CS 152/252A
Spring 2023 Computer Architecture Final

- You have 170 minutes unless you have DSP accommodations. Exam questions are roughly in the order they were covered in lecture. If a question is used for a clobber, it's labeled either (MT1) or (MT2) depending on the exam it clobbers.
- You must write your student ID on the bottom-left of every page of the exam (except this first one). You risk losing credit for any page you don't write your student ID on.
- For questions with length limits, do not use semicolons or dashes to lengthen your explanation.
- The exam is closed book, no calculator, and closed notes, other than three double-sided cheat sheet that you may reference.
- For multiple choice questions,
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

First name	
Last name	
SID	
Exam Room	
Name and SID of person to the right	
Name and SID of person to the left	
Discussion TAs (or None)	

While the statement of the Honor Code itself is brief, it is an affirmation of our highest ideals as Golden Bears.

Honor code: "As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others."

By signing below, I affirm that all work on this exam is my own work, and honestly reflects my own understanding of the course material. I have not referenced any outside materials (other than one double-sided cheat sheet), nor collaborated with any other human being on this exam. I understand that if the exam proctor catches me cheating on the exam, that I may face the penalty of an automatic "F" grade in this class and a referral to the Center for Student Conduct.

Signature: _____

THIS PAGE IS INTENTIONALLY LEFT BLANK

Q1. [20 pts] Iron Law & Pipelining (MT 1)

For questions (a) to (d), determine whether each given statement is true. If false, point out and replace the incorrect part.

Example: Lowering CPU clock frequency will (1) decrease (2) seconds-per-cycle because (3) each clock cycle now takes longer.

Answer: *The statement is false because part (1) is wrong.*
Replace with: *increase*

(a) [3 pts] Adding a branch delay slot might (1) increase (2) instructions-per-program because (3) the branch predictor might not be accurate.

Which option best describes the statement above?

- ○ The statement is true.
- ○ The statement is false because part (1) is wrong.
- ○ The statement is false because part (2) is wrong.
- ● The statement is false because part (3) is wrong.

If the statement is false, replace the incorrect part (under 1 sentence) so that the statement becomes true:

The compiler will have to add extra instructions to fill the slot

(b) [3 pts] In a classic 5-stage pipeline, supporting precise exceptions might (1) increase (2) cycles-per-instruction due to (3) added logic complexity.

Which option best describes the statement above?

- ○ The statement is true.
- ○ The statement is false because part (1) is wrong.
- ● The statement is false because part (2) is wrong.
- ○ The statement is false because part (3) is wrong.

If the statement is false, replace the incorrect part (under 1 sentence) so that the statement becomes true:

seconds-per-cycle

(c) [3 pts] Moving from a single-threaded core to SMT-enabled core might (1) increase (2) time-per-cycle due to (3) duplicated microarchitecture structures (PC, ArchRF, ...) and scheduling logic.

Which option best describes the statement above?

- ● The statement is true.
- ○ The statement is false because part (1) is wrong.
- ○ The statement is false because part (2) is wrong.
- ○ The statement is false because part (3) is wrong.

If the statement is false, replace the incorrect part (under 1 sentence) so that the statement becomes true:

N/A

(d) [3 pts] Stripmining on a vector processor might (1) increase (2) instructions-per-program due to (3) handling cases where the iteration count is not divisible by the vector length.

Which option best describes the statement above?

- ● The statement is true.
- ○ The statement is false because part (1) is wrong.
- ○ The statement is false because part (2) is wrong.
- ○ The statement is false because part (3) is wrong.

If the statement is false, replace the incorrect part (under 1 sentence) so that the statement becomes true:

N/A

(e) [4 pts] A 5-stage pipeline that has a 1-cycle ALU and a multi-cycle **unpipelined** FPU has WAW and structural hazards. We would like to pipeline the FPU to potentially help with these issues.

(i) [2 pts] Does pipelining the FPU help with WAW hazards? ○ Yes ● No

(ii) [2 pts] Does pipelining the FPU help with structural hazards? ● Yes ○ No

(f) [4 pts] Assume an out-of-order core is executing a store instruction before a load instruction to different addresses (i.e. instruction 0 = store to address A, instruction 1 = load to address B). For this particular core, the core designer allows for the load to complete before the store (the load is reordered before the store) if the store takes longer than the load to issue or execute. By completing before the store, the load is allowed to bring its data into the cache, potentially evicting older data. Note that the load can only complete, i.e the register for the load will not be modified until it commits.

Assume the older store throws an exception **after** the younger load completes. Is this behavior still valid for precise exceptions?

● Yes ○ No

Q2. [20 pts] Microcode Grab Bag (MT 1)

The questions in this section may be answered independently of one another.

It may be helpful to refer to **Appendix A** on microcoding while answering this question.

(a) [4 pts] The developer before you had tried to implement an instruction in microcode. However, since they didn't take CS 152/252A, their implementation might have a bug! They've left you the instruction and pseudocode, as well as their potentially buggy microcode.

Instruction:

```
BUGGY rd, rs1, rs2
```

Pseudocode:

```
if (R[rs1] != 0) {
    R[rd] = R[rd] + M[R[rs2]];
}
```

Microcode implementation:

Line	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
1	A <- R[rs1]	0	rs1	0	1	1	*	*	0	*	0	0	*	0	N	
2	if A == 0, jump MA <- R[rs2]	0	rs2	0	1	*	*	COPY_A	0	1	0	0	*	0	EZ	FETCH0
3	A <- Mem	0	*	0	0	1	*	*	*	0	*	1	*	0	N	
4	B <- R[rd]	0	rd	0	1	0	1	*	0	*	0	0	*	0	N	
5	R[rd] <- A + B	0	rd	1	0	*	*	ADD	1	*	0	0	*	0	J	FETCH0

(i) [2 pts] If there is an incorrect line of microcode in the above implementation, what line contains an error? If no lines contain errors, please mark "None of the above".

- ○ Line 1
- ○ Line 2
- ● Line 3
- ○ Line 4
- ○ Line 5
- ○ None of the above

(ii) [2 pts] Why is the line you marked above incorrect? You may write **at most 2 sentences** of explanation.

Note: if you marked "None of the above" for the previous subpart, leave this part blank.

