Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Computer Architecture (227-2210-00L)

ETH Zürich, Fall 2021

Prof. Onur Mutlu

Problem 1 (90 Points):	DRAM Latency	
Problem 2 (85 Points):	RowHammer	
Problem 3 (120 Points):	Processing-near-Memory	
Problem 4 (105 Points):	Memory Access Patterns	
Problem 5 (105 Points):	Emerging Memory Technologies	
Problem 6 (80 Points):	Asymmetric Multicore	
Problem 7 (115 Points):	Interconnects	
Problem 8 (BONUS: 70 Points):	Prefetching	
Total (770 (700 + 70 bonus) Points):		

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 10 single-sided (or 5 double-sided) A4 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam. Do not answer questions on them.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green, red or any other color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake. If you make assumptions, state your assumptions clearly and precisely.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You may be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

This page intentionally left blank

1 DRAM Latency [90 points]

You would like to understand the configuration of the DRAM subsystem of a computer using reverse engineering techniques. Your current knowledge of the particular DRAM subsystem is limited to the following information:

- The physical memory address is 24 bits.
- The DRAM subsystem consists of a single channel, 8 banks, and 1024 rows per bank.
- The DRAM is byte-addressable.
- The most-significant 3 bits of the physical memory address determine the bank. The following 10 bits of the physical address determine the row.
- The DRAM command bus operates at 500 MHz frequency.
- The memory controller uses the open-row DRAM row management policy.
- The memory controller uses first-ready first-come first-serve (FRFCFS) request scheduling policy. With this policy, in every clock cycle, the memory controller picks the oldest request (in the request buffer) for which the corresponding bank is ready to receive the DRAM command needed to serve the request. For example, if the oldest request needs to precharge bank X but the bank is not yet ready to be precharged (i.e., ACTIVATE-to-PRECHARGE latency is not yet satisfied), the memory controller checks the next oldest request in the request buffer. If none of the DRAM banks corresponding to the requests in the request buffer are ready, the memory controller does not issue any command in that cycle.
- The memory controller services requests in order with respect to each bank. In other words, for a given bank, the memory controller first services the oldest request in the *request buffer* that targets the same bank.

You realize that you can observe the memory requests that are waiting to be serviced in the request buffer. At a particular point in time, you take the snapshot of the request buffer and you observe the following requests in the request buffer:

decreasing age ↓

```
Read 0x660000 ← oldest
Read 0x460000
Read 0x640000
Read 0xA60000 ← youngest
```

At the same time you take the snapshot of the request buffer, you start probing the DRAM command bus. You observe the DRAM command type and the cycle (relative to the first command) in which the command is seen on the DRAM command bus. The following are the DRAM commands you observe on the DRAM bus while the requests above are serviced. Note that, although the memory controller uses the open-row policy, *a bank might be in precharged state (i.e., closed)* prior to the execution of the command sequence below if no rows are accessed in the bank after the system is powered up. Assume that all banks are ready for a DRAM command (i.e., no bank processes a previously issued command) at cycle 0.

```
Cycle 0  --- PRECHARGE
Cycle 1  --- ACTIVATE
Cycle 2  --- ACTIVATE
Cycle 4  --- ACTIVATE
Cycle 9  --- READ
Cycle 10 --- READ
Cycle 12 --- READ
Cycle 30 --- PRECHARGE
Cycle 34 --- ACTIVATE
Cycle 42 --- READ
```

Answer the following questions using the information provided above.

(a) [10 points] What is the size (in KB) of a single row? Explain.

2 KB
Explanation.
The entire address space consists of 24 bits and the most significant 3 bits identify the bank id, the next 10 bits identify the row id. So, the least significant 11 bits must be used as column id, which means the row size is $2^{11} bytes = 2KB$

(b) [30 points] What are the following DRAM timing parameters used by the memory controller, in terms of nanoseconds? If there is not enough information to infer the value of a timing parameter, write *unknown* and explain why an exact value cannot be determined. *Assume that READ-to-PRECHARGE latency is 1 cycle.*

i) ACTIVATE-to-READ latency

16 ns.

Explanation. After issuing the ACTIVATE command to bank 2 at cycle 1, the memory controller waits until cycle 9 to issue a READ command to the same bank, which indicates that the ACTIVATE-to-READ latency is 8 cycles. The command bus operates at 500 MHz, so it has 2 ns clock period. Thus, the ACTIVATE-to-READ is $8 * 2 = 16$ ns.

ii) ACTIVATE-to-PRECHARGE latency

52ns.

Explanation. The ACTIVATE command to bank 3 at cycle 4 is followed by a PRECHARGE command to the same bank at cycle 30. Thus, the ACTIVATE-to-PRECHARGE is 26 cycles, which is $26 * 2 = 52$ ns for the 500 MHz DRAM command bus.

iii) PRECHARGE-to-ACTIVATE latency

8 ns.

Explanation. The PRECHARGE-to-ACTIVATE latency can be easily seen in the two commands at cycles 30 and 34 to bank 3. The PRECHARGE-to-ACTIVATE latency is 4 cycles = 8 ns.

(c) [20 points] What is the row buffer state of the banks *prior* to the execution of any of the above requests? Fill in the table below indicating whether a bank has an open row or not. Write *unknown* in case the provided information is insufficient to determine the row buffer state.

	Open or Closed?
Bank 0	Unknown
Bank 1	Unknown
Bank 2	Closed
Bank 3	Open
Bank 4	Unknown
Bank 5	Closed
Bank 6	Unknown
Bank 7	Unknown

Explanation. By decoding the request addresses in the request buffer, we can find which bank each request targets. Looking at the commands issued for those requests, we can determine which requests needed PRECHARGE (row buffer conflict, the initially open row is unknown in this case) or ACTIVATE (the bank is initially closed).

time ↓

`0x660000` → `Bank: 3, Row: 192` (Row conflict, so bank 3 must have another row open.)
`0x460000` → `Bank: 2, Row: 192` (No PRECHARGE needed. The bank is closed.)
`0x640000` → `Bank: 3, Row: 128` (Serviced after the first request to bank 3)
`0xA60000` → `Bank: 5, Row: 192` (No PRECHARGE needed. The bank is closed.)

(d) [30 points] To improve performance, you decide to implement the idea of *Tiered-Latency DRAM (TL-DRAM)* in the DRAM chip. Assume that a bank consists of a single subarray. With TL-DRAM, an entire bank is divided into a near segment and far segment. When accessing a row in the near segment, the ACTIVATE-to-READ latency *reduces* by 2 cycles and the ACTIVATE-to-PRECHARGE latency reduces by 5 cycles. When precharging a row in the near segment, the PRECHARGE-to-ACTIVATE latency reduces by 1 cycle. When accessing a row in the far segment, the ACTIVATE-to-READ latency *increases* by 1 cycle and the ACTIVATE-to-PRECHARGE latency increases by 2 cycles. When precharging a row in the far segment, the PRECHARGE-to-ACTIVATE latency increases by 2 cycles. The following table summarizes the changes in the affected latency parameters.

Timing Parameter	**Near Segment Latency**	**Far Segment Latency**
ACTIVATE-to-READ	-2	$+1$
ACTIVATE-to-PRECHARGE	-5	$+2$
PRECHARGE-to-ACTIVATE	-1	$+2$

Assume that the rows in the near segment have smaller row ids compared to the rows in the far segment. In other words, physical memory row addresses 0 through $N - 1$ are the near-segment rows, and physical memory row addresses N through 1023 are the far-segment rows.

