ETH 263-2210-00L Computer Architecture, Fall 2020
HW 2: RowHammer, Computation in Memory (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Mohammed Alser, João Dinis Ferreira, Rahul Bera, Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gómez Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim, Nika Mansouri Ghiasi, Haiyu Mao, Lois Orosa Nogueira, Jisung Park, Minesh Patel, Gagandeep Singh, Kosta Stojiljkovic, Abdullah Giray Yaglikci

Given: Saturday, Oct 10, 2020
Due: **Thursday, Oct 22, 2020**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture20/`. Please check your inbox. You should have received an email with the password you should use to login. If you did not receive any emails, 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=13549`) as a single PDF file.

1. Critical Paper Reviews [1000 points]

Please read the guidelines for reviewing papers and check the sample reviews. We also assign you a **required reading** 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 submit 5 or 6 paper reviews, you will receive 250 or 500 BONUS points on top of 1000 points you may get from paper reviews, respectively (i.e., each additional submission is worth 250 BONUS points).

Guidelines   Sample reviews   Required Reading

Write an approximately one-page critical review for the following required reading (i.e., Paper #1) **and** *at least* 3 of the remaining 5 papers (i.e., papers from #2 to #6). 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) Ghose et al., "*Processing-in-Memory: A Workload-Driven Perspective*, in IBM Journal of Research & Development, 2019. `https://people.inf.ethz.ch/omutlu/pub/processing-in-memory_workload-driven-perspective_IBMjrd19.pdf`
2. Frigo et al., "*TRRespass: Exploiting the Many Sides of Target Row Refresh*, in Proceedings of the 41st IEEE Symposium on Security and Privacy, 2020. `https://people.inf.ethz.ch/omutlu/pub/rowhammer-TRRespass_ieee_security_privacy20.pdf`
3. Seshadri et al., "*RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization*, in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013. `https://people.inf.ethz.ch/omutlu/pub/rowclone_micro13.pdf`
4. Seshadri et al., "*Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology*, in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017. `https://people.inf.ethz.ch/omutlu/pub/ambit-bulk-bitwise-dram_micro17.pdf`
5. Ahn et al., "*A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing*, in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015. `https://people.inf.ethz.ch/omutlu/pub/tesseract-pim-architecture-for-graph-processing_isca15.pdf`
6. Boroumand et al., "*Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks*, in Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, 2018. `https://people.inf.ethz.ch/omutlu/pub/Google-consumer-workloads-data-movement-and-PIM_asplos18.pdf`

2. RowHammer [200 points]

2.1. RowHammer Properties

Determine whether each of following statements is true or false.

(a) Cells in a DRAM with a smaller technology node are more vulnerable to RowHammer.

1. [✓ True] 2. False

(b) Cells which have shorter retention times are especially vulnerable to RowHammer.

1. True 2. [✓ False]

(c) The vulnerability of cells in a victim row to RowHammer depends on the data stored in the victim row.

1. [✓ True] 2. False

(d) The vulnerability of cells in a victim row to RowHammer depends on the data stored in the aggressor row.

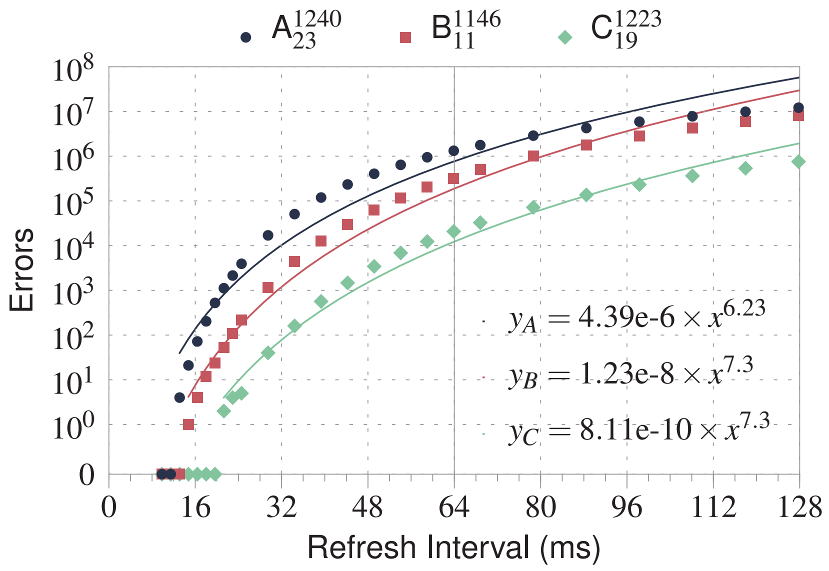
1. [✓ True] 2. False

(e) RowHammer-induced errors are mostly repeatable.

