2024 Computer Organization HW 2: Performance Modeling for the µRISC-V Processor

Computer Organization 2024 Programming Assignment II

Due Date: 23:59, May 8, 2024

Overview

RISC-V processors should support a *core* ISA for integer operations, including RV32I, RV32E, RV64I, or RV128I. Additional functionality could be adopted to augment the capability of target RISC-V processors. RISC-V has a series of *standard extensions* to provide additional support beyond the core ISA, such as floating point and bit manipulation *RISC-V ISA List*. On the other hand, there is also a series of *non-standard extensions*, which might be specialized for certain purposes and might conflict with other extensions. If you are interested in the related contents, please refer to the document *Extending RISC-V*.

Particularly, the RISC-V M extension defines multiplication and division operations for integers. The RISC-V F/D extensions are the computation operations for single/double precision floating point numbers. The RISC-V Vector extension is a promising extension for the AI computing as it enables the parallel processing of mathematic operations on a RISC-V processor. The V extension involves adding a vector computation engine on the RISC-V processor, compared with the serial computing on a typical processor.

In this assignment, you will be asked to convert the given C code segments into the corresponding assembly versions, based on the skills you learned from the previous programming assignment. You will use the RISC-V extensions to implement your programs. More importantly, you will be asked to collect the performance data for your written code, and you need to use the performance data to derive the execution time of your program on the RISC-V processor based on a basic performance model, which is provided in the following section. Besides, with the collected performance data, you are able to further characterize the performance of the running programs. A common performance characterization method to analyze if a given program is bounded by CPU or Memory (I/O) is provided below.

1. Performance Modeling

Based on your knowledge learned from **Chapter 1.6 Performance** of our course textbook (ISBN: 0128203315; ISBN-13: 9780128203316), you would derive the CPU execution time of a given program with clock cycles per instruction (CPI), instruction counts, and clock cycle time.

- The performance model used in this assignment uses cycle per instruction (CPI) to summarize the delivered performance of a RISC-V instruction executed on the target RISC-V processor, including the effect of the CPU pipeline and the memory subsystem.
- Given the above modeling concept, you will need to collect the performance data of your program to derive the CPU execution time. Specifically, you need to record the **instruction counts** of different types of RISC-V instructions.
  - The instructions are categorized into seven types and their instruction counts should be recorded (accumulated) in the seven counters: `add_cnt`, `sub_cnt`, `mul_cnt`, `div_cnt`,`lw_cnt`, `sw_cnt`, and `others_cnt`, respectively. The table below lists the instructions and their categories.

  - The CPIs for the seven types of instructions are defined in the given header files, `add_CPI`, `sub_CPI`, `mul_CPI`, `div_CPI`,`lw_CPI`, `sw_CPI`, and `others_CPI` as constants. You should not alter the constant values. The related information for the CPIs is available on the table, too.
  - The derived performance data should be stored in some variables, such as `pi_cycle_count` for the total cycle count calculated in the first execise.
  - NOTE: The `cycle_time` represents the clock cycle time for the target RISC-V processor. It is a constant data and its content should not be altered.
- Variables/Constants defined in the header files used in this assignment.

Var./Cons. Name	Definition
`add_cnt`	used to count `add{i}, fadd.s, fadd.d, vadd.vv, vadd.vx, vadd.vi` instruction
`sub_cnt`	used to count `sub{i}, fsub.s, fsub.d, vsub.vv, vsub.vx` instruction
`mul_cnt`	used to count `mul, fmul.s, fmul.d, vmul.vv, vmul.vx` instruction
`div_cnt`	used to count `div, fdiv.s, fdiv.d, vdiv.vv, vdiv.vx` instruction
`lw_cnt`	used to count `lw, lh, lb, li, lbu, lhu, vle8.v, vle16.v, vle32.v, vle64.v` instruction
`sw_cnt`	used to count `sw, sh, sb, vse8.v, vse16.v, vse32.v, vse64.v` instruction
`others_cnt`	used to count rest of instruction
`add_CPI`	CPI of instructions listed in `add_cnt`
`sub_CPI`	CPI of instructions listed in `sub_cnt`
`mul_CPI`	CPI of instructions listed in `mul_cnt`
`div_CPI`	CPI of instructions listed in `div_cnt`
`lw_CPI`	CPI of instructions listed in `lw_cnt`
`sw_CPI`	CPI of instructions listed in `sw_cnt`
`others_CPI`	CPI of rest of instructions

2. Performance Characterization

In the context of program performance, the terms “CPU-bound” and “Memory-bound” refer to where the bottleneck in a program’s execution might be.

