Computer Organization 2025 Programming Assignment II

Computer Organization 2025 Programming Assignment II: Performance Modeling for the μRISC-V Processor

Due Date: 23:59, May 14, 2025

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 (3. Performance Data Collection), 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 (1. Performance Modeling). Besides, with the collected performance data, you are able to further characterize the performance of the running programs, where a common performance characterization method to analyze if a given program is bounded by CPU or Memory (I/O) is provided in 2. Performance Characterization.

1. Performance Modeling

Based on your knowledge learned from **Chapter 1.6 Performance** of our course textbook, 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 fixed CPI numbers to summarize the delivered performance of RISC-V instructions 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 following table lists the variables used in this assignments to count the number of Integer, Memory Access, Floating Point operations. In addition, the CPIs of different instruction types are provided. The following explanation uses Integer and Memory Access instructions as an example to elaborate the usages of the counters and CPIs.
  - 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 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 derived performance data should be stored in some variables, such as `fft_cycle_count` for the total cycle count calculated in the first exercise.
  - **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 for the counters and CPIs used in this assignment are shown in the table. These data are defined in the header files included in this assignment files.

Var./Cons. Name	Definition
`add_cnt`	used to count `add{i}, vadd.vv, vadd.vx, vadd.vi` instruction
`sub_cnt`	used to count `sub{i}, vsub.vv, vsub.vx` instruction
`mul_cnt`	used to count `mul, vmul.vv, vmul.vx` instruction
`div_cnt`	used to count `div, vdiv.vv, vdiv.vx` instruction
`lw_cnt`	used to count `lw, lh, lb, 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
`fadd_cnt`	used to count `fadd.s, vfadd.vf, vfadd.vv` instruction
`fsub_cnt`	used to count `fsub.s, vfsub.vf, vfsub.vv` instruction
`fmul_cnt`	used to count `fmul.s, vfmul.vf, vfmul.vv` instruction
`fdiv_cnt`	used to count `fdiv.s, vfdiv.vf, vfdiv.vv` instruction
`flw_cnt`	used to count `flw` instruction
`fsw_cnt`	used to count `fsw` instruction
`dadd_cnt`	used to count `fadd.d` instruction
`dsub_cnt`	used to count `fsub.d` instruction
`dmul_cnt`	used to count `fmul.d` instruction
`ddiv_cnt`	used to count `fdiv.d` instruction
`dlw_cnt`	used to count `fld` instruction
`dsw_cnt`	used to count `fsd` 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`
`fadd_CPI`	CPI of instructions listed in `fadd_cnt`
`fsub_CPI`	CPI of instructions listed in `fsub_cnt`
`fmul_CPI`	CPI of instructions listed in `fmul_cnt`
`fdiv_CPI`	CPI of instructions listed in `fdiv_cnt`
`flw_CPI`	CPI of instructions listed in `flw_cnt`
`fsw_CPI`	CPI of instructions listed in `fsw_cnt`

`dadd_CPI`	CPI of instructions listed in `dadd_cnt`
`dsub_CPI`	CPI of instructions listed in `dsub_cnt`
`dmul_CPI`	CPI of instructions listed in `dmul_cnt`
`ddiv_CPI`	CPI of instructions listed in `ddiv_cnt`
`dlw_CPI`	CPI of instructions listed in `dlw_cnt`
`dsw_CPI`	CPI of instructions listed in `dsw_cnt`

2. Performance Characterization

In the context of program performance analysis, the terms “CPU-bound” and “memory-bound” indicate 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 (CPU core for calculations). In a CPU-bound environment, the processor is the primary component being used for execution. 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 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 processor core, while a memory-bound task is limited by the memory subsystem (e.g., 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)
- If the *ratio* > 0.5, then the program will be considered as a CPU-bound program.
- Otherwise, it is considered as a Memory-bound program.

3. 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 exercise 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 counters (i.e., `add_cnt`, `sub_cnt`, `mul_cnt`, `div_cnt`, `lw_cnt`, `sw_cnt`, `others_cnt`, etc.), 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., `fft.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).

4. Assignment

There are five exercises in this assignment, with a total score of **120 points**.

- **Exercise 1 (Fast Fourier Transform Calculation, 40%)**: Implement Fast Fourier Transform (FFT) and collect the performance data using inline assembly.
- **Exercise 2 (Array Multiplication with/without V extension, 40%)**
  - Exercise 2-1 (18%): Implement **single floating-point array multiplication without V extension** and collect the performance data using inline assembly.
  - Exercise 2-2 (22%): Implement **single floating-point array multiplication with V extension**, collect the performance data using inline assembly and compare the performance with Exercise 2-1.
- **Exercise 3 (Single/Double Floating-point Multiplication, 40%)**
  - Exercise 3-1 (16%): Implement **single floating-point multiplication** and collect performance data using inline assembly.
  - Exercise 3-2 (24%): Implement **double floating-point multiplication**, collect performance data using inline assembly and compare the performance and relative error with Exercise 3-1.

