CMU 18-447 Introduction to Computer Architecture, Spring 2014
Midterm Exam 2
Date: Wed., 4/23

Instructor: Onur Mutlu

TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bo Zhao, Paraj Tyle

Name:

Legibility & Name (5 Points):	
Problem 1 (55 Points):	
Problem 2 (40 Points):	
Problem 3 (40 Points):	
Problem 4 (40 Points):	
Problem 5 (55 Points):	
Problem 6 (45 Points):	
Bonus (40 Points):	
Total (280 + 40 Points):	

Instructions:

1. This is a closed book exam. You are allowed to have one letter-sized cheat sheet.

2. No electronic devices may be used.

3. This exam lasts 1 hour and 50 minutes.

4. Clearly indicate your final answer for each problem.

5. Please show your work when needed.

6. Please write your initials at the top of every page.

7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space required.

Tips:

- **Be cognizant of time.** Do not spend to much time on one question.

- **Be concise.** You will be penalized for verbosity.

- **Show work when needed.** You will receive partial credit at the instructors' discretion.

- **Write legibly.** Show your final answer.

1. Potpourri [55 points]

(a) **Caches, caches, caches [5 points]**

Assume you have two choices in designing the L1 cache of your company's next super-duper processor:

A. a 32KB 4-way associative sectored cache with 64-byte blocks and 8-byte subblocks

B. a 32KB 4-way non-sectored cache with 8-byte blocks

You have done extensive performance evaluations and you have found that performance of both of these options is the same for the workloads you care about.

Which cache design would you pick? Circle one.

A   B

Why? (in no more than 15 words)

Fewer bits needed for tag.

(b) **Synonyms and homonyms [5 points]**

Your sleepy friend is designing the shared last-level (L3) cache of a 16-core processor that implements an ISA with 4KB pages. The block size of the cache he is considering is 128 bytes. Your friend would like to make the cache very large (64MB), but he is concerned that the synonym problem, which we extensively covered in a lecture, would complicate the design of the cache. What solution would you suggest to your friend?

L3 cache should have no synonym/homonym problem (as it is physically addressed).

(c) **DRAM bitlines [5 points]**

Why are the bitlines long in commodity DRAM chips?

To amortize the high cost of sense amplifiers (to achieve higher density)

(d) **Prefetching algorithms and minimalist hardware** [**15 points**]

Suppose you have a memory access pattern that looks like this (all addresses are cache block addresses):

A, G, H, T, L, M, P, F, K, N, O, Y, X, Z, L, J, S,
A, G, H, T, L, M, P, F, K, N, O, Y, X, Z, L, J, S,
A, G, H, T, L, M, P, F, K, N, O, Y, X, Z, L, J, S,
A, G, H, T, L, M, P, F, K, N, O, Y, X, Z, L, J, S,
A, G, H, T, L, M, P, F, K, N, O, Y, X, Z, L, J, S,
...

The different letter addresses have nothing to do with each other (i.e., they come from random locations in the physical address space). The pattern continues for a long time.

Assume you have a fully-associative cache that can hold 16 blocks and implements LRU replacement.

i) What is the hit rate of this cache?

> 100%

ii) You would like to design a prefetcher to prefetch data into the cache. What kind of a prefetcher would you choose to use for this pattern?

Prefetcher type:

> None Alternatively, if you answer 0% in part i) Markov prefetcher

Justification:

> The correct answer for this part is there should be no problem with this access pattern Alternatively, if you answer 0% in part i) Markov Prefetcher will prefetch the requested block into the cache on the next time the same cache address is fetched.

iii) Now, assume you do not want a prefetcher and you would like to keep the cache the same. However, you are allowed to add very small hardware to minimize the cache miss rate for this program. What kind of a hardware structure would you add? (Hint: we discussed this in a lecture.)

Structure name:

> None
> Alternatively, if you answer 0% in part i), a victim cache

Justification:

> The correct answer for this part is there should be no problem with this access pattern Alternatively, if you answer 0% in part i) A victim cache will cache the block that get thrashed.

(e) **Multiprocessor type potpourri** [**16 points**]

i) In tightly coupled multiprocessors, memory address space is **shared** across all processors.

ii) "In tightly-coupled multiprocessors, memory can be physically distributed across processors."

