ETH 263-2210-00L Computer Architecture, Fall 2022
HW 1: Processing-in-memory (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Juan Gómez Luna, Mohammad Sadrosadati, Mohammed Alser, Rahul Bera, Nisa Bostanci, João Dinis Ferreira, Can Firtina, Nika Mansouri Ghiasi, Geraldo Francisco De Oliveira Junior, Konstantinos Kanellopoulos, Joël Lindegger, Rakesh Nadig, Ataberk Olgun, Abdullah Giray Yaglikci, Yahya Can Tugrul, Haocong Luo, Banu Cavlak, Aditya Manglik

Given: Friday, October 7, 2022
Due: **Friday, October 21, 2022**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture22/`. 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 (2-6).** You should upload your answers to the Moodle Platform (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=804436`) 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=807326`).
- 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]

You will do at least 5 readings for this homework, out of which 3 are tagged as **REQUIRED** papers. You may access them by simply clicking on the QR codes below or scanning them.

Required 1   Required 2   Required 3

Write an approximately one-page critical review for the readings (i.e., papers from #1 to #3 **and** at least 2 of the remaining papers, from #4 to #22). If you review a paper other than the 5 mandatory papers, you will receive 200 BONUS points on top of 1,000 points you may get from paper reviews (i.e., each additional submission is worth 200 BONUS points with a possibility to get up to 4400 points). Note that you will get **zero** points from the critical paper reviews if you do not submit the required paper reviews (i.e., papers from #1 to #3).

Please read the guideline slides for reviewing papers and watch Prof. Mutlu's guideline video on how to do a critical paper review. We also provide you with sample reviews which you can access using the QR code. 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. **We will give out extra credit that is worth 0.5% of your total course grade for each good review.**

Guideline Slides   Guideline Video   Sample Reviews

1. **(REQUIRED)** Richard Hamming, “You and Your Research,” Bell Communications Research Colloquium Seminar, 1986. `https://https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=youandyourresearch.pdf`
2. **(REQUIRED)** Onur Mutlu, “Intelligent Architectures for Intelligent Machines," World Conference on Energy Science and Technology (WCEST), 2021.
   - Talk: `https://www.youtube.com/watch?v=2OBdcw-ilQY&list=PL5Q2soXY2Zi8D_5MGV6EnXEJHnV2YFBJl&index=61`
   - Slide (PPT): `https://people.inf.ethz.ch/omutlu/pub/onur-TUBA-WCEST-Keynote-IntelligentArchitecturesForIntelligentSystems-August-11-2021.pptx`
   - Slide (PDF): `https://people.inf.ethz.ch/omutlu/pub/onur-TUBA-WCEST-Keynote-IntelligentArchitecturesForIntelligentSystems-August-11-2021.pdf`
3. **(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`
4. 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`
5. 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`
6. 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`
7. 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`
8. Singh et al., “FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications”, IEEE Micro 2021, `https://arxiv.org/pdf/2106.06433.pdf`
9. 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`
10. 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`
11. 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`
12. 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`
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. 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`
16. 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`
17. Ferreira et al., “pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables”, MICRO 2022 `https://arxiv.org/pdf/2104.07699.pdf`
18. Mao et al., “GenPIP: In-Memory Acceleration of Genome Analysis via Tight Integration of Basecalling and Read Mapping”, MICRO 2022 `https://arxiv.org/pdf/2209.08600.pdf`
19. Oliveira et al., “DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks”, IEEE Access 2021 `https://arxiv.org/pdf/2105.03725.pdf`
20. Xie et al., “Processing-in-Memory Enabled Graphics Processors for 3D Rendering”, HPCA 2017 `https://ieeexplore.ieee.org/document/7920862`
21. Gokhale et al., “Processing in memory: the Terasys massively parallel PIM array”, Computer 1995 `https://ieeexplore.ieee.org/document/375174`
22. Imani et al., “FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision”, ISCA 2019 `https://ieeexplore.ieee.org/document/8980299`

2. Caching vs. Processing-in-Memory [200 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.

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
$Latency = 2 * 5 + 3 * 50 = 160$ 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:
$250 + 160 + 70 + 7 * 70 + 6 * 25 = 1120$ 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) 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.

3. Processing-near-Memory [200 points]

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

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

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

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

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

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

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

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

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

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

App1:

For App1: Execution Time $= 3040 \times 10^8$ ns.

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

App2:

For App2: Execution Time $= 1775 \times 10^8$ ns.