- *CPU-bound:* This term describes a scenario where the execution of a task or program is highly dependent on the CPU. In a CPU-bound environment, the processor is the primary component being used for execution1. This means that other components in the computer system are rarely used during execution. If we want a program to run faster, then we have to increase the speed of the CPU. CPU-bound operations tend to have long CPU bursts. Examples of CPU-bound applications include High-Performance Computing (HPC) systems and graphics operations.
- *Memory-bound:* This term is often used to describe tasks that can slow things down due to memory related operations, such as memory swapping or excessive allocation. When a server is bounded by its memory, it means that the amount of throughput the server can process is limited by its memory. In other words, if you try to process more requests, the memory will reach its limit before the CPU does.

In summary, a CPU-bound task is limited by the computational power of the CPU, while a memory-bound task is limited by the amount of memory available. Optimizing your program’s performance often involves identifying whether it is CPU-bound or memory-bound and then making appropriate adjustments. For instance, a CPU-bound task might be optimized by improving the algorithm’s efficiency, while a memory-bound task might be optimized by improving data structures or memory management.

In this assignment, the source of your developed code is instrumented to obtained the performance data of the code. The following bullet defines the *ratio* that can be used to determine if a given program is either bounded by CPU or Memory.

- The *ratio* of the clock cycles spent on CPU and Memory (I/O) operations
  - It is a simple method used to calculate the ratio.
  - This is achieved by computing the clock cycles of the load/store instructions and the clock cycles of the instructions other than the load/store instructions (these instructions are assumed to be computations on CPU).
  - A formal formula: (clock cycles for the instructions other than load/store instructions)/(clock cycles for all the instructions)

3. What Should You Do in this Assignment?

There are four exercises in this assignment. You will receive the code with the following structure.

```
CO_StudentID_HW2.zip
└── CO_StudentID_HW2/
    ├── answer.h
    ├── pi.h
    ├── arraymul.h
    ├── exercise1_1.c
        ├── pi.c
    ├── exercise1_2.c
        ├── arraymul_baseline.c
    ├── exercise2_1.c
        ├── arraymul_improved_version1.c
    ├── exercise2_2.c
        ├── arraymul_improved_version2.c
    ├── arraymul_input.txt
    ├── N.txt
    ├── arraymul_baseline_cpu_time.txt
    ├── improved_version1_cpu_time.txt
    ├── test_exercise1_1
    ├── test_exercise1_2
    ├── test_exercise2_1
    ├── test_exercise2_2
    ├── makefile
```

- The flies that you should modify in this assignment are as follows.
  - answer.h
  - N.txt
  - pi.h
  - arraymul.h
  - pi.c
  - arraymul_baseline.c
  - arraymul_improved_version1.c
  - arraymul_improved_version2.c

3-0. Performance Data Collection

- The performance data collection is done by the source-level code instrumentation. This means you are responsible to insert the performance probes (i.e., performance analysis code) into your written assembly code (e.g., `pi.c` of the first exercise in 3-1).
- You need to insert the assembly code to *count* the number of executed instructions, according to the types of the instructions, and to store the accumulated counts in the respective counters. You will need to provide the contents of the seven counters, i.e., `add_cnt`, `sub_cnt`, `mul_cnt`, `div_cnt`,`lw_cnt`, `sw_cnt`, and `others_cnt`, as defined in the above table.
  - You may, for example, use the following instruction to increment the content in a counter. The example below increments the `lw_cnt` counter.

    > addi %[lw_cnt], %[lw_cnt], 1\n\t

- You also need to **compute** the total cycle count and the CPU execution time for a given program.
  - The total cycle count can be computed with *counter values* (the performance data you collect) and the *given CPIs* (the constants that have been defined properly in the header files).
  - With the cycle count, you can compute the CPU time easily based on the cycle time of the target processor defined in the header file (e.g., `pi.h` in our first exercise).
- You should also calculate the *ratio* of the time spent on CPU/Memory
  - This is done by using the above collected performance data *counter values* and the *given CPIs*, based on the formula provided above (2. Performance Characterization).

