Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Computer Architecture (227-2210-00L)

ETH Zürich, Fall 2020

Prof. Onur Mutlu

Problem 1 (175 Points):	Potpourri	
Problem 2 (105 Points):	DRAM Refresh	
Problem 3 (90 Points):	RowHammer	
Problem 4 (100 Points):	Processing-using-Memory	
Problem 5 (70 Points):	Emerging Memory Technologies	
Problem 6 (100 Points):	Prefetching	
Problem 7 (100 Points):	Cache Coherence	
Problem 8 (BONUS: 90 Points):	Genome Analysis	
Total (740 + 90 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 Potpourri [175 points]

A multiple choice question can potentially have multiple correct choices. 4 incorrectly-answered questions will cause you to lose 1 correctly-answered question (you can never go below 0, though, in the points you gain from this question). Choose wisely.

(a) **[8 points] In lecture, we covered the Bloom Filter data structure. As you should recall, Bloom Filter represents set membership. Which of the following is NOT correct about Bloom Filters?**

1) It has no false negatives.
2) It has false positives.
3) It is approximate.
4) It can be used in both hardware and software.
5) [✓ In the best case, its false positive rate is 100%.]

(b) **[8 points] Consider the following statements about different memory technologies. Pick the one that is NOT correct.**

1) DRAM is charge based memory.
2) Phase Change Memory is resistive memory.
3) Charge based memories are fundamentally less scalable than resistive memories (based on our current understanding of them).
4) [✓ Flash memory is resistive memory.]
5) Both Flash memory and Phase Change Memory are non-volatile memories.

(c) **[8 points] If activating row A causes some bits in row B to flip (due to RowHammer), then does activating row B cause some bits in row A to flip as well?**

1) No, never.
2) Yes, always.
3) [✓ Not necessarily.]
4) Only the bits that contain '1' in row A will be flipped to '0'.
5) Only the bits that contain '0' in row A will be flipped to '1'.

(d) **[8 points] DRAM latency has improved much less than capacity or bandwidth in recent years. The main inhibitor of DRAM latency is the long subarray bitlines. What is the main reason manufacturers do not reduce the latency?**

1) [✓ Shorter subarray bitlines result in larger chip area and higher module costs.]
2) Long subarray bitlines are necessary due to the manufacturing process technology limitation.
3) Long bitlines require an extremely large sense amplifier for fast sensing. DRAM technology does not allow to for a faster sense amplifier structure.
4) Reducing latency requires considerable engineering effort.
5) Latency is dependent on the memory controller design, which was not redesigned in recent years.

(e) [8 points] **What makes it very difficult to determine the retention time of a DRAM cell?**

1) Retention time of the cell can change randomly over time, due to the variable retention time phenomenon.
2) Retention time of the cell is dependent on the value stored in the cell and the cells around it.
3) The RowHammer phenomenon negatively affects the retention time of the cell.
4) [✓ Only 1) and 2).]
5) 1), 2), and 3).

(f) [8 points] **In lecture, we covered the idea of Heterogeneous Reliability Memory (HRM), where part of physical memory is built using very reliable DRAM chips whereas other parts are built using unreliable DRAM chips. The system intelligently partitions the data between the two types of memories. What is the fundamental property that HRM relies on?**

1) Even unreliable DRAM chips provide some amount of error correction.
2) [✓ Some data in a given application can tolerate the memory errors that may happen in unreliable DRAM chips]
3) The operating system can automatically detect and correct memory errors that happen in unreliable DRAM chips.
4) Only 1) and 2).
5) Only 2) and 3).

(g) [8 points] **Why is a DRAM cell faster to access at low temperatures compared to at high temperatures?**

1) DRAM cell loses charge quickly at low temperatures, which causes it to be refreshed more frequently, which, in turn, reduces the access latency of the cell.
2) [✓ DRAM cell stores more charge at low temperatures, which leads to faster sensing.]
3) DRAM cell is more reliable at low temperature, which enables it to be accessed faster since there is less need for error correction.
4) DRAM cell is less vulnerable to RowHammer at low temperatures, making it faster to access.
5) The question starts from a wrong axiom. DRAM cell is not faster to access at low temperatures.

(h) [8 points] **In lecture, we covered the concept of thread ranking, as a fundamental building block of modern memory controllers. The idea, as you should recall, is to rank the threads based on some characteristics, and use the ranking as a prioritization order between requests of different threads across all banks and memory controllers. Which of the following statements is NOT correct about thread ranking?**

1) Ranking a low-memory-intensity thread over a high-memory-intensity thread improves system throughput.
2) Ranking is complex to implement, compared to a baseline thread-unaware memory scheduler.
3) Ranking helps preserve bank-level parallelism of threads.
4) [✓ Ranking provides starvation freedom to different threads.]
5) Ranking can be determined by system software, if the interfaces to the memory controller are provided.

(i) [8 points] **We covered the idea of the Blacklisting Memory Scheduler, which essentially de-prioritizes the requests of an application that has recently been serviced by the memory scheduler with a series of consecutive requests. Basically, the application gets blacklisted for a short time. The rest of the scheduling policy is similar to baseline First-Ready First-Come-First-Serve (FR-FCFS) policy of modern memory schedulers. Which of the following is NOT correct about the Blacklisting Memory Scheduler?**

1) Blacklisting is simpler to implement than ranking of applications.
2) It adapts quickly to changing memory access behavior of different threads.
3) [✓ It explicitly tries to preserve bank-level parallelism of each thread.]
4) It takes into account row buffer locality in scheduling decisions.
5) It does not take into account requests coming from accelerators with strict QoS requirements.

(j) [8 points] **We discussed the idea of identifying "limiter threads" in a multithreaded application in order to prioritize them in the memory scheduler. Which of the following can be considered a limiter thread?**

1) A thread that is holding a contended lock.
2) A thread that arrives late at a barrier synchronization point.
3) [✓ A thread that is executing a pipeline stage with the lowest throughput.]
4) Only 1) and 2).
5) 1), 2), and 3).

(k) [8 points] **Which of the following interference control techniques can fundamentally reduce the load (i.e., number of outstanding requests) on the entire memory system?**

1) Application-aware memory request scheduling.
2) Application-aware data mapping.
3) [✓ Application-aware source throttling.]
4) Only 2) and 3).
5) 1), 2), and 3).

(l) [8 points] **Assume a toy system that has 32,000,000 bytes of DRAM. Assume each refresh consumes 1 milliWatts (i.e., 10^{-3} Watts). If you are told that the total power consumption spent on refresh is 320 Watts, what can you conclude about the DRAM system?**

1) The row size is 10 bytes.
2) [✓ The row size is 100 bytes.]
3) The row size is 1000 bytes.
4) The row size is 10,000 bytes.
5) The row size is not possible to determine.

(m) [8 points] **In lecture, we covered the idea of accelerating serialized code portions by shipping them to powerful cores in an asymmetric multicore system. Which of the following is NOT one of the key benefits bottleneck acceleration provides in an asymmetric multicore architecture?**

1) It lowers the burden on the programmer for parallel code optimization.
2) It reduces serialization due to contended locks.
3) It improves lock locality.
4) [✓ It increases the number of available parallel threads.]
5) It reduces the performance impact of hard-to-parallelize code sections.