Line 3 should be a S (spin) for uBr instead of N (next). This is since we must wait for memory to load into A, which requires us to spin on that line until the memory value has arrived. (**Note:** This was the intended error. However, there were two additional, accidental errors: ALUEn should be 0 to prevent the case of both memory and ALU writing to the bus at the same time; MemWr should be 0 to prevent writing garbage or some other value to the memory address. Credit was given if the student selected the correct option in part (i) and had at least one of these reasons.)

(b) [16 pts] **Reverse Engineering Microcode**

The aforementioned developer was, unfortunately, also a firm believer in self-documenting code and chose not to explain what some of the microcoded instructions do! In this part, we consider an implementation of the microcoded instruction `mystery`. The microcode for this instruction can be found on page 7 of your exam booklet.

(i) [9 pts] Analyze the encoded control signals for the `mystery` instruction and **complete the pseudocode** in the space provided in the table on page 7. If a row encodes multiple pseudo-operations, write both operations in the same pseudocode box. Unless the pseudocode for a row is already provided, you need to fill out the pseudocode for every row with microcode signals in the table.

(ii) [3 pts] In **one sentence** (or less), name or describe the **high-level operation** that this instruction implements. No credit will be given for simply translating the pseudocode to English.

This is an implementation of the `strlen` function.

—or—

This instruction counts the number of sequential, non-zero valued bytes in a buffer.

(iii) [2 pts] What is/are the input register(s) of the mystery instruction?

- [x] rs1
- [ ] rs2
- [ ] rd
- [ ] A
- [ ] B

(iv) [2 pts] What is/are the output register(s) of the mystery instruction?

- [ ] rs1
- [ ] rs2
- [x] rd
- [ ] A
- [ ] B

SID: _____

Microcode Implementation of the MYSTERY Instruction

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	µBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
MYSTERY:		0	*	0	0	0	1	COPY_A	1	*	0	0	*	0	N	*
		0	Rd	1	0	*	*	SUB	1	*	0	0	*	0	N	*
		0	Rs1	0	1	1	*	*	0	*	0	0	*	0	N	*
AGAIN:		0	*	0	0	0	*	COPY_A	1	1	0	0	*	0	N	*
		0	*	0	0	0	1	*	0	0	0	1	*	0	S	*
		0	*	0	0	0	0	COPY_B	0	*	0	0	*	0	EZ	FETCH0
		0	*	0	0	1	0	INC_A	1	*	0	0	*	0	N	*
		0	Rd	0	1	0	1	*	0	*	0	0	*	0	N	*
		0	Rd	1	0	0	*	INC_B	1	*	0	0	*	0	J	AGAIN

SID: _____

Microcode Implementation of the MYSTERY Instruction

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	µBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
MYSTERY:	B ← A	0	*	0	0	0	1	COPY_A	1	*	0	0	*	0	N	*
	R[Rd] ← A – B	0	Rd	1	0	*	*	SUB	1	*	0	0	*	0	N	*
	A ← R[Rs1]	0	Rs1	0	1	1	*	*	0	*	0	0	*	0	N	*
AGAIN:	MA ← A	0	*	0	0	0	*	COPY_A	1	1	0	0	*	0	N	*
	B ← M	0	*	0	0	0	1	*	0	0	0	1	*	0	S	*
	if EZ(B) goto FETCH0	0	*	0	0	0	0	COPY_B	0	*	0	0	*	0	EZ	FETCH0
	A ← A + 1	0	*	0	0	1	0	INC_A	1	*	0	0	*	0	N	*
	B ← R[Rd]	0	Rd	0	1	0	1	*	0	*	0	0	*	0	N	*
	R[Rd] ← B + 1 goto AGAIN	0	Rd	1	0	0	*	INC_B	1	*	0	0	*	0	J	AGAIN

Q3. [12 pts] Caches (MT 1)

Way prediction is an optimization technique used in set-associative caches. The principle is that we predict which cache way is most likely going to be accessed for a particular memory request. If our prediction is correct, there is no need to check the other ways in the cache. If it is incorrect, we proceed as though with a normal set associative access.

(a) First, consider a two way set associative cache which is designed with either way-prediction (one data way is read at a time) or concurrent data access (both data ways are read at the same time).

(i) [2 pts] True or False: In all circumstances, conventional concurrent data access caches have an AMAT *less than or equal* to that of way-predicted cache. ● True ○ False

In no more than 2 sentences, justify your answer:

> Even with 100% prediction accuracy, the way-predicted cache will perform a tag check and data access, which is the same as the concurrent access cache.

(ii) [2 pts] True or False: A concurrent data access cache will consume *equal or more* power than a way predicted cache in all circumstances. ● True ○ False

In no more than 2 sentences, justify your answer:

> In the case of a correct prediction, the way-predicting cache only has to activate 1 way, but in all cases the concurrent access cache will activate all the ways.

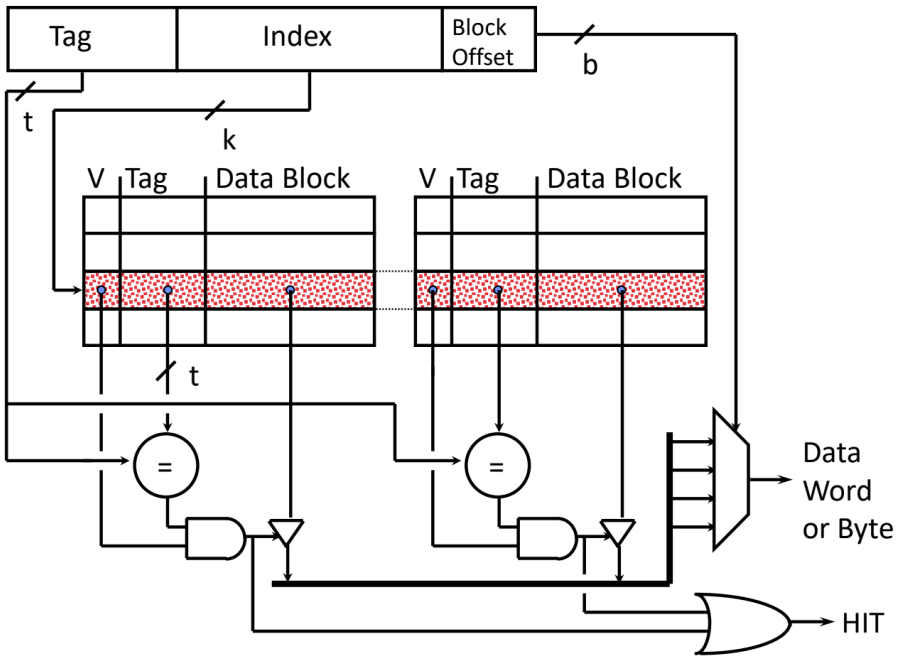


Figure 1: General 2-Way Set Associative Cache

(b) **(i)** [2 pts] Consider a 16 kB two-way set associative cache **without** way prediction. Given a hit time of 5 cycles, a hit rate of 80%, and an L2 access time of 30 cycles, what is the AMAT of this cache?