Is the above statement correct? Circle one: YES   NO

> yes

iii) There are three major reasons why the parallel portion of a program is not perfectly parallel: 1) synchronization, 2) load imbalance, 3) resource contention. List those reasons that apply to the following two types of multiprocessors:

Tightly-coupled:

> synchronization, resource contention, load imbalance

Loosely-coupled:

> synchronization, resource contention, load imbalance

iv) "A cache coherence mechanism is required to keep caches consistent in the presence of multiple processors." Circle the type(s) of multiprocessors to which this statement applies:

Tightly-coupled multiprocessors

Loosely-coupled multiprocessors

> Both types

v) "Loosely-coupled multiprocessors do not require synchronization operations."

Is the above statement correct? Circle one: YES   NO

> no

vi) "Hardware contexts in a multithreaded processor have to be tightly coupled."

Is the above statement correct? Circle one: YES   NO

> no

(f) **Directory** [**11 points**]

Assume we have a processor that implements the directory based cache coherence protocol we discussed in class. The physical address space of the processor is 32GB (2^{35} bytes) and a cache block is 128 bytes. The directory is equally distributed across randomly selected 32 nodes in the system.

You find out that the directory size in each of the 32 nodes is a total of 200 MB.

How many total processors are there in this system? Show your work.

Total number of lines in each nodes $= 2^{35-7-5} = 2^{23}$
$200MB = 25 * 2^{23}$ bytes $= 25 * 2^{26} bits$
This means that each cache block is $25 * 8 = 200$ bits. This means that there are 199 processors (199 bits for each processor and one present bit).

2. Cache Coherence [40 points]

We have a system with 4 byte-addressable processors. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. Coherence is maintained using the MESI protocol we discussed in class. Accessible memory addresses range from 0x30000000 - 0x3FFFFFFF. We show below the initial tag store state of the four caches.

Initial State

Cache 0	Tag	MESI State
Set 0	0x3FFFFF	S
Set 1	0x000000	I
Set 2	0x330000	E
Set 3	0x300000	I

Cache 1	Tag	MESI State
Set 0	0x3FFFFF	S
Set 1	0x300000	E
Set 2	0x300004	E
Set 3	0x330000	E

Cache 2	Tag	MESI State
Set 0	0x3FFFFF	S
Set 1	0x330000	E
Set 2	0x32FFFF	M
Set 3	0x32FFFF	E

Cache 3	Tag	MESI State
Set 0	0x000000	I
Set 1	0x000000	I
Set 2	0x000000	I
Set 3	0x000000	I

After 5 memory instructions are executed in this system, we find the final tag store state of the four caches to be as follows:

Final State

Cache 0	Tag	MESI State
Set 0	0x3FFFFF	I
Set 1	0x300000	M
Set 2	0x330000	E
Set 3	0x333333	E

Cache 1	Tag	MESI State
Set 0	0x3FFFFF	I
Set 1	0x300000	I
Set 2	0x32FFFF	S
Set 3	0x330000	E

Cache 2	Tag	MESI State
Set 0	0x3FFFFF	I
Set 1	0x330000	E
Set 2	0x32FFFF	S
Set 3	0x32FFFF	S

Cache 3	Tag	MESI State
Set 0	0x3FFFFF	M
Set 1	0x000000	I
Set 2	0x000000	I
Set 3	0x32FFFF	S

Assume that the offset within a cache block is 0 for all requests. In the below diagram, you are given partially what happened with each of the 5 memory instructions. In particular, we show you whether the instruction caused an eviction or a write-back in any cache and which cache the instruction was serviced by. We ask you to complete three pieces of information: 1) address of each request (Address), 2) whether the request is a Read or a Write (R/W), 3) which caches the request caused an eviction from if an eviction occurred due to the request (Evicted from cache 0, 1, 2, 3).

Instruction			Evicted from cache						Request served by cache			
Order	**Address**	**R/W**	Eviction?	**0**	**1**	**2**	**3**	Write back?	0	1	2	3
1	0x3FFFFF00	W	Yes	X	X	X		No				X
2	0x32FFFF80	R	No					Yes		X		
3	0x32FFFFC0	R	No					No				X
4	0x30000040	W	Yes		X			No	X			
5	0x333333C0	R	No					No	X			

3. TL-DRAM [40 points]

In class, we have seen the idea of the tiered-latency DRAM, TL-DRAM. Recall that in TL-DRAM, the bitline of a subarray is segmented into two portions by adding isolation transistors inbetween, creating two segments on the bitline: the near segment and the far segment. The near segment is close to the sense amplifiers whereas the far segment is far away.