If the above DRAM commands are issued 6 cycles faster with TL-DRAM compared to the baseline (the last command is issued in cycle 36), how many rows are in the near segment, i.e., what is N? Show your work.

The rows in the range of [0-192] should definitely be in the near segment because all requests in the request buffer perform low-latency DRAM access. The row open in bank 3 prior to servicing the 4 requests in the request buffer must be in the far segment but we do not know which row was open. Thus, N is a number larger than 192 but we cannot determine the exact value.

Explanation. Below, we show the command trace that results in 6 cycle reduction compared to the previous command trace. All 4 requests in the request buffer must access rows in the near segment. The unknown row open in bank 3 prior to servicing the request must be in the far segment. Otherwise, the total cycles will reduce by more than 6.

```
Cycle 0 -- PRECHARGE - Bank 3, an unknown row in the far segment
Cycle 1 -- ACTIVATE - Bank 2, row 192, which is in the near
segment
Cycle 2 -- ACTIVATE - Bank 5, row 192, which is in the near
segment
Cycle 6 -- ACTIVATE - Bank 3, row 192, which is in the near
segment
Cycle 7 -- READ - Bank 2
Cycle 8 -- READ - Bank 5
Cycle 12 -- READ - Bank 3
Cycle 27 -- PRECHARGE - Bank 3, precharging row 192 in the near
segment
Cycle 30 -- ACTIVATE - Bank 3, row 128, which is in the near
segment
Cycle 36 -- READ - Bank 3
```

2 RowHammer [85 points]

2.1 RowHammer Attacks

In order to characterize the vulnerability of your DRAM device to RowHammer attacks, you must be able to induce RowHammer bit flips. Assume the following about the target system:

- The CPU has a single-core in-order processor, and does not implement virtual memory.
- The physical memory address is 16 bits.
- The DRAM subsystem consists of two channels, four banks per channel, and 64 rows per bank.
- The memory controller employs the open-row policy.
- The memory controller does not employ any remapping or scrambling schemes for the physical address.
- There is no RowHammer mitigation mechanism implemented in any part of the system.
- All the cells in the DRAM subsystem are equally vulnerable to RowHammer-induced bit flips.

You implement several programs using the instructions shown in Table 1.

Instruction	**Description**	**Functionality**
`B LABEL`	Unconditional Branch	`PC = LABEL`
`STORE IMM, Rs`	Store word to memory	`MEM[IMM] = Rs`
`CLFLUSH IMM`	Cache line flush	`Flush cache line containing IMM`

Table 1: Instruction Descriptions.

(a) [20 points] You run Program 1 below, but you cannot observe any bit flips in the target system. You figured out that the number of activations is much lower than your expectation. Give **two** potential reasons that might cause Program 1 to fail to introduce a sufficient number of row activations to induce RowHammer bit flips.

Program 1

```
LOOP:
  STORE 0x8732, R0
  STORE 0x8734, R0
  CLFLUSH 0x8732
  CLFLUSH 0x8734
  B LOOP
```

1. These two addresses are mapped to different banks. Due to the open-row policy, neither row is precharged, hence they experience only one (or few) activations.
2. The system does not allow executing CLFLUSH instructions at the user level. Therefore, STORE instructions are serviced in the cache, causing no main memory requests.
3. These two addresses are mapped to the same DRAM row. Therefore, there are no row buffer conflicts.
4. Since this loop can send many write requests in a short amount of time, most of the write requests hit the write queue in the memory controller. Hence only few activations are performed to these rows.

(b) [30 points] You try Programs 2a, 2b, and 2c, but find that *only two of them can induce RowHammer bit flips* in your DRAM subsystem. Which program segment is the one that does *not* induce

RowHammer bit flips? Justify your answer.

Program 2a

```
LOOP:
  STORE 0x8732, R0
  STORE 0x98CD, R1
  CLFLUSH 0x8732
  CLFLUSH 0x98CD
  B LOOP
```

Program 2b

```
LOOP:
  STORE 0xF1AB, R0
  STORE 0x0054, R1
  CLFLUSH 0xF1AB
  CLFLUSH 0x0054
  B LOOP
```

Program 2c

```
LOOP:
  STORE 0x2B97, R0
  STORE 0xDA68, R1
  CLFLUSH 0x2B97
  CLFLUSH 0xDA68
  B LOOP
```

1. In order to introduce enough activations, two `STORE` instructions should access *different rows in the same bank*.
2. Three program segments are identical except for the memory addresses, so we can assume that only two program segments have two `STORE` instructions whose destination addresses are assigned to the same bank (but different rows).
3. Since the DRAM subsystem 1) consists of 8 banks and 2) employs no address remapping/scrambling schemes, two addresses assigned the same bank should satisfy a condition C: they have at least three same bit values at the same position.

Two addresses in each program are:

- **Program 2a**
  `1000 0111 0011 0010 (0x8732)`
  `1001 1000 1100 1101 (0x98CD)`
- **Program 2b**
  `1111 0001 1010 1011 (0xF1AB)`
  `0000 0000 0101 0100 (0x0054)`
- **Program 2c**
  `0010 1011 1001 0111 (0x2B97)`
  `1101 1010 0110 1000 (0xDA68)`

We can observe

1. Two paired addresses in every program segment have only three same bit values at the same position, i.e., satisfy C.
2. The position of the same bit values in Program 2b and Program 2c is the same, but different from Program 1. Therefore, if Program 2b can induce RowHammer bit flips, Program 2c should also be able to induce bit flips and Program 2a should not. On the other hand, if Program 2a can induce RowHammer bit flips, neither Program 2b nor Program 2c can induce bit flips.

Since only one program segment can *not* induce RowHammer bit flips, Program 2a is the one that can *not* induce RowHammer bit flips.

2.2 RowHammer Defenses

You are given a system that crashes frequently. After a thorough investigation, you notice that this crash happens when a DRAM row (say Row A) is activated many times. The evidence suggests that this failure is caused by RowHammer bit flips. Therefore, you analyze and collect the following information about the RowHammer vulnerability of the DRAM chip, the RowHammer defense mechanism, and the memory access pattern.

RowHammer vulnerability of the DRAM chip. A row needs to be activated at least 20000 times to induce a bit flip in its physically adjacent rows. The *blast radius* is one, meaning that hammering a DRAM row induces bit flips only in its physically adjacent rows.

RowHammer defense mechanism. The RowHammer defense mechanism is implemented in the memory controller. The defense mechanism deterministically and precisely counts the activation count of each row. When the activation count of a Row X reaches 4096, the defense mechanism sends row activation commands to row addresses X-1 and X+1.

Memory access pattern. The access pattern that causes RowHammer bit flips activates Row A 30000 times within a 32ms time window. During these activations, the defense mechanism activates rows A-1 and A+1 once every 4096 activations targeting row A.

(a) [15 points] Even though the RowHammer defense mechanism seems to refresh rows A-1 and A+1, you still observe RowHammer bit flips. Why is that? Explain.