> 5+0.2*30 = 11

(ii) [2 pts] Consider an 8kB **direct mapped** cache. Given a hit time of 2 cycles, a hit rate of 60%, and an L2 access time of 30 cycles, what is the AMAT of this cache?

2+0.4*30 = 14

(iii) [1 pt] Now consider a 16 kB two-way set associative cache **with** way prediction. What is the hit rate of this cache?

- ○ 60%
- ○ 70%
- ● 80%
- ○ None of the above

(iv) [3 pts] What is the AMAT of this cache?

(0.6*2 + 0.4*5) + 0.2*30 = 9.2

(v) [1 pt] In the case of a data cache, which of the following is the preferred input to the way-predictor (reduces latency to cache access)?

- ○ The data address
- ● PC

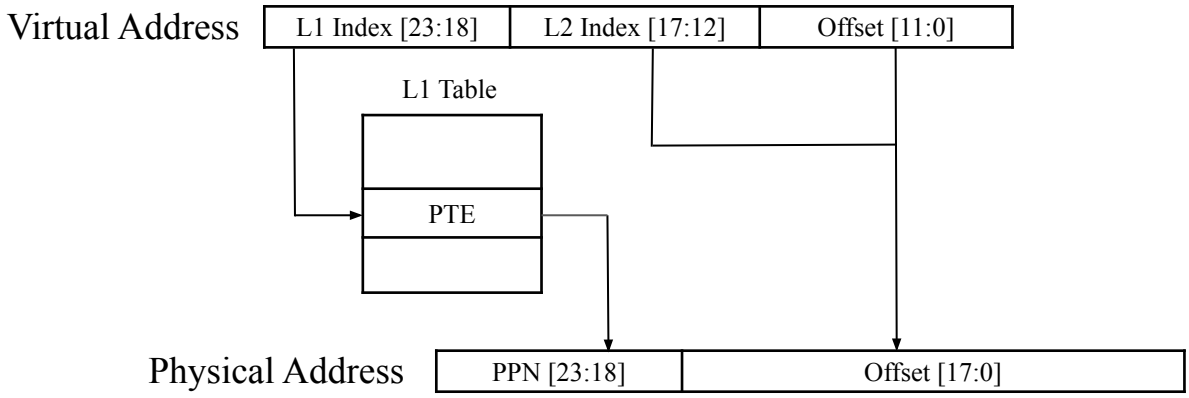
(vi) [2 pts] Justify your answer to the previous part.

The predicted way must be available before the actual data access to as to reduce latency. This is best done with the load PC.

Q4. [18 pts] Virtual Memory (MT 1)

Recall from discussion that superpages are memory pages of large sizes. Most general purpose processors support superpages because of many benefits they can bring to the table. Processes can specify if they would like to allocate superpages or "regular pages" for their workload. This question will explore superpaging in more detail.

As a quick refresher, here is a diagram showing how superpages are translated in a system with two-level page tables, where the Page Table Entry in the L1 table points to a superpage.



(a) VM Concept Blitz

(i) [2 pts] Let's recap some virtual memory concepts. **Select all that are true.**

- [x] Systems with virtual memory can give the illusion of more memory than is physically available.
- [x] Paging provides a layer of security.
- [ ] All modern systems must have virtual memory.
- [x] Virtual memory is expensive from a hardware and runtime perspective.
- ○ None of the above

(ii) [2 pts] Let's consider some superpaging concepts. **Select all that are true.**

- [ ] Superpaging reduces hardware complexity.
- [ ] A system that supports superpaging is less prone to external fragmentation.
- [x] With superpaging, TLB memory scope increases.
- [x] With superpaging, disk traffic can increase.
- ○ None of the above

(b) [10 pts] **Super Translation**

Consider a system that uses **32-bit** words, **10-bit** virtual addresses, **16-byte** pages, and **three-level** page tables. This system supports superpages. The system memory has a latency of **90 ns**. A secondary storage (disk) is attached to the system. The disk has a latency of **2.5 ms** and a speed of **0.1 byte/ns**.

In addition to contents, each PTE contains 1 bit indicating whether the pointed page is on disk, and 1 bit indicating if that page is a superpage. If a page is on disk, it needs to be transferred into memory before its content can be read. Note that here we have simplified the page fault handling process.

Below are the virtual addresses of two memory accesses, the content of the Page Table Base Register, and a table showing the contents of a portion of physical memory used for page tables. Recall that the content of a PTE stores the page number of the next-level page table.

Access Order	Virtual Address
1	0x0FA
2	0x349

Page Table Base Register
0x30

Addr	Contents	Superpage?	On Disk?
0x00	0x04	0	0
0x04	0x06	0	1
0x08			
0x0C	0x20	1	0
0x10			
0x14			
0x18	0xA1	0	1
0x1C			
0x20			
0x24	0x15	0	1
0x28	0x0A	0	1
0x2C	0x78	0	0
0x30	0x05	0	0
0x34			
0x38			
0x3C	0x90	1	1
0x40			
0x44			
0x48	0x12	1	1
0x4C	0x01		
0x50	0x09	1	0
0x54			
0x58			
0x5C	0x02	0	1

(i) [4 pts] What is the physical address of the first memory access?

0x78A

(ii) [4 pts] What is the physical address of the second memory access?

0x9049

(iii) [2 pts] Which of the two memory accesses has a **lower** latency? Here, latency is defined as the time between when an access begins the translation process and when the target byte is retrieved from memory. Assume that no other latency is involved except the ones mentioned above.

Hint: disk access time = latency + size-of-transfer / rate-of-transfer

● First memory access ○ Second memory access

(c) Superpage Scenarios

In the following scenarios, which paging mechanism would be better and why?

(i) [2 pts] Context switching among many apps that have a working set of 1 MB on a system with 4 MB pages and 16 MB superpages.