(n) [8 points] **To improve the performance of a program, we decide to split the program code into segments and run each segment on the most suitable core to run it. Which of the following can this approach NOT achieve by itself?**

1) It can accelerate segments/critical paths using specialized/heterogeneous cores.
2) It can exploit inter-segment parallelism.
3) [✓ It can improve the locality of inter-segment data.]
4) It can improve the locality of within-segment data.

(o) [8 points] **In a runahead execution processor, runahead mode is used to:**

1) [✓ Tolerate memory latency.]
2) Increase computational parallelism.
3) Increase power efficiency.
4) All of the above.

(p) [8 points] **Entering runahead mode in an out-of-order runahead execution processor requires checkpointing:**

1) The store buffer.
2) The L1 instruction cache.
3) [✓ The register file.]
4) All of the above.

(q) [8 points] **Exiting runahead mode in an out-of-order runahead execution processor requires:**

1) [✓ Flushing the pipeline.]
2) Flushing the branch predictor tables.
3) Flushing the prefetcher tables.
4) All of the above.

(r) [8 points] **We covered the design of the Tesseract system for graph processing. Recall that Tesseract exploits the logic layer in 3D-stacked memory to perform graph processing computations. Which one of the following is NOT true about Tesseract?**

1) The system is programmed using message passing.
2) The system makes use of aggressive prefetching.
3) [✓ The system provides cache coherence between the CPU cores and the computation logic in the logic layer of 3]
4) The system uses simple in-order cores in the logic layer of 3D-stacked memory.
5) The system exposes a very large amount of memory bandwidth to the cores.

(s) [8 points] **Name the three fundamental reasons as to why the parallel portion of a parallel program is NOT perfectly parallel.**

1) Synchronization overhead (e.g., updates to shared data)
2) Load imbalance overhead (imperfect parallelization)
3) Resource sharing overhead (contention among N processors)

(t) [15 points] **Consider the following statement: "A sequentially consistent multiprocessor guarantees that different executions of the same multithreaded program produce the same architecturally-exposed ordering of memory operations."**

1) Is this statement true or false? CIRCLE ONE.

   1. `True` [✓ 2. `False`]

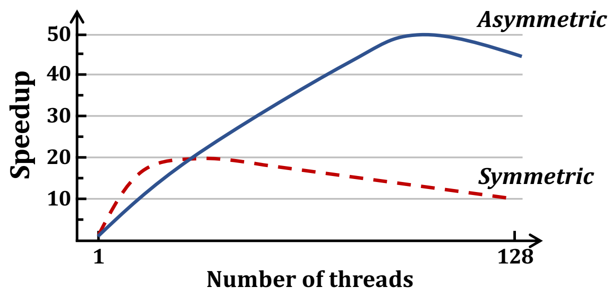
2) Explain your reasoning (less than 15 words).

   Sequential consistency makes no guarantees across different executions. (It is about the ordering of operations within the same execution)

3) Why do we want the property described above; i.e., the property that "different executions of the same multithreaded program produce the same architecturally-exposed ordering of memory operations"?

   Debugging ease

(u) **[8 points] The figure below shows the speedup curve for a workload on two systems: symmetric multicore and asymmetric multicore. Assume an area budget of 128 small cores.**



What is the performance improvement of the asymmetric multicore over the symmetric one?

$50/20 = 5/2 = 2.5$

2 DRAM Refresh [105 points]

2.1 Basics [30 points]

A memory system is composed of eight banks, and each bank contains 2^{16} rows. Every DRAM row refresh is initiated by a command from the memory controller, and it refreshes a single row in a single DRAM bank. Each refresh command keeps the command bus busy for 5 ns. We define *command bus utilization* as the fraction of total execution time during which the command bus is occupied.

1. [10 points] Given that the refresh interval is 64ms, calculate the command bus utilization of refresh commands. Show your work step-by-step.

Command bus is utilized for $8 \times 2^{16} \times 5ns$ at every 64ms.
$Utilization = (2^{19} \times 5ns)/(2^{6} \times 10^{6}ns) = 2^{13}/(2 \times 10^{5}) = 2^{12} \times 10^{-5} = 4.096\%$

2. [20 points] Now assume 60% of all rows can withstand a refresh interval of 128 ms. If we are able to customize the refresh rate of each row independently, up to how much could we reduce the command bus utilization of refresh commands? Calculate the reduction $(1 - \frac{new}{old})$ in bus utilization. Show your work step-by-step.

At every 128ms:
- 60% of the rows are refreshed once.
  Command bus is busy for: $0.6 \times 8 \times 2^{16} \times 5ns = 3 \times 2^{19}ns$
- 40% of the rows are refreshed two times.
  Command bus is busy for: $0.4 \times 8 \times 2^{16} \times 5ns \times 2 = 4 \times 2^{19}ns$

$Utilization = (3 + 4) \times 2^{19}ns/128ms = 7 \times 2^{12} \times 10^{-6}$

$Reduction = 1 - (0.7 \times 2^{12} \times 10^{-5})/(2^{12} \times 10^{-5}) = 30\%$

2.2 VRL: Variable Refresh Latency [75 points]

In this question, you are asked to evaluate "Variable Refresh Latency," proposed by Das et al. in DAC 2018.[1]

The paper presents two key observations:

- First, a cell's charge reaches 95% of the maximum charge level in 60% of the nominal latency value during a refresh operation. In other words, the last 40% of the refresh latency is spent to increase the charge of a cell from 95% to 100%. Based on this observation, the paper defines two types of refresh operations: (1) *full refresh* and (2) *partial refresh*. Full refresh uses the nominal latency value and restores the cell charge to 100%, while the latency of partial refresh is only 60% of the nominal latency value and it restores 95% of the charge.

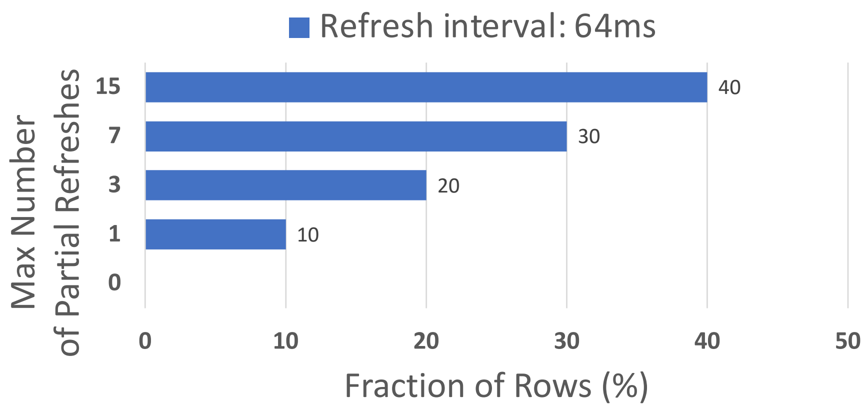
[1]Das, A. et al., *"VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency."* In Proceedings of the 55th Annual Design Automation Conference (DAC), 2018.

