1	/16
2	/18
3	/18
4	/16
5	/19
6	/13

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Fall 2024

Quiz #3

Name Solutions	*Athena login name*	*Score*

Recitation section

☐ WF 10, 34-301 (Varun)	☐ WF 2, 34-303 (Pleng)	☐ WF 12, 34-303 (Ezra)
☐ WF 11, 34-301 (Varun)	☐ WF 3, 34-303 (Pleng)	☐ WF 1, 34-303 (Ezra)
☐ WF 12, 34-302 (Keshav)	☐ WF 10, 34-302 (Hilary)	☐ WF 2, 34-302 (Jessica)
☐ WF 1, 34-302 (Keshav)	☐ WF 11, 34-302 (Hilary)	☐ WF 3, 34-302 (Jessica)
		☐ 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 (16 points)

Consider the following two processes, A and B, running on a standard RISC-V processor. **Code listings use virtual addresses.**

Program for process A

```
. = 0x10
  .ascii “Process A: 1\n”
. = 0x150
  .ascii “Process A: 2\n”

. = 0x200
loopA:
  li a0, 0x10
  li a7, 0x1B
  ecall
  li a0, 0x150
  li a7, 0x1B
  ecall
  j loopA
```

Program for process B

```
. = 0x0
  .ascii “Process B: 1\n”
. = 0x200
  .ascii “Process B: 2\n”

. = 0x600
loopB:
  li a0, 0x0
  li a7, 0x1B
  ecall
  li a0, 0x200
  li a7, 0x1B
  ecall
  li t0, 0x790
  sw a0, 0(t0)
  j loopB
```

These processes run on a custom OS that supports segmentation-based (base and bound) virtual memory, timer interrupts for scheduling processes, and a `print_string` system call for printing strings.

Processes invoke syscalls with the `ecall` instruction. The `print_string` system call takes the address of a string to print as the argument in register a0, and syscall number 0x1B in register a7. **It returns the length of the string that was printed. Note that the length of all of the strings is 13.**

Assume virtual addresses are translated with the following base and bound registers:
Process A: **base register = 0x100, bound register = 0x180**
Process B: **base register = 0x700, bound register = 0x1000**

(A) (2 points) What is the physical address of the start of the string located at 0x10 in Process A and the start of the string located at 0x200 in Process B?

Physical Address of string at 0x10 in Process A: _____0x110_______

Physical Address of string at 0x200 in Process B: _____0x900_______

(B) (3 points) While running the two processes, you notice that one of the processes crashes due to a segmentation fault.

Which process has a segmentation fault? Circle one: [✓ Process A] / Process B

Which instruction causes the segmentation fault? ______**li a0, 0x10**_________

(C) (2 points) Assume that all segmentation faults have been fixed. You decide to test only Process A first to see if it behaves as expected, and get the following incorrect output, where "Process A: 1" is printed repeatedly.

```
Process A: 1
Process A: 1
Process A: 1
…
```

You isolate the issue to the handling of the exception specified by the *mcause* register. **What bug in the code that handles this exception is causing the incorrect behavior?**

The exception handler is not setting the current process' pc correctly, so when the common handler returns after ecall, the pc is set to 0x200. This reloads a0, and then makes the ecall again.

ANOTHER ACCEPTED SOLUTION: The exception handler is not incrementing the current process' pc by 4. This makes it so that when the common handler returns after ecall, the pc is set to the pc of the first ecall instruction, so the ecall gets executed repeatedly.

The above solution is actually incorrect because when print_string returns, it puts 13 into a0. So, if the ecall gets executed again, it actually would not exhibit the behavior shown in the problem. We still accepted that solution as valid since that was the intent of the problem.

(D) (3 points) Assume that all issues have been fixed and the processes behave as expected. Say that Process A was scheduled first, and runs until the first `ecall` instruction completes and returns from the common handler. What are the values in the a0, a7, and pc (in virtual address) registers? Write CAN'T TELL if you can't tell a value from the information given.

a0: _________**13**_____________

a7: _________**0x1B**___________

pc: ________**0x20C**___________

Process that OS returns control to after the ecall: ______**Process A**__________

(E) (3 points) Still assume that all issues have been fixed. You run both processes and get the following output:

```
Process A: 1
Process B: 1
Process B: 2
Process A: 2
```

You pause the program immediately after the last line finishes printing and returns from the common handler. What are the values in the a0, t0, and pc (in virtual address) registers? Write CAN'T TELL if you can't tell a value from the information given.

a0: _________**13**_____________

t0: ____**CAN'T TELL**________

pc: _______**0x218**____________

(F) (3 points) Still assume that all issues have been fixed. Process A and B are now run until 4 lines are printed. For the following outputs, specify if that output could have been produced by our programs or not.