○ Superpaging ● Regular Paging ○ Neither is better than the other

In **no more than 2 sentences**, justify your answer:

We do not need superpages for the size of the working sets because it ends up wasting memory. This would result in potentially hitting disk more with enough apps running in parallel.

(ii) [2 pts] Training a machine learning model with a massive dataset (4 GB) – requiring multiple iterations through the data – on a system with 4 MB pages and 16 MB superpages.

● Superpaging ○ Regular Paging ○ Neither is better than the other

In **no more than 2 sentences**, justify your answer:

The spatial locality of the workload, paired with the need for more memory scope at the TLB level, points to superpaging. With larger TLB scope, CPI is less (no need for as many expensive page table walks during memory accesses) and hence, a faster completion of the task at hand.

Q5. [20 pts] Out-of-Order Pipelines (MT 2)

(a) [8 pts] **Avenue (Issue) Q**

In this part, you are working on the design of an out-of-order processor which has separate functional units for integer operations, floating point, and memory. Each functional unit has its own issue queue (also referred to as a reservation station or issue buffer) from which it is able to issue dispatched instructions.

(i) [1 pt] A coworker suggests that it is a good idea to use a large ROB (>200 entries) and size the issue queue of each individual functional unit such that it is the size of the ROB. When considering power, performance, and area trade offs, is your coworker's suggestion a good one?

○ Yes ● No

(ii) [3 pts] In no more than two sentences, justify your answer:

A 200 entry issue queue for each functional unit is not feasible. Issue queues are expensive from a logic and area perspective because they are responsible for evaluating both whether an instruction can issue at all and, amongst the ready candidates, which should issue. Additionally, updating large IQs when information about writebacks becomes available is very expensive as it requires essentially treating the IQ as content addressable memory (equivalently, a very highly associative cache!).

—*or*—

It doesn't make sense to size each IQ to the ROB as it is very unlikely that we'd ever actually *use* such capacity as most reasonable programs are a mix of integer, memory, and floating point operations.

(iii) [1 pt] Another coworker suggests that the ROB should have more entries than the total sum of all issue queue entries across all functional units. When considering power, performance, and area trade offs, is your coworker's suggestion a good one? You may ignore considerations around scaling the physical register file and free list.

● Yes ○ No

(iv) [3 pts] In no more than two sentences, justify your answer:

~~Scaling the ROB is cheap. Instructions are dispatched in-order and the ROB does not have any complex content~~ addressable memory behaviors. Since the ROB sets the maximum number of in-flight instructions in the pipeline (including undispatched and issued instructions as well as completed but uncommitted instructions!) whereas the IQ sizes determines the maximum number of instructions which can be considered for issue in a cycle. If the ROB is smaller or equal to the sum of the issue queues, the issue queues (and thus the amount of ILP available to us!) will be underutilized as the ROB will fill before we reach capacity. To keep the backend saturated, we want to be able to refill all IQs whenever an instruction issues as well as to track all possibly instructions as the flow through execution. We always want to scale the cheaper structure to keep the more expensive structure saturated: ~~why the ROB grows to fully cover the IQs.~~

(b) [12 pts] **They see me rollin', they hatin'**

In the tables below, update the ROB, rename table, and freelist to reflect the state of the processor after executing the given program and completing any necessary rollbacks using a multi-cycle unwind procedure. Assume that all instructions which can be committed are committed *before* any rollback operations begin. Additionally, assume that the pagefault exception is detected after the `sub` instruction has already begun executing and that the branch mispredict is resolved at some point after the pagefault exception.

- The free list operates as a FIFO queue; entries are popped from the left and freed entries are pushed on the right.
- When removing an item from the freelist, do not cross out entries; instead, mark an "X" in the row immediately below.
- If an instruction does not write to the register file, mark an "X" in the ROB.
- When completing the rename table, do not cross out entries. Instead, write the new physical register in the next box to the right. You may not need to use all spaces.

The first instruction has been completed in the tables for you. **All three of the tables below will be graded.**

PC	Instruction
00	`addi x2, x2, #1`
04	`ld x2, 0(x2)`
08	`beq x2, x0, label ; mispredicted as not taken, resolved very late`
0c	`mul x3, x2, x2`
10	`st x3, 0(x2) ; pagefault exception, detected early`
14	`sub x2, x2, x2`
	...
ff	`label: /* ommitted */`

ROB

#	Operation	Rd	Previous Rd	Committed	Rolledback?
0	addi	p8	p0	Y	N
1	ld	p2	p8	Y	N
2	beq	X	X	Y	N
3	mul	p9	p5	N	Y
4	st	X	X	N	Y
5	sub	p6	p2	N	Y

Freelist

p8	p2	p9	p6	p0	p8	p6	p9	
X	X	X	X					

Rename Table

Arch. Register	Physical Register				
x2	p0	p8	p2	p6	p2
x3	p5	p9	p5		

(i) [1 pt] After completing rollback, should an exception be raised?

○ Yes ● No

The pagefault occurred along a mispredict path. Exceptions caused by rolledback instructions must be suppressed.

(ii) [1 pt] After completing rollback, at what `PC` should execution continue at? Write `exception` if execution should continue at the exception handler.

ff

Q6. [10 pts] Multithreading (MT2)

(a) Match each of advantages and disadvantages to the most appropriate type of multi-threading. Each advantage/disadvantage should only be used once, so use it for the type of multithreading it *best* applies to. **Keep scratch work away from the multiple choice options for each question.**

1. Not possible on a single-issue processor.
2. Can, but not necessarily effective at, hiding the throughput losses from both long and very short stalls.
3. Useful only for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time.
4. Most effective at minimizing both horizontal and vertical waste.
5. In general, slows down execution of an individual thread, even a thread that is ready to execute and doesn't have stalls.
6. Doesn't need thread switching to be extremely low overhead.

(i) [2 pts] Coarse-grained multithreading:

Advantage:

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ● 6

Disadvantage:

○ 1 ○ 2 ● 3 ○ 4 ○ 5 ○ 6

(ii) [2 pts] Fine-grained multithreading:

Advantage:

○ 1 ● 2 ○ 3 ○ 4 ○ 5 ○ 6

Disadvantage:

○ 1 ○ 2 ○ 3 ○ 4 ● 5 ○ 6

(iii) [2 pts] Simultaneous multithreading:

Advantage:

○ 1 ○ 2 ○ 3 ● 4 ○ 5 ○ 6

Disadvantage:

● 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

(b) [4 pts] Suppose we have a superscalar out-of-order CPU and want to add support for simultaneous multithreading to it. Which of the following CPU components need to be duplicated to maintain program correctness?