3-1. *pi* calculation (40%)

In this exercise, you are asked to perform the *pi* calculation using the RISC-V assembly codes with the **RV64I** ISA. You need to collect the performance data of the developed assembly code, based on the descriptions provided in 3-0. Performance Data Collection. With the collected performance data, you should compute the *ratio* of the time spent on CPU and memory. You should follow the procedures below to accomplish this exercise.

- The *pi* calculation code should be provided in `pi.c`
  - **NOTE:** You should put your assembly code within the `pi.c` file, as indicated in `asm volatile( #include "pi.c" : [h] "+r"(p_h), ...);`.
- The performance probes should be inserted into your code to collect the performance data, i.e., `add_cnt`, `sub_cnt`, `mul_cnt`, `div_cnt`,`lw_cnt`, `sw_cnt`, and `others_cnt`.
- You also need to **compute** the total cycle count and the CPU execution time for the program.
  - You need to calculate `pi_cycle_count` based on the *counter values* and the *given CPIs* (the constants defined in `pi.h`).
    - The above table defines the variables for keeping the CPI values.
    - Please add the related code (formula) in `answer.h` to set up the value for `pi_cycle_count` by modifying `macro_pi_cycle_count`.
  - You need to compute `pi_cpu_time` with `cycle_time` (the constant of the cycle time for a 2.6GHz processor defined in`pi.h`).
    - Please add the related code (formula) in `answer.h` to compute the value for `pi_cpu_time` by modifying `macro_pi_cpu_time`.
- You need to calculate `pi_ratio` based on the *counter values* (the constants defined in `pi.h`).
  - Please add the related code (formula) in `answer.h` to set up the value for `pi_ratio` by modifying `macro_calc_pi_ratio`.
  - That is, you should follow the high-level concept of the formula provided above (2. Performance Characterization) to implement the `macro_calc_pi_ratio`.
- Information about the CO_StudentID_HW2.zip.
  - There is a header file (`pi.h`) specifying the constants/variables used by this assignment.
  - You can only change N(the number of iterations) in `pi.h`.
  - *NOTE:** Please do not modify the remaining parts of header file.
- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`N`	Number of iterations when calculating π
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz

  - You should run the *pi* program with different `N` values to get the result, which determines if the program is a CPU bound task or a Memory bound task.
- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
  1. The values of the seven counters. (28%)
     - `add_cnt` (4%)
     - `sub_cnt` (4%)

- `mul_cnt` (4%)
- `div_cnt` (4%)
- `lw_cnt` (4%)
- `sw_cnt` (4%)
- `others_cnt` (4%)

2. The total cycle count (`pi_cycle_count`). (4%)
3. The CPU time (`pi_cpu_time`). (4%)
4. What is the maximum value of N and the program is still memory bound? Please manually record the answer into `N.txt`.(4%)

- The C code for the exercise1_1.c is as follows.

```
//exercise1_1.c
/*
 * description: Gregory-Leibniz series
 *
 *    we use Leibniz formula to approximate Pi
 *
 *    pi/4 = (1 - 1/3 + 1/5 - 1/7 + 1/9 - ...)
 *    pi   = 4(1 - 1/3 + 1/5 - 1/7 + 1/9 - ...)
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <math.h>
#include "pi.h"
#include "answer.h"

int main()
{

    double pi = 0.0;
    printf("Number of iterations: %d\n", N);

    /* original code
    for (int i = 0; i < N; i++){
        term = (-1)^i / (2*i+1);
        pi += term;
    }
    */
    asm volatile(
        #include "pi.c"
    : [add_cnt] "+r"(add_cnt), [sub_cnt] "+r"(sub_cnt), [mul_cnt] "+r"(mul_cnt),
[div_cnt] "+r"(div_cnt), [lw_cnt] "+r"(lw_cnt), [sw_cnt] "+r"(sw_cnt),
[others_cnt] "+r"(others_cnt), [pi] "+f"(pi)
    : [N] "r"(N)
    : "f1", "f2", "t1", "t2", "t3", "t4"
    );

    pi=4*pi;

    printf("===== Question 1-1 =====\n");
    printf("PI = %.9lf\n", pi);

    printf("add counter used: %d\n", add_cnt);
    printf("sub counter used: %d\n", sub_cnt);
    printf("mul counter used: %d\n", mul_cnt);
    printf("div counter used: %d\n", div_cnt);
    printf("lw counter used: %d\n", lw_cnt);
    printf("sw counter used: %d\n", sw_cnt);
    printf("others counter used: %d\n", others_cnt);

    macro_pi_cycle_count
    printf("The total cycle count in this program: %.0f\n", pi_cycle_count);

    macro_pi_cpu_time
    printf("CPU time = %f us\n", pi_cpu_time);

    macro_calc_pi_ratio

    if(pi_ratio > 0.5){
        printf("This program is a CPU bound task.\n");
        }
    else{
        printf("This program is a Memory bound task.\n");
        }
    return(0);

}
```

3-2. Array multiplication (a baseline version) (40%)

You need to write the assembly code for the C-based array multiplication, as shown in `exercise1_2.c`, using the **RV64I** ISA. Besides, you should do the same as 3-1 *pi* calculation to collect performance data and derive related performance statistics.