This happens because the rows A-1 and A+1 are *not* mapped to the physically adjacent rows of row A. Even though the RowHammer defense mechanism in the memory controller refreshes A-1 and A+1, due to the DRAM-internal row address mapping, the memory controller does not refresh the rows that are physically adjacent to Row A. The system crashes because of the data corruption in the DRAM rows that are physically adjacent to the aggressor row (Row A).

(b) [10 points] Would implementing Probabilistic Row Activation (PARA), as proposed in the original RowHammer paper from ISCA 2014, in the memory controller instead of the explained defense mechanism solve the problem? Explain your reasoning.

No. PARA would also refresh the wrong set of DRAM rows.

(c) [10 points] Would implementing BlockHammer, as discussed in Lecture 16c, in the memory controller instead of the explained defense mechanism solve the problem? Explain your reasoning.

Yes. BlockHammer would stop the activations of targeting Row A before it is activated for 20000 times. Therefore, it would avoid experiencing a bit flip.

3 Processing-near-Memory [120 points]

You want to accelerate the following two pieces of code from Application 1 (App1) and Application 2 (App2).

```
// App1. Registers are 4-byte wide
movi R1, #0x1000            // Store the base address of A in R1
movi R2, #0x8000            // Store the base address of B in R2
movi R3, #0

Loop:
    ld R4, [R1, R3]          // R4 = MEM[R1 + R3]
    ld R5, [R2, R3]          // R5 = MEM[R2 + R3]
    mult R6, R4, #0xF        // R6 = R4 * 0xF
    add R6, R6, R5           // R6 = R6 + R5
    st [R1, R3], R6          // MEM[R1 + R3] = R6

    inc R3                   // R3++
    bne R3, 1000000000, Loop // If R3 != 1000000000, jump to Loop
```

```
// App2. Registers are 4-byte wide
movi R1, #0x1000 // Store the base address of A in R1
movi R2, #0x8000 // Store the base address of B in R2
movi R3, #0
movi R4, #0

Loop:
    ld R5, [R1, R3]          // R5 = MEM[R1 + R3]
    ld R6, [R1, R3, #4]      // R6 = MEM[R1 + R3 + #4]
    sub R5, R5, R6           // R5 = R5 - R6
    mult R5, R5, R5          // R5 = R5 * R5
    add  R4, R4, R5          // R4 = R4 + R5
    sqrt R4, R4              // R4 = sqrt(R4)
    st [R2, R3], R4          // MEM[R2 + R3] = R4

    inc R3, #2               // R3=R3+2
    bne R3, 1000000000, Loop // If R3 != 1000000000, jump to Loop
```

We make the following assumptions about the baseline CPU where both applications run:

- The CPU is a single-issue in-order processor and all load/store operations are serialized, i.e., the latency of multiple memory requests cannot be overlapped.
- The clock frequency of the CPU is 1 GHz.
- Each memory operation (i.e., `ld`, `st`) takes 100 ns.
- Each simple arithmetic operation (i.e., `add`, `sub`, `mult`, `inc`) and branch operation (i.e., `bne`) takes 1 clock cycle to execute.
- A complex arithmetic operation (i.e., `sqrt`) takes as long as 50 simple arithmetic operations.
- All memory operations (i.e., `ld`, `st`) and arithmetic operations (i.e., `add`, `sub`, `mult`, `inc`, `sqrt`) operate on 4 bytes of data.

(a) [30 points] What is the execution time of App1 and App2 when running on the baseline CPU? Show your work. Hint: Do *not* account for the execution time of the initial `movi` instructions, since it is negligible in comparison to the loops.

App1:

For App1: Execution Time = 3040×10^8 ns.

Explanation:
App1 executes three arithmetic operations (1 ns each), three load/store accesses (100 ns each), and one branch operation (1 ns). The loop repeats $1000000000 = 10^9$ times. Therefore:
$ExecutionTime = 10^9 \times (3 \times 100 + 4 \times 1)$
$ExecutionTime = 304 \times 10^9$ ns

App2:

For App2: Execution Time = 1775×10^8 ns.

Explanation:
App2 executes four arithmetic operations (1 ns each), one square root operation (50 ns), three load/store accesses (100 ns each), and one branch operation (1 ns). The loop repeats $\frac{1000000000}{2}$ times. The `sqrt` is equivalent to 50 operations and, thus, has a latency of 50 ns. Therefore:
$ExecutionTime = 5 \times 10^8 \times (3 \times 100 + 5 \times 1 + 1 \times 50)$
$ExecutionTime = 1775 \times 10^8$ ns

You have learned in class that Processing-near-Memory (PnM) architectures can accelerate memory-bound workloads, since they provide higher bandwidth and lower latency than conventional processor-centric architectures. You decide to design PnM accelerators for App1 and App2.

The memory device you use is an early-generation 3D-stacked memory with an internal memory bandwidth of only 40 GB/s (i.e., twice the bandwidth of a single-channel 2D DDR4 memory). The 3D-stacked memory allows you to embed compute resources at the base die of the memory cube, called *logic layer*. However, the logic layer imposes several design limitations:

- The area available to build your accelerator in the logic layer of the 3D-stacked memory is **100** mm^2.
- The maximum clock frequency of the accelerator in the logic layer of the 3D-stacked memory is $\frac{1}{10}\times$ that of baseline CPU.
- The accelerator consists of one or more *processing elements*, each of which contains several *functional units*. Table 1 shows the functional units available to build your accelerators.

- A processing element of the accelerator executes the same computation as an entire iteration of the loop (i.e., *all* the instructions in the loop body). The processing element executes an iteration completely before moving to the next iteration. Therefore, if there are N processing elements, N iterations of the loop are executed in parallel.
- The loop iterations are executed on the processing elements as decided by a compiler, which unrolls the loop and preassigns iterations to the processing elements (i.e., the compiler schedules the iterations statically).
- Each functional unit of a processing element of the accelerator can execute one instruction of the loop body at a time. The same functional unit can execute different instructions (e.g., an arithmetic unit is capable of executing `add`, `sub`, `mult`, `inc`) at different times. Two functional units can execute in parallel, if there is no dependence between the operands.

Table 1: Functional units available to build your accelerators.

Functional Unit	**Description**	**Latency (cycles)**	**Area (mm^2)**
Arithmetic Unit	Arithmetic unit capable of executing `add`, `sub`, `mult`, `inc`	1	1
Load and Store Unit	Load and store unit. It can issue 1 4-byte `ld`/`st` operation per cycle	1	1
Branch Unit	Executes conditional branches (`bne`) with 100% accuracy	1	1
Square Root Unit	Executes square root (`sqrt`) operations	70	30

You design your accelerator with the maximum possible number of processing elements, in order to be able to run in parallel as many loop iterations as possible. A processing element can have as many functional units as possible, in order to extract from the code as much Instruction Level Parallelism (ILP) as possible.

(b) [30 points] What is the area of the configuration of your PnM accelerator that provides the highest performance for each application while fitting in the PnM area budget? Show your work.

App1:

For App1: Area = 100 mm^2.

Explanation:
For App 1: We can have 25 processing elements within the given area budget. A single iteration of the loop requires 4 functional units (two load/store units, one arithmetic unit, and one branch unit). Note that 3 load/store units would not improve performance, since `st` cannot be executed in parallel with the `ld` instructions. The same applies to `mult` and `add`. Therefore:
$Area = 4 \times 25 = 100\ mm^2$.

