ETH 263-2210-00L Computer Architecture, Fall 2021
HW 2: Processing-in-Memory and Genome Analysis (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Juan Gómez Luna, Mohammed Alser, Jisung Park, Lois Orosa Nogueira, Gagandeep Singh, Haiyu Mao, Behzad Salami, Nour Almadhoun Alserr, Mohammad Sadr, Hasan Hassan, Can Firtina Geraldo Francisco De Oliveira Junior, Abdullah Giray Yaglikci, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Rakesh Nadig, João Dinis Ferreira, Haocong Luo, Roknoddin Azizibarzoki

Given: Mon, Oct 25, 2021
Due: **Sun, Nov 07, 2021**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture21/`. Please, check your inbox, you should have received an email with the password you should use to login. If you did not receive any email, contact comparch@lists.inf.ethz.ch. In the first page after login, you should click in "Computer Architecture Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-5).** You should upload your answers to the Moodle platform (`https://moodle-app2.let.ethz.ch/course/view.php?id=15536`) as a single PDF file.
- If you have any questions regarding this homework, please ask them the Moodle forum (`https://moodle-app2.let.ethz.ch/mod/moodleoverflow/view.php?id=657929`).
- Please note that the handin questions have a hard deadline. However, you can submit your paper reviews till the end of the semester.

1. Critical Paper Reviews [1,000 points]

Please read the guidelines for reviewing papers and check the sample reviews. We also assign you three **required readings** for this homework. You may access them by *simply clicking on the QR codes below or scanning them.* We will give out extra credit that is worth 0.5% of your total grade for each good review. If you review a paper other than the REQUIRED papers, you will receive 250 BONUS points on top of 1,000 points you may get from paper reviews (i.e., each additional submission is worth 250 BONUS points with a possibility to get up to 5,500 points).

Guidelines

Sample reviews

Required Reading 1

Required Reading 2

Required Reading 3

Write an approximately one-page critical review for the following required readings (i.e., papers #1 to #3) **and** *at least* 1 more from the remaining 19 papers (i.e., papers #4 to #22). A review with bullet point style is more appreciated. Try not to use very long sentences and paragraphs. Keep your writing and sentences simple. Make your points bullet by bullet, as much as possible.

