Final Exam

Computer Architecture (227-2210-00L)

ETH Zürich, Fall 2022

Prof. Onur Mutlu

Problem 1 (50 Points): Prefetching
Problem 2 (85 Points): RowHammer
Problem 3 (100 Points): Processing-in-Memory
Problem 4 (80 Points): DRAM Refresh
Problem 5 (70 Points): Genome Analysis Acceleration
Problem 6 (80 Points): Memory Quality-of-Service
Problem 7 (60 Points): Emerging Memory Technologies
Problem 8 (60 Points): Memory Consistency
Problem 9 (60 Points): BONUS: Cache Coherence
Problem 10 (90 Points): BONUS: Interconnects

Total (735 (585 + 150 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 Data Prefetching [50 points]

You are tasked with designing the memory hierarchy of an in-order, non-pipelined processor. The memory hierarchy is composed of an on-chip cache and an off-chip DRAM-based main memory, with an access latency of 50 cycles and 200 cycles (excluding on-chip cache latency), respectively. In the baseline processor design, the processor first accesses the cache. If the data is not present in the cache, then the processor accesses the main memory. Assume that the data is always present in the main memory.

You run an application that has the following memory request access pattern (note that these are cache block addresses). **Assume that all these memory requests miss the cache and go to the main memory in the baseline processor design.**

\[ A, A + 1, A + 8, A + 9, A + 16, A + 17, A + 24, A + 25, \ldots, A + 8n, A + 8n + 1 \]

Also assume that the application needs to execute only the memory requests and no other instructions.

(a) [15 points] What is the total number of cycles required for the baseline processor to execute the application? Explain briefly.

\[
500(n + 1) \text{ cycles}
\]

**Explanation:** The memory request to address \( A \) takes 50 cycles to access cache and 200 cycles to access the main memory. Hence a total of 250 cycles. Similarly, the memory request to address \( A + 1 \) takes 250 cycles. Thus the entire memory request sequence would take \( 500(n + 1) \) cycles to complete.

(b) [15 points] To make the processor faster, you design a next-block prefetcher. For every memory access to cacheline address \( X \), the next-block prefetcher fetches the data corresponding to address \( X + 1 \) to processor’s on-chip cache. Assume that the prefetcher fetches the data before the application demands it. What is the speedup of this new processor design over the baseline processor design mentioned in (a)? Explain briefly.

\[
1.66 \times
\]

**Explanation:** The prefetcher will fetch the data for address \( A + 1 \) to the cache by seeing the request to address \( A \). Hence, request to address \( A \) would take 250 cycles and request to address \( A + 1 \) would take 50 cycles. Thus, the entire memory request sequence will take \( 300(n + 1) \) cycles.

\[
\text{Speedup} = \frac{500(n + 1)}{300(n + 1)} = 1.66 \times
\]
(c) [10 points] Your colleague observes that accessing the cache to check whether a memory request would go to main memory also incurs a latency overhead. So he proposes an "off-chip predictor" that accurately predicts which memory request would go to the main memory. For those predicted requests, the processor can now directly access the main memory to get the data \textit{without even accessing the cache first}. If you implement this predictor in the baseline processor mentioned in (a), what would be the speedup of the new processor design over the baseline processor design, assuming 100% off-chip prediction accuracy? Explain briefly.

\[
1.25 \times
\]

\textbf{Explanation:}
Since each memory request would now take 200 cycles to finish, the entire memory request sequence would take 400(n + 1) cycles.

\[
\text{Speedup} = \frac{500(n+1)}{400(n+1)} = 1.25 \times
\]

(d) [10 points] If you implement both your prefetcher and your colleague's predictor together in the baseline processor, what would be the speedup of the new processor design over the baseline processor design? Explain briefly.

\[
2 \times
\]

\textbf{Explanation:}
For the memory request to address \(A\), the processor would take 200 cycles to complete, since the predictor will correctly predict to fetch the data directly from the main memory. For the memory request to address \(A+1\), the prefetcher will fetch the data to the cache. Hence the processor would take 50 cycles to complete.

\[
\text{Speedup} = \frac{500(n+1)}{250(n+1)} = 2 \times
\]
2 RowHammer [85 points]

2.1 RowHammer Properties [15 points]

Determine 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] RowHammer can be used for taking over a DRAM-based system.

1. True 2. False

(b) [3 points] DDR3 DRAM chips exhibit vulnerability to RowHammer but DDR4 and DDR5 DRAM chips are not vulnerable to RowHammer.

1. True 2. False

(c) [3 points] Mobile devices are RowHammer-safe because they use low power memory chips.

1. True 2. False

(d) [3 points] BlockHammer prevents RowHammer bitflips by refreshing potential victim rows.

1. True 2. False

(e) [3 points] There is no attack in the literature that defeats the RowHammer defenses which adopt security by obscurity approach (e.g., in-DRAM target row refresh mechanisms).

1. True 2. False

2.2 RowHammer Mitigation [70 points]

To design a memory-controller-based RowHammer mitigation mechanism, assume that the DRAM chips that will be used are organized as 1 channel, 1 rank, 16 banks, 8 KB row size, and 8 GB total capacity. Each DRAM row should be refreshed every 64 ms, and there should be at least 64 ns between two row activations targeting the same DRAM bank. The attacker needs to activate an aggressor row at least 1000 times without its neighboring rows (victim rows) being refreshed to induce a RowHammer bitflip in the victim rows, i.e., the RowHammer threshold \(N_{RH}\) is 1000.

(a) [20 points] How many rows in a bank can an attacker concurrently hammer enough times to induce bit flips? Show your work and explain clearly.

Let \(N_{ACT}\) be the maximum number of activations that can be issued to a bank within a refresh window. \(N_{ACT} = 64\text{ms}/64\text{ns} = 10^6\) row activations.

Each aggressor row should be activated 1000 times. Let \(N_{aggr}\) be the maximum number of aggressor rows that can be concurrently hammered and \(N_{RH}\) be the RowHammer threshold.

Then, an attacker can concurrently hammer \(N_{aggr} = N_{ACT}/N_{RH} = 10^6/(10^3) = 1000\) aggressor rows in a DRAM bank.
(b) [30 points] You are given a RowHammer defense mechanism, which remaps an aggressor row to another DRAM row within the same bank when the aggressor row’s activation count reaches a given threshold value \(N_{TH}\). To remap a DRAM row from row address A to row address B, the memory controller

- fetches row A’s content into a buffer, implemented in the memory controller.
- fetches row B’s content into another buffer, implemented in the memory controller.
- writes row A’s content from the buffer to DRAM row B.
- writes row B’s content from the buffer to DRAM row A.

