Name: SOLUTIONS Student ID:

Midterm Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2017

Prof. Onur Mutlu

Problem 1 (30 Points):	
Problem 2 (80 Points):	
Problem 3 (90 Points):	
Problem 4 (40 Points):	
Problem 5 (70 Points):	
Problem 6 (90 Points):	
Problem 7 (70 Points):	
Problem 8 (BONUS: 80 Points):	
Total (550 (470 + 80 bonus) Points):	

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 6 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You may be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

This page intentionally left blank

1 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.

$t_{wear\_out} = \frac{2^{30}}{2^2} \times 10^6 \times (t_{write\_MLC} + 5 + 28)$
$365 \times 24 \times 3600 \times 10^9\text{ns} = 2^{28} \times 10^6 \times (t_{write\_MLC} + 33)$
$t_{write\_MLC} = \frac{365 \times 24 \times 3600 \times 10^3}{2^{28}} - 33 = 84.5\text{ns}$

Explanation:

- Each memory cell should receive 10^6 writes.
- Since ETH-RAM is word addressable, the required amount of writes is equal to $\frac{2^{30}}{2^2} \times 10^6$ (there is no problem if 1 GB is assumed to be equal to 10^9 bytes).
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to $t_{write\_MLC} + 5 + 28$.

(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.

$t_{wear\_out} = \frac{2^{29}}{2^2} \times 10^7 \times (25.35 + 5 + 28) \times 10^{-9}$
$t_{wear\_out} = 78579686.3\text{s} = 2.49$ year

Explanation:

- Each memory cell should receive $10 \times 10^6 = 10^7$ writes.
- The memory capacity is reduced by 50% since we are using SLC: $Capacity = 2^{30}/2 = 2^{29}$
- The required amount of writes is equal to $\frac{2^{29}}{2^2} \times 10^7$.
- The SLC write latency is $0.3 \times t_{write\_MLC}$: $t_{write\_SLC} = 0.3 \times 84.5 = 25.35$ ns

2 Cache Performance Analysis [80 points]

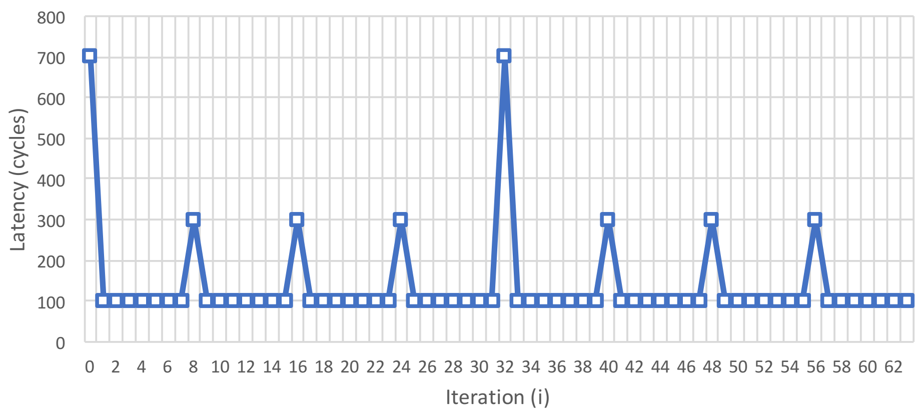
We are going to microbenchmark the cache hierarchy of a computer with the following two codes. The array `data` contains 32-bit unsigned integer values. For simplicity, we consider that accesses to the array `latency` bypass all caches (i.e., `latency` is *not* cached). `timer()` returns a timestamp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



L1 cache block size is 32 bytes, and L2 cache block size is 128 bytes.

Explanation:
The highest latency values (700 cycles) correspond to L2 cache misses. The latency of 300 cycles correspond to L1 cache misses that hit the L2 cache. The lowest latency (100 cycles) corresponds to L1 cache hits. We find L1 misses every 8 32-bit elements, and L2 misses every 32 32-bit elements. Thus, L1 cache blocks are of size 32 bytes, while L2 cache blocks are 128 bytes.

(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).

The maximum number of ways is 32.

