

Department of Computer Science
National Tsing Hua University
CS4100 Computer Architecture

Spring 2025, Homework 5

Due date: 5/31/2025 23:59

Academic Integrity Policy for Submissions

1. All submitted work—including both source code and written reports—must be completed independently by the student. Any form of plagiarism, as well as the use of automated tools, including AI assistants, code generators, or compilers that produce source code or report content on the student's behalf, is strictly prohibited.
2. This course enforces a strict zero-tolerance policy toward plagiarism and academic dishonesty. To uphold the integrity of submitted work, we will employ a plagiarism detection tool. Any submission with a high similarity score (e.g., above 25%) will be subject to detailed review. Verified cases of misconduct will be referred for disciplinary action in accordance with university regulations.

All submitted reports must include the following statement at the beginning.

(請詳閱上述規定，並於繳交之報告開頭附上方框內聲明內容並親自簽名).

Student Confirmation of Academic Integrity Policy

I have read and understood the Academic Integrity Policy. I acknowledge that all submitted work is my own and acknowledge the consequences of violating this policy.

Student Name: _____

Student ID: _____

Signature: _____

Date: _____

Part A: Coding and Simulation

In this assignment, we will use the **Ripes** RISC-V pipeline simulator to explore how pipelining works. Please ensure that Ripes is properly set up before you begin. Upon completing the assignment, submit your assembly code and report according to the specified format on EECLASS.

- **Environment Setup**

Before you begin, install and test Ripes simulator. Setup guides for different operating systems (Linux, macOS, Windows 10/11) can be found in the **setup folder**.

1. Assembly Coding

Use `cmul.S` as a reference to:

- (1) Write your own RISC-V assembly code.
- (2) Create a **flowchart** of your code and include **evidence of correct results** in your report.

The specific function you must implement is determined by the **last digit of your student ID**. Use the appropriate template in the individual folder. Reference links or sample CPP files are also included for guidance.

- Task A: `Binary_Search_template.S`: Student IDs ending in 0, 3, 5, 9
- Task B: `DFS_path_template.S`: Student IDs ending in 1, 4, 8
- Task C: `MCM_template.S`: Student IDs ending in 2, 6, 7

Please run your program using Ripes, and take time to familiarize yourself with the simulator starting this section:

<https://github.com/mortbopet/Ripes/blob/master/docs/introduction.md>

Ensure that your code passes the two test cases we provide and **print out the correct results** (listed in the comments of each template). For example, you must complete Task A and print out

output: Element is not present in array

for Test case 1, and

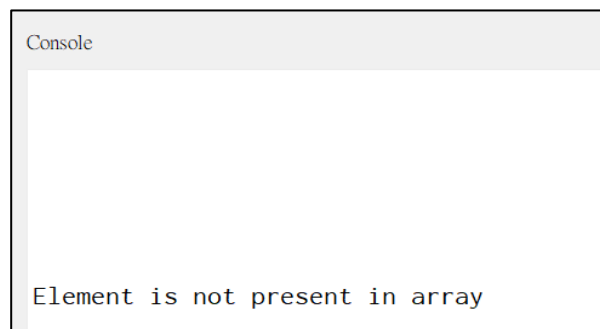
output: Element is present at index 2

for Test case 2 if your student ID ends with '0', '3', '5' or '9'

In your report, you must

1. Include screenshots of your simulation results as evidence of correctness.
2. Provide a flowchart of your implementation.
3. Explain your algorithm clearly and concisely.

For reference, here is a sample simulation result in the Ripes' GUI:



Grading Breakdown

- **Correctness on provided test cases: 44%**
- **Correctness on hidden test cases: 40%**
- **Code explanations, screenshots, and flowchart: 16%**

