1	/24
2	/20
3	/18
4	/20
5	/18

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Fall 2025

Quiz #3

Name	*Athena login name*	*Score*
SOLUTIONS		

Recitation section

□ WF 10, 34-301 (Jan)	□ WF 2, 34-302 (Varun)	□ WF 12, 34-303 (Nathan)
□ WF 11, 34-301 (Jan)	□ WF 3, 34-302 (Varun)	□ WF 1, 34-303 (Nathan)
□ WF 12, 34-302 (Abdullah)	□ WF 10, 34-302 (Christina)	□ WF 2, 34-303 (Grace)
□ WF 1, 34-302 (Abdullah)	□ WF 11, 34-302 (Christina)	□ WF 3, 34-303 (Grace)
		□ opt-out

Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for potential partial credit. You can use the extra white space and the back of each page for scratch work.

Problem 1. Operating Systems (24 points)

Two processes, A and B, run the RISC-V programs shown below. Code listings use virtual addresses. All pseudoinstructions in these programs translate into a single RV32I instruction.

Process A

```
.= 0x0
li a0, 0
li a1, 0x50
li a7, 0
loopA:
  slli a3, a0, 2
  lw a3, 0x50(a3)
  add a7, a7, a3
  lw a3, 0x50(zero)
  mv t0, a0
  mv t1, a1
  div a7, a3, a1
  mv a0, t0
  mv a1, t1
  bge a7, a1, end
  mv a3, a0
  addi a0, a0, 1
  blt a3, a1, loopA
end:
  ret
```

Process B

```
.= 0x0
li t0, 0x500
loopB:
  addi t1, t1, 1
  li a0, 0x50
  li a7, 0x1B
  ecall
  bneq t0, t1, loopB
ret

.= 0x50
  .ascii "Hello from Process B\n"
```

The processes run on an operating system that supports segmentation-based virtual memory. Virtual addresses are translated with the following base and bound registers:

- Process A: **base register** = 0x100, **bound register** = 0x100
- Process B: **base register** = 0x200, **bound register** = 0x100

The `print_string` system call has syscall number `0x1B` and takes the address of a string to print as its argument. For this problem, assume that system calls do not modify the value of any registers.

All instructions that are not in RV32I are illegal instructions, and illegal instructions can be emulated in software. **All registers and unspecified memory start with a default value of zero.**

(A) (2 points) What is the **physical address** of the following instructions/data:

Physical Address of slli instruction: 0x10C

Physical Address of start of string "Hello from Process B\n": 0x250

(B) (4 points) Below are some instructions. If the instruction ever causes an exception from the list below, specify the cause of the exception. Otherwise, write none.

Possible exceptions:

- Division by zero
- Illegal instruction
- System call
- Memory fault

Exception caused by `lw a3, 0x50(a3)` in Process A: Memory Fault

Exception caused by `lw a3, 0x50(zero)` in Process A: None

Exception caused by `div a7, a3, a1` in Process A: Illegal Instruction

Exception caused by `addi a0, a0, 1` in Process A: None

For parts (C) and (D), assume that:

- Instructions are executed on a single-cycle processor
- System calls do not change the values of any registers
- Illegal instructions are emulated in software
- All registers and unspecified memory start with a default value of zero

(C) (9 points) Every 20ms the processor triggers a timer interrupt and the OS kernel then performs a context switch between processes A and B. For this problem, we measure only userspace execution time. Time spent in the kernel (including system calls, exception handling, and interrupt handling) does not count toward the timing.

Assume that process A starts execution first and each userspace instruction takes 2ms to complete, even if it causes an exception. Specifically, at a timestamp of 0ms, the first instruction in process A is just about to begin, and at a timestamp of 2ms, the first instruction in process A has finished and the second instruction has not yet started.

At each **userspace** timestamp, determine which process is executing, which instruction has just completed execution, and what the values of the a0 and a7 registers are. Write "CAN'T TELL" if you can't tell a value from the information given.