Note: An alternative solution considers only one load/store unit, in order to fit more processing elements in the available area (thus, 33 processing elements). Note that the solution of the remaining sections would change.

App2:

For App2: Area = 68 mm^2.

Explanation:
For App 2: We can have 2 processing elements in the given area budget. A single iteration of the loop requires 4 functional units (two load/store units, one arithmetic unit, and one branch unit) and 1 `sqrt` functional unit. Therefore:
$Area = (4 + 30) \times 2 = 68\ mm^2$.

Note: An alternative solution considers only one load/store unit, in order to fit more processing elements in the available area (thus, 3 processing elements). Note that the solution of the remaining sections would change.

(c) [30 points] Are the PnM accelerators obtained in (b) capable of fully utilizing the memory bandwidth that the 3D-stacked memory provides for App1 and for App2? Show your work.

App1:

For App1: No.

Explanation:
The bandwidth of the 3D-stacked memory is 40 GB/s. Therefore, a PnM accelerator can load 40 bytes in 1 ns or 400 bytes in 10 ns (i.e., in a clock cycle of the accelerator).

App1 has two independent loads, each is 4 bytes. Thus, to fully utilize the memory bandwidth, we need $\frac{400}{8} = 50$ processing elements (i.e., 100 load/store units). Due to the area budget, we can fit 25 processing elements (50 load/store units) in the accelerator for App1. Thus, half of the bandwidth is utilized.

App2:

For App2: No.

Explanation:
App2 has two independent loads, each is 4 bytes. Thus, to fully utilize the memory bandwidth, we need $\frac{400}{8} = 50$ processing elements (i.e., 100 load/store units). Due to the area budget, we can fit only 2 processing elements (4 load/store units) in the accelerator for App2.

(d) [30 points] What is the speedup of the PnM accelerators compared to the execution of App1 and App2 on the baseline CPU? Show your work.

App1:

For App1: Speedup = 126.7×

Explanation:
The number of cycles executed by the PnM accelerator for App 1 is:
$Cycles = \frac{10^9}{25} \times 1 \times 6 = 24 \times 10^7$ cycles.
Since the clock frequency for the PnM accelerator is 10× smaller than the baseline, a clock cycle will be 10× longer. Thus, the execution time of the accelerator is:
$ExecutionTime = 24 \times 10^7 \times 1 \times 10 = 24 \times 10^8$ ns.
The execution time on the baseline CPU is (from part (a)):
$ExecutionTimeBaseline = 3040 \times 10^8$ ns.
As a result, the speedup provided by the accelerator is: $Speedup = \frac{3040 \times 10^8}{24 \times 10^8} = \frac{3040}{24} =$ 126.7×

App2:

For App2: Speedup = 0.92×

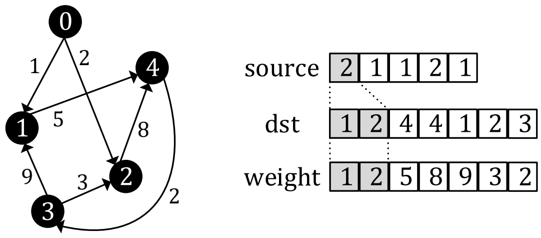
Explanation:
The number of cycles executed by the PnM accelerator for App 2 is:
$Cycles = \frac{5 \times 10^8}{2} \times (1 \times 7 + 70 \times 1) = 1925 \times 10^7$ cycles.
$ExecutionTime = 1925 \times 10^7 \times 1 \times 10 = 1925 \times 10^8$ ns.
The execution time on the baseline CPU is (from part (a)):
$ExecutionTimeBaseline = 1775 \times 10^8$ ns.
As a result, the speedup provided by the accelerator is: $Speedup = \frac{1775 \times 10^8}{1925 \times 10^8} = \frac{1775}{1925} = 0.92\times$

4 Memory Access Patterns & Interference [105 points]

Consider a graph that consists of V vertices and E edges and is represented using a compression format (CF). CF consists of 3 arrays:

- `source`: holds the number of edges (neighbors) of vertex i at position i.
- `dst`: holds the destination (neighbors) vertices' ids for each source vertex.
- `weight`: holds the weight of the edge connecting the source vertices and the destination vertices.

Figure 1 demonstrates an example graph (V=5, E=7) represented using the CF format.
The vertex with ID = 0 has 2 neighbors: destination vertices 1 and 2.



source 2 1 1 2 1

dst 1 2 4 4 1 2 3

weight 1 2 5 8 9 3 2

Figure 1: Graph with V=5 vertices and E=7 edges

Consider App 1, a graph analytics application that (i) visits every vertex of the graph, (ii) traverses its neighbors and (iii) calculates a property of the neighbor vertex.
All the data structures hold 4-byte integer values.

Graph analytics application (App 1)

```
old_rank = malloc(sizeof(int)*graph.vertices); // Data Structures of App 1
rank = malloc(sizeof(int)*graph.vertices);
dst = malloc(sizeof(int)*graph.edges);
weight = malloc(sizeof(int)*graph.edges);
source = malloc(sizeof(int)*graph.vertices);

initialize_data_structures();

int edge = 0;
for( i=0; i < graph.vertices; i++ )
    neighbors = source[i];
    for( j = 0; j < neighbors; j++ )
        rank[dst[edge]] += weight[edge] * old_rank[i];
        edge += 1;
```

Consider a system with the following configuration:

- The system consists of a single-core CPU and a single DRAM channel with 4 banks.
- All memory requests from the CPU are served by the DRAM. There are no caches in the system.
- Memory requests are serialized, i.e., memory requests do *not* overlap and CPU stalls until data arrives.
- The processor requests data from DRAM **only** to access the 5 data structures used in App 1.
- **No memory access reordering** is performed by the processor. For example, during the first iteration (i.e. $i = 0$, $j = 0$, $edge = 0$) the processor accesses memory in the following order: `source[0]`, `weight[0]`, `old_rank[0]`, `dst[0]`, `rank[dst[0]]` (assume a single request for reading and writing to rank), `source[1]`, etc.
- All the data structures are aligned at the start of a bank row.

- All the data structures are mapped in consecutive memory locations in each bank. For example, given a 4KB DRAM row size, rank[0]-rank[1023] is stored in Row 0, rank[1024-2047] in Row 1, etc.
- Latency of activating a row: 100 cycles.
- Latency of reading data from an activated row: 16 cycles.
- Latency of writing data to an activated row: 20 cycles.
- Latency of precharging a row: 80 cycles.
- DRAM Bus width: 4 bytes.
- For every request to DRAM, the processor receives 4 bytes.
- Latency of transferring 4 bytes: 4 cycles.
- Row buffer size: 2KB; Bank size: 64MB.
- Open-page row buffer policy.
- First come - first served scheduling policy.
- All the DRAM banks are initially precharged.