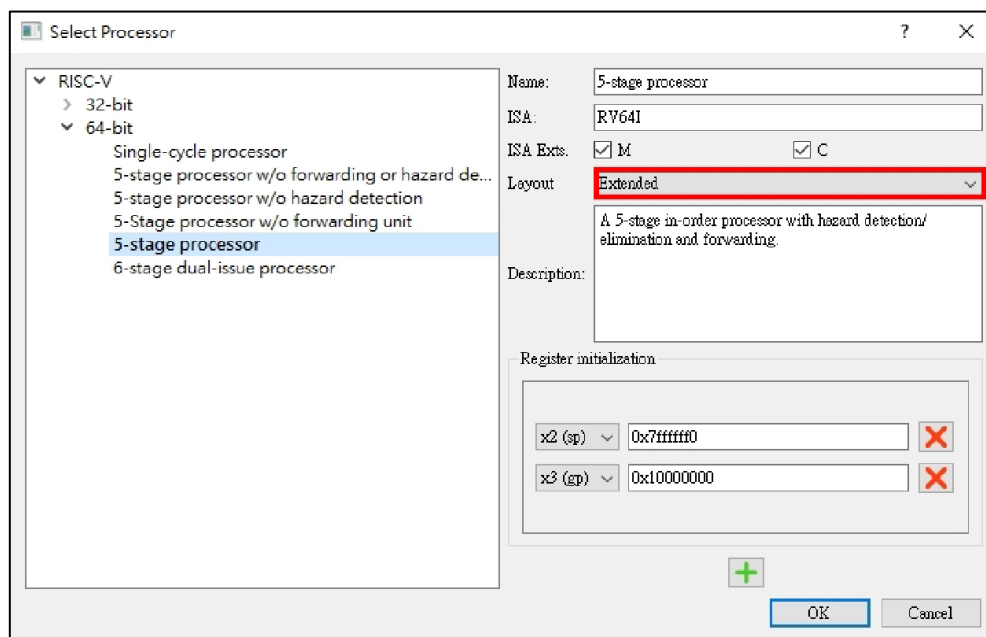
Note 1:

You must write your code using only **RV64I** base integer instructions. Please refer to [RV64i_Base_Integer_Instructions.pdf](#) for the complete instruction set.

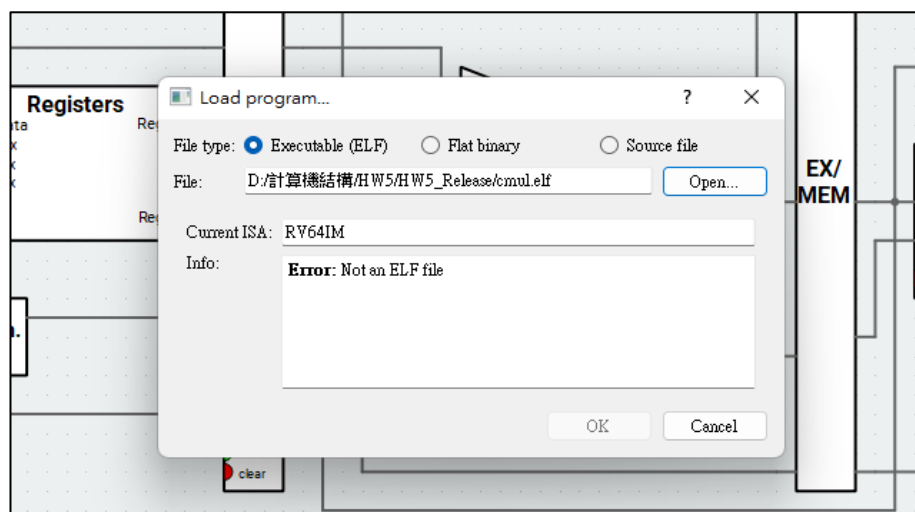
Note 2:

About the settings and simulation with Ripes.

- a. To better understand how your program executes, you can switch the processor view in Ripes to **Extended Mode**, which provides a more detailed visualization of the processor's components and pipeline stages.

