Family Name: SOLUTIONS First Name: Student ID:

Midterm Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2018

Prof. Onur Mutlu

Problem 1 (60 Points):	Cache Forward Engineering	
Problem 2 (60 Points):	Reverse Engineering Caches	
Problem 3 (60 Points):	DRAM Scheduling and Latency	
Problem 4 (60 Points):	DRAM Refresh	
Problem 5 (50 Points):	GPUs and SIMD	
Problem 6 (70 Points):	Vector Processing	
Problem 7 (60 Points):	Branch Prediction	
Total (420 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.

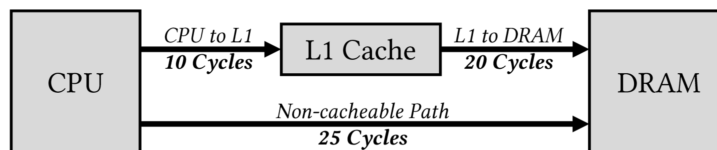
This page intentionally left blank

1 Cache Forward Engineering [60 points]

Consider a processor with the following configuration:

- In-order execution.
- Single L1 data cache that:
  - Is fully associative.
  - Has a capacity of 2 cache blocks.
  - Has a programmable replacement policy between LRU and Belady's Algorithm.

The system configuration looks like the following diagram:



There are two memory access paths:

- Through the L1 data cache, resulting in a miss latency of 30 cycles and a hit latency of 10 cycles
- Directly accessing memory (i.e., an *uncached* access) with a fixed latency of 25 cycles. Note that this type of access will bypass the cache entirely and will not affect the cache state.

However, only one path may be used at a time (i.e., all memory accesses are serialized).

You are interested in optimizing the execution time of the following performance-critical code, where LOAD($\mathbf{BLK_n}$) represents a load to the cache block address n:

```
1  set_cache_replacement_policy(DESIRED_POLICY); // LRU or Belady's
2
3  for(i = 0; /* infinite loop */; i = (i + 1) % 5)
4  {
5      if(i < 3)
6          LOAD(BLK1); // Load #1
7      if(i > 0 && i < 4)
8          LOAD(BLK2); // Load #2
9      if(i > 1)
10         LOAD(BLK3); // Load #3
11 }
```

This code results in accesses to three L1 cache block addresses in the following order (organized hierarchically in row-major format with one row per loop iteration for easier understanding):

$$\begin{array}{llllll}
\mathbf{BLK_1} & \rightarrow & & & & \\
\mathbf{BLK_1} & \rightarrow & \mathbf{BLK_2} & \rightarrow & & \\
\mathbf{BLK_1} & \rightarrow & \mathbf{BLK_2} & \rightarrow & \mathbf{BLK_3} & \rightarrow \\
 & & \mathbf{BLK_2} & \rightarrow & \mathbf{BLK_3} & \rightarrow \\
 & & & & \mathbf{BLK_3} & \rightarrow \textbf{repeat}
\end{array}$$

In this system, you have two degrees of freedom available to you for optimizing performance:

1. You may replace *any* of the three LOAD instructions on Lines 6, 8, or 10 with *uncached* LOAD (i.e., LOAD_UC) instructions.
2. You may choose between the *LRU* and the *Belady's* cache replacement policies on Line 1. For your reference:
   - The LRU replacement policy evicts the *least-recently used* cache line.
   - The Belady's replacement policy evicts the cache line that will be used *furthest into the future*.

Your task is to determine the *best performing* configuration.

1. [30 points] Fill in the following table assuming steady-state operation. You will need to provide the cache miss rate and the average latency for one iteration of the for loop. Assume that execution time is dominated by memory accesses such that there is no memory-level parallelism and the overall application latency can be computed directly from the memory access latencies.

Uncached Load Instructions	LRU Replacement		Belady's Replacement	
	Cache Miss Rate	Latency / Iteration	Cache Miss Rate	Latency / Iteration
None	3/9	150 Cycles	2/9	130 Cycles
#1	0/9	135 Cycles	0/9	135 Cycles
#2	0/9	135 Cycles	0/9	135 Cycles
#3	0/9	135 Cycles	0/9	135 Cycles
#1, #2	0/9	180 Cycles	0/9	180 Cycles
#1, #3	0/9	180 Cycles	0/9	180 Cycles
#2, #3	0/9	180 Cycles	0/9	180 Cycles
#1, #2, #3	0/9	225 Cycles	0/9	225 Cycles

2. [15 points] How many uncached loads are used in the overall best-performing configuration?

