# 2024 Computer Organization HW 3: Reducing Memory Access Overhead for the μRISC-V Processor

### Computer Organization 2024 Programming Assignment III

### Due Date: June 19, 2024 at 23:59

## Overview

Processor caches enhance the processor performance by accommodating recent or frequent-used data items in the fast memories, allowing these data items to be accessed faster, compared with accessing to the main memory. To reduce the memory access overhead, two approaches are commonly used in a hardware-software co-design process: 1) an application-aware cache design, and 2) a cache-architecture aware memory accessing in software.

In this assignment, you are required to perform both the hardware and software parts for the hardware-software co-design.

- When a processor cache is full, the cache replacement policy must choose which items to discard to make room for the new ones. **You have to implement a FIFO cache replacement policy**, and this policy should be implemented in the open-source RISC-V emulator, Spike. By implementing the policy with the cache simulation interfaces exposed by Spike, the RISC-V processor emulator can further estimate the performance delivered by various cache models. Note that the original version of Spike has implemented the functionality of cache simulations. That is, users can enable the cache simulations (e.g., L1 data and/or L1 instruction caches) by specifying which types of the cache simulations to be performed via parameter settings. Unfortunately, the current version of Spike only implements the *random* replacement policy and some other policies should be implemented in order to better evaluate different design alternatives.
- After implementing the replacement policy, **you are required to revise a given software program to reduce memory access overhead**. To achieve this, you need to understand the organization of the underlying cache (the L1 data cache of the emulated μRISC-V Processor) and use common algorithmic strategies for cache miss reduction (e.g., loop interchange, fusion, blocking, etc.).

The following two sections introduce the methods to improve the performance of data caching from a hardware perspective (cache replacement policy) and a software perspective (software methods to reduce cache misses), respectively. In this assignment, you will learn how to improve the data caching performance for the target μRISC-V Processor with the cache simulation offered by Spike.

# 1. Introduction to Cache Replacement Policies

As a cache replacement policy can greatly affect the cache hit/miss rate, it has been extensively studied and proposed. Each policy is suitable for certain purposes. The cache hit/miss rate is an important metric to evaluate the efficiency of a cache design. The three of the most commonly seen cache replacement policies are listed below.

- First In First Out (FIFO): The cache behaves the same as a FIFO queue, where it evicts the cache blocks in the same order as they were added to the cache without considering any other factors.
- Least Recently Used (LRU): Those the least recently used cache blocks are discarded first.
- Least-Frequently Used (LFU): Those cache blocks that are used least often are replaced first.

For more cache replacement policies (or the above three policies), please refer to this web page for more details.

# 2. Introduction to Software Methods for Cache Miss Reduction

The cache hit/miss rate can be influenced by multiple factors. It is not only affected by the replacement policy (implemented in the hardware caches) but also by the way of the implemented in a software program. Common strategies/algorithms for reducing cache miss rates, which you can apply in a program, are listed below. The key idea is to improve the spatial or temporal locality.

- Loop Interchage: Change the order you access data to improve the spacial locality on cache access.
- Loop Fusion: Combine the operation on same array at different loop to increase temporal locality.
- Loop Tiling: Breaking down a large loop into smaller blocks or tiles and organizing computations within each tile. Loop tiling enhances the reuse of data stored in the cache.

