1	/18
2	/20
3	/16
4	/16
5	/16
6	/14

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Spring 2023

Quiz #3

Name	*Athena login name*	*Score*

Recitation section

☐ WF 10, 34-302 (Alexandra)	☐ WF 2, 34-302 (Boom)	☐ opt-out
☐ WF 11, 34-302 (Alexandra)	☐ WF 3, 34-302 (Boom)	
☐ WF 12, 34-302(Georgia)	☐ WF 12, 35-308 (Keshav)	
☐ WF 1, 34-302 (Georgia)	☐ WF 1, 35-308 (Keshav)	

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 and Virtual Addresses (18 points)

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

These processes run on a custom operating system that supports segmentation-based (base and bound) virtual memory, timer interrupts for scheduling processes, and a `print_string` system call for printing strings. Additionally, the processor does not support the `pow` (exponentiation) instruction, so the operating system emulates it in software. The syntax for the `pow` instruction is:

```
pow rd, rs1, rs2
```

where Reg[rs1] contains the base number, and Reg[rs2] contains the exponent. The `pow` exception handler returns to the process that initially raised the exception after calculating the result of raising the base number in Reg[rs1] to the power of the exponent in Reg[rs2]. The result is stored in Reg[rd]. No other registers in the calling process are modified by the exception.

As usual, 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 0x13 in register a7.

```
program for process A:
. = 0x100
  li a0, 5
  li a1, 4
  pow a0, a0, a1
  li a2, 0x420
  sw a0, 0(a2)
  li a0, 0x360
  ecall
  ret

. = 0x360
stringA:
  .ascii "process A completed"

program for process B:
. = 0x500
  li a0, 3
  li a1, 5
  pow a0, a0, a1
  li a2, 0x800
  sw a0, 0(a2)
  li a0, 0x620
  ecall
  ret

. = 0x620
stringB:
  .ascii "process B completed"
```

Assume virtual addresses are translated with the following base and bound registers:

process A: base register = 0x50, bound register = 0x400;

process B: base register = 0x460, bound register = 0x700.

(A) (5 points) The OS schedules Process A first, but the processor does not support the `pow` instruction, so the OS emulates it in software. What are the values of a0, a1, a2, and pc (in virtual address) when the common handler returns to Process A after emulating the `pow` instruction? Assume all registers are initialized to 0 when a process starts execution.

a0: __5^4 or 625__

a1: _____4_____

a2: _____0_____

pc: ___0x10C___

Explanation:

After the OS emulates pow and returns control, a0 stores the result of pow (5^4 = 625), a1 stores 4, and a2 is at its initial value, which is 0.

The OS returns control to the instruction following pow, at pc = 0x100 + 3*4 = 0x10C

(B) (5 points) Just prior to Process A executing `li a2, 0x420`, a timer interrupt occurs and the OS switches to Process B. What are the values of a0, a1, a2, and pc (in virtual address) when the common handler returns to Process B? Assume all registers are initialized to 0.

a0: ____0______

a1: ____0______

a2: ____0______

pc: ___0x500___

Explanation:

Since Process B hasn't run yet, the pc is the pc of the first instruction (i.e., 0x500), and a0, a1 and a2 are at their initial value, which is 0.

(C) (4 points) In both Process A and Process B, which instructions (if any) involve illegal memory accesses that cause a segmentation fault? Explain why the instructions in your list result in segmentation faults and explain why all other instructions do not.

List of instructions that result in segmentation faults:

Process A: sw a0, 0(a2)
Process B: sw a0, 0(a2)

Explanation:

The sw instructions in both processes cause segmentation faults because their virtual addresses (0x420 in Process A and 0x800 in Process B) are over process bounds, i.e., 0x400 and 0x700.
The rest of the instructions are located within each process's bound, so they do not cause segmentation faults.

Assume that you correctly fixed Processes A and B so that there is no segmentation fault anymore.