Outputs:

```
Process B: 1
Process A: 1
Process B: 2
Process B: 1
```

Circle One:

[✓ Possible] / Not Possible

```
Process A: 1
Process A: 2
Process A: 1
Process B: 1
```

Circle One:

[✓ Possible] / Not Possible

```
Process A: 1
Process B: 2
Process B: 1
Process A: 2
```

Circle One:

Possible / [✓ Not Possible]

The first output is possible if B is scheduled first. The second output is possible if A is scheduled first and goes through one and half iterations of the loop before a timer interrupt. The second output is not possible because 'Process B: 2' is printed before 'Process B: 1'.

Problem 2. Virtual Memory (18 points)

Consider a RISC-V processor that has 16-bit virtual addresses, 2^{20} bytes of physical memory, and uses a page size of 2^{12} 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 physical page number, a dirty bit, and a resident bit. *Your final answer can be a product or exponent.*

Number of entries in the page table: ____2^4______

Size of page table entry (in bits): ____10______

Size of the page table (in bits): ___$10*2^4$____

(B) (1 point) Instead of using a page size of 2^{12} bytes, say we decide to use a page size of 2^8 bytes. What is the ratio of the **page table sizes** with the new page size of 2^8 bytes, compared to the old page size of 2^{12} bytes? *Your final answer can use fractions, products, and exponents.*

Ratio of page table sizes: ____$(2^8 * 14)/(10*2^4)$_____

For the rest of the problem, keep the **page size as 2^{12} bytes**, and assume a **hierarchical page table structure of 2 levels**, with address mapping as shown below:

VPN		Page Offset
1st level index	2nd level index	Page Offset

It is given to you that the **number of bits in the 1st and 2nd level indices are equal**.

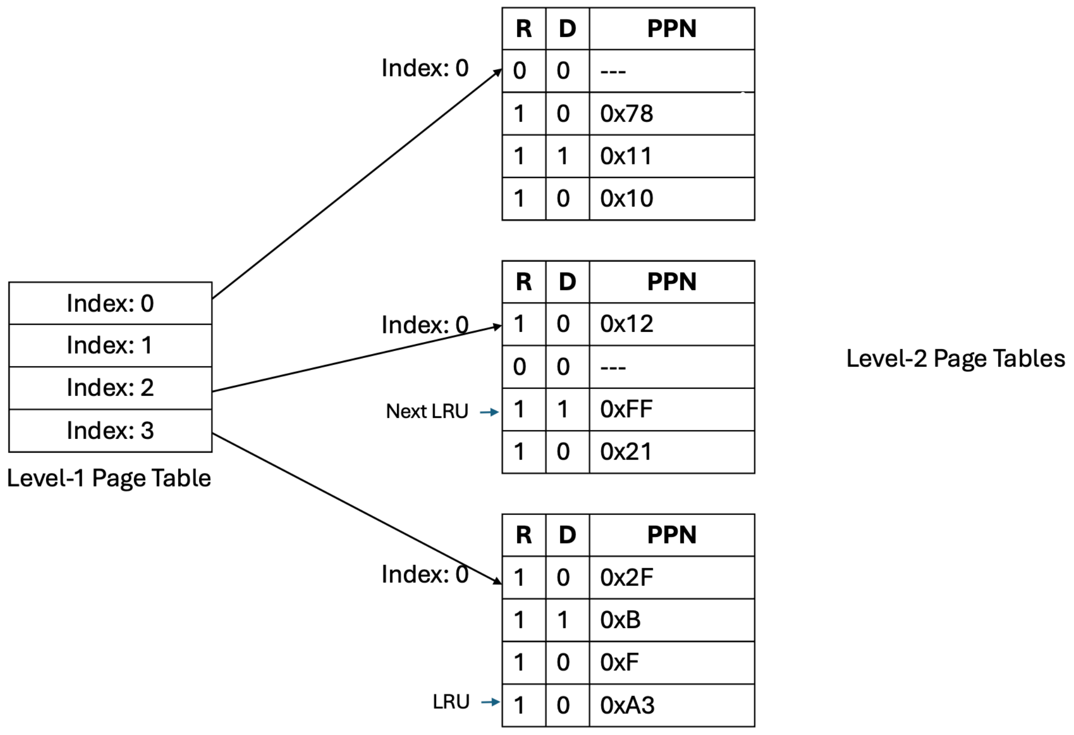
(C) (2 points) Calculate the following parameters relating to the size of each second-level page table. Each second-level page table entry contains a physical page number, a dirty bit, and a resident bit. *Your final answer can be a product or exponent.*

