Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2018

Prof. Onur Mutlu

Problem 1 (40 Points):	Emerging Memory Technologies	
Problem 2 (70 Points):	Memory Scheduling	
Problem 3 (80 Points):	Asymmetric Multicore	
Problem 4 (55 Points):	Multicore Cache Partitioning	
Problem 5 (40 Points):	Cache Coherence	
Problem 6 (45 Points):	Memory Consistency	
Problem 7 (65 Points):	Processing-in-Memory	
Problem 8 (BONUS: 50 Points):	GPU Programming	
Total (445 (395 + 50 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. If you make assumptions, state your assumptions clearly and precisely.
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 [40 points]

Researchers at Lindtel developed a new memory technology, L-RAM, which is non-volatile. The access latency of L-RAM is close to that of DRAM while it provides higher density compared to the latest DRAM technologies. L-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) [15 points] Lindtel markets a new computer system with L-RAM to have a lifetime of 2 years and the following specifications:

- 4 GBs of L-RAM as main memory with a *perfect* wear-leveling mechanism, i.e., writes are equally distributed over all the cells of L-RAM.
- The processor is in-order and there is no memory-level parallelism.
- It takes 4 ns to send a memory request from the processor to the memory controller and it takes 20 ns to send the request from the memory controller to L-RAM. The write latency of L-RAM is 40 ns.
- L-RAM is word-addressable. Thus, each write request writes 8 bytes to memory.

A student at ETH tests the lifetime of the system and finds that this new computer system *cannot* guarantee a lifetime of 2 years. She writes a program to wear out the entire L-RAM device as quickly as possible. How fast is she able to wear out the device? Show all work.

$t_{wear\_out} = \frac{2^{32}}{2^3} \times 10^6 \times (40 + 4 + 20)$
$t_{wear\_out} = 2^{35} \times 10^6$ ns
$t_{wear\_out} \approx 397.68$ days

Explanation:

- Each memory cell should receive 10^6 writes.
- Since ETH-RAM is word addressable, the required number of writes is equal to $\frac{2^{32}}{2^3} \times 10^6$.
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to $40 + 4 + 20$.

(b) [15 points] L-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 L-RAM cells by using the single-level cell (SLC) mode. When L-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency decreases by 75%. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with all else remaining the same in the system? Show your work.

$t_{wear\_out} = \frac{2^{31}}{2^3} \times 10^7 \times (10 + 4 + 20) \times 10^{-9}$
$t_{wear\_out} = 91268055.04\text{s} \approx 1056.34$ days

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: $Capacity = 2^{32}/2 = 2^{31}$
- The required number of writes is equal to $\frac{2^{31}}{2^3} \times 10^7$.
- The SLC write latency is $0.25 \times t_{write\_MLC}$: $t_{write\_SLC} = 0.25 \times 40 = 10\,\text{ns}$

(c) [10 points] Provide a mechanism that would increase the guaranteed lifetime of the computer system without changing the physical circuitry of L-RAM. From the baseline computer system in part (a), describe the changes required to guarantee a computer system lifetime of 2 years, with your mechanism. Be concrete and precise.

Artificially increase the time to either (1) send a memory request from the memory controller to L-RAM or (2) send a request from the processor to the memory controller by 54 ns. $730 * 3600 * 24 < \frac{2^{32}}{2^3} \times 10^6 \times (40 + 4 + 20 + x)$
$x > 53.4$ns

2 Memory Scheduling [70 points]

In lectures, we introduced a variety of ways to tackle memory interference. In this problem, we will look at the Blacklisting Memory Scheduler (BLISS) to reduce unfairness. There are two key aspects of BLISS that you need to know.

- When the memory controller services η consecutive requests from a particular application, this application is blacklisted. We name this non-negative integer η the **Blacklisting Threshold**.
- The blacklist is cleared periodically every **10000** cycles starting at $t = 0$.