	At Timestamp 26ms	**At Timestamp 30ms**	**At Timestamp 50ms**
Current Process	B	B	A
Completed Instruction	li a0, 0x50	ecall	addi a0, a0, 1
a0	0x50	0x50	1
a7	0	0x1B	0

(D) (3 points) We run the processes for some period of time before we kill them. We observe that we never run into a memory fault, and that we have printed out "Hello from Process B" 6,191 times. We discover that this is due to some error in our kernel. Why might we see this behavior? **Select all that are possible from below:**

(a) Upon returning to the userspace, exception handler sets pc to (mepc - 4)

(b) Upon returning to the userspace, exception handler sets pc to (mepc)

(c) Upon returning to the userspace, exception handler sets pc to (mepc + 4)

(d) Exception handler does not save or restore registers correctly

(e) None of the above

(E) (6 points) Assume we have the following `toy_handler` code for handling exceptions. You do not need to worry about the actual code in `toy_handler`, but may assume it works as expected and that it always returns to the instruction immediately after the instruction that caused the exception. The code for Process B is also included for reference.

Process B

```
.= 0x0
li t0, 0x500
loopB:
  addi t1, t1, 1
  li a0, 0x50
  li a7, 0x1B
  ecall
  bneq t0, t1, loopB
ret

.= 0x50
  .ascii "Hello from Process B\n"
```

Exception Handler (kernel space)

```
toy_handler:
  csrw mscratch, a1
  csrr a2, mepc
  mret

  and a2, a2, a1
  slli a2, a2, 1
```

For this question, you may assume that we are no longer considering any timer interrupts. Moreover, assume we use a standard 5-stage **pipelined** processor with **full bypassing** and branch annullment, where branches are predicted not taken, and `mret` behaves like a branch instruction in that it gets resolved in the EXE stage.

Process B is running and has just entered the loop. Fill in the pipeline diagram below beginning with the `ecall` instruction, which we assume is fetched at cycle 100. Assume exceptions are **handled lazily** (i.e. at the commit point). Use "?" if you cannot tell what instruction to write in a stage. No need to show used bypasses, if any. *Note: Extra copies of the pipeline diagram are provided at the end of the exam.*

We repeat the code below for your convenience:

Process B

```
.= 0x0
li t0, 0x500
loopB:
  addi t1, t1, 1
  li a0, 0x50
  li a7, 0x1B
  ecall
  bneq t0, t1, loopB
ret

.= 0x50
  .ascii "Hello from Process B\n"
```

Exception Handler (kernel space)

```
toy_handler:
  csrw mscratch, a1
  csrr a2, mepc
  mret

  and a2, a2, a1
  slli a2, a2, 1
```

	100	101	102	103	104	105	106	107	108	109
IF	**ecall**	bneq	ret	?	csrw	csrr	mret	and	slli	bneq
DEC		ecall	bneq	ret	NOP	csrw	csrr	mret	and	NOP
EXE			ecall	bneq	NOP	NOP	csrw	csrr	mret	NOP
MEM				ecall	NOP	NOP	NOP	csrw	csrr	mret
WB					NOP	NOP	NOP	NOP	csrw	csrr

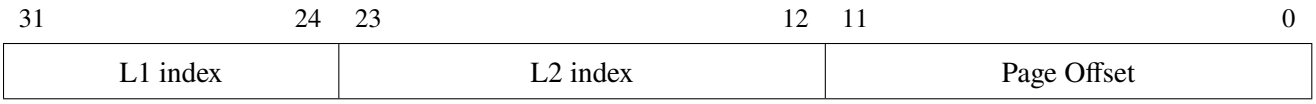
The following pipeline diagram where `ecall` does not get annulled in cycle 104 is also accepted because the `ecall` instruction itself will not write any registers or memory, so allowing it to continue to the Writeback stage does not change the processor's state.