(D) (4 points) During your testing, you notice both Process A and Process B still don't correctly print the string "process A completed" and "process B completed". Explain why and how to fix Process A and Process B so that they print the strings as intended.

Explanation:

Both the processes cannot print correctly because the sys_call code "0x13" is not stored in a7. This means that the "print_string" function will not get called by the syscall_eh exception handler and nothing will be printed.

Fix:

For Process A: insert `li a7, 0x13` between `li a0, 0x360` and `ecall`
For Process B: insert `li a7, 0x13` between `li a0, 0x620` and `ecall`

Problem 2. Virtual Memory (20 points)

Consider a RISC-V processor that has 32-bit virtual addresses, 2^{24} bytes of physical memory, and uses a page size of 2^8 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.*

Number of entries in the page table: ____2^{24}____

Size of page table entry (in bits): ____18____

Size of the page table (in bits): __$18*2^{24}$___

(B) (4 points) A program has been halted right before executing the following instruction, located at virtual address 0x3A0.

```
. = 0x3A0
lw x5, 0(x7) // x7 = 0x52C
sw x6, 4(x8) // x8 = 0x434
```

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	1	0	0xAA
1	1	0	0x43
2	0	---	---
3	0	---	---
4	1	0	0xA
LRU → 5	1	1	0x3B
6	1	1	0x24
next LRU → 7	1	0	0x1
	…		

For each virtual address accessed, please indicate, in the chart below, the virtual address, 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, please write N/A.* Please write all numerical values in hexadecimal.

Virtual Address	VPN	Page Fault (Yes/No)	PPN	Physical Address
0x3A0	0x3	Yes	0x3B	0x3BA0
0x52C	0x5	Yes	0x1	0x12C
0x3A4	0x3	No	0x3B	0x3BA4
0x438	0x4	No	0xA	0xA38

(C) (6 points) Fill in the final version of the Page Table after running the two instructions in part (B). **You may leave a row blank to indicate that the row is unchanged from the original page table.** You do not need to label any LRU entries.

VPN	R	D	PPN
0			
1			
2			
3	1	0	0x3B
4	1	1	0xA
5	1	0	0x1
6			
7	0	---	---
	…		

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

Evicted PPN(s) (hex): ______**0x3B, 0x1**______

Written back PPN(s) (hex): _______0x3B ________

Problem continued on next page.

Now consider using a two-level hierarchical page table where the VPN is divided evenly between the first and second levels of hierarchy, so the 1st and 2nd level have the same number of bits.

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

(D) (4 points) Calculate the following parameters relating to the size of each second-level page table. Each second-level page table entry contains 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^{12}______

Size of 2nd level page table entry (in bits): _____18______

Size of one 2nd level page table (in bits): ____$18*2^{12}$____

$18*2^{12}$ bits = $18 * 2^9 * 2^3$ bits = $18 * 2^9$ bytes = $36 * 2^8$ bytes = 36 pages

Number of pages required to hold one 2nd level page table: _____36______

(E) (4 points) Assume for simplicity that the size of each 1st level page table entry is equal to the size of a 2nd level page table entry (in bits). How much memory is needed to store the entire two-level hierarchical page table of a program that uses *only* the bottom 2Mbytes (2^{21} Bytes) of virtual addresses? First, find the number of pages required to hold the 1st level page table. Then, find the number of 2nd level page tables required for this process. *Your final answer can be a product or exponent.*

2^{12} * 18 bits = 36 pages

Number of pages required to hold 1st level page table: _____36_____

2 Mbytes = 2^{21} Bytes
PPN bits = 21 – 8 (page offset bits) = 13
Physical addresses for this process = 2^{13}
Each 2nd level page table holds the mapping of 2^{12} physical addresses, so we need 2 2nd level page tables to map 2^{13} addresses

Number of 2nd level page tables required for 2MByte process: ____2_____

36 + 2 * 36 = 108
Number of pages needed to store the hierarchical page table of this process: ____108___

Problem 3: Dingo the Exception Detective (16 points)

Dingo is trying to write a program for his RISC-V Operating System. Unfortunately, he got ahead of himself and did not test his exception handler implementation. Instead, he just started writing a user-space assembly program and is now wondering why it's not working. Help him figure out what's wrong with his work-in-progress program and handler!