- ■ Program Counter (PC) register
- ☐ Physical registers
- ☐ Functional units
- ☐ Functional unit issue queues
- ☐ Data memory ports
- ■ Architectural Register file
- ☐ Branch predictor
- ○ None of the above

Q7. [20 pts] Vectorizing Data Processing (MT2)

(a) In data processing, one of the most basic types of processing is summing a column of a table (otherwise known as a reduction) to a single overall value. Here is the psuedocode for an iterative sum reduction of a table's column:

```
# Assumptions:
#   - 'table' is a 2D array of type table[row][col] and has
#     'numRows' rows and 'numCols' columns
#   - 'table' is stored in column-major order
#     (all column values are stored contiguously in memory)
#   - 'col' < 'numCols' (within the bounds of the array)

int sum_col(int table[][], int col, int numRows):
    int sum = 0;
    for (int i = 0; i < numRows; ++i):
        sum += table[i][col];
```

We would like to convert the code to a vectorized implementation for a substantial performance speedup. You initially describe the following pseudocode for the vector implementation:

1. Set a scalar register sum to `0`
2. Run a stripmine loop (each iteration does a vector length (`VL`) element partial sum)
   (a) Naively load VL contiguous elements of column `col` into a vector register
   (b) Sum the entire vector register and add partial sum to overall scalar register sum
   (c) If more rows exist, loop back to (a) while also modifying next `VL` to be `max(VL, number of rows left)`
3. Return the overall scalar register sum

(i) [2+2 pts] If the table given was stored in row-major order (i.e. row values are stored in contiguous memory locations), would the prior vector implementation break?

● Yes ○ No

Explain in at most **two** sentences.

(Option 1) Now you would need to have strided load instructions that skip across row elements. (Option 2) You were already using strided load instructions (of stride 1) so the prior instruction is fine (just need to modify the stride).

(ii) [2+2 pts] The sum operation in the vector implementation is done iteratively (partial sums are iteratively added in the stripmine loop to the overall scalar sum). Is it possible to also vectorize these sums?

● Yes ○ No

Explain in at most **three** sentences.

Use a binary-addition tree. Store all partial sums in another vector register, then reduce that vector register.

(b) While getting the sum of column values is great, sometimes data processing requires sorting the output values and returning a new table. For the sake of simplicity, ignore the table from the previous parts. Instead we would like to sort a single array in a vectorized way. To do this, we can use a vectorized version of the quicksort algorithm, which recursively sorts an array by partitioning it (splitting it into 2 arrays) based on a chosen pivot element. This algorithm is known to be fast if the partitioning step in quicksort can be vectorized. The **iterative** pseudocode for this partitioning operation is given below assuming that the partition is done for an array that fits within a vector register completely:

```
1. set 'pivotValue' to a random element
2. move all values < 'pivotValue' one-by-one to left side of the array
3. move all values >= 'pivotValue' one-by-one to right side of the array
```

To help with this you are given new instruction called `vcompress` that allows elements selected by a vector mask register from a source vector register to be packed into contiguous elements at the start of a destination vector register.

```
Example use of 'vcompress' instruction

8 7 6 5 4 3 2 1 0   Element number

1 1 0 1 0 0 1 0 1   v0 = mask reg.
8 7 6 5 4 3 2 1 0   v1 = source reg.
1 2 3 4 5 6 7 8 9   v2 = dest. reg.

                    <--- execute vcompress v2, v1, v0 <---

0 0 0 0 8 7 5 2 0   v2 *after* vcompress
```

(i) [1+1+1+1 pts] You are given the following pseudocode that uses this new `vcompress` instruction to implement the partitioning step in a vectorized/optimized way. Fill in the blanks with phrases.

1. Make a mask based on if elements are [greater (or less) than] the `pivotValue`.
2. Use `vcompress` with that mask to move elements into the vector register.
3. Store the 'compressed' destination vector to memory.
4. [Invert] the mask.
5. Repeat step(s) [2].
6. Store values to memory [at an offset (memory now has contiguous vector)].
7. Load contiguous vector length memory back into the vector register.

(ii) [3 pts] If the partitioning step also required you to return the index of the `pivotValue` in the final vector register, how could you use the mask register given to the `vcompress` to determine the index?

- (●) (1) Sum all 1's in mask and use sum as index
- ( ) (2) Use highest index of 1 as index
- ( ) Use either (1) or (2)
- ( ) You can't use the mask to determine the index

(iii) [2+3 pts] Assuming the `vcompress` instruction **doesn't** exist, with strictly vector instructions (i.e. no iterative scalar loops), what single type of vector instruction not used in the pseudocode might allow you to implement this functionality?

Scatter/Gather Load/Store

Why? Explain in at most **three** sentences.

(3 sentence version) Same as before, you just would replace the vcompress step and final vector register load with scatter (and potentially gather) operations to store the values in contiguous memory. Make a mask based on if elements are greater than the pivot. Scatter the values into consecutive memory locations. Invert mask. Scatter the values into consecutive memory locations with offset. Load the value back into the vector register.

Q8. [24 pts] Cache Coherence

(a) [9 pts] **MOESI: Must Observe, Every State Identified!**

Important **assumptions** for this question:

1. ***A*** is a processor with a private cache. So is ***B***, and so is ***C***.
2. All private caches follow the MOESI protocol, and can snoop on other caches using a shared bus.
3. The MOESI protocol referred to in this part can be viewed in the below diagram that contains the possible MOESI state transitions.