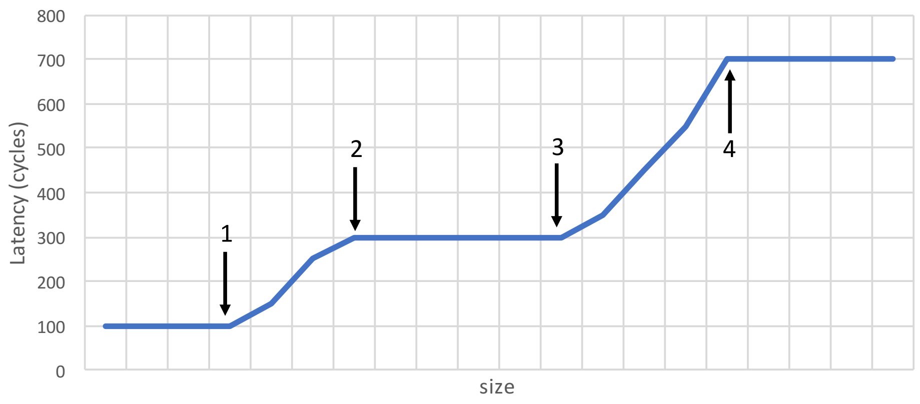
Explanation:
In L1 there are a total of 128 cache blocks. If the accessed space is 1056 elements, 33 cache blocks are read. In case of having more than 32 ways, there would be a hit in the second loop. However, with 32 or less ways, the cache block that contains `data[0]` is replaced in the last iteration of the first loop.

(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).

It is FIFO.

Explanation:
In the second loop of code (2), the last cache block that is accessed (`data[1024]`) replaces the cache block that contains `data[0]`, if the replacement policy is FIFO. If the replacement policy is LRU, the LRU cache block (the one that contains `data[32]`). Because we observe that `latency[1]` is 100 cycles, and it corresponds to the access to `data[32]`, we conclude the replacement policy is FIFO.

(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:

The entire array fits in L1. In arrow 1 the size of the array is the same as the size of L1 (4kB).

Between arrow 1 and arrow 2:

Some accesses in the second run hit in L1 and other accesses hit in L2.

Between arrow 2 and arrow 3:

All accesses in the second run hit in L2.

Between arrow 3 and arrow 4:

Still some accesses in the second run hit in L2. Arrow 3 marks the size of the L2 cache.

After arrow 4:

The accesses of the second run always miss in L2, and it is necessary to access main memory.

Explain as needed (if you need more):

3 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?

32 warps

Explanation: number of warps = $\lceil$ (number of elements) / (warp size) $\rceil$ = $\lceil 1008/32 \rceil$ = 32 warps

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

0.984375

Explanation: We have 31 fully-utilized warps and one warp with only 16 active threads. In case with no branch divergence, the SIMD utilization would be: $(32 \times 31 + 16)/(32 \times 32) = 0.984375$

(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)

For each 32 consecutive elements: ((B[i]==0 || C[i]==0) || (B[i] < 0 & C[i] < 0) || (B[i] > 0 & C[i] > 0)) || ((B[i] < 0 & C[i] > 0) || (B[i] > 0 & C[i] < 0))

Explanation: We can have two possibilities:
(1) No threads inside a warp take the first "if". It is possible if for 32 consecutive elements, the following condition is true:
(B[i]==0 || C[i]==0) || (B[i] < 0 & C[i] < 0) || (B[i] > 0 & C[i] > 0)

(2) All threads inside a warp take the first "if". It is possible if for 32 consecutive elements, the following condition is true:
(B[i] < 0 & C[i] > 0) || (B[i] > 0 & C[i] < 0)
This condition also ensures that for the second "if", all threads take the branch or no threads take the branch.
Finally, for every 32 consecutive elements, we should have one of the mentioned possibilities. For the warp with 16 active threads, we should have one of mentioned possibilities for every 16 consecutive elements

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

$((32+32+4)\times31+16+16+4)/(32\times6\times32)=0.3489$

Explanation: The first two lines must be executed by every thread in a warp. The minimum utilization results when a single thread from each warp passes both conditions, and every other thread fails to meet the condition on the first "if". The thread per warp that meets both conditions, executes four instructions.