1. [✓ True] 2. False

2.2. RowHammer Mitigations

One research group rigorously investigated the relationship between the DRAM refresh rate and RowHammer-induced errors using real DRAM modules from three major memory manufacturers: A, B, and C. The following figure shows the characterization results of the *most RowHammer-vulnerable* modules of each DRAM manufacturer A, B, and C (A_{23}, B_{11}, and C_{19}, respectively). As shown in the figure below, we can achieve an *effectively-zero* probability of RowHammer-induced errors when we reduce the refresh interval to 8 ms. This is because the probability of RowHammer-induced errors becomes less than that of a random DRAM circuit fault.



As a leading computer architect of a company, you are asked to design an efficient RowHammer mitigation technique based on the characterization results.

(a) A junior engineer suggests to simply reduce the refresh interval from the default 64 ms to 8 ms. How will the bank utilization U and the DRAM energy consumption E of all refresh operations be changed over the current system?

$E' = 8 \times E$
$U' = 8 \times U$

(b) A DRAM manufacturer releases a higher capacity version of the same DRAM module (4x the total capacity). After rigorously reverse-engineering, you determine that the manufacturer doubles both the (1) number of rows in a bank, and (2) number of banks in a module without changing any other aspects (e.g., row size, bus rate, and timing parameters). Your company considers upgrading the current system with the higher-capacity DRAM module and asks your opinion. In the current system, the bank utilization of refresh operations is 0.05 when the refresh interval is 64 ms. Would reducing the refresh interval to 8 ms still be viable (in terms of RowHammer protection capability and performance overheads) in a module with 2x the number of rows per bank and 2x the number of banks per module? How about in a module with 4x the number of rows per bank and 4x the number of banks per module?

With a 2x increase, it is still viable, but very impractical. Most throughput is consumed by refresh operations ($U = 0.05 \times 8 \times 2 = 0.8$)

With 4x increase, it is impossible to perfectly defend RowHammer attacks by reducing the refresh interval, because we cannot refresh every row within 8 ms ($U = 0.05 \times 8 \times 4 = 1.6 > 1$.) Furthermore, throughput loss would reaches very close to 1.

(c) Due to significant overheads of reducing the refresh interval, your team decides to find other RowHammer mitigation techniques with lower overhead. The junior engineer proposes a counter-based approach as follows:

In this approach, the memory controller maintains a counter for each row R, which increments when an adjacent row is activated, and resets when Row R is activated or refreshed. If the value of a row R's counter exceeds a threshold value, T, the memory controller activates row R and resets its respective counter.

Here are some specifications on the current memory system:

- Interface: DDR3-1333
- $CL = 7$
- $tRCD = 13.5$ ns
- $tRAS = 35$ ns
- $tWR = 15$ ns
- $tRP = 13.5$ ns
- $tRFC = 120$ ns
- Configuration: Per-channel memory controller deals with 2 ranks, each of which has 8 banks. The number of rows per bank is 2^{15}. Each row in one bank is 8 KiB.

Are the given specifications enough to implement the proposed approach? If no, list the specifications additionally required.

No. We should know the exact mapping between row addresses and physical rows.

(d) Suppose that all the necessary specifications for implementing the proposed approach are known. Calculate the maximum value of T that can guarantee the same level of security against RowHammer attacks over when adopting 8-ms refresh interval?

T should be less than or equal to the maximum number of activations within 8 ms.

$$\begin{aligned} T \leq 8 \times 10^{-3}/t_{RC} &= 8 \times 10^{-3}/(t_{RAS} + t_{RP}) \\ &= 8 \times 10^{-3}/48.5 \times 10^{-9} \\ &= 164948.45... \\ \therefore T &= 164948 \end{aligned}$$

(e) Calculate the number of bits required for counters in each memory controller. How does it change when the number of rows per bank and the number of banks per chip are doubled?