	100	101	102	103	104	105	106	107	108	109
IF	**ecall**	bneq	ret	?	csrw	csrr	mret	and	slli	bneq
DEC		ecall	bneq	ret	NOP	csrw	csrr	mret	and	NOP
EXE			ecall	bneq	NOP	NOP	csrw	csrr	mret	NOP
MEM				ecall	NOP	NOP	NOP	csrw	csrr	mret
WB					ecall	NOP	NOP	NOP	csrw	csrr

Problem 2. Virfual Memory (20 points)

You are a TA for a computer architecture class at your college. You have been tasked to set up the class website, so you write a simple web server that takes web requests and serves lab handout files to students. The server runs on a RISC-V processor with the following specifications:

- 32-bit virtual addresses
- 26-bit physical addresses
- Two-level page table system
- Level 1 (L1) page table uses 8 bits for indexing
- Level 2 (L2) page table uses 12 bits for indexing
- Page size is 4 KB (1 KB = 1024 Bytes)



(A) (6 points) Both levels of page table's entries consist of two status bits (a resident and a dirty bit) and the PPN. Calculate the following:

Number of entries in one L1 table: 256 or 2^8

Number of entries in one L2 table: 4096 or 2^{12}

Size of one entry (*in bytes*): 2

Total size of one L1 table (*in bytes*): 512

Total size of one L2 table (*in bytes*): 8192

The page tables live in main memory. When a page table is created, the OS first checks how many pages are needed to store the table, say n. It then allocates n free pages and inserts an empty table into that space. If the table does not fit exactly into n pages, the remaining space on the last page is not used for anything else and is considered as a part of the pagetable's memory overhead.

(B) (2 points) Calculate the number of pages required to store a table of each level. If the answer is a fraction, round up to the nearest integer.

Number of pages for one L1 table: 1

Number of pages for one L2 table: 2

The server code occupies **8KB** in memory, you can assume this includes all the necessary local variables as well. Each lab handout file also takes up **8KB** in memory. The class material is laid out in the virtual address space as described below, with each range inclusive of the start and end addresses:

Server Code:	0x00000000 – 0x00001FFF
Lab 1 handout:	0x01000000 – 0x01001FFF
Lab 2 handout:	0x02000000 – 0x02001FFF
Lab 3 handout:	0x03000000 – 0x03001FFF
Lab 4 handout:	0x04000000 – 0x04001FFF
Lab 5 handout:	0x05000000 – 0x05001FFF
Lab 6 handout:	0x06000000 – 0x06001FFF
Lab 7 handout:	0x07000000 – 0x07001FFF

You plan to deploy your web server to the cloud, and you need to configure the maximum amount of memory your server can use. The server code and the handouts together total 64KB of memory. You decide to be safe and set the maximum memory to 128KB and release the website to the students.

Very soon after deployment, you notice that the server overflows its memory limit and crashes. Students report that they cannot access the labs anymore, and it is your job to fix the problem. Your friend is suspicious of the virtual memory layout so you decide to investigate.

(C) (3 points) How many L2 page tables are needed to map all the lab files and the server code in the current layout?

Number of L2 page tables: 8

(D) (3 points) How much memory is being used by the two level page table? *You can leave your answer as a product or exponent.*

Total memory consumed by page tables (*in KB*): 68

One page for the L1 table which is 4KB and two pages for each of the 8 L2 tables for a total of 16 pages or 64KB. So the L1 and L2 together are 68KB.

Your friend suggests that you can reduce the memory consumed by the page tables by moving the handout files to different locations in the virtual address space. You set out to find a new layout that minimizes the memory used by the pagetables.

(E) (6 points) Propose a new layout for the lab handout files in the virtual address space that minimizes the memory used by the page tables. For each file, provide its new starting virtual address, the files should not overlap. Then calculate the memory usage for your new layout. *You can leave your answer as a product or exponent.*

New starting address for the code: 0x00000000

New starting address for Lab 1: 0x00002000

New starting address for Lab 2: 0x00004000

New starting address for Lab 3: 0x00006000

New starting address for Lab 4: 0x00008000

New starting address for Lab 5: 0x0000A000

New starting address for Lab 6: 0x0000C000

New starting address for Lab 7: 0x0000E000

There are many possible layouts that work, this is just one example. Generally, the top 8 bits must be the same and the next 12 bits must be unique across files. Note that the if it is the 13th bit that is different, it must jump by at least 0x2000 to avoid overlap.