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

You have learned in class that Processing-near-Memory (PnM) architectures can accelerate memory-bound workloads, since they provide higher bandwidth and lower latency than conventional processor-centric architectures. You decide to design PnM accelerators for App1 and App2.
The memory device you use is an early-generation 3D-stacked memory with an internal memory bandwidth of only 40 GB/s (i.e., twice the bandwidth of a single-channel 2D DDR4 memory). The 3D-stacked memory allows you to embed compute resources at the base die of the memory cube, called logic layer. However, the logic layer imposes several design limitations:

- The area available to build your accelerator in the logic layer of the 3D-stacked memory is **100** mm^2.
- The maximum clock frequency of the accelerator in the logic layer of the 3D-stacked memory is $\frac{1}{10}\times$ that of baseline CPU.
- The accelerator consists of one or more processing elements, each of which contains several functional units. Table 1 shows the functional units available to build your accelerators.
- A processing element of the accelerator executes the same computation as an entire iteration of the loop (i.e., all the instructions in the loop body). The processing element executes an iteration completely before moving to the next iteration. Therefore, if there are N processing elements, N iterations of the loop are executed in parallel.
- The loop iterations are executed on the processing elements as decided by a compiler, which unrolls the loop and preassigns iterations to the processing elements (i.e., the compiler schedules the iterations statically).
- Each functional unit of a processing element of the accelerator can execute one instruction of the loop body at a time. The same functional unit can execute different instructions (e.g., an arithmetic unit is capable of executing `add`, `sub`, `mult`, `inc`) at different times. Two functional units can execute in parallel, if there is no dependence between the operands.

Table 1. Functional units available to build your accelerators.

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

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

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

App1:

For App1: Area = 100 mm^2.

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

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

App2:

For App2: Area = 68 mm^2.

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

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

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

App1:

For App1: No.

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

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

App2:

For App2: No.

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

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

App1:

For App1: Speedup = 126.7×

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

App2:

For App2: Speedup = 0.92×

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

4. Processing in Memory: Ambit [400 points]

4.1. In-DRAM Bitmap Indices I [200 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.

One real-world application that can benefit from Ambit's in-DRAM bulk bitwise operations is the database bitmap index, as we also discussed in the lecture. By using bitmap indices, we want to run the following query on a database that keeps track of user actions: "How many unique users were active every week for the past w weeks?" Every week, each user is represented by a single bit. If the user was active a given week, the corresponding bit is set to 1. The total number of users is u.

We assume the bits corresponding to one week are all in the same row. If u is greater than the total number of bits in one row (the row size is 8 kilobytes), more rows in different subarrays are used for the same week. We assume that all weeks corresponding to the users in one subarray fit in that subarray.

We would like to compare two possible implementations of the database query:

- CPU-based implementation: This implementation reads the bits of all u users for the w weeks. For each user, it `and`s the bits corresponding to the past w weeks. Then, it performs a bit-count operation to compute the final result.
  Since this operation is very memory-bound, we simplify the estimation of the execution time as the time needed to read all bits for the u users in the last w weeks. The memory bandwidth that the CPU can exploit is X bytes/s.
- Ambit-based implementation: This implementation takes advantage of bulk `and` operations of Ambit. In each subarray, we reserve one Accumulation row and one Operand row (besides the control rows that are needed for the regular operation of Ambit). Initially, all bits in the Accumulation row are set to 1. Any row can be moved to the Operand row by using RowClone (recall that RowClone is a mechanism that enables very fast copying of a row to another row in the same subarray). t_{rc} and t_{and} are the latencies (in seconds) of RowClone's `copy` and Ambit's `and` respectively.
  Since Ambit does not support bit-count operations inside DRAM, the final bit-count is still executed on the CPU. We consider that the execution time of the bit-count operation is negligible compared to the time needed to read all bits from the Accumulation rows by the CPU.

(a) What is the total number of DRAM rows that are occupied by u users and w weeks?

$TotalRows = \lceil \frac{u}{8 \times 8k} \rceil \times w.$

Explanation:
The u users are spread across a number of subarrays:
$NumSubarrays = \lceil \frac{u}{8 \times 8k} \rceil.$

Thus, the total number of rows is:
$TotalRows = \lceil \frac{u}{8 \times 8k} \rceil \times w.$

(b) What is the throughput in users/second of the Ambit-based implementation?

$Thr_{Ambit} = \frac{u}{\lceil \frac{u}{8\times 8k} \rceil \times w \times (t_{rc}+t_{and}) + \frac{u}{X\times 8}}$ users/second.

Explanation:
First, let us calculate the total time for all bulk `and` operations. We should add t_{rc} and t_{and} for all rows:
$t_{and-total} = \lceil \frac{u}{8\times 8k} \rceil \times w \times (t_{rc}+t_{and})$ seconds.