To reduce unfairness, memory requests in BLISS are prioritized in the following order:

- Non-blacklisted applications' requests
- Row buffer hit requests
- Older requests

The memory system for this problem consists of 2 channels with 2 banks each. Tables 1 and 2 show the memory request stream in the same bank for both applications at different times ($t = 0$ and $t = 10$). For both tables, a request on the left-hand side is older than a request on the right-hand side in the same table. The applications do not generate more requests than those shown in Tables 1 and 2. The memory requests are labeled with numbers that represent the row position of the data within the accessed bank. Assume the following for all questions:

- A row buffer *hit* takes **100 cycles**.
- A row buffer *miss* (i.e., opening a row in a bank with a closed row buffer) takes **200 cycles**.
- A row buffer *conflict* (i.e., closing the currently open row and opening another one) takes **250 cycles**.
- All row buffers are closed at time $t = 0$

Application A (Channel 0, Bank 0)								
Application B (Channel 0, Bank 0)	Row 2	Row 2	Row 2	Row 2	Row 2	Row 3	Row 3	Row 4

Table 1: Memory requests of the two applications at $t = 0$

Application A (Channel 0, Bank 0)	Row 3	Row 7	Row 2	Row 0	Row 5	Row 3	Row 8	Row 9
Application B (Channel 0, Bank 0)	Row 2	Row 2	Row 2	Row 2	Row 2	Row 3	Row 3	Row 4

Table 2: Memory requests of the two applications at $t = 10$. Note that none of the Application B's existing requests are serviced yet.

(a) [15 points] Compute the slowdown of each application using the FR-FCFS scheduling policy after both threads ran to completion. We define:

$$slowdown = \frac{memory\ latency\ of\ the\ application\ when\ run\ together\ with\ other\ applications}{memory\ latency\ of\ the\ application\ when\ run\ alone}$$

Show your work.

$slowdown_A = \sim 1.53$
$slowdown_B = 1.25$

Explanation:
For both applications, the first request will incur row buffer miss penalty, and the rest of the requests will either be hits or conflicts.
Application A (alone) $= 200 + 100 + 250 * 6 = 1800$ *cycles*
Application B (alone) $= 200 + 100 * 4 + 250 + 100 + 250 = 1200$ *cycles*

Applications A (with B, FR-FCFS) $= 200+100*4+100+250+100+100*2+250+250*5 = 2750$ *cycles*
Applications B (with A, FR-FCFS) $= 200 + 100 * 4 + 100 + 250 + 100 + 100 * 2 + 250 = 1500$ *cycles*

From the two tables above we know that all requests of application B were issued before any of the application A's requests were issued. Thus, all requests of B are prioritized unless there is a row hit for A's requests.
$slowdown_A = \frac{2750}{1800} = \sim 1.53$

$slowdown_B = \frac{1500}{1200} = 1.25$

(b) [15 points] If we use the BLISS scheduler, for what value(s) of η (the Blacklisting Threshold) will the slowdowns of **both** applications be equal to those obtained with FR-FCFS?

For $\eta \geq 6$ or $\eta = 0$.

Explanation:
We want both A and B to complete without blacklisting or to complete both blacklisted, thus $\eta \geq 6$ and $\eta = 0$, respectively.

(c) [15 points] For what value(s) of η (the Blacklisting Threshold) will the slowdown of A be < 1.5?

Impossible. Slowdown for A will always be ≥ 1.5

Explanation: For the give memory requests, it is not possible to find η that blacklists B but not A. Thus, the smallest slowdown for A is the case explained in the solution of part (b).

(d) [15 points] For what value(s) of η (the Blacklisting Threshold) will B experience the maximum slowdown it can possibly experience with the Blacklisting Scheduler?

For $\eta = 5$.

Explanation: We already know that the slowdowns will be equal to the slowdown with FR-FCFS when $\eta \geq 6$ or $\eta = 0$. If we execute the memory requests for the rest of possible η values, we find that $\eta = 5$ causes application B to complete after 2150 cycles, which is the largest.

(e) [10 points] What is a simple mechanism (that we discussed in lectures) that we can use instead of BLISS to make the slowdowns of both A and B equal to 1.00?

Memory Channel Partitioning (MCP)

Explanation: With MCP, each application will operate on an independent channel, without any interference with the other application.

3 Asymmetric Multicore [80 points]

A microprocessor manufacturer asks you to design an asymmetric multicore processor for modern workloads. You should optimize it assuming a workload with 80% of its work in the parallel portion. Your design contains one large core and several small cores, which share the same die. Assume the total die area is 32 units.