- As shown in the `arraymul_baseline()` function, you are responsible for writing the assembly for the for-loop code: `for (...) y[i] = h[i] * x[i] + c;`.
  - **NOTE:** You should put your assembly code within the `arraymul_baseline.c` file, as indicated in `asm volatile( #include "arraymul_baseline.c" : [h] "+r"(p_h), ...);` within the `arraymul_baseline()` function in `exercise1_2.c`.
  - The header file `arraymul.h` specifies the constants/variables used in this assignment. You are allowed to change `arr_size` (array size) in `arraymul.h` to meet the exercise requirements.
  - **NOTE:** Please do not modify the rest of the header file.
- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition

Var./Cons. Name	Definition
x[ ]	Input array 1 in arraymul.h
h[ ]	Input array 2 in arraymul.h
y[ ]	Output array in arraymul.h
arraymul_baseline_cycle_count	Clock cycle in arraymul_baseline.c you need to calculate
cycle_time	The given clock cycle time of the target RISC-V processor running at 2.6 GHz
arraymul_baseline_cpu_time	The CPU time in arraymul_baseline.c you need to calculate
arr_size	Size of the array

student_id	student_id = your_student_id % 100
	i.g. F12345678:
	student_id = 12345678 % 100 = 78

- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
    1. The values of seven counters. (28%)
        - add_cnt (4%)
        - sub_cnt (4%)
        - mul_cnt (4%)
        - div_cnt (4%)
        - lw_cnt (4%)
        - sw_cnt (4%)
        - others_cnt (4%)
    2. The total cycle count (arraymul_baseline_cycle_count). (4%)
    3. The CPU time (arraymul_baseline_cpu_time). (4%)
    4. Choose any arr_size but **log~2~(arr_size) must be a integer** and answer the question: Is this program a CPU bound or Memory bound program? (4%)
- The arraymul_baseline() function in exercise1_2.c is as follows.

```
//The code snippet for arraymul_baseline() in exercise1_2.c
void arraymul_baseline(){
    short int *p_h = h;
    short int *p_x = x;
    short int *p_y = y;
    short int id = student_id; // id should be your_student_id % 100, please check
the header file, arraymul.h.
    /* original C code
    for (int i = 0; i < arr_size; i++){
        p_y[i] = p_h[i] * p_x[i] + id;
    }
    */

    asm volatile(
        #include "arraymul_baseline.c"
    : [h] "+r"(p_h), [x] "+r"(p_x), [y] "+r"(p_y), [add_cnt] "+r"(add_cnt),
[sub_cnt] "+r"(sub_cnt), [mul_cnt] "+r"(mul_cnt), [div_cnt] "+r"(div_cnt),
[lw_cnt] "+r"(lw_cnt), [sw_cnt] "+r"(sw_cnt), [others_cnt] "+r"(others_cnt)
    : [id] "r"(id), [arr_size] "r"(arr_size)
    : "t0", "t1"

    );

    printf("\n===== Question 1-2 =====\n");

    printf("output: ");
    for (int i = 0; i < arr_size; i++){
        printf(" %d ", y[i]);
    }

    printf("\n");

    printf("add counter used: %d\n", add_cnt);
    printf("sub counter used: %d\n", sub_cnt);
    printf("mul counter used: %d\n", mul_cnt);
    printf("div counter used: %d\n", div_cnt);
    printf("lw counter used: %d\n", lw_cnt);
    printf("sw counter used: %d\n", sw_cnt);
    printf("others counter used: %d\n", others_cnt);

    macro_arraymul_baseline_cycle_count
    printf("The total cycle count in this program: %.0f\n",
arraymul_baseline_cycle_count);

    macro_arraymul_baseline_cpu_time
    printf("CPU time = %f us\n", arraymul_baseline_cpu_time);

    macro_calc_arraymul_baseline_ratio

    if(arraymul_baseline_ratio > 0.5)
        printf("This program is a CPU bound task.\n");
    else
        printf("This program is a Memory bound task.\n");

    //record the cpu time
    FILE *fp;
    fp = fopen("arraymul_baseline_cpu_time.txt", "w");
    fprintf(fp, "%f", arraymul_baseline_cpu_time);
    fclose(fp);
}
```