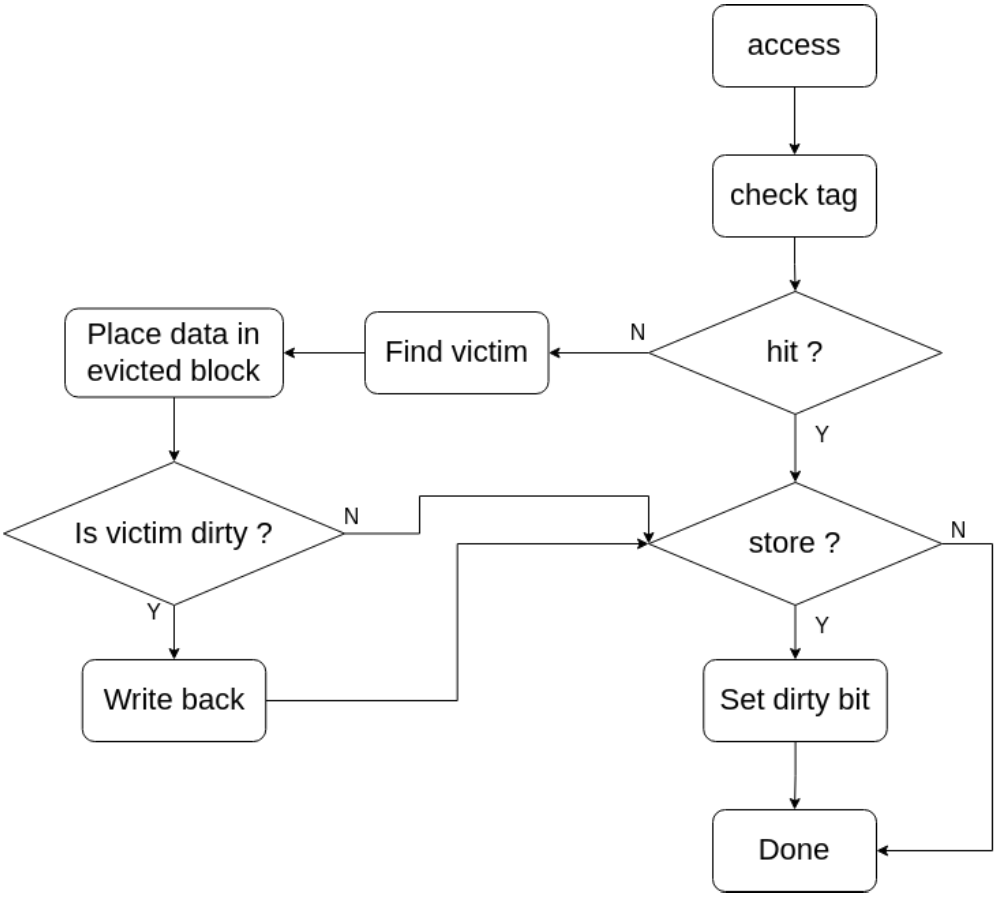
# 3. Cache Simulations in Spike

During program emulation by Spike, the emulated instructions (the addresses of the instructions) are fed into the L1 *instruction cache* simulator and the data required by the emulated instructions (e.g., accessed via the memory instructions) are fed to the L1 *data cache* simulator. The cache simulator(s) are represented as the `cache_sim_t` class objects in Spike. The creation of the cache simulators is determined by command line parameters of Spike; for example, the `dc` flag is used to activate a L1 data cache simulator, and its parameters are used to specify the attributes of the created data cache, such as *set* and *ways*. Please refer to the following section (4. What Should You Do in this Assignment?) for more details.

The `cache_sim_t` class defines the behaviors of the cache simulators created by Spike, where the prototypes are defined in `riscv/cachesim.h` and the implementation is in `riscv/cachesim.cc`. Both data and instruction caches inherit the interfaces defined in

the `cache_sim_t` class. There are some important member functions related to manipulate the created cache(s), such as `access`, `check_tag`, and `victimize`. In the following paragraphs, we briefly introduce the high-level concepts of `access`, and `check_tag`. **You should trace the source code to figure out more details of the member functions so as to understand the mechanism and workflow of the caches simulated by the `cache_sim_t` objects.**

The workflow of the data cache simulation in Spike is highlighted as follows. The data cache simulation starts when a memory read/write is emulated by Spike, and the `access` function is invoked. The prototype of the `access` member function is `cache_sim_t::access(uint64_t addr, size_t bytes, bool store)`, where the parameter `addr` contains the address to the to-be-accessed data, `bytes` represent the size of the to-be-accessed data, and `store` refers to its a read or store operation. The high level workflow is illustrated in the following image.