- *Large core*: For a large core that is n times faster than a single small core, you will need n^3 units of die area (n is a positive integer). The dynamic power of this core is $6 \times n$ Watts and the static power is n Watts.
- *Small cores*: You will fit as many small cores as possible, after placing the large core. A small core occupies 1 unit of die area. Its dynamic power is 1 Watt and its static power is 0.5 Watts.

The parallel portion executes *only* on the small cores, while the serial portion executes *only* on the large core.

Please answer the following questions. Show your work. Express your equations and solve them. You can approximate some computations, and get partial or full credit.

(a) [15 points] What configuration (i.e., number of small cores and size of the large core) results in the best performance?

One large core and 24 small cores. The large core will occupy 8 units of die area.

Explanation:
Given that the large core occupies n^3 units, the number of small cores will be $32 - n^3$. Thus, the speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32-n^3}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32-n^3}$ seconds.

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	0.20	0.03	0.23
2	24	0.10	0.03	**0.13**
3	5	0.07	0.16	0.23

These calculations can be approximated without a calculator:

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	$0.20 \,/\, 1 = 0.20$	$0.02 < 0.80 \,/\, 31 < 0.03$	> 0.22
2	24	$0.20 \,/\, 2 = 0.10$	$0.03 < 0.80 \,/\, 24 < 0.04$	$<$ **0.14**
3	5	$0.20 \,/\, 3 = 0.07$	$0.80 \,/\, 5 = 0.16$	> 0.22

(b) [10 points] The energy consumption should also be a metric of reference in your design. Compute the energy consumption for the best configuration in part (a).

$E_{total} = 26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

Explanation:
We can calculate the energy consumption as:
$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + P_{large\_static} \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 2 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

This result can be approximated without a calculator:
$E_{total} < 26 \times 0.10 + 38 \times 0.04 = 2.6 + 1.52 = 4.12$ Joules.

(c) For the best configuration obtained in part (a), you are considering to use the large core to collaborate with the small cores on the execution of the parallel portion.

(i) [10 points] What is the overall performance improvement, compared to the performance obtained in part (a), if the large core collaborates on the parallel portion?

If the large core collaborates with the small cores in the parallel portion, the best-case speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32-n^3+n}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32-n^3+n}$ seconds.

The execution time of the serial part t_{serial}, which takes significantly longer than the parallel part (about 3 times longer), does not change. By using the large core to collaborate in the parallel portion, the execution time of the parallel part $t_{parallel}$ decreases from $\frac{0.8}{24}$ to $\frac{0.8}{24+2}$, i.e., a speedup of $\frac{13}{12}$, which is less than 10%. Thus, the overall performance improvement from using the large core to collaborate in the parallel portion is negligible.

(ii) [10 points] What is the overall energy change, compared to the energy obtained in part (b), if the large core collaborates on the parallel portion?

If the large core collaborates in the parallel portion, we calculate the energy consumption as:
$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + (P_{large\_dynamic} + P_{large\_static}) \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + 7 \times n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 14 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 50 \times t_{parallel} \simeq 2.6 + 2.0 = 4.6$ Joules.

We assume that $t_{parallel}$ has a very small change, as discussed above. If we compare this equation to the energy equation in part (b), we observe that the energy consumption increases by $P_{large\_dynamic} \times t_{parallel} = 6 \times n \times t_{parallel} = 12 \times t_{parallel}$ Joules. Since the energy consumption of the parallel portion is $38 \times t_{parallel}$ Joules in part (b), there is an energy increase in the parallel portion of more than 30% (i.e., $\frac{12}{38}$). The overall energy increase is more than 11%.

(iii) [5 points] Discuss whether it is worth using the large core to collaborate with the small cores on the execution of the parallel portion.

It is not really worth using the large core in the parallel part. While the performance improvement is negligible, the overall energy consumption increases by more than 11%.

(d) [15 points] Now assume that the serial portion can be optimized, i.e., the serial portion becomes smaller. This gives you the possibility of reducing the size of the large core, and still improving performance. For a large core with an area of $(n-1)^3$, where n is the value obtained in part (a), what should be the fraction of serial portion that would lead to better performance than in part (a)?

10%.