(a) Why is access to a row in the near segment faster in TL-DRAM compared to a commodity DRAM chip?

> The bitline is shorter (thus lower capacitance) than commodity DRAM

(b) Why is access to a row in the far segment slower in TL-DRAM compared to a commodity DRAM chip?

> There is an additional resistance from the isolation transistor

Now, assume that we have a system that uses the near segment as a cache to the far segment, and far segment contains main memory locations. Near segment is not visible to software and the rows that are cached in it are completely managed by the memory controller. The far segment is inclusive of the near segment. The far segment has 960 rows, whereas the baseline commodity DRAM has 1024 rows in a subarray.

(c) What is the capacity loss in main memory size when we use TL-DRAM as opposed to commodity DRAM? Express this as a fraction of total memory capacity lost (no need to simplify the fraction).

> $\frac{64}{1024}$

(d) What is the tag store size that needs to be maintained on a per subarray basis in the DRAM controller if the near segment is used as a fully-associative write-back cache? Assume the replacement algorithm is *not MRU* and use the **minimum number of bits** possible to implement the replacement algorithm. Show your work.

Answer:

> 774 bits

Show your work here:

> 1 valid bit, 1 dirty bit, 10 bits tags, 64 different entries (one for each row)
> Then add 6 bits to point where the MRU is
> Total size = 64*12 + 6 = 768 + 6 = 774 bits

Now assume near segment and far segment are exclusive. In other words both contain memory rows and a memory row can be only in one of the segments. When a memory row in the far segment is referenced, it is brought into the near segment by exchanging the non-MRU row in the near segment with the row in the far segment.

(e) What is the capacity loss in main memory size when we use TL-DRAM as opposed to commodity DRAM? Express this as a fraction of total memory capacity lost (no need to simplify the fraction).

> 0%

(f) What is the tag store size that needs to be maintained on a per subarray basis in the DRAM controller if the near segment is used as a fully-associative write-back cache? Assume the replacement algorithm is *not MRU* and use the **minimum number of bits** possible to implement the replacement algorithm. Show your work.

Answer:

> 10246 bits

Show your work here:

> Near segment: (10 bits tags) * 64row = 640 + 6 MRU bits = 646 bits
> A row swap can lead to any physical pages in the subarray getting mapped to any row in the subarray. This means we need the tag bits in each of the far segment rows.
> Far segment: 10 bits tags * 960 rows = 9600 bits
> Total size = 10246 bits

(g) Now, assume we decide to increase the number of rows in the near segment, while keeping the total number of rows in the subarray constant. Explain what happens to the following parameters (i.e., whether each increases, decreases or stays roughly the same). State your assumptions.

Near segment access latency:

> increases

Far segment access latency:

> stays roughly the same

Near segment access energy:

> increases

Far segment access energy:

> stay roughly the same

Hit rate in the near segment:

> increases

4. Multithreading [40 points]

Assume you have three multithreaded in-order processors with 16-bit physical address space. We will compare their performance in executing two threads. Each processor has a 256-byte, 2-way set-associative L1 cache. The cache line is 16 bytes. The cache uses the LRU replacement policy and takes 5 cycles to determine whether the cache is a hit or a miss. If the access is a hit, the data is returned at the same time as the hit/miss signal. If the access is a miss, it takes 50 more cycles to service the cache miss (i.e., the data is returned 55 cycles after the cache access starts).

(a) How many bits are used for tag, index and byte in block for this cache?

Tag:

> 9 bits

Index:

> 3 bits

Byte in block:

> 4 bits

The following table shows the operations performed by the two threads that are used to evaluate the performance of the three processors. Assume all operations are independent. Assume that between each memory access there are 20 independent add instructions. Assume that each add instruction takes one cycle and the processor always starts execution with Thread 1. The cost of a context switch from one thread to another is 10 cycles, unless otherwise stated.

Thread 1	Thread 2
Memory read from address 0x1020	Memory read from address 0x2020
20 add instructions	20 add instructions
Memory read from address 0x11a2	Memory read from address 0x2122
20 add instructions	20 add instructions
Memory read from address 0x1021	Memory read from address 0x2021
20 add instructions	20 add instructions
Memory read from address 0x11a3	Memory read from address 0x2123
20 add instructions	20 add instructions
Memory read from address 0x1022	Memory read from address 0x2220
20 add instructions	20 add instructions
Memory read from address 0x11a1	Memory read from address 0x2322
20 add instructions	20 add instructions
Memory read from address 0x1023	Memory read from address 0x2221
20 add instructions	20 add instructions
Memory read from address 0x11a2	Memory read from address 0x2323

(b) Assume Processor 1 first executes Thread 1 to completion and then executes Thread 2. What are the individual execution times and the total execution time (in cycles) of the two threads on this processor?