1. **(REQUIRED)** Seshadri et al., “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology”, MICRO, 2017. https://people.inf.ethz.ch/omutlu/pub/ambit-bulk-bitwise-dram_micro17.pdf
2. **(REQUIRED)** Senol Cali el al., “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis", MICRO, 2020. https://people.inf.ethz.ch/omutlu/pub/GenASM-approximate-string-matching-framework-for-genome-analysis_micro20.pdf
3. **(REQUIRED)** Ahn et al., “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing”, ISCA 2015, https://people.inf.ethz.ch/omutlu/pub/tesseract-pim-architecture-for-graph-processing_isca15.pdf
4. Mutlu et al., “A Modern Primer on Processing in Memory”, Invited Book Chapter in Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann, Springer https://people.inf.ethz.ch/omutlu/pub/ModernPrimerOnPIM_springer-emerging-computing-bookchapter21.pdf
5. Ahn et al., “PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture”, ISCA 2015, https://people.inf.ethz.ch/omutlu/pub/pim-enabled-instructons-for-low-overhead-pim_isca15.pdf
6. Boroumand et al., “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks”, ASPLOS 2018, https://people.inf.ethz.ch/omutlu/pub/Google-consumer-workloads-data-movement-and-PIM_asplos18.pdf
7. Singh et al., “FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications”, IEEE Micro 2021, https://arxiv.org/pdf/2106.06433.pdf
8. Seshadri et al., “RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization”, MICRO 2013, https://people.inf.ethz.ch/omutlu/pub/rowclone_micro13.pdf
9. Seshadri et al., “Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses”, MICRO 2015, https://people.inf.ethz.ch/omutlu/pub/GSDRAM-gather-scatter-dram_micro15.pdf
10. Wang et al., “FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching”, MICRO 2020, https://people.inf.ethz.ch/omutlu/pub/FIGARO-fine-grained-in-DRAM-data-relocation-and-caching_micro20.pdf
11. Hajinazar and Oliveira et al., “SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM”, ASPLOS 2021, https://people.inf.ethz.ch/omutlu/pub/SIMDRAM_asplos21.pdf
12. Alser et al., “Accelerating Genome Analysis: A Primer on an Ongoing Journey”, IEEE Micro 2020 https://people.inf.ethz.ch/omutlu/pub/AcceleratingGenomeAnalysis_ieeemicro20.pdf
13. Lee et al., “Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology”, ISCA 2021 https://ieeexplore.ieee.org/document/9499894
14. Harold Stone, “A Logic-in-Memory Computer”, TC 1970 https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=stone_logic_in_memory_1970.pdf
15. Dunn and Sadasivan et al., “SquiggleFilter: An Accelerator for Portable Virus Detection”, MICRO 2021, https://arxiv.org/pdf/2108.06610.pdf
16. Boroumand et al., “Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks”, PACT 2021 https://people.inf.ethz.ch/omutlu/pub/Google-neural-networks-for-edge-devices-Mensa-Framework_pact21.pdf
17. Giannoula et al., “SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures”, HPCA 2021 https://people.inf.ethz.ch/omutlu/pub/SynCron-synchronization-for-near-data-processing-systems_hpca21.pdf
18. Alser et al., “GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping”, Bioinformatics 2017, https://people.inf.ethz.ch/omutlu/pub/gatekeeper_FPGA-genome-prealignment-accelerator_bionformatics17.pdf
19. Alser et al., “Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment”, Bioinformatics 2019, https://people.inf.ethz.ch/omutlu/pub/shouji-genome-prealignment-filter_bionformatics19.pdf
20. Alser et al., “SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs”, Bioinformatics 2020, https://people.inf.ethz.ch/omutlu/pub/SneakySnake_UniversalGenomePrealignmentFilter_bioinformatics20.pdf

21. Kim et al., “GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies”, APBC 2018, `https://arxiv.org/pdf/1711.01177.pdf`
22. Turakhia et al., “Darwin: A Genomics Co-processor Provides up to 15,000× acceleration on long read assembly”, ASPLOS 2018, `http://bejerano.stanford.edu/papers/p199-turakhia.pdf`

2. Processing-in-Memory [150 points]

You have been hired to accelerate ETH's student database. After profiling the system for a while, you found out that one of the most executed queries is to *"select the hometown of the students that are from Switzerland and speak German"*. The attributes *hometown*, *country*, and *language* are encoded using a four-byte binary representation. The database has 32768 (2^{15}) entries, and each attribute is stored contiguously in memory. The database management system executes the following query:

```
bool position_hometown[entries];
for(int i = 0; i < entries; i++){
    if(students.country[i] == "Switzerland" && students.language[i] == "German"){
        position_hometown[i] = true;
    }
    else{
        position_hometown[i] = false;
    }
}
```

(a) You are running the above code on a single-core processor. Assume that:

- Your processor has an 8 MB direct-mapped cache, with a cache line of 64 bytes.
- A hit in this cache takes one cycle and a miss takes 100 cycles for both load and store operations.
- All load/store operations are serialized, i.e., the latency of multiple memory requests cannot be overlapped.
- The starting addresses of *students.country*, *students.language*, and *position_ hometown* are 0x05000000, 0x06000000, 0x07000000 respectively.
- The execution time of a non-memory instruction is zero (i.e., we ignore its execution time).

How many cycles are required to run the query? Show your work.

Cycles = cache_hits×1 + cache_misses×100 = 0×1 + (3×32×1024)×100

Explanation:
Since the cache size is 8 MB (2^{23}), direct-mapped, and the block size is 64 bytes (2^6), the address is divided as:

- block = address[5:0]
- index = address[22:6]
- tag = address[31:23]

The loop repeats for the total number of entries in the database (32×1024 times). In each iteration, the code loads addresses 0x05000000 and 0x06000000. It also stores the computation at address 0x07000000 (three memory accesses in total per cycle). All three addresses have the same index bits, but different tags. The cache hit rate is 0% since every memory access causes the eviction of the cache line that was just loaded into the cache.