   0

3. [15 points] Which replacement policy is used in the overall best-performing configuration?

   Belady's Replacement

2 Reverse Engineering Caches [60 points]

You are trying to reverse-engineer the characteristics of a cache in a system, so that you can design a more efficient, machine-specific implementation of an algorithm you are working on. To do so, you have come up with three patterns that access various *bytes* in the system in an attempt to determine the following four cache characteristics:

- Cache block size (8, 16, 32, 64, or 128 B)
- Cache associativity (1-, 2-, 4-, or 8-way)
- Cache size (4 or 8 KB)
- Cache replacement policy (LRU or FIFO)

However, the only statistic that you can collect on this system is *cache hit rate* after performing the access pattern. Here is what you observe:

Sequence	Addresses Accessed (Oldest → Youngest)								Hit Rate
1.	0	4	8	16	64	128			1/2
2.	31	8192	63	16384	4096	8192	64	16384	5/8
3.	32768	0	129	1024	3072	8192			1/3

Assume that the cache is initially empty at the beginning of the first sequence, but not at the beginning of the second and third sequences. The sequences are executed back-to-back, i.e., no other accesses take place between the three sequences. Thus, **at the beginning of the second (third) sequence, the contents are the same as at the end of the first (second) sequence**.

Based on what you observe, what are the following characteristics of the cache? Explain to get points.

(a) [15 points] Cache block size (8, 16, 32, 64, or 128 B)?

64 B.

Explanation:
Cache hit rate is 1/2 in sequence 1. This means that there are 3 hits, which are necessarily in addresses 4, 8, and 16. Thus, cache block size might be 32 or 64 B.

In sequence 2, there are only three misses, which are in addresses 8192, 16384, and 4096. The remaining 5 accesses are hits. For 63 to be a hit, the cache block size should be 64 B.

(b) [15 points] Cache associativity (1-, 2-, 4-, or 8-way)?

4-way.

Explanation:
We already know that the cache block size is 64 B. Thus, there are 6 offset bits.

If 1-way, 63 would miss, since 8192 would map to the same set regardless of cache size (i.e., bits 6 to 12 are equal).

Addresses 0, 4096, 8192, 16384, and 32768 map to the same set. If 2-way, the second access to 8192 and 16384 (in sequence 2) would not hit.

If 8-way, 1024 and 3072 would map to the same set as 0, 4096, 8192, 16384, and 32768, since they all share bits 6 to 9. In that case, 0 and 8192 would be both hits in sequence 3. This is not possible because there are only two hits in sequence 3. 32768, 1024, and 3072 are compulsory misses, while 129 is a hit (address 128 was accessed by sequence 1). Thus, either 0 or 8192 should miss in sequence 3.

(c) [15 points] Cache size (4 or 8 KB)?

8 KB.

Explanation:
We know that the cache is 4-way associative. The access to address 0 in sequence 3 is a miss, because the cache block was replaced by address 32768. Thus, access to 8192 should be a hit.

If 4-way and 4 KB, 1024 and 3072 would map to the same set as 8192. In that case, 8192 would miss. So the size of the cache should be 8 KB.

(d) [15 points] Cache replacement policy (LRU or FIFO)?

LRU.

Explanation:
For 8192 to hit in sequence 3, 4096 should have been replaced by 0. So the replacement policy is LRU, because FIFO would have replaced 8192.

3 DRAM Scheduling and Latency [60 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, 2 banks, and 64 rows per bank.
- The DRAM is byte-addressable.
- The most-significant bit of the physical memory address determines the bank. The following 6 bits of the physical address determine the row.
- The DRAM command bus operates at 1 GHz 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.
- The memory controller services requests in order with respect to each bank. In other words, for a given bank, the memory controller first services the oldest request in the *request buffer* that targets the same bank. If all banks are ready to service a request, the memory controller first services the oldest request in the request buffer.