Number of entries in each 2nd level page table: ______2^2_______

Size of 2nd level page table entry (in bits): ______10_______

(D) (8 points) You now run a test program on this processor. Execution of this test program is halted just before executing the following two instructions. The state of the hierarchical page table is shown below; the least recently used page ("LRU") and next least recently used page ("next LRU") are indicated where necessary. `x1` has been set to `0x9000`. Assume all physical pages are in use. Execution resumes and the following two instructions are executed:

```
. = 0xBFFC
lw x12, 0xF(x0)
sw x12, 0x0(x1) // x1 = 0x9000
```



For each virtual address accessed, please indicate, in the chart below, the virtual address, the VPN, whether 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. **Assume that we use an LRU replacement policy on the 2nd level of Page Tables.**

Virtual Address	VPN	Page Fault (Yes/No)	PPN	Physical Address
0xBFFC	0xB	No	0x21	0x21FFC
0xF	0x0	Yes	0xA3	0xA300F
0xC000	0xC	No	0x2F	0x2F000
0x9000	0x9	Yes	0xFF	0xFF000

(E) (2 points) Also, specify which PPN(s) were evicted, and which were written back to memory during execution of the two instructions from part (D). If there are no pages to list, then enter NONE.

Evicted PPN(s) (hex): _____**0xA3, 0xFF**_______

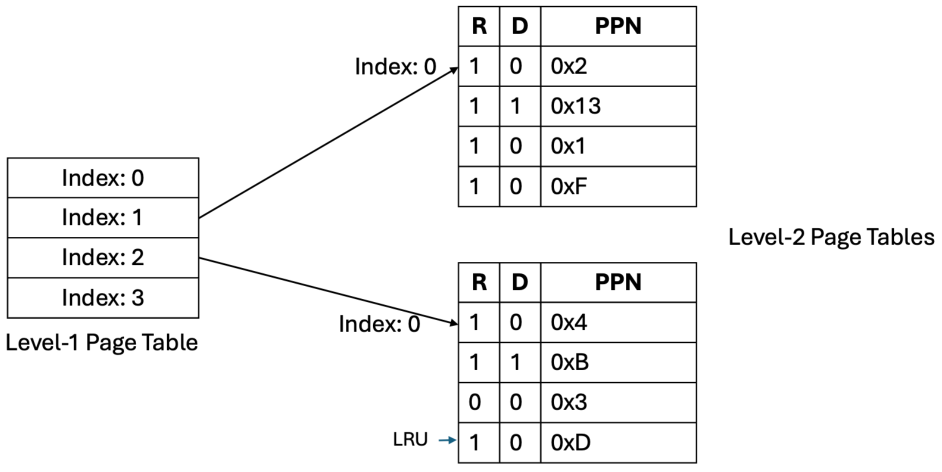
Written back PPN(s) (hex): _________0xFF_______

(F) (3 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 0xB2A0. `x4` has been set to `0x9204`:

```
. = 0xB2A0
sw x3, 4(x4) // x4 = 0x9204
```

The contents of the TLB and the hierarchical page table are shown below. Assume that all physical pages are currently in use. **Assume that we use an LRU replacement policy on the 2nd level of Page Tables.**

	TLB				
	VPN	**V**	**R**	**D**	**PPN**
LRU →	0x1	1	1	1	0x0
	0x9	1	1	0	0xB
	0x8	1	1	0	0x4
	0x0	0	0	0	0x99



Fill out the updated state of the TLB after these operations. You may mark a row as "NO CHANGE" if it remains unchanged. Please write all numerical values in hexadecimal.

