1	/18
2	/18
3	/16
4	/16
5	/16
6	/16

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Spring 2026

Quiz #3

Name	*Kerberos*	*Score*
Solutions		

Recitation section

☐ WF 10, 34-302 (Jan)	☐ WF 2, 34-302 (Abdullah)	☐ WF 12, 35-308 (Nathan)
☐ WF 11, 34-302 (Jan)	☐ WF 3, 34-302 (Abdullah)	☐ WF 1, 35-308 (Nathan)
☐ WF 12, 34-302 (Qihang)	☐ WF 10, 35-308 (Raul)	☐ WF 2, 8-205 (Aarush)
☐ WF 1, 34-302 (Qihang)	☐ WF 11, 35-308 (Raul)	☐ WF 3, 8-205 (Aarush)
		☐ 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 & Virtual Memory (18 points)

Consider two processes running on a RISC-V processor with virtual memory. The system has a 1-level page table per process and a 2-entry fully associative TLB that uses LRU replacement. The page size is 256 (2^8) bytes. Virtual addresses are 16 bits and physical addresses are 12 bits. The TLB is flushed on every context switch.

For each of the following questions (A–F), either provide the requested answer, or if there is not enough information to determine the answer, then write CAN'T TELL. Assume all memory accesses are legal and assume that neither process has run yet.

Initial state

TLB: initially empty.

Page table entries for Process A:

VPN	**R**	**D**	**PPN**
0x10	1	0	0x3
0x11	1	0	0x7
0x20	1	0	0x5

Page table entries for Process B:

VPN	**R**	**D**	**PPN**
0x10	1	0	0x9
0x12	1	0	0x4
0x20	1	0	0x6

Initial physical memory contents:

Physical address	**Data**
0x0340	111
0x0440	555
0x0540	333
0x0640	666
0x0750	222
0x0940	444
All other physical addresses	Unknown

Assume that both Process A and Process B begin with x1 = 0x1040, x2 = 0x2040, and PC = 0x1000. **Assume that it is valid for instructions and data to live in the same page.**

Process A

```
. = 0x1000
lw x3, 0(x1)
lw x4, 0(x2)
sw x3, 0x10(x2)
lw x5, 0x10(x2)
ret
```

Process B

```
. = 0x1000
lw x3, 0(x1)
sw x3, 0(x2)
lw x4, 0(x2)
lw x5, 0x240(x1)
ret
```

(A) **(4 points)** After Process A executes its first 3 instructions, what are the values of x3 and x4, and what are the contents of the TLB (ignoring status bits) prior to any context switch?

x3: 111

x4: 333

TLB contents:

VPN	PPN
0x10	0x3
0x20	0x5

(B) **(3 points)** Immediately after (A), the OS context switches to Process B. It executes its first 2 instructions, what is the value of x3, and what are the contents of the TLB (ignoring status bits) just before the context switch back to Process A?

x3: 444

TLB contents:

VPN	PPN
0x10	0x9
0x20	0x6

(C) **(2 points)** When the OS performs context switch, which of the following pieces of state must be saved as part of the old process's state so that it can later resume correctly?
Circle all that apply.

1. Virtual PC
2. Physical PC
3. Register values
4. Physical pages used by old process
5. Processor pipeline state

(D) **(4 points)** Immediately after (B), the OS context switches to Process A until it completes execution, then switches to Process B. After Process B completes execution, what are the values of x4 and x5, and what are the final contents of the TLB (ignoring status bits)?

x4: 444

x5: CAN'T TELL

Final TLB contents:

VPN	PPN
0x10	0x9
0x12	0x4

(E) **(3 points)** Suppose the TLB were NOT flushed on a context switch, and the TLB stored only VPN → PPN translations (with no process identifier). Assume the same execution sequence and context switches of parts A-D. Determine which address translation would be the first to be incorrect as a result? Provide the virtual address together with its incorrect physical address translation as well as what the correct one should have been.

Virtual address: _____0x1000_____

Wrong physical address used: _____0x300_____

Correct physical address: _____0x900_____

(F) **(2 points)** Now suppose each TLB entry also includes a processID (PID) field, so the TLB stores (PID, VPN) → PPN and is not flushed on a context switch.