(b) Recall that in class we discussed AMBIT, which is a DRAM design that can greatly accelerate Bulk Bitwise Operations by providing the ability to perform bitwise AND/OR/XOR of two rows in a subarray. AMBIT works 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 the table below (e.g., $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation).
In those instructions, AMBIT copies the source rows D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) AMBIT executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, copying a source row to DCC_i flips all bits in the source row and stores the result in both the source row and DCC_i. Assume that:

- The DRAM row size is **8 Kbytes**.
- An ACTIVATE command takes 50 cycles to execute.
- A PRECHARGE command takes 20 cycles to execute.
- DRAM has a single memory bank.
- The syntax of an AMBIT operation is: *bbop_* [and/or/xor] *destination, source_1, source_2.*
- Addresses 0x08000000 and 0x09000000 are used to store partial results.
- The rows at addresses 0x0A000000 and 0x0B00000 store the codes for *"Switzerland"* and *"German"*, respectively, in each four bytes throughout the entire row.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

i) The following code aims to execute the query *"select the hometown of the students that are from Switzerland and speak German"* in terms of Boolean operations to make use of AMBIT. Fill in the blank boxes such that the algorithm produces the correct result. Show your work.

```
for(int i = 0; i < □ ; i++){

  bbop_□ 0x08000000, 0x05000000 + i*8192, 0x0A000000;

  bbop_□ 0x09000000, 0x06000000 + i*8192, 0x0B000000;

  bbop_□ 0x07000000, 0x08000000, 0x09000000;
}
```

1st box = Number of iterations = $\frac{database\_size}{row\_buffer\_size} = \frac{32*1024*4\ bytes}{8*1024\ bytes} = 16$
2nd box = bbop_xor
3rd box = bbop_xor
4th box = bbop_or

Explanation:
AMBIT can execute the query as follows:
T1 = country XOR "Switzerland"
T2 = language XOR "German"
hometown = T1 OR T2

T1 and T2 are auxiliary rows used to store partial results.

ii) How much speedup does AMBIT provide over the baseline processor when executing the same query? Show your work.

Speedup = $\frac{3\times100\times32\times1024}{16\times2\times(25\times50+11\times20)+16\times(11\times50+5\times20)}$

Explanation:
To compute an XOR operation, AMBIT emits 25 ACTIVATE and 11 PRECHARGE commands. To compute an OR operation, it sends 11 ACTIVATE and 5 PRECHARGE commands.

3. In-DRAM Bit Serial Computation [150 points]

Recall that in class, we discussed Ambit, 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 Ambit is logically complete, it is possible to implement any other logic gate (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 **??** shows. With such a layout, it is possible to perform bit-serial arithmetic computations in Ambit.

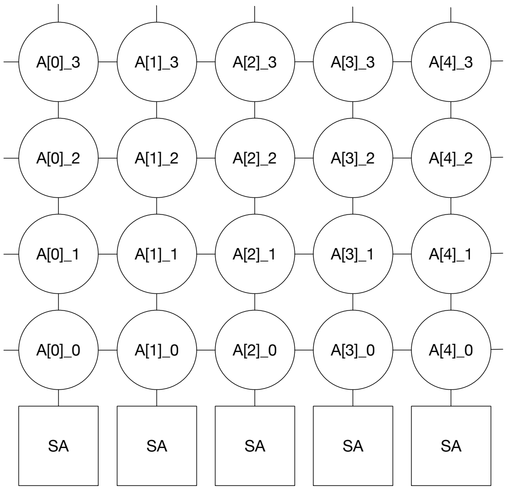


Figure 1. In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`.