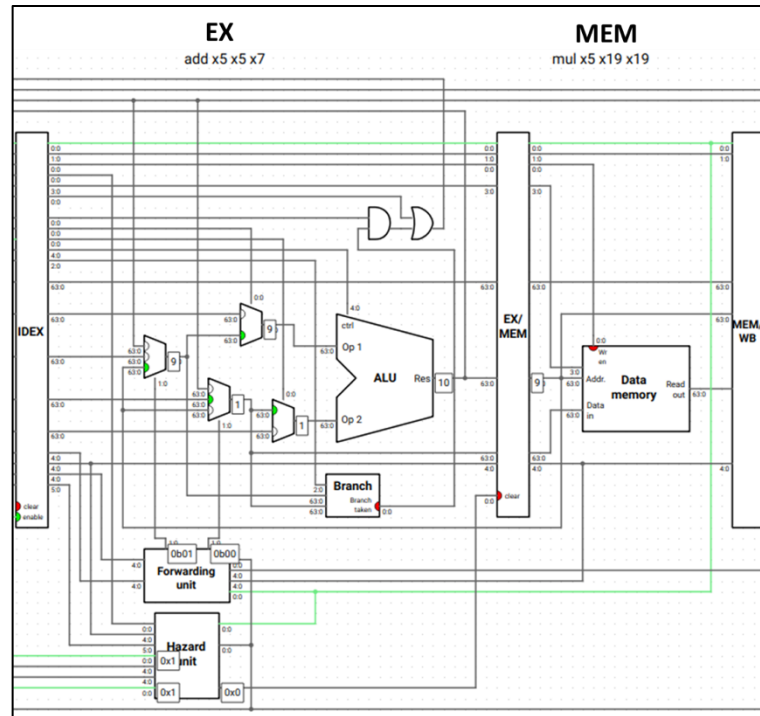


- b. Avoid using **Chinese characters** (or any non-ASCII characters) in your file path, as they may cause unexpected errors during simulation or file access.

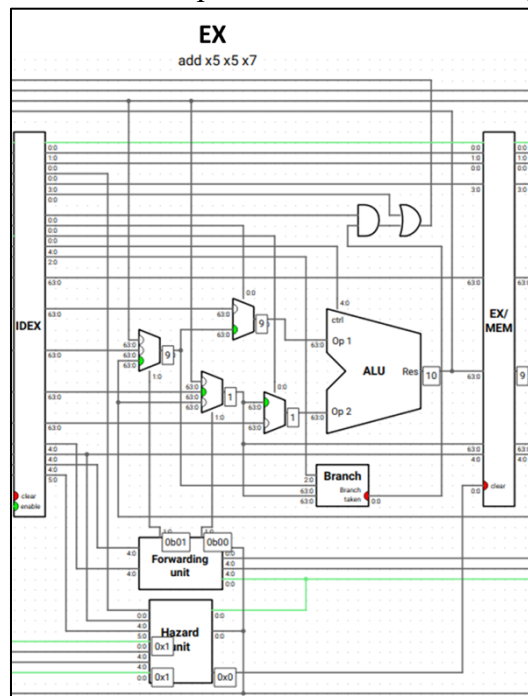


- c. For the EX hazard, the MUX control of ForwardA is (0b01) in Ripes, which is different from the definition in lecture notes and textbook (ForwardA=10); for

the MEM hazard, the MUX control of ForwardA is (0b10) is in Ripes, which is different from the definition in lecture notes and textbook (ForwardA=01). Please follow Ripes' definition in this assignment.




- d. Additionally, Ripes detects control hazards in EX stage, instead of ID stage discussed in lecture notes and textbook for an improved pipelined implementation. Please follow Ripes' definition in this assignment.

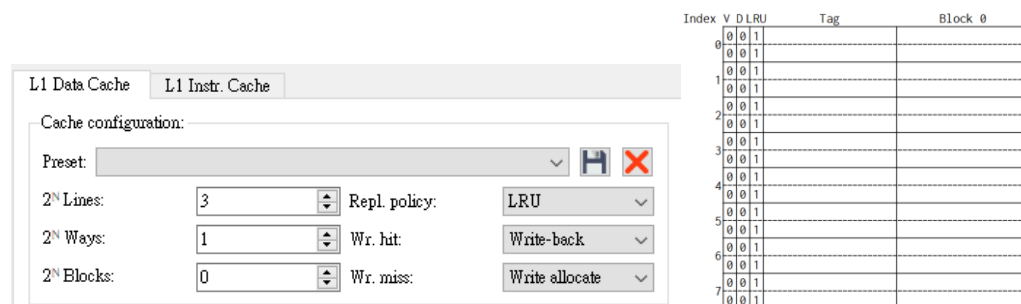


Note 3:

We will use the cache simulator in Ripes to observe the behavior of the cache. First of all, please follow the steps below to set up the cache simulator in Ripes.

- Please load your program `HW5_{ProblemName}_{yourstudent_ID}.elf` into Ripes. Also, remember to set the processor and the global pointer (x3/gp) to *64-bit 5-stage processor* and `base+0x800`, respectively. Be careful that the directory path of your program (.elf file) should contain only alphanumeric characters.

- Click the “Cache button ” in the left panel to open the cache simulator.



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

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 a 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.