Would execution still be correct? _____Yes_____

How many misses from the original execution now become hits? _____0_____

Problem 2. Virtual Memory (18 points)

Consider a RISC-V processor that includes 2^{32} bytes of virtual memory, 2^{26} bytes of physical memory, and uses a page size of 2^{16} bytes.

(A) (2 points) Calculate the following parameters relating to the size of the page table assuming a single-level (flat) page table. Each page table entry contains a dirty bit and a resident bit. *Your final answer can be a product or exponent.*

Size of page table entry (in bits): ____12_____

Number of entries in page table: ____2^{16}_____

(B) (5 points) A program has been halted right before executing the following sequence of instructions beginning at virtual address `0x10000`.

```
. = 0x10000
   j label1

. = 0x3FFA0
label1:
   sw x4, 0(x5) // x5 = 0x5303C
   lw x3, 8(x7) // x7 = 0x67890
```

Page Table

VPN	R	D	PPN
LRU → 0	1	0	0x17
1	1	0	0x9A
2	1	1	0x41
3	0	---	---
Next LRU → 4	1	1	0x10
5	0	---	---
6	1	1	0x27
7	1	0	0x5
…			

The first 8 entries of the page table are shown above. The page table uses an LRU replacement policy. Assume that all physical pages are currently in use.

In the table below, specify all virtual addresses accessed when executing this sequence of instructions. **List the addresses in the order they are accessed.** For each virtual address, please indicate the VPN, whether or not the access results in a page fault, the PPN, and the physical address. Please write all numerical values in **hexadecimal**.

Virtual Address	VPN	Page Fault (Yes/No)	PPN	Physical Address
0x10000	0x1	No	0x9A	0x9A0000
0x3FFA0	0x3	Yes	0x17	0x17FFA0
0x5303C	0x5	Yes	0x10	0x10303C
0x3FFA4	0x3	No	0x17	0x17FFA4
0x67898	0x6	No	0x27	0x277898

(C) (2 points) Which virtual page(s), if any, need(s) to be evicted from physical memory as a result of executing the code sequence above? If a virtual page is evicted, then determine if its corresponding physical page had to be written back to disk. If so, provide the PPN that needs to be written back to disk, otherwise enter None to indicate that no write back is required.

VPN ____0x0_____ **PPN** ____None_____**written back to disk**

VPN ____0x4_____ **PPN** ____0x10_____**written back to disk**

(D) (5 points) Now consider the same RISC-V processor, but with a 4-element, **direct mapped** Translation Lookaside Buffer (TLB) running the same sequence of instructions as before. Assume that the two least significant bits of the VPN are used to index into the TLB. Assuming the contents of the TLB and page table were as shown below just prior to executing the code sequence, **update all changed rows** in the blank TLB and page table provided below with their updated values after executing the same code sequence as before. Assume that if the dirty bit gets updated in the TLB, that the corresponding dirty bit in the page table only gets updated when the VPN to PPN translation is evicted from the TLB.

TLB

index	VPN	V	R	D	PPN
0	0x10	1	1	0	0x19
1	0x19	1	1	1	0x60
2	0x6	1	1	1	0x27
3	0x33	1	1	0	0x70

Page Table

VPN	R	D	PPN
LRU → 0	1	0	0x17
1	1	0	0x9A
2	1	1	0x41
3	0	---	---
Next LRU → 4	1	1	0x10
5	0	---	---
6	1	1	0x27
7	1	0	0x5
	…		