(a) [25 points] Identify the memory access pattern (Random/Irregular, Strided (Stride > 1), Stream (Stride $= 1$)) of each data structure used in the application. Reason about your answer.

source: Stream

dst: Stream

weight: Stream

rank: Random/Irregular

old_rank: Stream

(b) [30 points] Consider a case where data structures **rank** and **dst** are mapped in Bank 0, **weight** is mapped in Bank 1, **old_rank** is mapped in Bank 2 and **source** is mapped in Bank 3. Calculate the number of row buffer conflicts after executing App 1 (Lines 9-14) using an input graph with V=2^{19} and E=2^{22}. Show your work.

rank and **dst** are mapped in the same bank and are accessed interchangeably. Hence, a row conflict occurs for every load/store to **rank and dst**, which are accessed E=2^{22} times each.
weight, old_rank and source invoke row buffer conflicts only when the CPU reads an entire row. **weight, old_rank** are accessed once for every neighbor, E=2^{22} times in total each one of them. **source** is accessed once per vertex, V=2^{19} times.
The first access at each bank leads to a row miss not a row conflict.
Total row buffer conflicts: $2^{22} + 2^{22} + 2^{22}/2^9 + 2^{22}/2^9 + 2^{19}/2^9 - 4$

(c) [30 points] Calculate the number of cycles that the processor stalls while waiting for data from memory when executing App 1 using a new input graph with V=2^{19} and E=2^{22} that leads to exactly 2^{20} row buffer conflicts. Show your work.

Total number of reads/writes: 2^{22} for **rank** (Write), 2^{22} for **dst**, 2^{22} for **weight**, 2^{22} for **old_rank**, 2^{19} for **source**.
In the case of row conflict the row needs to be precharged (80 cycles) and activated (100 cycles) since we have open-row policy.
In the case of a row miss, the row needs to be activated (100 cycles).
In case of reads/writes, 16/20 cycles are spent to read data from the row buffer and 4 to send it to the processor/DRAM.

Formula: $Row\_Conflicts * (80 + 100) + (Row\_Misses) * 100 + Reads * (16 + 4) + Writes * (20 + 4)$

Final answer: $2^{20} * (80+100) + 4 * 100 + (2^{22} + 2^{22} + 2^{22} + 2^{19}) * (16+4) + (2^{22}) * (20+4)$

(d) [20 points] We replace the DRAM of the previously used system with 2 different DRAM devices, N-DRAM and RL-DRAM.

- Each one of these devices is connected to the processor using a separate channel.
- N-DRAM has the same characteristics as the DRAM described in the previous questions.
- RL-DRAM operates with 0.5x the internal latency (i.e. $Precharge, Activate, Read/Write$ latencies are halved) of N-DRAM and consists of **only** 1 bank.

Given this system, how would you place the data structures of App 1 across N-DRAM and RL-DRAM to maximize peformance? Give your reasoning.

Randomly/Irregularly accessed data structure **rank** should be mapped to RL-DRAM. This choice stems from the fact that the spatial and temporal locality will be low and the row conflict rate will be high. Hence, reducing the row conflict latency will reduce the latency of accessing **rank**. Stream data structures **weight, source, dst, old_rank** should be mapped to N-DRAM. N-DRAM has 4 banks and the 4 data structures can be mapped to each one of them.

5 Emerging Memory Technologies [105 points]

5.1 Non-Volatile Memory (NVM) [15 points]

Indicate whether each of the following statements is true or false. *Note: we will subtract 1.5 points for each **incorrect** answer. (The minimum score you can get for this question is 0 points.)*

(a) [3 points] Data is written into Phase Change Memory (PCM) by injecting current to change the magnetic polarity of phase change material.

1. `True` [✓ 2. `False`]

(b) [3 points] PCM can be denser than DRAM while DRAM is faster and more durable than PCM. Hence, there is an opportunity to benefit from both DRAM and PCM by building a hybrid DRAM-PCM memory system.

[✓ 1. `True`] 2. `False`

(c) [3 points] Multi-Level Cell (MLC) NVM has lower latency and energy consumption than Single-Level Cell (SLC) NVM.

1. `True` [✓ 2. `False`]

(d) [3 points] NVM has lower endurance than DRAM because writes to NVM take much longer.

1. `True` [✓ 2. `False`]

(e) [3 points] It takes the same energy to write "00" as to write "11" into an MLC PCM cell.

1. `True` [✓ 2. `False`]

5.2 PCM-based Main Memory [60 points]

A student at ETH wants to build a computer system using PCM as the main memory. Since PCM has limited endurance (i.e., a memory cell fails after 10^7 writes are performed to the cell), the student designs a perfect wear-leveling mechanism (i.e., a mechanism that equally distributes writes across all of memory cells).

The student wants to estimate the worst-case lifetime of PCM when used as main memory. Therefore, she executes a test program to wear out the entire PCM *as quickly as possible*. The test program runs special instructions to bypass the cache hierarchy and repeatedly writes data into different pages until *all* the PCM cells are worn out. The student's measurements show that PCM stops functioning (i.e., all its cells are worn-out) in 3 years. Assume the following:

- The processor employs in-order instruction execution.
- There is no memory-level parallelism (i.e., there is a single bank in the memory system).
- It takes 8 ns to send a memory request from the processor to the memory controller.
- It takes 13 ns to send the request from the memory controller to PCM.
- The write latency of PCM is 43 ns.
- Each write request is fully serialized, i.e., there are three steps of write requests: (1) memory request from CPU to controller, (2) write request from controller to PCM, and (3) data write to PCM cells. None of the steps can be pipelined.
- PCM requests (read or write) are performed at 64-byte granularity.
- PCM adopts a quad-level cell (QLC) technique that stores four bits in a single memory cell.

(a) [30 points] What is the capacity of the PCM the student uses? Show your work. *Hint:* 3 years ≈ 10^{17} ns.

10 GB

Explanation:

- Each memory cell should serve 10^7 writes.
- Since PCM performs writes at 64-byte granularity (i.e., 2^9-bit), the required number of writes is equal to $\frac{capacity}{2^9} \times 10^7$.
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to 8+13+43 = 64 ns.

$$\therefore \frac{capacity}{2^9} \times 10^7 \times 64 = 10^{17}$$

$$capacity = \frac{2^9 \times 10^{17}}{2^6 \times 10^7}$$

$$= 8 \times 10^{10}$$

(b) [30 points] The student decides to improve the lifetime of PCM cells by using the single-level cell (SLC) mode in which a single memory cell stores a single bit. When PCM operates in the SLC mode, each cell's endurance increases by a factor of 10 while the write latency of PCM decreases (compared to the QLC mode). To measure the lifetime of PCM in the SLC mode, the student repeats the same experiment performed in part (a) while keeping everything else in the system the same. The result shows that the lifetime of PCM increases by 2× in SLC mode compared to the QLC mode. Assume the capacity of PCM in QLC mode is 8 GB. What is the write latency of SLC PCM? Show your work.

