Department of Computer Science
National Tsing Hua University
CS4100 Computer Architecture
Spring 2023 Homework 6

Deadline: 2023/06/09 23:59

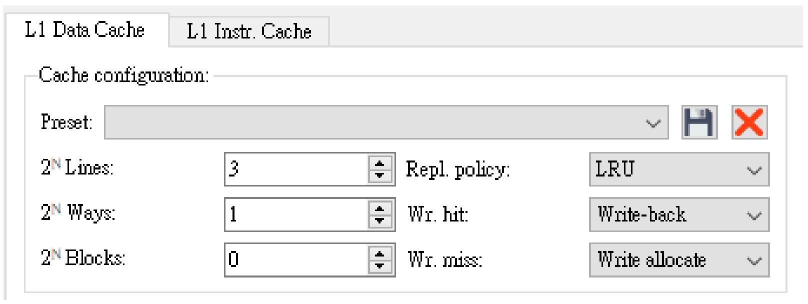
Q1 is a simulation assignment in which we will use the cache simulator in Ripes, a lightweight RISC-V pipeline simulator, to observe the behavior of the cache.

First of all, please follow the following steps to set up the cache simulator in Ripes.

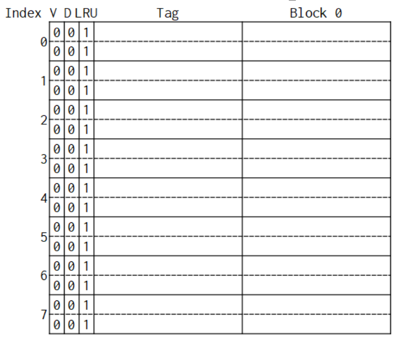
(1) Please load the program *QuickSort.elf* into Ripes. Also, don't forget to set the processor and the global pointer (x3/gp) to be *64-bit 5-stage processor* and *base+0x800*, respectively. This step is the same as what we did in HW5. You may check the setup tutorial released with HW5 if you are not sure whether you're doing it right.
Note 1: Be careful that the directory path of your program (.elf file) should contain only alphanumeric characters.

(2) Click the "Cache button " in the left panel to open the cache simulator.

(3) Set the cache configuration of L1 data cache to be 2^3 *lines*, 2^1 *ways*, and 2^0 *blocks*, as the picture shows.



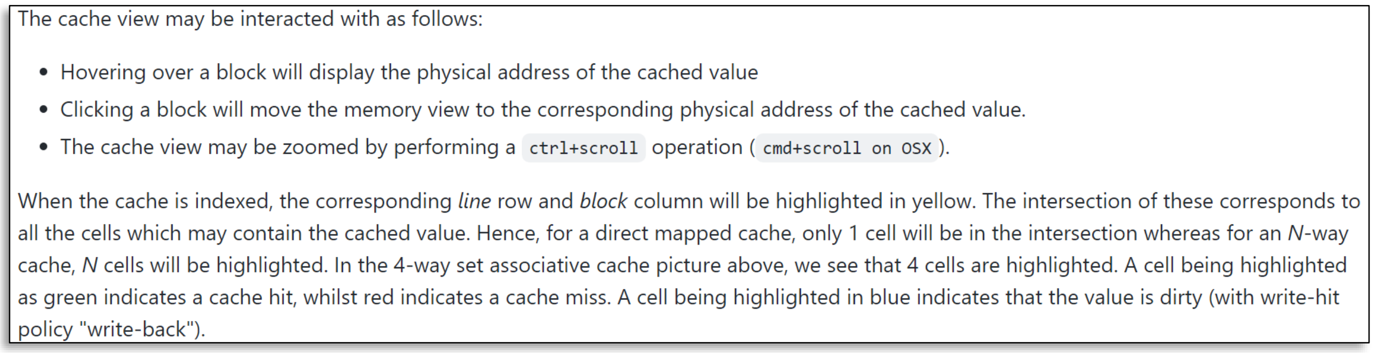
2^N Lines:	3	Repl. policy:	LRU
2^N Ways:	1	Wr. hit:	Write-back
2^N Blocks:	0	Wr. miss:	Write allocate



Index	V	D	LRU	Tag	Block 0
0	0	0	1		
	0	0	1		
1	0	0	1		
	0	0	1		
2	0	0	1		
	0	0	1		
3	0	0	1		
	0	0	1		
4	0	0	1		
	0	0	1		
5	0	0	1		
	0	0	1		
6	0	0	1		
	0	0	1		
7	0	0	1		
	0	0	1		

This setting implies that our data cache would be a 2-way set associative cache with 8 sets (index 0~7) and 1 double word (i.e., 8 bytes) for each block. Thus, we have 16 blocks with the size of 128 bytes in total.
Also, keep in mind that we adopt LRU, write-back, and write allocate for replacement policy and write handling. For LRU in Ripes, the smaller the value, the more recently used the corresponding block.

