HW06: Caches
CSE 30321 Computer Architecture
Names: Aaron Wang and Ethan Little

Preamble:

1. ***Copy this assignment from our Google Drive directory, not from Gradescope. Downloading the PDF from Gradescope may not preserve all answer boxes.***

2. ***Reminder:*** point totals differ across assignments mainly as a way to allow for partial credit, *but all assignments carry the same weight* (see the syllabus for details).

3. Enter your answers in the boxes provided. Each empty box should be filled in with an answer. You can either type your answers directly or insert images as long as they are legible. For solutions that require code, use a **fixed-width font (Consolas preferred) no smaller than 10 points with proper indentation**. Save your solutions as a single PDF file and upload them to Gradescope.

4. You are encouraged to work in groups of 2. Please type your names at the top of this document.

Problem 1: Direct-Mapped Caches

Question 1

Consider a 64-bit, byte-addressable machine with 64 Byte blocks and a 16 MiB direct-mapped cache. Determine the number of bits required for byte offset, block offset, index, and tag *and* fill in the table noting the bit range for each field (see Question 2 as an example). If a bit "range" is a single bit, just write the same number twice.

64-bit machine means words are 8 bytes.
Number of bits for byte offset: lg(64/8)=3
Number of bits for block offset:lg(64/8)=3
Number of bits for index: lg(#blocks in cache)=lg(16*2^{20}/64)=lg(2^{4+20-6})=18
Number of bits for tag:64-18-3-3=40

Bit range:	63	24	23	6	5	3	2	0
Field:	Tag		Index		Block offset		Byte offset	

Question 2

Consider a 32-bit, byte-addressable machine with 16-word blocks and a 4 MiB direct-mapped cache. You are given the number of bits required for byte offset, block offset, index, and tag as well as the table noting the bit range for each field.

Number of bits for byte offset: 2
Number of bits for block offset: 4
Number of bits for index: 16
Number of bits for tag: 10

Bit ranges:	31	22	21	6	5	2	1	0
Field:	Tag		Index		Block offset		Byte offset	

Given the access pattern in the table below, **simulate the state of the cache (which is initially empty)** by filling the tables below with the data in each cache block, the tag of each cache block, and whether each access is a hit or miss, and *why*. You are ***required*** to show the "history" of the access pattern if/when cache blocks are evicted; indicate evictions by crossing out old tags. This will help us to assign partial credit if needed.

Here are some suggestions for how to complete this, and additional **requirements in bold:**

1. Refer to our examples from class where we showed how data moves in/out of caches. You are doing the exact same thing for this problem but with a different cache organization and access pattern.
2. You are given one table for the data array and a separate table for the tag array. This is similar to what was shown in class with the data stored in each block (at left) and the tag of each block (at right):

	1				0				word
Index	3	2	1	0	3	2	1	0	byte
0									
1									
2									
3									

	tag
Index	
0	
1	
2	
3	

3. Though the cache has a large number of blocks, the tables are set up to show only the cache blocks that are involved in the given accesses. Since there are 6 accesses, at *most* 6 cache blocks will be touched, so you *may* have empty rows in each table. The tables are also set up to show each word in each block, but not each byte.
4. Rather than make up example data for this problem, just **put an X in cache blocks where the data resides, and highlight each word that is accessed. Color code your accesses in the order they appear – any color is fine, but make a key for the graders so they know the order, for example "first access is yellow; second access is blue; third access is green; etc."**

   For the cache example shown above, if there is an access to word 1 of the block at index 2 (i.e., block offset = 1, index = 2) followed by an access to word 0 of the block at index 2, it would be filled like this:

	Word	
Index	1	0
0x0		
0x1		
0x2	X	X
0x3		

5. **Show all your work** as you follow the steps we took in class to fill each of the 3 tables for each access. **Treat each access sequentially, from top to bottom – you can imagine these as a sequence of addresses from load instructions.**

Table showing data in each cache block with indices listed **in hex:**

	Word															
Index	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0x8EEE	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
0xCEEE	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X X X

Table showing tag (in hex) in each cache block with indices listed **in hex:**

Index	Tag
0x8EEE	0x2A8
0xCEEE	~~0x2A8~~; ~~0x0C9~~; 0x2A8;

Table indicating hit or miss for each access, in order. For “why”, on a hit just note whether it was due to temporal or spatial locality; for miss, note either “empty” or “tag mismatch.”