43 ns

Explanation:

- Each memory cell should receive $10 \times 10^7 = 10^8$ writes.
- The memory capacity is reduced by 4× in the SLC mode compared to the QLC mode.
  $\therefore C_{\text{SLC}} = \frac{C_{\text{QLC}}}{2^2}$.
- The required number of writes to wear out the entire PCM in the SLC mode is equal to $\frac{C_{\text{SLC}}}{2^{11}}$.
- The total latency of each memory access in the SLC mode is equal to $(8 + 13 + write\_latency\_SLC)$.
- The lifetime in the SLC mode is 6 years $= 2 \times 10^{17}$ (double the lifetime in the QLC mode).

$$\therefore \frac{C_{\text{QLC}}}{2^{11}} \times 10^8 \times (8 + 13 + write\_latency\_SLC) = 2 \times 10^{17}$$

$$write\_latency\_SLC = \frac{2^{12} \times 10^9}{8 \times 10^9 \times 8} - 21$$

5.3 Processing using Non-Volatile Memory [30 points]

In class, we learned that a resistive-RAM (ReRAM) crossbar array can significantly accelerate vector-matrix multiplications (VMM). Figure 1 shows the architecture of a 4×4 ReRAM crossbar array designed to perform the following vector-matrix multiplication:

$$I(i_1, i_2, i_3, i_4) \times W \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{pmatrix} = O(o_1, o_2, o_3, o_4).$$

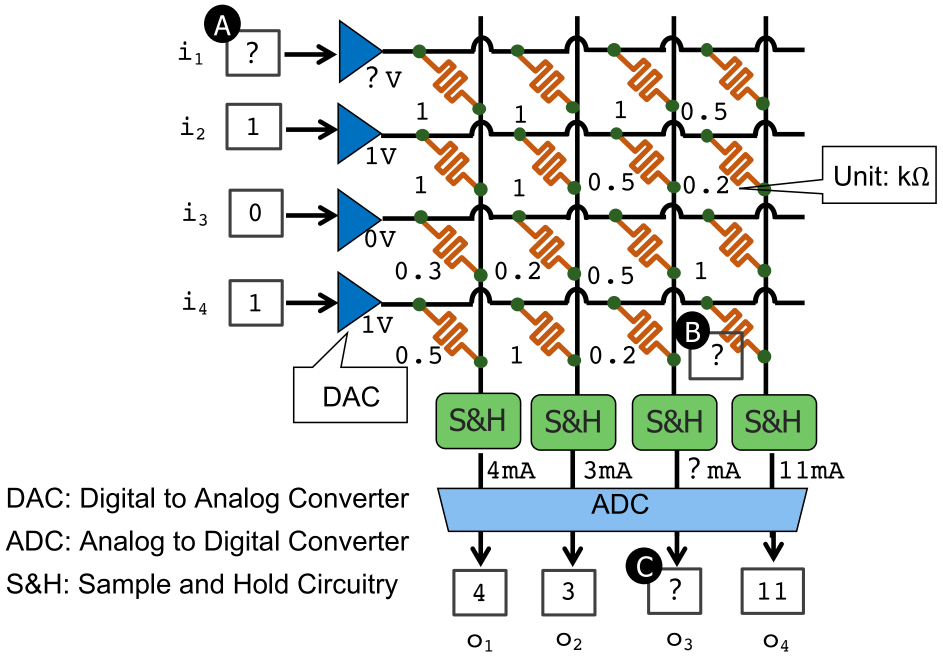


Figure 1: A 4×4 Computation-Capable ReRAM Crossbar Array.

Recall that a typical crossbar array performs a VMM operation based on Kirchhoff's Law. The VMM computation is conducted via the following four steps.

Step 1: We store the matrix into the 2D array of ReRAM cells. Resistance of each ReRAM cell represents the corresponding value in the matrix.

Step 2: A DAC (Digital-to-Analog Converter) converts a digital input number to a voltage and applies the voltage to the corresponding wordline.

Step 3: A S&H (Sample and Hold) unit collects the current that comes through the corresponding bitline.

Step 4: An ADC (Analog-to-Digital Converter) converts the current to the digital number (i.e., an output).

Please identify the unknown values in the three boxes in Figure 1. For your convenience, each number in Figure 1 is converted to a decimal number. Show your work.

- A: 1 (A = (4 - 1/0.5 - 1/1)/1 = 1)
- B: 0.25 (B = 1/(11 - 1/0.2 - 1/0.5) = 0.25)
- C: 8 (C = 1/1 + 1/0.5 + 1/0.2 = 8)

6 Asymmetric Multicore [80 points]

A microprocessor manufacturer asks you to design a multicore processor for modern workloads. You should optimize it assuming a workload with 60% of its work in the parallel portion and 40% in the serial portion. You are tasked to compare two configurations that can fit into the processor's die area: 1) Small Cores (SC), and 2) Large + Small Cores (LSC). These consist of the following:

- *SC*: A design that contains 8 small cores, which share the same die. Seven of these small cores operate at a *baseline* fixed instruction throughput. The eighth small core is an overclockable core, which can operate at either: 1) *baseline* throughput, or 2) *overclocked* with 2× the baseline throughput.
- *LSC*: A design that contains 1 large core and 4 small cores that all share the same die. The four small cores operate at the baseline throughput. The large core is 4× faster than a small core.

In addition, Table 1 provides the static power (i.e., when the core is idle) and the dynamic power (i.e., when the core is active) of each of the cores.

Table 1: Power consumption of cores in different modes.

Core	**Mode**	**Static Power (W)**	**Dynamic Power (W)**
Small	Baseline	0.5	1
	Overclocked	1	8
Large	Baseline	2	4

The SC processor executes the parallel portion on *all* the small cores (including the overclockable core, operating at the baseline throughput), and the serial portion *only* on the overclockable core (using either baseline or overclocked options). The LSC processor executes the parallel portion only on the small cores, and the serial portion only on the large core.

Please answer the following questions.

(a) [20 points] Which of the three design configurations (SC, SC with overclocked core, or LSC) results in the highest performance? Show your work.

The LSC configuration is the best option.

Explanation:
The best-case speedup for the SC configuration when running using the baseline throughput can be calculated as:
$Speedup_1 = \frac{1}{\frac{0.4}{1}+\frac{0.6}{8}} = \frac{40}{19}$.

The best-case speedup for the SC configuration when running using the overclocked throughput can be calculated as:
$Speedup_2 = \frac{1}{\frac{0.4}{2}+\frac{0.6}{8}} = \frac{40}{11}$.

The best-case speedup for the LSC configuration can be calculated as:
$Speedup_3 = \frac{1}{\frac{0.4}{4}+\frac{0.6}{8}} = \frac{40}{10}$.

(b) [20 points] The energy consumption should also be a metric of reference in your design. Which of the three design configurations (SC, SC with overclocked core, or LSC) results in the lowest energy consumption? Show your work.

The LSC configuration is the best option.

Explanation:
We can calculate the energy consumption as:
$E_{total} = E_{serial} + E_{parallel} = P_{serial} \times t_{serial} + P_{parallel} \times t_{parallel}$

The energy consumption for the SC configuration can be calculated as:
$E_{total1} = (1 + 7 \times 0.5) \times 0.4 + (1 \times 8) \times \frac{3}{40} = 2.4.$

The energy consumption for the SC configuration with overclocked core can be calculated as:
$E_{total2} = (8 + 7 \times 0.5) \times 0.2 + (1 \times 8) \times \frac{3}{40} = 2.9.$

The energy consumption for the LSC configuration can be calculated as:
$E_{total3} = (4 + 4 \times 0.5) \times 0.1 + (2 + 1 \times 4) \times \frac{6}{40} = 1.5.$

(c) [20 points] At least what ratio of a workload should be spent on the parallel section so that the *SC* configuration, *even without overclocking,* performs better than the *LSC* configuration? Show your work.

More than $\frac{6}{7}$.