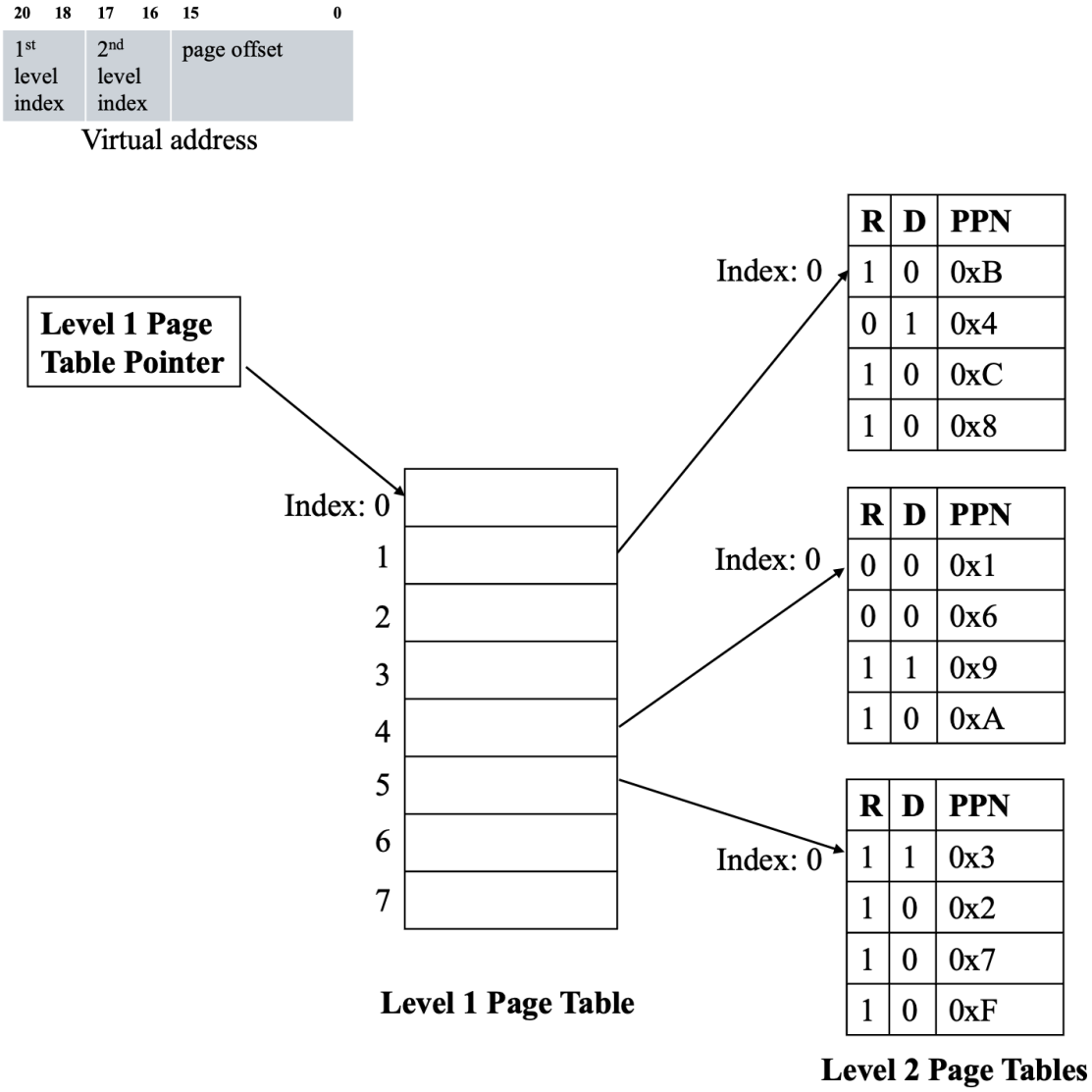
TLB

index	VPN	V	R	D	PPN
0					
1	0x5	1	1	1	0x10
2					
3	0x3	1	1	0	0x17

Page Table

VPN	R	D	PPN
0	0	---	---
1			
2			
3	1	0	0x17
4	0	---	---
5	1	0	0x10
6			
7			
	…		

(E) (4 points) Now consider a new processor with a 5-bit VPN that is translated to a PPN using the two-level hierarchical page table shown below. The top 3 bits of the VPN are used as the first level index, and the bottom 2 bits of the VPN are used as the second level index. The bottom 16 bits of the virtual address are the page offset. The values listed in the level 2 page tables are the PPNs.



Translate the following virtual addresses to physical addresses using this two-level hierarchical page table. If a virtual address does not map to a physical address according to the diagram above, then write PAGE FAULT.

Virtual Address	Physical Address
0x124400	0x94400
0x55374	PAGE FAULT
0x73899	0x83899

Problem 3. Exceptions (16 points)

Ben Bitdiddle wants to transmit messages character-by-character to his friend Alice.

He recently learned about a scheme called **run-length encoding**, where rather than having to transmit each character in a message, he can compress multiple consecutive characters by just sending the character and the number of repetitions. For example, "AABBCDDEEE" would be transmitted as (A, 2), (B, 2), (C, 1), (D, 2), (E, 3).

