1 GPUs and SIMD [90 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program. As we saw in lecture and practice exercises, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers, so there are no loads and stores in this program. (Hint: Notice that there are 6 instructions in each thread.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU. Please assume that all values in arrays B and C have magnitudes less than 10 (i.e., $|B[i]| < 10$ and $|C[i]| < 10$, for all i).

```
for (i = 0; i < 1008; i++) {
  A[i] = B[i] * C[i];
  if (A[i] < 0) {
    C[i] = A[i] * B[i];
    if (C[i] < 0) {
      A[i] = A[i] + 1;
    }
    A[i] = A[i] - 2;
  }
}
```

Please answer the following six questions.

(a) [10 points] How many warps does it take to execute this program?

(b) [10 points] What is the *maximum* possible SIMD utilization of this program?

(c) [20 points] Please describe what needs to be true about arrays B and C to reach the *maximum* possible SIMD utilization asked in part (b). (Please cover all possible cases in your answer)

(d) [10 points] What is the *minimum* possible SIMD utilization of this program?

(e) [20 points] Please describe what needs to be true about arrays B and C to reach the *minimum* possible SIMD utilization asked in part (d). (Please cover all possible cases in your answer)

(f) [20 points] Now consider a GPU that employs *Dynamic Warp Formation (DWF)* to improve the SIMD utilization. As we discussed in the class, DWF dynamically merges threads executing the same instruction (after branch divergence). What is the maximum achievable SIMD utilization using DWF under the conditions you found in part (e)? Explain your answer.

2 In-DRAM Bitmap Indices [70 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) [15 points] What is the total number of DRAM rows that are occupied by u users and w weeks?

(b) [20 points] What is the throughput in users/second of the Ambit-based implementation?

(c) [20 points] What is the throughput in users/second of the CPU implementation?

(d) [15 points] 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.

3 Cache Performance Analysis [80 points]

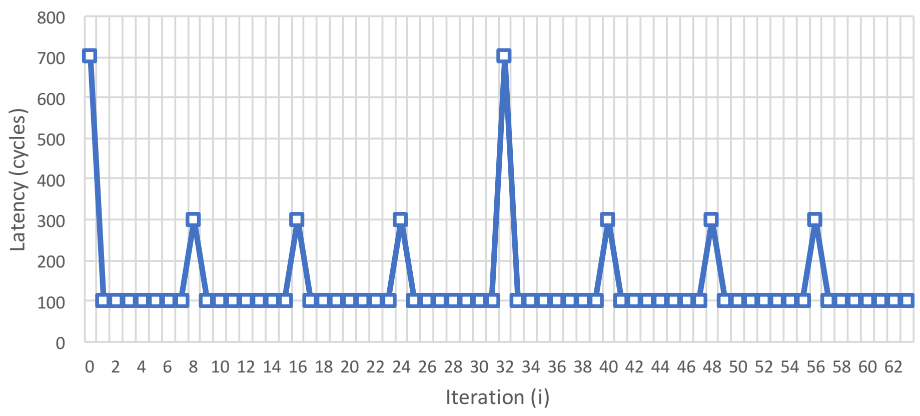
We are going to microbenchmark the cache hierarchy of a computer with the following two codes. The array `data` contains 32-bit unsigned integer values. For simplicity, we consider that accesses to the array `latency` bypass all caches (i.e., `latency` is *not* cached). `timer()` returns a timestamp in cycles.

```
(1) j = 0;
     for (i=0; i<size; i+=stride){
        start = timer();
        d = data[i];
        stop = timer();
        latency[j++] = stop - start;
     }

(2) for (i=0; i<size1; i+=stride1){
        d = data[i];
     }
     j = 0;
     for (i=0; i<size2; i+=stride2){
        start = timer();
        d = data[i];
        stop = timer();
        latency[j++] = stop - start;
     }
```

The cache hierarchy has two levels. L1 is a 4kB set associative cache.