You realize that you can observe the memory requests that are waiting to be serviced in the request buffer. At a particular point in time, you take the snapshot of the request buffer and you observe the following requests in the request buffer (in descending order of request age, where the oldest request is on the top):

time ↓

```
Read 0xD780
Read 0x280C
Read 0xE4D0
Read 0x2838
```

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  --- READ
Cycle 1  --- PRECHARGE
Cycle 8  --- PRECHARGE
Cycle 13 --- ACTIVATE
Cycle 18 --- READ
Cycle 20 --- ACTIVATE
Cycle 22 --- READ
Cycle 25 --- 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? If there is not enough information to infer the value of a timing parameter, write *unknown*.

i) ACTIVATE-to-READ latency

5 ns.

Explanation. After issuing the ACTIVATE command at cycle 13, the memory controller waits until cycle 18, which indicates that the ACTIVATE-to-READ latency is 5 cycles. The command bus operates at 1 GHz, so it has 1 ns clock period. Thus, the ACTIVATE-to-READ is $5 * 1 = 5$ ns.

ii) ACTIVATE-to-PRECHARGE latency

Unknown.

Explanation. In the command sequence above, there is not a PRECHARGE command that follows an ACTIVATE command with a known issue cycle. Thus, we cannot determine the ACTIVATE-to-PRECHARGE latency.

iii) PRECHARGE-to-ACTIVATE latency

12 ns.

Explanation. The PRECHARGE-to-ACTIVATE latency can be easily seen in the first two commands at cycles 1 and 13. The PRECHARGE-to-ACTIVATE latency is 12 cycles = 12 ns.

iv) READ-to-PRECHARGE latency

8 ns.

Explanation. The READ command at cycle 0 is followed by a PRECHARGE command to the same bank at cycle 8. There are idle cycles before cycle 8, which indicates that the memory controller delayed the PRECHARGE command until cycle 8 because the timing constaints but not because the command bus was busy. Thus, the READ-to-PRECHARGE is 8 cycles, which is $8*1 = 8$ ns for the 1 GHz DRAM command bus.

v) READ-to-READ latency

4 ns.

Explanation. Bank 0 receives back-to-back reads at cycles 18 and 22. The READ-to-READ latency is 4 cycles, which is $4 * 1 = 4$ ns for the 1 GHz DRAM command bus.

(b) [20 points] What is the status of the banks *prior* to the execution of any of 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		

(c) [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 1 cycle and the ACTIVATE-to-PRECHARGE latency reduces by 3 cycles. When precharging a row in the near segment, the PRECHARGE-to-ACTIVATE latency reduces by 3 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. When precharging a row in the far segment, the PRECHARGE-to-ACTIVATE latency increases by 2 cycles. The following table summarizes the changes in the affected latency parameters.

Timing Parameter	**Near Segment Latency**	**Far Segment Latency**
ACTIVATE-to-READ	-1	$+1$
ACTIVATE-to-PRECHARGE	-3	$+2$
PRECHARGE-to-ACTIVATE	-3	$+2$

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 63 are the far-segment rows.

If the above DRAM commands are issued 2 cycles faster with TL-DRAM compared to the baseline (the last command is issued in cycle 23), how many rows are in the near segment, i.e., what is N? Show your work.

The rows in the range of [0-43] should definitely be in the near segment. Row 50 should definitely be in the far segment. Thus, N is a number between [44-50].

Explanation. There should be at least 44 rows in the near segment (rows 0 to 43) since rows until row id 43 need to be accessed with low latency to get 2 cycle reduction. The unknown open row in bank 0 should be in the near segment to get the 2 cycle improvement. Row 50 is in the far segment because if it was in the near segment, the command would have been finished in cycle 21, i.e., 4 cycles sooner instead of 2 cycles sooner. Thus, the number of rows in the near segment N is a number between 44 and 50.
Here is the new command trace:

```
Cycle 0 -- READ - Bank 1
Cycle 1 -- PRECHARGE - Bank 0, an unknown row in the near segment
Cycle 8 -- PRECHARGE - Bank 1, row 43, which is in the near
segment
Cycle 10 -- ACT - Bank 0, row 20, which is in the near segment
Cycle 14 -- READ - Bank 0
Cycle 17 -- ACTIVATE - Bank 1, Row 50, which is in the far
segment
Cycle 18 -- READ - Bank 0
Cycle 23 -- READ - Bank 1, Row 0
```

4 DRAM Refresh [60 points]

4.1 Basics [15 points]

A memory system is composed of eight banks, and each bank contains 2^{15} rows. 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 the fraction of total execution time during which the command bus is occupied.