To do this, he writes a program in C, which defines a Message struct as follows:

```
typedef struct {
    char letter;
    int freq;
} Message;
```

*Note that the data within a struct is stored **contiguously** in memory. For the purposes of this problem, there are no layout transformations (e.g. padding) performed by the compiler.*

Ben stores a list of messages to send in a C array consisting of Message structs.

```
Message messages[3] = {{'A',0x01234567}, {'B',0x23456789},
{'C',0x3456789A}};
```

To better understand the memory layout, we have filled out the contents of the 15 bytes of memory associated with this message assuming that the message array is located at 0x1000. 'A' is 0x41, 'B' is 0x42, 'C' is 0x43.

Address	0x1009	0x1008	0x1007	0x1006	0x1005	0x1004	0x1003	0x1002	0x1001	0x1000
Data	0x23	0x45	0x67	0x89	0x42	0x01	0x23	0x45	0x67	0x41

Address	0x1013	0x1012	0x1011	0x1010	0x100F	0x100E	0x100D	0x100C	0x100B	0x100A
Data						0x34	0x56	0x78	0x9A	0x43

(A) (2 points) Ben and Alice soon realize their RISC-V processors only support *memory-aligned loads*, i.e data can only be accessed from an address which is a multiple of the data size. For example, load word instructions need to occur at addresses that are multiples of 4. Misaligned loads cause exceptions.

Alice claims that she can't receive Ben's messages because his Message struct leads to runtime errors due to misaligned loads when attempting to access elements in the messages array. **Is Alice correct? Explain why or why not.**

Yes, Alice is correct. When we try to access the `freq` attribute in a message, there is no guarantee that its address is word-aligned as `Message` structs in the array are offset by 5 bytes.

(B) (9 points) Unfortunately for Alice, Ben is too lazy to change his original code. Alice, having taken 6.191, remembers that operating systems can *emulate instructions* that are not supported by the processor. Fill in the helper function below for emulating misaligned load word instructions at the OS level. You may assume that when this helper is called, the instruction at the current PC is a valid load word instruction – the current process state curProc is passed in.

```
typedef struct {
    int pc;
    int regs[32];
    …
} ProcState;
```

Assume that, unlike lab 7, the ProcState stores all 32 registers including x0 so `curProc->regs[rd]` *holds the contents of register* `rd`. Assume you are given a function `va_to_pa` that performs virtual-to-physical address translation. Recall that the user program uses virtual addresses. Use your RISC-V ISA to help complete `misaligned_lw_eh` (misaligned lw exception handler).

```
int va_to_pa(ProcState* curProc, int va);

ProcState* misaligned_lw_eh(ProcState* curProc) {

   int pa = va_to_pa(curProc, curProc->pc); // physical address of instr
   int inst = *((int *) pa);                // instruction

   int rd = _______(inst >> 7) & 0x01F ______;
   int rs1 = ______(inst >> 15) & 0x01F _____;
   int imm = inst >> 20;

   // Determine address to load from
   int virtual_addr = ___ curProc->regs[rs1] + imm _____;
   char* phys_addr = ____(char*)va_to_pa(curProc, virtual_addr)__;

   // Get 4 bytes of data using byte-level loads
   uint8_t byte0 = _____*phys_addr ___________________;
   uint8_t byte1 = _____*(phys_addr + 1) _____________;
   uint8_t byte2 = _____*(phys_addr + 2) _____________;
   uint8_t byte3 = _____*(phys_addr + 3) _____________;

   // Combine the 4 bytes into a 32-bit data value
   int data = (int) byte0 + (int) ____(byte1 << 8)_________________
                          + (int) _____(byte2 << 16)________________
                          + (int) _____(byte3 << 24)________________;

   // Sets rd register
   curProc->regs[rd] = data;
   curProc->pc = ____ curProc->pc + 4____;
   return curProc;
}
```

(C) (5 points) Alice notices multiple instances where a misaligned load accesses data from different pages which leads to a worst-case scenario of two distinct page faults. Angry at Ben, Alice decides to assume the portion of the data in the second page is zero and *not attempt to read the data from the second page.* Modify your exception handler code from above to do this.