(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)

Exactly 1 of every 32 consecutive elements in arrays B and C should have this condition:
B[i] > 0 & C[i] < 0

Explanation: Only one thread in a warp should pass the first condition. Therefore, exactly 1 of every 32 consecutive elements in arrays B and C should have the following condition:
(B[i] < 0 & C[i] > 0) || (B[i] > 0 & C[i] < 0)
However, the thread in a warp which passes the first condition should also pass the second condition. Therefore, the only condition which ensures the minimum possible SIMD utilization is: exactly 1 of every 32 consecutive elements in arrays B and C should have this condition: B[i] > 0 & C[i] < 0. For the warp with 16 active threads, this condition should be true for exactly 1 of the 16 elements.

(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? Explain your answer (Hint: The *maximum* SIMD utilization can happen under the conditions you found in part (e)).

$((32+32)\times 31+16+16+32\times 4)/((32+32)\times 32+32\times 4) = 0.985$

Explanation: DWF can ideally merge 32 warps with only one active threads in a single warp. This could be happen if none of the threads inside the warp have overlaps. If it happens, we will run 31 fully utilized warps and one warp with 16 active threads for the first and second instructions. Then, we will have only one fully-utilized warp which executes the remaining 4 instructions.

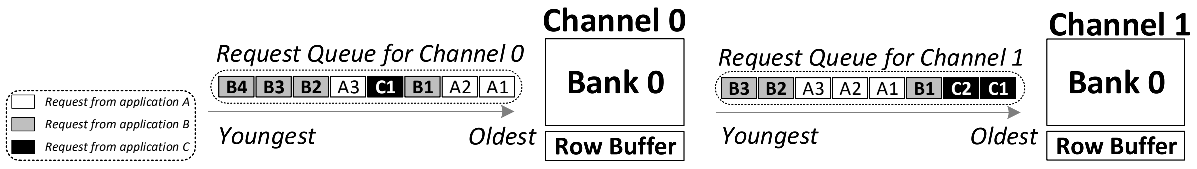
4 Memory Scheduling [40 points]

4.1 Basics and Assumptions

To serve a memory request, the memory controller issues one or multiple DRAM commands to access data from a bank. There are four different DRAM commands as discussed in class.

- `ACTIVATE`: Loads the row (that needs to be accessed) into the bank's row-buffer. This is called opening a row. **(Latency: 15ns)**
- `PRECHARGE`: Prepares the bank for the next access by closing the row in the bank (and making the row buffer empty). **(Latency: 15ns)**
- `READ/WRITE`: Accesses data from the row-buffer. **(Latency: 15ns)**

The diagrams below show the snapshots of memory controller's *request queues* at time 0, i.e., t_0, when applications A, B, and C are executed together on a multi-core processor. Each application runs on a separate core but shares the memory subsystem with other applications. Each request is color-coded to denote the application to which it belongs. Additionally, each request is annotated with a number that shows the order of the request among the set of enqueued requests of the application to which it belongs. For example, A3 means that this is the third request from application A enqueued in the request queue. Assume all memory requests are reads and a read request is considered to be served when the `READ` command is complete (i.e., 15 ns after the request's `READ` command is issued).



Request from application A
Request from application B
Request from application C
Request Queue for Channel 0
B4 B3 B2 A3 C1 B1 A2 A1
Youngest
Oldest
Channel 0
Bank 0
Row Buffer
Request Queue for Channel 1
B3 B2 A3 A2 A1 B1 C2 C1
Youngest
Oldest
Channel 1
Bank 0
Row Buffer

Assume also the following:

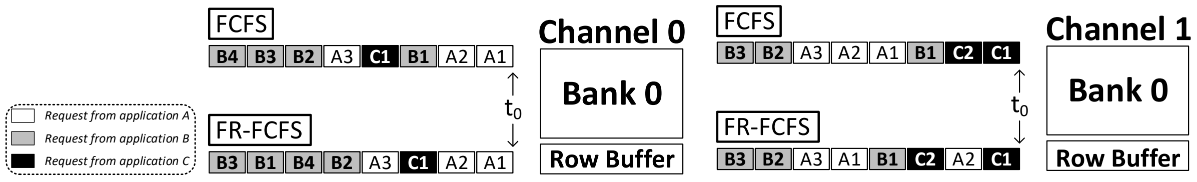
- The memory system has two DRAM channels, one DRAM bank per channel, and four rows per bank.
- All the row-buffers are closed (i.e., empty) at time 0.
- All applications start to stall at time 0 because of memory.
- No additional requests from any of the applications arrive at the memory controller.
- An application (A, B, or C) is considered to be stalled until *all* of its memory requests (across all the request buffers) have been served.

4.2 Problem Specification

The below table shows the stall time of applications A, B, and C with the FCFS (First-Come, First-Served) and FR-FCFS (First-Ready, First-Come, First-Served) scheduling policies.

Scheduling	Application A	Application B	Application C
FCFS	195 ns	285 ns	135 ns
FR-FCFS	135 ns	225 ns	90 ns

The diagrams below show the scheduling order of requests for Channel 0 and Channel 1 with the FCFS and FR-FCFS scheduling policies.



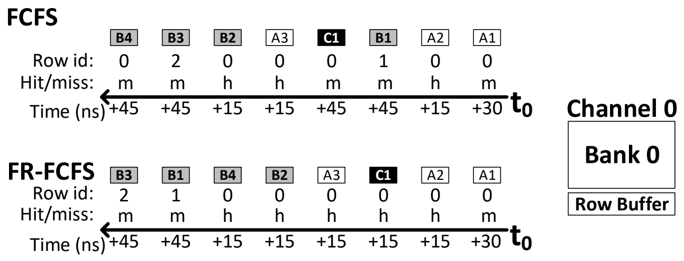
What are the numbers of row hits and row misses for each DRAM bank with either of the scheduling policies? Show your work.

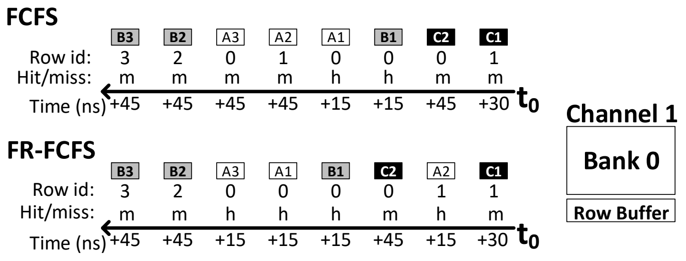
Channel 0, hits:	FCFS: 3, FR-FCFS: 5
Channel 0, misses:	FCFS: 5, FR-FCFS: 3
Channel 1, hits:	FCFS: 2, FR-FCFS: 4
Channel 1, misses:	FCFS: 6, FR-FCFS: 4

Extra space for explanation (use only if needed):

To calculate the number of hits and misses we should consider the following facts:

- The first request of each channel will be always a row-buffer miss and it requires one `ACTIVATE` and one `READ` command which lead to a 30 ns delay.
- For all requests in each channel, except for the first one, a row-buffer miss requires one `PRECHARGE`, one `ACTIVATE`, and one `READ` command which lead to a 45 ns delay.
- A row-buffer hit requires one `READ` command which leads to a 15 ns delay.
- The stall time of each application is equal to the maximum service time of its requests in both Channel 0 and Channel 1.
- When using the FR-FCFS policy, the requests will be reordered to exploit row buffer locality. For example, the B1 request in Channel 0 is reordered in FR-FCFS with respect to FCFS and executed after C1, A3, B2, and B4 requests. This means that A2, C1, A3, B2, and B4 are all accessing the same row.


