1	/22
2	/22
3	/20
4	/18
5	/18

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Spring 2025

Quiz #3

Name	*Athena login name*	*Score*
Solutions		

Recitation section

☐ WF 10, 34-302 (Hilary)	☐ WF 2, 34-302 (Raymond)	☐ WF 12, 35-308 (Keshav)
☐ WF 11, 34-302 (Hilary)	☐ WF 3, 34-302 (Raymond)	☐ WF 1, 35-308 (Keshav)
☐ WF 12, 34-302 (Ezra)	☐ WF 10, 35-308 (Harry)	☐ WF 2, 8-205 (Vedantha)
☐ WF 1, 34-302 (Ezra)	☐ WF 11, 35-308 (Harry)	☐ WF 3, 8-205 (Vedantha)
		☐ 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 (22 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 RISC-V instruction. **Assume all registers and memory start with a default value of zero.**

Program for process A	**Program for process B**

`. = 0x0`
`li t0, 0x100`
`lw a0, 0(t0)`
`lw a1, 0x150(t0)`
`muli t1, t0, 2`
`div t1, t0, a0`
`addi a1, a1, 4`
`sw t1, 0(t0)`
`sw a1, 0x190(x0)`
`unimp`
	`. = 0x0`
	`li t0, 0x50`
	`li t1, 0x300`
	`loop:`
	`  addi t0, t0, 0x100`
	`  lw a1, 0(t0)`
	`  add a0, a0, a1`
	`  blt t0, t1, loop`
	`end:`
	`  sw t0, 0x700(x0)`
	`unimp`

(A) (4 points) These processes run on a custom OS that supports segmentation-based (base and bound) virtual memory. Process A's virtual memory base is at physical address 0x200 and its virtual memory bound is 0x200 (exclusive). Process B's virtual memory base is at physical address 0x400 and its virtual memory bound is 0x300 (exclusive). Which instructions will cause a segmentation fault assuming program execution continues past segmentation faults? You may not need all blanks.

Circle one: (Process A) / Process B

Full instruction: _____ lw a1, 0x150(t0)_____

Circle one: Process A / (Process B)

Full instruction: _____ lw a1, 0x0(t0)_____

Circle one: Process A / (Process B)

Full instruction: _____ sw t0, 0x700(x0)_____

Assume for the following parts that we use paging based virtual memory rather than segmentation. For both Process A and Process B, the page size is 2^{10} bytes and only the virtual page with VPN = 0 is resident in main memory. All other VPNs will raise a Page Fault exception on the first access.

The OS uses timer interrupts to switch between executing process A and B; if a timer interrupt occurs while one process is executing, the other process begins executing after the exception is handled.

In addition, the RISC-V processor has additional hardware to support the `div rd, rs1, rs2` instruction, which divides `rs1` by `rs2` and writes the result into `rd`. The `muli rd, rs1, imm`

instruction, which writes the product of `rs1` and `imm` into `rd`, is unsupported in hardware and must be emulated in software by the OS.

(B) (5 points) The OS schedules process B first but sends a timer interrupt while process B is executing its `addi` instruction in the first iteration of its loop. What are the values of registers `t0, t1, a0, a1, pc` (in virtual address) in processes A and B after the timer interrupt? Recall that **all registers and memory start with a default value of zero.**

Process A:

t0: **0**

t1: **0**

a0: **0**

a1: **0**

pc: **0**

Process B:

t0: **0x50**

t1: **0x300**

a0: **0**

a1: **0**

pc: **0x8**

Process that OS returns control to after timer interrupt: **A**

No instructions in Process A have been executed, so all registers and pc are still 0. The `addi` instruction in process B did not finish executing before the timer interrupt, so `t0` was not updated to 0x150. When the OS switches back to Process B, execution will begin at the `addi` instruction.

(C) (3 points) Process A begins executing. What is the address of the first instruction that raises an exception in Process A, and what is its cause?