Evaluate the worst-case data movement overhead of this mechanism in terms of the fraction of time spent for this data movement for a RowHammer threshold value \(N_{TH}\) of 1000 and DRAM channel bandwidth of 20GB/s. Show your work and explain clearly.

Hint: First, calculate the number of bytes that need to be moved between the memory controller and the DRAM chips in a 64ms time window. Then, calculate the time spent to perform this data movement, assuming that the memory controller can fully utilize the memory bandwidth for this data movement. Finally, calculate the fraction of time spent for this data movement in a 64ms time window. You can assume 20GB/s is 20,000MB/s to simplify your calculation.

\[
\text{The attacker can perform 1 million activations in a DRAM bank in a refresh window of 64ms. Therefore, the attacker can trigger the remapping mechanism 1000 times in a DRAM bank in a refresh window.}
\]

\[
\text{Since the attacker can concurrently hammer rows in all banks, the total number of remapping operations is } 1000 \times 16 = 16000 \text{ in a 64ms time window.}
\]

\[
\text{Each remapping operation requires 32KB of data movement (8KB for fetching row A, 8KB for fetching row B, 8KB for writing back into row B and 8KB for writing back into row A). Therefore, the total data movement is } 16000 \times 32KB = 500MB \text{ in a refresh window.}
\]

\[
\text{The memory bandwidth is 20,000MB/s. Therefore, the time spent for data movement is } 500MB/20,000MB/s = 25ms.
\]

\[
\text{Finally, the fraction of time spent for data movement is } 25ms/64ms = 39\%. \quad (25/64 \text{ is sufficient for the answer.)}
\]
(c) [20 points] List one advantage and one disadvantage of each of PARA and BlockHammer mechanisms. Explain your answer.

This question has more than one correct answers. We provide an example below.

- PARA:
  - Advantage: Low hardware complexity. PARA is stateless, therefore it does not have a significant area overhead and its area overhead does not increase as RowHammer worsens.
  - Disadvantage: Large performance overhead. PARA performs preventive refresh operations increasingly more aggressively as DRAM chips become more vulnerable to RowHammer.

- BlockHammer:
  - Advantage: Low performance overhead when there is no RowHammer attack and significant performance improvement under a RowHammer attack. BlockHammer achieves this by throttling the attacker.
  - Disadvantage: High hardware complexity. BlockHammer maintains a significant number of counters (in counting Bloom filters) and a large history buffer both of whose area overhead significantly increase as RowHammer worsens.
3 In-DRAM Bit-Serial Computation [100 points]

Recall that in class, we discussed Am-bit, which is a DRAM design that can greatly accelerate bulk bitwise operations by providing the ability to perform bitwise AND/OR of two rows in a subarray and NOT of one row. Since Am-bit is logically complete, it is possible to implement any other logic function (e.g., XOR). To be able to implement arithmetic operations, bit shifting is also necessary. There is no way of shifting bits in DRAM with a conventional layout, but it can be done with a bit-serial layout, as Figure 1 shows. With such a layout, it is possible to perform bit-serial arithmetic computations in Am-bit.

![Figure 1: In-DRAM bit-serial layout for array A, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: \( A[i]_j \) represents bit \( j \) of element \( i \). Bit 0 is the least significant bit of an element, and bit \( n-1 \) is the most significant bit, for an element size of \( n \).]

We want to evaluate the potential performance benefits of using Am-bit for arithmetic computations by implementing a simple workload: element-wise subtraction of two arrays. Listing 1 shows a sequential code for the subtraction of two input arrays \( A \) and \( B \) into output array \( C \).

Listing 1: Sequential CPU implementation of element-wise subtraction of arrays \( A \) and \( B \).

```c
for(int i = 0; i < num_elements; i++){
    C[i] = A[i] - B[i];
}
```

We compare two possible implementations of the element-wise subtraction of two arrays: a CPU-based and an Am-bit-based implementation. We make two assumptions. First, we use the most favorable layout for each implementation (i.e., conventional layout for CPU, and bit-serial layout for Am-bit). Second, both implementations can operate on array elements of any size (i.e., bits/element):

- **CPU-based implementation**: This implementation reads elements of \( A \) and \( B \) from memory, subtracts them, and writes the resulting elements of \( C \) into memory.

  Since the computation is simple and regular, we can use a simple analytical performance model for the execution time of the CPU-based implementation: \( t_{cpu} = K \times num\_operations + \frac{num\_bytes}{M} \), where \( K \) represents the cost per arithmetic operation and \( M \) is the DRAM bandwidth. (Note: \( num\_operations \) should include only the operations for the array subtraction.)

- **Am-bit-based implementation**: This implementation assumes a bit-serial layout for arrays \( A, B, \) and \( C \). It performs subtractions in a bit-serial manner, which only requires XOR, AND, NOT, and OR operations, as you can see in the 1-bit full subtractor in Figure 2.
Ambit implements these operations by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) operations. For example, to compute AND, OR, and XOR operations, Ambit issues the sequence of commands described in Table 1, where AAP(X,Y) represents double row activation of rows X and Y followed by a precharge operation, and AAAP(X,Y,Z) represents triple row activation of rows X, Y, and Z followed by a precharge operation.

In those instructions, Ambit copies the source rows Di and Dj to auxiliary rows (Bi). Control rows Ci dictate which operation (AND/OR) Ambit executes. The DRAM rows with dual-contact cells (i.e., rows DCCi) are used to perform the bitwise NOT operation on the data stored in the row. Basically, the NOT operation copies a source row to DCCi, inverts all bits of the row, and stores the result in both the source row and DCCi. Assume that:

- The DRAM row size is 8 KB.
- An ACTIVATE command takes 40 ns to execute.
- A PRECHARGE command takes 20 ns to execute.
- DRAM has a single memory bank.
- The syntax of an Ambit operation is: \texttt{bbop\_\_and/or/xor destination, source\_1, source\_2}.
- The rows at addresses 0x00500000, 0x00600000, 0x00700000, 0x00800000, and 0x009000000 are used to store partial results. Initially, they contain all zeroes.
- The rows at addresses 0x000A0000, 0x000B0000, and 0x000C0000 store arrays A, B, and C, respectively.
- The rows at addresses 0x00D00000 and 0x00E00000 store pre-initialized rows containing all ones and all zeroes elements, respectively.
- These are all byte addresses. All these rows belong to the same DRAM subarray.

Table 1: Sequences of ACTIVATE and PRECHARGE operations for the execution of Ambit’s AND, OR, and XOR operations.