The project files of this assignment will look like this:

```
CO_StudentID_HW2/
├── macro_define.h
├── arraymul.h
├── fft.h
├── exercise1.c
    ├── complex_add.c
    ├── complex_sub.c
    ├── complex_mul.c
    ├── bit_reverse.c
    ├── log2.c
    ├── pi.c
├── exercise2_1.c
    ├── arraymul_baseline.c
├── exercise2_2.c
    ├── arraymul_improved.c
├── exercise3_1.c
    ├── arraymul_float.c
├── exercise3_2.c
    ├── arraymul_double.c
├── arraymul_baseline_cpu_time.txt
├── arraymul_input.txt
├── arraymul_input2.txt
├── arraymul_vector_cpu_time.txt
├── float_cpu_time.txt
├── float_result.txt
├── judge_exercise1
├── judge_exercise2_1
├── judge_exercise2_2
├── judge_exercise3_1
```

```
├── judge_exercise3_2
└── makefile
```

Important Notes

- You **should** write the code on your own.
- You **must** write your code inside `asm volatile()`. Any modifications outside of `asm volatile()` are **not allowed**, unless explicitly specified.
- There will be **hidden test cases** for all exercises, please make sure your program can run correctly.
- The following C code contains only essential parts for the explanation of this assignment. Please download the project files from NCKU Moodle as the primary source for code development.
- **!!Pseudo instructions are not allowed in this assignment.!!**
- **Please don't write any comments in the C files of inline assembly codes.**
- **Remember to check the course announcements for the latest updates and reminders.**

Exercise 1. Fast Fourier Transform Calculation (40%)

In this exercise, you are asked to perform the Fast Fourier Transform (FFT) calculation using the RISC-V assembly codes with the **RV64G** ISA and C codes. Besides, you need to collect the performance data of the developed assembly code, based on the descriptions provided in *3. Performance Data Collection*. With the collected performance data, you should compute the *ratio* of the time spent on CPU and memory.

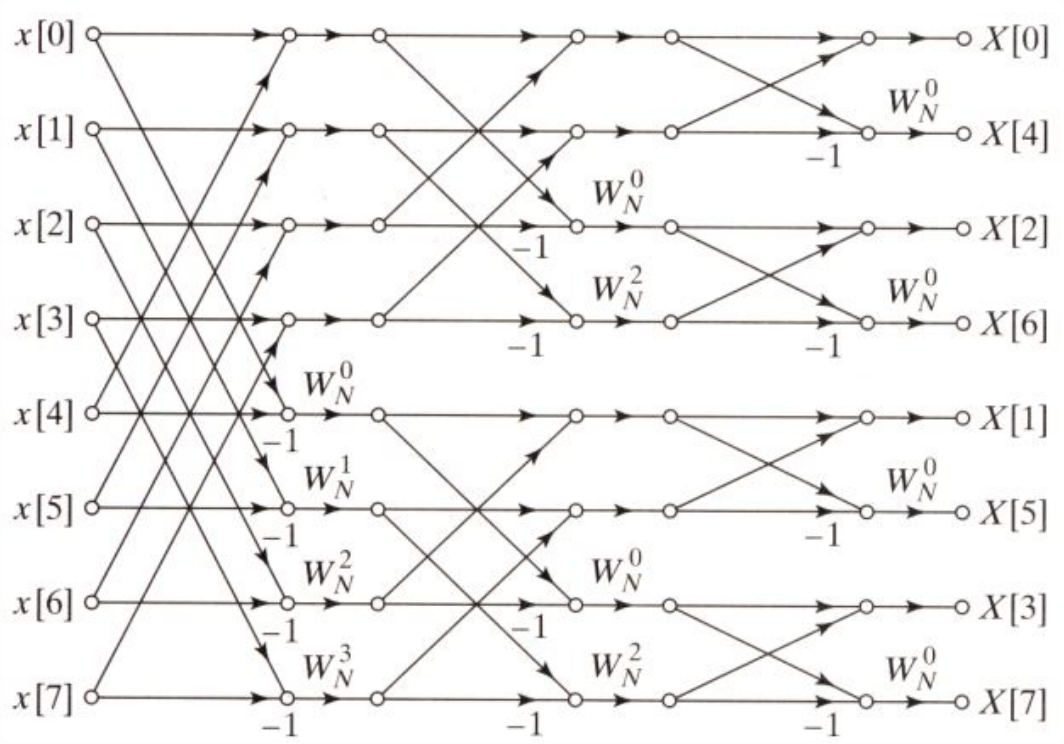
We follow *Cooley-Tukey Fast Fourier Transform Algorithm* which uses *Divide and Conquer* method to implement *iterative radix-2 DIF (Decimation-in-frequency) FFT*. This algorithm restricts the number of input data that must be the power of two.