Explanation:
We call t_{total} the total execution time with a large core with $n = 2$, as obtained in part (a), and t'_{total} for a smaller core with $n = 1$. We can obtain the new parallel fraction p from the following equation:

$t_{total} > t'_{total}$;

$0.13 > \frac{1-p}{n-1} + \frac{p}{32-(n-1)^3}$;

$0.13 > \frac{1-p}{1} + \frac{p}{31}$;

$p > 0.90$.

The serial portion should be *at most* 10%.

(e) [15 points] Your design is so successful for desktop processors that the company wants to produce a similar design for mobile devices. The power budget becomes a constraint. For a maximum of total power of 20W, how much would you need to reduce the dynamic power consumption of the large core, if at all, for the best configuration obtained in part (a)? Assume again that the parallel fraction is 80% of the workload. (Hint: Express the dynamic power of the large core as $D \times n$ Watts, where D is a constant).

We have to reduce the dynamic power consumption of the large core by *at least* 20×.

Explanation:
We calculate the total power as the total energy divided by the total execution time:
$P_{total} = \frac{E_{total}}{t_{total}}$ Watts;

$P_{total} = \frac{E_{large}+E_{small}}{t_{total}} \leq 20$ Watts;

We express the dynamic power of the large core as $D \times n$. From part (a) we know n, t_{serial}, $t_{parallel}$ and t_{total}, from part (b) we know E_{small}:

$\frac{(D+1)\times n\times t_{serial}+n\times t_{parallel}+E_{small}}{t_{total}} = \frac{(D+1)\times 2\times 0.10+n\times 0.03+2.00}{0.13} \leq 20$ Watts;

$D \leq 0.3$.

In mobile devices, the dynamic power of the large core has to be $\leq 0.3 \times n$ Watts (given the assumptions in the question). Since the dynamic power of the large core is $6 \times n$ Watts in the desktop processor, we have to reduce the dynamic power consumption of the large core by *at least* 20× for mobile devices.

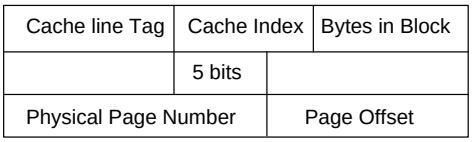
4 Multicore Cache Partitioning [55 points]

Suppose we have a system with 32 cores that share a physical second-level cache. Assume each core is running a single single-threaded application, and all 32 cores are concurrently running applications. Assume that the page size of the architecture is 8KB, the block size of the cache is 128 bytes, and the cache uses LRU replacement. We would like to ensure each application gets a *dedicated* space in this shared cache without any interference from other cores. We would like to enforce this using the OS-based page coloring mechanism to partition the cache, as we discussed in lecture. Recall that with page coloring, the operating system ensures, using virtual memory mechanisms, that the applications do not contend for the same space in the cache.

(a) [10 points] What is the minimum size the L2 cache needs to be such that each application is allocated its dedicated space in the cache via page coloring? Show your work.

256KB.

Explanation:
For OS based page coloring to work in this case, we need at least 32 colors. This means we need at least 5 bits of the cache index to intersect with the physical page number.



So, with associativity A, page size P, the minimum cache size is given by,
$C \geq A \times 2^5 \times P = A \times 32 \times P$
$C \geq A \times 32 \times 8KB = A \times 256KB$
Minimum cache size (associativity = 1) is 256KB

(b) [10 points] Assume the cache is 4MB, 32-way associative. Can the operating system ensure that the cache is partitioned such that no two applications interfere for cache space? Show your work.

No.

Explanation:
For a given associativity, minimum cache size = A × 256KB (from part a). Therefore, for a 32-way associative cache, minimum cache size required for the OS to ensure partitioning without interference is 32 × 256KB = 8MB. Since the cache size is only 4MB, the OS, in this case, cannot ensure partitioning without interference.

(c) Assume you would like to design a 32MB shared cache such that the operating system has the ability to ensure that the cache is partitioned such that no two applications interfere for cache space.

(i) [5 points] What is the minimum associativity of the cache such that this is possible? Show your work.

Minimum associativity = 1.

Explanation:
From part a),
$C \geq A \times 256KB$
$32000KB \geq A \times 256KB$
Therefore, minimum associativity = 1

(ii) [10 points] What is the maximum associativity of the 32MB cache such that this is possible? Show your work.

Maximum associativity = 128.