User-space Program

```
.= 0x100
main:
    addi a1, zero, 0x600
    lw a0, 0(a1)
    lw a2, -4(a1)
    beqz a2, mylabel
    slli a3, a1, 4
    addi a0, a0, 4
    j done

mylabel:
    .word 0xdeadcafe        // Invalid instr
    addi a3, zero, 0x400
    sw a0, 0(a3)
    sw a0, 0(a1)
done:
    j done
```

Common Handler

```
handler:
    mret
    addi a4, a4, 1
    csrr a5, mepc
    addi a5, a5, 4
    csrw mepc, a5
```

Dingo found that **the first `lw` instruction triggers an exception because `0x600` is not mapped into the program's memory space.** Assume that exceptions are **handled lazily** before entering the commit point (i.e., exceptions are triggered right before the instruction that causes the exception enters the Write Back stage). Also assume that the **`mret`** instruction acts like a branch instruction in that branch decisions are resolved in the EXE stage. The **`mret`** instruction updates the pc to the value in the **`mepc`** register.

(A) (6 points) Help Dingo fill out the pipeline diagram of the running program (starting at `main`) and answer the question below. Assume full bypassing. You do not need to show the use of bypass paths.

Cycle	0	1	2	3	4	5	6	7	8	9
IF	addi	lw	lw	bez	slli	mret	addi	csrr	lw	lw
DEC		addi	lw	lw	bez	NOP	mret	addi	NOP	lw
EXE			addi	lw	lw	NOP	NOP	mret	NOP	NOP
MEM				addi	lw	NOP	NOP	NOP	mret	NOP
WB					addi	NOP	NOP	NOP	NOP	mret

What instruction is in the IF stage in cycle 22? ______ lw _______

(B) (6 points) Dingo thinks he found the problem and updated his exception handler. Now the program runs to completion (i.e., it reaches the `done` label). Dingo thinks multiple exceptions occur while executing the program. He knows the first `lw` still causes an exception because **`0x600` is not mapped into program's memory space**, but he's not sure which other instructions cause exceptions. To help you figure out which instructions caused an exception, Dingo provides a register dump that shows the contents of some registers at the time the process begins to repeatedly execute the `j done` instruction.

Assume that none of the instruction fetches cause exceptions. Also, assume all registers are zero at the start of execution. **Note that this exception handler does not save the state of the interrupted process, so registers are shared between the user program and the exception handler.**

For each instruction that triggers an exception, mark the corresponding `[ ]` box with an `X`. Additionally, fill in the missing values of the register dump. *Hint: You can deduce many values in the code based on knowing the total number of exceptions triggered by this program.*

As a reminder, `csrr rd, mepc` reads the value of the `mepc` register, writing it into the `rd` register. Likewise, `csrw mepc, rs1` writes the `mepc` register with the value of register `rs1`. `mret` returns to the address in the `mepc` register.

```
Triggers      User-space Program          New Common Handler      Register Dump
Exception?

           .= 0x100                     handler:                 a1: __0x600__
           main:                            addi a4, a4, 1
    [   ]      addi a1, zero, 0x600         csrr a5, mepc        a2: ____0____
    [ X ]      lw a0, 0(a1)                 addi a5, a5, 4
    [   ]      lw a2, -4(a1)                csrw mepc, a5        a3: __0x400__
    [   ]      beqz a2, mylabel             mret
    [   ]      slli a3, a1, 4                                    a4: ___0x3___
    [   ]      addi a0, a0, 4
               j done                                            a5: __0x12C__

           mylabel:                                            mepc: __0x12C__
    [ X ]      .word 0xdeadcafe         // invalid instr
    [   ]      addi a3, zero, 0x400
    [   ]      sw a0, 0(a3)
    [ X ]      sw a0, 0(a1)
           done:
               j done
```