(4) Finally, you are ready to run some simulations to see how the cache works in real-time. To stop and check the current CPU/cache state at certain instructions, you can switch to "Editor" panel and add some breakpoints by clicking the left blue bar.
Besides, here are some useful tips from Ripes' docs for you to interact with the cache simulator and observe how it works.



The cache view may be interacted with as follows:

- Hovering over a block will display the physical address of the cached value
- Clicking a block will move the memory view to the corresponding physical address of the cached value.
- The cache view may be zoomed by performing a `ctrl+scroll` operation (`cmd+scroll on OSX`).

When the cache is indexed, the corresponding *line* row and *block* column will be highlighted in yellow. The intersection of these corresponds to all the cells which may contain the cached value. Hence, for a direct mapped cache, only 1 cell will be in the intersection whereas for an *N*-way cache, *N* cells will be highlighted. In the 4-way set associative cache picture above, we see that 4 cells are highlighted. A cell being highlighted as green indicates a cache hit, whilst red indicates a cache miss. A cell being highlighted in blue indicates that the value is dirty (with write-hit policy "write-back").

Note 2: Don't get confused if you find any mismatches between the cache simulator and the memory view in Ripes. A mismatch happens because the processor models in Ripes actually don't access the cache simulator when accessing memory. That is, Ripes doesn't follow the cache simulator to maintain its memory view. You may read the docs if you want to know more about the details. However, focusing on the cache simulator is good enough to observe the cache's behavior for this homework.

Note 3: Under the limitation of its lightweight and simplicity, the cache simulator of Ripes is not quite powerful. For example, after a write hit/miss, the displayed value of the corresponding block won't be changed to the written value immediately. (It will be updated the next time the block is read.) But the block will be marked as dirty correctly and you can see the block being highlighted in blue immediately. Therefore, don't be picky about what the value is in the "block" column of the cache table.

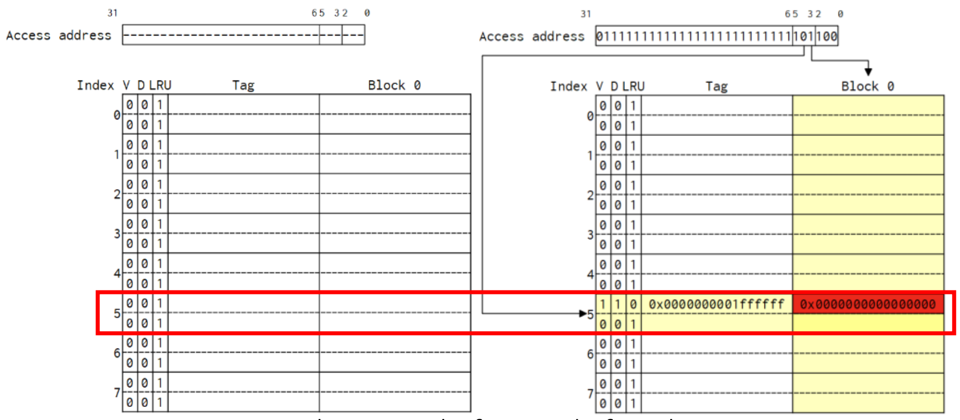
Note 4: Similar as Note 3, the cache simulator doesn't work perfectly with the "Undo button <". For example, you may find a case of write hit that just made the corresponding block dirty. But when you click the undo button, you might find that its dirty bit (D) didn't return to be 0 correctly. Every time this kind of things happens, you just need to memorize the instruction at which you found the case and add a breakpoint to rerun the simulation and stop before that instruction, then you would be able to get a correct screenshot.