TLB				
VPN	**V**	**R**	**D**	**PPN**
0xB	1	1	0	0xD
0x9	1	1	1	0xB
NO CHANGE				
NO CHANGE				

Problem 3. Exceptions (18 points)

We are trying to run the following piece of code. Unfortunately, our processor does not implement the divide instruction. Instead, we choose to emulate the instruction within the operating system.

```
. = 0x000
      addi a0, x0, 0x1
      addi a1, x0, 0x1
      bnez a1, second
first:
      div a2, a1, a0
      addi a3, x0, 0x1
second:
      div a2, a1, a0
      add x0, x0, x0
      add x0, x0, x0
      add x0, x0, x0
      …


// Kernel space
common_handler:
      csrw mscratch, x1
      lw x1, curProc
      sw x2, 0x8(x1)
      sw x3, 0xc(x1)
      sw x4, 0x10(x1)
      sw x5, 0x14(x1)
      sw x6, 0x18(x1)
     …
```

Note that the division instruction exception is detected in the decode stage.

(A) (4 points) Fill in all the white boxes in the 5-stage pipeline diagram for the execution of this code. Assume branches are resolved in the execute stage, and there is full bypassing. Assume that exceptions are handled lazily (at the commit point). **You do not need to include bypassing arrows.**

Cycle	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
IF	addi	addi	bnez	div	addi	div	add	add	add	csrw
DEC		addi	addi	bnez	div	NOP	div	add	add	NOP
EXE			addi	addi	bnez	NOP	NOP	div	add	NOP
MEM				addi	addi	bnez	NOP	NOP	div	NOP
WB					addi	addi	bnez	NOP	NOP	NOP

(B) (4 points) Fill this diagram out again, but now assume that we handle exceptions immediately, as soon as they are detected. **You do not need to include bypassing arrows.**

Cycle	0	1	2	3	4	5	6	7	8	9
IF	addi	addi	bnez	div	addi	div	add	csrw	lw	sw
DEC		addi	addi	bnez	div	NOP	div	NOP	csrw	lw
EXE			addi	addi	bnez	NOP	NOP	NOP	NOP	csrw
MEM				addi	addi	bnez	NOP	NOP	NOP	NOP
WB					addi	addi	bnez	NOP	NOP	NOP

(C) (2 points) Suppose Alice is writing a program. Alice forgets about the `ecall` instruction and instead uses the jump instruction to call the common handler directly, instead of using the `ecall`.

```
example_program:
   li a0, 32
   li a1, 10
   li a7, SYS_SEMINIT
   j common_handler

common_handler:
   // Save all the registers into the curProc data structure
   csrw mscratch, x1
      ...
   // Setup the necessary registers to call the dispatcher
   // Call the dispatcher
   // Load all the registers from the curProc data structure
   // Return to the calling process
   mret
```

Will this work as intended (circle one)? **YES** [✓ **NO**]

Why or why not?

No, the ecall instruction switches us to kernel mode which gives us access to priviliged registers as well as the kernell code. It also saves the pc into the mepc so that mret can properly return to the correct location in the user program.

(D) (8 points) We are running the following piece of code. Determine which lines of code can be executing at the time each possible exception or interrupt happens. If there are fewer than 3 locations where the exception/interrupt can occur, please list out their PCs (e.g., `0x4`). If there are 3 or more locations, then write "MANY PCs." If no PCs can be executing, write "NONE".

Assume:

- The page table initially has allocated one page for VPN 0x0.
- Each page is 2^{12} bytes
- Page faults are handled by the OS
- None of these instructions cause a segmentation fault.
- If the process encounters a division by zero, it is immediately killed by the OS.
- **Division instruction is implemented in hardware.**

PC	**Instruction**
`0x00`	`lui a1, 1   // a1 = 0x1000`
`0x04`	`lui a2, 5   // a2 = 0x5000`
`0x08`	`lw a4, 0x40(a1)`
`0x0c`	`lw a5, 0x0(a1)`
`0x10`	`lw a6, 0x0(a2)`
`0x14`	`li a0, 0`
`0x18`	`li a7, SYS_PUTCHAR`
`0x1c`	`ecall`
`0x20`	`add a0, x0, x0`
`0x24`	`div a2, a2, a0`

Interrupt/Exception Type	**Current Executing Instructions**
Page fault	**0x08, 0x10**
Timer interrupt	**MANY PCs**
System call	**0x1c**
Division by zero	**0x24**
Illegal opcode	**NONE**

Problem 4. Synchronized Space Shenanigans (16 points)

The Earth Space Research Organization (ESRO) has decided to launch a rocket to provide supplies to the astronauts living in the World Space Station (WSS). ESRO has equipped the rocket with five boosters.