- 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 the “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 4:

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 5:

Due to its lightweight and simplified design, the cache simulator in Ripes has certain limitations. For example, after a write hit or miss, the displayed value in the corresponding block does not update immediately. It will only reflect the written value upon the next read access to that block. However, the block is correctly marked as dirty, and it is immediately highlighted in blue to indicate a write. Therefore, please do not rely on the "block" column values in the cache table for verifying write behavior.

Note 6:

Similar to Note 5, the cache simulator in Ripes does not interact reliably with the "Undo button ". For instance, after a write hit that correctly marks a block as dirty, clicking the Undo button may not properly reset the dirty bit (D) 0. If this occurs, simply note the instruction where the issue appeared, then set a breakpoint before that instruction and rerun the simulation. This will allow you to reproduce the correct state and capture an accurate screenshot.

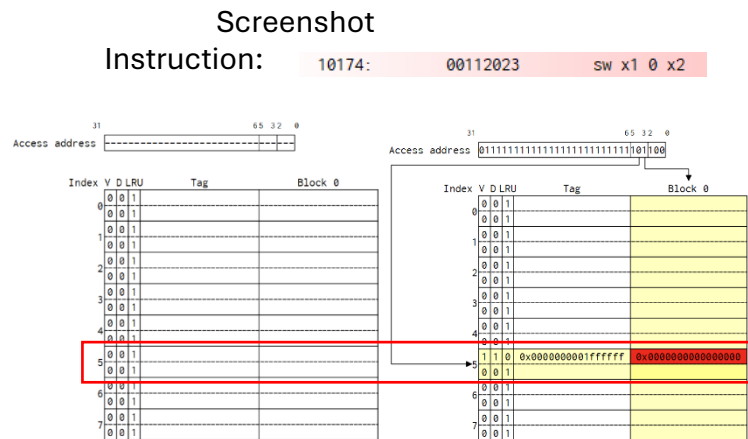
2. (14%) Please run the simulation and identify the following cases. For each cases in (a)~(c), include the screenshots of the following in your report:

- i. The load/store instruction where the case occurs
- ii. The accessed memory address
- iii. The cache states before and after executing that load/store instruction

In addition, provide a brief explanation of what occurred in each case. You may follow the format used in the provided example question.

Example: A write miss at a set (index) where all ways are initially vacant.

Solution:



Explanation

With write allocate, a write miss triggers a block fetch from memory. After the write miss at index 5, the valid bit (V) and dirty bit (D) are both set to 1, and the LRU bit is updated to 0, indicating that the block is now the most recently used.

Cache states before and after the write miss

- (a) (3%) Provide a case of cache insertion at an index (a set) where exactly one way is already occupied. Be sure to explain how the LRU bits of both ways in that set are updated.
- (b) (3%) Provide a case of a cache hit where the block becomes dirty as a result (i.e., it was not dirty before the hit).
- (c) (3%) Provide a case of a cache block replacement where a write-back to memory is required.
- (d) (5%) Improve the hit rate by designing your own cache configuration. You may adjust parameters such as associativity or cache size. Include the following:
 - i. A screenshot of your custom cache design and configuration.
 - ii. A screenshot showing the improved hit rate.
 - iii. A brief explanation of why your configuration improves the hit rate.

This concludes the simulation portion of the assignment. Please proceed with the following written exercises.

Part B: Written Exercises

1. (12%) A processor includes a 64 KB, 4-way set associative L1 data cache with 64-byte blocks. The memory is byte-addressable and uses 38-bit addresses.
 - (a) (2%) How many sets are in the L1 cache?
 - (b) (4%) Break down the 38-bit address into its cache components: tag, index, block offset, and byte offset.
 - (c) (4%) Assuming that each cache block contains one valid bit, one dirty bit, and tag bits in addition to its data, what is the total number of Kbytes required to implement this cache?
 - (d) (2%) The processor accesses memory through a 16-bit memory bus. The cache hit time is 1 cycle, and the hit rate is 92%. A miss penalty takes 150 clock cycles, including all memory access latency and data transfer. Compute the average memory access time (AMAT) in cycles.
2. (24%) Consider two different cache configurations for a system using 23-bit word addresses. Each cache has a total data storage capacity of 256 bytes. All caches are initially empty. The two cache configurations are as follows:
 - Cache 1: Direct-mapped cache with eight-word blocks
 - Cache 2: Two-way set associative cache with four-word blocks and LRU (Least Recently Used) replacement policy.