You may assume that the constants `PAGE_SIZE, VIRTUAL_ADDR_SPACE_SIZE, PHYS_ADDR_SPACE_SIZE`, in bytes, are accessible within this function.

```
{
   …
// Assume all code before this was correct and all defined
// variables are available for use in your code.

   int page_offset = _____((int)(phys_addr)) % PAGE_SIZE _______;

   // Get 4 bytes of data using byte-level loads

   uint8_t byte0 = ___*phys_addr ____________________________________________;

   uint8_t byte1 = ((page_offset+1<PAGE_SIZE) ? (int)(*(phys_addr+1)):0);

   uint8_t byte2 = ((page_offset+2<PAGE_SIZE) ? (int)(*(phys_addr+2)):0);

   uint8_t byte3 = ((page_offset+3<PAGE_SIZE) ? (int)(*(phys_addr+3)):0);

   … // Assume all code after this was correct
}
```

Problem 4. Synchronization (16 points)

Seeking joy outside of debugging Lab 6 code during OH, the 6.191 TAs decided to open an amusement park with their main attraction, the *Kernel Coaster*.

Before opening the park, the TAs wanted to implement two safety checks on the ride, both of which require access to the ride's control panel and gate systems. The safety checks should be able to be run concurrently.

Shared Memory:

```
semaphore control_panel = 1
semaphore gate = 1
```

safety_check_1:

```
1  wait(control_panel)
2  wait(gate)

3  complete_safety_check_1()

4  signal(gate)
5  signal(control_panel)

6  goto safety_check_1
```

safety_check_2:

```
1  wait(gate)
2  wait(control_panel)

3  complete_safety_check_2()

4  signal(control_panel)
5  signal(gate)

6  goto safety_check_2
```

(A) (2 points) One of the TAs tries writing the two safety checks shown above. However, as written, these safety checks don't work. In **at most one sentence**, explain what is wrong with the code.

The current code can lead to deadlock (e.g., process 1 calls `wait(control_panel)` then process 2 calls `wait(gate)`).

(B) (2 points) Luckily, one of the other TAs realized that the issue in part (A) could be resolved by modifying **exactly two lines** of the **Safety Check 2** code. Clearly indicate which two line numbers are changed and write the corrected versions of those lines below.

Lines in Safety Check 2 Code changed: __Lines 1 and 2__

Corrected Code:

```
wait(control_panel)
wait(gate)
```

The opening day of the amusement park turned into a disaster after the TAs did not properly set up the *Kernel Coaster* ride system. To fix the issues, the TAs recruit 6.191 students to implement a synchronization solution using semaphores in pseudocode.

The TAs intend to operate multiple coaster cars concurrently on the *Kernel Coaster*, each serving parkgoers safely and efficiently.

The *Kernel Coaster* consists of three main components:

1) Loading/Unloading Platform: where parkgoers board a coaster car
   - Only one coaster car may occupy the platform at a time
   - Parkgoers may only board/exit a car while it is on the platform
   - A coaster car may only depart the platform to begin the ride when it has **exactly** 4 passengers on board
   - A coaster car may only depart the platform to go to storage once all 4 of its riders have exited the car.
   - Multiple passengers can board/exit concurrently
2) Tracks: where the actual roller coaster ride takes place
   - Multiple coaster cars can be on the track at the same time without safety issues
   - Assume that passenger `ride()` method and its corresponding coaster `run_ride()` method complete at the same time so passengers can't `exit_car()` during the ride.
3) Storage: where the coaster cars that are not on the platform or tracks are held
   - Coaster cars must go to storage after all passengers have exited the car (i.e., they cannot immediately start loading new passengers)

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 on the following page.

Note: If you are writing multiple, consecutive lines of the same code, you may consolidate the lines using the following notation:

```
wait(sem)
wait(sem)
wait(sem)
```

May be abbreviated as

```
wait(sem) 3x
```

(C) (12 points)

Shared Memory:

```
// Specify your semaphores and initial values here
semaphore platform_available = 1
semaphore can_board = 0
semaphore boarded = 0
semaphore can_exit = 0
semaphore exited = 0
```

kernelCoasterCar:

```
wait(platform_available)

arrive_at_platform()

signal(can_board) 4x

wait(boarded) 4x

leave_platform()

signal(platform_available)

go_to_tracks()
run_ride()

wait(platform_available)

arrive_at_platform()

signal(can_exit) 4x

wait(exited) 4x

leave_platform()

signal(platform_available)

go_to_storage()

goto kernelCoasterCar
```

passenger:

```
wait(can_board)

board_car()

signal(boarded)

ride()

wait(can_exit)

exit_car()

signal(exited)

goto passenger
```