1. [5 points] Given that the refresh interval is 64ms, calculate the command bus utilization of refresh commands. Show your work step-by-step.

   Command bus is utilized for $8 \times 2^{15} \times 5ns$ at every 64ms.
   $Utilization = (2^{18} \times 5ns)/(2^6 \times 10^6 ns) = 2^{12}/(2 \times 10^5) = 2^{11} \times 10^{-5} = 2.048\%$

2. [10 points] If 60% of all rows can withstand a refresh interval of 128 ms, how does the command bus utilization of refresh commands change? Calculate the reduction in bus utilization. Show your work step-by-step.

   At every 128 ms:

   - 60% of the rows are refreshed once.
     Command bus is busy for: $0.6 \times 8 \times 2^{15} \times 5ns = 3 \times 2^{18} ns$
   - 40% of the rows are refreshed twice.
     Command bus is busy for: $0.4 \times 8 \times 2^{15} \times 5ns \times 2 = 4 \times 2^{18} ns$

   $Utilization = (3 + 4) \times 2^{18} ns/128ms = 0.7 \times 2^{11} \times 10^{-5}$

   $Reduction = 1 - (0.7 \times 2^{11} \times 10^{-5})/(2^{11} \times 10^{-5}) = 30\%$

4.2 VRL: Variable Refresh Latency [45 points]

In this question, you are asked to evaluate "Variable Refresh Latency," proposed by Das, A et al. in DAC 2018[1]

The paper presents two key observations:

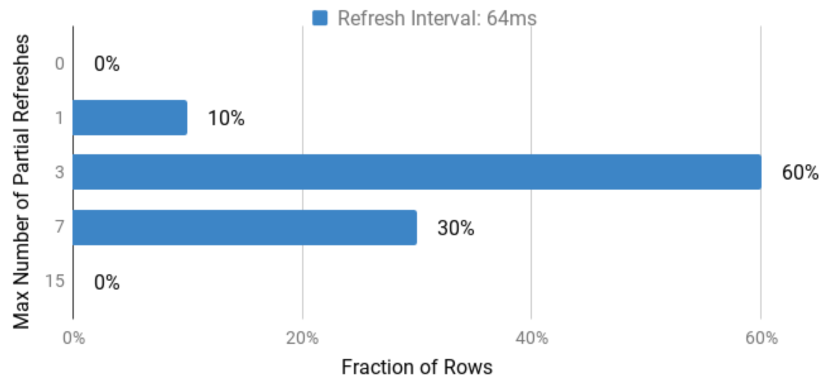
- First, a cell's charge reaches 95% of the maximum charge level in 60% of the nominal latency value during a refresh operation. In other words, the last 40% of the refresh latency is spent to increase the charge of a cell from 95% to 100%. Based on this observation, the paper defines two types of refresh operations: (1) *full refresh* and (2) *partial refresh*. Full refresh uses the nominal latency and restores the cell charge to 100%, while the latency of partial refresh is only 60% of the nominal value and it restores 95% of the charge.
- Second, a fully refreshed cell operates correctly even after multiple partial refreshes, but it needs to be fully refreshed again after a finite number of partial refreshes. The maximum number of partial refreshes before a full refresh is required varies from cell to cell.

[1] Das, A. et al., *"VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency."* In Proceedings of the 55th Annual Design Automation Conference (DAC), 2018.

The **key idea** of the paper is to apply a *full refresh* operation **only when necessary** and use *partial refresh* operations at all other times.

(a) [15 points] Consider a case in which:

- Each row must be refreshed every 64 ms. In other words, the refresh interval is 64 ms.
- Row refresh commands are evenly distributed across the refresh interval. In other words, all rows are refreshed exactly once in any given 64 ms time window.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank. For example, if the maximum number of refreshes is three, those rows can be partially refreshed for at most three refresh intervals, and the fourth refresh operation must be a full refresh.
- If all rows were always fully refreshed, the time that a bank is busy, serving the refresh requests within a refresh interval would be T.



How much time does it take (in terms of T) for a bank to refresh all rows within a refresh interval, after applying Variable Refresh Latency?

Full refresh latency = T, partial refresh latency = 0.6T.