As for the `cache_sim_t::check_tag(uint64_t addr)` function, it is used to check the existence of a to-be-accessed data in the simulated data cache (i.e., `check tag` rectangle in the workflow image). The index to the *set* for the to-be-accessed data is calculated by the `addr` argument via the formula: `idx = (addr >> idx_shift) & (sets-1)`. If you have configured a four-set data cache, the `idx` value is the remainder of the dividing by 4 operation; that is, the value range of `idx` is from 0 to 3. Next, all of the *tag* in the corresponding *ways* are compared with the input argument `addr` to find the existence of the data with the `addr` in the data cache. If a match is found, it means a cache it; otherwise, it is a cache miss.

When a cache miss occurs, the `victimize` function is invoked to find a victim data cache block to store the new input data. Currently, Spike implements only the *random* algorithm to randomly select the victim data block.
When a cache miss occurs, a victim block should be selected to store the input

data. In particular, Spike leverages the LFSR to implement a pseudo-random cache replacement policy.

**Note that** computer simulations do not always replicate real hardware specifications and behaviors for a physical computer system entirely. Instead, they focus on some designated problems faced by a component of the computer system. They use different workloads and simulation configurations to evaluate the tradeoffs among design alternatives. For example, in this assignment, we focus on the L1 data cache designs and try to find out a good cache design that can lead to the lowest data cache miss rate of the given workload. Furthermore, the overhead incurred by the cache misses data is not taken into account, and the implementation overhead (e.g., hardware area, power consumption, computation latency) of different replacement policies is not calculated, too.

# 4. What You Should Do in this Assignment

In this assignment, you have to do the following three things.

1. You are responsible for implementing a FIFO replacement policy within the Spike's cache simulation module.
2. You are required to reduce the memory access overhead of the provided two programs.
3. You are going to make an appointment with TA for a demo session. In your registered demo session, you are required to explain the idea and implementation of your developed code for the FIFO replacement policy (in Spike) and the revised software programs (emulated by Spike).

The above three items are further divided into two parts listed below.

- First part (Programming part):
  - Implement the FIFO cache replacement policy into Spike. (Please refer to Section 4.1.1).
  - Reduce memory access overhead of the given programs. (Please refer to Sections 4.1.2 and 4.1.3).
- Second part (Demo part):
  - Explain how the Spike cache replacement policy works.
  - Explain the design philosophy of your revised programs, e.g., the techniques/concepts you used in the revised program.

Please submit the above code before the deadline specified at the beginning of this document. **Late submissions will deduct 30% of the total score.**

## 4.1 First part (40 %)

### Overview

In this part, you are required to **implement the FIFO data cache replacement policy** (refer to Section 4.1.1) and **revise the software programs to reduce the memory access overhead** (refer to Sections 4.1.2 and 4.1.3). It is important to note that the performance of your revised programs, e.g., L1 data cache misses, should be evaluated on top of your implemented FIFO cache replacement policy. To this end, **after** implementing your cache replacement policy in `riscv/cachesim.h` and `riscv/cachesim.cc`, **you must *rebuild* your Spike simulator**.Please refer to *HW0* to

check how to build your revised Spike simulator.

The revised software programs are evaluated based on the *improvements* made in cache performance.
The improvements will be calculated by the following formula:

$$\mathrm{MemCycle}_{Ori} \,/\, \mathrm{MemCycle}_{Imp}$$

$\mathrm{MemCycle}_{ori}$ indicates the memory cycles spent on the *original version* of a given program, whereas $\mathrm{MemCycle}_{Imp}$ indicates the memory cycles spent on your *improved version*. Furthermore, the memory cycles can be derived by the equation:

$$\mathrm{MemCycle} = \mathrm{cache}_{hit} \times \mathrm{latency}_{hit} + \mathrm{cache}_{miss} \times \mathrm{penalty}_{miss}$$

This equation means that the total cycles spent on the memory accesses are the cycles for memory accesses *hit* and *missed* in the L1 data cache. The *hit* (or *missed*) memory accesses can be further calculated by multiplying the number of *hit* (or *missed*) by the *hit latency* (or *miss penalty*). The hit latency indicates the processor cycles required to access the data in the data cache, whereas the miss penalty represents the processor cycles required to access the data from the next-level memory (i.e., main memory in this assignment) since these data do not exist in the L1 data cache.

