

**Department of Computer Science**  
**National Tsing Hua University**  
**CS4100 Computer Architecture**  
**Spring 2022 Homework 6**  
 Deadline: 2022/06/12 23:59


Q1 and Q2 are simulation questions in which we will use the cache simulator in Ripes, a light-weight RISC-V pipeline simulator, to observe the behavior of cache.

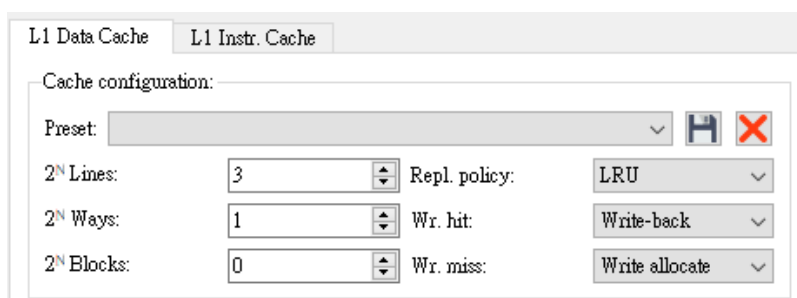
First of all, please follow the following steps to set up the cache simulator in Ripes.

- (1) Please load the program you wrote in HW5 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 right.

Note 1: The correctness of your program doesn't matter for HW6. So, don't worry if you messed up the previous homework. Or, if you encountered difficulties doing Q1 and Q2 with your own program, you may directly use the provided "quick\_sort.elf" which helps you away from any potential troubles : )

Note 2: 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.



| Index | V | D | LRU | Tag | Block 0 |
|-------|---|---|-----|-----|---------|
| 0     | 0 | 0 | 1   |     |         |
| 0     | 0 | 0 | 1   |     |         |
| 1     | 0 | 0 | 1   |     |         |
| 1     | 0 | 0 | 1   |     |         |
| 2     | 0 | 0 | 1   |     |         |
| 2     | 0 | 0 | 1   |     |         |
| 3     | 0 | 0 | 1   |     |         |
| 3     | 0 | 0 | 1   |     |         |
| 4     | 0 | 0 | 1   |     |         |
| 4     | 0 | 0 | 1   |     |         |
| 5     | 0 | 0 | 1   |     |         |
| 5     | 0 | 0 | 1   |     |         |
| 6     | 0 | 0 | 1   |     |         |
| 6     | 0 | 0 | 1   |     |         |
| 7     | 0 | 0 | 1   |     |         |
| 7     | 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 3: Don't get confused if you find any mismatches between the cache simulator and the memory view in Ripes. 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

Note 4: 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.

Now, let's get started with Q1 and Q2!

- Ex: One case of a write miss at an index (a set) with all ways vacant.

### Screenshots (5 points)

Instruction: 00174: 00112023 sw x1 0 x2

Access address: 00174 (bits 31, 65, 32, 0)

Access address: 00112023 (bits 31, 65, 32, 0)

Index V D LRU Tag Block 0

Index V D LRU Tag Block 0

Cache state (Index 0-7):

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

Cache state (Index 0-7):

| Index | V | D | LRU | Tag              | Block 0            |
|-------|---|---|-----|------------------|--------------------|
| 0     | 0 | 0 | 1   |                  |                    |
| 0     | 0 | 0 | 1   |                  |                    |
| 1     | 0 | 0 | 1   |                  |                    |
| 1     | 0 | 0 | 1   |                  |                    |
| 2     | 0 | 0 | 1   |                  |                    |
| 2     | 0 | 0 | 1   |                  |                    |
| 3     | 0 | 0 | 1   |                  |                    |
| 3     | 0 | 0 | 1   |                  |                    |
| 4     | 0 | 0 | 1   |                  |                    |
| 4     | 0 | 0 | 1   |                  |                    |
| 5     | 1 | 1 | 0   | 0x0000000001ffff | 0x0000000000000000 |
| 5     | 0 | 0 | 1   |                  |                    |
| 6     | 0 | 0 | 1   |                  |                    |
| 6     | 0 | 0 | 1   |                  |                    |
| 7     | 0 | 0 | 1   |                  |                    |
| 7     | 0 | 0 | 1   |                  |                    |

Cache states before and after the write miss

- Note that if you can't find such cases, you could add additional load/store instructions to your program or increase the number of variables and array size to create the case you need. You can also use the given ELF example which should have all the cases.

- This is the end of simulation questions. Please continue with Q3 ~ Q6.

3. (20 points) Suppose a cache receives the following memory addresses for accessing 32-bit single words:  
(Note: Memory addresses are to the bytes.)

0x74, 0x3c, 0xa4, 0x70, 0x34, 0x08, 0xc4, 0xb8, 0x94, 0xb8

For each reference, identify (i) the index, (ii) the tag, and (iii) whether it is a hit or a miss, for each of the following caches. Assume each cache is initially empty.

- (a) (10 points) A direct-mapped cache with 1-word blocks and a total size of 8 words.  
(b) (10 points) A 2-way set associative cache with 2-word blocks and a total size of 16 words.
4. (10 points) Consider the Hamming single error correcting (SEC) code.
- (a) (4 points) Let  $A = 10011100$  be an 8-bit data. Show how to get a 12-bit Hamming SEC code word  $B$  for  $A$ .  
(b) (4 points) Suppose  $C$  is obtained by inverting the fifth bit from the left of  $B$  and hence has a single bit error. Show how to find and correct the single bit error in  $C$ .  
(c) (2 points) What is the minimum number of parity bits required to protect a 256-bit data using the Hamming SEC code?
5. (10 points) The following table gives some information for each of the instruction and data caches attached to each of the two processors, P1 and P2.

|    | I-cache miss rate | D-cache miss rate | Clock period |
|----|-------------------|-------------------|--------------|
| P1 | 5%                | 8%                | 0.5 ns       |
| P2 | 4%                | 6%                | 1 ns         |

Assume each processor has a base CPI of 1 without any memory stalls. Each access to the instruction or data memory takes 50 ns. And 40% of all instructions have data accesses. Note that a base CPI of 1 means that an instruction completes in one clock cycle, unless either the instruction access or the data access causes a cache miss.

- (a) (8 points) What is the total CPI for P1 and P2?  
(b) (2 points) Which processor is faster if they have the same amount of instructions?
6. (20 points) The following data constitute a stream of virtual byte addresses generated by a processor.

0x4a90, 0x210c, 0xa3b5, 0x0e18, 0x904f

With 4 KB pages, a four-entry fully associative TLB, and approximate LRU replacement, 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.

The initial TLB and page table states are as below.

TLB

| Valid | Tag | Physical Page Number | Reference (Used) |
|-------|-----|----------------------|------------------|
| 1     | 0x5 | 11                   | 1                |
| 1     | 0x0 | 5                    | 0                |
| 1     | 0xa | 3                    | 0                |
| 0     | 0x3 | 6                    | 1                |

Page table

| Index | Valid | Physical Page Number or in Disk |
|-------|-------|---------------------------------|
| 0     | 1     | 5                               |
| 1     | 0     | Disk                            |
| 2     | 0     | Disk                            |
| 3     | 1     | 6                               |
| 4     | 1     | 9                               |
| 5     | 1     | 11                              |
| 6     | 0     | Disk                            |
| 7     | 1     | 4                               |
| 8     | 0     | Disk                            |
| 9     | 0     | Disk                            |
| a     | 1     | 3                               |
| b     | 1     | 12                              |

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 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 reset 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.