(C) (4 points) Dingo wants to emulate a new instruction (which his RISC-V processor does not implement) using his exception handler: the **`0xdeadcafe`** instruction. Dingo wants the opcode for this instruction to be `0xdeadcafe`. As you may have noticed, he already added this instruction to his program (with `.word 0xdeadcafe`). Now he wants to implement its functionality. Dingo wants this instruction, when executed, to set registers `a0` and `a1` to `0xdeadcafe`. Other exceptions should not change `a0` and `a1`. How can he achieve this by modifying his common handler? In your solution, use only temporary registers (`t0-t6`) and `a0`, `a1`, and `a5`.

```
handler:
    addi a4, a4, 1
    csrr a5, mepc

    li t0, 0xdeadcafe
    lw t1, 0(a5)
    bne t1, t0, return
    li a0, 0xdeadcafe
    li a1, 0xdeadcafe

    // or equivalent…



return:
    addi a5, a5, 4
    csrw mepc, a5
    mret
```

Problem 4. Synchronization (16 points)

Martha is opening a new pancake restaurant, and for the grand opening, she plans to have 100 guests over. She will be serving them all their special – a stack of 3 surprise pancakes, being any selection of blueberry, chocolate chip, banana, Nutella, or peanut butter pancakes.

However, she can't make them all herself before the event starts, so she employs multiple pancake chefs to help her. Each chef operates as a thread running the `make_pancakes` function, whose pseudocode is shown below. Her one limitation is that her kitchen has only one pan.

Shared Memory:

```
// flavors is an array containing all possible flavors
flavors = ["blueberry", "chocolate", "banana", "nutella",
           "peanut butter"]
num_flavors = 5
flavor_idx = 0;
```

```
make_pancakes:

    //get next pancake flavor
    ingredient = flavors[flavor_idx]
    flavor_idx = (flavor_idx + 1) % num_flavors

    get_ingredients(ingredient)

    whisk()

    cook_on_pan()

    add_to_stack()

    goto make_pancakes
```

(A) (2 points) Suppose two threads, A and B, are running the `make_pancakes` code above without any synchronization. For each of the following failure scenarios, circle whether it is possible or not:

1. Initially, A and B start making the same flavor of pancake

   Possible / **Not Possible**

2. A and B both use the pan at the same time, resulting in a disgusting mix of flavors

   Possible / **Not Possible**

Martha has found parallelized cooking to be really efficient, but she realized she forgot about the plating and serving side of the restaurant! She has hired one waiter who can remove pancakes from the 1 stack made by all the chefs together to plate 3 pancakes at a time on a plate to then serve to the guests.

However, the waiter is also busy managing the storefront and doesn't want to plate a stack of 3 pancakes to then serve if there are not at least 3 pancakes in the stack. Also, the stack will fall over if there are more than 9 pancakes, so a chef must wait to stack a cooked pancake if there are already 9 pancakes in the stack. Martha's kitchen also only has 1 pan, so only one chef can be using the pan to cook their pancake at one time. Assume that there are more than 1 but fewer than 5 chefs and thus threads running the `make_pancakes` function, and that there is a single waiter and thus one thread running the `serve_pancakes` function.

(B) (14 points) Define and add semaphores on the next page to enforce these constraints:

1. Each chef should make the next available flavor of pancakes following the previous chef to select a flavor, with the first available flavor being blueberry and after a peanut butter pancake, blueberry should be made next.
2. Only one chef can be using the pan to cook their pancake at one time.
3. The stack should never have more than 9 pancakes.
4. stack_3() should never be called until there are at least 3 pancakes on the stack.
5. After 300 pancakes are made for the 100 guests, no more pancakes should be made.
6. As long as there are still pancakes left to be made, avoid deadlock.
7. Use no more than 5 semaphores, and do not add any additional precedence constraints.