- Second, a fully refreshed cell operates correctly even after multiple partial refreshes, but it needs to be fully refreshed again after a finite number of partial refreshes. The maximum number of partial refreshes before a full refresh is required varies from cell to cell.

The **key idea** of the paper is to apply a *full refresh* operation **only when necessary** and use *partial refresh* operations at all other times.

(a) [25 points] Consider a case in which:

- Each row must be refreshed every 64 ms. In other words, the refresh interval is 64 ms.
- Row refresh commands are evenly distributed across the refresh interval. In other words, all rows are refreshed exactly once in any given 64 ms time window.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank. For example, if the maximum number of refreshes is three, those rows can be partially refreshed for at most three refresh intervals, and the fourth refresh operation must be a full refresh.
- If all rows were always fully refreshed, the time that a bank is busy serving the refresh requests within a refresh interval would be T.



How much time does it take (in terms of T) for a bank to refresh all rows within a refresh interval, after applying Variable Refresh Latency?

Full refresh latency = T, partial refresh latency = 0.6T.

10% of the rows are fully refreshed at every other interval:
$0.1 \times (1/2 \times 0.6T + 1/2 \times T)$
20% of the rows are fully refreshed after every three partial refresh:
$0.2 \times (3/4 \times 0.6T + 1/4 \times T)$
30% of the rows are fully refreshed after every seven partial refresh:
$0.3 \times (7/8 \times 0.6T + 1/8 \times T)$
40% of the rows are fully refreshed after every fifteen partial refresh:
$0.4 \times (15/16 \times 0.6T + 1/16 \times T)$

Then, new refresh latency of a bank would be 0.665T.

(b) [25 points] You find out that you can relax the refresh interval, and define your baseline as follows:

- 75% of the rows are refreshed at every 128ms; 25% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- All rows are always fully refreshed.
- A single refresh command costs $0.2/N\ ms$, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.

Calculate the refresh overhead for the baseline.

At every 128ms:

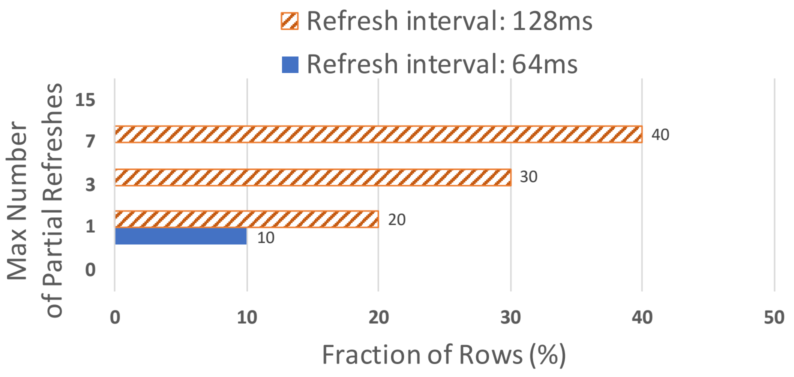
25% of the rows are refreshed twice, 75% of the rows are refreshed once.

Total time spent for refresh in a 128 ms. interval is $(0.75N + 2 \times 0.25N) \times 0.2/N = 0.25ms$.

Then refresh overhead is 0.25/128

(c) [25 points] Consider a case where:

- 90% of the rows are refreshed at every 128ms; 10% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank.
- A single refresh command costs $0.2/N$ *ms*, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.



Calculate the refresh overhead. Show your work step-by-step. Then, compare it against the baseline configuration (part b). How much reduction ($1 - \frac{new}{old}$) do you see in the performance overhead of refreshes?

Full refresh of a row costs 0.2/N ms. Then, partial refresh of a row costs 0.12/N ms

At every 8×128 *ms*:

- 10% of the rows are refreshed 16 times:
  8 times *fully refreshed* and 8 times *partially refreshed.*
- 20% of the rows are refreshed 8 times:
  4 times *fully refreshed* and 4 times *partially refreshed.*
- 30% of the rows are refreshed 8 times:
  2 times *fully refreshed* and 6 times *partially refreshed.*
- 40% of the rows are refreshed 8 times:
  1 time *fully refreshed* and 7 times *partially refreshed.*

The Total time spent for refresh is:
$= (0.1N \times 8 + 0.2N \times 4 + 0.3N \times 2 + 0.4N \times 1) \times 0.2/N$
$+ (0.1 \times 8 + 0.2N \times 4 + 0.3N \times 6 + 0.4N \times 7) \times 0.12/N$

$= (0.8 + 0.8 + 0.6 + 0.4) \times 0.2 + (0.8 + 0.8 + 1.8 + 2.8) \times 0.12$
$= 0.52 + 0.744 = 1.264ms$
Then, refresh overhead is $1.264/(8 \times 128)$
So, the reduction is $1 - (1.264/8)/0.25 \approx 36.8\%$.

3 RowHammer [90 points]

3.1 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 point.)*

(a) [3 points] Violating DRAM timing parameters is necessary to induce RowHammer bit flips.

1. `True` 2. [✓ `False`]

(b) [3 points] SECDED (Single Error Correction Double Error Detection) Hamming ECC cannot guarantee RowHammer-safe operation.

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] We can more effectively induce bit flips in a given victim row by hammering rows in different banks.

1. `True` 2. [✓ `False`]

(e) [3 points] In DRAM TRR (Target Row Refresh) mechanism provides RowHammer-safe operation. The only problem is that it is not implemented in all DRAM chips.

1. `True` 2. [✓ `False`]

3.2 RowHammer Mitigation [75 points]

You are assigned to implement a RowHammer mitigation mechanism within the memory controller of a new processor. The DRAM chips that will be used with the processor are organized as 1 channel, 1 rank, 8 banks, 8 KB row size, and 1 GB total capacity. The DRAM protocol specifies that each 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 RowHammer threshold of the chips is 50,000 row activations per aggressor row during a double-sided attack, while no hammer count for a single-sided attack can induce bit flips within 64 ms.

(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. $N_{ACT} = 64ms/64ns = 10^6$ row activations.

Each aggressor row should be activated 50000 times. Let N_{aggr} be the maximum number of aggressor rows that can be concurrently hammered and N_{RH} be the RowHammer threshold.

Then, $N_{aggr} = N_{ACT}/N_{RH} = 10^6/(5.10^4) = 20$ aggressor rows.

(b) [20 points] What is the maximum number of victim rows that can be affected in a bank? Show your work and explain clearly.

As single-sided attacks fail, we should only count the rows whose both adjacent neighbors are aggressor rows. In a double-sided attack every victim row is attacked through two aggressor rows. This scheme can be shown as AVA, where each A represents an aggressor row and each V represents a victim row. An attacker can concurrently hammer 20 aggressor rows (see part a).

Then, the total number of victim rows is 10 when naively performing a double-sided attack.

However, to maximize the number of victim rows we can perform the many-sided attack that TRRespass paper proposes (...$AVAVA$...). In this scheme, an attacker can attack to N_{vict} victim rows by hammering N_{aggr} aggressor rows, where $N_{aggr} = N_{vict} + 1$.