Explanation:

$\frac{1}{(1-P)+\frac{P}{8}} > \frac{1}{\frac{(1-P)}{4}+\frac{P}{4}}$

$(1-P) + \frac{P}{8} < \frac{(1-P)}{4} + \frac{P}{4}$

$\frac{8-8p+P}{8} < \frac{1}{4}$

$8 - 7P < 2$

$P > \frac{6}{7}.$

(d) [20 points] In order to improve the performance of the LSC configuration, you come up with hardware design optimizations to improve the throughput of the large core. You expect that these optimizations will increase the throughput of the large core by $T\times$. Given an application with 90% of its work in the parallel portion, is it possible for the LSC configuration to outperform the SC configuration *with* overclocking? If yes, for which values of T? Show your work.

No, it is not possible.

Explanation:

In order for the LSC configuration to perform better than the SC configuration with overclocked core, the following equation needs to hold true for a non-empty set of positive values of T:

$$\frac{1}{\frac{1}{10\times2}+\frac{9}{10\times8}} < \frac{1}{\frac{1}{10\times4\times T}+\frac{9}{10\times4}}$$

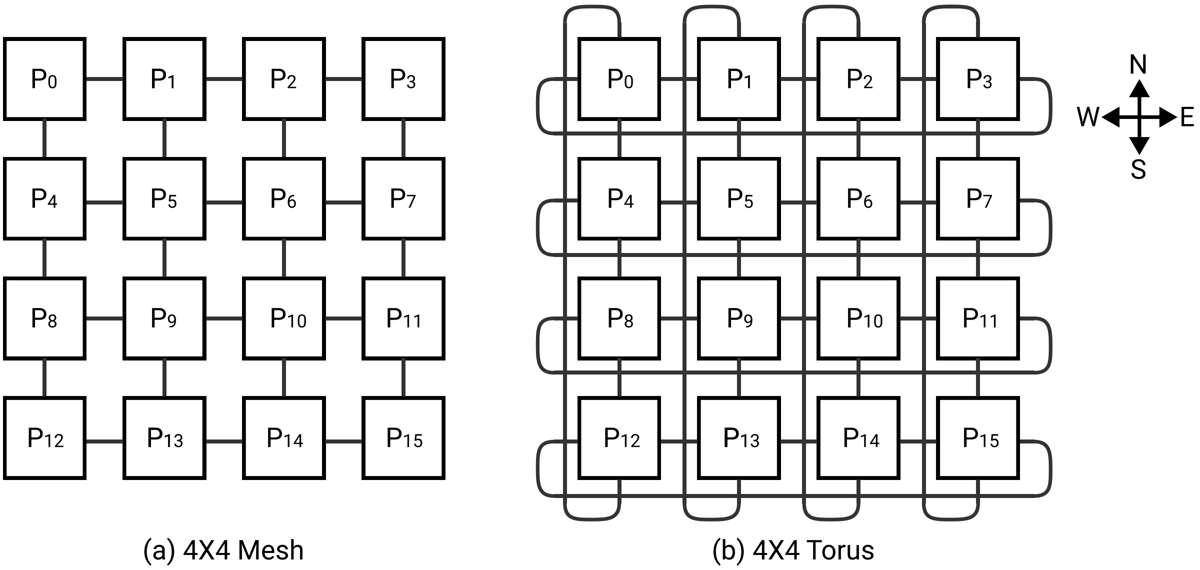
$$\frac{1}{10\times2}+\frac{9}{10\times8} > \frac{1}{10\times4\times T}+\frac{9}{10\times4}$$

$$13 > \frac{2}{T}+18$$

Since the resulting equation cannot hold true for any positive value of T, we conclude that it is not possible to improve the performance of the large core such that the LSC configuration outperforms the SC configuration for this workload.

7 Interconnection Networks [115 points]

Suppose you want to connect 16 processors using two topologies: 4 × 4 Mesh and 4 × 4 Torus with bi-directional links, similar to Figure 1:



(a) 4X4 Mesh (b) 4X4 Torus

Figure 1: 2D mesh and 2D torus topologies connecting 16 processors.

You know that there are eight possible turns in the 2D mesh and the 2D torus topologies, as shown in Figure 2:

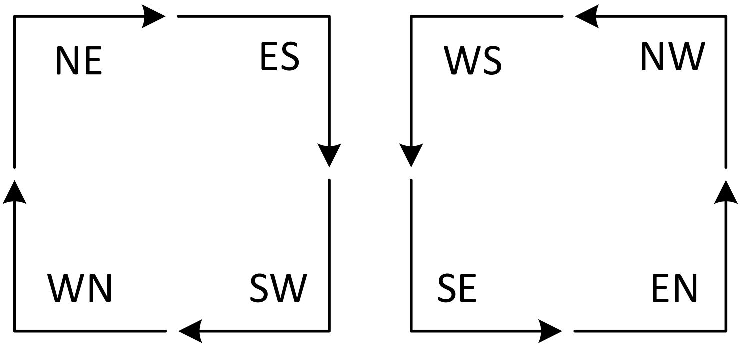


Figure 2: Eight possible turns in 2D mesh and 2D torus topologies.

You are considering three different *minimal* routing algorithms based on which turns are forbidden:

- **Routing Algorithm 1:** Turns NE, SW, NW, and SE are forbidden. The other four turns are allowed.
- **Routing Algorithm 2:** Turns SW and NW are forbidden. The other six turns are allowed.
- **Routing Algorithm 3:** Turns WN and NW are forbidden. The other six turns are allowed.

For the rest of this question, we address source-destination pairs using <src, dest> representation. Please answer the following questions and show your work.

(a) [50 points] Show all possible paths between source and destination pairs <P_5, P_{11}> and <P_5, P_{12}>, using each of the previously mentioned routing algorithms. For each path, mention the source router, intermediate routers, and destination router.

2D Mesh

Algorithm 1, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}
Algorithm 1, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}
Algorithm 2, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}, P_5-P_6-P_{10}-P_{11}, P_5-P_9-P_{10}-P_{11}
Algorithm 2, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}
Algorithm 3, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}, P_5-P_6-P_{10}-P_{11}, P_5-P_9-P_{10}-P_{11}
Algorithm 3, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}, P_5-P_9-P_8-P_{12}, P_5-P_9-P_{13}-P_{12}

2D Torus

Algorithm 1, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}, P_5-P_4-P_7-P_{11}
Algorithm 1, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}, P_5-P_4-P_0-P_{12}
Algorithm 2, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}, P_5-P_6-P_{10}-P_{11}, P_5-P_9-P_{10}-P_{11}, P_5-P_4-P_7-P_{11}
Algorithm 2, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}, P_5-P_4-P_0-P_{12}
Algorithm 3, <P_5, P_{11}>	P_5-P_6-P_7-P_{11}, P_5-P_6-P_{10}-P_{11}, P_5-P_9-P_{10}-P_{11}, P_5-P_9-P_8-P_{11}, P_5-P_4-P_8-P_{11}, P_5-P_4-P_7-P_{11}
Algorithm 3, <P_5, P_{12}>	P_5-P_4-P_8-P_{12}, P_5-P_9-P_8-P_{12}, P_5-P_9-P_{13}-P_{12}

(b) [45 points] Which of these three algorithms are deadlock free? For each case, if deadlocks can hap pen, provide a deadlock scenario. Otherwise, prove that deadlock occurrence is impossible (Hint: Use contradiction).