Addr of first instruction that raises an exception in A: **0xC**

Cause for Exception (circle one): Page Fault / (Unsupported Instruction) / Division By Zero

(D) (5 points) Process A continues executing and eventually raises a Division By Zero exception because of the `div` instruction. The OS has a unique way of handling Division By Zero exceptions: it changes the divisor from 0 to 1 and returns control back to the same process, so that the RISC-V processor can re-execute the `div` instruction. What are the values of registers `t0, t1, a0, a1, pc` (in virtual address) after the OS returns control to Process A?

t0: 0x100

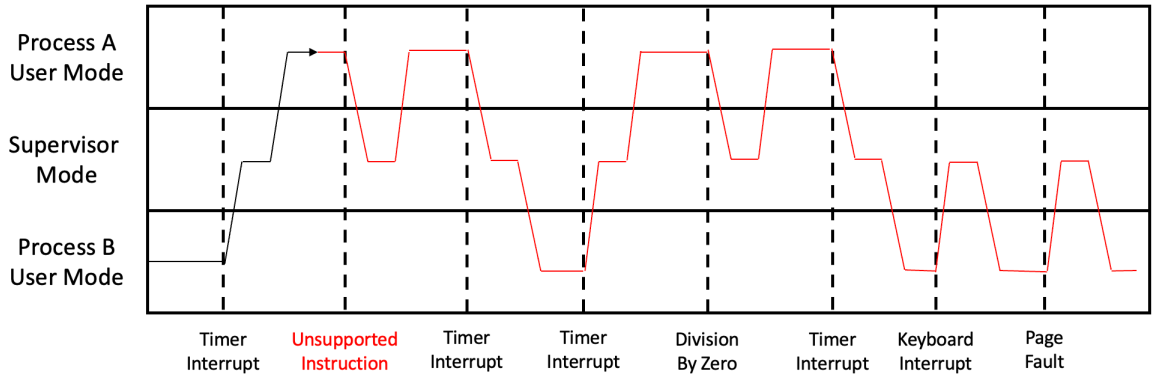
t1: 0x200

a0: 1

a1: 0

pc: 0x10

(E) (5 points) Below is a list of all exceptions raised during the execution of Process A and Process B, in sequential order. The Page Fault exception is raised in Process B by the `sw t0, 0x700(x0)` instruction. Fill in the rest of the diagram to indicate when the processor is running in user mode for Process A, user mode for Process B, or supervisor mode. The first timer interrupt is drawn as an example. Assume that **no additional exceptions occur** beyond those listed in the diagram, and that timer interrupts always switch control to the other process.



Problem 2. Virtual Memory (22 points)

Using his newfound knowledge of virtual memory, Ben Bitdiddle decides to analyze the page table characteristics of his RISC-V processor. His processor contains 2^{23} bytes of virtual memory, 24-bit physical addresses, and page sizes of 4096 (2^{12}) bytes per page.

(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): _____14_____

Number of entries in page table: _____2^{11}_____

(B) (1 point) Assuming the page table is not in physical memory, what is the maximum fraction of virtual memory that can be resident in physical memory at any given time?

Fraction of virtual memory that can be resident in physical memory:

_____ 1 *or* 100%_____

(C) (2 points) If we **double the size of our virtual memory** but keep the same page size and physical memory size, what effect will the change have on the size of a page table entry and on the number of entries in the page table? Use letters "a" through "e" to indicate how the *new* value of the parameter compares to the *old* value of the parameter:

(a) doubled (b) increased by 1 (c) stays the same (d) decreased by 1 (e) halved

Width of each page table entry in bits: _____ c _____

Number of entries in the page table: _____ a _____

For the rest of the problem, keep the amount of virtual memory as 2^{23} bytes.

(D) (8 points) A program has been halted right before executing the following instructions, located at virtual address 0x2FC.

```
. = 0x2FC
lw x2, 0(x6) // x6 = 0x1C5C
sw x3, 4(x7) // x7 = 0x6954
```

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

Page Table

VPN	R	D	PPN
0	0	--	---
1	1	0	0xDC
2	1	0	0x43
Next LRU → 3	1	0	0xE5
4	0	--	---
LRU → 5	1	1	0xA2
6	0	--	---
7	1	1	0x10
	…		

In the table below, specify which virtual address(es) are accessed when executing these instructions. For each virtual address, please indicate the VPN, whether or not the access results in a page fault, the PPN, and the physical address. *If there is not enough information given to determine a given value, write N/A.* Please write all numeric values in hexadecimal.