The procedure of this algorithm and the graph are listed below (Assume the number of input data is eight):

1. Execute *butterfly operation* iteratively (the lines in the graph), and merge the results together.
    - W is the twiddle factor (`W = e^(-j2πk/N)`), while `e^(-j*θ) = cos(θ) - (j * sin(θ))` .
      `j` is imaginary number, `k` is the current index, and `N` is the number of input data.
    - Original input data is `x[0]~x[7]` . After the operation, we will get the data with the order like: `x[0]x[4]x[2]x[6]x[1]x[5]x[3]x[7]` . As shown in right-side of the following graph.

2. Use *bit-reverse operation* to re-order the result of *butterfly operation*.
    - The result of *butterfly operation* is `x[0]x[4]x[2]x[6]x[1]x[5]x[3]x[7]`. After re-ordering, we will get the data with the order as original input: `x[0]~x[7]`.
3. Finally, get the result of FFT (`X[0]~X[7]`).

For more explanation of this algorithm, please refer to Cooley-Tukey DIF FFT Introduction and Cooley-Tukey DIT FFT Introduction.



Important things you need to know to accomplish this exercise:

- There are six functions which are `complex_add()`, `complex_sub()`, `complex_mul()`, `bit_reverse()`, `log2()`, and `pi()` in `exercise1.c`. You need to complete them by assembly code.
    - **NOTE:** You should put your assembly code within the respective C file (e.g., `complex_add.c`), as indicated below.
    - **NOTE:** Please do not modify the rest of the program.

```
Complex complex_add(Complex a, Complex b)
{
    Complex result;
    asm volatile(
        #include "complex_add.c"
        :[C_Re] "=f"(result.Re), [C_Im] "=f"(result.Im), [fadd_cnt] "+r"(fadd_cn
        :[A_Re] "f"(a.Re), [B_Re] "f"(b.Re), [A_Im] "f"(a.Im), [B_Im] "f"(b.Im))
    );
    return result;
}
```

- Input:
  - This exercise takes input from the file: `fft_input.txt`.
  - **NOTE:** It must be **eight pairs** of **single floating-point** numbers, from `-10.0` to `10.0`.
  - Each two numbers represent the real and imaginary parts of a complex element, so the elements in data will be: `1.0+0.0j, 1.0+0.0j, 1.0+0.0j, 1.0+0.0j, 0.0+0.0j, 0.0+0.0j, 0.0+0.0j, 0.0+0.0j`.

```
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

- Output:
  An example output of the FFT program is shown below. You need to implement the six functions and the contents of the X array will be printed out by itself.

```
===== Question 1-1 =====
FFT result:
X[0] = 4.000000 + 0.000000j
X[1] = 1.001207 + -2.414567j
X[2] = 0.000000 + 0.000000j
X[3] = 1.000914 + -0.413859j
X[4] = 0.000000 + 0.000000j
X[5] = 0.999793 + 0.414567j
X[6] = 0.000000 + 0.000000j
X[7] = 0.998086 + 2.413859j

PI = 3.140593

add counter used: 2007
sub counter used: 0
mul counter used: 1000
...
```

- The performance probes should be inserted into your code to collect the performance data, e.g., `add_cnt`, `mul_cnt`, and `lw_cnt`. A complete list for the related counters is provided in Scoring Criteria below.
  - You can also refer to the table in *1. Performance Modeling*.
  - There is an example to insert performance probes.

```
"addi %[fadd_cnt], %[fadd_cnt], 1\n\t" // A instruction to collect
// the performance probe for "fadd.s" instruction below.
"fadd.s f0, f2, f3\n\t"
"addi %[fmul_cnt], %[fmul_cnt], 1\n\t" // A instruction to collect
// the performance probe for "fmul.s" instruction below.
"fmul.s f1, f0, f0\n\t"
"addi %[flw_cnt], %[flw_cnt], 1\n\t" // A instruction to collect
```