We want to evaluate the potential performance benefits of using Ambit for arithmetic computations by implementing a simple workload, the element-wise addition of two arrays. Listing **??** 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];
}
```

We compare two possible implementations of the element-wise addition of two arrays: a CPU-based and an Ambit-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 Ambit). 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, adds 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 addition.
- *Ambit-based implementation*: This implementation assumes a bit serial layout for arrays `A`, `B`, and `C`. It performs additions in a bit serial manner, which only requires XOR, AND, and OR operations, as you can see in the 1-bit full adder in Figure **??**.

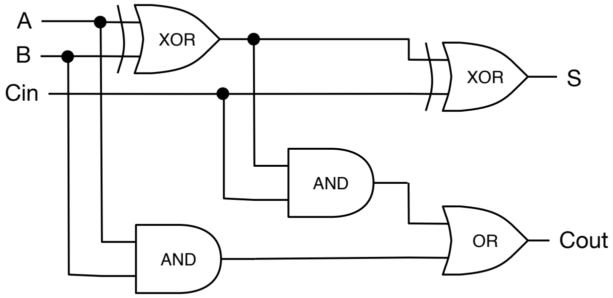


Figure 2. 1-bit full adder.

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 **??**, 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 D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) Ambit executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, the NOT operation copies a source row to DCC_i, flips all bits of the row, and stores the result in both the source row and DCC_i. 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.
- The syntax of an Ambit operation is: *bbop_* [and/or/xor] *destination, source_ 1, source_ 2.*
- The rows at addresses 0x00700000, 0x00800000, and 0x00900000 are used to store partial results. Initially, they contain all zeroes.
- The rows at addresses 0x00A00000, 0x00B00000, and 0x00C00000 store arrays `A`, `B`, and `C`, 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.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

(a) 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 addition for 65,536 4-bit elements and measure $t_{cpu} = 100$ us. In the second experiment, you run the STREAM-Copy benchmark for 102,400 4-bit elements and measure $t_{cpu} = 10$ us. 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?

$M = 10.24$ GB/s and $K = 1.38$ 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 102,400 4-bit elements:
$t_{cpu} = \frac{num\_bytes}{M}$;
$10 \times 10^{-6} = \frac{102{,}400 \times 4 \times 2}{8 \times M}$;
$M = 10.24$ GB/s.

Then, we obtain K with the measurement for the array addition. For `num_operations`, we count the same number as `num_elements`. For `num_bytes`, we count three arrays of 65,536 4-bit elements:
$t_{cpu} = K \times num\_operations + \frac{num\_bytes}{M}$;
$100 \times 10^{-6} = K \times 65,536 + \frac{65{,}536 \times 4 \times 3}{8 \times 10.24 \times 10^{9}}$;
$K = 1.38$ ns/operation.

(b) Write the code for the Ambit-based implementation of the element-wise addition of arrays `A` and `B`. The resulting array is `C`.

```


// As we are doing bit serial computation, we need a for loop

// with as many iterations as the number of bits per element.

// We call n the number of bits per element.


for(int i = 0; i < n;  i++){

  bbop_xor 0x00C00000+i*0x2000, 0x00A00000+i*0x2000, 0x00B00000+i*0x2000;

  bbop_and 0x00700000, 0x00A00000+i*0x2000, 0x00B00000+i*0x2000;

  bbop_and 0x00800000, 0x00900000, 0x00C00000+i*0x2000;