Access (hexadecimal address)	Hit or Miss?	Why?
0xAA23BB80	Miss	Empty
0xAA33BB80	Miss	Empty
0x3273BB80	Miss	Tag mismatch

0x3273BBB0	Hit	Spatial locality
0xAA33BB80	Miss	Tag mismatch
0xAA23BB90	Hit	Spatial locality

Table showing data in each cache block with indices listed **in hex:**

	Word															
Index	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0x8EEE	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
0xCEEE	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X	X X X

Table showing tag (in hex) in each cache block with indices listed **in hex:**

Index	Tag
0x8EEE	0x2A8
0xCEEE	~~0x2A8~~ ~~0x0C9~~ 0x2A8

Table indicating hit or miss for each access, in order. For “why”, on a hit just note whether it was due to temporal or spatial locality; for miss, note either “empty” or “tag mismatch.”

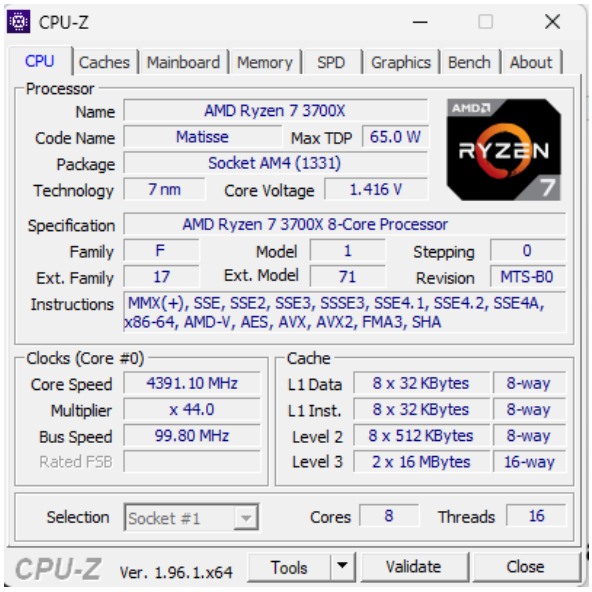
Access (hexadecimal address)	Hit or Miss?	Why?
0xAA23BB80	Miss	Empty
0xAA33BB80	Miss	Empty
0x3273BB80	Miss	Tag mismatch
0x3273BBB0	Hit	Spatial locality
0xAA33BB80	Miss	Tag

		mismatch
0xAA23BB90	Hit	Temporal locality

Problem 2: Set Associative Caches

Question 1

Consider the processor in my desktop machine at home:



For the **L3 Cache only**, determine the number of bits required for byte offset, block offset, index, and tag *and* fill in the table noting the bit range for each field (see Question 2 as an example). There are two L3 caches – each shared among 8 cores – but consider just one independently (i.e., it is 16MiB). Assume MB = MiB in the CPU-Z output. Most of the information can be found in the screenshot, above, but you may need to do some additional research (e.g., to determine cache block size, byte-addressability, etc.). You don't need to cite your sources for this – we've discussed it all in class, or you can do a Web search, etc. If a bit "range" is a single bit, just write the same number twice.

Block size: 64 bytes; Word Size: 8 bytes
Number of bits for byte offset: lg(64/8)=3
Number of bits for block offset:lg(64/8)=3
Number of bits for index: lg(#blocks in cache/associativity)=lg(16*2^{20}/(64*16))=lg($2^{4+20-6-4}$)=14
Number of bits for tag:64-14-3-3=44

Bit range:	63	20	19	6	5	3	2	0
Field:	Tag		Index		Block offset		Byte offset	

Question 2

Consider a 64-bit, byte-addressable machine with 8-word blocks and a 16 MiB 2-way set associative cache. You are given the number of bits required for byte offset, block offset, index, and tag as well as the table noting the bit range for each field.

Number of bits for byte offset: 3
Number of bits for block offset: 3
Number of bits for index: 17
Number of bits for tag: 41

Bit ranges:	63	23	22	6	5	3	2	0
Field:	Tag		Index		Block offset		Byte offset	

Given the access pattern in the table below, **simulate the state of the cache (which is initially empty)** by filling the tables below with the data in each cache block, the tag of each cache block, and whether each access is a hit or miss. You are ***required*** to show the "history" of the access pattern if/when cache blocks are evicted; indicate evictions by crossing out old tags. This will help us to assign partial credit if needed.