1. (15 points) Please run the simulation to find the following cases. For (a)~(c) case, please show the screenshots of (i) the load/store instruction at which you find the case, (ii) accessed memory address, and (iii) cache states before and after that load/store instruction. Also, please give a simple explanation of what happened for each case. You may follow the example question we made to answer the following cases.
Ex: One case of a write miss at an index (a set) with all ways vacant.
Sol:

Screenshots

Explanation

Instruction:
```
10174:        00112023        sw x1 0 x2
```



Index	V	D	LRU	Tag	Block 0
0	0	0	1		
	0	0	1		
1	0	0	1		
	0	0	1		
2	0	0	1		
	0	0	1		
3	0	0	1		
	0	0	1		
4	0	0	1		
	0	0	1		
5	0	0	1		
	0	0	1		
6	0	0	1		
	0	0	1		
7	0	0	1		
	0	0	1		

Index	V	D	LRU	Tag	Block 0
0	0	0	1		
	0	0	1		
1	0	0	1		
	0	0	1		
2	0	0	1		
	0	0	1		
3	0	0	1		
	0	0	1		
4	0	0	1		
	0	0	1		
5	1	1	0	0x0000000001ffffff	0x0000000000000000
	0	0	1		
6	0	0	1		
	0	0	1		
7	0	0	1		
	0	0	1		

Cache states before and after the write miss

With write allocate, we fetch the block on a write miss. After the write miss at index 5, the valid bit (V) and dirty bit (D) will be 1, and the LRU bit will become 0 to represent that the block is the most recently used.

(a) (3 points) One case of cache insertion at an index (a set) with exactly one way already occupied. Don't forget to explain the change of LRU bits of both ways in that set.
(b) (3 points) One case of a hit that makes a block become dirty. (Not dirty before that hit)
(c) (3 points) One case of a replacement with write-back needed.
(d) (6 points) Improve the hit rate by designing your own cache. You may adjust the associativity or the cache size. Please give some screenshots to show (i) the cache you design and (ii) the improvement of the hit rate. A brief discussion about why the cache you design makes the hit rate higher is also needed.

This is the end of the simulation assignment. Please continue with the following written assignments.

2. (12 points) Consider a 10-bit data, $D = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}\}$, to be protected by using a Hamming single error correcting code with an additional parity bit, p_{11}, so that double errors can be detected.
   (a) (2 points) Let the encoded codeword $C = \{p_1, p_2, d_1, p_4, d_2, d_3, d_4, p_8, d_5, d_6, d_7, d_8, d_9, d_{10}, p_{11}\}$. Show the encoding process of D =1001011001 to obtain the encoded codeword C.
   (b) (2 points) Assume no error occurs. Show the decoding process of C (i.e., locating/correcting the

single error, detecting double errors, or confirming that there is no error; you have to verify if the result is right or wrong) from (a).

(c) (2 points) From (a), suppose d_5 of C is inverted. Show the decoding process.

(d) (2 points) From (a), suppose p_1 and d_8 of C are inverted. Show the decoding process.

(e) (2 points) From (a), suppose p_1, d_8, and p_{11} of C are inverted after the encoding. Show the decoding process.

(f) (2 points) What is the minimum number of parity bits required to protect 128-bit data using the same method to construct a SECDED code?

3. (12 points) For a direct-mapped cache design with a 14-bit memory address, the following bit fields of the address are used to access the cache.

Field	Tag	Index	Block Offset	Byte Offset
Bits	13:9	8:4	3:2	1:0

(a) (2 points) What is the cache block size (in bytes)?

(b) (2 points) How many blocks does the cache have?

(c) (2 points) What is the total cache size (in bytes)?

(d) (2 points) What is the actual SRAM memory size used to implement this cache in bytes? Please also consider the valid bit, dirty bit, and tag bits for each block.

(e) (2 points) With the same cache size in (c), derive the bit fields of the address for a 2-way set associative cache design.

(f) (2 points) From (e), what is the actual SRAM memory size used to implement this cache in bytes? Please also consider the valid bit, dirty bit, and tag bits for each block; an extra reference bit for each set.

4. (8 points) Consider the cache architecture of a specific processor that its base CPI without memory stalls is 1. Suppose that the L-1 miss rate is 5%. The main memory access delay is 200 cycles. There are several options of L-2 and L-3 caches in the following table.

Level	Name (Placement)	Access Cycle	Local Miss Rate
L-2	L2-DM (Direct Mapped)	L2 access cycle = 16	L2 Miss Rate = 4.0%
L-2	L2-4WAY(4-way Set Associative)	L2 access cycle = 20	L2 Miss Rate = 3.5%
L-3	L3-8WAY (8-way Set Associative)	L3 access cycle = 50	L3 Miss Rate = 2.0%

Please note the following definitions:

- Local Miss Rate: the fraction of references to one level of a cache that miss. That is, the number of misses in a cache divided by the total number of memory accesses to that cache.
- Global Miss Rate: the fraction of references that miss in all levels of a multilevel cache. That is, the number of misses in the cache divided by the total number of memory accesses generated by the CPU.