```
// the performance probe for "flw" instruction below.
"flw f0, 0(t0)\n\t"
...
```

- You also need to **compute** the total cycle count (`fft_cycle_count`), the CPU execution time (`fft_cpu_time`) and CPU instruction/Memory instruction ratio (`fft_ratio`) for the program.
    - The program output will read the above variables and print accordingly.

- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`iter`	Number of iterations when calculating π. **Do Not Modified**
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz. **Do Not Modified**
`N`	Number of elements for the input data. **Do Not Modified**
`fft_cycle_count`	The total clock cycle used in exercise 1. You need to calculate and define the formula in `macro_define.h` by modifying `macro_fft_cycle_count`
`fft_cpu_time`	The CPU time in exercise 1. You need to calculate and define the formula in `macro_define.h` by modifying `macro_fft_cpu_time`
`fft_ratio`	The ratio used to determine whether the program is a CPU bound or Memory bound program. You should follow the high-level concept of the formula provided above (2. Performance Characterization) to implement the `macro_calc_fft_ratio` in `macro_define.h`

- The files you will modify in this exercise:
    - `complex_add.c`
    - `complex_sub.c`
    - `complex_mul.c`
    - `log2.c`

    - `bit_reverse.c`
    - `pi.c`
    - `macro_define.h`

- **Scoring Criteria**: Your obtained scores of this exercise is determined by the correctness of your reported performance data. If the result is incorrect, you won't get the scores below.

    - **NOTE:** When judging the exercise, we will replace `fft_input.txt` with hidden test cases to verify your code.

    1. The result of Fast Fourier Transform. (8%)
        - The score is determined by the correctness of the printed data for the X array.
        - **NOTE:** You may modify `fft_input.txt` to verify the correctness of your code. It is important that the input file must contain **eight pairs** of **single floating-point** numbers (a total of sixteen numbers).
    2. The values of the counters. (22%)
        - `add_cnt` (2%)
        - `sub_cnt` (2%)
        - `mul_cnt` (2%)
        - `div_cnt` (2%)
        - `lw_cnt` (2%)
        - `sw_cnt` (2%)
        - `fadd_cnt` (2%)
        - `fsub_cnt` (2%)
        - `fmul_cnt` (2%)
        - `fdiv_cnt` (2%)
        - `others_cnt` (2%)
    3. The total cycle count (`fft_cycle_count`). (2%)
    4. The CPU time (`fft_cpu_time`). (4%)
    5. Is this program CPU bound or Memory bound program? (4%)
        - The answer is generated automatically based on your provided data for `fft_ratio` .

- The partial C code for the exercise1.c is as follow.

```
#include <stdio.h>
#include <stdint.h>
```

```c
#include <stdlib.h>
#include <math.h>
#include "fft.h"
#include "macro_define.h"

typedef struct {
    float Re;
    float Im;
} Complex;


Complex complex_add(Complex a, Complex b)
{
    Complex result;
    asm volatile(
        #include "complex_add.c"
        :[C_Re] "=f"(result.Re), [C_Im] "=f"(result.Im), [fadd_cnt] "+r"(fadd_cn
        :[A_Re] "f"(a.Re), [B_Re] "f"(b.Re), [A_Im] "f"(a.Im), [B_Im] "f"(b.Im))
    return result;
}

Complex complex_sub(Complex a, Complex b)
{
    Complex result;
    asm volatile(
        #include "complex_sub.c"
        :[C_Re] "=f"(result.Re), [C_Im] "=f"(result.Im), [fsub_cnt] "+r"(fsub_cn
        :[A_Re] "f"(a.Re), [B_Re] "f"(b.Re), [A_Im] "f"(a.Im), [B_Im] "f"(b.Im))
    return result;
}

Complex complex_mul(Complex a, Complex b)
{
    Complex result;
    asm volatile(
        #include "complex_mul.c"
        : [C_Re] "=f"(result.Re), [C_Im] "=f"(result.Im), [fmul_cnt] "+r"(fmul_c
        : [A_Re] "f"(a.Re), [B_Re] "f"(b.Re),
        [A_Im] "f"(a.Im), [B_Im] "f"(b.Im)
        : "f1", "f2", "f3", "f4"
    );
    return result;
}

uint32_t Log2(uint32_t N)
{
    uint32_t log = 0;
    asm volatile(
        #include "log2.c"
        : [log] "+r"(log), [N] "+r"(N), [add_cnt] "+r"(add_cnt), [others_cnt] "+
        :
        : "x0", "t0", "t1"
    );
    return log;
}

float PI(void)
{
```

```
    asm volatile(
        #include "pi.c"
        : [add_cnt] "+r"(add_cnt), [fadd_cnt] "+r"(fadd_cnt), [sub_cnt] "+r"(sub_
        : [N] "r"(iter)
        : "f1", "f2", "t1", "t2", "t3", "t4"
    );
    pi = 4 * pi;
    return pi;
}

uint32_t bit_reverse(uint32_t b, uint32_t m)
{
    asm volatile (
        #include "bit_reverse.c"
        : [b] "+r"(b), [others_cnt] "+r"(others_cnt), [lw_cnt] "+r"(lw_cnt)
        : [temp] "r"(32 - m)
        : "t0", "t1", "t2"
    );
    return b;
}

void fft(Complex *x, uint32_t N)
{
    uint32_t k = N, n;
    float pi = PI();
    float thetaT = pi / N;
    // float thetaT = PI / N;
    Complex phiT = {cos(thetaT), -sin(thetaT)};
    Complex T, temp;

    while (k > 1) {
        n = k;
        k >>= 1;
        phiT = complex_mul(phiT, phiT);
        T.Re = 1.0;
        T.Im = 0.0;

        for (uint32_t l = 0; l < k; l++)
        {
            for (uint32_t a = l; a < N; a += n)
            {
                uint32_t b = a + k;
                temp = complex_sub(x[a], x[b]);
                x[a] = complex_add(x[a], x[b]);
                x[b] = complex_mul(temp, T);
            }
            T = complex_mul(T, phiT);
        }
    }