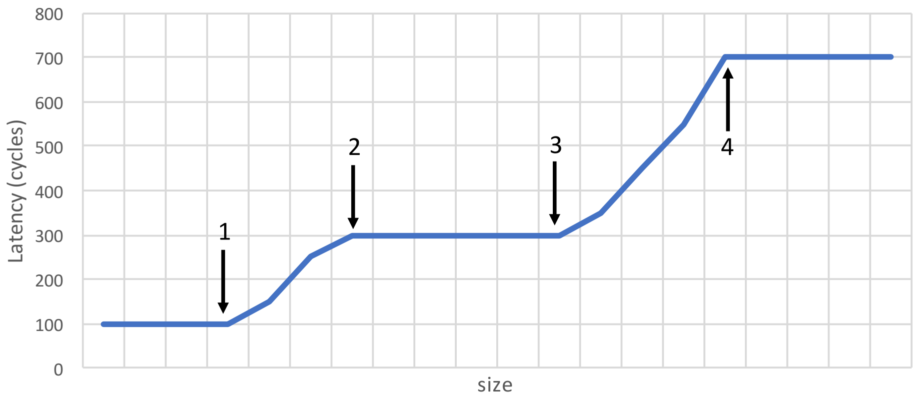
(a) [15 points] When we run code (1), we obtain the latency values in the following chart for the first 64 reads to the array `data` (in the first 64 iterations of the loop) with `stride` equal to 1. What are the cache block sizes in L1 and L2?



(b) [20 points] Using code (2) with `stride1` = `stride2` = 32, `size1` = 1056, and `size2` = 1024, we observe `latency[0]` = 300 cycles. However, if `size1` = 1024, `latency[0]` = 100 cycles. What is the maximum number of ways in L1? (Note: The replacement policy can be either FIFO or LRU).

(c) [20 points] We want to find out the exact replacement policy, assuming that the associativity is the maximum obtained in part (b). We first run code (2) with `stride1` = 32, `size1` = 1024, `stride2` = 64, and `size2` = 1056. Then (after resetting `j`), we run code (1) with `stride` = 32 and `size` = 1024. We observe `latency[1]` = 100 cycles. What is the replacement policy? Explain. (Hint: The replacement policy can be either FIFO or LRU. You need to find the correct one and explain).

(d) [25 points] Now we carry out two consecutive runs of code (1) for different values of `size`. In the first run, `stride` is equal to 1. In the second run, `stride` is equal to 16. We ignore the latency results of the first run, and average the latency results of the second run. We obtain the following graph. What do the four parts shown with the arrows represent?



Before arrow 1:

Between arrow 1 and arrow 2:

Between arrow 2 and arrow 3:

Between arrow 3 and arrow 4:

After arrow 4:

Explain as needed (if you need more):

4 SIMD [90 points]

We have two SIMD engines: 1) a traditional vector processor and 2) a traditional array processor. Both processors can support a vector length up to 16.

All instructions can be fully pipelined, the processor can issue one vector instruction per cycle, and the pipeline does not forward data (no chaining). For the sake of simplicity, we ignore the latency of the pipeline stages other than the execution stages (e.g, decode stage latency: 0 cycles, write back latency: 0 cycles, etc).

We implement the following instructions in both designs, with their corresponding execution latencies:

Operation	Description	Name	Latency of a single operation (VLEN=1)
VADD	VDST ← VSRC1 + VSRC2	vector add	5 cycles
VMUL	VDST ← VSRC1 * VSRC2	vector mult.	15 cycles
VSHR	VDST ← VSRC >> 1	vector shift	1 cycles
VLD	VDST ← mem[SRC]	vector load	20 cycles
VST	VSRC → mem[DST]	vector store	20 cycles

- All the vector instructions operate with a vector length specified by VLEN. The VLD instruction loads VLEN consecutive elements from the DST address specified by the value in the VDST register. The VST instruction stores VLEN elements from the VSRC register in consecutive addresses in memory, starting from the address specified in DST.
- Both processors have eight vector registers (VR0 to VR7) which can contain up to 16 elements, and eight scalar registers (R0 to R7). The entire vector register needs to be ready (i.e., populated with all VLEN elements) before any element of it can be used as part of another operation.
- The memory can sustain a throughput of one element per cycle. The memory consists of 16 banks that can be accessed independently. A single memory access can be initiated in each cycle. The memory can sustain 16 parallel accesses if they all go to different banks.