Explanation:
From part a),
C $\geq$ A $\times$ 256KB; A $\leq$ C / 256KB; A $\leq$ 32MB / 256KB
A $\leq$ 128
Therefore, maximum associativity is 128.

(d) [5 points] Suppose we decide to change the cache design and use utility based cache partitioning (UCP) to partition the cache, instead of OS-based page coloring. Assume we would like to design a 4MB cache with a 128-byte block size. What is the minimum associativity of the cache such that each application is guaranteed a minimum amount of space without interference? Recall that UCP aims to minimize the cache miss rate by allocating more cache ways to applications that obtain the most benefit from more ways, as we discussed in lecture.

Minimum associativity = 32.

Explanation:
Utility based cache partitioning needs to give at least one way for each application. Otherwise, the application will receive no cache space. Hence, the minimum associativity is 32.

(e) [5 points] Is it desirable to implement UCP on a cache with this minimum associativity? Why, why not? Explain.

No, it is not desirable to implement UCP.

Explanation:
There will be no benefit gained from UCP since UCP guarantees at least one way per application. This means all applications will be allocated exactly one way of the cache, i.e. the cache is equally and statically partitioned regardless of applications' utility for caching.

(f) [5 points] What is the maximum associativity of a 4MB cache that uses UCP such that each application is guaranteed a minimum amount of space without interference?

32k ways.

Explanation:
The maximum associativity corresponds to a fully associative design. For the given configuration, it is 4 MB / 128 bytes = 2^{22} / 2^7 = 2^{15} = 32k ways.

(g) [5 points] Is it desirable to implement UCP on a cache with this maximum associativity? Why, why not? Explain.

No.

Explanation:
It is not desirable to implement UCP with this maximum associativity because the overhead of UCP for 32 applications on this cache will likely outweigh its benefits. UCP will only work with LRU replacement policy. But implementing LRU on top of a 32k-way cache is impractical. Also the number of counters needed by UCP and the partitioning solution space for UCP are very large for such a cache.

5 Cache Coherence [40 points]

We have a system with 4 byte-addressable processors {P0, P1, P2, P3}. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. All caches are connected to and actively snoop a global bus, and cache coherence is maintained using the MESI protocol, as we discussed in class. Note that on a write to a cache block in the S state, the block will transition directly to the M state. Accessible memory addresses range from `0x00000` – `0xfffff`.

Each processor executes the following instructions in a *sequentially consistent* manner:

P0	
0	`st r0, 0x1ff40`
-	
-	

P1	
1	`st r0, 0x110c0`
2	`st r1, 0x11080`
3	`ld r2, 0x1ff00`

P2	
4	`ld r0, 0x1ff40`
5	`ld r1, 0x110f0`
-	

P3	
-	
-	
-	

After executing the above 6 memory instructions, the *final* tag store state of each cache is as follows:

Final Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0	`0x1ff`	`S`
Set 1	`0x1ff`	`S`
Set 2	`0x110`	`I`
Set 3	`0x110`	`I`

Cache for P1		
	Tag	*MESI state*
Set 0	`0x1ff`	`S`
Set 1	`0x1ff`	`I`
Set 2	`0x110`	`M`
Set 3	`0x110`	`M`

Cache for P2		
	Tag	*MESI state*
Set 0	`0x10f`	`I`
Set 1	`0x1ff`	`S`
Set 2	`0x10f`	`M`
Set 3	`0x110`	`I`

Cache for P3		
	Tag	*MESI state*
Set 0	`0x133`	`E`
Set 1	`0x000`	`I`
Set 2	`0x000`	`I`
Set 3	`0x10f`	`I`

(a) [30 points] Fill in the following tables with the *initial* tag store states (i.e., *Tag* and *MESI* state) before having executed the six memory instructions shown above. Answer X if a tag value is unknown, and for the *MESI* states, write in *all possible values* (i.e., `M`, `E`, `S`, and/or `I`).