Thread 1:

> 280 cycles

Thread 2:

> 670 cycles
> Alternatively, if you assume that thread 2 starts after thread 1 finishes: 380 cycles

Total:

> 2 misses and 6 hits from thread 1 = 140 cycles 4 misses and 4 hits from thread 2 = 240 cycles
> total time for memory instructions = 380 cycles
> total time for compute instructions = 14*20 = 280 cycles
> total execution time is 670 cycles (660 + 10 context switch penalty)

(c) Assume Processor 2 employs fine-grained multithreading, with no context switch penalty between threads. The cache is single ported. What are the individual execution times and the total execution time (in cycles) of the two threads on this processor?

Thread 1:

> 1105 cycles (Total - 55 cycles cache miss from the last cache miss in thread 2)

Thread 2:

> 1160 cycles

Total:

> All cache accesses are misses in the cache. Total time = 16*55 = 880 cycles
> total time for compute instructions = 14*20 = 280 cycles
> total execution time is 1160 cycles

(d) Assume Processor 2 employs switch-on-event multithreading. The processor switches to the other thread when the currently-executing thread incurs a cache miss. The cache is dual-ported. What are the individual execution times and the total execution time (in cycles) of the two threads on this processor?

Thread 1:

> 580 cycles (Total - 35 cycles cache miss from the last cache miss in thread 2)

Thread 2:

> 615 cycles

Total:

> Total time = 525+90 = 615 cycles

Show your work:

> 5 cycles : first cache miss in thread 1
> 10 cycles: context switch to thread 2
> 5 cycles : first cache miss in thread 2
> 10 cycles: context switch back to thread 1
> 25 cycles: the first cache miss in thread 1 is done
> 20 cycles: finish executing 20 compute instructions in thead 1
> —— 75 cycles
> 5 cycles : reach the second cache miss in thread 1
> 10 cycles: context switch to thread 2
> 0 cycles : finish the first cache miss in thread 2
> 20 cycles: finish executing 20 compute instructions in thread 2
> 5 cycles : reach the second cache miss in thread 2
> 10 cycles: context switch to thread 1
> 5 cycles : the second cache miss in thread 1 is done
> 20 cycles: finish executing 20 compute instructions in thread 1
> —— 75 cycles
> Repeat this 75 cycles 5 times = 375 cycles
> 5 cycles : reach the eigth cache miss in thread 1
> 10 cycles: context switch to thread 2
> 0 cycles : finish the seventh cache miss in thread 2
> 20 cycles: finish executing 20 compute instructions in thread 2
> 5 cycles : reach the eigth cache miss in thread 2
> 10 cycles: context switch back to thread 1
> 5 cycles : finish the eigth cache miss in thread 1
> 10 cycles: context switch to thread 2
> 25 cycles: finish the eigth cache miss in thread 2
> —— 90 cycles
> Total time = 525+90 = 615 cycles

5. Tracing the Cache [55 points]

Assume you have three toy CPUs: 6808-D, 6808-T, and 6808-F. All three CPUs feature one level of cache. The cache size is 128 bytes, the cache block size is 32 bytes, and the cache uses LRU replacement. The only difference between the three CPUs is the associativity of the cache:

- 6808-D uses a direct mapped cache.

- 6808-T uses a two-way associative cache.

- 6808-F uses a fully associative cache.

You run the SPECMem3000 program to evaluate the CPUs. This benchmark program tests only memory read performance by issuing read requests to the cache. Assume that the cache is empty before you run the benchmark.

The cache accesses generated by the program are as follows, in order of access from left to right:

A, B, A, H, B, G, H, H, A, E, H, D, H, G, C, C, G, C, A, B, H, D, E, C, C, B, A, D, E, F

Each letter represents a unique cache block. **All 8 cache blocks are contiguous in memory.** However, the ordering of the letters does not necessarily correspond to the ordering of the cache blocks in memory.

For 6808-D, you observe the following cache misses in order of generation:

A, B, A, H, B, G, A, E, D, H, C, G, C, B, D, A, F

(a) By using the above trace, please identify which cache blocks are in the same set for the 6808-D processor. Please be clear.