Then, we calculate the time needed to compute the bit count on CPU:
$t_{bitcount} = \frac{\frac{u}{8}}{X} = \frac{u}{X\times 8}$ seconds.

Thus, the throughput in users/s is:
$Thr_{Ambit} = \frac{u}{t_{and-total}+t_{bitcount}}$ users/second.

(c) What is the throughput in users/second of the CPU implementation?

$Thr_{CPU} = \frac{X\times 8}{w}$ users/second.

Explanation:
We calculate the time needed to bring all users and weeks to the CPU:
$t_{CPU} = \frac{\frac{u\times w}{8}}{X} = \frac{u\times w}{X\times 8}$ seconds.

Thus, the throughput in users/s is:
$Thr_{CPU} = \frac{u}{t_{CPU}} = \frac{X\times 8}{w}$ users/second.

(d) What is the maximum w for the CPU implementation to be faster than the Ambit-based implementation? Assume u is a multiple of the row size.

$w < \frac{1}{1-\frac{X}{8k}\times(t_{rc}+t_{and})}$.

Explanation:
We compare t_{CPU} with $t_{and-total} + t_{bitcount}$:
$t_{CPU} < t_{and-total} + t_{bitcount}$;

$\frac{u\times w}{X\times 8} < \frac{u}{8\times 8k} \times w \times (t_{rc}+t_{and}) + \frac{u}{X\times 8}$;

$w < \frac{1}{1-\frac{X}{8k}\times(t_{rc}+t_{and})}$.

4.2. In-DRAM Bitmap Indices II [200 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 sub-array. 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.

5. In-DRAM Bit Serial Computation [200 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 1 shows. With such a layout, it is possible to perform bit-serial arithmetic computations in Ambit.

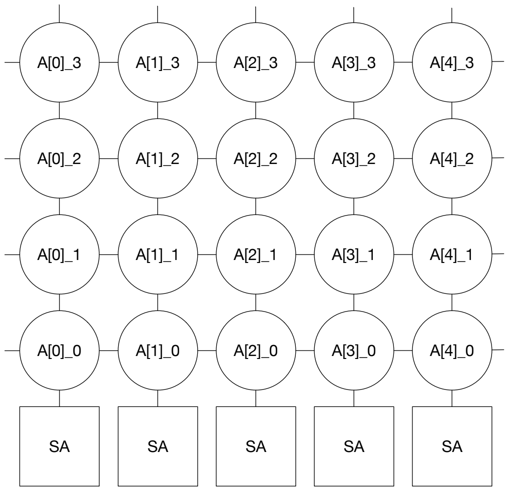


Figure 1. In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`.

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 2 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 2.

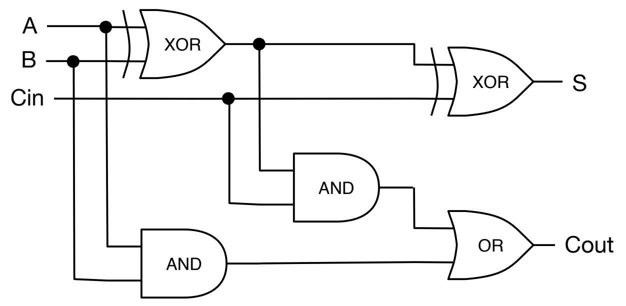


Figure 2. 1-bit full adder.

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 2, 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 2. 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 ($\underline{n}$ 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).

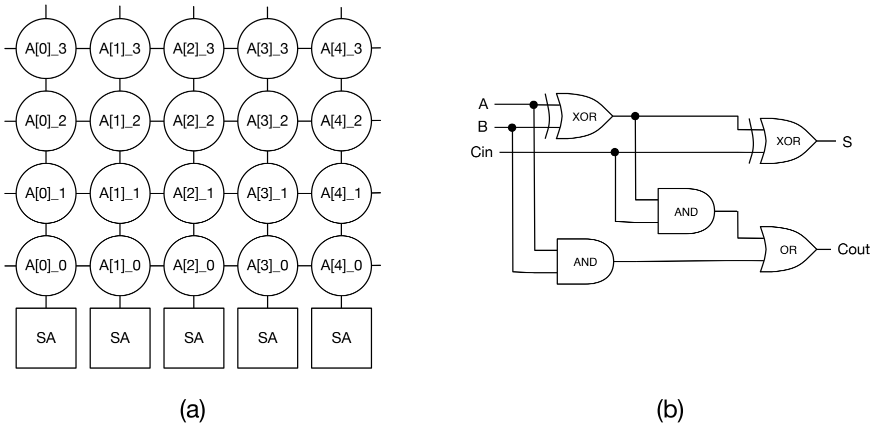
There is no number of bits per element that makes the CPU faster than Ambit.