Number of L2 page tables needed in new layout: 1

Total memory consumed by page tables (*in KB*): 12

Problem 3. Coffee Shop Synchronization (18 points)

A busy specialty café runs several barista threads that make drinks and a single server thread that serves them. There is exactly **one espresso machine**, which only the `pull_espresso_shot()` call uses. Likewise, there is exactly **one milk steaming machine**, which only the `steam_milk()` call uses. If two baristas try to use the same machine, they collide and ruin both drinks.
There is also a **pickup counter** that can hold at most **12 finished drinks** waiting for customers. Multiple baristas or the server can access the counter at the same time. The server takes finished drinks, one at a time, from the counter and serves them. The server cannot take a drink from an empty counter.
The café tracks a "total latte art score" for all drinks made during the shift.

```
Shared Memory:
int total_art_score = 0;
```

barista code

```
barista:
  pull_espresso_shot();
  steam_milk();
  pour_latte_art();

  int drink_rating = score_art();
  int total  = total_art_score + drink_rating;
  total_art_score = total;

  place_on_counter();
  goto barista;
```

server code

```
serve_loop:
  take_from_counter();
  serve_to_customer();
  goto serve_loop;
```

(A) (6 points) Three baristas, Bean, Crema, and Foam, run the `barista` code above concurrently with **no** additional synchronization. For each scenario below, circle whether it is **Possible** or **Not Possible**, and briefly **explain your answer**.

1. Initially `total_art_score == 5`. Bean has `drink_rating = 1`, Crema has `drink_rating = 3`. After both finish their `total_art_score` updates, `total_art_score == 5`.

Possible **Not Possible**

Explain:
Each barista performs a read–modify–write on `total_art_score`. Starting from 5, the two updates can only write 6 (5+1), 8 (5+3), or 9 (if both increments are preserved). There is no interleaving where the final value stays 5 after either barista completes its write.

2. Initially `total_art_score == 0`. Bean has `drink_rating = 2`, Crema has `drink_rating = 4`, Foam has `drink_rating = 1`. After all three finish their `total_art_score` updates, `total_art_score == 4`.

Possible **Not Possible**

Explain:
This can happen if Crema's update is the last write and she read `total_art_score == 0`. For example, if Crema reads first (seeing 0), then Bean and Foam do their read/modify/writes in any order, and finally Crema writes her saved value `0 + 4`, the final value will be 4.

3. Bean and Crema try to use the espresso machine at the same time and collide at the machine, ruining both drinks.

Possible **Not Possible**

Explain:
There is no mutual exclusion around `pull_espresso_shot()`, so two baristas can enter this call concurrently even though there is only one machine.

(B) (12 points) Now the manager wants to enforce proper coordination using **semaphores**. The system must satisfy:

- At most **one** barista may be in `pull_espresso_shot()` at a time (only one espresso machine).
- At most **one** barista may be in `steam_milk()` at a time (only one milk steaming machine).
- The pickup counter holds at most **12 drinks** at any time, and the server never takes from an empty counter.
- Baristas may still prepare drinks (pulling espresso, steaming milk, pouring latte art, calling `score_art()`) even when the counter is full; they should only be blocked when trying to place a finished drink on the counter (`place_on_counter()`). Multiple baristas or the server can access the counter at the same time.
- The value of `total_art_score` must equal the sum of all `drink_rating` for drinks up to that point.
- Locks that protect shared variables should be held for the shortest reasonable code region (do not wrap more code in a lock than necessary).
- There should be **no deadlocks**.
- You may not introduce extra precedence constraints beyond what is required by the conditions above.
- You may use at most **5 semaphores**, and you may **not** initialize any semaphore to a negative value.

You may only add semaphore declarations and initial values in **Shared Memory**, and `wait(sem)` and `signal(sem)` calls in the code below. Semaphore calls should be written on their own lines between existing statements. You may not change the order of the given statements, and you may not add new shared variables other than semaphores. Complete the code.