<table>
<thead>
<tr>
<th>Dk = Di AND Dj</th>
<th>Dk = Di OR Dj</th>
<th>Dk = Di XOR Dj</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAP (Di, B0)</td>
<td>AAP (Di, B1)</td>
<td>AAP (Di, DCC0)</td>
</tr>
<tr>
<td>AAP (Di, B1)</td>
<td>AAP (Di, B0)</td>
<td>AAP (Di, DCC1)</td>
</tr>
<tr>
<td>AAP (Ci, B2)</td>
<td>AAP (Ci, B2)</td>
<td>AAP (B0, DCC0, B2)</td>
</tr>
<tr>
<td>AAAP (B0, B1, B2)</td>
<td>AAAP (B0, B1, B2)</td>
<td>AAP (B1, DCC0, B2)</td>
</tr>
<tr>
<td>AAP B0, Dk</td>
<td>AAP B0, Dk</td>
<td>AAP (B0, Dk)</td>
</tr>
<tr>
<td>AAP (Di, B0)</td>
<td>AAP (Di, B1)</td>
<td>AAP (Di, DCC0)</td>
</tr>
<tr>
<td>AAP (Di, B1)</td>
<td>AAP (Di, B0)</td>
<td>AAP (Di, DCC1)</td>
</tr>
<tr>
<td>AAP (Ci, B2)</td>
<td>AAP (Ci, B2)</td>
<td>AAP (B0, DCC0, B2)</td>
</tr>
<tr>
<td>AAAP (B0, B1, B2)</td>
<td>AAAP (B0, B1, B2)</td>
<td>AAP (B1, DCC0, B2)</td>
</tr>
<tr>
<td>AAP B0, Dk</td>
<td>AAP B0, Dk</td>
<td>AAP (B0, Dk)</td>
</tr>
<tr>
<td>AAP (Di, B0)</td>
<td>AAP (Di, B1)</td>
<td>AAP (Di, DCC0)</td>
</tr>
<tr>
<td>AAP (Di, B1)</td>
<td>AAP (Di, B0)</td>
<td>AAP (Di, DCC1)</td>
</tr>
<tr>
<td>AAP (Ci, B2)</td>
<td>AAP (Ci, B2)</td>
<td>AAP (B0, DCC0, B2)</td>
</tr>
<tr>
<td>AAAP (B0, B1, B2)</td>
<td>AAAP (B0, B1, B2)</td>
<td>AAP (B1, DCC0, B2)</td>
</tr>
<tr>
<td>AAP B0, Dk</td>
<td>AAP B0, Dk</td>
<td>AAP (B0, Dk)</td>
</tr>
</tbody>
</table>
(a) [30 points] For the CPU-based implementation, you want to obtain $K$ and $M$. To this end, you run two experiments. In the first experiment, you run your CPU code for the element-wise array subtraction for 65,536 32-bit elements and measure $t_{cpu} = 100 \mu s$. In the second experiment, you run the STREAM-Copy benchmark for 128,000 32-bit elements and measure $t_{cpu} = 10 \mu s$. The STREAM-Copy benchmark simply copies the contents of one input array $A$ to an output array $B$.

What are the values of $K$ and $M$?

$$M = 10.24 \text{ GB/s} \text{ and } K = 1.38 \text{ ns/operation.}$$

**Explanation:**

We first calculate $M$ by using the measurement for the STREAM-Copy benchmark, which does not involve any computation. For `num_bytes`, we count two arrays of 128,000 32-bit elements:

$$t_{cpu} = \frac{num\_bytes}{M};$$

$$\Rightarrow 10 \times 10^{-6} = \frac{128,000 \times 32 \times 2}{8 \times M};$$

$$\Rightarrow M = 102.4 \text{ GB/s}. $$

Then, we obtain $K$ with the measurement for the array subtraction. For `num_operations`, we count the same number as `num_elements`. For `num_bytes`, we count three arrays of 65,536 32-bit elements:

$$t_{cpu} = K \times num\_operations + \frac{num\_bytes}{M};$$

$$\Rightarrow 100 \times 10^{-6} = K \times 65,536 + \frac{65,536 \times 32 \times 3}{8 \times 1024 \times 10^9};$$

$$\Rightarrow K = 1.4 \text{ ns/operation.}$$

(b) [30 points] Write the code for the Ambit-based implementation of the $n$-bit element-wise subtraction of arrays $A$ and $B$ using `bbop_ [and/or/xor]` instructions. The resulting array is $C$. Show your work.

```c
for(int i = 0; i < n; i++) {
    bbop_xor 0x00600000, 0x00A00000, 0x00B00000;
    bbop_xor 0x00C00000, 0x00600000, 0x00700000;
    bbop_xor 0x00800000, 0x00D00000, 0x00A00000;
    bbop_and 0x00900000, 0x00800000, 0x00B00000;
    bbop_xor 0x00800000, 0x00D00000, 0x00600000;
    bbop_and 0x00600000, 0x00800000, 0x00700000;
    bbop_or 0x00700000, 0x00600000, 0x00900000;
}
```
**Explanation:**

First, since there is no bop for negating the inputs of the AND gates in Figure 2, we need to implement the NOT operation using XOR (i.e., $\text{NOT}(A) \equiv \text{XOR}(1, A)$) in lines 7 and 11.

Second, the loop needs to execute $n \times$ in order to implement a $n$-bit subtraction operation.

(c) [25 points] Compute the maximum throughput (in arithmetic operations per second, OPS) of the Ambit-based implementation as a function of the element size $n$ (i.e., bits/element). Note: To simplify your calculations, you can assume that 8 KB $\approx 65,000$.

\[
\text{Thr}_{\text{ambit}} \approx \frac{10}{n \times 10^{-9}} \text{ OPS} \approx \frac{10}{n} \text{ GOPS}.
\]

**Explanation:**

Since DRAM has one single bank (and we can operate on a single subarray), the maximum throughput is achieved when we use complete rows. As the row size is 8 KB, the maximum array size that we can work with is 65,536 elements.