Here are some suggestions for how to complete this, and additional **requirements in bold:**

1. Refer to our examples from class where we showed how data moves in/out of caches. You are doing the exact same thing for this problem but with a different cache organization and access pattern.

2. You are given one table for the data array and a separate table for the tag array. This is similar to what was shown in class with the data stored in each set/way (at left) and the tag of each set/way (at right):

Cache example setup: 4 blocks; 2 words per block; 4 bytes per word
2-way set associative

		1				0				word
Set (Index)	Way	3	2	1	0	3	2	1	0	byte
0	0									
0	1									
1	0									
1	1									

		tag
Set (Index)	Way	
0	0	
0	1	
1	0	
1	1	

3. Though the cache has a large number of sets, the tables are set up to show only the cache sets that are involved in the given accesses. Since there are 6 accesses, at *most*

6 cache sets will be touched, so you *may* have empty rows in each table. The tables are also set up to show each word in each way (block), but not each byte.

4. Rather than make up example data for this problem, just **put an X in cache blocks where the data resides, and highlight each word that is accessed. Color code your accesses in the order they appear – any color is fine, but make a key for the graders so they know the order, for example "first access is yellow; second access is blue; third access is green; etc."**

   For the cache example shown above, if there is a single access that maps to set 0, way 0, word 1 (i.e., block offset = 1, index = 0), followed by an access to set 0, way 0, word 0 it would be filled like this:

		Word	
Set (Index)	Way	1	0
0x0	0x0	X	X
0x0	0x1		
0x1	0x0		
0x1	0x1		

5. **Show all your work** as you follow the steps we took in class to fill each of the 3 tables for each access. **Treat each access sequentially, from top to bottom – you can imagine these as a sequence of addresses from load instructions.**
   a. If there is a conflict, evict the **least recently used** tag from the set.

Table showing data in each cache set and way with indices listed **in hex:**

		Word							
Set (Index)	Way	7	6	5	4	3	2	1	0
0x08EEE	0x0	X	X	X	X	X	X	X	X
	0x1								
0x0CEEE	0x0	X	X	X	X	X	X	X	X X
	0x1								
0x1CEEE	0x0	X	X	X	X	X	X	X	X
	0x1								
	0x0								
	0x1								
	0x0								

	0x1								
	0x0								
	0x1								

Table showing tag (in hex) in each cache set and way with indices listed **in hex:**

Set (Index)	Way	Tag
0x08EEE	0x0	0x00000154
	0x1	
0x0CEEE	0x0	0x00000154
	0x1	
0x1CEEE	0x0	0x00000064
	0x1	
	0x0	
	0x1	
	0x0	
	0x1	
	0x0	
	0x1	

Table indicating hit or miss for each access, in order. For “why”, on a hit just note whether it was due to temporal or spatial locality; for miss, note either “all ways empty” or “no tag match found.”
(Note: assume the upper 32 bits of each address are all 0)

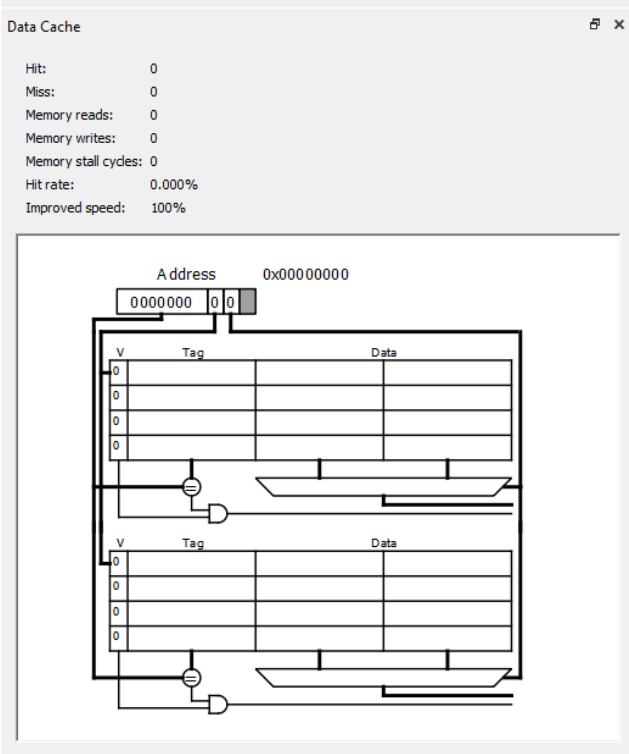
Access (hexadecimal address)	Hit or Miss?	Why?
0xAA23BB80	Miss	All Ways Empty
0xAA33BB80	Miss	All Ways Empty