In this assignment, we assume the *hit latency* is one cycle, and the *miss penalty* is thirty cycles. An example performance data for the L1 data cache simulation done by Spike is shown below. As the example shows, the memory cycles of *original version* is 19,636,074 cycles, which is derived by:

$$19,636,074 = ((5,245,118 - 433,720) + (871,214 - 32,478)) \times 1 + (433,720 + 32,478) \times 30$$

The same formula apply to the *improved version*, the memory cycles are $9,805,410$. Based on the above information, the improved ratio is

$$\frac{19,636,074}{9,805,410} \approx 2.003$$

```
Original version                    | Improved version
D$ Bytes Read:            24874040  | D$ Bytes Read:            30530544
D$ Bytes Written:         4006529   | D$ Bytes Written:         4504069
D$ Read Accesses:         5245118   | D$ Read Accesses:         6595544
D$ Write Accesses:        871214    | D$ Write Accesses:        995455
D$ Read Misses:           433720    | D$ Read Misses:           48792
D$ Write Misses:          32478     | D$ Write Misses:          27567
D$ Writebacks:            47520     | D$ Writebacks:            40871
D$ Miss Rate:             7.622%    | D$ Miss Rate:             1.006%
                                    |
Memory access overhead =  19636074  | Memory access overhead = 9805410
----------------------------------- | -----------------------------------

Improved ratio:  2.0025755
```

In this assignment, you should use the configuration for the L1 data cache: **4 ways, 8 sets, and 32-byte cacheline**. That is, the configuration setting for the cache simulation should be set with: `--dc=8:4:32` when running the Spike simulator (with `RV64GC` ISA). An example command to run the Spike simulation is listed below.

```
$ spike --isa=RV64GC --dc=8:4:32 $RISCV/riscv64-unknown-elf/bin/pk <your program>
```

### 4.1.1 Implement FIFO data cache replacement policy

To implement your FIFO replacement policy, you should use the Spike source code from the repository `riscv-isa-sim` that has been installed in *HW0*. Particularly, you need to revise the following two source files that are used for cache simulations.

- Please add your implementations of the cache replacement policy in the following files.
  - **riscv/cachesim.h**: You may like to add required data structures for the cache simulation in this file, so as to keep essential information to facilitate cache simulation.
  - **riscv/cachesim.cc**: You are required to add your code into this file to implement the mechanism of the replacement policies. For example, you need to inject your code to the member functions of the `cache_sim_t` class to do what is necessary for the cache simulation in different stages (of the above workflow image). You might like to refer to cache replacement policies first to understand the mechanisms of the to-be-implemented replacement policy.

In order to implement FIFO data cache replacement policy into Spike simulator, you should refer to Section 3 for the cache simulation in Spike. It is important to note that you can only modified **riscv/cachesim.cc** and **riscv/cachesim.h** files in source code.

> Hint: To implement FIFO cache replacement policy, you can use the third-party implementation of data structures inside `riscv/cachesim.cc`. For example, since Spike are wrote in C++, you can leverage the `queue` data structure from STL to facilitate the cache replacement policy implementation.
>
> Note that if you **cannot** provide a valid implementation for the FIFO data cache replacement policy, you can upload the *random* policy offered by the Spike simulator (i.e., the original version released in Spike source code). In this case, you can still work on the enhancement of software programs (Sections 4.1.2 and 4.1.3). Of course, you will lose 10 points from the first part for not being able to provide a FIFO data cache replacement policy.

### 4.1.2 Enhancement of software program, *2D convolution*, to reduce memory access overheads

2D convolution is a mathematical operation, which is commonly used in signal processing and image analysis applications. Examples of image analysis applications include blurring, sharpening, edge detection, and feature extraction. Moreover, in the applications of deep learning, 2D convolution plays a pivotal role in convolutional neural networks (CNNs).

2D convolution combines two matrices, typically an input matrix, such as an image, and a smaller matrix known as a kernel or filter. This operation involves sliding the kernel over the input matrix, computing element-wise multiplication at each position, and summing the results to generate a single value in the output matrix.