  bbop_xor 0x00C00000+i*0x2000, 0x00900000, 0x00C00000+i*0x2000; // S

  bbop_or 0x00900000, 0x00700000, 0x00800000; // Cout

}
```

(c) Compute the maximum throughput (in arithmetic operations per second, OPS) of the Ambit-based implementation as a function of the element size (i.e., bits/element).

$Thr_{ambit} = \frac{32}{n \times 10^{-9}}$ OPS $= \frac{32}{n}$ 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 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):
$t_{ambit} = (2 \times t_{XOR} + 2 \times t_{AND} + t_{OR}) \times n$;
$t_{ambit} = 2030 \times n$ ns.

Second, we obtain the throughput in arithmetic operations per second (OPS) as:
$Thr_{ambit} = \frac{65{,}536}{2030 \times n \times 10^{-9}}$; $Thr_{ambit} = \frac{32}{n \times 10^{-9}}$ OPS $= \frac{32}{n}$ GOPS.

(d) Determine the element size (in bits) for which the CPU-based implementation is faster than the Ambit-based implementation (Note: Use the same array size as in the previous part).

There is no number of bits per element that makes the CPU faster than Ambit.

Explanation:
We want to find n such that $Thr_{ambit} < Thr_{cpu}$, or $t_{ambit} > t_{cpu}$. If we use arrays of size 65,536 elements, we can write the following expression:
$t_{ambit} > t_{cpu}$;
$2030 \times n \times 10^{-9} > 1.38 \times 65,536 \times 10^{-9} + \frac{65{,}536 \times 3 \times n}{8 \times 10.24 \times 10^{9}}$;
This expression only returns a negative value of n. Thus, there is no n that makes the CPU faster than Ambit.

4. Caching vs. Processing-in-Memory [150 points]

We are given the following piece of code that makes accesses to integer arrays A and B. The size of each element in both A and B is 4 bytes. The base address of array A is $0x00001000$, and the base address of B is $0x00008000$.

```
movi R1, #0x1000 // Store the base address of A in R1
movi R2, #0x8000 // Store the base address of B in R2
movi R3, #0

Outer_Loop:
    movi R4, #0
    movi R7, #0
    Inner_Loop:
        add R5, R3, R4  // R5 = R3 + R4
        // load 4 bytes from memory address R1+R5
        ld R5, [R1, R5] // R5 = Memory[R1 + R5],
        ld R6, [R2, R4] // R6 = Memory[R2 + R4]
        mul R5, R5, R6  // R5 = R5 * R6
        add R7, R7, R5  // R7 += R5
        inc R4          // R4++
        bne R4, #2, Inner_Loop // If R4 != 2, jump to Inner_Loop