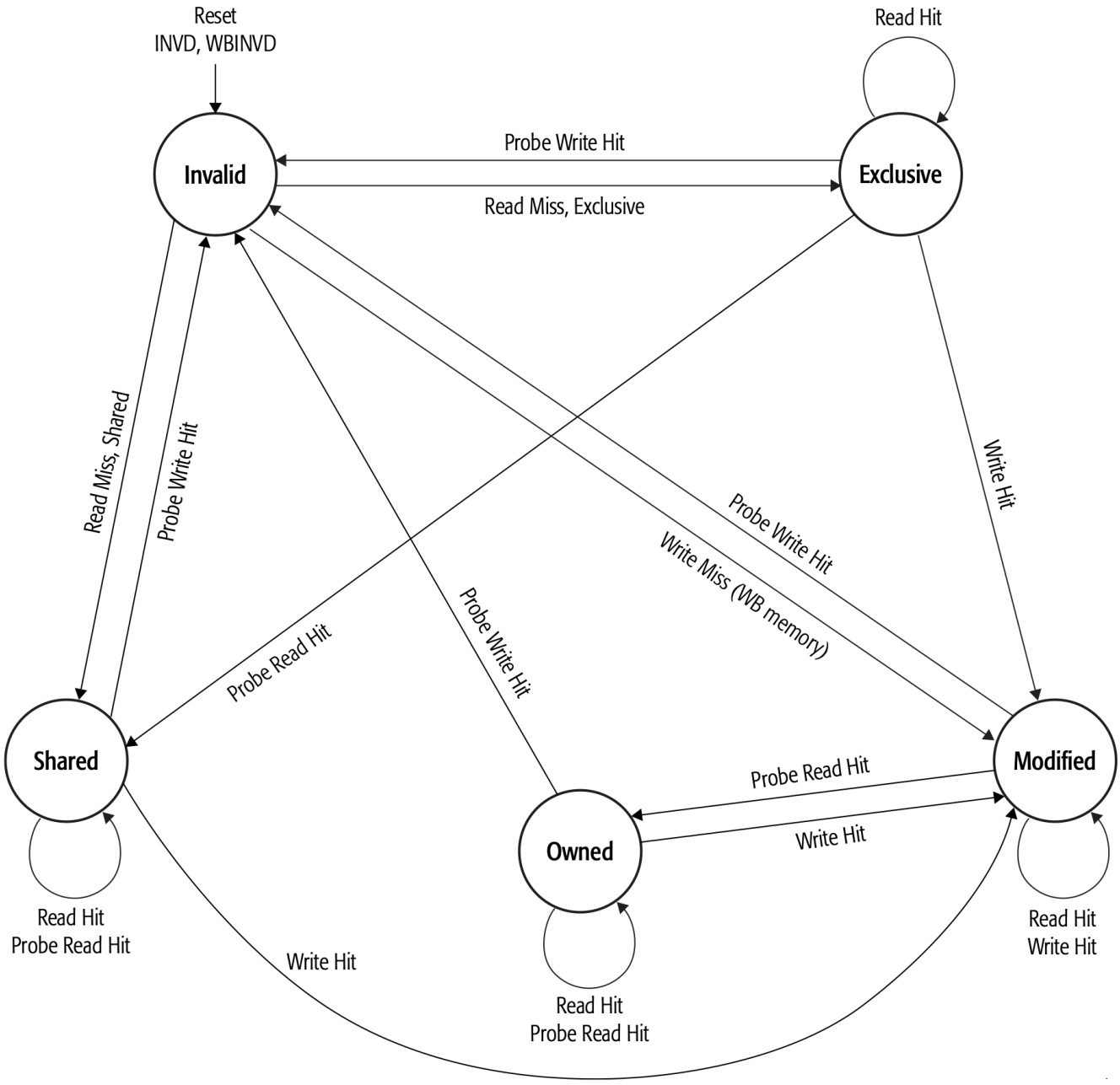


Figure 7-2. MOESI State Transitions

For each of the following subparts, please select the initial and next cache states to represent the transition which the given description best describes.

Here are what the options refer to: M (Modified), O (Owned), E (Exclusive), S (Shared), I (Invalid).

Important: Consider each subpart **independently** of the others. **Assume location X is initially neither** in A's nor B's cache, at the **start of each subpart**.

(i) [2 pts] Both A and B have read from location X. At this point, A's cache is in state `StateA1`, and B's cache is in state `StateB1`. Now, A writes to location X. At this point, A's cache is in state `StateA2`, and B's cache is in state `StateB2`.

A's cache transitions from `StateA1` to `StateA2`.

`StateA1:` ○ M ○ O ○ E ● S ○ I

`StateA2:` ● M ○ O ○ E ○ S ○ I

B's cache transitions from `StateB1` to `StateB2`.

`StateB1:` ○ M ○ O ○ E ● S ○ I

`StateB2:` ○ M ○ O ○ E ○ S ● I

(ii) [2 pts] B **reads** from location X. At this point, A's cache is in state `StateA1`, and B's cache is in state `StateB1`. Now, A reads from location X. At this point, A's cache is in state `StateA2`, and B's cache is in state `StateB2`.

A's cache transitions from `StateA1` to `StateA2`.

`StateA1:` ○ M ○ O ○ E ○ S ● I

`StateA2:` ○ M ○ O ○ E ● S ○ I

B's cache transitions from `StateB1` to `StateB2`.

`StateB1:` ○ M ○ O ● E ○ S ○ I

`StateB2:` ○ M ○ O ○ E ● S ○ I

(iii) [2 pts] B **writes** to location X. At this point, A's cache is in state `StateA1`, and B's cache is in state `StateB1`. Now, A reads from location X. At this point, A's cache is in state `StateA2`, and B's cache is in state `StateB2`.

A's cache transitions from `StateA1` to `StateA2`.

`StateA1:` ○ M ○ O ○ E ○ S ● I

`StateA2:` ○ M ○ O ○ E ● S ○ I

B's cache transitions from `StateB1` to `StateB2`.

`StateB1:` ● M ○ O ○ E ○ S ○ I

`StateB2:` ○ M ● O ○ E ○ S ○ I

(iv) [3 pts] A, B, C read from location X. A writes to location X. B reads from location X. C writes to location X. A reads from location X. At this point, A's cache is in state `StateA1`, B's cache is in state `StateB1`, and C's cache is in state `StateC1`.

`StateA1:` ○ M ○ O ○ E ● S ○ I

`StateB1:` ○ M ○ O ○ E ○ S ● I

`StateC1:` ○ M ● O ○ E ○ S ○ I

(b) [15 pts] **Right Writes Despite Network Plights...**

For directory based cache coherence, we have so far assumed that the network is reliable. What if it's not? Without this assumption, say we now have an **unreliable** network between the caches and the directory controller.

If a cache sends a request or response to the directory controller, the message might get dropped by the network instead of reaching the directory controller. In that case, the directory controller would never see *that specific* message, since it **did not make it through the unreliable network** successfully. Similarly, a message *from the directory controller* may never reach the cache it was intended for.