Therefore, the answer is $20 - 1 = 19$

(c) [20 points] To detect a RowHammer attack, you consider using existing hardware performance counters. Sort the following performance metrics from *the most accurate* to *the least accurate* as an indicator of an ongoing RowHammer attack. Explain your reasoning clearly. You may not get points without explanation even if the ordering is correct.

– AM: Available Memory: The amount of physical memory, free and available to processes.

– RBHR: Row Buffer Hit Rate: The fraction of requests that hit the row buffer

– LLC MPKI: Last Level Cache Misses per Kilo Instructions

– RBCPKI: Row Buffer Conflicts per Kilo Instructions

$RBCPKI > RBHR > LLCMPKI > AM$

RBCPKI: This is the most related one because a RowHammer attack needs to create as many row buffer conflicts as possible within a limited time window.

RBHR: Row buffer hit rate gives an idea about the row buffer conflicts. However, a low row buffer hit rate can be misleading if the memory is mostly idle.

LLC MPKI: This metric measures the memory access intensity of concurrent workloads. However, it does not provide any information about the rate of row activations, which is essential for RowHammer attacks.

AM: The memory footprint of a RowHammer attack can be as small as a few pages. Therefore, the amount of available physical memory does not give a hint about a RowHammer attack.

(d) [15 points] As the RowHammer mitigation mechanism's designer, you have a chance to change the memory controller's row policy. From the perspective of RowHammer mitigation, which page policy would you choose? Please mark below and explain.

Open-Row Policy Closed-Row Policy [✓ Does Not Matter]

The memory controller's row policy does not matter for the RowHammer mitigation mechanism because in both open and closed policies the attacker is able to issue the same activation rate.

If the DRAM chip was also vulnerable to single-sided RowHammer attack, then closed-row policy would have allowed the attacker to reach the target hammer count faster than open-row policy because the attacker would not need to activate a different row between two activations targeting the single aggressor row to be able to create a row buffer conflict. That being said, a designer might prefer open-row policy to increase the probability of a RowHammer access experiencing a row buffer hit, which is likely in attacks where (1) the attacker is not able to precisely time every request (e.g., when used clflush) or (2) accesses from other applications can hit the agggressor row, causing some RowHammer accesses to experience row buffer hits.

4 Processing-using-Memory [100 points]

One promising trend in the Processing-in-Memory paradigm is Processing-using-Memory (PuM), which exploits the analog operation of memory cells to execute bulk bitwise operations. A pioneering proposal in PuM in DRAM technology is Ambit, which we discussed in class. Ambit provides the ability to perform bitwise AND/OR of two rows in a subarray and NOT of one row. Since Ambit is logically complete, it is possible to implement any other logic gate (e.g., XOR). However, to be able to implement arithmetic operations (e.g., addition), bit shifting is also necessary. There is no way of shifting bits in DRAM with a conventional layout, but there are two possible approaches to modifying DRAM to enable bit shifting.

The first approach uses a bit-serial layout (i.e., it changes the horizontal layout to vertical), as Figure 1(a) shows. With such a layout, it is possible to perform **bit-serial** arithmetic computations inside DRAM. For example, performing an addition in a bit-serial manner only requires XOR, AND, and OR operations, as the 1-bit full adder in Figure 1(b) shows.

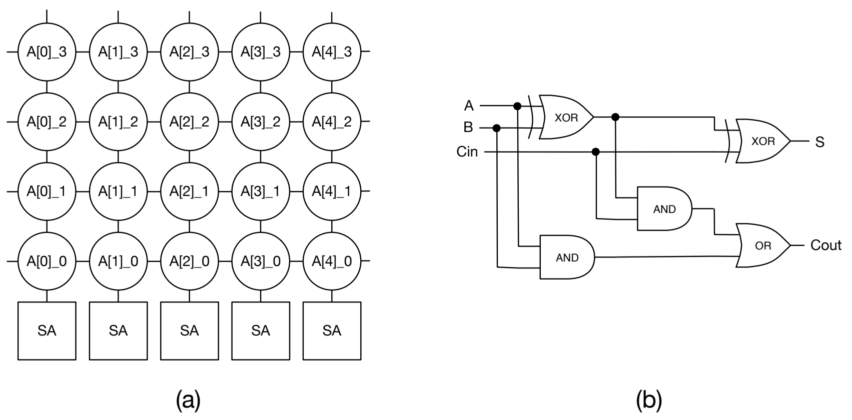


Figure 1: (a) In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`. (b) 1-bit full adder.

The second approach uses the conventional horizontal layout, but extends the DRAM subarray with *shifting lines*, which connect each bitline to the previous sense amplifier (SA) to enable left shifting. Figure 2(a) illustrates the second approach, where dashed lines represent the shifting lines. With such shifting lines, it is possible to perform **bit-parallel** arithmetic computations inside DRAM. For example, an addition can be performed in a bit-parallel manner using a parallel adder, such as the Kogge-Stone adder. Figure 2(b) shows an example of an 8-bit Kogge-Stone adder.

We want to compare the potential performance of both approaches to arithmetic computation by implementing a simple workload, the element-wise addition of two arrays. Listing 1 shows a sequential code for the addition of two input arrays `A` and `B` into output array `C`.

Listing 1: Sequential CPU implementation of element-wise addition of arrays `A` and `B`.

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

As you know from lectures and homeworks, Ambit implements bitwise operations by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) commands. For example, to compute AND, OR, and XOR operations, Ambit issues the sequence of commands described in Figure 3, where `AAP(X,Y)` represents two consecutive activations of two row addresses `X` and `Y` (each of which may correspond to 1, 2, or 3 rows) followed by a precharge operation, and `AP(X)` represents one activation of row address `X` followed by a precharge operation.

In those instructions, Ambit copies the source rows `Di` and `Dj` to auxiliary row addresses (`Bi`). Some of the auxiliary row addresses (e.g., `B12` in Figure 3) correspond to 3 rows, enabling Triple-Row Activation