(a) (5 points) Calculate the effective CPI for each possible multilevel (i.e., 2-level or 3-level) cache design. Which one provides the best performance based on the effective CPI?

(b) (3 points) From (a), consider the multilevel cache design with the best performance. Suppose you want to further reduce the effective CPI to 1.8 or lower by improving the L-2 cache with fewer access cycles (assuming the local miss rate remains the same). What are the maximum access cycles of this improved L-2 cache to meet this requirement?

5. (23 points) Consider that a computer P uses a 128-byte 2-way set associative cache as a primary cache. The cache adopts the write-back and write-allocate policies. Each block in the cache has 16 bytes. The physical byte address of the computer has 12 bits. The block replacement policy is LRU (Least-Recently Used).

Suppose the cache receives the following memory accesses. Show the content of the cache after each memory access. Indicate if it is a read/write miss/hit for each access. Also, indicate if there occurs a block replacement, write allocate, or write back.

No.	Read or Write	Address
1	Read	`0x340`
2	Read	`0x000`
3	Read	`0x1d8`
4	Write	`0x354`
5	Read	`0xa61`
6	Write	`0xa61`
7	Read	`0x3ec`
8	Read	`0xa62`
9	Read	`0x3ea`
10	Read	`0x422`

Suppose initially the cache has one entry as shown in the following table. For each block (way), the **`V`** denotes the valid bit (V=1 indicates a valid block; V=0 or a blank field indicates an invalid block), and the **`D`** is the dirty bit (D=1 indicates a dirty block). For each set, the **`R`** denotes the reference bit (R=0 indicates that Way 0 is the least recently used between the two ways in the same set; R=1 indicates that Way 1 is the least recently used).

Additionally, **`Mem[0x000-00f]`** represents a data block of 16 bytes from **`Mem[0x000]`** to **`Mem[0x00f]`**. Note that the empty tag for the existing entry in Set 0 is also for you to fill in.

		Way 0				Way 1			
	R	V	D	Tag	Data	V	D	Tag	Data
Set 0	1	1	0	000000_2	`Mem[0x000-00f]`				
Set 1									
Set 2									
Set 3									

6. (10 points) A byte-addressable virtual memory system has the following characteristics:
   - Each virtual address is 16 bits.
   - Each physical address is 14 bits.
   - The page size is 512 bytes.
   - Each process has a one-level page table, and each page table entry (PTE) is 4 bytes.
   - The TLB is directed-mapped and has 16 entries.

   (a) (2 points) How many bits are required for the page offset?
   (b) (2 points) What is the maximum number of virtual pages a process could have?
   (c) (2 points) How many physical pages are there?
   (d) (2 points) What is the maximum page table size for a process?
   (e) (2 points) Assuming each entry of the TLB has a valid bit, a dirty bit, a reference bit, a tag, and a physical page number, what is the total number of bits in the TLB?

7. (20 points) The following addresses constitute a stream of virtual byte addresses generated by a processor.

`0x5368, 0x02c3, 0x434b, 0x6812, 0xaf50`

Assume that the main memory has adequate space to accommodate the requested physical page brought from the disk without page replacement. Therefore, you can choose any page number as long as it is not used in the page table. Suppose that a page has 4KB. A four-entry fully associative TLB with the approximate LRU replacement policy (see the appendix) is utilized.

The initial TLB and page table states are listed below. Suppose the ***replacement pointer*** points to the first block.

Valid	Tag	Physical Page Number	Reference (Used)	
1	0x4	6	1	← Replacement Pointer
1	0x1	2	0	
1	0xa	3	1	
0	0x3	5	0	

Index	Valid	Physical Page Number or in Disk
0	0	Disk
1	1	2
2	0	Disk
3	1	5
4	1	6
5	1	11
6	1	7
7	0	Disk
8	0	Disk
9	0	Disk
a	1	3
b	0	Disk

For each access shown above, list whether the access is a hit or miss in the TLB, whether the access causes a page fault, and the updated state of the TLB.

Appendix: An approximate LRU can be implemented as follows:

- A ***used bit*** (initial 0) is associated with every block.
- When a block is accessed (hit or miss), its used bit is set to 1.
- On a replacement, a ***replacement pointer*** circularly scans through the blocks to find a block with

its used bit = 0 to replace.

  - The replacement pointer points to the first block initially, and scans circularly through the blocks in a set during the operation.

- Along the way, the replacement pointer also resets the encountered used bits from 1 to 0.

  - Alternatively, if on an access, all other used bits in a set are 1, they are reset to 0 except the bit of the block that is accessed.