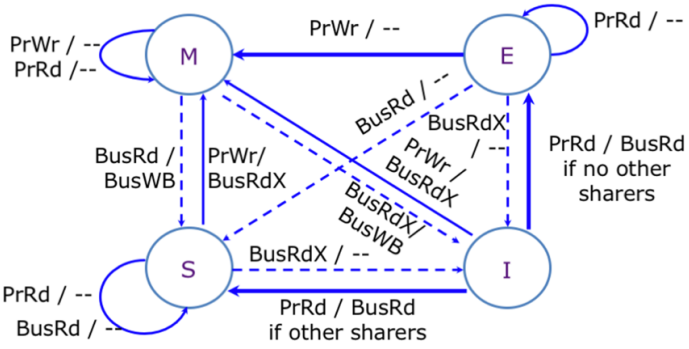
Explanation:

- `platform_available = 1`: mutex so at most one car is on the platform at any time
- `can_board = 0`: prevents passengers from boarding when there are no cars on the platform
- `boarded = 0`: prevents cars from beginning the ride when there are not exactly 4 passengers on board
- `can_exit = 0`: similar to `can_board --` prevents passengers from exiting the car during the ride
- `exited = 0`: makes sure all the passengers exit the car before going back into storage

Problem 5. Cache Coherence (16 points)

Ben Bitdiddle is building an optimized coherence protocol. His baseline is a system with two processors, P1 and P2, which have private, write-back caches that are kept coherent with a snoopy MESI protocol, shown below. Both caches have two lines and are initially empty. Consider the following sequence of accesses:

```
I0 P1: read A
I1 P2: read B
I2 P2: write A
I3 P1: read A
I4 P1: write B
I5 P2: read B
I6 P1: read A
```



(A) (6 points) Assume blocks A and B do not conflict in the cache. Fill in the following table showing the cache line states for A and B *after* each access. **For each bus transaction, specify which processor initiated it and which address it is for (e.g., P1: BusRd(A)).**

Access	*Shared bus transaction*	*Processor P1's cache*		*Processor P2's cache*	
Initial state		A: I	B: I	A: I	B: I
After P1 reads A	P1: BusRd(A)	A: E	B: I	A: I	B: I
After P2 reads B	P2: BusRd(B)	A: E	B: I	A: I	B: E
After P2 writes A	P2: BusRdX(A)	A: I	B: I	A: M	B: E
After P1 reads A	P1: BusRd(A), P2: BusWB(A)	A: S	B: I	A: S	B: E
After P1 writes B	P1: BusRdX(B)	A: S	B: M	A: S	B: I
After P2 reads B	P2: BusRd(B), P1: BusWB(B)	A: S	B: S	A: S	B: S
After P1 reads A	--	A: S	B: S	A: S	B: S

Ben implements a variant of the protocol with *delayed invalidations*: when a line is invalidated, the cache is allowed to retain read-only permission for the line until the next cache miss. Ben's specific implementation adds an invalidation buffer to the cache, and uses it to track delayed invalidations:

- When the cache observes a BusRdX request for a line in S, it keeps the line in S (instead of transitioning to I), and records the address in the invalidation buffer.
- When the cache observes a BusRdX request for a line in M or E, it transitions the line to S (and issues a BusWB if it was in M), and records the address in the invalidation buffer.
- When a cache next issues a BusRd or BusRdX for any address (i.e., it handles a miss for any address), it invalidates the cache lines of **all the addresses** in the invalidation buffer**, and clears the invalidation buffer.**

(B) (6 points) Repeat part A using this MESI protocol variant with delayed invalidations. List the contents of each cache's invalidation buffer in addition to the information part A asks for.