(a) [10 points] Which processor (array or vector processor) is more costly in terms of chip area? Explain.

(b) [25 points] The following code takes 52 cycles to execute on the vector processor:

```
VADD VR2 ← VR1, VR0
VADD VR3 ← VR2, VR5
VMUL VR6 ← VR2, VR3
```

What is the VLEN of the instructions? Explain your answer.

How long would the same code execute on an array processor with the same vector length?

(c) [25 points] The following code takes 94 cycles to execute on the vector processor:

```
VLD   VR0 ← mem[R0]
VLD   VR1 ← mem[R1]
VADD VR2 ← VR1, VR0
VSHR VR2 ← VR2
VST  VR2 → mem[R2]
```

Assume that the elements loaded in VR0 are all placed in different banks, and that the elements loaded into VR1 are placed in the same banks as the elements in VR0. Similarly, the elements of VR2 are stored in different banks in memory. What is the VLEN of the instructions? Explain your answer.

(d) [30 points] We replace the memory with a new module whose characteristics are unknown. The following code (the same as that in (c)) takes 163 cycles to execute on the vector processor:

```
VLD   VR0 ← mem[R0]
VLD   VR1 ← mem[R1]
VADD VR2 ← VR1, VR0
VSHR VR2 ← VR2
VST  VR2 → mem[R2]
```

The VLEN of the instructions is 16. The elements loaded in VR0 are placed in consecutive banks, the elements loaded in VR1 are placed in consecutive banks, and the elements of VR2 are also stored in consecutive banks. What is the number of banks of the new memory module? Explain.

5 DRAM Refresh [80 points]

A memory system is composed of eight banks, and each bank contains 32K rows. The row size is 8 KB.

Every DRAM row refresh is initiated by a command from the memory controller, and it refreshes a single row. Each refresh command keeps the command bus busy for 5 ns.

We define *command bus utilization* as a fraction of the total time during which the command bus is busy due to refresh.

The retention time of each row depends on the temperature (T). The rows have different retention times, as shown in the following Table 1:

Retention Time	Number of rows
$(128 - T)$ ms, $0°C \leq T \leq 128°C$	2^8 rows
$2 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	2^{16} rows
$4 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	all other rows
$8 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	2^8 rows

Table 1: Retention time

5.1 Refresh Policy A [20 points]

Assume that the memory controller implements a refresh policy where all rows are refreshed with a fixed refresh interval, which covers the worst-case retention time (Table 1).

(a) [5 points] What is the maximum temperature at which the DRAM can operate reliably with a refresh interval of 32 ms?

(b) [15 points] What command bus utilization is directly caused by DRAM refreshes (with refresh interval of 32 ms)?

5.2 Refresh Policy B [15 points]

Now assume that the memory controller implements a refresh policy where all rows are refreshed only *as frequently as required* to correctly maintain their data (Table 1).

(a) [15 points] How many refreshes are performed by the memory controller during a 1.024 second period? (with T=64°C)

5.3 Refresh Policy C [25 points]

Assume that the memory controller implements an even smarter policy to refresh the memory. In this policy, the refresh interval is fixed, and it covers the worst-case retention time (64ms), as the refresh policy in part 5.1. However, as an optimization, a row is refreshed only if it has *not* been **accessed** during the past refresh interval. For maintaining correctness, if a cell reaches its maximum retention time without being refreshed, the memory controller issues a refresh command.

(a) [5 points] Why does a row *not* need to be refreshed if it was accessed in the past refresh interval?

(b) [20 points] A program accesses all the rows repeatedly in the DRAM. The following table shows the access interval of the rows, the number of rows accessed with the corresponding access interval, and the retention times of the rows that are accessed with the corresponding interval.