Initial Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0	`0x1ff`	`M, E, S`
Set 1	`X`	`M, E, S, I`
Set 2	`0x110`	`M, E, S, I`
Set 3	`0x110`	`M, E, S, I`

Cache for P1		
	Tag	*MESI state*
Set 0	`X`	`M, E, S, I`
Set 1	`0x1ff`	`M, E, S, I`
Set 2	`X`	`M, E, S, I`
Set 3	`X`	`M, E, S, I`

Cache for P2		
	Tag	*MESI state*
Set 0	`0x10f`	`I`
Set 1	`X`	`M, E, S, I`
Set 2	`0x10f`	`M`
Set 3	`X`	`M, E, S, I`

Cache for P3		
	Tag	*MESI state*
Set 0	`0x133`	`E`
Set 1	`0x000`	`I`
Set 2	`0x000`	`I`
Set 3	`0x10f`	`I`

(b) [10 points] In what order did the memory operations enter the coherence bus?

time →					
0	4	5	1	2	3

6 Memory Consistency [45 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

	Thread T0		**Thread T1**
Instr. T0.0	`X[0] = 1;`	Instr. T1.0	`X[0] = 0;`
Instr. T0.1	`X[0] += 1;`	Instr. T1.1	`flag[0] = 1;`
Instr. T0.2	`while(flag[0] == 0);`	Instr. T1.2	`b = X[0];`
Instr. T0.3	`a = X[0];`		
Instr. T0.4	`X[0] = a * 2;`		

`X` and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

(a) [10 points] What could be possible final values of `a` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 1, or 2.

Explanation:
The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag[0]`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1. There are *at least* three possible sequentially-consistent orderings that lead to *at most* three different values of `a` at the end:
Ordering 1: T1.0 → T0.0 → T0.1 → T0.3 - Final value: `a` = 2.
Ordering 2: T0.0 → T1.0 → T0.1 → T0.3 - Final value: `a` = 1.
Ordering 3: T0.0 → T0.1 → T1.0 → T0.3 - Final value: `a` = 0.

(b) [10 points] What could be possible final values of `X[0]` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 2, or 4.

Explanation:
The value of `X[0]` is twice the value of `a`:
Ordering 1: T1.0 → T0.0 → T0.1 → T0.3 → T0.4 - Final value: `X[0]` = 4.
Ordering 2: T0.0 → T1.0 → T0.1 → T0.3 → T0.4 - Final value: `X[0]` = 2.
Ordering 3: T0.0 → T0.1 → T1.0 → T0.3 → T0.4 - Final value: `X[0]` = 0.

(c) [10 points] What could be possible final values of `b` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 1, 2, or 4.

Explanation:
Because there are no specific instructions to enforce the execution ordering of T1.2, `b` can have any of the values that `X[0]` can have during the execution of the two threads.

(d) [15 points] The programmer wants `a` and `b` to have the same value at the end of the execution of both threads. The final value of `a` and `b` should be the same value as in the original program (i.e., the possible final values of `a` that you found in part (a)). What *minimal* changes should the programmer make to the program?
(Hint: You can use more flags if necessary.)

She needs two more flags to enforce ordering.

Explanation:
Since the final value should be the same as in the original program, we have to maintain the `flag[0]` in T1.1 and T0.2. Then, `b` should not be updated until `X[0]` has the value that will be stored in `a`. Thus, either before or after T0.3, we need to set a new flag (`flag[1]`) that will be checked by Thread 1 before updating `b`. Finally, we cannot update `X[0]` until `b` has its final value. Thread 1 will set `flag[2]` only after `b` is updated. The modified code will be as follows:

	Thread T0
Instr. T0.0	`X[0] = 1;`
Instr. T0.1	`X[0] += 1;`
Instr. T0.2	`while(flag[0] == 0);`
Instr. T0.3	`a = X[0];`
Instr. T0.4	**flag[1] = 1;**
Instr. T0.5	**while(flag[2] == 0);**
Instr. T0.6	`X[0] = a * 2;`

	Thread T1
Instr. T1.0	`X[0] = 0;`
Instr. T1.1	`flag[0] = 1;`
Instr. T1.2	**while(flag[1] == 0);**
Instr. T1.3	`b = X[0];`
Instr. T1.4	**flag[2] = 1;**

7 Processing-in-Memory [65 points]

You have been hired to accelerate ETH's student database. After profiling the system for a while, you found out that one of the most executed queries is to *"select the hometown of the students that are from Switzerland and speak German"*. The attributes *hometown*, *country*, and *language* are encoded using a four-byte binary representation. The database has 32768 (2^{15}) entries, and each attribute is stored contiguously in memory. The database management system executes the following query:

```
bool position_hometown[entries];
for(int i = 0; i < entries; i++){
    if(students.country[i] == "Switzerland" && students.language[i] == "German"){
        position_hometown[i] = true;
    }
    else{
        position_hometown[i] = false;
    }
}
```