    // Bit-reverse
    uint32_t m = Log2(N);
    for (uint32_t a = 0; a < N; a++)
    {
        uint32_t b = a;
        b = bit_reverse(b, m);
        if (b > a) {
            temp = x[a];
```

```
            x[a] = x[b];
            x[b] = temp;
        }
    }
}
```

Exercise 2-1. Array Multiplication without V Extension (18%)

In this exercise, you need to perform element-wise multiplication of two arrays, using the **RV64G** ISA. Besides, you should do the same as exercise 1 *FFT* calculation to collect performance data and derive related performance statistics.

For example, we have two arrays: `h[4] = {0.1, 0.2, 0.3, 0.4}` and `x[4] = {0.2, 0.3, 0.4, 0.5}`, then the result of the array multiplication looks like this: `y[4] = {0.02, 0.06, 0.12, 0.20}`.

- As shown in the function `arraymul_baseline()`, you are responsible for writing the assembly for the for-loop code: `for (...) y[i] = h[i] * x[i] + id;`.
  - **NOTE:** You should put your assembly code within the `arraymul_baseline.c` file, as indicated in `asm volatile( #include "arraymul_baseline.c" ...);` within the `arraymul_baseline()` function in `exercise2_1.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 perform a larger size of array multiplication, but **it must be the power of 2 and 2 < `arr_size` <= 128**.
    - You need to change `student_id` based on your student id. The details of how to set up the `student_id` is elaborated in the table below. You will obtain wrong results if you do not modify `student_id` to set up `id` correctly. (This `id` is used in `for (...) y[i] = h[i] * x[i] + id;`)
  - **NOTE:** Please do not modify the rest of the program.

```
void arraymul_baseline(){
    float *p_h = h;
    float *p_x = x;
    float *p_y = y;
    float id = student_id;// id = your_student_id % 100;
    int arr_length = arr_size;
    /* 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), [mul
```

```
    : [id] "f"(id)
    : "f0", "f1"

    );
    ...
```

- Input
  - This exercise takes input from `arraymul_input.txt`.
  - The input values for this exercise consist of floating-point numbers accurate to six decimal places, with values ranging from `0.0` to `100.0`.

```
26.113884 29.327766 12.934367 61.528931 24.393055 36.206884 23.293595 21.1805
```

  - This exercise reads two arrays, each with `arr_size` elements, from `arraymul_input.txt`. Assume `arr_size = 4`, then `h[]` and `x[]` will be:

```
h[4] = {26.113884, 29.327766, 12.934367, 61.528931}
x[4] = {24.393055, 36.206884, 23.293595, 21.180578}
```

  - The value of `arr_size` ranges from 2 to 128, and it should be a power of 2.

- Output
  - An example output when the `student_id` sets to zero.

```
===== Question 2-1 =====
array size = 4
student id = 0
output:  636.997437  1061.866943  301.287903  1303.218384