The number of bits B of a single counter should meet $2^B > T = 0.164 \times 10^5$.
$2^{17} < T < 2^{18}, \therefore B = 18$
$18\ [\frac{bit}{row}] \times (2^{15})\ [\frac{row}{bank}] \times 16\ [bank] = 9 \times 2^{20}$ bits $= 9$ Mib

It will increase to 36 Mib with 2x rows and 2x banks.

(f) Obviously, we can reduce the size of counters in each memory controller, if we use a smaller value for T. What are its drawbacks?

Performance and energy overheads from unnecessary activations.

(g) You recall *PARA (Probabilistic Adjacent Row Activation)* which is proposed in the first RowHammer paper. Here is how a PARA-enabled memory controller works (for more details, see Section 8.2 in the paper[1]):

Whenever a row is closed, the controller flips a biased coin with a probability p of turning up heads, where $p << 1$. If the coin turns up heads, the controller opens one of its adjacent rows where either of the two adjacent rows are chosen with equal probability ($p/2$). Due to its probabilistic nature, PARA does not guarantee that the adjacent will always be refreshed in time. Hence, PARA cannot prevent disturbance errors with absolute certainty. However, its parameter p can be set so that disturbance errors occur at an extremely low probability many orders of magnitude lower than the failure rates of other system components (e.g., more than 1% of hard-disk drives fail every year.)

Suppose that the probability of experiencing an error for PARA in 64 ms is 1.9×10^{-22} when $p = 0.001$ at the *worst operating conditions.* How can we estimate the probability of experiencing an error in one year based on that value? Show your work. (If you just put an answer, you will get no points for this problem.)

Let P_I be the probability of experiencing an error for PARA in a time interval I (i.e., $P_{64ms} = 1.9 \times 10^{-22}$). Since we can consider a year as a sequence of independent 64-ms time windows, the number of errors in a year can be modeled as a random variable X that is binomially-distributed with parameters $B(N,\ P_{64ms})$ where N is the number of 64-ms time windows in a year (i.e., $N = (365 \times 24 \times 60 \times 60)/(64 \times 10^{-3}) = 492,750,000$). Therefore, the probability of that at least one error occurs in a year can be calculated as follows:

$$\begin{aligned}
\therefore P_{1year} &= P(X \geq 1) \\
&= 1 - P(X = 0) \\
&= 1 - (1 - P_{64ms})^N \\
&= 1 - (1 - 1.9 \times 10^{-22})^{492750000} \\
&= 1 - 0.99999999999990637750000000438259 \\
&= 9.36224999999561741... \times 10^{-14}
\end{aligned}$$

[1] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu, "Flipping Bits in Memory without Accessing Them: an Experimental Study of DRAM Disturbance Errors," in Proceedings of the 41st Annual International Sympoisum on Computer Architecture, 2014. `https://people.inf.ethz.ch/omutlu/pub/dram-row-hammer_isca14.pdf`

3. Processing in Memory: Ambit [200 points]

3.1. In-DRAM Bitmap Indices I

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 $t_{and-total} + t_{bitcount}$:
$t_{CPU} < t_{and-total} + t_{bitcount}$;

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

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

3.2. In-DRAM Bitmap Indices II

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 MiB 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 MiB (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 + i*8192, 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 $= \frac{3\times 100\times 32\times 1024}{16\times 2\times(25\times 50+11\times 20)+16\times(11\times 50+5\times 20)}$

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.

4. 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.