3-3. Array multiplication (a vectorized version with the V extension, *improved_version1*) (50%)

You need to re-write the assembly code, which you build in the previous exercise, using the **RISC-V V** extension. Before you write the code, you are suggested to study the RISC-V V extension document (from p. 10 to p. 31 and p. 55) to get familiar with the concept of RISC-V vector programming. After you write the vectorized program, you should collect the performance data and derive related performance statistics as you did in the previous exercise.

- As shown in the `improved_version1` function, you are responsible for writing the assembly for the for-loop in C: `for (...) y[i] = h[i] * x[i] + c;`.
    - **NOTE:** You should put your assembly code within the `arraymul_improved_version1.c` file, as indicated in `asm volatile( #include "arraymul_improved_version1.c" : [h] "+r" (p_h), ...);` within the `improved_version1()` function in `exercise2_1.c`.
    - Your code should use the *RISC-V V Extension* and run with Spike simulator using the specific configurations (i.e., **vlen=128, elen=16**). The vectorized version would improve the execution efficiency, thanks to the parallel computations done in the vector computation engine.
    - **NOTE:** Please do not modify the rest of the header file.
- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`x[ ]`	Input array 1 in `arraymul.h`
`h[ ]`	Input array 2 in `arraymul.h`
`y[ ]`	Output array in `arraymul.h`
`improved_version1_cycle_count`	Clock cycle in `vectorsum_improved_version1.c` you need to calculate
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz
`improved_version1_cpu_time`	The CPU time in `vectorsum_improved_version1.c` you need to calculate
`arr_size`	Size of the arrays used in this exercise
`student_id`	`student_id` = your_student_id % 100 i.g. F12345678: `student_id` = 12345678 % 100 = 78

- Your obtained scores of this exercise is determined by the correctness of your reported performance data and the efficiency of your developed code against the serial version in the previous exercise.
    1. The values of seven counters. (28%)
        - `add_cnt` (4%)
        - `sub_cnt` (4%)
        - `mul_cnt` (4%)
        - `div_cnt` (4%)
        - `lw_cnt` (4%)
        - `sw_cnt` (4%)
        - `others_cnt` (4%)
    2. The total cycle count (`improved_version1_cycle_count`). (4%)
    3. The CPU time (`improved_version1_cpu_time`). (4%)
    4. Achieved speedup. (14%)
        - If `6 < speedup`, you get **(14 pt)**.
        - If `4 < speedup < 6`, you get **(9 pt)**.

    - If $2 < speedup < 4$, you get **(5 pt)**.
    - If $speedup < 2$, you get **(0 pt)**.
- The improved_version1 function in exercise2_1.c is as follows.

```
//The code snippet for improved_version1() in exercise2_1.c
void improved_version1(){
    short int *p_h = h;
    short int *p_x = x;
    short int *p_y = y;
    short int id = student_id;// id = your_student_id % 100;
    /* original C code
    for (int i = 0; i < arr_size; i++){
        p_y[i] = p_h[i] * p_x[i] + id;
    }
    */

    asm volatile(
        #include "arraymul_improved_version1.c" // Write your code in this file:
arraymul_improved_version1.c

    : [h] "+r"(p_h), [x] "+r"(p_x), [y] "+r"(p_y), [add_cnt] "+r"(add_cnt),
[sub_cnt] "+r"(sub_cnt), [mul_cnt] "+r"(mul_cnt), [div_cnt] "+r"(div_cnt),
[lw_cnt] "+r"(lw_cnt), [sw_cnt] "+r"(sw_cnt), [others_cnt] "+r"(others_cnt)
    : [id] "r"(id), [arr_size] "r"(arr_size)
    : "t0", "v0", "v1", "v2"
    );

    printf("\n===== Question 2-1 =====\n");
    printf("output: ");
    for (int i = 0; i < arr_size; i++){
        printf(" %d ", y[i]);
    }

    printf("\n");

    printf("add counter used: %d\n", add_cnt);
    printf("sub counter used: %d\n", sub_cnt);
    printf("mul counter used: %d\n", mul_cnt);
    printf("div counter used: %d\n", div_cnt);
    printf("lw counter used: %d\n", lw_cnt);
    printf("sw counter used: %d\n", sw_cnt);
    printf("others counter used: %d\n", others_cnt);
    macro_improved_version1_cycle_count
    printf("The total cycle count in this program: %.0f\n",
improved_version1_cycle_count);


    macro_improved_version1_cpu_time
    printf("CPU time = %f us\n", improved_version1_cpu_time);
    FILE *fp_1;
    fp_1 = fopen("improved_version1_cpu_time.txt", "w");
    fprintf(fp_1, "%f", improved_version1_cpu_time);
    fclose(fp_1);

    float speedup = 0.0;

    FILE *fp_2;
    fp_2 = fopen("arraymul_baseline_cpu_time.txt", "r");
    fscanf(fp_2, "%f", &speedup);
    fclose(fp_2);
    speedup = speedup / improved_version1_cpu_time;
    printf("The V extension version is %f times faster than the baseline
version\n", speedup);
}
```