First, we obtain the execution time as a function of the number of bits per element. Each XOR operation employs 25 ACTIVATION and 11 PRECHARGE operations. For AND and OR, 11 ACTIVATION and 5 PRECHARGE operations. Thus, the execution time of the bit-serial computation on the entire array can be computed as ($n$ is the number of bits per element):

\[
t_{\text{ambit}} = (4 \times t_{\text{XOR}} + 2 \times t_{\text{AND}} + t_{\text{OR}}) \times n;
\]

\[
\Rightarrow t_{\text{ambit}} = 6,500 \times n \text{ ns}.
\]

Second, we obtain the throughput in arithmetic operations per second (OPS) as:

\[
\text{Thr}_{\text{ambit}} = \frac{65,536}{6,500 \times n \times 10} \text{ OPS}.
\]

\[
\text{Thr}_{\text{ambit}} \approx \frac{65,000}{6,500 \times n \times 10} \text{ OPS} \approx \frac{10}{n \times 10^{-9}} \text{ OPS} \approx \frac{10}{n} \text{ GOPS}.
\]

(d) [15 points] Determine the element size $n$ (in bits) for which the CPU-based implementation is faster than the Ambit-based implementation. Use the same array size as in (a).

\[n \text{ > 14 bits.}\]

**Explanation:**

We want to find $n$ such that $\text{Thr}_{\text{ambit}} < \text{Thr}_{\text{cpu}}$, or $t_{\text{ambit}} > t_{\text{cpu}}$. If we use arrays of size 65,536 elements, we can write the following expression:

\[
t_{\text{ambit}} > t_{\text{cpu}};
\]

\[
\Rightarrow 6,500 \times n \times 10^{-9} > 1.4 \times 65,536 \times 10^{-9} + \frac{65,536 \times 3 \times n}{8 \times 1024 \times 4 \times 10^9};
\]

\[
\Rightarrow n > 14 \text{ bits}; \text{ where } n \text{ is a natural number.}
4 DRAM Refresh [80 points]

Assume you work as a system architect for a company that operates its own data centers. One day, your boss learned that refresh is one of the major challenges in DRAM scaling, and asks you to take a look at DRAM refresh in your existing systems. You find that the typical node in your data centers has the following DRAM organization and configuration:

- Four channels, one rank per channel, eight chips per rank, 16 banks per chip, and $2^{17}$ rows per bank.
- The refresh interval ($t_{REFI}$, i.e., how often the memory controller sends a refresh command) is $3.9\mu s$, and the refresh window ($t_{REFW}$, i.e., how often every DRAM row is refreshed) is $32ms$.
- A refresh command refreshes a certain number (assume a power of two) of rows in all banks of a rank. It occupies the command bus for $1ns$. The DRAM device spends $600ns$ ($t_{RFC}$) refreshing the rows for every refresh command.

Note: For this question, it is not necessary to evaluate your solution arithmetically to the end. Show your steps and reasoning clearly is enough to receive full points.

(a) [10 points] How many DRAM rows in a bank are refreshed for a refresh command? Select your answer below and show your steps. (Hint: Round the number of refresh commands per refresh window to the nearest power of two.)

A. 4 Rows  
B. 8 Rows  
C. 16 Rows  
D. 32 Rows

\[
32ms/3.9\mu s = 8192 \text{ refresh commands are issued per } t_{REFW}. \text{ So a refresh command refreshes } \frac{131072}{8192} = 16 \text{ rows in a bank.}
\]

(b) [10 points] What are refresh commands’ 1) command bus occupancy (i.e., the fraction of time that the command bus is occupied by the refresh command) and 2) bank occupancy (i.e., the fraction of time that the bank is serving refresh commands) in a refresh window?

\[
\text{Every } 32ms, 8192 \text{ refresh commands are issued. The command bus occupancy is } \frac{8192\times1ns}{32ms} = 0.0256\%. \text{ The bank occupancy is } \frac{8192\times600ns}{32ms} = 15.36\%.
\]
The engineering team of your company develops your own DRAM technology where they claim that a DRAM cell can retain data for 128 ms (i.e., it only needs to be refreshed every 128 ms). However, upon further testing, you find this technology is very susceptible to the Variable Retention Time (VRT) phenomenon: a VRT cell can retain data for 128 ms in its low-leakage state, but only 16 ms in its high-leakage state. A normal cell (i.e., without the VRT phenomenon) can always retain data for 128 ms. For simplicity, assume a normal DRAM cell never becomes a VRT cell and vice versa.

(c) [30 points] Assume you design a smart memory controller that knows the distribution of the VRT cells in the DRAM device. This memory controller can send refresh commands at different intervals to different groups of DRAM rows. What is the minimum number of DRAM rows without any VRT cell in a bank such that the bank occupancy of refresh operations of your new DRAM device is no larger than that of your existing DRAM? Assume the same DRAM organization, the same tRFC value, and a refresh command refreshes the same number of DRAM rows as in (a) and (b).

The bank occupancy of the existing DRAM device is 15.36%. All 131072 rows are divided into 8192 row groups, a refresh command will refresh all 16 rows in a group. For the new device, consider a time window of 128 ms, assume $X$ row groups have no VRT cells at all (i.e., they can be refreshed every 128 ms): $X$ refresh operations will happen for these groups, and $\frac{128\text{ms}}{16\text{ms}}$ refresh operations will happen for the rest of the row groups. Then the bank occupancy is $\frac{600\text{ms} \times X + 600\text{ms} \times (8192 - X) \times \frac{16}{128\text{ms}}}{128\text{ms}}$. Solve it for $\leq 15.36\%$ gives $X \geq 4682$. Therefore, at least $4682 \times 16 = 74912$ rows have to contain no VRT cells at all.
Unfortunately, the engineering team reports that they could not guarantee that many VRT-cell-free rows in a bank (as you calculate in (c)). As a result, you look into the workload’s memory access pattern to find opportunities to reduce the refresh cost. Luckily, your data center is dedicated to run a particular workload with a very regular access pattern. For simplicity, consider a single VRT cell located in Row A:

- The workload accesses Row A exactly every 64ms, as shown at timestamps t1 and t3 below.
- You configure the system such that Row A is refreshed every 64ms, as shown at timestamps t0 and t2 below.
- Refresh operations are not synchronized with the workload accesses. Assume that t2 can happen at any timestamp between t1 and t3 with equal probability.
- The cell changes its leakage state with a 30% of probability when the wordline of Row A is enabled.
- Initially, the cell is at low-leakage state, and gets immediately refreshed, as shown at timestamp t0 below.

Calculate the probability of not violating the retention time of the cell for 64ms of execution. (Hint: You can create a tree for the probabilistic state transitions, as we partially provide below.)

We define $t_D$ as $t_1 - t_0$.

There are three paths to satisfy the retention time of the VRT cell until it reaches to $t_2$.

1. It remains in the low-leakage state during both the refresh at $t_0$ and the workload access at $t_1$. Probability of this case is $0.7 \times 0.7$.
2. It remains in the low-leakage state during the refresh at $t_0$, but switches to high-leakage during the workload access at $t_1$. In this case $t_2 - t_1$ should be smaller than 16ms to satisfy the retention time. Then, the probability of this case is $0.7 \times 0.3 \times P(t_D > 48ms)$.
3. It switches to high-leakage state during the refresh at $t_0$, and switches back to low-leakage state during the workload access at $t_1$. In this case $t_1 - t_0$ should be smaller than 16ms to satisfy the retention time. Then, the probability of this case is $0.3 \times 0.3 \times P(t_D < 16ms)$.

The desired probability is the sum of these three probabilities.