Virtual Address	VPN	Page Fault (Yes/No)	PPN	Physical Address
0x2FC	0x0	Yes	0xA2	0xA22FC
0x1C5C	0x1	No	0xDC	0xDCC5C
0x300	0x0	No	0xA2	0xA2300
0x6958	0x6	Yes	0xE5	0xE5958

(E) (2 points) Which virtual page(s), if any, need(s) to be written back to disk as a result of executing the two instructions above? Provide the VPN(s) and its corresponding PPN(s). Enter None, if no write back to disk is required. You may not need to use both lines.

VPN _____0x5_____ PPN ____0xA2_____written back to disk

VPN _____ PPN _____written back to disk

(F) (2 points) Ben is curious if changing the page size will affect the number of page faults he encounters. For each of the following page sizes, please write the number of unique VPNs that will be accessed while running the above code.

2^8 bytes per page: _____4____

2^{16} bytes per page: _____1____

For the rest of the problem, keep the original page size of 2^{12} bytes.

(G) (5 points) Consider the same RISC-V processor. We add a 4-element, fully-associative Translation Lookaside Buffer (TLB) with an LRU replacement policy. A program running on the processor is halted right before executing the following instruction located at address `0xF0A0`:

```
. = 0xF0A0
lw x1, 0x20(x7) // x7 = 0x2FE8
```

The contents of the TLB and the first 8 entries of the page table are shown below. The page table and TLB use an LRU replacement policy. Assume that all physical pages are currently in use.

TLB

	VPN	R	D	PPN
LRU →	0x70	1	0	0xA2
	0x2	1	1	0x52
	0xD3	1	1	0x65
	0xF	1	0	0x10

Page Table

VPN	R	D	PPN
LRU → 0	1	0	0xB9
1	0	0	----
2	1	1	0x52
3	0	0	----
4	1	0	0x19
5	1	1	0x89
Next LRU → 6	1	0	0xA7
7	0	0	----
	…		

In the table below, specify which virtual address(es) are accessed when executing this instruction. For each virtual address, please indicate the VPN, whether or not the access results in a TLB Hit, whether or not the access results in a page fault, the PPN, and the physical address. *If there is not enough information given to determine a given value, please write N/A.* Please write all numerical values in hexadecimal.

Virtual Address	VPN	TLB Hit (Yes/No)	Page Fault (Yes/No)	PPN	Physical Address
0xF0A0	0xF	Yes	No	0x10	0x100A0
0x3008	0x3	No	Yes	0xB9	0xB9008

Problem 3. Exception (mis)handler (20 points)

Mr. Chet G. Peaty has been hired as a coding assistant at your startup that makes new and exciting RISC-V exception handlers for clients. He is generally a great programmer, but you need to be very careful when using the code he produces, should there be any subtle bugs.

Your clients can only afford cheap RV32I processors, so they have contracted you to build an exception handler to handle multiply instructions. A colleague of yours has already written the multiply instruction emulator, which takes the instruction word in `a0` and a pointer to the `curProc` struct in `a1`, extracts the fields from the instruction word, reads the source registers from the `curProc` struct, performs the multiplication and stores the results in the destination register in the `curProc` struct (assume the correct result is placed in the destination register of the `curProc` struct and that no other values are modified). Also, **assume the exception handler does not support any other types of exceptions.**

Mr. Chet G. Peaty has produced the following, possibly faulty, `ex_handler` code, and here's what his code does:

- Saves user process registers and exception PC value to the `curProc` struct in memory.
- Reads the instruction that faulted, calls the multiply instruction emulator with it.
- Restores the user process registers and PC value from the `curProc` struct.
- Jumps back to user space.

```
/*         *** USER SPACE ***
 * (code written by a client company) */

calc_volume:
    lw a1, 0x0(a0)
    lw a2, 0x4(a0)
    mul a3, a1, a2
    lw a4, 0x8(a0)
    mul a0, a3, a4
    ret
```

```
/*        *** KERNEL SPACE ***       */
/* Data used by exception handler    */
typedef struct {
    int pc;
    int regs[31];
} ProcState;

ProcState* curProc;

/* Emulates the mul instruction. Stores
 * the result in the destination reg in
 * the ProcState. Does not modify any
 * other regs or pc in the ProcState.
 * Args (a0): Instruction to emulate
 *      (a1): Pointer to a ProcState */

emulate_mul:
    // (assume this works correctly)
    // ...
    ret
```

```
/*         *** KERNEL SPACE ***
 * (code by Mr. Chet G. Peaty)
 * Exception handler entry point. */

ex_handler:
// Reg[mscratch] = a1
    csrw mscratch, a1
    // save regs to curProc
    // using mscratch
    // ...

    // save pc to curProc
    csrr a2, mepc
    lw a1, curProc
    sw a2, 0(a1)

    // read mul inst. from memory
    lw a0, 0(a2)
    call emulate_mul

    // restore pc from curProc
    lw a1, curProc
    lw a2, 0(a1)
    csrw mepc, a2

    // restore regs from curProc
    // ...

    // return to the user process
    mret

// other unrelated instructions
addi a4, a5, 4
xori a2, a3, -1
```

Recall that `csrr` reads from a CSR (control and status register) and `csrw` writes to a CSR. `mret` returns from the exception handler to user mode. **`mret` behaves just like a branch instruction in that it gets resolved in the EXE stage.**

(A) (6 points) Assume this code is running on a standard 5 stage pipelined RISC-V processor **with full bypassing and annulment**. Assume that it uses **lazy exception handling** (just before the commit point), and that branches are always predicted not taken and are resolved in the EXE stage.

Please fill in the pipeline diagram until the cycle in which the `csrw` instruction from the exception handler enters the IF stage. You may leave any columns blank after this. You may ignore the shaded cells and may write a dash (--) to indicate a nop. You do not need to draw any bypassing arrows but make sure to account for any required stalls. The first `lw` instruction of the `calc_volume` function has been indicated for you.

	100	101	102	103	104	105	106	107	108	109	110
IF	lw	lw	mul	lw	mul	ret	csrw				
DEC		lw	lw	mul	lw	mul	--				
EXE			lw	lw	mul	lw	--				
MEM				lw	lw	mul	--				
WB					lw	lw	--				

(B) (2 points) Your client just sent feedback on your exception handler. They say their program never returns from a call to the `calc_volume` function. Briefly explain the reason for this behavior.

Explanation:

The handler does not add 4 to pc so it keeps jumping back to the faulting mul instruction which keeps triggering the exception handler, and goes into this infinite loop.

(C) (5 points) Once again assuming lazy exception handling, please fill in the following pipeline diagram starting from when the exception handler first returns to the user space code, to when the exception handler is fetched the next time (i.e. when the `csrw` instruction enters the IF stage). You may leave any columns blank after this. You may ignore the shaded cells and may write a dash (--) to indicate a nop. You do not need to draw any bypassing arrows but make sure to account for any required stalls. The `mret` instruction of the handler has been indicated for you. *Hint: Recall that with the code as currently written, the `calc_volume` function never returns.*

	200	201	202	203	204	205	206	207	208	209	210
IF	mret	addi	xori	mul	lw	mul	ret	csrw			
DEC		mret	addi	--	mul	lw	mul	--			
EXE			mret	--	--	mul	lw	--			
MEM				mret	--	--	mul	--			
WB					mret	--	--	--			

(D) (2 points) Since Mr. Chet G. Peaty's work is mostly correct, you decide to apply a small fix yourself to the exception handler. Add **exactly one instruction** to the handler by filling in one of the blanks below in order to fix the exception handler.

```
ex_handler:
    csrw mscratch, a1

    _____________________________________
    // save regs to curProc
    // using mscratch
    // ...

    _____________________________________
    // save pc to curProc
    csrr a2, mepc

    Either: addi a2, a2, 4_____________
    lw a1, curProc
    sw a2, 0(a1)

    // read mul inst. from memory
    lw a0, 0(a2)

    _____________________________________
    call emulate_mul

    // restore pc from curProc
    lw a1, curProc
    lw a2, 0(a1)

    Or: addi a2, a2, 4_____________
    csrw mepc, a2

    // restore regs from curProc
    // ...

    // return to the user process

    _____________________________________
    mret
```

(E) (5 points) In this question, assume that:

- We use **a single-cycle processor with a single memory that holds both data and instructions**.
- Virtual memory is not used, so all code (including user code and exception handler) runs on the **same address space**.

