

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

**6.191 Computation Structures**  
Spring 2024

1	/12
2	/18
3	/18
4	/17
5	/16
6	/19

**Quiz #3**

<i>Name</i>	<i>Athena login name</i>	<i>Score</i>
<i>Recitation section</i>		
<input type="checkbox"/> WF 10, 34-302 (Wendy) <input type="checkbox"/> WF 2, 34-302 (Catherine) <input type="checkbox"/> opt-out <input type="checkbox"/> WF 11, 34-302 (Wendy) <input type="checkbox"/> WF 3, 34-302 (Catherine) <input type="checkbox"/> WF 12, 34-302 (Adrianna) <input type="checkbox"/> WF 12, 35-308 (Shabnam) <input type="checkbox"/> WF 1, 34-302 (Adrianna) <input type="checkbox"/> WF 1, 35-308 (Shabnam)		

**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 (12 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.

Program for process A	Program for process B
<pre> . = 0x10 li t1, 0x20 add t0, t0, t1 addi t1, t0, 4 li a1, 0x505 sw a1, 0(t0) unimp                     </pre>	<pre> . = 0x1000 xori t0, t0, 0x60 sw t1, 4(t0) lw a1, 4(t0) add t0, a1, t1 li t1, 0x100 unimp                     </pre>

These processes run on a custom OS that supports segmentation-based (base and bound) virtual memory, and timer interrupts for scheduling processes. This OS has two unusual features:

First, this OS uses **very frequent timer interrupts** to interleave the execution of both processes: process A is interrupted after executing only two instructions, and process B is interrupted after executing only one instruction.

Second, the OS kernel **does not save, restore, or access the temporary registers (t0-t6)** when servicing exceptions or context-switching across processes.

Assume if a register has not been modified as a result of an instruction, its default value is zero.

(A) (5 points) The OS schedules process B first. What are the values of registers **t0**, **t1**, **a1**, **pc** (in virtual address) in processes A and B after the **fifth** timer interrupt takes place?

**Process A:**

t0: \_\_\_\_\_  
t1: \_\_\_\_\_  
a1: \_\_\_\_\_  
pc: \_\_\_\_\_

**Process B:**

t0: \_\_\_\_\_  
t1: \_\_\_\_\_  
a1: \_\_\_\_\_  
pc: \_\_\_\_\_

**Process that OS returns control to after fifth timer interrupt:** \_\_\_\_\_

(B) (3 points) Consider the same situation, but the OS now schedules process A first. What are the values of registers `t0`, `t1`, `a1`, `pc` (in virtual address) in processes A and B after the **third** timer interrupt is observed?

**Process A:**

`t0:` \_\_\_\_\_

`t1:` \_\_\_\_\_

`a1:` \_\_\_\_\_

`pc:` \_\_\_\_\_

**Process B:**

`t0:` \_\_\_\_\_

`t1:` \_\_\_\_\_

`a1:` \_\_\_\_\_

`pc:` \_\_\_\_\_

**Process that OS returns control to after third timer interrupt:** \_\_\_\_\_

(C) (4 points) Since temporary registers are shared among processes in this OS, we may be able to use them to implement synchronization primitives without using shared memory. Consider the implementation of a semaphore below using the `t6` register:

```
signal:  addi t6, t6, 1
```

```
wait:    ble t6, zero, wait
         addi t6, t6, -1
```

Assume that `t6` is not used by any other code, and that processes can be interleaved arbitrarily. Does this implementation work, and under what circumstances? **Circle one of the options below and explain your answer.** Your explanation should justify the circumstances in which this implementation works, and unless you find it works in all cases, include a counterexample showing when it breaks.

- 1) Does not work even with two processes
- 2) Works only up to two processes
- 3) Works only up to three processes
- 4) Works with any number of processes

**Explanation:**

**Problem 2. Virtual Memory (18 points)**

Consider a RISC-V processor that uses  $2^{28}$  bytes of virtual memory,  $2^{24}$  bytes of physical memory, and uses a page size of  $2^{13}$  bytes.