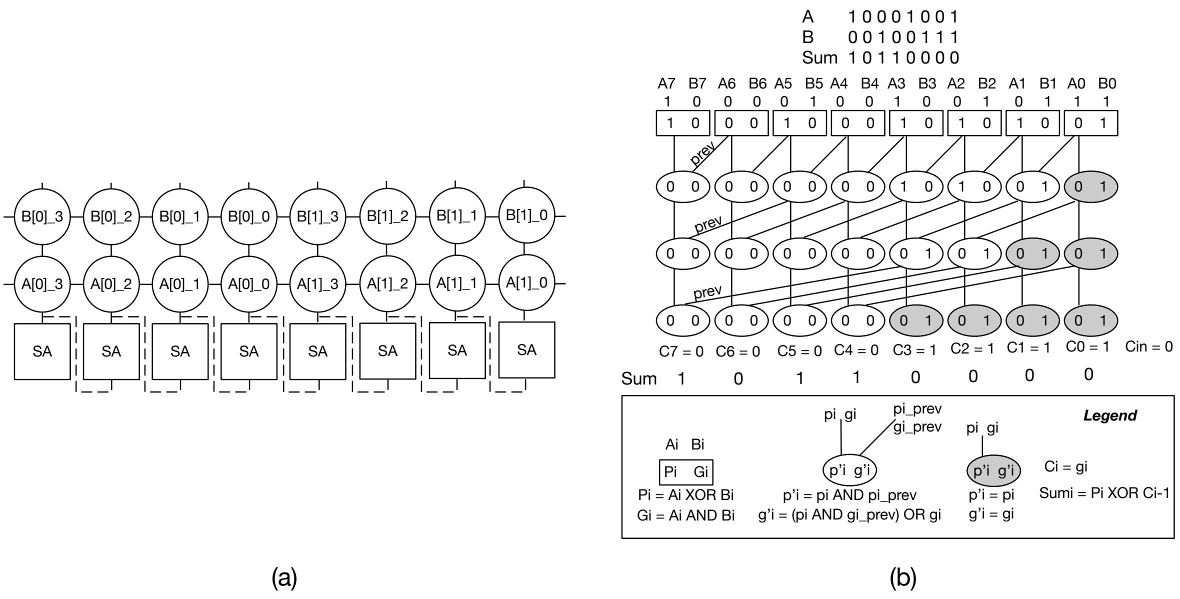


Figure 2: (a) In-DRAM bit-parallel layout for arrays `A` and `B`, which contain two 4-bit elements each. DRAM cells in the same bitline contain the same bits of equal-index elements of different arrays. `A[i]_j` represents bit `j` of element `i`. (b) Example of an 8-bit Kogge-Stone adder. A (10001001) and B (00100111) are the two input operands.

a) Dk = Di **and** Dj

```
AAP (Di,  B0) ;T0 = Di
AAP (Dj,  B1) ;T1 = Dj
AAP (C0,  B2) ;T2 = 0
AAP (B12, Dk) ;Dk = T0 & T1
```

b) Dk = Di **nand** Dj

```
AAP (Di,  B0) ;T0 = Di
AAP (Dj,  B1) ;T1 = Dj
AAP (C0,  B2) ;T2 = 0
AAP (B12, B5) ;DCC0 = !(T0 & T1)
AAP (B4,  Dk) ;Dk = DCC0
```

c) Dk = Di **xor** Dj
= (Di & !Dj) | (!Di & Dj)

```
AAP (Di,  B8)  ;DCC0 = !Di, T0 = Di
AAP (Dj,  B9)  ;DCC1 = !Dj, T1 = Dj
AAP (C0,  B10) ;T2 = T3 = 0
 AP (B14)      ;T1 = DCC0 & T1
 AP (B15)      ;T0 = DCC1 & T0
AAP (C1,  B2)  ;T2 = 1
AAP (B12, Dk)  ;Dk = T0 | T1
```

or/nor/xnor can be implemented by appropriately modifying the control rows of and/nand/xor.

Figure 3: Command sequences for different bitwise operations in Ambit. Notice that AND and OR need the same sequence of commands. Reproduced from Seshadri et al., MICRO 2017.

(TRA), which is the basic operation in Ambit. 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`, flips all bits of the row, and stores the result in both the source row and `DCCi`. Assume that:

- The DRAM row size is 8 Kbytes.
- An ACTIVATE command takes 20ns to execute.
- A PRECHARGE command takes 10ns to execute.
- DRAM has a single memory bank.
- Arrays `A`, `B`, and `C` are properly aligned in both bit-serial and bit-parallel approaches.
- In the bit-parallel approach, a shift operation by one bit requires one `AAP`.

(a) [40 points] Compute the maximum throughput in terms of addition operations per second (OPS) of the bit-serial approach as a function of the element size (i.e., bits/element).

$Throughput_{bit-serial} = \frac{65{,}536}{1220 \times n} GOPS.$

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 8KB, 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 12 ACTIVATION and 7 PRECHARGE operations. For AND and OR, 8 ACTIVATION and 4 PRECHARGE operations. Thus, the execution time of the bit-serial computation on one DRAM subarray can be computed as (n is the number of bits per element):
$t_{bit-serial} = (2 \times t_{XOR} + 2 \times t_{AND} + t_{OR}) \times n;$
$t_{bit-serial} = 1220 \times n$ ns.

Second, we obtain the throughput in arithmetic operations per second (OPS) as:
$Throughput_{bit-serial} = \frac{65{,}536}{1220 \times n \times 10^{-9}} = \frac{65{,}536}{1220 \times n} GOPS.$

(b) [35 points] Compute the maximum throughput in terms of addition operations per second (OPS) of the bit-parallel approach as a function of the element size (i.e., bits/element). Hint: $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$.

$Throughput_{bit-parallel} = \frac{65{,}536}{(820 + log\ n \times (500 + 100 \times n)) \times 10^{-9} \times n} GOPS.$

Explanation:
This approach requires $log\ n$ iterations. Before the first iteration (iteration 0), one XOR and one AND are executed. After the last iteration (iteration $(log\ n) - 1$), one XOR is executed. In each iteration, two AND and one OR are executed. It is also necessary to shift p_i_prev and g_i_prev by an amount that depends on the iteration number: in iteration 0, we shift 1 bit; in iteration 1, 2 bits; in iteration 2, 4 bits... Thus, the shift amount is 2^i, where i is the iteration number.
First, we obtain the execution time as a function of the number of bits per element (n). The execution time of the bit-parallel computation on one DRAM subarray can be computed as:
$t_{bit-parallel} = (2 \times t_{XOR}) + t_{AND} + log\ n \times [2 \times t_{AND} + t_{OR} + \sum_{i=0}^{(log\ n)-1} (2^i \times 2 \times t_{SHIFT})];$
$t_{bit-parallel} = 820 + log\ n \times (600 + 100 \times (n-1)) = 820 + log\ n \times (500 + 100 \times n)$ ns.

Second, we obtain the throughput in arithmetic operations per second (OPS). We should take into account that the number of elements per DRAM row is also a function of n, i.e., $\frac{65{,}536}{n}$:
$Throughput_{bit-parallel} = \frac{65{,}536/n}{(820 + log\ n \times (500 + 100 \times n)) \times 10^{-9}} = \frac{65{,}536}{(820 + log\ n \times (500 + 100 \times n)) \times 10^{-9} \times n} GOPS.$

(c) [25 points] Determine the element size (in bits) for which one approach (i.e., bit-serial or bit-parallel) is preferred over the other one.

There is no number of bits per element (greater than 1) that makes the bit-parallel approach faster than the bit-serial approach.

Explanation:
We want to find n such that $Throughput_{bit-serial} < Throughput_{bit-parallel}$. If we use consider one DRAM subarray:
$Throughput_{bit-serial} < Throughput_{bit-parallel}$;
$\frac{65,536}{1220 \times n} < \frac{65,536}{(820 + log\ n \times (500 + 100 \times n)) \times n}$; $1220 > 820 + log\ n \times (500 + 100 \times n)$; $400 > log\ n \times (500 + 100 \times n)$;
There is no n greater than 1 that makes the bit-parallel implementation higher throughput than the bit-serial implementation.

5 Emerging Memory Technologies [70 points]

5.1 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 point.)*

(a) [3 points] Phase Change Memory (PCM) is more technology-scalable than DRAM.