FCFS
B4 B3 B2 A3 C1 B1 A2 A1
Row id: 0 2 0 0 0 1 0 0
Hit/miss: m m h h m m h m
Time (ns) +45 +45 +15 +15 +45 +45 +15 +30 t0
Channel 0
Bank 0
Row Buffer
FR-FCFS
B3 B1 B4 B2 A3 C1 A2 A1
Row id: 2 1 0 0 0 0 0 0
Hit/miss: m m h h h h h m
Time (ns) +45 +45 +15 +15 +15 +15 +15 +30 t0


FCFS
B3 B2 A3 A2 A1 B1 C2 C1
Row id: 3 2 0 1 0 0 0 1
Hit/miss: m m m m h h m m
Time (ns) +45 +45 +45 +45 +15 +15 +45 +30 t0
Channel 1
Bank 0
Row Buffer
FR-FCFS
B3 B2 A3 A1 B1 C2 A2 C1
Row id: 3 2 0 0 0 0 1 1
Hit/miss: m m h h h m h m
Time (ns) +45 +45 +15 +15 +15 +45 +15 +30 t0

5 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.)

At the end of the 7th stage.

Explanation: $Total\ cycles = 12 + 1000 + 200 * X - 1$
$2211 = 1011 + 200 * X$
$1400 = 200 * X$
$X = 6$
Each branch causes 6 idle cycles (bubbles), thus branches are resolved at the end of 7th stage.

(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:

YES, if the static prediction direction is *always not taken*.

Explanation: Such a predictor makes 6 mispredictions, which is the number resulting in 115 cycles execution time for the above program.

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:

NO.

Explanation: There is not a configuration for this branch predictor that results in 6 mispredictions for the above program.

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

`YES` [✓ `NO`]

Explain:

NO.

Explanation: BTFN predictor does not make exactly 6 mispredictions for the above program.

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:

YES, if *local* history registers with *00* or *01* initial values are used.

Explanation: Such a configuration yields 6 mispredictions, which results in 115 cycles execution time for the above program.

6 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.

Array processor

Explanation: An array processor requires 16 functional units for an operation whereas a vector processor requires only 1.

(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.

VLEN: 10

Explanation: 5+(VLEN-1)+5+(VLEN-1)+15+(VLEN-1) = 52 ⇒ VLEN = 10

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

25 cycles

Explanation: there are data dependencies among instructions ⇒ 5+5+15= 25 cycles

(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.

VLEN: 8

Explanation: 20+20+(VLEN-1)+5+(VLEN-1)+1+(VLEN-1)+20+(VLEN-1) = 94.
⇒ VLEN = 8

(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.

[Correction] The number of cycles should be 170 instead of 163. For grading this question the instructor took into account only the student's reasoning.

Number of banks: 8

Explanation: Assuming that the number of banks is power of two, 20*(16/banks)+ 20*(16/banks)+ (banks-1)+ 5+ (VLEN-1)+ 1+ (VLEN-1)+ 20*(16/banks)+ (banks-1) = 170 ⇒ banks=8

7 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 ands 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?

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

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) [20 points] 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) [20 points] 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) [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.

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

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

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

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

8 BONUS: 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?

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) [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.

900 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 1910 cycles.

Here is the access pattern for the first outer loop iteration:

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

The first 4 references are loads, the last (A[0]) is a store. The cache is initially empty. We have a cache miss for A[0]. A[0] and A[1] is fetched to 0th index in the cache. Then, B[0] is a miss, and it is conflicting with A[0]. So, A[0] and A[1] are evicted. Similarly, all cache blocks in the first iteration are conflicting with each other. Since we have only cache misses, the latency for those 5 references is $5 * 50 = 250$ cycles

The status of the cache after making those seven references is:

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

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

Cache hits/misses in the order of the references:
H, M, M, H, M
$Latency = 2 * 5 + 3 * 50 = 165$ cycles

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

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

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

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

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

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

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

$4 - A[4], B[0], A[5], B[1], A[4]$

Cache hits/misses:
H, H, H, H, H
$Latency : 5 * 5 = 25$ cycles

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

After this point, single-miss and zero-miss (all hits) iterations are interleaved until the 16th iteration.

Overall Latency:
$165 + 70 + (70 + 25) * 7 = 900$ cycles

(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.)

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) [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?

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) [10 points] 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.