add counter used: 16
sub counter used: 0
mul counter used: 0
...
```

- The performance probes should be inserted into your code to collect the data for performance counters. The Scoring Criteria section contains a complete list for these counters.
  - You can also refer to the table in *1. Performance Modeling*.
  - The method to implement performance probes is the same as the one of exercise 1.

- You also need to **compute** the total cycle count (`arraymul_baseline_cycle_count`), the CPU execution time (`arraymul_baseline_cpu_time`) and CPU instruction/Memory instruction ratio (`arraymul_baseline_ratio`) for the program.

- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`arr_size`	Size of the array
`h[ ]`	Input array 1 in `arraymul.h`
`x[ ]`	Input array 2 in `arraymul.h`
`y[ ]`	Output array in `arraymul.h`
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz **Do Not Modified**
`arraymul_baseline_cycle_count`	The total clock cycle in `arraymul_baseline.c` , you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_baseline_cycle_count`
`arraymul_baseline_cpu_time`	The CPU time in `arraymul_baseline.c` , you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_baseline_cpu_time`
`arraymul_baseline_ratio`	The ratio used to determine whether the program is a CPU bound or Memory bound program. You should follow the high-level concept of the formula provided above (2. Performance Characterization) to implement the `macro_calc_arraymul_baseline_ratio` in `macro_define.h`
`student_id`	Defined in `arraymul.h` `student_id` = your_student_id % 100 i.g. F12345678: `student_id` = 12345678 % 100 = 78

- The files you will modify in this exercise:
  - `arraymul_baseline.c`
  - `macro_define.h`

- **Scoring Criteria**: Your obtained scores of this exercise is detemined by the correctness of your reported performance data.
  - **NOTE:** When judging the exercise, we will change the array size and use hidden test cases to verify your code.

  1. The computation result for the array multiplication. (2%)
     - **NOTE:** If the result of the y array is incorrect, you won't get the scores below.
  2. The values of the counters. (13%)
     - `add_cnt` (1%)
     - `sub_cnt` (1%)
     - `mul_cnt` (1%)
     - `div_cnt` (1%)
     - `lw_cnt` (1%)
     - `sw_cnt` (1%)
     - `fadd_cnt` (1%)
     - `fsub_cnt` (1%)
     - `fmul_cnt` (1%)
     - `fdiv_cnt` (1%)
     - `flw_cnt` (1%)
     - `fsw_cnt` (1%)
     - `others_cnt` (1%)
  3. The total cycle count (`arraymul_baseline_cycle_count`). (1%)
  4. The CPU time (`arraymul_baseline_cpu_time`). (1%)
  5. Choose any `arr_size` but **it must be the power of 2 and 2 < `arr_size` <= 128** and answer the question: Is this program a CPU bound or Memory bound program? (1%)
     - In your submitted code, you should choose a value for `arr_size`.
     - The result for the array multiplication is generated automatically based on your provided data for `arraymul_baseline_ratio` and the chosen `arr_size`.

- The arraymul_baseline() function in exercise2_1.c is as follow.

```c
void arraymul_baseline(){
    float *p_h = h;
    float *p_x = x;
    float *p_y = y;
    float id = student_id;// id = your_student_id % 100;
    int arr_length = arr_size;
    /* 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), [mul_
    : [id] "f"(id)
    : "f0", "f1"

    );

    printf("output: ");
    for (int i = 0; i < arr_size; i++){
        printf(" %.6f ", y[i]);
    }

    printf("\n\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("fadd counter used: %d\n", fadd_cnt);
    printf("fsub counter used: %d\n", fsub_cnt);
    printf("fmul counter used: %d\n", fmul_cnt);
    printf("fdiv counter used: %d\n", fdiv_cnt);
    printf("flw counter used: %d\n", flw_cnt);
    printf("fsw counter used: %d\n", fsw_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_cy

    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);
}
```

Exercise 2-2. Array Multiplication with V Extension (22%)

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.

The operation is the same as exercise 2-1, we perform element-wise multiplication of two arrays. Assume we have two arrays: `h[4] = {0.1, 0.2, 0.3, 0.4}` and `x[4] = {0.2, 0.3, 0.4, 0.5}`, then the result of the array multiplication looks like this: `y[4] = {0.02, 0.06, 0.12, 0.20}`.

- As shown in the function `improved_version()`, you are responsible for writing the assembly for the for-loop in C: `for (...) y[i] = h[i] * x[i] + id;`.
  - **NOTE:** You should put your assembly code within the `arraymul_improved.c` file, as indicated in `asm volatile( #include "arraymul_improved.c" ...);` within the `improved_version()` 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=128, elen=32**). The vectorized version would improve the execution efficiency, thanks to the parallel computations done in the vector computation engine.
    - The counters used for the instructions for the V extension are defined in the table of 1. Performance Modeling.
  - 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 perform a larger size of array multiplication, but **it must be the power of 2 and 2 < `arr_size` <= 128**.
    - You need to change `student_id` based on your student id. The details of how to set up the `student_id` is elaborated in the table below. You will obtain wrong results if you do not modify `student_id` to set up `id` correctly. (This `id` is used in `for (...) y[i] = h[i] * x[i] + id;`).
  - **NOTE:** Please do not modify the rest of the program.

```
void improved_version(){
    float *p_h = h;
    float *p_x = x;
    float *p_y = y;
    float 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;
    }
    */
    int arr_length = arr_size;
    asm volatile(
        #include "arraymul_improved.c"
        : [h] "+r"(p_h), [x] "+r"(p_x), [y] "+r"(p_y), [add_cnt] "+r"(add_cnt), [
        : [id] "f"(id)
        : "t0", "v0", "v1", "v2"
    );
     ...
```

- Input
  - This exercise takes input from the file `arraymul_input.txt`.
  - The input values for this exercise consist of floating-point numbers accurate to six decimal places, with values ranging from `0.0` to `100.0`.