10% of the rows are fully refreshed at every other interval:
$0.1 \times (0.5 \times 0.6T + 0.5 \times T)$
60% of the rows are fully refreshed after every three partial refresh:
$0.6 \times (0.75 \times 0.6T + 0.25 \times T)$
30% of the rows are fully refreshed after every seven partial refresh:
$0.3 \times (0.875 \times 0.6T + 0.125 \times T)$

Then, new refresh latency of a bank would be 0.695T.

(b) [15 points] You find out that you can relax the refresh interval, and define your baseline as follows:

- 90% of the rows are refreshed at every 128ms; 10% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- All rows are always fully refreshed.
- A single refresh command costs $0.2/N$ $ms.$, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.

Calculate the refresh overhead for the baseline.

At every 128ms:

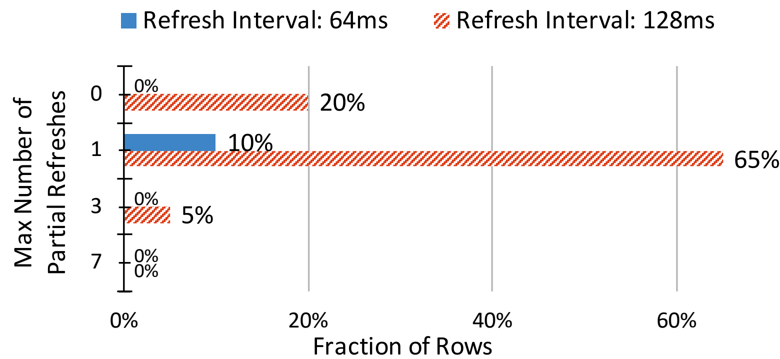
10% of the rows are refreshed twice, 90% of the rows are refreshed once.

Total time spent for refresh in a 128 ms. interval is $(0.9N + 2 \times 0.1N) \times 0.2/N = 0.22ms.$

Then refresh overhead is 0.22/128

(c) [15 points] Consider a case where:

- 90% of the rows are refreshed at every 128ms; 10% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank.
- A single refresh command costs $0.2/N\ ms.$, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.



Calculate the refresh overhead. Show your work step-by-step. Then, compare it against the baseline configuration (the previous question). How much reduction do you see in the performance overhead of refreshes?

Full refresh of a row costs 0.2/N ms. Then, partial refresh of a row costs 0.12/N ms

At every $4 \times 128\ ms$:

- 20% of the rows are refreshed for 4 times:
  4 times *fully refreshed* and 0 times *partially refreshed.*
- 10% of the rows are refreshed for 8 times:
  4 times *fully refreshed* and 4 times *partially refreshed.*
- 65% of the rows are refreshed for 4 times:
  2 times *fully refreshed* and 2 times *partially refreshed.*
- 5% of the rows are refreshed for 4 times:
  1 time *fully refreshed* and 3 times *partially refreshed.*

Total time spent for refresh is:
$= (0.2N \times 4 + 0.1N \times 4 + 0.65N \times 2 + 0.05N \times 1) \times 0.2/N$
$+ (0.2N \times 0 + 0.1N \times 4 + 0.65N \times 2 + 0.05N \times 3) \times 0.12/N$

$= (0.8 + 0.4 + 1.3 + 0.05) \times 0.2 + (0.4 + 1.3 + 0.15) \times 0.12$
$= 2.55 \times 0.2 + 1.85 \times 0.12$
$= 0.51 + 0.222 = 0.732$ ms.
Then, refresh overhead is: $0.732/(4 \times 128)$

So, the reduction is $1 - (0.732/4)/0.22 = 1.7/22 \approx 7.7\%$.

5 GPUs and SIMD [50 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.

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 64 threads, and there are 64 SIMD lanes in the GPU.

```
for (i = 0; i < 4096; i++) {
    if (B[i] < 8888) {
        A[i] = A[i] * C[i];
        A[i] = A[i] + B[i];
        C[i] = B[i] + 1;
    }
    if (B[i] > 8888) {
        A[i] = A[i] * B[i];
    }
}
```

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

> Warps = (Number of threads) / (Number of threads per warp)
> Number of threads = 2^{12} (i.e., one thread per loop iteration).
> Number of threads per warp = 64 = 2^6 (given).
> Warps = $2^{12}/2^6 = 2^6$

(b) [20 points] When we measure the SIMD utilization for this program with one input set, we find that it is 134/320. What can you say about arrays A, B, and C? Be precise (Hint: Look at the "if" branch).