Access interval	**Number of rows**	**Retention times**
1ms	2^{16} rows	64ms, 128ms or 256ms
60ms	2^{16} rows	64ms, 128ms or 256ms
128ms	all other rows	128ms or 256ms

What command bus utilization is directly caused by DRAM refreshes?

5.4 Refresh Policy D [20 points]

Assume that the memory controller implements an approximate mechanism to reduce refresh rate using Bloom filters, as we discussed in class. For this question we assume the retention times in Table 1 with a constant temperature of $64°C$.

One Bloom filter is used to represent the set of all rows that require a 64 ms refresh rate.

The memory controller's refresh logic is modified so that on every potential refresh of a row (every 64 ms), the refresh logic probes the Bloom filter. If the Bloom filter probe results in a "hit" for the row address, then the row is refreshed. Any row that does *not* hit in the Bloom filter is refreshed at the default rate of once per 128 ms.

(a) [20 points] The memory controller performs 2107384 refreshes in total across the channel over a time interval of 1.024 seconds. What is the false positive rate of the Bloom filter? (NOTE: False positive rate = Total number of false positives / Total number of accesses).

- Hint: $2107384 = 2^3 * (2^{18} + 2^{11} - 2^{10} + 2^9 - 2^8 - 1)$

6 Caching vs. Processing-in-Memory [80 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) [15 points] What is the execution time of the above piece of code in cycles?

(b) [25 points] 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.

(c) [15 points] 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.)

(d) [15 points] 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?

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

7 Cache Reverse Engineering [60 points]

Consider a processor using a 4-block LRU-based L1 data cache. Starting with an empty cache, an application accesses three L1 cache blocks in the following order, where consecutive numbers (e.g., n, $n+1$, $n+2, ...$) represent the starting addresses of consecutive cache blocks in memory:

$$n \rightarrow n+2 \rightarrow n+4$$

7.1 Part I: Vulnerability [35 points]

A malicious programmer realizes she can reverse engineer the number of sets and ways in the L1 data cache by issuing *just* two more accesses and observing *only* the cache hit rate across these two accesses. Assume that she can insert the malicious accesses only after the above three accesses of the program.

1. [15 points] What are the next two cache blocks she should access? (e.g., [n+?, n+?])

2. [10 points] How many L1 sets/ways would there be if the cache hit rate over the 2 extra instructions was:

L1 hit rate	# sets	# ways
100%		
50%		
0%		

3. [10 points] What should the next two accesses be if the replacement policy had been Most Recently Used (MRU)? (e.g., [n+?, n+?])

7.2 Part II: Exploitation [25 points]

Assuming the original cache design (i.e., with an LRU replacement policy) is using a 1-set (4-way) configuration, the malicious programmer decides to disrupt a high-security banking application, allowing her to transfer large amounts of foreign assets into her own secure Swiss account. By using a carefully designed concurrently running process to interfere with the banking application, she would like to induce slowdown, thereby exposing opportunity for her mischief.

Assume that the unmodified banking application issues the following access pattern, where each number represents X in $n + X$:

$$0 \rightarrow 6 \rightarrow 1 \rightarrow 7 \rightarrow 6 \rightarrow 1 \rightarrow 4 \rightarrow 0 \rightarrow 5 \rightarrow 0 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 7 \rightarrow 2 \rightarrow 4$$

1. [10 points] What is the unmodified banking application's cache hit rate?

2. [15 points] Now, assume that the malicious programmer knows the access pattern of the banking application. Using this information, she is able to inject a **single** extra cache access in between each of the banking application's accesses (i.e., she can interleave a malicious access pattern with the normal application's execution).

   What is the minimum cache hit rate (not counting extra malicious accesses) that the malicious programmer can induce for the banking application?

8 Emerging Memory Technologies [30 points]

Computer scientists at ETH developed a new memory technology, ETH-RAM, which is non-volatile. The access latency of ETH-RAM is close to that of DRAM while it provides higher density compared to the latest DRAM technologies. ETH-RAM has one shortcoming, however: it has limited endurance, i.e., a memory cell stops functioning after 10^6 writes are performed to the cell (known as cell wear-out).