All boosters run the same code. Each booster must complete a set of pre-launch checks. Each booster can independently run the pre-launch checks. **Finally, the boosters must ignite only after *all five* boosters have completed the pre-launch checks. Each booster must call ignite for itself.**

Engineers at ESRO have written the following code to help ignite the boosters:

Shared Memory:

```
int num_ready_boosters = 0;
```

booster_code:

```
    prelaunch_check()

    num_ready_boosters = num_ready_boosters + 1

    if( num_ready_boosters == 5) {

              ignite()
    }
```

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

1. None of the boosters ever ignite.

   [✓ Possible] / Not Possible

2. All boosters ignite **after** all five boosters have completed the pre-launch checks **and** None of the boosters ignite **before** all five boosters have completed the pre-launch checks.

   [✓ Possible] / Not Possible

3. A booster ignites before all five boosters have completed the pre-launch checks.

   Possible / [✓ Not Possible]

4. What synchronization issue exists in the above code?

   [✓ Race Condition] / Deadlock / Both

(B) (12 points) You notice that the above booster code can fail. To avoid a failed launch you decide to help out the ESRO engineers by correcting the booster code to meet the following requirements:

- Each booster can run pre-launch checks concurrently with other boosters.
- None of the boosters should ignite until all five boosters have completed pre-launch checks.
- After all boosters have completed pre-launch checks, all boosters must ignite.
- You can use at most 2 semaphores to complete the code, and you cannot initialize your semaphores to negative values.
- There should be no race conditions or deadlocks in your code.
- You should not introduce any extra constraints.
- You may only add semaphore declaration and initialization in shared memory, and wait(sem) and signal(sem) instructions in the booster code.

Complete the code below. **Notice that the ignite() function is now outside the if condition. Hint: Think carefully about how the if statement affects the semaphore values.**

Shared Memory:

```
int num_ready_boosters = 0;
semaphore lock = 1
semaphore barrier = 0
```

```
booster_code:

    prelaunch_check()

    wait(lock)

    num_ready_boosters = num_ready_boosters + 1

    signal(lock)

    if( num_ready_boosters == 5) {

         signal(barrier)

    }
    wait(barrier)

    signal(barrier)

    ignite()
```

Since num_ready_boosters is a shared variable that is being read from and written to by multiple boosters, there is a race condition. Hence, we put a lock around "num_ready_boosters = num_ready_boosters + 1" to prevent race conditions. .

Since none of the boosters can ignite before all boosters have completed the pre-launch checks, we have a precedence constraint. Anything inside the if block precedes the ignite call. Hence, using our recipe we add a wait(barrier) before the ignite(call) and a signal(barrier) inside the if condition.

Notice here that this signal(barrier) signals just one waiting booster to proceed and only one of the boosters can move forward with ignite(). The other boosters are still waiting! Hence, the booster that is able to proceed and call ignite() has to signal to another booster that it can ignite! Therefore we need the signal(barrier) just before ignite().

This leads to a cascade of signal(barrier) calls between the boosters. This common synchronization pattern goes by the name of a ***non-reusable barrier*** or ***non-reusable turnstile***.

If you are interested in learning more synchronization patterns, refer to "The Little Book of Semaphores": https://greenteapress.com/wp/semaphores/

Alternate solution:

Shared Memory:

```
int num_ready_boosters = 0;
semaphore lock = 1
semaphore barrier = 0
```

```
booster_code:

    prelaunch_check()


    wait(lock)

    num_ready_boosters = num_ready_boosters + 1

    signal(lock)


    if( num_ready_boosters == 5) {
         signal(barrier)
         signal(barrier)
         signal(barrier)
         signal(barrier)
         signal(barrier)
    }

    wait(barrier)

    ignite()
```

In the first solution, we added a signal(barrier) call just before ignite() to make sure all 5 boosters eventually ignite(). An alternative way of ensuring all 5 boosters ignite is to call signal(barrier) five times once the if condition is satisfied!