A: Nothing

B: 2 in every 64 of B's elements are less than 8888, the rest are 8888

C: Nothing

(c) [10 points] Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise. If NO, explain why not.

> Yes. All consecutive 64 elements of B should be either:
> (1) All of B's elements are equal to 8888, or
> (2) All of B's elements are less than 8888, or
> (3) All of B's elements are greater than 8888.

(d) [10 points] What is the lowest SIMD utilization that this program can yield? Explain.

132/384. 1 in every 64 of B's elements are greater than 8888, and 1 in every 64 of B's elements are less than 8888, and the rest of the elements are 8888.

6 Vector Processing [70 points]

A vector processor implements the following ISA:

Opcode	Operands	Latency (cycles)	Description
LD	V_{STR}, #n	1	$V_{STR} \leftarrow$ n (V_{STR} = Vector Stride Register)
LD	V_{LEN}, #n	1	$V_{LEN} \leftarrow$ n (V_{LEN} = Vector Length Register)
LDM	V_i	1	$V_{MSK} \leftarrow LSB(V_i)$ (V_{MSK} = Vector Mask Register)
VLD	V_i, #Address	22, pipelined	$V_i \leftarrow Mem[Address]$
VST	V_i, #Address	22, pipelined	$Mem[Address] \leftarrow V_i$
VADD	V_i, V_j, V_k	4, pipelined	$V_i \leftarrow V_j + V_k$
VMUL	V_i, V_j, V_k	6, pipelined	$V_i \leftarrow V_j * V_k$
VNOT	V_i	4, pipelined	$V_i \leftarrow BitwiseNOT(V_i)$
VCMPZ	V_i, V_j	4, pipelined	if(V_j == 0) $V_i \leftarrow$ 0xFFFF; else $V_i \leftarrow$ 0x0000

Assume the following:

- The processor has an in-order core.
- The size of a vector element is 2 bytes.
- Each vector register V_i contains V_{LEN} vector elements. The total number of vector registers is 8.
- LD and LDM are not pipelined. They execute in one single cycle.
- LDM moves the least-significant bit (LSB) of each vector element in a vector register V_i into the corresponding position in V_{MSK}. This instruction is executed in one single cycle for all vector elements.
- V_{STR} and V_{LEN} are 16-bit registers. V_{MSK} has V_{LEN} bits.
- V_{MSK} enables predicated execution. Assume a simple implementation in which all V_{LEN} operations are executed, but the result writeback is turned off according to V_{MSK} (0 means writeback is turned off). Assume instructions LDM and VNOT are not subject to the mask (i.e., the result writeback is turned on for every vector element).
- The main memory is byte addressable.
- The main memory has N banks. N is a power of two. Vector elements stored in consecutive memory addresses are interleaved between the memory banks. For instance, if a vector element at address A maps to bank B, a vector element at address $A + 2$ maps to bank $(B + 1)$ % N, where % is the modulo operator and N is the number of banks.
- There is one single memory port, which is used for reads and writes.
- The processor *does not* support chaining between vector functional units.

(a) [5 points] What should the minimum value of N be to avoid additional stalls when executing a single VLD or VST instruction, assuming a vector stride of 1? Explain.

$N = 32$.

Explanation:
32 banks are needed because the latency of VLD and VST is 22 cycles, and N is a power of two.

Consider the following piece of code:

```
for (i = 0; i < 64; i++){
    if (A[i] == 0)
        B[i] = C[i];
    else
        B[i] = C[i] * A[i];
}
```

(b) [10 points] Translate the code into assembly language with the minimum number of instructions by using the provided ISA. Assume $V_{LEN} = 64$. (Hint: `C` should be loaded only once, before evaluating the `if` statement.)

```
LD VLEN, 64        # Load Vector Length Register
LD VSTR, 1         # Load Vector Stride Register
VLD V1, A          # Read from array A
VLD V2, C          # Read from array C
VCMPZ V3, V1       # Compare V1 to 0
LDM V3             # Load Vector Mask Register
VST B, V2          # Write to array B
VNOT V3            # BitwiseNOT
LDM V3             # Load Vector Mask Register
VMUL V4, V2, V1    # Multiply
VST B, V4          # Write to array B
```

(c) [15 points] What is the total number of cycles needed to execute the program in part (b)?