3-4. Array multiplication (an improved, vectorized version, *improved_version2*) (10%)

You need to modify the assembly code, which you developed in the previous exercise, with the **RISC-V V** extension. It is important to note that the specification of the vector engine of our virtual RISC-V (i.e., **vlen=256, elen=16**) is different from the one used in the previous exercise (i.e., **vlen=128, elen=16**). Specifically, you are asked to achieve **the same speed up** as `arraymul_improved_version1.c` in the previous exercise. In other words, to get familiar with the design of the vector engine, the performance improvement of this assignment against the baseline version in 3-2 array multiplication should be the same as the performance improvement achieved in 3-3. Array multiplication (a vectorized version with the V extension, improved_version1). Besides, you should do the same as 3-3 to collect performance data and derive related performance statistics.

- As shown in the `improved_version2()` function, you are responsible for writing the assembly for the for-loop code: `for (...) y[i] = h[i] * x[i] + c;`.
    - **NOTE:** You should put your assembly code within the `arraymul_improved_version2.c` file, as indicated in `asm volatile( #include "arraymul_improved_version2.c" : [h] "+r" (p_h), ...);` within the `improved_version2()` function in `exercise2_2.c`.
    - Your code should use the *RISC-V V Extension* and run with Spike simulator using the specific configurations (i.e., **vlen=256, elen=16**).
    - **NOTE:** Please do not modify the rest of the header file.
- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`x[ ]`	Input array 1 in `arraymul.h`
`h[ ]`	Input array 2 in `arraymul.h`
`y[ ]`	Output array in `arraymul.h`
`improved_version2_cycle_count`	Clock cycle in `vectorsum_improved_version2.c` you need to calculate
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz
`improved_version2_cpu_time`	The CPU time in `vectorsum_improved_version2.c` you need to calculate

Var./Cons. Name	Definition
arr_size	Size of the arrays used in this exercise

student_id	student_id = your_student_id % 100
	i.g. F12345678:
	student_id = 12345678 % 100 = 78

- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
    1. Achieved speedup.(10%)
        - As described above, the printed out speedup number should be **1**.
- The improved_version2 function in exercise2_2.c is as follows.

```
//The code snippet for improved_version2() in exercise2_2.c
void improved_version2(){
    short int *p_h = h;
    short int *p_x = x;
    short int *p_y = y;
    short int id = student_id;// id = your_student_id % 100;

    /* original C code
    for (int i = 0; i < arr_size; i++){
        p_y[i] = p_h[i] * p_x[i] + id;
    }
    */

    asm volatile(
        #include "arraymul_improved_version2.c"

    : [h] "+r"(p_h), [x] "+r"(p_x), [y] "+r"(p_y), [add_cnt] "+r"(add_cnt),
[sub_cnt] "+r"(sub_cnt), [mul_cnt] "+r"(mul_cnt), [div_cnt] "+r"(div_cnt),
[lw_cnt] "+r"(lw_cnt), [sw_cnt] "+r"(sw_cnt), [others_cnt] "+r"(others_cnt)
    : [id] "r"(id), [arr_size] "r"(arr_size)
    : "t0", "v0", "v1", "v2"
    );

    printf("\n===== Question 2-2 =====\n");
    printf("output: ");
    for (int i = 0; i < arr_size; i++){
        printf(" %d ", y[i]);
    }
    printf("\n");

    printf("add counter used: %d\n", add_cnt);
    printf("sub counter used: %d\n", sub_cnt);
    printf("mul counter used: %d\n", mul_cnt);
    printf("div counter used: %d\n", div_cnt);
    printf("lw counter used: %d\n", lw_cnt);
    printf("sw counter used: %d\n", sw_cnt);
    printf("others counter used: %d\n", others_cnt);
    macro_improved_version2_cycle_count
    printf("The total cycle count in this program: %.0f\n",
improved_version2_cycle_count);


    macro_improved_version2_cpu_time
    printf("CPU time = %f us\n", improved_version2_cpu_time);

    float speedup = 0.0;

    FILE *fp;
    fp = fopen("improved_version1_cpu_time.txt", "r");
    fscanf(fp, "%f", &speedup);
    fclose(fp);
    speedup = speedup / improved_version2_cpu_time;
    printf("The enhanced V extension version is %f times faster than the baseline
vectorized version\n", speedup);
}
```