Note that the retention time is inevitably violated in the fourth path, where the cell switches to high-leakage state during the refresh at $t_0$, and preserves its leakage state during the workload access at $t_1$. This happens because $t_D$ cannot be smaller than 16ms and larger than 48ms at the same time.
5 Genome Analysis [70 points]

Recall that the goal of mapping a read is to find substrings in some reference genome that the read is similar (i.e., matches) to. A read and the reference genome are each a string over the alphabet \( \Sigma = \{A, C, G, T\} \). For the rest of the question, assume that the reference genome consists of \( m = 3 \times 10^9 \) characters, and that any read consists of \( n = 10^3 \) characters.

(a) [10 points] You are given an algorithm that can find the substrings in an \( m \)-character reference genome that match an \( n \)-character read string by computing a dynamic programming table of \( n \times m \) cells. How many cells must be computed to map \( 10^6 \) reads with this Naive strategy?

**Answer:** \( 3 \times 10^{18} \) cells
Each read consists of \( n = 10^3 \) characters. The reference consists of \( m = 3 \times 10^9 \) characters.
To map one read, the algorithm computes \( n \times m = 10^3 \times 3 \times 10^9 = 3 \times 10^{12} \) cells.
To map \( 10^6 \) reads, the algorithm computes \( 10^6 \times 3 \times 10^{12} = 3 \times 10^{18} \) cells.

(b) [10 points] To reduce the number of computed cells, you decide to follow the Seed-and-Extend paradigm. The seeding step provides a number of candidate locations in the reference where the read may match. To evaluate the reference substring at each candidate location, a dynamic programming algorithm computes a dynamic programming table of \( n^2 \) cells, where \( n \) is the length of the read. For the rest of the question, assume that (1) the seeding step provides 100 candidate locations per read, and (2) that seeding requires no computation. How many cells must be computed to map \( 10^6 \) reads?

**Answer:** \( 10^{14} \) cells
Each read consists of \( n = 10^3 \) characters. For each read, 100 candidate locations are evaluated.
To map one read, the algorithm computes \( 100 \times n^2 = 100 \times (10^3)^2 = 10^8 \) cells.
To map \( 10^6 \) reads, the algorithm computes \( 10^6 \times 10^8 = 10^{14} \) cells.

(c) [10 points] To further reduce the number of computed cells, you decide to follow the Seed-Filter-Extend paradigm and introduce a pre-alignment filter as discussed in the lecture. The new pre-alignment filter successfully filters out 72\% of candidate locations. How many cells must be computed to map \( 10^6 \) reads?

**Answer:** \( 2.8 \times 10^{13} \)
Each read consists of \( n = 10^3 \) characters. For each read, \( 100 - 72 = 28 \) candidate locations are evaluated.
To map one read, the algorithm computes \( 28 \times n^2 = 28 \times (10^3)^2 = 2.8 \times 10^7 \) cells.
To map \( 10^6 \) reads, the algorithm computes \( 10^6 \times 2.8 \times 10^7 = 2.8 \times 10^{13} \) cells.

(d) [10 points] The pre-alignment filter you use executes \( 0.1 \times n^2 \) operations to evaluate each candidate location of a read, where \( n \) is the length of the read. How many operations of the pre-alignment filter are required to examine the candidate locations of all \( 10^6 \) reads?

**Answer:** \( 10^{13} \)
Each read consists of \( n = 10^3 \) characters. For each read, the filter algorithm evaluates all 100 candidate locations.
To evaluate one location, the filter algorithm executes \( 0.1 \times n^2 = 0.1 \times (10^3)^2 = 10^5 \) operations.
To map one read, the filter algorithm executes \( 100 \times 10^5 = 10^7 \) operations.
To map \( 10^6 \) reads, the filter algorithm executes \( 10^6 \times 10^7 = 10^{13} \) operations.
(e) [10 points] Your CPU of choice runs at 1GHz. At each clock cycle, the CPU can either calculate one dynamic programming cell or perform one operation of the pre-alignment filter. How long (in CPU seconds) does it take to map $10^6$ reads with each of these three strategies (i.e., Naive, Seed-and-Extend, or Seed-Filter-Extend)?

At 1GHz the CPU takes $1ns = 10^{-9}s$ per cell or operation.

- Naive: For $3 \times 10^{18}$ cells (from (a)): $3 \times 10^{18} \times 10^{-9}s = 3 \times 10^9s \approx 8.3 \times 10^8h \approx 95.1y$
- Seed-and-Extend: For $10^{14}$ cells (from (b)): $10^{14} \times 10^{-9}s = 10^5s \approx 27.7h$
- Seed-Filter-Extend: For $2.8 \times 10^{13}$ cells (from (c)) and $10^{13}$ filter operations (from (d)): $(2.8 \times 10^{13} + 10^{13}) \times 10^{-9}s = 3.8 \times 10^4s \approx 10.5h$

(f) [10 points] You are given a hardware accelerator for only the dynamic programming algorithm. The hardware accelerator runs at 1GHz and calculates 10 cells per cycle. The CPU can still execute both the dynamic programming algorithm and the pre-alignment filtering algorithm. The accelerator and CPU can run in parallel. Which of the three strategies (i.e., Naive, Seed-and-Extend, or Seed-Filter-Extend) is the fastest? Support your answer with numbers.

A. Naive  B. Seed-and-Extend  C. Seed-Filter-Extend

Accelerator and CPU can together either calculate $11 \times 10^9$ cells per second, or $10^{10}$ cells and $10^9$ operations.

- Naive: For $3 \times 10^{18}$ cells (from (a)) with CPU+accelerator: $3 \times 10^{18}/(11 \times 10^9) = \frac{3}{11} \times 10^9s \approx 7.5 \times 10^8h \approx 8.6y$
- Seed-and-Extend: For $10^{14}$ cells (from (b)) with CPU+accelerator: $10^{14}/(11 \times 10^9) = \frac{1}{11} \times 10^5s \approx 2.5h$
- Seed-Filter-Extend: For $10^{13}$ filter operations (from (d)) with the CPU: $(10^{13}) \times 10^{-9}s = 10^4s \approx 2.7h$

This alone is already slower than Seed-and-Extend, so we do not need to calculate the time for dynamic programming.

(g) [10 points] Which of the three strategies would be the fastest if the hardware accelerator could also execute filter operations, at 10 operations per cycle? Show your work briefly.