This operation is mathematically represented as $I'(i,j) = \sum_n \sum_m I(i+m, j+n)$

$\cdot C(m,n)$, where *I* is the matrix for an input image, *C* represents the kernel matrix for convolution, and *i, j, m,* and *n* are indexes to certain positions in the matrices.

As 2D convolution is widely adopted in different applications, it is crucial to have an efficient 2D convolution implementation to improve the application performance. In this exercise, you need to do the following things.

1. Your goal is to reduce the memory cycles of the 2D convolution program.
2. You will be provided with a simplified 2D convolution implementation, which is put under the path `Q1/conv2d.S` of your downloaded file for this assignment.
3. You need to implement your improved algorithm in `Q1/conv2d-better.S`.
4. Please write your code in pure assembly code.
5. Note that, you don't need to consider integer overflow.
6. The L1 data cache is specified as 4 way, 8 set and 32-byte cachelines.

The high-level concept (in C code) of the 2D convolution in `conv2d.S` is given below. A tip to accomplish this exercise is that you can try to improve the memory cycles of the following code first (by reducing the data cache miss rate). Then, you can convert your revised C code to its assembly version `Q1/conv2d-better.S`.

```
void conv2d(int *input, int *kernel, int w, int h, int ksize, int s){
    for (int j = 0; j <= w - ksize; j++){
        for (int i = 0; i <= h - ksize; i++){
            int sum = 0;
            for (int n = 0; n < ksize; n+=s){
                for (int m = 0; m < ksize; m+=s){
                    sum += input[(i + m) * w + j + n] * kernel[m * ksize + n];
                }
            }
            input[i*w + j] = sum;
        }
    }
}
```

### 4.1.3 Enhancement of software program, *matrix multiplication*, to reduce memory access overheads

Matrix multiplication is another fundamental mathematical operation that combines two matrices to produce a third matrix. It can be mathematically expressed as $C_{ij} = \sum_k A_{ik}B_{kj}$, where *A*, *B*, and *C* are all matrices. Matrix multiplication is widely used in image processing, game engines, quantum simulation, and many other fields. In fact, matrix multiplication is so important in real-world applications that matrix related operations are developed and included in some library implementations for high performance computations. For example, BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. There are BLAS implementations in a variety of programming languages, such as C, Fortran, and CUDA.

In this exercise, you need to do the following things.

1. Your goal is to reduce the memory cycles of the matrix multiplication program.
2. You will be provided with a baseline version of the matrix multiplication program in `Q2/mm.c` within your downloaded file for this assignment.

3. You need to implement an improved version in the `mm-better.c` file within the `Q2` folder.
4. Please write your code in C.

5. Note that, you don't need to consider the integer overflow

6. The L1 data cache is specified as 4 way, 8 set and 32-byte cachelines.

```
void matrix_multiplication(int *a, int *b, int *output, int i, int k, int j){
    for(int x = 0; x < i; x++){
        for (int y = 0; y < j; y++){
            int sum = 0;
            for (int z = 0; z < k; z++){
                sum += a[x * k + z] * b[z * j + y];
            }
            output[x * j + y] = sum;
        }
    }
    return;
}
```

### 4.2 Second part (80 %)

Your job is to register a demo session, which will be announced later, and elaborate your design philosophy of your implementation. Your score is based on the answers you provide to the TA.

You have to provide your following answers during the demo session.

1. Describe the workflow and mechanism in Spike, related to cache simulation. (10%)
2. Describe the concept behind your modified conv2d algorithm. (25%)
3. Describe the concept behind your modified matrix multiplication algorithm. (25%)
4. Bonus: Explain your implementation of question 1 and 2. Does your implementation fit in any size of matrix multiplication? If yes, please explain why. If not, please provide an opposite example. (20%)

## 5. Test Your Developed Code

We have implemented the relevant code for you guys to check your developed code. You can run `make check` under homework folder to check the correctness and the *improved ratio* of your enhanced programs.