Note: Each word is four bytes, so each word address maps to four bytes of data.

- (a) (2%) For each cache, determine the number of tag bits stored in each block.
 - (b) (2%) Compute the total number of bits required to implement each cache, including valid, dirty, tag, and data bits.
 - (c) (20%) Given the following read sequence of word-address references:
60, 61, 62, 68, 56, 57, 32, 33, 63, 64, 33, and 30.
For Cache 1 and Cache 2, indicate whether each reference is a hit or miss; show the final cache contents (i.e., the tag value and the word addresses stored in each block or set) after the last reference.
3. (18%) You are analyzing two processor candidates, P1 and P2, each with a single-level L1 data cache. The main memory access takes 80 ns, and 45% of instructions involve data memory accesses (assume that instruction fetches are ignored in this analysis). The following table describes the L1 cache characteristics for two processor candidates, P1 and P2:

Processor	L1 Cache Size	L1 Miss Rate	L1 Hit Time
P1	1 KB	9.0%	0.625 ns
P2	2 KB	8.0%	0.5 ns

Assumption: The L1 hit time determines the cycle times of each processor, i.e., the processor's clock cycle equals its L1 hit time.

- (a) (2%) What are the clock rates (in GHz) of P1 and P2?
 - (b) (2%) Compute the Average Memory Access Time (AMAT) in ns for both P1 and P2.
 - (c) (2%) Assuming a base CPI of 1.0, calculate the total CPI for both P1 and P2.
 - (d) (4%) Which processor is faster? You must justify your answer.

Now, assume an L2 cache is added to P1 with the following characteristics:

L2 Cache Size	L2 Hit Time	L2 Miss Rate (Local)
512 KB	2.5 ns	92%

Use P1's L1 cache characteristics as previously listed. The L2 miss rate indicated in the table is its local miss rate.

- (e) (3%) Compute the AMAT for P1 with the L2 cache. Assuming a base CPI of 1.0, calculate the total for P1 with the L2 cache. Now compare P1 with the L2 cache to P2. Which processor is faster?
 - (f) (5%) From (d), if P1 is faster, what L1 miss rate would P2 require to match P1's performance? If P2 is faster, what L1 miss rate would P1 require to match P2's performance?
4. (16%) Consider a processor with the following memory system characteristics:

Virtual Memory System

Virtual address	42 bits
Physical address	36 bits
Page size	8 KB
L1 TLBs	Separate instruction and data TLBs, each fully associative with 128 entries
L2 TLBs	Unified TLB (for both I/D), 4-way set associative with 512 entries

Cache Organization

Characteristic	L1 Cache	L2 Cache
Type	Split instruction/data caches	Unified (instruction and data)
Size	32 KB for each instruction/data cache	512 KB
Associativity	2-way set associative	4-way set associative
Block size	128 bytes	128 bytes

- (a) Break down the **virtual address** into fields used by the L1 data TLB.
 - (b) Break down the **virtual address** into fields used by the L2 TLB.
 - (c) Break down the **physical address** into fields used by the L1 data cache.
 - (d) Break down the **physical address** into fields used by the L2 cache.
5. (16%) Consider the (7,4) Hamming single error correcting (SEC) code. Each codeword contains 7 bits, including four data bits (d_1 , d_2 , d_3 , and d_4) and three parity bits (p_1 , p_2 , and p_3). Please write down the calculation process in detail. Otherwise, you will get zero points.
- (a) (2%) Show how to encode a four-bit data (1010).
 - (b) (4%) Consider the single error correcting, double error detecting (SECDED) code with an additional parity bit p_4 . Show how to encode a four-bit data (1101).
 - (c) (4%) For the same SECDED, show how to find and correct/detect error(s) in an eight-bit information (01100100).
 - (d) (4%) For the same SECDED, show how to find and correct/detect error(s) in an eight-bit information (11110101).
 - (e) (2%) What is the minimum number of parity bits required to protect 96-bit data using the same method to construct a SECDED code?

Submission

Please submit the following files **individually** (do **not** upload any compressed or zipped files):

1. Report (**PDF format**) named as HW5_{your student ID}.pdf
(e.g., HW5_123456789.pdf).
2. Assembly Source File should be named as
 - HW5_Binary_Search_{your student ID}.S
 - HW5_DFS_path_{your student ID}.S
 - HW5_MCM_{your student ID}.S(e.g., HW5_Binary_Search_123456789.S).