A and B
C and G
H and D
E and F

(b) Please write down the sequence of cache misses for the 6808-F processor in their order of generation. (Hint: You might want to write down the cache state after each request).

> By simulating the cache and using the requests
>
> ```
> Req: A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
> M? : x x   x   x         x   x     x x         x x x x x x   x x x x x
> MRU: A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
>   : - A B A H B G G H A E H D H G G C G C A B H D E E C B A D E
>   : - - - - B A H B B G H A E E D H H H H G C A B H D D E C B A D
> LRU: - - - - - A A A B G G A A E D D D D H G C A B H H D E C B A
> ```

(c) For 6808-T, you observed the following five cache misses in order of generation:

A, B, H, G, E

But, unfortunately, your evaluation setup broke before you could observe **all** cache misses for the 6808-T.

Using the given information, which cache blocks are in the same set for the 6808-T processor?

> We first simulate the cache up to the point it breaks. By simulating the cache and using the requests
>
> ```
> Req : A B A H B G H H A E
> M?  : x x   x   x         x
> MRU0: A B A A B B B B A E
> LRU0: - A B B A A A A B A
> MRU1: - - - H H G H H H H
> LRU1: - - - - - H G G G G
> ```
>
> If H was in the same set as A and B, then the B right after H would have missed. Similarly if G was in the same set as A and B, the A right before E would have missed. Using this information and the sets calculated in part (a), the sets are respectively
>
> A, B, E, and F
> H, G, C, and D

(d) Please write down the sequence of cache misses for the 6808-T processor in their order of generation.

```
Req : A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
M?   : x x   x   x       x   x   x x           x x x x     x   x x
MRU0: A B A A B B B B A E E E E E E E E A B B B E E E B A A E F
LRU0: - A B B A A A A B A A A A A A A A E A A A B B B E B B A E
MRU1: - - - H H G H H H H H D H G C C G C C C H D D C C C C D D D
LRU1: - - - - - H G G G G G H D H G G C G G G C H H D D D D C C C
```

(e) What is the cache miss rate for each processor?

6808-D:

$\frac{17}{30}$

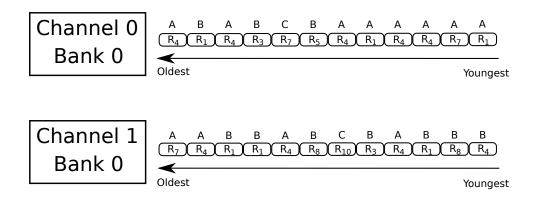6808-T:

$\frac{16}{30}$

6808-F:

$\frac{19}{30}$

6. Memory Request Scheduling [45 points]

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: Restores the contents of the bank's row-buffer back into the row. This is called closing a row. **(Latency: 15ns)**

- READ/WRITE: Accesses data from the row-buffer. **(Latency: 15ns)**

The following figure shows the snapshots of the memory request buffers (in the memory controller) at time t_0. Assume all requests are read requests. Each request is annotated with the address (or index) of the row that the request needs to access (e.g., R_3 means that the request is for the 3rd row). The letter above each request represents which application issued the request. There are three applications running at this time: A, B and C. Assume each application is running on a separate core.

A memory request is considered to be served when the READ command is complete (i.e., 15ns after the request's READ command had been issued). Each application (A, B, or C) is considered to be stalled until *all of its memory requests (across all the request buffers)* have been served. Assume that, initially (at t_0) Channel 0 Bank 0 has the 4th row and Channel 1 Bank 0 has the 7th row loaded in their respective row-buffers. Finally, assume no additional requests from any of the applications arrive at the memory controller.

(a) Using the **FR-FCFS** (or, row-hit-first) scheduling policy we discussed in class, what is the stall time of each application?

Application A:

255ns

Application B:

330ns

Application C:

285ns

Show your work:

Application A: max(8 Hits + 3 Misses, 3 Hits + 1 Misses) = 255ns Application B: max(8 Hits + 4 Misses, 7 Hits + 5 Misses) = 330ns Application C: max(7 Hits + 3 Misses, 7 Hits + 4 Misses) = 285ns

A TA looks at Application C, and sees that it is a non-memory intensive program. To speed up C, the TA suggests using a very simple batching technique. The idea is that in each bank, the first N requests received in time are prioritized over the younger requests. Then, the next N requests are prioritized, and so on. Thus, the new scheduling policy for each bank is as follows.