Explanation:
We want to find $\underline{n}$ 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 $\underline{n}$. Thus, there is no $\underline{n}$ that makes the CPU faster than Ambit.

6. Processing-using-Memory [200 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 3(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 3(b) shows.



A[0]_3 A[1]_3 A[2]_3 A[3]_3 A[4]_3
A[0]_2 A[1]_2 A[2]_2 A[3]_2 A[4]_2
A[0]_1 A[1]_1 A[2]_1 A[3]_1 A[4]_1
A[0]_0 A[1]_0 A[2]_0 A[3]_0 A[4]_0
SA SA SA SA SA
(a)
A
B
Cin
XOR
XOR
S
AND
AND
OR
Cout
(b)

Figure 3. (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 4(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 4(b) shows an example of an 8-bit Kogge-Stone adder.

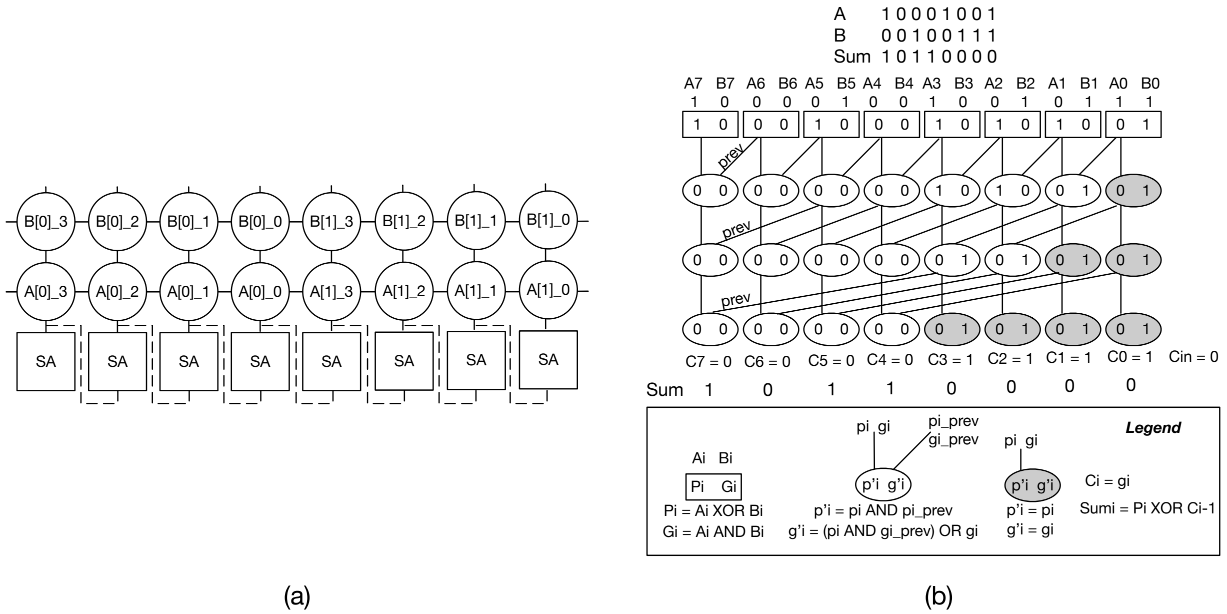


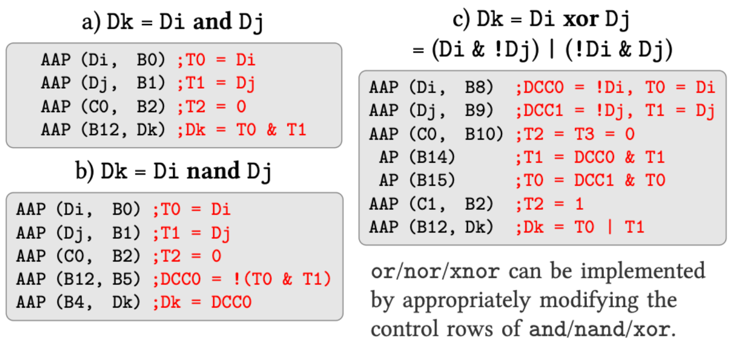
Figure 4. (a) In-DRAM bit-parallel layout for arrays `A` and `B`, which contain two 4-bit elements each. DRAM cells in the same bitline contain the same bits of equal-index elements of different arrays. `A[i]_j` represents bit `j` of element `i`. (b) Example of an 8-bit Kogge-Stone adder. A (10001001)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

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 2 shows a sequential code for the addition of two input arrays A and B into output array C.

Listing 2. 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 5, 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.



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 5. 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.

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 5) correspond to 3 rows, enabling Triple-Row Activation (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) 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) 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) 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.