```
26.113884 29.327766 12.934367 61.528931 24.393055 36.206884 23.293595 21.1805
```

  - This exercise reads two arrays, each with `arr_size` elements, from `arraymul_input.txt`. Assume `arr_size = 4`, then `h[]` and `x[]` will be:

```
h[4] = {26.113884, 29.327766, 12.934367, 61.528931}
x[4] = {24.393055, 36.206884, 23.293595, 21.180578}
```

  - The value of `arr_size` ranges from 2 to 128, and it should be a power of 2.

- Output
  - An example output when the `student_id` sets to zero.

```
===== Question 2-2 =====
array size = 4
student id = 0
output:  636.997437  1061.866943  301.287903  1303.218384

add counter used: 3
sub counter used: 1
mul counter used: 0
...
```

- The performance probes should be inserted into your code to collect the performance data. A complete list of the to-be-collected performance counters is listed in Scoring Criteria below.
    - You can also refer to the table in *1. Performance Modeling*.
    - The method to implement performance probes is the same as the one of exercise 1.

- You also need to **compute** the total cycle count (`arraymul_vector_cycle_count`), the CPU execution time (`arraymul_vector_cpu_time`).

- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`h[ ]`	Input array 1 in `arraymul.h`
`x[ ]`	Input array 2 in `arraymul.h`
`y[ ]`	Output array in `arraymul.h`
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz **Do Not Modified**
`arr_size`	Size of the arrays used in this exercise
`arraymul_vector_cycle_count`	The total clock cycle in `arraymul_improved.c` , you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_vector_cycle_count`
`arraymul_vector_cpu_time`	The CPU time in `arraymul_improved.c` , you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_vector_cpu_time`
`student_id`	`student_id` = your_student_id % 100 i.g. F12345678: `student_id` = 12345678 % 100 = 78

- The files you will modify in this exercise:
    - `arraymul_improved.c`

    - `macro_define.h`

- **Scoring Criteria**: Your obtained scores of this exercise is detemined by the correctness of your reported performance data.

    - **NOTE:** When judging the exercise, we will change the array size and use hidden test cases to verify your code.

    1. The computation result. (2%)
        - **Note:** If the result is incorrect, you won't get the scores below.
    2. The values of the counters. (13%)
        - `add_cnt` (1%)
        - `sub_cnt` (1%)
        - `mul_cnt` (1%)
        - `div_cnt` (1%)
        - `lw_cnt` (1%)
        - `sw_cnt` (1%)
        - `fadd_cnt` (1%)
        - `fsub_cnt` (1%)
        - `fmul_cnt` (1%)
        - `fdiv_cnt` (1%)
        - `flw_cnt` (1%)
        - `fsw_cnt` (1%)
        - `others_cnt` (1%)
    3. The total cycle count (`arraymul_vector_cycle_count`). (1%)
    4. The CPU time (`arraymul_vector_cpu_time`). (1%)
    5. Achieved speedup. (5%)
        - You should choose an `arr_size` that **must match with the `arr_size` used in exercise 2-1** to calculate the speedup achieved by the vectorized version over the serial version.
        - The speedup will be calculated automatically if you provide the correct data above (total cycle count and the CPU time).
        - If achieved speedup < 2, get 0%
        - If achieved speedup => 2, get 5%

- The improved_version() function in exercise2_2.c is as follow.

```
void improved_version(){
    float *p_h = h;
    float *p_x = x;
    float *p_y = y;
    float 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;
    }
    */
    int arr_length = arr_size;
    asm volatile(
        #include "arraymul_improved.c"
        : [h] "+r"(p_h), [x] "+r"(p_x), [y] "+r"(p_y), [add_cnt] "+r"(add_cnt),
        : [id] "f"(id)
        : "t0", "v0", "v1", "v2"
    );

    for (int i = 0; i < arr_size; i++){
        printf(" %.6f ", y[i]);
    }

    printf("\n\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("fadd counter used: %d\n", fadd_cnt);
    printf("fsub counter used: %d\n", fsub_cnt);
    printf("fmul counter used: %d\n", fmul_cnt);
    printf("fdiv counter used: %d\n", fdiv_cnt);
    printf("flw counter used: %d\n", flw_cnt);
    printf("fsw counter used: %d\n", fsw_cnt);
    printf("others counter used: %d\n", others_cnt);
    macro_arraymul_vector_cycle_count
    printf("The total cycle count in this program: %.0f\n", arraymul_vector_cycle