- (A) (2 points) Calculate the following parameters related to the size of the page table. Assume 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:** \_\_\_\_\_

**Size of page table entry (in bits):** \_\_\_\_\_

- (B) (2 points) If we changed the RISC-V processor to use a page size of  $2^{12}$  bytes, how would each of the following change? *Assume this is the only change being made to the system.*

**Number of virtual pages (select one of the choices below):**

UNCHANGED ... +1 ... -1 ... 2x ... 0.5x ... CAN'T TELL

**Number of physical pages (select one of the choices below):**

UNCHANGED ... +1 ... -1 ... 2x ... 0.5x ... CAN'T TELL

**Size of page table entries (in bits) (select one of the choices below):**

UNCHANGED ... +1 ... -1 ... 2x ... 0.5x ... CAN'T TELL

**Number of page offset bits in virtual address (select one of the choices below):**

UNCHANGED ... +1 ... -1 ... 2x ... 0.5x ... CAN'T TELL

Assume  $2^{12}$  byte pages for the rest of the problem.

(C) (4 points) A program has been halted right before executing the following instruction, located at virtual address 0x674.

```
. = 0x674
lw x6, 8(x3) // x3 = 0x4FF8
```

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.

VPN	R	D	PPN
0	1	0	0x13
1	1	0	0xFA
2	1	1	0x33
LRU → 3	1	0	0xB
4	1	0	0x1
5	0	---	---
6	1	1	0x20
7	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 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
0x	0x		0x	0x
0x	0x		0x	0x

(D) (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 0x34A0:

```
. = 0x34A0
lw x3, 0(x4) // x4 = 0xCACA0
```

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

VPN	V	R	D	PPN
0xD1	1	1	0	0x0
0xCA	1	1	1	0x31
0xCAC	1	1	1	0x8
0x7	1	1	0	0x99

LRU →

VPN	R	D	PPN
0	1	0	0x12
1	1	1	0xFA
2	1	1	0x30
3	0	---	---
4	1	0	0xE
5	1	1	0x4
6	1	0	0x70
7	1	1	0x85
...			

LRU → 2  
Next LRU → 4

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.

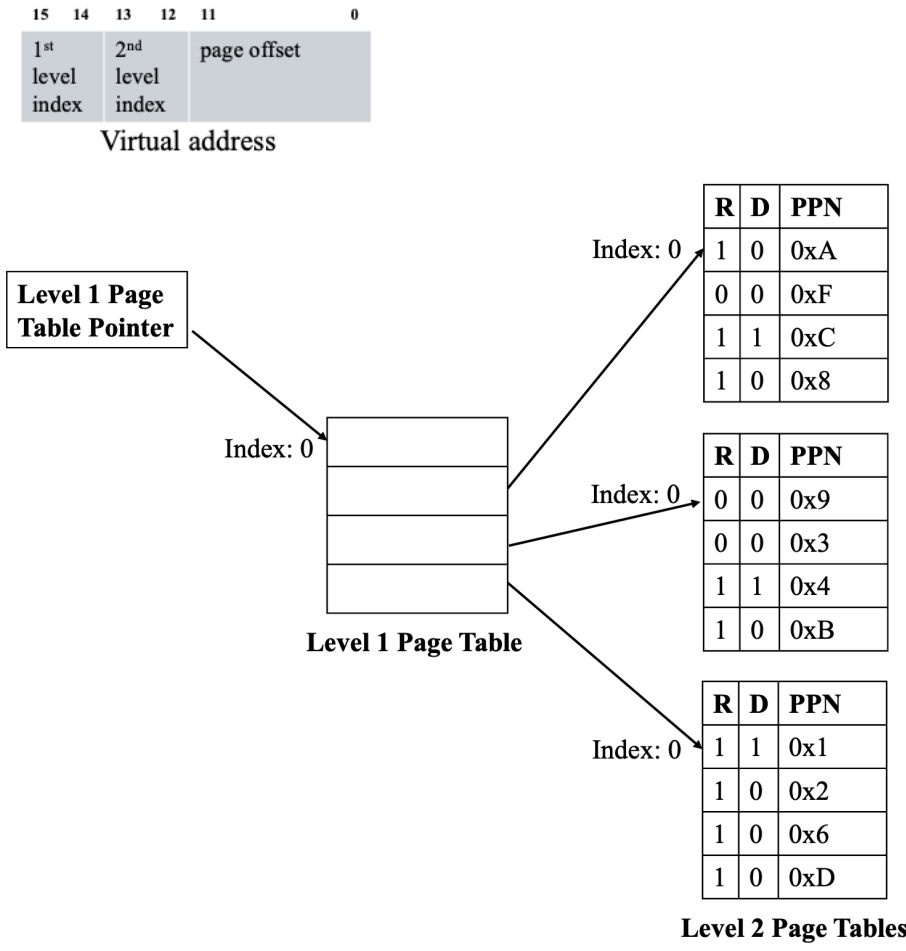
Virtual Address	VPN	TLB Hit (Yes/No)	Page Fault (Yes/No)	PPN	Physical Address
0x	0x			0x	0x
0x	0x			0x	0x

(E) (2 points) Using the page table from part (D), please indicate the PPN corresponding to the physical page that would be evicted upon a page fault, and if that physical page would need to be written back to disk.