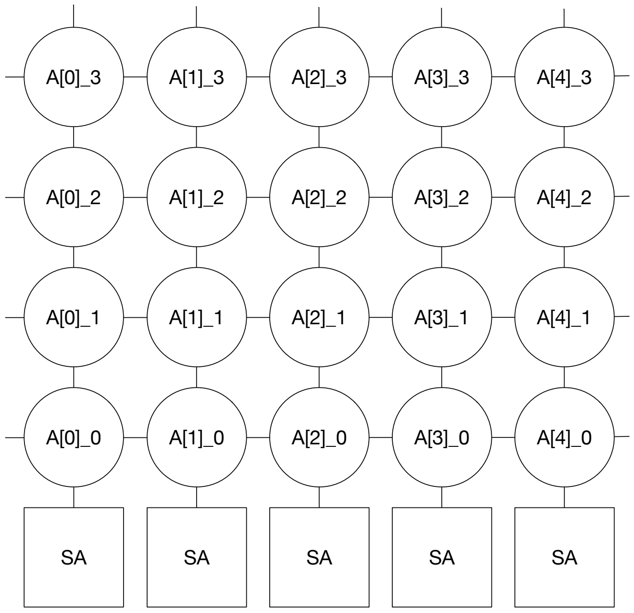


Figure 1. In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`.

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 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];
}
```

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.

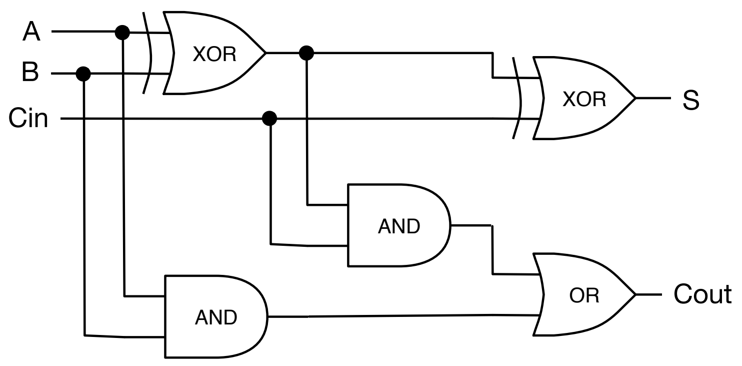


Figure 2. 1-bit full adder.

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 1, where $AAP(X,Y)$ represents double row activation of rows X and Y followed by a precharge operation, and $AAAP(X,Y,Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation.
In those instructions, Ambit copies the source rows 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.

5. 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
        add R4, R4, #4  // R4 += 4
        bne R4, #8, Inner_Loop // If R4 != 8, jump to Inner_Loop

    //store the data of R7 in memory address R1+R3
    st [R1, R3], R7     // Memory[R1 + R3] = R7,
    add R3, R3, #4      // R3 += 4
    bne R3, #64, Outer_Loop // If R3 != 64, 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? Show your work.

4000 cycles.

Explanation: There are 5 memory accesses for each outer loop iteration. The outer loop iterates 16 times, and each memory access takes 50 cycles.
$16 * 5 * 50 = 4000$ 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.

1120 cycles.

Explanation.
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 1120 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	

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 $= 2 \times 5 + 3 \times 50 = 160$ cycles
Cache Status: A(0,1) is in set 0, A(2,3) is in set 1, and 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: $4 \times 5 + 1 \times 50 = 70$ cycles
Cache Status: B(0,1) is in set 0, A(2,3) is in set 1, and 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: $4 \times 5 + 1 \times 50 = 70$ cycles
Cache Status: B(0,1) is in set 0, A(2,3) is in set 1, A(4,5) is in set 2, and 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: $5 \times 5 = 25$ cycles
Cache Status: B(0,1) is in set 0, B(2,3) is in set 1, A(4,5) is in set 2, and the rest of the cache is empty.

After this point, single-miss and zero-miss (all hits) iterations are interleaved until the 16th iteration.
Overall Latency: $250 + 160 + 70 + (70 + 25) \times 6 + 70 = 1120$ 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? Show your work. (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.)

800 cycles.

Explanation: Same as for the processor core without a cache, but the memory access latency is 10 cycles instead of 50. $16 * 5 * 10 = 800$

(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? Show your work.

No, she is not correct.

Explanation: Increasing the cache capacity does not help because doing so cannot eliminate the conflicts to Set 0 in the first two iterations of the outer loop.

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

Increasing the associativity of the cache.

Explanation: Although there is enough cache capacity to exploit the locality of the accesses, the fact that in the first two iterations the accessed data map to the same set causes conflicts. To improve the hit rate and the performance, we can change the address-to-set mapping policy. For example, we can change the cache design to be set-associative or fully-associative.