Declare semaphores below

```
Shared Memory:
int total_art_score = 0;
// Specify your semaphores and initial values here:
semaphore espresso_sem = 1;
semaphore milk_sem = 1;
semaphore counter_slots = 12;
semaphore drinks_ready = 0;
semaphore score_lock = 1;
```

barista code

```
barista:

  wait(espresso_sem);
  pull_espresso_shot();
  signal(espresso_sem);


  wait(milk_sem);
  steam_milk();
  signal(milk_sem);


  pour_latte_art();


  int drink_rating = score_art();


  wait(score_lock);
  int total = total_art_score + drink_rating;
  total_art_score = total;
  signal(score_lock);


  wait(counter_slots);
  place_on_counter();
  signal(drinks_ready);


  goto barista;
```

server code

```
serve_loop:


  wait(drinks_ready);
  take_from_counter();
  signal(counter_slots);


  serve_to_customer();


  goto serve_loop;
```

Explanation:

- `espresso_sem = 1`: ensures at most one barista is in `pull_espresso_shot()` at a time, modeling the single espresso machine.
- `milk_sem = 1`: ensures at most one barista is in `steam_milk()` at a time, modeling the single milk steaming machine.
- `counter_slots = 12` and `drinks_ready = 0`: implement a bounded buffer of size 12 between baristas and the server. Baristas only block at `place_on_counter()` when no slots remain; the server

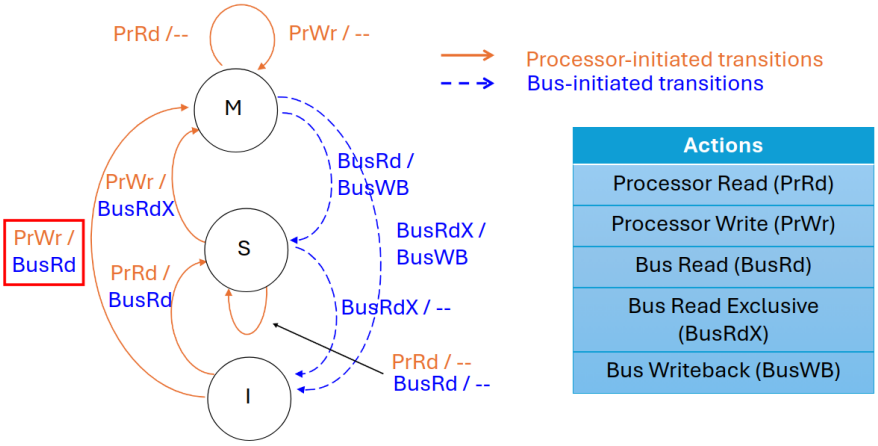
blocks on `drinks_ready` when there are no drinks to serve.

- `score_lock = 1`: protects the read–modify–write on `total_art_score`, so its value always equals the sum of all `drink_rating` values for drinks produced so far, while keeping the critical section as small as possible.
- There is no cyclic waiting pattern among the semaphores, so the solution avoids deadlock and does not add unnecessary precedence constraints beyond those required.

Problem 4. Cache Coherence (20 points)

Ben Bitdiddle is designing a 2-core processor system in which each core, `P1` and `P2`, has its own private cache. He is implementing a snoopy-based, write-invalidate MSI coherence protocol, but unfortunately he has made a bug. *At the end of the problem you can find a diagram of a correct MSI protocol and an additional table for your convenience.*

(A) (10 points) The bug in Ben's implementation occurs when a processor performs a write while in the I state: instead of initiating a BusRdX transaction, it incorrectly initiates a BusRd. The state transition diagram for Ben's protocol is shown below, with the incorrect transition highlighted in a red box. All other transitions correctly follow the standard MSI protocol.



Using Ben's protocol, fill in the following table showing the cache line state for A after each access. For each bus transaction, specify which processor initiated it (e.g., P1: BusRd(A)), or write NONE if there are no bus transactions for a given access. Not specifying which processor initiated the request will not receive full credit. Then answer the questions below the table.