Problem 5. Cache Coherence (19 points)

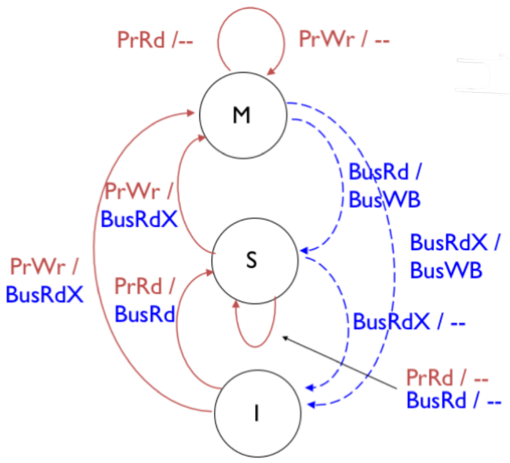
Ben Bitdiddle has a four-core processor system, where each core has its own cache. Ben has the option to use either a snoopy-based, write invalidate MSI or a snoopy-based, write invalidate MESI protocol, and is trying to decide which is better for optimizing the following code where `S1` and `S2` are semaphores initialized to 0. Assume that `X` and `Y` map to different lines of the cache.

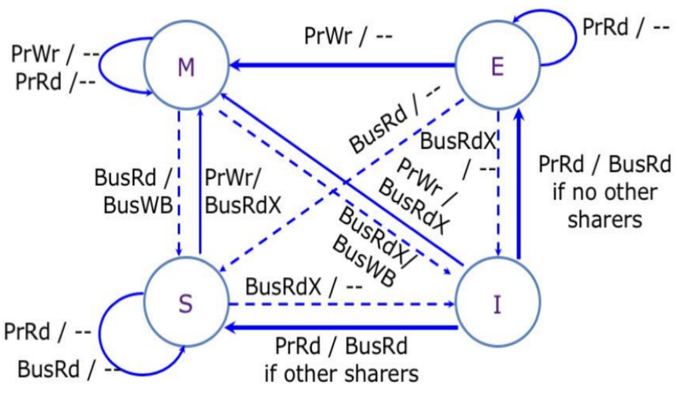
Core A runs:
```
proc1:
   lw a1, X
   add a1, a1, a0
   sw a1, Y
   signal(S1)
   signal(S1)
   signal(S1)
   wait(S2)
   wait(S2)
   wait(S2)
   lw a1, Y
   sw a1, X
```

Core B runs:
```
proc2:
   wait(S1)
   lw a1, Y
   add a1, a1, a0
   sw a1, Y
   signal(S2)
```

Core C runs:
```
proc2:
   wait(S1)
   lw a1, Y
   add a1, a1, a0
   sw a1, Y
   signal(S2)
```

Core D runs:
```
proc2:
   wait(S1)
   lw a1, Y
   add a1, a1, a0
   sw a1, Y
   signal(S2)
```





Ben decides to first observe how MSI and MESI perform when run on a subset of the processes. He modifies proc1 to only perform 1 `signal(S1)` and 1 `wait(S2)` and then runs just Core A and Core B.

Core A runs:
```
proc1:
   lw a1, X
   add a1, a1, a0
   sw a1, Y
   signal(S1)
   wait(S2)
   lw a1, Y
   sw a1, X
```

Core B runs:
```
proc2:
   wait(S1)
   lw a1, Y
   add a1, a1, a0
   sw a1, Y
   signal(S2)
```

The semaphores guarantee that data accesses proceed in the following order. **The table is given for your convenience and will not be graded.**

Access	*Shared Bus Transactions*	*Cache A*		*Cache B*	
Initial state		X: I	Y: I	X: I	Y: I
A: lw a1, X	A: BusRd(X)	X: S/E	Y:	X:	Y:
A: sw a1, Y	A: BusRdX(Y)	X: S/E	Y: M	X:	Y:
B: lw a1, Y	B: BusRd(Y)-> A: BusWB(Y)	X: S/E	Y: S	X:	Y: S
B: sw a1, Y	B: BusRdX(Y)	X: S/E	Y: I	X:	Y: M
A: lw a1, Y	A: BusRd(Y)-> B: BusWB(Y)	X: S/E	Y: S	X:	Y: S
A: sw a1, X	A: BusRdX(X)/None	X: M	Y: S	X:	Y: S

(A) (4 points) For each protocol, how many of each of the following bus requests occur for the series of accesses listed above?

Protocol	*# of BusRd*	*# of BusRdX*	*# of BusWB*
MSI	3	3	2
MESI	3	2	2

(B) (1 point) Assuming that instructions always interleave in the same order, when switching from a MSI protocol to a MESI protocol, which of the following # of bus requests *could* decrease?