How can we still **ensure coherency** under these conditions? **For this problem, consider the following scenario.**

Let A and B be cores. Let DC represent the directory controller. The available messages are:

- `WriteReq(X)`: a write request to store data X into memory location L, from a cache to DC.
- `WriteRsp()`: a write response for memory location L, from DC to a cache.
- `ReadReq()`: a read request to load from memory location L, from a cache to DC.
- `ReadRsp(X)`: a read response containing the data X that was at memory location L, from DC to a cache.
- `InvReq()`: an invalidate request for removing memory location L from the cache, from DC to a cache.
- `InvRsp()`: an invalidate response that memory location L has been removed from the cache, from a cache to DC.

For this question, **assume** that A and B read and write from a **single memory location** L, which is initialized to 0.

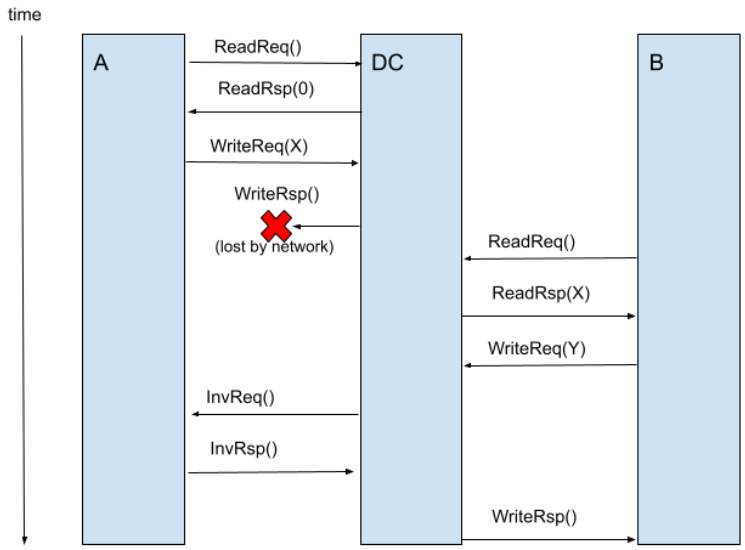
A's code:
```
read()
write(X)
```

B's code:
```
if read() == X:
    write(Y)
    R = read()
else:
    R = Z
```

(i) [2 pts] Assuming that A's and B's caches are coherent, what are the possible values of `R`?

- ☐ 0
- ☐ X
- ■ Y
- ■ Z

As *A* and *B* execute their code, the following series of events takes place:



The same series of events in a list format:

1. *A* sends *DC* a `ReadReq()`.
2. *DC* receives *A*'s `ReadReq()`, and sends *A* a `ReadRsp(0)`.
3. *A* receives *DC*'s `ReadRsp(0)`.
4. *A* sends *DC* a `WriteReq(X)`.
5. *DC* receives *A*'s `WriteReq(X)`.
6. *DC* sends *A* a `WriteRsp()`, but this never makes it through the network to *A*.
7. *B* sends *DC* a `ReadReq()`.
8. *DC* receives *B*'s `ReadReq()`, and sends *B* a `ReadRsp(X)`.
9. *B* receives *DC*'s `ReadRsp(X)`.
10. *B* sends *DC* a `WriteReq(Y)`.
11. *DC* receives *B*'s `WriteReq(Y)`, and sends *A* an `InvReq()`.
12. *A* receives *DC*'s `InvReq()`, and sends *DC* an `InvRsp()`.
13. *DC* receives *A*'s `InvRsp()`, and sends *B* a `WriteRsp()`.

(ii) [1 pt] After the above events have occurred, from the view point of *A*, what value does the memory location *L* currently have?

○ 0 ○ *X* ○ *Y* ○ *Z* ● Not in *A*'s cache

(iii) [1 pt] After the above events have occurred, from the view point of *B*, what value does the memory location *L* currently have?

○ 0 ○ *X* ● *Y* ○ *Z* ○ Not in *B*'s cache

(iv) [1 pt] After the above events have occurred, from the view point of *DC*, what value does the memory location *L* currently have?

○ 0 ○ *X* ● *Y* ○ *Z*

So far, from *A*'s perspective, its `WriteReq(X)` never receives a response. Let's assume that to resolve the issue of *A* never receiving a write response from *DC*, *A* sets up a timer.

With this new addition, when *A* sends out its initial write request, it starts the timer. If *A* has not received a corresponding write response after some amount of time T_1, *A* will send another write request, and restart the timer.

Assume that **no** changes are made to *DC* so far, and that upon receiving any write request from *A*, *DC* will be able to successfully send *A* a write response through the network.

A's timer goes off after *DC* sends a `WriteRsp()` to *B* in the initial series of events. Then, the following events occur:

1. *A* sends another `WriteReq(X)` to *DC*.
2. *DC* receives *A*'s `WriteReq(X)`.
3. *DC* sends *B* a `InvReq()`.
4. *B* receives *DC*'s `InvReq()` and sends an `InvRsp()`.
5. *DC* receives *B*'s `InvRsp()` and sends *A* an `WriteRsp()`.

(v) [2 pts] After all the above events have occurred, from the view point of *A*, what value does the memory location *L* currently have?

○ 0 ● *X* ○ *Y* ○ *Z* ○ Not in *A*'s cache

(vi) [2 pts] After all the above events have occurred, from the view point of *B*, what value does the memory location *L* currently have?

○ 0 ○ *X* ○ *Y* ○ *Z* ● Not in *B*'s cache

(vii) [2 pts] After all the above events have occurred, from the view point of *DC*, what value does the memory location *L* currently have?

○ 0 ● *X* ○ *Y* ○ *Z*

Now, *B* wants to read what the value at memory location *L* actually is. The following events occur:

1. *B* sends *DC* a `ReadReq()`.
2. *DC* receives *B*'s `ReadReq()`.
3. *DC* sends *B* a `ReadRsp(_)`.
4. *B* receives *DC*'s `ReadRsp(_)`.

(viii) [3 pts] What is the blank character (_) that was sent in the `ReadRsp(_)` in the above events? In other words, what value does *B* actually get from its second `read()`?

○ 0 ● *X* ○ *Y* ○ *Z* ○ Not in *B*'s cache

(ix) [1 pt] Considering the example broken down in the previous parts, should *A* send multiple write requests at any time for the same `write()` function call in its code?

○ Yes ● No

Q9. [24 pts] Memory Consistency and Synchronization

(a) Memory Consistency True/False

For parts (i) through (iv), determine whether each given statement about memory consistency is true or false. Explain your reasoning **in no more than two sentences**.