**Evicted physical page number (0x):** \_\_\_\_\_

**Writeback necessary (Yes/No):** \_\_\_\_\_

(F) (3 points) Now consider a different processor with a 4-bit VPN that is translated to a PPN using the two-level hierarchical page table shown below. The top 2 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 12 bits of the virtual address are the page offset.



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
0x9345	0x_____
0xEBAA	0x_____
0x7899	0x_____

### Problem 3. Exceptions (18 points)

Paige Tabelle is using a RISC-V system with segmentation-based virtual memory and is currently running Process A in user space with base register =  $0x300$  and bound register =  $0x1000$ .

(A) (2 points) For the following 2 accesses, find whether a bound violation exception occurs. If the virtual address is in bounds, translate it to a physical address. Otherwise, write N/A.

i.  $0xD50$     **Out of bounds:**    **YES**    **NO**    **Physical Address:** \_\_\_\_\_

ii.  $0x200$     **Out of bounds:**    **YES**    **NO**    **Physical Address:** \_\_\_\_\_

Paige did not test her implementation of Process A, so her program encounters several exceptions during execution. To handle these exceptions, Paige implements a toy handler in kernel-space that counts the number of exceptions before continuing execution of Process A.

Process A runs on a standard RISC-V processor. Assume all registers are zero at the start of execution, and that exceptions are enabled.

(B) (4 points) Help Paige determine which instructions in Process A trigger an exception. In the listing below, write a **Y** in the brackets in the first column for each instruction that causes an exception. Leave all other brackets blank.

```

Instr.      // Process A code           // Toy handler in
address    .= 0xFC8                      // kernel space
[ ] 0x0FC8  li a0, 0x0
[ ] 0x0FCC  li a7, 0x18
[ ] 0x0FD0  ecall
[ ] 0x0FD4  li a0, 0x100
[ ] 0x0FD8  li a1, 0x400
[ ] 0x0FDC  li a2, 0x345
           loop:
[ ] 0x0FE0      sw  a2, 0(a0)
[ ] 0x0FE4      addi a0, a0, 4
[ ] 0x0FE8      blt  a0, a1, loop
[ ] 0x0FEC      lui  a3, 0x2
[ ] 0x0FF0      lw  a4, 0(a3)
[ ] 0x0FF4      .word 0xcafe
[ ] 0x0FF8      mv  a2, zero
[ ] 0x0FFC      add  a1, a1, zero
[ ] 0x1000      add  a0, a0, zero
[ ] 0x1004      ret

           xori a7, a7, 2 // some extra
           andi a7, a7, 7 // code
```



In the questions below, assume we use a standard 5-stage pipelined processor with **full bypassing**, where branches are predicted not taken and resolved in EXE. You **do not** need to show any bypass paths that are used.

(C) (4 points) Fill in the pipeline diagram below beginning with the `blt` instruction at the end of loop. Assume execution is on the **last loop iteration** (`a0 = 0x400` when execution reaches the `blt` instruction). Assume that exceptions are **handled lazily** (i.e., at the commit point).

	100	101	102	103	104	105	106	107
IF	blt							
DEC								
EXE								
MEM								
WB								

(D) (4 points) Paige explores other ways of exception handling. Assume now that exceptions are **handled eagerly** (i.e., in the stage that triggers them, like branches). Fill in the pipeline diagram below beginning with the `blt` instruction at the end of loop. Assume execution is on the **last loop iteration** (`a0 = 0x400` when execution reaches the `blt` instruction).

	100	101	102	103	104	105	106	107
IF	blt							
DEC								
EXE								
MEM								
WB								

(E) (4 points) Assume that the `add a0, a0, zero` instruction raises an exception. Paige's toy handler has almost finished handling this exception. Complete the pipeline diagram below to show how the handler returns execution to Process A. Assume that exceptions are **handled lazily**, and that `mret` is resolved in the EXE stage, like normal branches.