2D Mesh

Routing Algorithm 1:

Deadlock Free

Explanation: This routing algorithm is a special case, and thus more restricted version of the next algorithm. If we can prove that the next algorithm is deadlock free, it will also be a solution for this routing algorithm.

Routing Algorithm 2:

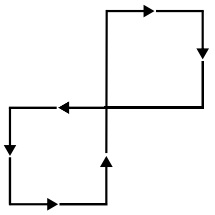
Deadlock Free

Explanation: We prove that there exists no deadlock using this routing algorithm in a 2D-Mesh topology. Let's assume for the sake of contradiction, that there exists an arbitrary scenario that will cause a deadlock in the network. This means that there exists a clock-wise (counter clock-wise) cycle using a subset of the possible turns in our routing algorithm. Consider the rightmost edge of this clock-wise (counter clock-wise) cycle; if at any point you reach this edge, for you to move along the clock-wise (counter clock-wise) cycle you need to take at least one SW (NW) turn, which is forbidden in the routing algorithm. This is a contradiction with the assumptions, hence the routing algorithm for a 2D-Mesh is deadlock free.

Routing Algorithm 3:

Deadlocks can happen

Explanation: A deadlock can happen. For instance, consider there are eight packets with the following sources and destinations: $<P_1, P_6>$, $<P_2, P_5>$, $<P_6, P_4>$, $<P_5, P_8>$, $<P_4, P_9>$, $<P_8, P_5>$, $<P_9, P_1>$, and $<P_5, P_2>$. If all these eight packets are injected into the network together at once, a deadlock can happen. The following figure shows how combining the truns these packets should take can form a deadlock circle:



2D Torus

Routing Algorithm 1:

Deadlocks can happen

Explanation: For instance, consider there are four packets with the following sources and destinations: $<P_0, P_2>$, $<P_1, P_3>$, $<P_2, P_0>$, and $<P_3, P_1>$. If all these four packets are injected into the network together at once, a deadlock can happen.

Routing Algorithm 2:

Deadlocks can happen

Explanation: For instance, consider there are four packets with the following sources and destinations: $<P_0, P_2>$, $<P_1, P_3>$, $<P_2, P_0>$, and $<P_3, P_1>$. If all these four packets are injected into the network together at once, a deadlock can happen.

Routing Algorithm 3:

Deadlocks can happen

Explanation: For instance, consider there are four packets with the following sources and destinations: $<P_0, P_2>$, $<P_1, P_3>$, $<P_2, P_0>$, and $<P_3, P_1>$. If all these four packets are injected into the network together at once, a deadlock can happen. In addition, we have already shown that this routing algorithm is not deadlock free for a 2D mesh topology. Hence, it is not deadlock free for 2D torus nighter.

You are about to design a routing algorithm for each of these two topologies with the following two requirements: Your routing algorithm (1) should be deadlock free, and (2) should *not* be deterministic.

(c) [20 points] Which routing algorithm out of the three earlier routing algorithms do you choose for each of these two topologies? Explain your reasoning.

2D mesh

Routing Algorithm 2

Explanation: Only routing algorithms 1 and 2 are deadlock free, and only algorithm 2 is not deterministic.

2D torus

None

Explanation: I would not choose any of these three algorithms, since none of them is deadlock free.

8 BONUS: Prefetching using Reinforcement Learning [70 points]

You are designing a hardware prefetcher for a processor using a reinforcement learning (RL) agent, as discussed in lecture about the Pythia prefetcher. For a memory request to cacheline address A, the prefetcher selects a prefetch offset O and issues a prefetch memory request to cacheline address $A + O$. For every prefetch request, the memory hierarchy provides a numerical reward R to the prefetcher that can take a value of one of the following three reward levels:

- Accurate (R_A), signifying that the prefetch request was demanded by the processor.
- Inaccurate (R_{IN}), signifying that the prefetch request was not demanded by the processor.
- No-prefetch (R_{NP}), signifying that the prefetcher did not prefetch anything.

In the initial configuration of the prefetcher, you set the values of the reward levels as follows: $R_A = 20$, $R_{IN} = -4$, and $R_{NP} = -10$.

Recall that the ***coverage*** of a prefetcher is defined as the fraction of a program's memory requests correctly prefetched by the prefetcher, while the ***accuracy*** of a prefetcher is defined as the fraction of prefetched requests that are actually demanded by the program.

(a) [25 points] Which of the following statements, if any, are **CORRECT** if you set $R_{NP} = 1000$ in the initial prefetcher configuration? All other reward level values, except R_{NP}, remain the same as the initial configuration. Select **ALL** that apply and explain briefly. You may get partial credits for a partially-complete answer, given a correct explanation.

A. The coverage of the prefetcher will significantly *increase*.

B. The coverage of the prefetcher will significantly *decrease*.

C. The prefetcher will start prefetching aggressively.

D. The accuracy of the prefetcher may *increase*.

(B) and (D).

Setting $R_{NP} \gg R_A$ will strongly encourage the prefetcher not to prefetch. As a result, the prefetcher's coverage will drop significantly. However, the accuracy of the prefetcher may increase, as the number of generated prefetch requests will significantly reduce.

(b) [25 points] Which of the following statements, if any, are **CORRECT** if you set $R_{IN} = -500$ in the initial prefetcher configuration? All other reward level values, except R_{IN}, remain the same as the initial configuration. Select **ALL** that apply and explain briefly. You may get partial credits for a partially-complete answer, given a correct explanation.

A. The prefetcher will aggressively prefetch, even though the prefetches might be inaccurate.

B. The accuracy of the prefetcher will likely *increase*.

C. The accuracy of the prefetcher will likely *decrease*.

D. The coverage of the prefetcher might *increase or decrease*.

(B) and (D).

Setting $R_{IN} \ll R_{NP}$ will strongly discourage the prefetcher to generate any prefetch request that might not be accurate. As a result, the accuracy of the prefetcher will likely increase. However, the coverage of the prefetcher might increase or decrease.

(c) [20 points] You want to make the prefetcher system-aware by incorporating into the decision making process *the type of power source* used in the system. You want to configure the prefetcher in the following way:

i. The prefetcher should generate accurate prefetches whenever possible, irrespective of whether the system is connected to an external power adapter or running on a battery.

ii. If the system is connected to an external power adapter, the prefetcher should continue to prefetch, even if the prefetch might be inaccurate.

iii. If the system is running on battery, the prefetcher should prefer not to generate a prefetch request if the prefetch is likely to be inaccurate.

How would you configure the values of the three reward levels? The relative ordering of the reward level values is sufficient for a correct answer, rather than their exact values. Fill in the blanks below and explain your reasoning briefly in the provided box.

```
if(power_source == battery)
```

$$\boxed{R_{IN}} < \boxed{R_{NP}} < \boxed{R_{A}}$$

```
else if (power_source == external_adapter)
```

$$\boxed{R_{NP}} < \boxed{R_{IN}} < \boxed{R_{A}}$$

When the system is running on a battery, we set $R_{IN} < R_{NP} < R_A$, because we want the prefetcher to prefer not to prefetch rather than inaccurate prefetching. When the system is connected to an external power adapter, we set $R_{NP} < R_{IN} < R_A$, because we want the prefetcher to continue prefetching, even if the prefetch might be inaccurate.