# of BusRd:	Could decrease	[✓ Stays the same]
# of BusRdX:	[✓ Could decrease]	Stays the same
# of BusWB:	Could decrease	[✓ Stays the same]

After observing how his code performs on MSI and MESI with just 2 cores, he thinks he has enough information to decide which is better for his 4 core system.

(C) (2 points) Ben's MESI protocol takes 10ns longer than his MSI protocol per cache access. Suppose all bus transactions(BusRd, BusRdX, BusWB) take an additional 80ns if they are called. For Ben's system that has a total of 10 data accesses(1 `lw X`, 1 `sw X`, 4 `lw Y`, 4 `sw Y`), how many bus transactions need to be saved for the MESI protocol to take less total time than the MSI protocol?

B = num bus transactions with MESI
B + N = num bus transactions for MSI
$10 * 10 + 80 B < 80 (N + B)$
$100 < 80N$
$N > 1.25 = 2$

Minimum number of saved bus transactions: ________2_________

(D) (1 point) What is the maximum number of bus transactions that would be saved if Ben uses the MESI protocol over the MSI protocol? (Hint: pay close attention to what the semaphores guarantee about data access order)

Maximim number of saved bus transactions using Ben's code: ________1_________

(E) (1 point) Should Ben use his MSI or MESI protocol (circle one)?

[✓ **MSI**] **MESI**

(F) (10 points) Suppose Ben's program interleaves data accesses in the following order:

Core A	Core B	Core C	Core D
(1) lw a1, X (2) sw a1, Y (9) lw a1, Y (10) sw a1, X	(4) lw a1, Y (5) sw a1, Y	(3) lw a1, Y (7) sw a1, Y	(6) lw a1, Y (8) sw a1, Y

Fill in the following table using the protocol you selected in part E (**this table WILL be graded**). Include all shared bus transactions. Include the address associated with each bus transaction (e.g., **BusRdX(Y)**). If no bus transactions occur, write N/A in the corresponding box. **Cache states left blank will be assumed to be Invalid.**

Access	*Shared Bus Transactions*	*Cache A*		*Cache B*		*Cache C*		*Cache D*	
Initial state		X: I	Y: I	X: I	Y: I	X: I	Y: I	X: I	Y: I
A: lw a1, X	A: BusRd(X)	X:S	Y:	X:	Y:	X:	Y:	X:	Y:
A: sw a1, Y	A: BusRdX(Y)	X:S	Y:M	X:	Y:	X:	Y:	X:	Y:
C: lw a1, Y	C: BusRd(Y)-> A: BusWB(Y)	X:S	Y: S	X:	Y:	X:	Y: S	X:	Y:
B: lw a1, Y	B: BusRd(Y)	X:S	Y: S	X:	Y: S	X:	Y: S	X:	Y:
B: sw a1, Y	B: BusRdX(Y)	X:S	Y: I	X:	Y:M	X:	Y: I	X:	Y:
D: lw a1, Y	D: BusRd(Y)-> B: BusWB(Y)	X:S	Y: I	X:	Y: S	X:	Y: I	X:	Y: S
C: sw a1, Y	C: BusRdX(Y)	X:S	Y: I	X:	Y: I	X:	Y:M	X:	Y: I
D: sw a1, Y	D: BusRdX(Y)-> C: BusWB(Y)	X:S	Y: I	X:	Y: I	X:	Y: I	X:	Y:M
A: lw a1, Y	A: BusRd(Y)-> D: BusWB(Y)	X:S	Y: S	X:	Y: I	X:	Y: I	X:	Y: S
A: sw a1, X	A: BusRdX(X)	X:M	Y: S	X:	Y: I	X:	Y: I	X:	Y: S

Problem 6. Cache Me If You Can (13 points)

Oh no, Didit is broken! The 6.004 TAs are scrambling to put together a grading script that calculates student grades so that they can get class grades submitted before the deadline.

The student grades are stored in an array `int G[S][N]` such that S is the number of students in the class, N is the total number of assignments in the class and `G[s][n]` corresponds to the grade obtained by student number s on assignment number n.

Additionally, there's an array called `int W[N]` which stores the per-assignment weights. The TAs want to calculate the total score of the class in a register variable called `sum`. All the arrays are stored in **row-major configuration.**

(A) (2 points) A couple of TAs suggest the following two versions of the script:

Version A:
```
void grade() {
    int sum = 0;
    for (int n = 0; n < N; n++)
        for (int s = 0; s < S; s++)
            sum += W[n] * G[s][n];
}
```

Version B:
```
void grade() {
    int sum = 0;
    for (int s = 0; s < S; s++)
        for (int n = 0; n < N; n++)
            sum += W[n] * G[s][n];
}
```

Which version has better data locality? Why?