1. [✓ `True`] 2. `False`

(b) [3 points] PCM read/write operations are more energy efficient than DRAM read/write operations.

1. `True` 2. [✓ `False`]

(c) [3 points] PCM provides shorter access latency but has lower endurance compared to NAND flash memory.

1. `True` 2. [✓ `False`]

(d) [3 points] Row-buffer hit latencies of DRAM and PCM are comparable.

1. [✓ `True`] 2. `False`

(e) [3 points] Row-buffer miss penalty is smaller in PCM than in DRAM, since PCM commonly employs a small row buffer.

1. `True` 2. [✓ `False`]

5.2 SAFARI-RAM [55 points]

Researchers in the SAFARI Research Group developed a new non-volatile memory technology, SAFARI-RAM. The read and write latency of SAFARI-RAM is close to that of DRAM while providing higher memory density compared to the latest DRAM technologies. However, SAFARI-RAM has one shortcoming: it has limited endurance, i.e., a memory cell fails after 10^7 writes are performed to the cell (known as *cell wear-out*).

A bright ETH student has built a computer system using SAFARI-RAM as the main memory. SAFARI-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes across all of the cells of the main memory.

The student wants to estimate the worst-case lifetime of SAFARI-RAM when used as main memory. The student executes a test program to wear out the entire SAFARI-RAM *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 SAFARI-RAM cells are worn out. The student's measurements show that SAFARI-RAM stops functioning (i.e., all its cells are worn-out) in 2.5 years. Assume the following:

- The processor is in-order.
- There is no memory-level parallelism (i.e., there is a single bank in the memory system).
- It takes 32 ns to send a memory request from the processor to the memory controller.
- It takes 52 ns to send the request from the memory controller to SAFARI-RAM.
- The write latency of SAFARI-RAM is 172 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 SAFARI-RAM, and (3) data write to SAFARI-RAM cells. None of the steps can be pipelined.
- SAFARI-RAM requests are issued at *page* granularity, where the page size is 4,096 bytes (4 KiB).
- SAFARI-RAM adopts a quad-level cell (QLC) technique that stores four bits in a single memory cell.

(a) [30 points] What is the capacity of SAFARI-RAM? Show your work. *Hint:* 2.5 years $\approx 8 \times 10^{16}$ ns.

128 GB

Explanation:

- Each memory cell should serve 10^7 writes.
- Since SAFARI-RAM performs writes at page granularity, the required number of writes is equal to $\frac{capacity}{2^{12}} \times 10^7$.
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to 32+52+172 = 256 ns.

$$\therefore \frac{capacity}{2^{12}} \times 10^7 \times 256 = 8 \times 10^{16}$$

$$capacity = \frac{2^{15} \times 10^{16}}{2^8 \times 10^7}$$

$$= 2^7 \times 10^9$$

(b) [25 points] The student decides to improve the lifetime of SAFARI-RAM cells by using the single-level cell (SLC) mode in which a single memory cell stores a single bit. When SAFARI-RAM operates in the SLC mode, each cell's endurance increases by a factor of 10 while the write latency of SAFARI-RAM decreases to $p\%$ of the write latency of the QLC mode. To measure the lifetime of SAFARI-RAM 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 SAFARI-RAM increases by 2× in SLC mode compared to the QLC mode. Formulate p using C_{QLC}, which denotes the capacity of SAFARI-RAM in the QLC mode.

$p = \frac{1}{172} \times (\frac{2^{18} \times 10^{10}}{C_{\text{QLC}}} - 8400)$

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 SAFARI-RAM in the SLC mode is equal to $\frac{C_{\text{SLC}}}{2^{12}}$.
- The total latency of each memory access in the SLC mode is equal to $(32 + 52 + 172 \times \frac{p}{10^2})$.
- The lifetime in the SLC mode is 5 years = 16×10^{16} (double the lifetime in the QLC mode).

$$\therefore \frac{C_{\text{SLC}}}{2^{12}} \times 10^8 \times (84 + 172 \times \frac{p}{10^2}) = 16 \times 10^{16}$$

$$172 \times \frac{p}{10^2} = \frac{2^{18} \times 10^8}{C_{\text{QLC}}} - 84$$

$$p = \frac{1}{172} \times (\frac{2^{18} \times 10^{10}}{C_{\text{QLC}}} - 8400)$$

6 Prefetching [100 points]

A processor is observed to have the following access pattern to cache blocks. Note that the addresses are *cache block addresses, not byte addresses.* This pattern is repeated for a large number of iterations.

Access Pattern P: A, $A+3$, $A+6$, A, $A+5$

Each cache block is 8 KB. The hardware has a fully associative cache with LRU replacement policy and a total size of 24 KB.

None of the prefetchers mentioned in this problem employ confidence bits, but they all start out with empty tables at the beginning of the access stream shown above. Unless otherwise stated, assume that 1) each access is separated long enough in time such that all prefetches issued can complete before the next access happens, and 2) the prefetchers have large enough resources to detect and store access patterns.

(a) [20 points] You have a stream prefetcher (i.e., a next-N-block prefetcher), but you do not know the prefetch degree (N) of it. However, you have a magical tool that displays the coverage and accuracy of the prefetcher. When you run a large number of repetitions of access pattern P, you get 40% coverage and 10% accuracy. What is the degree of this prefetcher (i.e., N)?

Next 4 **blocks**.

40% coverage with a stream prefetcher for this pattern means blocks $A+3$ and $A+6$ are prefetched. Possible N at this point are 3 and 4. Accuracy 10% $= 2/(N*5)$, so N is 4.

(b) [20 points] You are not satisfied with the performance of the stream prefetcher, so you decide to switch to a PC-based stride prefetcher that issues prefetch requests based on the stride detected for each memory instruction. Assume all memory accesses are incurred by the same load instruction (i.e., the same PC value) and the initial stride value for the prefetcher is set to 0.

Underline which of the cache block addresses are prefetched by this prefetcher:

A, A+3, <u>**A+6**</u>, A, A+5
A, A+3, <u>**A+6**</u>, A, A+5
A, A+3, <u>**A+6**</u>, A, A+5
A, A+3, <u>**A+6**</u>, A, A+5

Explain clearly to get points.

This prefetcher remembers the last stride and applies that to prefetch the next block from the current access.

(c) [20 points] You are not satisfied with the performance of the stride prefetcher either. So, you decided to switch again to a Markov prefetcher with a 12-entry correlation table (assume each entry can store a single address to prefetch, and *remembers the most-recent correlation*). When all the entries are filled, the prefetcher replaces the entry that is least-recently accessed.

Mark which of the cache block addresses are prefetched by this prefetcher:

A, A+3, A+6, A, A+5
A, A+3, **A+6**, **A**, A+5
A, A+3, **A+6**, **A**, A+5
A, A+3, **A+6**, **A**, A+5

Explain clearly to get points.

All entries are filled after the first repetition, except the entry for $A+5$. Accesses to $A+3$ and $A+5$ thrash each other for block A's next-block entry, so they cannot be correctly prefetched.

(d) [20 points] After how many repetitions of access pattern P does the Markov prefetcher from part (c) start to provide more coverage than the stream prefetcher from part (a), if it can at all? Show your work.