Access	*Shared bus transactions*	*P1's cache*	*P2's cache*
Initial state		A: I	A: I
`I0:` After P1 writes A	P1: BusRd(A)	A: M	A: I
`I1:` After P2 reads A	P2: BusRd(A); P1: BusWB(A)	A: S	A: S
`I2:` After P2 writes A	P2: BusRdX(A)	A: I	A: M
`I3:` After P1 writes A	P1: BusRd(A); P2: BusWB(A)	A: M	A: S
`I4:` After P2 reads A	NONE	A: M	A: S
`I5:` After P1 reads A	NONE	A: M	A: S

Select the correct option and list all load accesses from the above sequence (e.g., `I0`, `I1`, etc.) which return stale data, or write NONE if there are no such accesses.

Due to this bug some loads return stale data (select one):

Yes **No**

Affected accesses: I4

Select the correct option and list all accesses from the above sequence (e.g., `I0`, `I1`, etc.) whose hit/miss status got changed due to the bug, i.e. accesses which:

- Result in a cache miss, but would have been a hit under a correct MSI protocol, or
- Result in a cache hit, but would have been a miss under a correct MSI protocol

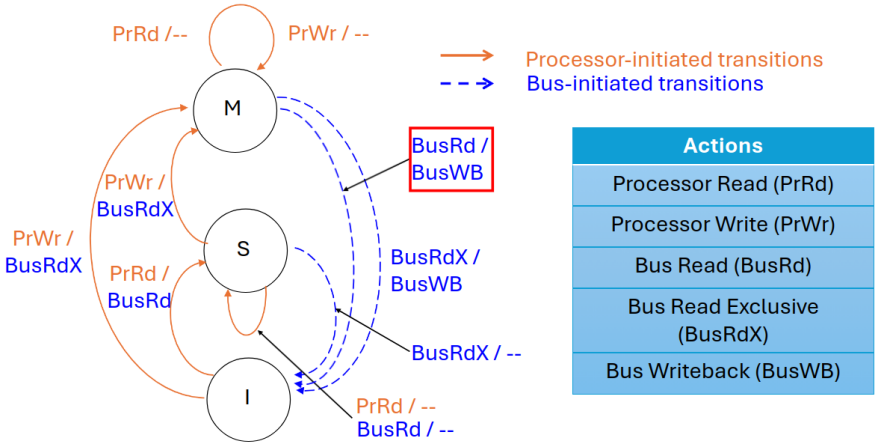
or write NONE if there are no such accesses.

Due to this bug the system experiences a different hit ratio (select one):

Lower Higher **No difference**

Accesses whose hit/miss status got changed: I4

(B) (10 points) Ben was able to track down and fix his first bug, but in the process he introduced a new one. With this new bug, a Bus Read received while in the M state incorrectly transitions the cache line to the I state, rather than the correct S state. The state transition diagram for Ben's updated protocol is shown below, with the incorrect transition highlighted in a red box. All other transitions correctly follow the standard MSI protocol.



Using Ben's updated protocol, fill in the following table showing the cache line state for A after each access. For each bus transaction, specify which processor initiated it (e.g., P1: BusRd(A)), or write NONE if there are no bus transactions for a given access. Not specifying which processor initiated the request will not receive full credit. Then answer the questions below the table.

Access	*Shared bus transactions*	*P1's cache*	*P2's cache*
Initial state		A: I	A: I
`I0:` After P1 writes A	P1: BusRdX(A)	A: M	A: I
`I1:` After P2 reads A	P2: BusRd(A); P1: BusWB(A)	A: I	A: S
`I2:` After P2 writes A	P2: BusRdX(A)	A: I	A: M
`I3:` After P1 writes A	P1: BusRdX(A); P2: BusWB(A)	A: M	A: I
`I4:` After P2 reads A	P2: BusRd(A); P1: BusWB(A)	A: I	A: S
`I5:` After P1 reads A	P1: BusRd(A)	A: S	A: S

Select the correct option and list all load accesses from the above sequence (e.g., `I0`, `I1`, etc.) which return stale data, or write NONE if there are no such accesses.