	300	301	302	303	304	305	306
IF	addi	csrw	mret				
DEC		addi	csrw				
EXE			addi				
MEM							
WB							

#### Problem 4. Synchronization (17 points)

Octavian the octopus has enlisted a horde of his baby brothers to make paper flower bouquet party favors for his older sister's birthday party, and simultaneously trick them into practicing their addition skills by keeping a counter on a shared chalkboard.

Octavian has come up with a procedure for his baby brothers to follow, the `bouquetHelper` function described in pseudocode below. Unfortunately, his baby brothers are bad at sharing, so they can't take flowers from the table at the same time.

<b>Shared Memory:</b>  <code>leaf_counter = 0;</code>
<b>bouquetHelper:</b> <code>take_3_flowers_from_table() assemble_bouquet() count = count_leaves() sum = leaf_counter + count leaf_counter = sum put_away_bouquet() goto bouquetHelper</code>

(A) (3 points) Suppose three of Octavian's brothers, Orange, Olive, and Ochre, follow the `bouquetHelper` code above without any synchronization. Assume they add numbers correctly. For each of the following failure scenarios, circle whether it is possible or not:

1. `leaf_counter` is at value 5. Orange counts 1 leaf and Olive counts 3, and `leaf_counter` is still at 5 after Orange and Olive update it:  
**Possible / Not Possible**
2. `leaf_counter` is at value 0. Orange counts 6 leaves, Olive counts 2, and Ochre counts 4. `leaf_counter` ends up being 10:  
**Possible / Not Possible**
3. Olive and Ochre try to take paper flowers from the table at the same time, which causes them to start fighting:  
**Possible / Not Possible**

Octavian is in charge of actually making the paper flowers, via the `flowerMaker` procedure. Armed with a stack of colored paper, he cuts out each individual flower, folds it into the right shape, then places it on the table for his baby brothers to take. One sheet of paper can be used for 2 flowers, and Octavian doesn't want to pick up more sheets of paper than needed.

In addition, the table only has enough space for 12 paper flowers before they start falling on the ground, and remember that Octavian's baby brothers cannot be trusted to take flowers from the table at the same time. Assume that Octavian has at least 3 baby brothers helping him, using the `bouquetHelper` procedure, and his goal is to make a total of 60 bouquets.

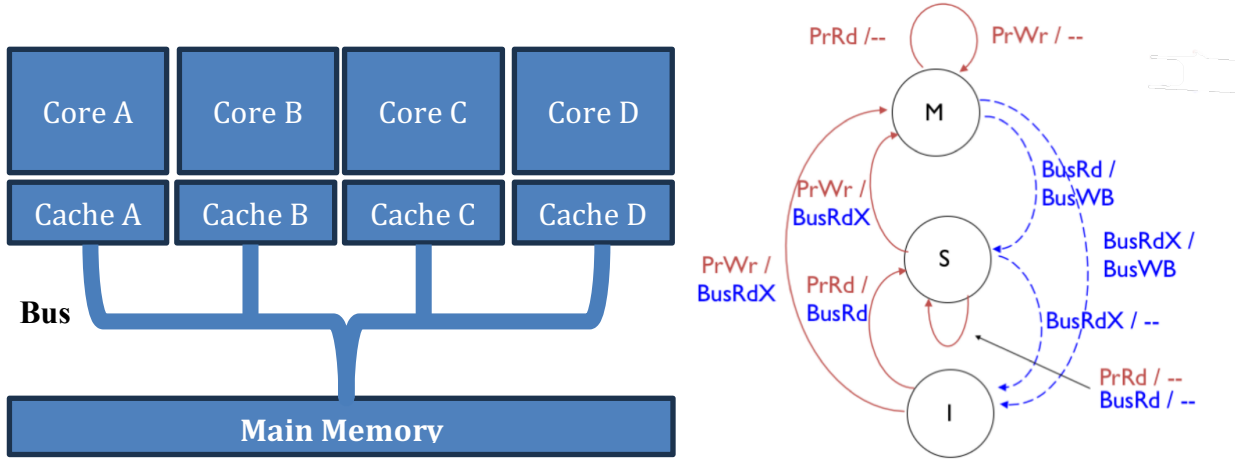
(B) (14 points) **Define and add semaphores below to enforce these constraints:**

1. Octavian and his baby brothers should finish making exactly 60 bouquets, with 3 paper flowers per bouquet, and should not begin any steps for making more bouquets.
2. The final value of `leaf_counter` is the total number of leaves in all of the bouquets.
3. The table holds at most 12 paper flowers at any given time.
4. Only 1 baby brother can take paper flowers from the table at the same time. There must be at least 3 flowers on the table whenever a brother tries to take flowers.
5. Before all 60 bouquets have been finished, avoid deadlock.
6. Use at most 5 semaphores, and do not add any additional precedence constraints.