A bright ETH student has built a computer system using 1 GB of ETH-RAM as main memory. ETH-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes over all of the cells of the main memory.

(a) [15 points] This student is worried about the lifetime of the computer system she has built. She executes a test program that runs special instructions to bypass the cache hierarchy and repeatedly writes data into different words until **all** the ETH-RAM cells are worn-out (stop functioning) and the system becomes useless. The student's measurements show that ETH-RAM stops functioning (i.e., all its cells are worn-out) in one year (365 days). Assume the following:

- The processor is in-order and there is no memory-level parallelism.
- It takes 5 ns to send a memory request from the processor to the memory controller and it takes 28 ns to send the request from the memory controller to ETH-RAM.
- ETH-RAM is word-addressable. Thus, each write request writes 4 bytes to memory.

What is the write latency of ETH-RAM? Show your work.

(b) [15 points] ETH-RAM works in the multi-level cell (MLC) mode in which each memory cell stores 2 bits. The student decides to improve the lifetime of ETH-RAM cells by using the single-level cell (SLC) mode. When ETH-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency decreases by 70%. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with everything else remaining the same in the system? Show your work.

9 Branch Prediction [70 points]

A processor implements an *in-order* pipeline with *12 stages.* Each stage completes in a single cycle. The pipeline stalls on a conditional branch instruction until the condition of the branch is evaluated. However, you *do not* know at which stage the branch condition is evaluated. Please answer the following questions.

(a) [15 points] A program with 1000 dynamic instructions completes in 2211 cycles. If 200 of those instructions are conditional branches, at the end of which pipeline stage the branch instructions are resolved? (Assume that the pipeline does not stall for any other reason than the conditional branches (e.g., data dependencies) during the execution of that program.)

(b) In a new, higher-performance version of the processor, the architects implement a *mysterious* branch prediction mechanism to improve the performance of the processor. They keep the rest of the design exactly the same as before. The new design with the mysterious branch predictor completes the execution of the following code in 115 cycles.

```
MOV R1, #0 // R1 = 0

LOOP_1:
    BEQ R1, #5, LAST // Branch to LAST if R1 == 5
    ADD R1, R1, #1   // R1 = R1 + 1
    MOV R2, #0       // R2 = 0
LOOP_2:
    BEQ R2, #3, LOOP_1 // Branch to LOOP_1 if R2==3.
    ADD R2, R2, #1     // R2 = R2 + 1
    B LOOP_2           // Unconditional branch to LOOP_2

LAST:
    MOV R1, #1         // R1 = 0
```

Assume that the pipeline never stalls due to a data dependency. Based on the given information, determine which of the following branch prediction mechanisms could be the *mysterious* branch predictor implemented in the new version of the processor. For each branch prediction mechanism below, you should circle the configuration parameters that makes it match the performance of the mysterious branch predictor.

i) [15 points] **Static Branch Predictor**
Could this be the mysterious branch predictor?

`YES` `NO`

If YES, for which configuration below is the answer *YES*? Pick an option for each configuration parameter.

i. Static Prediction Direction

`Always taken` `Always not taken`

Explain:

ii) [15 points] **Last Time Branch Predictor**
Could this be the mysterious branch predictor?

`YES` `NO`

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

i. Initial Prediction Direction

`Taken` `Not taken`

ii. Local for each branch instruction (PC-based) or global (shared among all branches) history?

`Local` `Global`

Explain:

iii) [10 points] **Backward taken, Forward not taken (BTFN)**
Could this be the mysterious branch predictor?

`YES` `NO`

Explain:

iv) [15 points] **Two-bit Counter Based Prediction** (using saturating arithmetic)
Could this be the mysterious branch predictor?

`YES` `NO`

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

i. Initial Prediction Direction

```
00 (Strongly not taken)        01 (Weakly not taken)
10 (Weakly taken)              11 (Strongly taken)
```

ii. Local for each branch instruction (i.e., PC-based, without any interference between different branches) or global (i.e., a single counter shared among all branches) history?