Access	*Shared bus transaction*	*Processor P1's cache*			*Processor P2's cache*		
		A	*B*	*Inv Buffer*	*A*	*B*	*Inv Buffer*
Initial state		I	I	---	I	I	---
After P1 reads A	`P1: BusRd(A)`	E	I	---	I	I	---
After P2 reads B	`P2: BusRd(B)`	E	I	---	I	E	---
After P2 writes A	`P2: BusRdX(A)`	S	I	A	M	E	---
After P1 reads A	`--`	S	I	A	M	E	---
After P1 writes B	`P1: BusRdX(B)`	I	M	---	M	S	B
After P2 reads B	`--`	I	M	---	M	S	B
After P1 reads A	`P1: BusRd(A),` `P2: BusWB(A)`	S	M	---	S	S	B

Recall that in coherent memory, all loads and stores to the same address can be placed in a global order (i.e., an order that is the same for all processes).

In general, accesses can follow a logical order that does not match the physical-time order in which the accesses happened. Within a process, accesses cannot be reordered, as doing so would violate sequential semantics; but because accesses from multiple concurrent processes are interleaved, accesses by different processes in this logical order need not interleave in the same way as they do in physical time. For example, consider the following sequence of accesses: (1) process P1 reads A, (2) process P2 writes A, and (3) process P1 reads A again. If the value of A that P1 reads in (3) is the old value of A (not the value written by P2 in (2)), then logically (3) occurs before (2), and there is a global logical order (1), (3), (2).

(C) (4 points) Does Ben's coherence protocol with delayed invalidations maintain coherence for the sequence of accesses from part (B)? If so, specify a valid global logical order for the sequence of accesses. If not, explain why not. The sequence of accesses is repeated below, with a number indicating the physical-time order of each access.

1: P1 reads A
2: P2 reads B
3: P2 writes A
4: P1 reads A
5: P1 writes B
6: P2 reads B
7: P1 reads A

Global logical order of accesses 1-7 or explanation if not possible:

Multiple correct answers: Order of P1 must remain fixed: (1, 4, 5, 7) and order of P2 must remain fixed: (2, 3, 6). In addition, 4 must precede 3, and 6 must precede 5 to ensure that writes occur after old values are read.

Since 4 must precede 3 but follow 1, and 6 must precede 5 but follow 3, we get (1, 4, 3, 6, 5, 7). We also know that 2 must precede 3, so the possible orders are:

2, 1, 4, 3, 6, 5, 7
1, 2, 4, 3, 6, 5, 7
1, 4, 2, 3, 6, 5, 7

Problem 6. Data Locality (16 points)

In this question, we will optimize the implementation of matrix-vector product to improve its data locality: given a matrix M and a vector V, compute O such that $O[i] = \sum_j M[i][j] \times V[j]$.

Assumptions:

- Caches: **Separate caches are used for matrix M and vector V**. Both caches have a single cache line with a block size of 2 words.
- Data size: The data in matrix M and vector V is all 32-bit values and the elements are aligned so that M[0][0] and V[0] go in word 0 of the cache block. Matrix M is stored in row-major order.
- **You can ignore accesses to vector O for all parts of this problem.**

(A) (4 points) Here are two implementations of the computation. Analyze the number of cycles the program spends **accessing input memories for matrix M and vector V** with the following cache characteristics:

- Hit latency: 1 cycle
- Miss latency: 3 cycles (includes hit latency)

Hint: One of the dimensions of the matrix is `N` so your answer should be in terms of `N`. You can assume that N > 2.

i. What is the number of cycles this implementation spends on memory accesses?

```
int M[N][4], V[4], O[N];

for (int j = 0; j < 4; j++) {
  for (int i = 0; i < N; i++) {
    O[i] += M[i][j] * V[j]
  }
}
```

Number of cycles spent accessing matrix M: _____12N_____

Number of cycles spent accessing vector V: _____4N + 4_____

- M's access pattern is column-major. This means that each access will incur a cache miss penalty. For example, when M[0][0] is loaded, M[0][1] is also loaded because the line has a block size of 2. However, the loop access M[1][0] next so this results in a miss. This pattern keeps repeating. There are a total of 4N accesses for a total of 12N cycles.
- V[j] is repeatedly accessed for all values of the inner loop i. There are two cases:
  - $i = 0$: V[j] element is accessed for the first time and incurs a three cycle penalty for V[0] and V[2] and one cycle for V[1] and V[3] since they were fetched together with V[0] and V[2] respectively. Total cost: 8 cycles.
  - $i > 0$: V[j] has already been retrieved so there is a single cycle penalty for all remaining accesses. Since there are a total of 4N accesses, this incurs $4(N-1)$ cycles for all accesses with i> 0.
  - **So total accesses for V[j] = 8 + 4(N-1) = 4N + 4**

ii. What is the number of cycles this implementation spends on memory accesses?

```
int M[N][4], V[4], O[N];

for (int i = 0; i < N; i++) {
  for (int j = 0; j < 4; j++) {
    O[i] += M[i][j] * V[j]
  }
}
```