    //store the data of R7 in memory address R1+R3
    st [R1, R3], R7     // Memory[R1 + R3] = R7,
    inc R3              // R3++
    bne R3, #16, Outer_Loop // If R3 != 16, jump to Outer_Loop
```

You are running the above code on a single-core processor. For now, assume that the processor *does not* have caches. Therefore, all load/store instructions access the main memory, which has a fixed 50-cycle latency, for both read and write operations. Assume that all load/store operations are serialized, i.e., the latency of multiple memory requests *cannot* be overlapped. Also assume that the execution time of a non-memory-access instruction is zero (i.e., we ignore its execution time).

(a) What is the execution time of the above piece of code in cycles?

5 memory accesses per outer loop iteration.
$16 * 5 * 50 = 4000$ cycles

(b) Assume that a 128-byte private cache is added to the processor core in the next-generation processor. The cache block size is 8-byte. The cache is direct-mapped. On a hit, the cache services both read and write requests in 5 cycles. On a miss, the main memory is accessed and the access fills an 8-byte cache line in 50 cycles. Assuming that the cache is initially empty, what is the new execution time on this processor with the described cache? Show your work.

At the beginning A and B conflict in the first two cache lines. Then the elements of A and B go to different cache lines. The total execution time is 1910 cycles.
Here is the access pattern for the first outer loop iteration:
$0 - A[0], B[0], A[1], B[1], A[0]$
The first 4 references are loads, the last (A[0]) is a store. The cache is initially empty. We have a cache miss for A[0]. A[0] and A[1] is fetched to 0th index in the cache. Then, B[0] is a miss, and it is conflicting with A[0]. So, A[0] and A[1] are evicted. Similarly, all cache blocks in the first iteration are conflicting with each other. Since we have only cache misses, the latency for those 5 references is $5 * 50 = 250$ cycles
The status of the cache after making those seven references is:

Cache Index	Cache Block
0	~~A(0,1)~~, ~~B(0,1)~~, ~~A(0,1)~~, ~~B(0,1)~~, A(0,1)

Second iteration on the outer loop:
$1 - A[1], B[0], A[2], B[1], A[1]$

Cache hits/misses in the order of the references:
H, M, M, H, M
$Latency = 2 * 5 + 3 * 50 = 165$ cycles
Cache Status:
- A(0,1) is in set 0
- A(2,3) is in set 1
- the rest of the cache is empty

$2 - A[2], B[0], A[3], B[1], A[2]$

Cache hits/misses:
H, M, H, H, H
$Latency : 4 * 5 + 1 * 50 = 70$ cycles

Cache Status:
- B(0,1) is in set 0
- A(2,3) is in set 1
- the rest of the cache is empty

$3 - A[3], B[0], A[4], B[1], A[3]$

Cache hits/misses:
H, H, M, H, H
$Latency : 4 * 5 + 1 * 50 = 70$ cycles

Cache Status:
- B(0,1) is in set 0
- A(2,3) is in set 1
- A(4,5) is in set 2
- the rest of the cache is empty

$4 - A[4], B[0], A[5], B[1], A[4]$
Cache hits/misses:
H, H, H, H, H
$Latency : 5 * 5 = 25$ cycles

Cache Status:
- B(0,1) is in set 0
- B(2,3) is in set 1
- A(4,5) is in set 2
- the rest of the cache is empty

After this point, single-miss and zero-miss (all hots) iterations are interleaved until the 16th iteration.
Overall Latency:
$165 + 70 + (70 + 25) * 7 = 900$ cycles

(c) You are not satisfied with the performance after implementing the described cache. To do better, you consider utilizing a processing unit that is available *close to the main memory.* This processing unit can directly interface to the main memory with a *10-cycle* latency, for both read and write operations. How many cycles does it take to execute the same program using the in-memory processing units? (Assume that the in-memory processing unit does not have a cache, and the memory accesses are serialized like in the processor core. The latency of the non-memory-access operations is ignored.)

$16 * 5 * 10 = 800$

(d) You friend now suggests that, by changing the cache capacity of the single-core processor (in part (b)), she could provide as good performance as the system that utilizes the memory processing unit (in part (c)). Is she correct? What is the minimum capacity required for the cache of the single-core processor to match the performance of the program running on the memory processing unit?

Increasing the cache capacity does not help.

(e) [10 points] What other changes could be made to the cache design to improve the performance of the single-core processor on this program?

Since we don't have capacity and getting conflict due to the direct-mapped cache, we can change the cache design to be set-associative or fully-associative.

5. Genome Analysis [150 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:

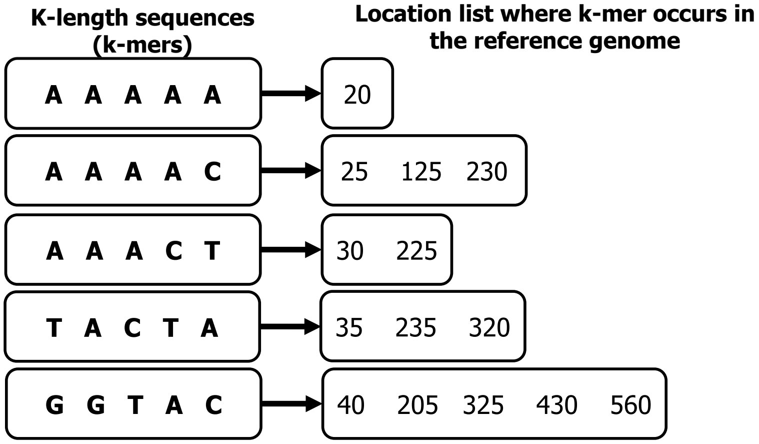
(1) The hash table-based read mapper first constructs a hash table that stores the list of locations in the reference genome that each possible short segment (i.e., k-mer, where k is the length of the segment) appears. Querying the hash table with a k-mer returns a list of locations for that k-mer.

(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

5.1. Edit Distance Computation

(a) 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) 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
```

5.2. Read Mapping

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) 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) 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