Shared Memory:

```
// flavors is an array containing the possible flavors
flavors = ["blueberry", "chocolate", "banana", "nutella", "peanut butter"]
num_flavors = 5
flavor_idx = 0;

// Specify your semaphores and initial values here
remaining_orders = 300
pancake_idx = 1
pan = 1
stack_space = 9
on_stack = 0
```

make_pancakes:

```
    wait(remaining_orders)

    wait(pancake_idx)

    //get next pancake flavor
    ingredient = flavors[flavor_idx]
    flavor_idx = (flavor_idx + 1) % num_flavors

    signal(pancake_idx)

    get_ingredients(ingredient)

    whisk()

    wait(pan)

    cook_on_pan()

    signal(pan)

    wait(stack_space)

    add_to_stack()

    signal(on_stack)

    goto make_pancakes
```

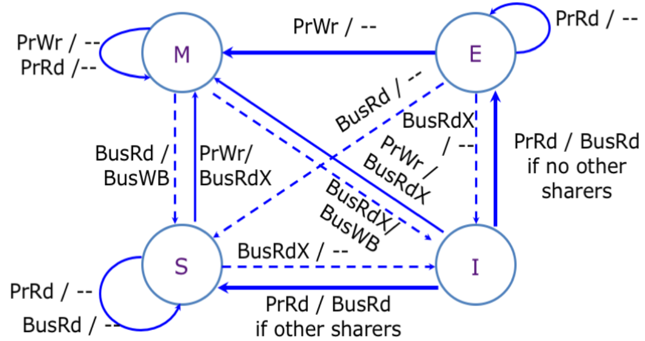
serve_pancakes:

```
    wait(on_stack)
    wait(on_stack)
    wait(on_stack)

    stack_3()

    signal(stack_space)
    signal(stack_space)
    signal(stack_space)

    serve()

    goto serve_pancakes
```

Problem 5. Cache Coherence (16 points)

Alice and Bob are two threads that exchange messages through a shared memory location L. Alice and Bob take turns accessing L: periodically, each of them wakes up, reads the message in L, and writes a new message in L for the other one to read. This results in the following memory access sequence:

```
Alice: read L
Alice: write L
Bob: read L
Bob: write L
Alice: read L
Alice: write L
                    ...
```



Suppose that Alice and Bob run in two processor cores with private caches, kept coherent with a snoopy, bus-based, write-invalidate MESI protocol (whose state-transition diagram is shown above). Assume write-back, write-allocate caches.

(A) (4 points) Fill in the following table showing the bus transactions that result from each access, and the states for L's cache line after each access.

Access	*Shared bus transactions*	*Alice's cache*	*Bob's cache*
Initial state		L: I	L: I
After Alice reads L	BusRd	L: E	L: I
After Alice writes L		L: M	L: I
After Bob reads L	BusRd, BusWB	L: S	L: S
After Bob writes L	BusRdX	L: I	L: M
After Alice reads L	BusRd, BusWB	L: S	L: S
After Alice writes L	BusRdX	L: M	L: I

(B) (2 points) We are interested in minimizing the number of cache misses, i.e., accesses that cannot be satisfied by the local cache and result in a bus transaction. In steady state (i.e., after Alice and Bob have exchanged many messages), what is the hit rate of this sequence of accesses? **Consider every access that requires a bus transaction as a miss.**

100 % misses in steady state (I->S for reads, and S->M for writes)

Hit Rate: ______0______

(C) (2 points) Would using an MSI protocol (instead of MESI) improve hit rate? Briefly explain why or why not.

No, MSI would work the same way in steady state, as the E state is not used.