5 repetitions.

Coverage for Markov prefetcher from part (c) is $(0+2+3*(N-2))/(5N)$. This is equal to 40% when $N=4$, so it outperforms 40% at the 5-th repetition. We gave full credit if you wrote either 4 or 5, depending on the work shown.

(e) [20 points] You do not like the high hardware cost (i.e., the number of correlation entries) of the Markov prefetcher, and want to reduce the hardware cost by reducing the number of correlation table entries. What is the minimum number of entries that gives the same prefetcher performance as the 12-entry Markov prefetcher from part (c)? Similar to the last part, assume each entry can store a single next address to prefetch and remembers the most recent correlation. Show your work.

4 entries.

With 4 different accesses, we need at least 4 entries.

7 Cache Coherence [100 points]

We have a system with 4 processors {P0, P1, P2, P3} that can access memory at byte granularity. Each processor has a private data cache with the following characteristics:

- Capacity of 256 bytes.
- Direct-mapped.
- Write-back.
- Block size of 64 bytes.

Each processor has also a dedicated private cache for instructions. The characteristics of the instruction caches are not necessary to solve this question.

All data 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 transitions directly to the M state. The range of accessible memory addresses is from `0x00000` *to* `0xfffff`.

The semantics of the instructions used in this question are as follows:

Opcode	Operands	Description
ld	rx,[ry]	rx $\leftarrow$ Mem[ry]
st	rx,[ry]	rx $\rightarrow$ Mem[ry]
addi	rx,#VAL	rx $\leftarrow$ rx + VAL
subi	rx,#VAL	rx $\leftarrow$ rx - VAL
j	TARGET	jump to TARGET
bneq	rx,ry,TARGET	if([rx]!=[ry]) jump to TARGET

(a) [50 points] This is the initial state of the data caches in all processors:

Initial Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	`0x100`	`M`
1	`0x010`	`S`
2	`0x100`	`I`
3	`0x222`	`E`

Cache for P1		
Set	*Tag*	*MESI state*
0	`0x100`	`I`
1	`0x333`	`E`
2	`0x100`	`E`
3	`0x333`	`S`

Cache for P2		
Set	*Tag*	*MESI state*
0	`0x101`	`E`
1	`0x010`	`S`
2	`0x010`	`S`
3	`0x222`	`I`

Cache for P3		
Set	*Tag*	*MESI state*
0	`0x102`	`M`
1	`0x010`	`S`
2	`0x010`	`S`
3	`0x333`	`S`

This is the final state of the data caches in all processors (the shadowed sets in the figure represent the sets that change compared to the initial state):

Final Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	**0x102**	**S**
1	**0x102**	**E**
2	**0x102**	**E**
3	**0x333**	**I**

Cache for P1		
Set	*Tag*	*MESI state*
0	**0x102**	**S**
1	**0x333**	**I**
2	0x100	E
3	**0x333**	**M**

Cache for P2		
Set	*Tag*	*MESI state*
0	0x101	E
1	**0x333**	**M**
2	0x010	S
3	**0x222**	**E**

Cache for P3		
Set	*Tag*	*MESI state*
0	**0x102**	**S**
1	**0x333**	**I**
2	0x010	S
3	**0x333**	**I**

Make the following assumptions:

- Each processor executes the instructions in a *sequentially consistent* manner.
- You can make use of five registers: r0, r1, r2, r3, and r4.
- The ordering between two instructions from different processors might be ambiguous when there is no synchronization. If the order between two memory requests from different processors is ambiguous, and if the ordering is important for the final result, indicate the ordering between the two instructions in your solution.
- The initial values of all registers are the same in all processors, and they contain the following values:

r0=0x22200 r1=0x10200 r2=0x3338f r3=0x00000 r4=0x102C0

What are the minimum sequences of instructions in each processor that lead to the caches final state? Fill in the blanks. Write one instruction per line. Show your work as needed.

	P0
0	`addi r2,#0x40`
1	`ld r3,[r2]`
2	`LP:ld r3,[r1]`
3	`addi r1,#0x40`
4	`bneq r1,r4,LP`
5	
6	

	P1
0	`ld r3,[r1]`
1	`addi r2,#0x40`
2	`st r3,[r2]%(after P0: line 1)`
3	
4	
5	
6	

	P2
0	`addi r0,#0xC0`
1	`subi r2,#0x40`
2	`ld r3,[r0]`
3	`st r3,[r2]`%(after P3: line 1)
4	
5	
6	

	P3
0	`subi r2,#0x40`
1	`ld r3,[r2]`
2	
3	
4	
5	
6	

NOTE: This may not be the only solution.

(b) [50 points] This part is independent of part (a). All four processors execute the following code:

P0, P1, P2, and P3

```
LOOP: st r0, [r1]
      addi r1, #0x40
      bneq r1, r4, LOOP
```

The final state of the tag store is the following:

Final Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	0x102	M
1	0x102	M
2	0x102	M
3	0x100	M

Cache for P1		
Set	*Tag*	*MESI state*
0	0x102	I
1	0x102	I
2	0x102	I
3	0x102	M

Cache for P2		
Set	*Tag*	*MESI state*
0	0x106	M
1	0x106	M
2	0x104	M
3	0x104	M

Cache for P3		
Set	*Tag*	*MESI state*
0	0x106	I
1	0x106	I
2	0x106	M
3	0x106	M

Make the following assumptions about the initial state of the caches:

- The tag is the same in all sets of a processor (but the tags might be different among processors).
- The MESI state of all cache lines in all processors is the same.

What are the initial state values of the registers r0, r1, and r4 in all four processors? Show your work.

Solution:
P0: r0 = X, r1 = 0x10200, r4 = 0x102C0
P1: r0 = X, r1 = 0x102C0, r4 = 0x10300
P2: r0 = X, r1 = 0x10600, r4 = 0x10680
P3: r0 = X, r1 = 0x10680, r4 = 0x106C0 (or 0x10700)

NOTE: This may not be the only solution.

Explanation:

- The code executes store instructions in a loop.
- The address of the cache line is contained in r1.
- In each iteration we load the next cache line, i.e., we 1) increase the set while maintaining the same tag, or 2) increase the tag if the current address maps to set 3.
- We know from P0 (we can make a similar observation from P2) that we can use the code to modify sets 0 and 1, or sets 2 and 3, but not all at the same time.
- Because all sets are in Modified state, we can conclude that the initial state was Modified, and therefore, all the MESI states were initially in Modified state.
- P1 has some cache lines in Invalid state. For these lines to be Invalid, P0 has to invalidate them. Therefore, we know that the code for the first loop has to store cache lines with the tag 0x102 in sets 0, 1, and 2. Taking this into consideration, the only code option for P1 requires to store a cache line with tag 0x102 in set 3. A similar reasoning can be applied to processor 2 and 3.

What is the initial state of all caches? Fill in the blanks below.