You ask Mr. Chet G. Peaty to simplify the exception handler and change it such that it stops running into infinite loops. But his new handler, shown below, does not work correctly either.

```
//   *** USER SPACE ***

// same code as above
calc_volume:
    lw a1, 0x0(a0)
    lw a2, 0x4(a0)
    mul a3, a1, a2
    lw a4, 0x8(a0)
    mul a0, a3, a4
    ret

// unrelated instructions
addi t6, t9, 4
xor s1, s2, s3
```

```
//  *** KERNEL SPACE ***

ex_handler:
    csrr a0, mepc
    lw a1, nop_slide
    sw a1, 0(a0)
    mret

nop_slide:
    sll x0, x1, x2
    add x0, x3, x4
    xor x0, x5, x6
```

The client now uses this exception handler on a single cycle RISC-V processor and calls `calc_volume` a hundred times in a loop. How many times does the exception handler get called? Circle one and briefly explain.

Still runs into an infinite loop

Once

Twice

A hundred times

Two hundred times

Explanation:

Each call of the exception handler replaces the faulting instruction with a sll x0, x1, x2 instruction (effectively a nop). Once both the mul instructions have been replaced, the exception handler is never called again.

Problem 4. E-Commerce Synchronization (18 points)

You and your friends want to build the world's biggest e-commerce company, so you've started **6191 Mercato.** In your first prototype there are two types of threads. **There can be multiple instances of each type of thread.**

The first thread type is the producer, which is responsible for creating the products you plan to sell. Your warehouse can store at most 500 products. Any product above this capacity is wasted, so you want to avoid this. The second thread type is the sales service, which ensures that a customer can purchase a completed product. The `buy_product()` function removes a product from the warehouse and ships it to the customer.

<table>
<tr><td colspan="2">Shared Memory:

<pre>int stored = 0</pre></td></tr>
<tr><td>Producer Code:

<pre>if stored < 500 {

   stored = stored + 1

   create_product()

}</pre></td><td>Sales Code:

<pre>if stored > 0 {

   buy_product()

   stored = stored - 1

}</pre></td></tr>
</table>

(A) (4 points) Using the code given above, answer if the following conditions are possible:

1. You exceed your maximum capacity of 500 products.
   Two machines running production code can run "stored = stored + 1" concurrently, resulting in stored not being properly incremented to reflect both of these products being created and stored in the warehouse.

   Possible / **Not Possible**

2. A customer tries to buy a nonexistent product.

   Two machines running sales code can run "stored = stored - 1" concurrently, resulting in stored not being properly incremented to reflect both of these products being purchased.

   Possible / **Not Possible**

(B) (14 points) Customers have started complaining about receiving faulty products, so in your second iteration of 6191 Mercato, you introduce a single **Quality Assurance (QA)** thread. After a product has been created and stored in the warehouse, the QA thread must inspect the product before a customer is able to purchase it. However, performing a QA inspection on an individual product is costly, so you want the QA thread to operate on batches of ten products at once.

Here are the conditions your code should meet:

- You should not have more than 500 products stored in the warehouse at a time.
- There can be multiple producer and sales threads, but there is only one QA thread.
- You should perform inspections on batches of 10 products at a time.
- You should only allow a customer to purchase a product when it is actually ready for purchase (has been created and passed an inspection).
- You should keep track of the total number of products created.
- You can use at most 4 semaphores to complete the code, and you cannot initialize your semaphores to negative values.
- There should be no deadlocks in your code.
- You should not introduce any extra precedence constraints.
- You may only add semaphore declaration and initialization in shared memory, and wait(sem) and signal(sem) calls in the code.

Complete the code below. If you want to call wait or signal on semaphore *sem* multiple times (say *n* times) in a row, you can simply write "*wait(sem)* x *n*" or "*signal(sem)* x *n*".