<b>Shared Memory:</b> <code>leaf_counter = 0;</code> <code>// Specify your semaphores and initial values here</code>	
<b>flowerMaker:</b>  <code>get_sheet_of_paper()</code>  <code>cut_and_fold_flower()</code>  <code>put_flower_on_table()</code>  <code>cut_and_fold_flower()</code>  <code>put_flower_on_table()</code>  <code>goto flowerMaker</code>	<b>bouquetHelper:</b>  <code>take_3_flowers_from_table()</code>  <code>assemble_bouquet()</code>  <code>count = count_leaves()</code>  <code>sum = leaf_counter + count</code>  <code>leaf_counter = sum</code>  <code>put_away_bouquet()</code>  <code>goto bouquetHelper</code>

**Problem 5. Cache Coherence (16 points)**

Consider a four-core system where each core has a private cache. Caches are kept coherent using a **snoopy-based, write-invalidate MSI protocol**, shown below. The **caches are initially empty**.

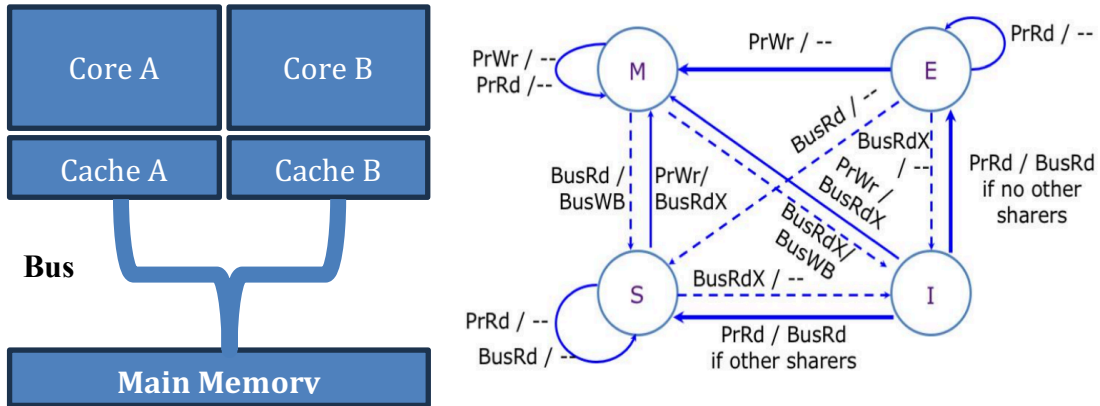


(A) (6 points) Complete the *Access* and *Shared bus transactions* columns of the following partially filled in table. **If there are multiple bus transactions, list them in order.**

Core A	Core B	Core C	Core D
LD X	LD X	LD X	LD X
ST X	ST X	ST X	ST X

<i>Access</i>	<i>Shared bus transactions</i>	<i>Cache A</i>	<i>Cache B</i>	<i>Cache C</i>	<i>Cache D</i>
Initial state		X: I	X: I	X: I	X: I
Core: ___ Inst: _____	_____	X: I	X: S	X: I	X: I
Core: ___ Inst: _____	_____	X: I	X: M	X: I	X: I
Core: ___ Inst: _____	_____	X: S	X: S	X: I	X: I
Core: ___ Inst: _____	_____	X: S	X: S	X: S	X: I
Core: ___ Inst: _____	_____	X: M	X: I	X: I	X: I
Core: ___ Inst: _____	_____	X: S	X: I	X: I	X: S
Core: ___ Inst: _____	_____	X: I	X: I	X: I	X: M
Core: ___ Inst: _____	_____	X: I	X: I	X: M	X: I

(B) (6 points) Now consider a **two-core** system where each core has a private cache. These caches are kept coherent using a **snoopy-based, write-invalidate MESI protocol**, as shown below. The **caches are initially empty**.