Initial Tag Store States

Cache for P0		
Set	*Tag*	*MESI state*
0	`0x100`	`M`
1	`0x100`	`M`
2	`0x100`	`M`
3	`0x100`	`M`

Cache for P1		
Set	*Tag*	*MESI state*
0	`0x102`	`M`
1	`0x102`	`M`
2	`0x102`	`M`
3	`0x102`	`M`

Cache for P2		
Set	*Tag*	*MESI state*
0	`0x104`	`M`
1	`0x104`	`M`
2	`0x104`	`M`
3	`0x104`	`M`

Cache for P3		
Set	*Tag*	*MESI state*
0	`0x106`	`M`
1	`0x106`	`M`
2	`0x106`	`M`
3	`0x106`	`M`

8 BONUS: Genome Analysis [90 points]

During a process called read mapping in genome analysis, each genomic read (i.e., DNA sequence fragment) is mapped onto one or more possible locations in the reference genome based on the similarity between the read and the reference genome segment at that location. A read mapper applies the following 3-step hash table-based mapping method:

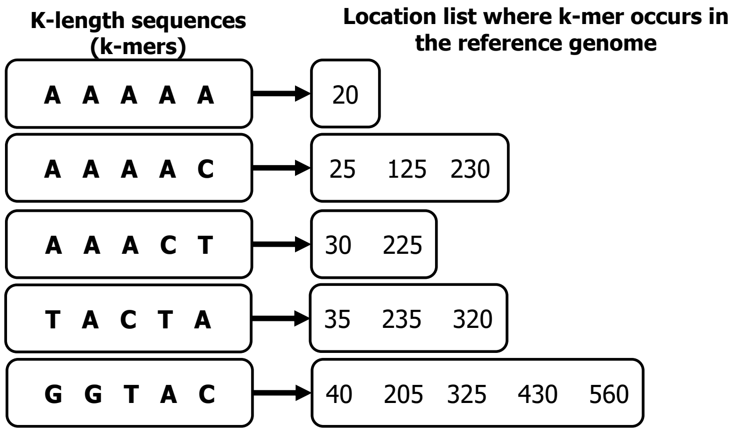
(1) The hash table-based read mapper first constructs a hash table that stores the list of locations in the reference genome that each possible short segment (i.e., k-mer, where k is the length of the segment) appears. Querying the hash table with a k-mer returns a list of locations for that k-mer.

(2) For each read, the mapper extracts 3 consecutive non-overlapping 5-mers and uses them to query the hash table.

(3) For each location of a k-mer, the mapper examines the differences between the entire read that includes the k-mer and the corresponding reference segment using the function: (*edit_ distance()*). Allowable edit operations include: (1) *substitution*, (2) *insertion*, and (3) *deletion of a character*. For example, edit operations between the read sequence ATATTTATA and the reference sequence ATAAGAT are as follows:

```
ATATTTATA-    Read sequence
|||   | |     (3 matches, 3 insertions, 1 match, 1 substitution, 1 match, 1 deletion)
ATA---AGAT    Reference sequence
```

The hash table (constructed in Step 1) is provided below. It includes a list of 5-mers extracted from the human reference genome and their corresponding location lists (each number represents the starting location of that k-mer in the reference genome sequence). If a k-mer does *not* exist in the hash table, you can assume it does not appear in the reference genome. Answer the following questions based on this hash table whenever needed.


K-length sequences (k-mers)
Location list where k-mer occurs in the reference genome
A A A A A → 20
A A A A C → 25 125 230
A A A C T → 30 225
T A C T A → 35 235 320
G G T A C → 40 205 325 430 560

8.1 Edit Distance Computation [45 points]

(a) [20 points] Compute the *edit distance* for the following read and reference sequence pair and provide the complete list of the edit operations used for calculation. Show your work.

Read sequence: ATCCTTAAATCTAAAATT
Reference sequence: CCTTAGAAACTTAA

8 Edits.

We use the following encoding to show the edits and matches: I: insertion, M: matches, D: deletions, S: substitutions

```
ATCCTTA-AATCTAAAATT    Read sequence
  ||||| || || ||       (2I, 5M, 1D, 2M, 1S, 2M, 1S, 2M, 3I: 8 edits)
--CCTTAGAAACTTAA---    Reference sequence
```

(b) [25 points] We would like to figure out (i.e., reverse engineer) the read sequence based on the following information available to us:

- The length of the read is 10.
- The *first* 5-mer of a read is found at the location 430 from the hash table.
- *edit_ distance()* function returns edit distance value of 3 between the read sequence and the human reference sequence starting from location 430. At least one of these three edits is a **deletion**.
- The reference sequence segment used for the *edit_ distance()* calculation is GGTACATAG.

Write down a read sequence that fits the criteria and show the complete list of edit operations. Note that more than one solution is possible.

There are several read sequences that may fit this criteria. Important thing is to notice that first 5 characters are an exact match between the reference genome and the read sequence. However, one of the remaining four characters must be deleted in the read sequence (i.e., ATAG). The rest of the edits must be insertions so that the read length becomes 10 and we have the edit value of 3. One can insert such insertions anywhere. For example:
Read Sequence: GGTACAAGTT

```
GGTACA-AGTT    Read sequence
|||||| ||      (6M, 1D, 2M, 2I: 3 edits)
GGTACATAG--    Reference sequence
```

8.2 Read Mapping [45 points]

Suppose that you would like to map the following reads to the human reference genome sequence. Each read is separated into smaller subsequences (k-mers) by underscores for readability.

read 1 = AAAAA_AAAAC_AAACT
read 2 = TACTA_GGTAC_AAACT
read 3 = GGTAC_AAACT_AAAAT
read 4 = AAAAC_TACTA_GGTAC

(a) [20 points] How many times will the edit distance function, *edit_ distance()*, be invoked when following the mapping steps described at the beginning of the question?

34 times.

Explanation: the number of times that the *edit_ distance()* function will be invoked is equal to the total number of occurrences of all k-mers that exist in both the hash table and the queried read. The frequency of each k-mer is given below:

read 1 = AAAAA_AAAAC_AAACT	1 + 3 + 2
read 2 = TACTA_GGTAC_AAACT	3 + 5 + 2
read 3 = GGTAC_AAACT_AAAAT	5 + 2 + 0
read 4 = AAAAC_TACTA_GGTAC	3 + 3 + 5

(b) [25 points] Suppose you want to change Step 3 to *additionally* include "*Adjacency Filtering*" (as discussed in lecture) in order to find *exact matches* before calling the *edit_ distance()* function. This means that, if we find an exact match via Adjacency Filtering, the *edit_ distance()* function is *not* invoked. Adjacency Filtering checks if first, second, and third k-mers extracted from the read exist in the reference genome at locations x, $x + k$, and $x + 2 \times k$, respectively.

At what locations in the reference genome does Adjacency Filtering find exact matches?

2 times: At locations (20, 25, 30) for read 1 and (225, 230, 235) for read 4.

Explanation: If all 3 5-mers are adjacent in the reference genome then the *edit_ distance()* function will not be invoked once for these three locations. The locations of the adjacent k-mers for each read are listed below:

read 1 = AAAAA_AAAAC_AAACT	20, 25, 30
read 2 = TACTA_GGTAC_AAACT	k-mers are not adjacent
read 3 = GGTAC_AAACT_AAAAT	Last k-mer does not exist
read 4 = AAAAC_TACTA_GGTAC	k-mers are not adjacent