Number of cycles spent accessing matrix M: _____8N_____

Number of cycles spent accessing vector V: _____8N_____

- M's access pattern is row-major. Load penalty for M[i][0] is 3 cycles and M[i][1] is one cycle. Same pattern for M[i][2] with 3 cycles and M[i][3] with 1 cycle. Total penalty: 8*N* cycles.
- For every iteration of i, V[0] misses, V[1] hits, V[2] misses, V[3] hits. Total cost: 8N.

(B) (6 points) Someone suggests that there is an implementation that only spends a total of 12*N* + 4 cycles accessing both matrix M and vector V input memories.

i. Starting with the program in (**A.ii**), can the number of cycles needed to access M be reduced? Why or why not?

No. M elements are all used exactly once and cannot benefit from caching. Accesses only benefit from having a block size of two.

ii. Starting with the program in (**A.ii**), can the number of cycles needed to access V be reduced? Why or why not?

Yes. Values within V are reused during the computation and can benefit from caching.

iii. Implement a version of this program that spends at most 12*N* + 4 cycles accessing input memories (ignore accesses to O) and explain its cache access behavior.

```
for (int j = 0; j < 4; j+=2) {
  for (int i = 0; i < N; i++) {
    for (int jj = j; jj < j+2; jj++) {
      O[i] += M[i][jj] * V[jj]
  }
}
```

If j only iterates through two values during each iteration of i then V[jj] accesses will follow the pattern described in A.i. In addition, access M[i][0] would be a miss but M[i][1] would be a hit. Similarly, M[i][2] would be a miss but M[i][3] would be a hit. So the cost for accessing matrix M is 8N. So ,the total number of cycles is 12N + 4.

(C) (6 points) Assume that you have been given a new machine whose caches now have a block size of **8 words** and continue to have a single line in each cache. Assume:

- Matrix M and vector V continue to each use their own cache.
- M[0][0] and V[0] go in word 0 of the cache line.
- As before, cache hits are 1 cycle and misses are 3 cycles (includes hit latency).

How many cycles does each implementation spend on accessing each input memory.

i. Implementation from (A.i):

Number of cycles spent accessing matrix M: _____8N_____

Number of cycles spent accessing vector V: _____4N + 2_____

The first time V is accessed, it incurs a 3 cycle penalty but completely fits in the cache line. This means that every subsequent access will always incur a one cycle penalty. The total is 3 + 4N – 1 = 4N + 2

Accessing M[0][0] will miss and incur a 3 cycle penalty but it will bring two rows of M into the cache. Since the next access is M[1][0], it will hit. The third access of M[2][0] misses and the pattern is repeated. So every 2 values of M incur a 4 cycle penalty. So for 4N accesses will take 8N cycles.

ii. Implementation from (A.ii):

Number of cycles spent accessing matrix M: _____5N_____

Number of cycles spent accessing vector V: _____4N + 2_____

Same behavior for V as for A.i.

Since M is now accessed in row major order, for every 8 accesses, it incurs 1 miss and 7 hits. So, on average each access costs (3+7)/8 = 5/4. So the total cost is 5/4 (4N) = 5N.

iii. Your implementation:

Number of cycles spent accessing matrix M: _____6N_____

Number of cycles spent accessing vector V: _____4N + 2_____

Same behavior for V as for A.i.

When M[0][0] is accessed, it misses and brings in the first two rows of M. Next, M[0][1] access is a hit as are M[1][0] and M[1][1]. But M[2][0] will again be a miss. So we have 1 miss and 3 hits for every 4 acceses. So, on average each accesses costs (3+3)/4 = 3/2. So the total cost is 3/2 (4N) = 6N.

END OF QUIZ 3!