Version with better Data Locality (circle): A [✓ B]

Explanation:

When arrays are stored in row-major configuration that means that they are stored row per row so cells in adjacent columns are adjacent in memory. When we bring a new line into the cache, it corresponds to one row and multiple columns. This means that we want our innermost loop to iterate over columns so that each access of the inner loop (after the first which brings the line into the cache) is a hit.

The program runs on a CPU with separate instruction and data caches. For this problem, assume that instructions, W, and G arrays **reside in three different caches respectively**, and we want to analyze the data misses on the W array. The W array cache is a **fully associative** cache with **4 words per block,** with **LRU eviction policy**. Assume that we run **Version B** of the code and that there are 64 students in the class (**S** = **64**) and 16 assignments (**N** = **16**). The table below shows the number of data misses on the W array for different cache sizes.

W Array Cache Size words (lines x words/line)	Data misses on W
12 (3 x 4)	256
16 (4 x 4)	4

Cache diagrams and code are available on the following pages for assistance and scratch work.

(B) (3 points) Why does increasing the cache size slightly have such a large improvement in the number of misses on the `W` array? Briefly explain.

Explanation:
In order to store the entire weights array in the weights cache, we need four cache lines (say CL0, CL1, CL2 and CL3). Since the 12-word cache only has capacity for three cache lines, every cache line CLi that is brought into the cache evicts CL[(i+1) % 4].

In the second scenario, i.e. the 16-word cache, there are four cache lines, so once the entire W array is brought into the cache after 4 misses (resulting from the first time each line is brought into the cache), there are no more misses and the entire array now resides in the cache.

(C) (4 points) Unfortunately, the TAs will be unable to get more cache in time for the deadline. Help the TAs tile **Version B** of the code with a **tile size of T**. Assume that S and N are divisible by T. Your goal is to minimize the number of misses on W, and you can only tile either S or N. Which one should you tile on? Complete the code below showing a tiled implementation of the code.

```
Version C:
void grade() {
    int sum = 0;

        for ( int n = 0; n < N; n += T               )

            for ( int s = 0; s < S; s++                  )

                for ( int nn = 0; nn < T; nn++                 )

                    sum += W[n + nn] * G[s][n + nn]
}
```

(D) (4 points) Calculate the total number of data-misses on the `W` array using your tiled code from part C. Assume the **tiling factor T = 4** and that you are using the **12 word (3 x 4) cache**.

Cache diagrams and code are available on the following pages for assistance and scratch work.

Data misses on the W array: ________4_________

With a tile size of 4, only 4 elements of array W are accessed at a time and they all fit in one line of the cache. Bringing this line into the cache incurs one data miss. The 4 elements of W are accessed by the two inner loops without having to evict any line of W's cache. So for every S*4 accesses, there is 1 data miss. Once we get to the next iteration of the outermost loop, we need to bring in another line of W into the cache, and the same pattern repeats where we have a total of S*4 accesses with a single data miss. In total, we have 4 data misses, one per line of W.

W array cache diagrams for fully associative caches with <u>12 words</u>, 4 words per block (not graded):

Version B:
```
void grade() {  // S = 64, N = 16
    int sum = 0;
    for (int s = 0; s < S; s++)
        for (int n = 0; n < N; n++)
            sum += W[n] * G[s][n];
}
```

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Extra Copies:

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

W array cache diagrams for fully associative caches with <u>16 words</u>, 4 words per block (not graded):

Version B:
```
void grade() {  // S = 64, N = 16
    int sum = 0;
    for (int s = 0; s < S; s++)
        for (int n = 0; n < N; n++)
            sum += W[n] * G[s][n];
}
```

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Extra Copies:

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

W array cache diagrams for fully associative caches with 12 words, 4 words per block, and a tile size of 4. (not graded):

Version C:
```
void grade() {   // S = 64, N = 16, T = 4
    int sum = 0;
    for (                                        )
        for (                                        )
            for (                                       )
                    sum +=
}
```

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Extra Copies:

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

Tag	Word 3	Word 2	Word 1	Word 0
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]
	W[ ]	W[ ]	W[ ]	W[ ]

END OF QUIZ 3!