Note that before testing your code, **please follow the guide in *HW0* to rebuild your Spike simulator** if you have implemented your FIFO cache replacement policy and you would like to see how it works.

```
$ make check
```

If you have correctly implemented your algorithm, the output will look similar as below. Q1 and Q2 refer to the exercises in Sections 4.1.2 and 4.1.3, respectively. Your code will be tested by the TA with different inputs, so **please submit your code after passing the check.**

```
============== Q1 ==============
Original version
```

```
D$ Bytes Read:            9388618
D$ Bytes Written:         2756484
D$ Read Accesses:         2005180
D$ Write Accesses:        478403
D$ Read Misses:           95201
D$ Write Misses:          41069
D$ Writebacks:            108920
D$ Miss Rate:             5.487%

Memory access overhead =  6435413 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:            9388618
D$ Bytes Written:         2756484
D$ Read Accesses:         2005180
D$ Write Accesses:        478403
D$ Read Misses:           34629
D$ Write Misses:          41069
D$ Writebacks:            52533
D$ Miss Rate:             3.048%

Memory access overhead =  4678825 (cpu cycle)
-----------------------------------
Improved ratio:  1.3754335757374982
Output Correctness:  Pass
-----------------------------------
=============== Q2 ===============
Original version
D$ Bytes Read:            24875390
D$ Bytes Written:         4006372
D$ Read Accesses:         5245258
D$ Write Accesses:        871183
D$ Read Misses:           433660
D$ Write Misses:          32457
D$ Writebacks:            47479
D$ Miss Rate:             7.621%

Memory access overhead =  19633834 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:            30531894
D$ Bytes Written:         4503912
D$ Read Accesses:         6595684
D$ Write Accesses:        995424
D$ Read Misses:           48732
D$ Write Misses:          27546
D$ Writebacks:            40830
D$ Miss Rate:             1.005%

Memory access overhead =  9803170 (cpu cycle)
-----------------------------------
Improved ratio:  2.002804603000866
Output Correctness:  Pass
-----------------------------------
```

If you receive a *Fail* on `Output Correctness` row after running `make check`, you can use `make diffq1` or `make diffq2` depending on which exercise you failed, to compare your output with the expected answer. This command will compare `*/ans.output` and `*/stu.output`, which indicate the output of the answer and the student (yours), respectively.

# 6. Submission of Your Assignment

The scoring criteria of each part is listed below:

- First part
  - Implement FIFO replacement policy (10%)
  - Reduce miss rate of given 2D convolution algorithm (15%)
    - improve ratio: > 1.3 (15 points)
    - improve ratio: > 1 (10 points; 1 < improve ratio < 1.3)
  - Reduce miss rate of given matrix multiplication algorithm (15%)
    - improve ratio: > 1.8 (15 points)
    - improve ratio: > 1 (10 points; 1 < improve ratio < 1.8)
- Second part
  - Describe the workflow and mechanism in Spike, related to cache simulation. (10%)
  - Describe the concept behind your modified conv2d algorithm. (25%)
  - Describe the concept behind your modified matrix multiplication algorithm. (25%)
  - Bonus: Explain your implementation of question 1 and 2. Does your implementation fit in any size of matrix multiplication? If yes, please explain why. If not, please provide an opposite example. (20%)

To submit your assignment, please follow the instructions below. Replace `StudentID` with your actual student ID. Do not name your folder and file with the literal string `StudentID`!!!

1. Compress your source code into a zip file.
2. Submit your homework to NCKU Moodle.
3. The zipped file and its internal directory organization of your developed code should be identical to the example below.
   - **NOTE**: Replace all following "*StudentID*" with your student ID number.

```
StudentID_HW3.zip/
└── StudentID_HW3/
    ├── cachesim.h
    ├── cachesim.cc
    ├── Q1
    │   └── conv2d-better.S
    └── Q2
        └── mm-better.c
```

**!!! Incorrect format (either the file structure or file name) will lose 10 points. !!!**

**!!! Late submission will lose 30% of total score. (Deadline: June 19, 2024 at 23:59) !!!**

**!!! Late submission is allowed for ONLY one week after deadline. (June 26, 2024 at 23:59) !!!**

# 7. Reference

- Cache replacement policy
- Loop Interchage
- Loop Fusion
- Loop Tiling
- Linear-Feedback Shift Register (LFSR)