Due to this bug some loads return stale data (select one):

Yes **No**

Affected accesses: NONE

Select the correct option and list all accesses from the above sequence (e.g., `I0`, `I1`, etc.) whose hit/miss status got changed due to the bug, i.e. accesses which:

- Result in a cache miss, but would have been a hit under a correct MSI protocol, or
- Result in a cache hit, but would have been a miss under a correct MSI protocol

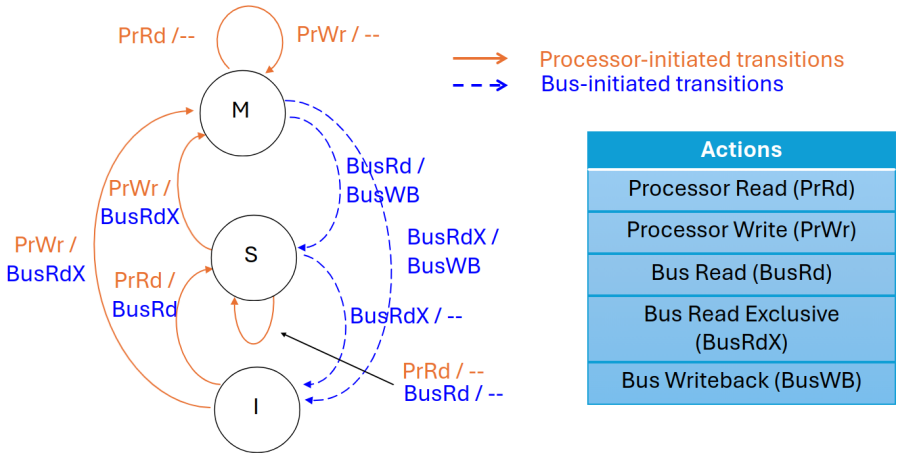
or write NONE if there are no such accesses.

Due to this bug the system experiences a different hit ratio (select one):

Lower **Higher** **No difference**

Accesses whose hit/miss status got changed: `I5`

This table is not graded.
You may wish to fill out the chart below assuming a valid MSI for comparison with the faulty MSI protocols.



Access	*Shared bus transactions*	*P1's cache*	*P2's cache*
Initial state		A: I	A: I
`I0:` After P1 writes A		A:	A:
`I1:` After P2 reads A		A:	A:
`I2:` After P2 writes A		A:	A:
`I3:` After P1 writes A		A:	A:
`I4:` After P2 reads A		A:	A:
`I5:` After P1 reads A		A:	A:

Problem 5. Performance Engineering and Loops (18 points)

We compute a vector-matrix multiplication kernel $C = A \cdot B$, where

- $A[0 \ldots n-1]$ is a n-element vector,
- $B[0 \ldots n-1][0 \ldots n-1]$ is a $n \times n$ matrix, and
- $C[0 \ldots n-1]$ is the output n-element vector.

Each element in these arrays is 4 bytes (one word). The matrix B is stored in row-major order, and vector elements are all stored consecutively in memory.
We run the program on a cache with the following specification:

- Block size: 4 words (so 4 elements per block).
- Associavity and Capacity: 4 sets, 4 ways (total cache capacity = $4 \times 4 \times 4 = 64$ words).

We show two versions of code to implement this kernel.

Version 1 (outer loop over different rows in B)	**Version 2 (outer loop over different columns in B)**

`// Version 1`
`for (int i = 0; i < n; ++i) {`
`  for (int j = 0; j < n; ++j) {`
`  C[j] += A[i] * B[i][j];`
`  }`
`}`
	`// Version 2`
	`for (int j = 0; j < n; ++j) {`
	`  for (int i = 0; i < n; ++i) {`
	`  C[j] += A[i] * B[i][j];`
	`  }`
	`}`

(A) (2 points) Assume (for questions A and B) that each array (A, B, and C) is serviced by its *own* cache (no interference between arrays). Use $n = 2048$.

Which version has better cache performance? **Version 1** **Version 2**

Explain:
Version 1 is better because it streams rows of B and C sequentially (good spatial locality), whereas Version 2 repeatedly accesses B with a large stride (bad locality), causing many more misses.