We examine a new sequence of accesses on addresses X and Y. Fill in the following table showing the bus transactions that result from each access, and the cache contents, along with the state of each line. **Each cache can hold only one line**. Clearly indicate the line in the cache and its state (e.g., “X: P”, “Y: S”, etc.). If there are multiple bus transactions, list them in order.

<i>Access</i>	<i>Shared bus transactions</i>	<i>Cache A</i>	<i>Cache B</i>
Initial state		X: I	X: I
Core: A Inst: LD X			
Core: A Inst: ST X			
Core: A Inst: ST Y			
Core: B Inst: LD Y			
Core: B Inst: ST X			
Core: A Inst: ST Y			
Core: A Inst: ST X			
Core: B Inst: ST X			

(C) (4 points) Consider the same two-core system as before. We're interested in comparing the performance of the MESI and MSI protocols.

*Note:* In the questions below, it is sufficient to consider sequences of at most three accesses, and **any sequence you give should have at most three accesses.**

a. Are there access sequences that result in more traffic with MSI than MESI? If so, give an access sequence that shows more bus transactions in MSI. If not, explain why not.

b. Are there access sequences that result in more traffic with MESI than MSI? If so, give an access sequence that shows more transactions in MESI. If not, explain why not.

**Problem 6. Parallel Processing and Performance Engineering (19 points)**

(A) (5 points) Reorder the following loops to maximize data locality. Assume arrays are stored in row-major order, and each cache line stores multiple elements of each array.

(i)

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i][i] += A[j][j] * B[j][i];
```

*Reordered loop nest:*

```
for (_____ )
  for (_____ )
    C[i][i] += A[j][j] * B[j][i];
```

(ii)

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      for (int m = 0; m < N; m++)
        C[i][m][k] += A[k][i][m] * B[m][j][k];
```

*Reordered loop nest:*

```
for (_____ )
  for (_____ )
    for (_____ )
      for (_____ )
        C[i][m][k] += A[k][i][m] * B[m][j][k];
```

(B) (8 points) The following three programs multiply three  $N \times N$  matrices with different loop orderings. The matrices are stored in row-major order. Each program runs on a **4-way** cache with a **total of 256 words** and a **block size of 8 words**.

**Program 1:**

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
        for (int m = 0; m < N; m++)
            for (int j = 0; j < N; j++)
                D[i][j] += A[i][k] * B[k][m] * C[m][j];
```

**Program 2:**

```
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        for (int k = 0; k < N; k++)
            for (int m = 0; m < N; m++)
                D[i][j] += A[i][k] * B[k][m] * C[m][j]
```

**Program 3:**

```
for (int k = 0; k < N; k++)
    for (int m = 0; m < N; m++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                D[i][j] += A[i][k] * B[k][m] * C[m][j];
```

Assume that data cache and instruction cache are separate. For different values of  $N$ , which program is optimal in terms of data cache misses? Circle one option.

(i)  $N = 4$

<b>Program 1</b>	<b>Program 2</b>	<b>Program 3</b>	<b>All are the same.</b>
------------------	------------------	------------------	--------------------------

(ii)  $N = 8$

<b>Program 1</b>	<b>Program 2</b>	<b>Program 3</b>	<b>All are the same.</b>
------------------	------------------	------------------	--------------------------

(iii)  $N = 16$

<b>Program 1</b>	<b>Program 2</b>	<b>Program 3</b>	<b>All are the same.</b>
------------------	------------------	------------------	--------------------------

(iv)  $N = 64$

<b>Program 1</b>	<b>Program 2</b>	<b>Program 3</b>	<b>All are the same.</b>
------------------	------------------	------------------	--------------------------



(C) (6 points) Tile the optimal program that you chose in (B)-(iv) (i.e.,  $N = 64$ ) using a tile size of  $T$  elements per dimension (i.e., each matrix should be accessed in tiles of  $T \times T$  elements each). Write the full loop nest of this tiled program. Assume  $N$  is a multiple of  $T$ .

Assuming the same cache configuration as in (B)-(iv), what is the maximum value of  $T$  that would minimize cache misses?

**Maximum value of  $T$ :** \_\_\_\_\_ **elements per dimension**

**Extra pipeline diagrams for problem 3:**

	100	101	102	103	104	105	106	107
IF								
DEC								
EXE								
MEM								
WB								

	100	101	102	103	104	105	106	107
IF								
DEC								
EXE								
MEM								
WB								