4. Test Your assignment

The local-judge system is used to check the results of your developed code. You can run your developed programs and validate their results via the `make` commands below. The following example commands can do individual tests for each exercise.

- Test your code in exercise1_1.c

  ```
  $ make judge_exercise1_1
  ```

- Test your code in exercise1_2.c

  ```
   $ make judge_exercise1_2
  ```

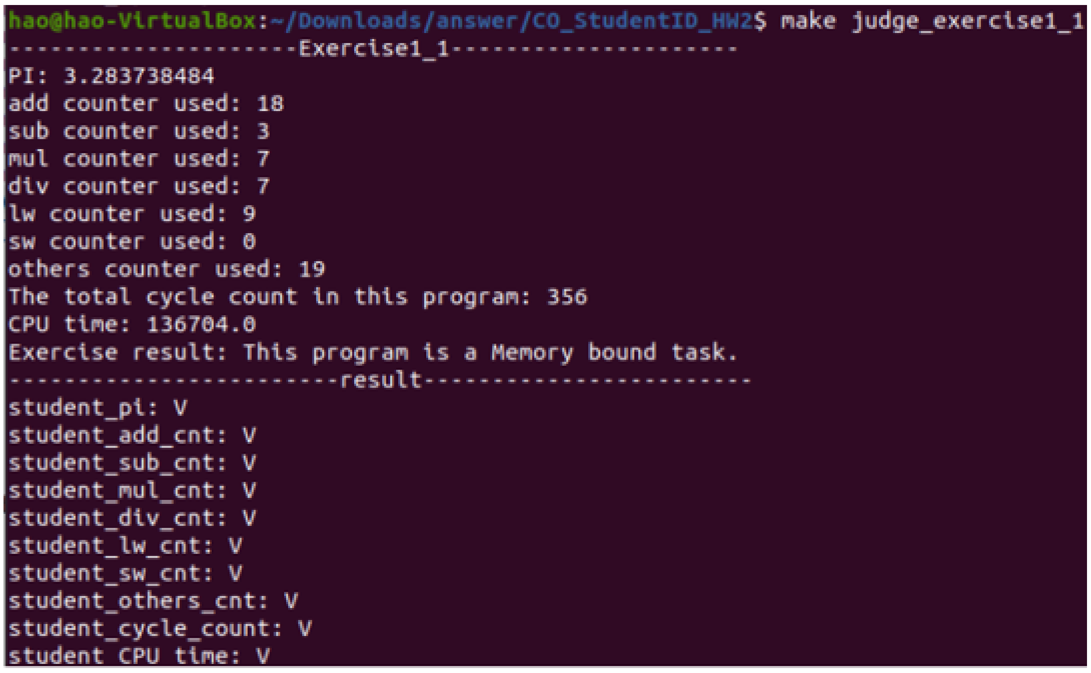
- Test your code in exercise2_1.c

  ```
  $ make judge_exercise2_1
  ```