(D) (5 points) We add a *self-invalidation* instruction to the processor: `inv <address>` invalidates the cache line containing `<address>`. If the line is dirty, it is written back to main memory (through a BusWB transaction). Alice and Bob are modified to run `inv L` after they `write L`.

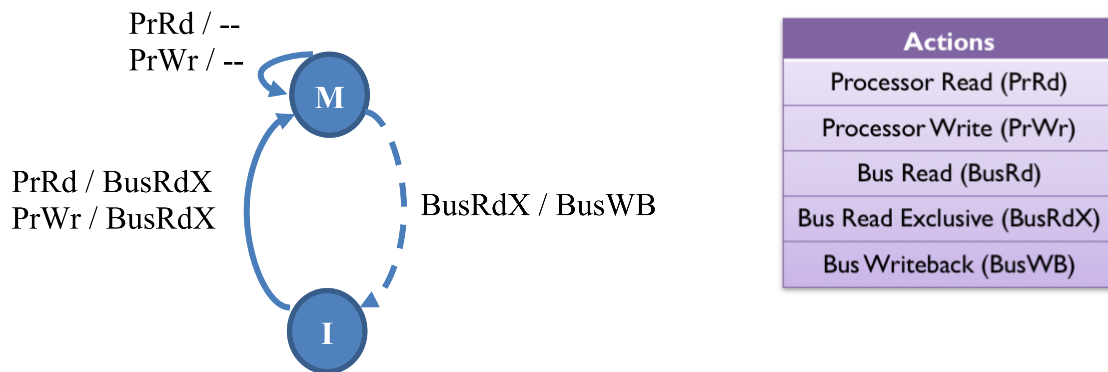
Fill in the diagram below, assuming that we use a MESI protocol. What is the hit rate of this access sequence in steady state? (**Consider only reads and writes as accesses; invs are not accesses.**)

Access	*Shared bus transactions*	*Alice's cache*	*Bob's cache*
Initial state		L: I	L: I
After Alice reads L	BusRd	L: E	L: I
After Alice writes L		L: M	L: I
After Alice invs L	BusWB	L: I	L: I
After Bob reads L	BusRd	L: I	L: E
After Bob writes L		L: I	L: M
After Bob invs L	BusWB	L: I	L: I

Hit rate for access sequence in steady state: _____50_____ %

(E) (3 points) This inv mechanism is too complicated for what we want to achieve–let's keep it simple. Write the state-transition diagram for a **two-state coherence protocol** that improves hit rate over MESI for this access sequence. Name each of the two states, and include transitions to cover all processor and bus actions in the protocol. The table below shows all possible actions; you must support processor reads and writes, but need not use all possible bus actions in your protocol.

Your protocol should be write-invalidate, and work for write-back, write-allocate caches. Using your protocol, what is the hit rate of our access sequence?



Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Read Exclusive (BusRdX)
Bus Writeback (BusWB)

Hit rate for access sequence in steady state with your protocol: _____50_____ %

Problem 6: Loop Ordering and Caches (14 points)

Consider the following program:

```
int A[8][8];
int B[8][8];
int C[8][8][8];

for (int i = 0; i < 8; i++) {
  for (int j = 0; j < 8; j++) {
     for (int k = 0; k < 8; k++) {
       C[i][j][k] = A[i][j] + B[j][k];
     }
   }
}
```

Consider a **three-way set associative cache with 4 sets. The block size is 4 words.** This cache is only used for data, and not for instructions. **The cache is partitioned across the arrays so that all accesses to array A map to way 0, array B map to way 1, and array C map to way 2 of the cache.** Recall that arrays are stored in row major order in memory. For array C this means that the k elements of each C[i][j] are stored consecutively in memory, then come the next value of j with its k elements, and so on.

(A) (2 points) In the inner "k" loop, how many different elements of each array are accessed?

