7 Processing-in-Memory [65 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) [25 points] 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) [20 points] 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) [20 points] 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.