(i) [1 pt] Memory consistency models are not applicable to uniprocessor systems.

- ○ True
- ● False

A single processor can still have multiple threads that can share memory and thus, warrants a memory consistency model.

(ii) [1 pt] Memory consistency models are not applicable to systems without caches.

- ○ True
- ● False

Though caches can certainly complicate implementations of memory consistency models, they are not the reason for existence of memory consistency models. Memory consistency models pertain to the order of execution of loads and stores in a single processor, and interactions with other processors can affect the memory values.

(iii) [2 pts] On an multicore system with 4 processors that utilize out-of-order completion, it is possible to implement sequential consistency.

- ● True
- ○ False

An example of this would be if the out-of-order cores implemented in-order commit of instructions, and memory operations directly entered a FIFO queue connected to main memory.

(iv) [2 pts] Adding a data prefetch unit alters the behavior of a sequentially consistent system.

- ● True
- ○ False

Although the viable results of the code on the sequentially consistent system will not change, the **probability** of the results will change. The orderings of execution are dependent on the speed of loading and storing data in memory, which the data prefetch unit affects.

(b) Building a Strong Fence

(i) [2 pts] In **no more than 2 sentences**, explain how the code below can digress from the desired functionality – refer to the commented code to understand the goal. Assume the processor abides by a **weak memory consistency model** (fully relaxed constraints).

> `Ready` can be set to 1 without `A` being updated, allowing `B` to be updated with the wrong value of `A`. Since there is no coded relationship between `Ready` and `A`, the processor can set `B` to `A` before checking whether `Ready` equals 1.

(ii) [4 pts] **Optimally insert fences in the code below for it to achieve the desired functionality.**

Recall that `fence w,r` means that all write instructions prior to the fence must complete before all read instructions after the fence. Combining constraints into one fence instruction will count as multiple fences; `fence w, wr` will count as two fences, for instance. So optimally inserting fences would require choosing minimally invasive fences. Write in your fence instructions (with proper syntax) between the lines of assembly code below.

Note that `x2` and `x3` in both P1 and P2 point to the memory address for `A` and `Ready` respectively. Note that `x6` in P2 points to the memory address for `B`.

```
P1                                          P2

li x1  1                                    li  x1  1

sw x1  0(x2)   #A = 1            loop:      lw  x5  0(x3)    #While (Ready != 1);

sw x1  0(x3)   #Ready = 1;                  bne x5  x1  loop

                                            lw  x4  0(x2)

                                            sw  x4  0(x6)    #B = A
```

```
P1
li x1 1
sw x1 0(x2) #A = 1
FENCE W, W
sw x1 0(x3) #Ready = 1;

P2
li x1 1
loop: lw x5 0(x3x) #While (Ready != 1);
bne x5 x1 loop
FENCE R, R
lw x4 0(x2)
sw x4 0(x6) # B = A
```

(c) [12 pts] **Implementing Synchronization Primitives**

Recall the **load-reserved** and **store-conditional** synchronization primitives discussed in lecture. Your pet hamster gives you the following two RISC-V atomic instructions, and tasks you with implementing a lock function for critical sections of code in his new indie video game. The instructions use special register(s) to hold the reservation flag and address, and the outcome of store-conditional.

lr.w rd, rs1

- R[rd] = M[R[rs1]]
- place reservation on M[R[rs1]]

sc.w rd, rs1, rs2

- if M[R[rs1]] is reserved, then R[rd] = 0 and M[R[rs1]] = R[rs2]
- else, R[rd] = 1

(i) [4 pts] The first step is to implement the EXCH function, which uses the load-reserved and store-conditional synchronization primitives to atomically exchange the value stored in **Mem[a0]** with **a1**. Fill in the first empty box with the instruction that's supposed to be in [BLANK 1], and the second empty box with the instruction that's supposed to be in [BLANK 2].

```
// Arguments:
//     a0: The memory address for the atomic exchange
//     a1: The value to be atomically written to Mem[a0]
// Returns:
//     a0: The previous value of Mem[a0]

EXCH:
    lr.w t0, a0
    [BLANK 1]
    [BLANK 2]
    mv a0, t0
    ret
```

BLANK 1

sc.w t1, a0, a1

BLANK 2

bnez t1, EXCH

(ii) [3 pts] With this new atomic exchange synchronization function, you work with your hamster to develop the following lock function for critical sections of code:

```
// Arguments:
//     a0: The memory address of the lock

LOCKIT:
    addi, sp, sp, -8
    sw ra, 0(sp)
    sw s0, 4(sp)
    mv s0, a0

spin:
    mv a0, s0
    li a1, 1
    jal ra, EXCH
    bnez a0, spin

    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 8
    ret
```

However, you begin to notice significant performance issues in certain sections of the code when multiple threads are competing for a lock. You are currently running the game on a potato which implements the **MSI** (Modified, Shared, Invalid) cache coherence protocol. Why might the above lock function not be ideal for our particular coherence setup? **Explain using (2) sentences max**.

Under the MSI cache coherence protocol, each processor would be generating write requests during the EXCH function and thus invalidating each other during the spin procedure. This would generate excessive bus traffic and degrade performance.

(iii) [5 pts] Concerned about the portability of the new indie game to platforms that implement MSI coherence, you work with your hamster to implement a new LOCKIT function that avoids the issue above. Fill in the corresponding blanks in the skeleton below to complete the LOCKIT function.

```
// Arguments:
//     a0: The memory address of the lock

LOCKIT:
    addi, sp, sp, -8
    sw ra, 0(sp)
    sw s0, 4(sp)
    mv s0, a0

spin1:
    mv a0, s0
    li a1, 1
spin2:
    [BLANK 1]
    [BLANK 2]
    [BLANK 3]
    bnez a0, spin1

    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 8
    ret
```

BLANK 1

lw t0, 0(s0)

BLANK 2

bnez t0, spin2

BLANK 3

jal ra, EXCH

Appendix A. A Cheat Sheet for the Bus-based RISC-V Implementation

For your reference, we have reproduced the bus-based datapath diagram as well as a summary of some important information about microprogramming in the bus-based architecture.

Remember that you can use the following ALU operations:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B

Table H1-2: Available ALU operations

Also remember that μBr (*micro*branch) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. If S, the μPC spins if *busy?* is asserted, otherwise goes to (*current state* +1).