Inner "k" loop:

```
for (int k = 0; k < 8; k++) {
  C[i][j][k] = A[i][j] + B[j][k];
}
```

Elements of A: _____1_____

Elements of B: _____8_____

Elements of C: _____8_____

(B) (2 points) What fraction of the accesses to elements of C are cache **misses** for the entire program?

Each access to C accesses a new element not seen before. C is traversed in row-by-row order, so the next element of C accessed is directly adjacent in memory. This means that when a block is brought in from memory from a miss on an access to C, it contains the next 3 elements to be accessed by C. Thus, only one in 4 accesses are a cache miss as after each miss there will be 3 following hits.

Order of C[i,j,k] array in its cache for i = 0, j = 0 and j = 1:
C[0,0,0] … C[0,0,3]
C[0,0,4] … C[0,0,7]
C[0,1,0] … C[0,1,3]
C[0,1,4] … C[0,1,7]

Miss Rate: ______1/4_______

(C) (2 points) What fraction of the accesses to elements of B are cache **misses** for the entire program?

B is accessed in row-by-row order, meaning all accesses are sequential in memory. Like array C, when a block is brought in, it contains 4 elements, so only one in 4 accesses are cache misses. Since only two rows of array B fit in the cache at a time, by the time the loop goes back to j = 0, row 0 has been replaced and needs to be brought back into the cache.

Order of B[j,k] array in its cache for j = 0 and j = 1:
B[0,0] … B[0,3]
B[0,4] … B[0,7]
B[1,0] … B[1,3]
B[1,4] … B[1,7]

Miss Rate: ______1/4______

(D) (2 points) What fraction of the accesses to elements of A are cache **misses** for the entire program?

A is traversed in row-by-row order. The same element of A is accessed in the inner k loop 8 different times. It will not be kicked out of the cache, since it is accessed in every calculation and will not be the LRU. Thus, once a block is brought in for A, the next three elements will stick around for subsequent inner k loops. We only need one initial miss of an element of A to cover hits for four inner k loops, or 32 total accesses to A.

Order of A[i][j] array in its cache for i = 0 and i = 1:
A[0,0] … A[0,3]
A[0,4] … A[0,7]
A[1,0] … A[1,3]
A[1,4] … A[1,7]

Miss Rate: ______1/32______

Consider reordering the loops as follows (swapping the j and k loops):

```
for (int i = 0; i < 8; i++) {
  for (int k = 0; k < 8; k++) {
     for (int j = 0; j < 8; j++) {
       C[i][j][k] = A[i][j] + B[j][k];
     }
   }
}
```

Now answer the following questions with the same data cache as before.

(E) (2 points) What fraction of the accesses to elements of C are cache **misses** for the entire program?

C is now traversed in column-by-column order. When C[i,0,0] is brought into cache so are C[i,0,1-3]. However, now our inner loop is j, so next you will access C[i,1,0] which is not in the cache. This will be brought into set 2. Then C[i,2,0] will be brought into set 0 and evict

C[i,0,0], so by the time you get to k = 1, you have evicted C[i,0,1] from the cache. This means that you don't get to take advantage of the special locality and you miss every single time.

Miss Rate: ______1_______

(F) (2 points) What fraction of the accesses to elements of B are cache **misses** for the entire program?

B is now traversed in column-by-column order. You first bring in B[0][0-3] into set 0. Next you bring B[1][0-3] into set 2. By the time you get to B[2][0-3] you need to evict B[0][0-3] so after you complete a whole column, you have evicted the low value of j so once again you miss every time.

Miss Rate: _____1_______

(G) (2 points) What fraction of the accesses to elements of A are cache **misses** for the entire program?

A continues to be accessed in row-by-row order. You first bring A[0,0-3] into set 0, A[0,4-7] into set 1. So out of the 8 values of the inner loop j, you miss twice and hit 6 times. For the next 7 values of k you get all hits. So you have a miss rate of 2/64 or 1/32. Once you move to the next value of i, the process repeats itself.

Miss Rate: _____1/32_______

END OF QUIZ 3!