A. Naive  B. Seed-and-Extend  C. Seed-Filter-Extend

The accelerator can execute everything the CPU can execute, but 10 times faster. Thus, the overall runtimes with CPU+accelerator are simply 11 times less than the ones calculated in (e). We know from (e) that Seed-Filter-Extend is the fastest strategy.
6 Memory Quality of Service [80 points]

In lectures, we introduced a variety of ways to tackle memory interference. In this problem, we will examine the Blacklisting Memory Scheduler (BLISS) to reduce unfairness. There are two key aspects of BLISS that you need to know:

- When the memory controller services $\eta$ consecutive requests from a particular application, this application is blacklisted. We name this non-negative integer $\eta$ the **Blacklisting Threshold**.
- The blacklist is cleared periodically every $10000$ cycles starting at $t = 0$.

To reduce unfairness, memory requests in BLISS are prioritized in the following order:

- Non-blacklisted applications' requests
- Row buffer hit requests
- Older requests

The memory system for this problem consists of 2 channels with 2 banks each. Tables 1 and 2 show the memory request stream in the same bank for both applications at different times ($t = 0$ and $t = 10$). For both tables, a request on the left-hand side is older than a request on the right-hand side in the same table. The applications do not generate more requests than those shown in Tables 1 and 2. The memory requests are labeled with numbers that represent the row position of the data within the accessed bank. Assume the following for all questions:

- A row buffer hit takes 200 cycles.
- A row buffer miss (i.e., opening a row in a bank with a closed row buffer) takes 400 cycles.
- A row buffer conflict (i.e., closing the currently open row and opening another one) takes 550 cycles.
- All row buffers are closed at time $t = 0$

<table>
<thead>
<tr>
<th>Application A (Channel 0, Bank 0)</th>
<th>Row 2</th>
<th>Row 2</th>
<th>Row 2</th>
<th>Row 2</th>
<th>Row 2</th>
<th>Row 3</th>
<th>Row 3</th>
<th>Row 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application B (Channel 0, Bank 0)</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 3</td>
<td>Row 3</td>
<td>Row 4</td>
</tr>
</tbody>
</table>

Table 2: Memory requests of the two applications at $t = 0$

<table>
<thead>
<tr>
<th>Application A (Channel 0, Bank 0)</th>
<th>Row 3</th>
<th>Row 7</th>
<th>Row 2</th>
<th>Row 0</th>
<th>Row 5</th>
<th>Row 3</th>
<th>Row 8</th>
<th>Row 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application B (Channel 0, Bank 0)</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 2</td>
<td>Row 3</td>
<td>Row 3</td>
<td>Row 4</td>
</tr>
</tbody>
</table>

Table 3: Memory requests of the two applications at $t = 10$. Note that none of the Application B's existing requests are serviced yet.

(a) [25 points] Compute the slowdown of each application using the FR-FCFS scheduling policy after both threads run to completion. We define:

$$\text{slowdown} = \frac{\text{memory latency of the application when run together with other applications}}{\text{memory latency of the application when run alone}}$$

Show your work.
\[ \text{slowdown}_A = \sim 1.50 \]
\[ \text{slowdown}_B = 1.24 \]

**Explanation:**
For both applications, the first request will incur row buffer miss penalty, and the rest of the requests will either be hits or conflicts.

**Application A (alone)**
\[ 400 + 200 + 550 \times 6 = 3900 \text{ cycles} \]

**Application B (alone)**
\[ 400 + 200 \times 4 + 550 + 200 + 550 = 2500 \text{ cycles} \]

**Applications A (with B, FR-FCFS)**
\[ 400 + 200 \times 4 + 200 + 550 + 200 + 200 \times 2 + 550 + 550 \times 5 = 5850 \text{ cycles} \]

**Applications B (with A, FR-FCFS)**
\[ 400 + 200 \times 4 + 200 + 550 + 200 + 200 \times 2 + 550 = 3100 \text{ cycles} \]

From the two tables above we know that all requests of application B were issued before any of the application A’s requests were issued. Thus, all requests of B are prioritized unless there is a row hit for A’s requests.

\[ \text{slowdown}_A = \frac{5850}{3900} = 1.50 \]
\[ \text{slowdown}_B = \frac{3100}{2500} = 1.24 \]

(b) [15 points] If we use the BLISS scheduler, for what value(s) of \( \eta \) (the Blacklisting Threshold) will the slowdowns of both applications be equal to those obtained with FR-FCFS?

For \( \eta \geq 6 \) or \( \eta = 0 \).

**Explanation:**
We want both A and B to complete without blacklisting or to complete both blacklisted, thus \( \eta \geq 6 \) and \( \eta = 0 \), respectively.
(c) [15 points] For what value(s) of $\eta$ (the Blacklisting Threshold) will the slowdown of $A$ be < 1.5?

Impossible. Slowdown for $A$ will always be $\geq 1.5$

**Explanation:** For the given memory requests, it is not possible to find $\eta$ that blacklists $B$ but not $A$. Thus, the smallest slowdown for $A$ is the case explained in the solution of part (b).

(d) [15 points] For what value(s) of $\eta$ (the Blacklisting Threshold) will $B$ experience the maximum slowdown it can possibly experience with the Blacklisting Scheduler?

For $\eta = 5$.

**Explanation:** We already know that the slow downs will be equal to the slowdown with FR-FCFS when $\eta \geq 6$ or $\eta = 0$. If we execute the memory requests for the rest of possible $\eta$ values, we find that $\eta = 5$ causes application $B$ to complete after 4550 cycles, which is the largest.

(e) [10 points] What is a simple mechanism (that we discussed in lectures) that we can use instead of BLISS to make the slowdowns of both $A$ and $B$ equal to 1.00? Why? Explain briefly.

**Explanation:** Memory Channel Partitioning (MCP)

**Explanation:** With MCP, each application will operate on an independent channel, without any interference with the other application.
7 Emerging Memory Technologies [60 points]

7.1 Phase Change Memory [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] During a multi-level cell (MLC) PCM read operation, the time taken to determine the value of bit 1 is the same as that of bit 0.

1. True 2. False

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

1. True 2. False

(d) [3 points] PCM has lower endurance than DRAM because writes to PCM 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

7.2 ETH-MEM [45 points]
Computer scientists at ETH have developed a new non-volatile memory technology, ETH-MEM. The access latency of ETH-MEM is close to that of DRAM and ETH-MEM provides higher density compared to the latest DRAM technologies. However, compared to DRAM, ETH-MEM has limited endurance, i.e., a memory cell stops functioning after $10^6$ writes are performed on the cell (this phenomenon is known as cell wear-out).

A bright ETH student has built a computer system using 1GB of ETH-MEM as main memory. ETH-MEM exploits a perfect wear-leveling mechanism, i.e., writes are equally distributed over all of the cells of the main memory.

(a) [25 points] The student is worried about the lifetime of the computer system she has built. She executes a test program that runs special instructions to bypass the cache hierarchy and repeatedly writes data into different words until all the ETH-MEM cells are worn-out (i.e., stop functioning) and the system becomes useless. The student’s measurements show that ETH-MEM stops functioning (i.e., all its cells are worn-out) in one year.