`Local` `Global`

Explain:

10 DRAM Scheduling and Latency [80 points]

You would like to understand the configuration of the DRAM subsystem of a computer using reverse engineering techniques. Your current knowledge of the particular DRAM subsystem is limited to the following information:

- The physical memory address is 16 bits.
- The DRAM subsystem consists of a single channel and 4 banks.
- The DRAM is byte-addressable.
- The most-significant 2 bits of the physical memory address determine the bank.
- The DRAM command bus operates at 500 MHz frequency.
- The memory controller issues commands to the DRAM in such a way that *no command* for servicing a *later* request is issued before issuing a READ command for the current request, which is the oldest request in the request buffer. For example, if there are requests A and B in the request buffer, where A is the older request and the two requests are to different banks, the memory controller does *not* issue an ACTIVATE command to the bank that B is going to access *before* issuing a READ command to the bank that A is accessing.

You realize that you can observe the memory requests that are waiting to be serviced in the request buffer. At a particular point of time, you take the snapshot of the request buffer and you observe the following requests in the request buffer.

Requests in the request buffer (in descending order of request age, where the oldest request is on the top):

time

```
Read 0x4C80
Read 0x0140
Read 0x4EC0
Read 0x8000
Read 0xF000
Read 0x803F
Read 0x4E80
```

At the same time you take the snapshot of the request buffer, you start probing the DRAM command bus. You observe the DRAM command type and the cycle (relative to the first command) at which the command is seen on the DRAM command bus. The following are the DRAM commands you observe on the DRAM bus while the requests above are serviced.

```
Cycle 0 --- PRECHARGE
Cycle 6 --- ACTIVATE
Cycle 10 --- READ
Cycle 11 --- READ
Cycle 21 --- PRECHARGE
Cycle 27 --- ACTIVATE
Cycle 31 --- READ
Cycle 32 --- ACTIVATE
Cycle 36 --- READ
Cycle 37 --- READ
Cycle 38 --- READ
Cycle 42 --- PRECHARGE
Cycle 48 --- ACTIVATE
Cycle 52 --- READ
```

Answer the following questions using the information provided above.

(a) [15 points] What are the following DRAM timing parameters used by the memory controller, in terms of nanoseconds?

i) ACTIVATE-to-READ latency

ii) ACTIVATE-to-PRECHARGE latency

iii) PRECHARGE-to-ACTIVATE latency

(b) [20 points] What is the row size in bytes? Explain your answer.

(c) [20 points] What is the status of the banks *prior* to the execution of any of the the above requests? In other words, which rows from which banks were open immediately prior to issuing the DRAM commands listed above? Fill in the table below indicating whether a bank has an open row, and if there is an open row, specify its address. If there is not enough information to infer the open row address, write *unknown*.

	Open or Closed?	**Open Row Address**
Bank 0		
Bank 1		
Bank 2		
Bank 3		

(d) [25 points] To improve performance, you decide to implement the idea of Tiered-Latency DRAM (TL-DRAM) in the DRAM chip. Assume that a bank consists of a single subarray. With TL-DRAM, an entire bank is divided into a near-segment and far-segment. When accessing a row in the near-segment, the ACTIVATE-to-READ latency *reduces* by 2 cycles and the ACTIVATE-to-PRECHARGE latency reduces by 5 cycles. When accessing a row in the far-segment, the ACTIVATE-to-READ latency *increases* by 1 cycle and the ACTIVATE-to-PRECHARGE latency increases by 2 cycles.

Assume that the rows in the near-segment have smaller row ids compared to the rows in the far-segment. In other words, physical memory row addresses 0 through $N - 1$ are the near-segment rows, and physical memory row addresses N through $M - 1$ are the far-segment rows.

If the above DRAM commands are issued 5 cycles faster with TL-DRAM compared to the baseline (the last command is issued in cycle 47), how many rows are in the near-segment? Show your work.