(B) (8 points) Two-dimensional tiling. Still assume the three arrays are serviced from seprate caches. Now suppose you are allowed to tile *both* dimensions of matrix B, using the same tile size t. That is, you operate on a small submatrix (a $t \times t$ "tile") of B and the corresponding portions of A and C before moving to the next tile.

Pick the tile size so that one tile of A, B, C can all fit in their own caches. Still use $n = 2048$. Recall the cache configuration below:

- Block size: 4 words (so 4 array elements per block).
- Associavity and Capacity: 4 sets, 4 ways (total cache capacity = $4 \times 4 \times 4 = 64$ words).

Tile size t = 4

We need to consider both capacity misses and conflict misses. When the tile size is 4, each row of the tile occupies exactly one cache block, and all of these blocks map to the same cache set.
For example, suppose matrix B begins at an address ending in 0000. In the first tile, the elements B[0][0] to B[0][3] occupy one cache block that maps to set 0. The next row, B[1][0] to B[1][3], also maps to set 0, and so does every other row in the tile. As long as the tile has no more than 4 rows, all rows fit into the 4-way associative set without eviction.
However, when the tile size becomes greater than 4, meaning the tile contains more than 4 rows, the fifth row's block must also map to the same cache set. Since the set can hold only 4 blocks, bringing in this fifth block will evict one of the existing ones, resulting in conflict misses.

Write the outer tile loops (i_t,j_t) in the order that maximizes reuse of B's tile. You can ignore cases where n does not divide evenly by t.

```
// Tiled version
for ( int i_t = 0; i_t < n; i_t += t )
  for (int j_t = 0; j_t < n; j_t += t)
    for (int i = i_t; i < i_t + t; i++)
      for (int j = j_t; j < j_t + t; j++)
        C[j] += A[i] * B[i][j];
```

If you are allowed to choose different tile sizes for the row and column dimensions of matrix B, how should you choose them to make the tile as large as possible while still fitting in the cache? Select one from below.

Same number of columns and rows in a tile **More columns in a tile** **More rows in a tile**

(C) (2 points) Real-world complication: shared cache. In practice, all arrays (A, B, and C) share the same cache rather than being stored in separate caches. As a result, we must consider two kinds of cache misses:

- **Capacity misses:** occur when the total working set of the inner loops exceeds the total cache capacity, even if the cache is fully associative.
- **Conflict misses:** occur when multiple memory blocks map to the same cache set and evict each other, even though the cache as a whole is not full.

For now, consider only *capacity misses*. Assume a 64-word *fully-associative* cache (no set conflicts). The cache block size is the same as before, 4 words per cache block. What tile size t should you choose so that all the array elements (including all the three arrays) accessed by the two innermost loops (iterating over i and j, not i_t or j_t) in question (B) can fit in the cache? *Note that tile size does not need to be a power of 2.*

The following table can help you think and solve the problem:

Number of Cache Blocks Occupied by Each Array (Will Not Grade)			
Array	tile size = 6	tile size = ____	tile size = ____
A	2		
B	12		
C	2		

Max tile size to avoid capacity misses: 6

(D) (6 points) Conflict misses. With a tile size of 4, now analyze potential *conflict misses* when the cache is no longer fully associative (4 sets, 4 ways, 4 elements per cache block). Assume the starting address of the three arrays A, B, C all end with `0x0000`. Use the same array size as before $n = 2048$.
Fill in the table below to indicate which arrays suffer from conflict misses in each cache set. Analyze the case when $i\_t = 0$ and $j\_t = 8$. Write "None" if no array elements are mapped to the corresponding set.

Cache Set	**Arrays Mapping to This Set**	**Likely Conflict Misses (Yes/No)**
Set 0	A	No
Set 1	None	No
Set 2	B, C	Yes
Set 3	None	No

Will the conflict pattern (number of misses per tile) change for different tiles? (select one or multiple)

No Change **Change for different i_t** **Change for different j_t**