Assume the following:

- The processor is in-order and there is no memory-level parallelism.
- Each write request is fully serialized, i.e., there are three steps in each write request: (1) memory request from CPU to memory controller, (2) write request from memory controller to ETH-MEM, and (3) data write to ETH-MEM cells. None of the steps can be pipelined.
- It takes 5 ns to send a memory request from the processor to the memory controller and it takes 35 ns to send the request from memory controller to ETH-MEM.
- ETH-MEM is word-addressable. Thus, each write request writes 4 bytes to memory.
- For ease of calculation, you can assume that 1GB is equal to $10^9$ bytes.

What is the write latency of ETH-MEM? Show your work. Hint: 1 year $\approx 3 \times 10^{16}$ ns.
\[
t_{\text{wear\_out}} = \frac{\text{1GB}}{4^2} \times 10^6 \times (t_{\text{write}} + 5 + 35)
\]
\[
t_{\text{wear\_out}} = \frac{10^6}{4} \times 10^6 \times (t_{\text{write}} + 5 + 35)
\]
\[
3 \times 10^{16} \text{ns} = \frac{10^9}{4} \times 10^6 \times (t_{\text{write}} + 40)
\]
\[
t_{\text{write}} = \frac{3 \times 10^{16} \times 4}{10^6} - 40 \approx 80 \text{ ns}
\]

**Explanation:**
- Each memory cell will be written 10^6 times.
- Since ETH-MEM is word-addressable, the required number of writes is equal to 10^6 × 10^6 (we have assumed that 1GB is equal to 10^9 bytes).
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to \( t_{\text{write}} + 5 + 35 \).

(b) [20 points] ETH-MEM currently works in the multi-level cell (MLC) mode in which each memory cell stores 2 bits. The student decides to improve the lifetime of ETH-MEM cells by using the single-level cell (SLC) mode. When ETH-MEM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency reduces to 32 ns. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with everything else remaining the same in the system? Show your work.

\[
t_{\text{wear\_out}} = \frac{\text{1GB} \times 0.5}{4^2} \times 10^7 \times (t_{\text{write\_SLC}} + 5 + 35)
\]
\[
t_{\text{wear\_out}} = 0.5 \times 10^9 \times 10^7 \times (32 + 5 + 35)
\]
\[
t_{\text{wear\_out}} = 0.5 \times 10^9 \times 10^7 \times (72)
\]
\[
t_{\text{wear\_out}} = 9 \times 10^{16} \text{ ns} \approx 3 \text{ years}
\]

**Explanation:**
- In SLC mode, each memory cell will be written \( 10 \times 10^6 \) times = \( 10^7 \) writes.
- The memory capacity is reduced by 50% since we are using SLC mode.
  
  \( \text{Capacity} = 0.5 \times 1\text{GB} \)
8 Memory Consistency [60 points]

A programmer writes the following three C code segments. She wants to run them concurrently on a multicore processor, called SC, using three different threads, each of which will run on a different core. The processor implements sequential consistency, as we discussed in the lecture.

<table>
<thead>
<tr>
<th>Thread T0</th>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. T0.0</td>
<td>Instr. T1.0</td>
<td>Instr. T2.0</td>
</tr>
<tr>
<td></td>
<td>$a = 1$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instr. T1.1</td>
<td>$v = c$;</td>
</tr>
<tr>
<td></td>
<td>$c = 1$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instr. T2.1</td>
</tr>
<tr>
<td></td>
<td>$w = a$;</td>
<td></td>
</tr>
</tbody>
</table>

$a, u, c, v, w$ have been allocated in main memory. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a single instruction.

(a) [30 points] What are the possible values of variables $u, v, w$ after the program finishes its execution? Explain your answer by providing the order of execution of instructions for every output.

**Explanation.**

- $(u,v,w) = (0, 0, 0)$: T1.0, T2.0, T2.1
- $(u,v,w) = (0, 0, 1)$: T1.0, T2.0, T0.0, T2.1
- $(u,v,w) = (0, 1, 0)$: T1.0, T1.1, T2.0, T2.1, T0.0
- $(u,v,w) = (0, 1, 1)$: T1.0, T0.0, T1.1, T2.0, T2.1
- $(u,v,w) = (1, 0, 0)$: T2.0, T2.1, T0.0, T1.0, T1.1
- $(u,v,w) = (1, 0, 1)$: T2.0, T0.0, T0.1, T2.1, T1.1
- $(u,v,w) = (1, 1, 0)$: Not sequentially consistent output
- $(u,v,w) = (1, 1, 1)$: T0.0, T1.0, T1.1, T2.0, T2.1

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called WC, that implements weak consistency. As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is no guarantee on the ordering of instructions as seen by different cores.

(b) [15 points] What are the possible final values of variables $u, v, w$ after the program finishes its execution that were not possible with the sequentially consistent memory model? Explain your answer by providing the order of execution of instructions for every output.

- $(u,v,w) = (1, 1, 0)$ can appear in the WC model.

2 scenarios:

1. T2.1, T0.0, T1.0, T1.1, T2.0
2. T1.1, T2.0, T2.1, T0.0, T1.0
After several months spent debugging her code, the programmer learns that the new processor includes a `memory_fence()` instruction in its ISA. The semantics of `memory_fence()` is as follows for a given thread that executes it:

1. Wait (stall the processor) until all preceding memory operations from the thread complete in the memory system and become visible to other cores.

2. Ensure no memory operation from any later instruction in the thread gets executed before the `memory_fence()` is retired.

(c) [15 points] What are the minimal changes should the programmer make to her program to ensure that the possible final values of u, v, w on WC are the same as the ones in part (a) on SC? Explain your answer.

```
Use memory fences between T2.0 and T2.1 as well as T1.0 and T1.1.

Explanation.

<table>
<thead>
<tr>
<th>Thread T0</th>
<th>Thread T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. T0.0 a = 1;</td>
<td>Instr. T1.0 y = a;</td>
</tr>
<tr>
<td></td>
<td>Instr. memory_fence();</td>
</tr>
<tr>
<td></td>
<td>Instr. T1.1 c = 1;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. T2.0 v = c;</td>
</tr>
<tr>
<td>Instr. memory_fence();</td>
</tr>
<tr>
<td>Instr. T2.1 w = a;</td>
</tr>
</tbody>
</table>
```
9 BONUS: Cache Coherence [60 points]

We have a system with four byte-addressable processors \(\{P0, P1, P2, P3\}\). Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. All caches are connected to and actively snoop a global bus, and cache coherence is maintained using the MESI protocol, as we discussed in class. Note that on a write to a cache block in the S state, the block will transition directly to the M state. Accessible memory addresses range from \(0x00000 - 0xffff\).