(a) [25 points] You are running the above code on a single-core processor. Assume that:

- Your processor has an 8 MB direct-mapped cache, with a cache line of 64 bytes.
- A hit in this cache takes one cycle and a miss takes 100 cycles for both load and store operations.
- All load/store operations are serialized, i.e., the latency of multiple memory requests cannot be overlapped.
- The starting addresses of *students.country*, *students.language*, and *position_hometown* are 0x05000000, 0x06000000, 0x07000000 respectively.
- The execution time of a non-memory instruction is zero (i.e., we ignore its execution time).

How many cycles are required to run the query? Show your work.

Cycles = cache_hits×1 + cache_misses×100 = 0×1 + (3×32×1024)×100

Explanation:
Since the cache size is 8 MB (2^{23}), direct-mapped, and the block size is 64 bytes (2^6), the address is divided as:

- block = address[5:0]
- index = address[22:6]
- tag = address[31:23]

The loop repeats for the total number of entries in the database (32×1024 times). In each iteration, the code loads addresses 0x05000000 and 0x06000000. It also stores the computation at address 0x07000000 (three memory accesses in total per cycle). All three addresses have the same index bits, but different tags. The cache hit rate is 0% since every memory access causes the eviction of the cache line that was just loaded into the cache.

(b) Recall that in class we discussed AMBIT, which is a DRAM design that can greatly accelerate Bulk Bitwise Operations by providing the ability to perform bitwise AND/OR/XOR of two rows in a subarray. AMBIT works by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) operations. For example, to compute AND, OR, and XOR operations, AMBIT issues the sequence of commands described in the table below (e.g., $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation).

In those instructions, AMBIT copies the source rows D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) AMBIT executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, copying a source row to DCC_i flips all bits in the source row and stores the result in both the source row and DCC_i. Assume that:

- The DRAM row size is **8 Kbytes**.
- An ACTIVATE command takes 50 cycles to execute.
- A PRECHARGE command takes 20 cycles to execute.
- DRAM has a single memory bank.
- The syntax of an AMBIT operation is: *bbop_* [and/or/xor] *destination, source_ 1, source_ 2.*
- Addresses 0x08000000 and 0x09000000 are used to store partial results.
- The rows at addresses 0x0A000000 and 0x0B00000 store the codes for *"Switzerland"* and *"German"*, respectively, in each four bytes throughout the entire row.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

i) [20 points] The following code aims to execute the query *"select the hometown of the students that are from Switzerland and speak German"* in terms of Boolean operations to make use of AMBIT. Fill in the blank boxes such that the algorithm produces the correct result. Show your work.

```
for(int i = 0; i < [          ] ; i++){

  bbop_[          ] 0x08000000, 0x05000000 + i*8192, 0x0A000000;

  bbop_[          ] 0x09000000, 0x06000000 + i*8192, 0x0B000000;

  bbop_[          ] 0x07000000, 0x08000000, 0x09000000;
}
```

1st box = Number of iterations = $\frac{database\_size}{row\_buffer\_size} = \frac{32*1024*4\ bytes}{8*1024\ bytes} = 16$
2nd box = bbop_xor
3rd box = bbop_xor
4th box = bbop_or

Explanation:
AMBIT can execute the query as follows:
T1 = country XOR "Switzerland"
T2 = language XOR "German"
hometown = T1 OR T2

T1 and T2 are auxiliary rows used to store partial results.

ii) [20 points] How much speedup does AMBIT provide over the baseline processor when executing the same query? Show your work.

Speedup = $\frac{3\times100\times32\times1024}{16\times2\times(25\times50+11\times20)+16\times(11\times50+5\times20)}$

Explanation:
To compute an XOR operation, AMBIT emits 25 ACTIVATE and 11 PRECHARGE commands. To compute an OR operation, it sends 11 ACTIVATE and 5 PRECHARGE commands.

8 BONUS: GPU Programming [50 points]

An inexperienced CUDA programmer is trying to optimize her first GPU kernel for performance. After writing the first version of the kernel, she wants to find the best *execution configuration* (i.e., grid size and block size[1]).