Shared Memory:

```
int count = 0     // keeps track of the total number of products created
lock = 1, capacity = 500, inspect = 0, finished = 0
```

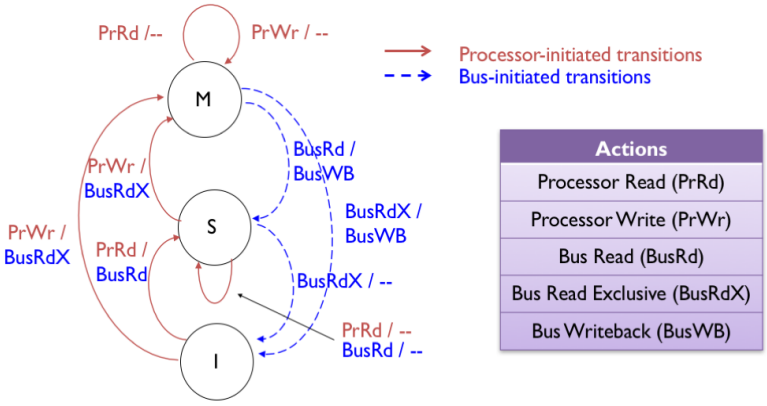
Producer:	**QA:**	**Sales:**
`wait(capacity)`	`wait(inspect) x 10`	`wait(finished)`
`wait(lock)` `count = count + 1`	`inspect_10()`	`buy_product()`
`signal(lock)`	`signal(finished) x 10`	`signal(capacity)`
`create_product()`		
`signal(inspect)`		
`goto Producer`	`goto QA`	`goto Sales`

Problem 5. Cache Coherence (18 points)

Ben Bitdiddle is designing a snoopy-based, write-invalidate MSI protocol for write-back caches. Suppose processors P1 and P2 have private snoopy caches. Both caches are initially empty. Consider the following sequence of accesses:

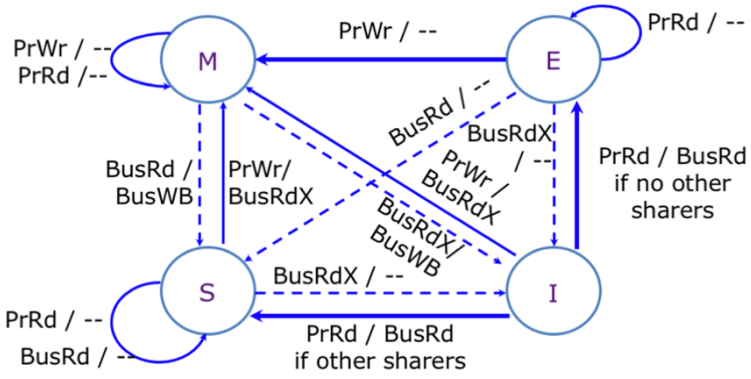
```
I0 P1: read A
I1 P2: read B
I2 P1: write A
I3 P1: write B
I4 P2: write B
I5 P1: read B
I6 P2: read A
```

(A) (9 points) Assume blocks A and B do not conflict in the cache. Using the **MSI protocol**, 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)). We provide you with the MSI cache coherence state transition diagram for reference.



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: S	B: I	A: I	B: I
After P2 reads B	P2: BusRd(B)	A: S	B: I	A: I	B: S
After P1 writes A	P1: BusRdX(A)	A: M	B: I	A: I	B: S
After P1 writes B	P1: BusRdX(B)	A: M	B: M	A: I	B: I
After P2 writes B	P2: BusRdX(B) P1: BusWB(B)	A: M	B: I	A: I	B: M
After P1 reads B	P1: BusRd(B) P2: BusWB(B)	A: M	B: S	A: I	B: S
After P2 reads A	P2: BusRd(A) P1: BusWB(A)	A: S	B: S	A: S	B: S

(B) (9 points) Repeat part A using a **MESI protocol**. We provide you with the MESI cache coherence state transition diagram for reference.



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 P1 writes A		A: M	B: I	A: I	B: E
After P1 writes B	P1: BusRdX(B)	A: M	B: M	A: I	B: I
After P2 writes B	P2: BusRdX(B) P1: BusWB(B)	A: M	B: I	A: I	B: M
After P1 reads B	P1: BusRd(B) P2: BusWB(B)	A: M	B: S	A: I	B: S
After P2 reads A	P2: BusRd(A) P1: BusWB(A)	A: S	B: S	A: S	B: S

After first access: A will be in E in P1.
After second access: B will be in E in P2.
After third access: No BusRdX required, E -> M in P1 for A.

End of Quiz 3