Each processor executes the following instructions in a sequentially consistent manner:

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>st r0, 0x111A0</td>
<td>3</td>
<td>ld r0, 0x111B0</td>
</tr>
<tr>
<td>1</td>
<td>st r1, 0x00110</td>
<td>4</td>
<td>ld r2, 0xB0070</td>
</tr>
<tr>
<td>2</td>
<td>ld r2, 0xB00F0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>st r0, 0x111A0</td>
<td>6</td>
<td>st r1, 0xB0040</td>
</tr>
<tr>
<td>7</td>
<td>st r0, 0x00110</td>
<td>8</td>
<td>st r1, 0xB0040</td>
</tr>
</tbody>
</table>

After executing the above 9 memory instructions, the final tag store state of each cache is as follows:

**Final Tag Store States**

<table>
<thead>
<tr>
<th>Cache for P0</th>
<th></th>
<th>Cache for P1</th>
<th></th>
<th>Cache for P2</th>
<th></th>
<th>Cache for P3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
</tr>
<tr>
<td>Set 0</td>
<td>0x001</td>
<td>S</td>
<td>Set 0</td>
<td>0x001</td>
<td>I</td>
<td>Set 0</td>
<td>0x001</td>
</tr>
<tr>
<td>Set 1</td>
<td>0xB00</td>
<td>I</td>
<td>Set 1</td>
<td>0xB00</td>
<td>I</td>
<td>Set 1</td>
<td>0xB00</td>
</tr>
<tr>
<td>Set 2</td>
<td>0x111</td>
<td>S</td>
<td>Set 2</td>
<td>0x111</td>
<td>I</td>
<td>Set 2</td>
<td>0x111</td>
</tr>
<tr>
<td>Set 3</td>
<td>0xB00</td>
<td>I</td>
<td>Set 3</td>
<td>0xB00</td>
<td>S</td>
<td>Set 3</td>
<td>0xB00</td>
</tr>
</tbody>
</table>

(a) [25 points] Fill in the following tables with the initial tag store states (i.e., Tag and MESI state) before having executed the nine memory instructions shown above. Answer X if a tag value is unknown, and for the MESI states, write in all possible values (i.e., M, E, S, and/or I).

**Initial Tag Store States**

<table>
<thead>
<tr>
<th>Cache for P0</th>
<th></th>
<th>Cache for P1</th>
<th></th>
<th>Cache for P2</th>
<th></th>
<th>Cache for P3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
<td>Tag</td>
<td>MESI state</td>
</tr>
<tr>
<td>Set 0</td>
<td>X</td>
<td>M, E, S, I</td>
<td>Set 0</td>
<td>0x001</td>
<td>M, E, S, I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>0xB00</td>
<td>M, E, S, I</td>
<td>Set 1</td>
<td>X</td>
<td>M, E, S, I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td>X</td>
<td>M, E, S, I</td>
<td>Set 2</td>
<td>X</td>
<td>M, E, S, I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>0xB00</td>
<td>I</td>
<td>Set 3</td>
<td>0xB00</td>
<td>I</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(b) [25 points] In what order did the memory operations enter the coherence bus? Show your work.

| time → | 3 | 4 | 0 | 1 | 2 | 7 | 8 | 5 | 6 |

Instruction 1 should be before instruction 7, instructions 4 and instruction 8 should be before 6 but the order of instruction 4 and instruction 8 is not important, instruction 3 should be before instruction 0 and instruction 0 should be before instruction 5.

(c) [10 points] What is the minimum number of writebacks needed for this instruction sequence? Explain your answer.

Three. After 0, 1, and 8, related cache lines are requested by other processors, and this causes writebacks to the main memory.
10 **BONUS: Interconnects** [90 points]

Suppose you would like to connect 2048 processors, and you are considering two different interconnection network topologies:

- Bi-directional Ring
- Hypercube

Please answer the following questions. Show your work.

(a) [10 points] Calculate the number of network links for each of these two topologies. (Hint: a single network link is bi-directional)

<table>
<thead>
<tr>
<th>Topology</th>
<th>Links Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>$2048$</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$2^N \times \frac{N}{2}$ links $\rightarrow 2048 \times \frac{11}{2} = 11264$ links</td>
</tr>
</tbody>
</table>

(b) [10 points] Calculate the maximum routing distance within the network (i.e., network diameter) in terms of hops for each of these two topologies.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Hops Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>$\frac{2048}{2} = 1024$ hops</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$\log(2048) = 11$ hops</td>
</tr>
</tbody>
</table>

(c) [15 points] Assume a network link can be faulty. For each topology, what is the minimum possible number of faulty links that are needed to make at least one processor unreachable from any other processor?

<table>
<thead>
<tr>
<th>Topology</th>
<th>Links Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>2 links</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$\log(2048)$ links $\rightarrow 11$ links</td>
</tr>
</tbody>
</table>
You realize that the cost of hypercube topology is not acceptable for connecting 2048 processors. After a lot of brainstorming with your colleagues, you come up with the nice idea of combining ring and hypercube topologies, such that each node of an $n$-dimensional hypercube topology is replaced by a bi-directional ring topology with $n$ nodes. You name your topology *Cube-Connected Ring* with the following formulation:

- The cube-connected ring of dimension $n$ is a graph of $n \times 2^n$ nodes.
- Each node is represented by pairs of $(x, y)$ where $0 \leq x < 2^n$ and $0 \leq y < n$.
- Each node $(x, y)$ is connected to three neighbors: $(x, (y + 1) \mod n)$, $(x, (y - 1) \mod n)$ (two neighbors in the ring), and $(x \oplus 2^n, y)$ (one neighbor in the hypercube), where $\oplus$ is bitwise exclusive or (XOR) operation.

(d) [30 points] Draw the cube-connected ring topology for dimension $n = 4$. Clearly show the nodes’ addresses, i.e., $(x, y)$, and the links.
(e) [15 points] Calculate the number of network links needed to connect 2048 processors using the cube-connected ring topology. (Hint: a single network link is bi-directional)

\[ n = 8, \text{#links} = \frac{2048 \times 3}{2} = 3072 \]

(f) [10 points] Calculate the diameter of the cube-connected ring topology that connects 2048 processors.

\[ n = 8, \text{e.g., src}=(0,0) \land \text{dest}=(255,4) \]
\[ (0,0) - (1,0) - (1,1) - (3,1) - (3,2) - (7,2) - (7,3) - (15,3) - (15,4) - (31,4) - (31,5) - (63,5) - (63,6) - (127,6) - (127,7) - (256,7) - (256,6) - (256,5) - (256,4) \rightarrow \text{Diameter} = 18 \]