As she assigns one thread per input element, calculating the grid size (i.e., total number of blocks) is trivial. For N input elements, the grid size is $\lceil \frac{N}{block\_size} \rceil$, where $block\_size$ is the number of threads per block. So, the challenging part will be to figure out what is the block size that produces the best performance. She will try 5 different block sizes (64, 128, 256, 512, and 1024 threads).

She has learned that a general recommendation for kernel optimization is to maximize the *occupancy* of the GPU cores, i.e., Streaming Multiprocessors (SMs). Occupancy is defined as the ratio of active threads to the maximum possible number of active threads per SM.

In order to calculate the occupancy, it is necessary to take the available SM resources into account. She knows that in each SM of her GPU:

- The total *scratchpad memory* or *shared memory* is 16 KB.
- The total *number of 4-byte registers* is 16384.

In her first version of the kernel code, each thread needs 2 4-byte elements in shared memory for its private use. In addition, each block needs 10 4-byte elements in shared memory for communication across threads.

She has also learned that she can obtain the number of registers that each thread needs by using a special compiler flag. This way, she finds that each thread in the first version of the kernel uses 9 registers.

(a) [15 points] After reasoning some time about the amount of shared memory that her code needs, she decides to first test a block size of 128 threads. Why do you think she chose that number? Show your work.

She calculated the maximum number of threads that an SM can hold according to the shared memory usage. 128 threads per block results in that maximum.

Explanation:
Given the shared memory needs of her code, each block uses $2 \times block\_size + 10$ 4-byte elements, that is, $4 \times (2 \times block\_size + 10)$ bytes.
Since the shared memory available per SM is 16 KB, the number of blocks and the number of threads that each SM can allocate is as follows:

$block\_size$	Blocks/SM	Threads/SM
64	29	1856
128	15	**1920**
256	7	1792
512	3	1536
1024	1	1024

She decides to test a block size of 128 threads because this achieves the highest number of active threads per SM (i.e., the highest occupancy).

[1] We use NVIDIA terminology in this question.

However, after testing other block sizes, she finds out that using 256 threads per block provides higher performance than using 128. She does not understand why, so she looks for some information in the documentation of her GPU that can lead her to an explanation. There, she finds two more SM hardware constraints. In each SM:

- The *maximum number of blocks* is 8.
- The *maximum number of threads* is 2048.

Take into account these new constraints when answering parts (b), (c), and (d).

(b) [10 points] Can you explain why using 256 threads per block perform better than 128? Show your work.

The limitation in the maximum number of blocks per SM makes that the configuration with the highest occupancy is 256 threads per block.

Explanation:
This is the corrected table after taking the limitation in the maximum number of blocks into account:

block_size	Blocks/SM	Threads/SM
64	8	512
128	8	1024
256	7	**1792**
512	3	1536
1024	1	1024

(c) [15 points] What is the occupancy limitation due to register usage, if any? Explain and show your work.

There is *no* occupancy limitation due to the register usage.

Explanation:
The register usage depends on the block size. As she knows the number of registers per thread (9), she can calculate the total register needs:

block_size	Blocks/SM	Threads/SM	Registers/block	Registers/SM
64	8	512	576	4608
128	8	1024	1152	9216
256	7	1792	2304	16128
512	3	1536	4608	13824
1024	1	1024	9216	9216

In all cases, the total register usage is lower than 16384. (NOTE: The solution is correct if only calculated for 256 threads.)

(d) [10 points] The performance obtained by the first kernel version does not fulfill the acceleration needs. Thus, the programmer writes a second kernel version that reduces the number of instructions at the expense of using one more register per thread. What would be the highest occupancy for the second kernel? For what block size(s)?

The highest occupancy will be $\frac{1536}{2048} = 0.75$. It can be obtained with blocks of 256 or 512 threads.

Explanation:
The number of registers per thread is 10 in the second kernel. The configuration with 256 threads per block for the second kernel version will be able to allocate one less block per SM (6) than for the first kernel version (7):

block_size	Blocks/SM	Threads/SM	Registers/block	Registers/SM
64	8	512	640	5120
128	8	1024	1280	10240
256	6	**1536**	2560	15360
512	3	**1536**	5120	15360
1024	1	1024	10240	10240