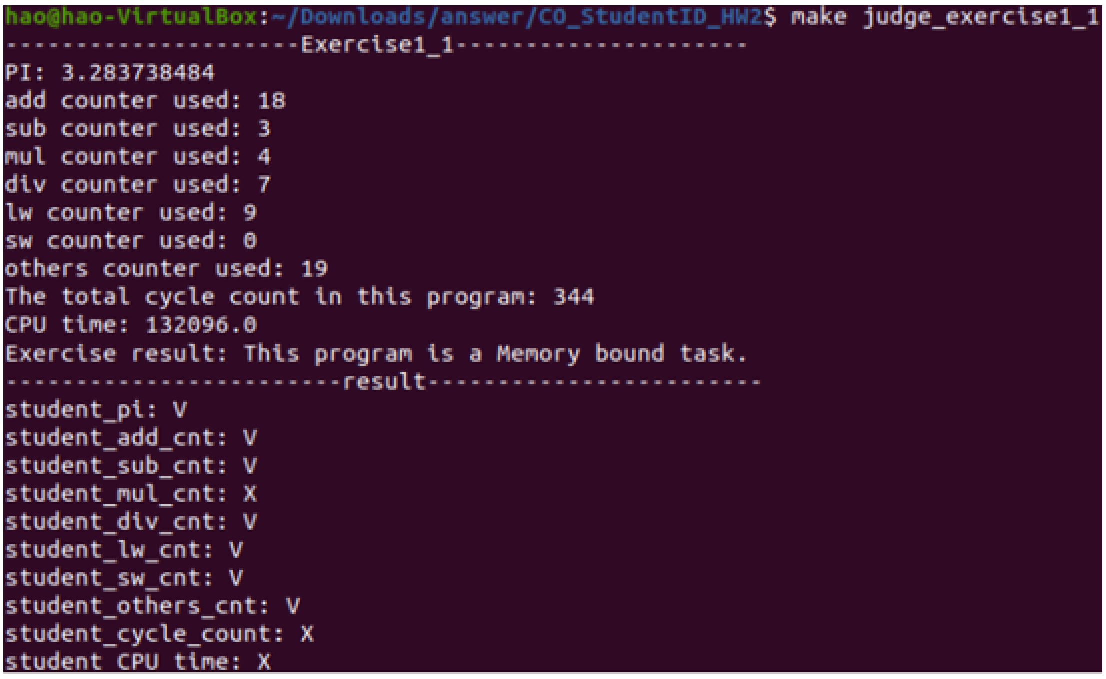
- Test your code in exercise2_2.c

  ```
  $ make judge_exercise2_2
  ```

> If the path of your installed proxy kernel is not `/opt/riscv/riscv64-unknown-elf/bin/pk`, you should change it in `PK_PATH` in makefile.

- Example outputs of the make commands
  - ***Pass:*** 

```
hao@hao-VirtualBox:~/Downloads/answer/CO_StudentID_HW2$ make judge_exercise1_1
--------------------Exercise1_1--------------------
PI: 3.283738484
add counter used: 18
sub counter used: 3
mul counter used: 7
div counter used: 7
lw counter used: 9
sw counter used: 0
others counter used: 19
The total cycle count in this program: 356
CPU time: 136704.0
Exercise result: This program is a Memory bound task.
-----------------------result-----------------------
student_pi: V
student_add_cnt: V
student_sub_cnt: V
student_mul_cnt: V
student_div_cnt: V
student_lw_cnt: V
student_sw_cnt: V
student_others_cnt: V
student_cycle_count: V
student CPU time: V
```

  - ***Error:*** 

```
hao@hao-VirtualBox:~/Downloads/answer/CO_StudentID_HW2$ make judge_exercise1_1
--------------------Exercise1_1--------------------
PI: 3.283738484
add counter used: 18
sub counter used: 3
mul counter used: 4
div counter used: 7
lw counter used: 9
sw counter used: 0
others counter used: 19
The total cycle count in this program: 344
CPU time: 132096.0
Exercise result: This program is a Memory bound task.
-----------------------result-----------------------
student_pi: V
student_add_cnt: V
student_sub_cnt: V
student_mul_cnt: X
student_div_cnt: V
student_lw_cnt: V
student_sw_cnt: V
student_others_cnt: V
student_cycle_count: X
student CPU time: X
```

5. Submission of Your Assignment

Your developed codes should be put into the folder: `CO_StudentID_HW2`. Please follow the instructions below to submit your programming assignment.

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

```
CO_StudentID_HW2.zip
└── CO_StudentID_HW2/
```

```
        ├── answer.h
        ├── N.txt
        ├── pi.c
        ├── arraymul_baseline.c
        ├── arraymul_improved_version1.c
        ├── arraymul_improved_version2.c
```

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

7. References

- RISC-V V Extension
- RISC-V ISA List
- Extending RISC-V
- RISC-V-Spec