395 cycles.

Explanation:

```
LD      |1|
LD        |1|
VLD         | 22 | - 63 - |
VLD                       | 22 | - 63 -  |
VCMPZ                      |4| - 63 - |
LDM                                   |1|
VST                                      | 22 | - 63 -    |
VNOT                                      |4| - 63 - |
LDM                                                 |1|
VMUL                                                  |6| - 63 - |
VST                                                             | 22 | - 63 -|
```

(d) [15 points] If we execute the same code in a vector processor *supporting chaining*, what is the total number of cycles for the same program?

332 cycles.

Explanation:

```
LD       |1|
LD         |1|
VLD          | 22 | - 63 - |
VLD                        | 22 | - 63 - |
VCMPZ            |4| - 63 - |
LDM                          |1|
VST                                     | 22 | - 63 - |
VNOT                                     |4| - 63 - |
LDM                                                |1|
VMUL                                                 |6| - 63 - |
VST                                                    | 22 | - 63 - |
```

(e) [15 points] If we add one more memory port to the vector processor *supporting chaining*, what is the total number of cycles for the same program?

258 cycles.

Explanation:

```
LD       |1|
LD         |1|
VLD          | 22 | - 63 - |
VLD           | 22 | - 63 -  |
VCMPZ          |4| - 63 - |
LDM                       |1|
VST                          | 22 | - 63 -     |
VNOT                          |4| - 63 - |
LDM                                       |1|
VMUL                                        |6| - 63 - |
VST                                            | 22 | - 63 - |
```

(f) [10 points] A more efficient predicated execution is a density-time implementation, which scans the Vector Mask Register and only executes elements with non-zero masks. For the original vector processor (i.e., with one single memory port and no chaining), what would be the minimum number of cycles? (Note: Consider *no* overhead from scanning the Vector Mask Register.)

257 cycles.

Explanation:
The minimum number of cycles would be for all `A[i] = 0`, so that the last two instructions (`VMUL` and `VST`) wouldn't be executed.

7 Branch Prediction [60 points]

Consider the following high level language code segment:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;

for (i = 0; i < 1000; i ++)      // Branch 1: Loop Branch
{
    // Branch 1: Taken
    if (i % 2 == 0)              // Branch 2: If Condition 1
        // Branch 2: Taken
        if (i % 3 == 0)          // Branch 3: If Condition 2
            sum1 += array[i];    // Branch 3: Taken
        else
            sum2 += array[i];
    else
        if (i % 4 == 0)          // Branch 4: If Condition 3
            sum3 += array[i];    // Branch 4: Taken
        else
            sum4 += array[i];
}
```

(a) [20 points] What is the prediction accuracy for each of the four branches using a per-branch last-time predictor (assume that every per-branch counter starts at "not-taken")? Please show all of your work.

Branch 1:

999/1001. The branch is mispredicted the first and last time it's executed.

Branch 2:

$0/1000 \times 100 = 0\%$. The branch changes direction every time it's executed.

Branch 3:

33.2%. The pattern repeats Miss, Miss, Hit

Branch 4:

100%. The branch is never taken.

(b) [20 points] What is the prediction accuracy for each of the four branches when a per-branch 2-bit saturating counter-based predictor is used (assume that every per-branch counter starts at "strongly not-taken")? Please show all of your work.

Branch 1:

998/1001. The branch is mispredicted the first two times and last time it's executed.

Branch 2:

$500/1000 \times 100 = 50\%$. The counter changes between "strongly not-taken" to "weakly not-taken" every iteration. Branch is taken every other time.

Branch 3:

66.6%. The pattern repeats Miss, Hit, Hit

Branch 4:

100%. The branch is never taken.

(c) [20 points] What is the prediction accuracy for both Branch 2 and Branch 3, when the counter starts at (i) “weakly not-taken” and (ii) “weakly taken”?

Branch 2 (i)

(i) 0%. The counter alternates between “weakly not-taken” and “weakly taken” missing on each

Branch 2 (ii)

(ii) 50%. The counter alternates between “weakly taken” and “strongly taken” hitting every other

Branch 3 (i)

(i) 66.8%. Miss, Hit, Hit repeats

Branch 3 (ii)

(ii) 66.2%. After the predictor settles to “strongly not-taken”, the pattern repeats Miss, Hit, Hit