>
</table>

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?

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?

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?

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.

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
```