`0x3273BB80`	Miss	all ways empty
`0x3273BBB0`	Hit	Spatial locality
`0xAA33BB80`	Hit	Temporal locality
`0xAA23BB90`	Hit	spatial locality

Problem 3: Cache Performance

Download `hw06_arr.s`. This assembly program is set up to create an array of 1024 (identical) elements in memory, and then read those back from memory to compute the sum of all elements. This is a contrived example used as a proxy for a program that strides over a large array, without extra setup code to slow the simulator down. Read the assembly code to ensure you understand what it is doing.

Open `hw06_arr.s` in QtRVSim. Create a simulation with the default parameters for "Pipelined with hazard unit and cache." Click the *Windows* menu and *Data Cache* to bring up the data cache view which tracks hits, misses, and hit rate (among other metrics):



Note: the default configuration also creates a program (instruction) cache. We will ignore that for this problem, ***focus only on the Data Cache.***

Tip: read through the following two questions in full before you run the program in the simulator so you know what data to collect, when you might want to pause the simulator, etc. so you can avoid re-running the program and save yourself some time.

Question 1

Run `hw06_arr.s`. It may take a couple minutes to run through the ~20000 cycles; if you are using a laptop, plugging into external power should prevent your CPU clock rate being throttled by battery saver mode.

Record the total hits, misses, and hit rate for the data cache. Explain why this hit rate was achieved based on the access pattern and the cache design parameters. *Be very specific*, referencing which array element accesses result in hits and which result in misses, and why. *Every single array element access* should be covered in your explanation, either directly, or describing (with justification) a repeated pattern.

Hints: 1) click the *File* menu and *New Simulation* and look at the default Data Cache parameters; 2) uncomment the `ebreak` right before `sum_loop` and think about why there are 2 memory reads reported, and 1 cache hit (it might help to look at what data is in the cache at this point); 3) before running at “max speed”, step through for a few loop iterations and observe how hits, misses, and total memory reads changes, as well as what is stored in the cache.

Hits: 513

Misses: 1536
First miss comes from: lw x6, 0(x6).
First hit comes from: sw x8, 0(x9) as it was just loaded in
The next 1023 misses come from: sw x8, 0(x9) as those elements are not in the cache.

Now to the sum_loop with: lw x10, 0(x9)
The first time is a miss
The second time is a hit as it was just loaded due to spatial locality.
This cycle continues for the 1024 instructions accounting for 512 hits and 512 misses.

Hit rate: 25.037%

Explanation for hit rate:513/(513+1536)=25.037%.

Question 2

Calculate the average memory access time based on your answers from Question 1. The default memory access time in QtRVSim is 10 cycles. Assume the data cache access time is 1 clock cycle, and assume the clock rate is 1GHz.

Note: QtRVSim reports a large number of memory stall cycles, you should not use this in your calculations.

AMAT calculation:

((1+10)(1-25.037%)+1(25.037%))/1GHz=8.49 nS

What to Turn In

Fill in the boxes on the previous pages with answers to all the questions. Save your Google Doc as a PDF and upload it to **Gradescope** for grading. Note Gradescope can be accessed via our Canvas site, or by visiting gradescope.com. Below is a checklist for this assignment:

Problem 1 (30 points)

	Deliverable	Points
1.	Question 1 - direct-mapped cache field calculations	10
2.	Question 2 - direct-mapped cache simulation	20

Problem 2 (30 points)

	Deliverable	Points
1.	Question 1 - set associative cache field calculations	10
2.	Question 2 - set associative cache simulation	20

Problem 3 (25 points)

	Deliverable	Points
1.	Question 1 - report and *explain* statistics	15
2.	Question 2 - AMAT calculation	10