- Create a batch of the N oldest memory requests if there isn't already a batch with remaining requests.

- Within the active batch, follow the FR-FCFS scheduling policy.

(b) Describe how this might improve the performance of Application C.

Improves fairness as non-memory intensive applications do not have to wait for younger row hits from memory-intesive applications to be fulfilled.

(c) Describe one advantage and one disadvantage of using such a mechanism over FR-FCFS on an arbitrary memory request pattern.

Advantage:

Better fairness/better performance/reduced starvation for computation intensive applications

Disadvantage:

Fewer row hits/increased latency for memory intensive applications when run with multiple applications

(d) Assume the TA simulates the suggested batching algorithm, and observes the following stall times for each application.

- Application A stall time was 4 Row Hits + 8 Row Misses
- Application B stall time was 3 Row Hits + 9 Row Misses
- Application C stall time was 2 Row Hits + 5 Row Misses

What are possible values of N?

$N = 3$ and $N = 4$

(e) For the given access pattern, what values of N can be chosen such that the stall time of Application C is lower than its stall time when using FR-FCFS?

$1 <= N <= 11$

7. [Extra credit] Building Multicore Processors [40 points]

You are hired by Amdahl's Nano Devices (AND) to design their newest multicore processor.
Ggl, one of AND's largest customers, has found that the following program can predict people's happiness.

```
for (i = 12; i < 2985984; i++) {
  past = A[i-12]
  current = A[i]
  past *= 0.37
  current *= 0.63
  A[i] = past + current
}
```

A is a large array of 4-byte floating point numbers, gathered by Ggl over the years by harvesting people's private messages. Your job is to create a processor that runs this program as fast as possible.

Assume the following:

- You have magically fast DRAM that allows infinitely many cores to access data in parallel. We will relax this strong assumption in parts (d), (e), (f).

- Each floating point instruction (addition and multiplication) takes 10 cycles.

- Each memory read and write takes 10 cycles.

- No caches are used.

- Integer operations and branches are fast enough that they can be ignored.

(a) Assuming infinitely many cores, what is the maximum steady state speedup you can achieve for this program? Please show all your computations.

> The cycle counts are extra information that are not necessary.
> Instead, the loop body is sequential, while 12 consecutive iterations are parallel. Notice that the 13th iteration is dependent on the 1st iteration.
> Therefore $\frac{1}{12}$ of the program is serial. Using Amdahl's law, this solves to a maximum speedup of 12.

(b) What is the minimum number of cores you need to achieve this speedup?

> 12 cores are needed.

(c) Briefly describe how you would assign work to each core to achieve this speedup.

> Assign iteration `i` to processor `i%12`.

It turns out magic DRAM does not exist except in Macondo[1]. As a result, you have to use cheap, slow, low-bandwidth DRAM. To compensate for this, you decide to use a private L1 cache for each processor. The new specifications for the DRAM and the L1 cache are:

- DRAM is shared by all processors. DRAM may only process one request (read or write) at a time.
- DRAM takes 100 cycles to process any request.
- DRAM prioritizes accesses of smaller addresses and write requests. (Assume no virtual memory)
- The cache is direct-mapped. Each cache block is 16 bytes.
- It takes 10 cycles to access the cache. Therefore, a cache hit is processed in 10 cycles and a cache miss is processed in 110 cycles.

All other latencies remain the same as specified earlier. Answer parts (d), (e), (f) assuming this new system.

(d) Can you still achieve the same steady state speedup as before? Circle one: YES   NO

Please explain.

> No. Notice that the serial bottleneck is now caused by the DRAM. In steady state, only one memory access is performed for every four iterations. So four iterations take $100 + 200 = 300$ cycles. 12 iterations requires 900 cycles.
> In order to achieve 12x speedup, each core needs to complete each iteration in 75 cycles. However, fetching from memory alone requires 100 cycles. Therefore, it is not possible.
>
> By just saying that it is now slower than before is not enough, since the single core baseline is also slower.

---

[1] An imaginary town featured in *One Hundred Years of Solitude* by the late Colombian author Gabriel García Márquez (1927-2014).

(e) What is the minimum number of cores your processor needs to provide the maximum speedup?

> 3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
> Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

(f) Briefly describe how you would assign work to each core to achieve this speedup.

> 3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
> Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

Stratchpad

Stratchpad

Stratchpad