    macro_arraymul_vector_cpu_time
    printf("CPU time = %f us\n", arraymul_vector_cpu_time);
    FILE *fp_1;
    fp_1 = fopen("arraymul_vector_cpu_time.txt", "w");
    fprintf(fp_1, "%f", arraymul_vector_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 / arraymul_vector_cpu_time;
    printf("V extension ISA faster %f times than baseline ISA\n", speedup);
}
```

Exercise 3-1. Single Floating-point Multiplication (16%)

In this exercise, you need to perform the multiplication of two arrays with **single floating-point (float)** data, using RISC-V assembly codes with **RV64G** ISA. Besides, you should do the same as previous exercises to collect performance data and derive related performance statistics.

This exercise performs element-wise multiplication followed by a complete product reduction (multiplying all the element-wise results together). It will read two arrays from `arraymul_input.txt`. Assume the two arrays are: `h[4] = {0.1, 0.2, 0.3, 0.4}` and `x[4] = {0.2, 0.3, 0.4, 0.5}`, then the result looks like this: `result = 0.0000288`.

- As shown in the function `arraymul_float()`, you are responsible for writing assembly code to perform the multiplication on both arrays and produce the final result. The execution will be done by a for loop like: `for(...) result = result * h[i] * x[i];`.
  - **NOTE:** You should put your assembly code within the `arraymul_float.c` file, as indicated in `asm volatile( #include "arraymul_float.c" ...);` within the `arraymul_float()` function in `exercise3_1.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`, but **it must be the power of 2 and 2 < `arr_size` <= 16**.
  - **NOTE:** Please do not modify the rest of the program.

```
void arraymul_float(){
    float *p_h = h;
    float *p_x = x;
    int arr_length = arr_size;
    /* original C code
    for (int i = 0; i < arr_size; i++){
        single_floating_result = single_floating_result * p_h[i] * p_x[i];
    }
    */
    asm volatile(
        #include "arraymul_float.c"
        : [h] "+r"(p_h), [x] "+r"(p_x), [result] "+f"(single_floating_result), [
        :
        : "t0", "f0", "f1", "f2", "f3"
    );
    ...
```

- Input
  - This exercise reads input from `arraymul_input2.txt`.
  - The input values for this exercise consist of floating-point numbers accurate to six decimal places, with values ranging from `0.0` to `10.0`.

```
7.22 1.0 5.08 7.3 8.48 7.58 3.61 6.74 1.83 9.82 1.72 7.49 0.03 6.13 8.96 ...
```

  - This exercise reads two arrays, each with `arr_size` elements, from `arraymul_input.txt`. Assume `arr_size = 4`, then `h[]` and `x[]` will be:

```
h[4] = {7.22 1.0 5.08 7.3}
x[4] = {8.48 7.58 3.61 6.74}
```

  - The value of `arr_size` ranges from 2 to 16, and it should be a power of 2.

- Output
  - An example output when the `arr_size` sets to four.

```
===== Question 3-1 =====
array size = 4
output:  418750.968750

add counter used: 12
sub counter used: 0
mul counter used: 0
...
```

- The performance probes should be inserted into your code to collect the performance data. A complete list of the to-be-collected performance counters is listed in Scoring Criteria below.
  - You can also refer to the table in *1. Performance Modeling*.
  - The method to implement performance probes is the same as the one of exercise 1.

- You also need to **compute** the total cycle count (`arraymul_baseline_cycle_count`) and the CPU execution time (`arraymul_baseline_cpu_time`).

- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`arr_size`	Size of the array
`h[ ]`	Input array 1 in `arraymul.h`
`single_floating_result`	Output variable in `arraymul.h` which is initialized to `1.0`

`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz **Do Not Modified**
`arraymul_baseline_cycle_count`	The total clock cycle in `arraymul_float.c`, you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_baseline_cycle_count`
`arraymul_baseline_cpu_time`	The CPU time in `arraymul_float.c`, you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_baseline_cpu_time`

- The files you will modify in this exercise:
  - `arraymul_float.c`
  - `macro_define.h`

- **Scoring Criteria**: Your obtained scores of this exercise is detemined by the correctness of your reported performance data.
  - **NOTE:** When judging the exercise, we will change the array size and use hidden test cases to verify your code.

  1. The computation result for the multiplications on the two given arrays. (1%)
     - **Note:** If the `result` is incorrect, you won't get the scores below.
  2. The values of the counters. (13%)
     - `add_cnt` (1%)
     - `sub_cnt` (1%)
     - `mul_cnt` (1%)
     - `div_cnt` (1%)
     - `lw_cnt` (1%)
     - `sw_cnt` (1%)
     - `fadd_cnt` (1%)
     - `fsub_cnt` (1%)
     - `fmul_cnt` (1%)
     - `fdiv_cnt` (1%)
     - `flw_cnt` (1%)
     - `fsw_cnt` (1%)

    - `others_cnt` (1%)
3. The total cycle count (`arraymul_baseline_cycle_count`). (1%)
4. The CPU time (`arraymul_baseline_cpu_time`). (1%)

Exercise 3-2. Double Floating-point Multiplication (24%)

You need to re-write the assembly code which you build in exercise 3-1, using **RV64G** ISA. Different from the previous exercise (using *float* data type), in this exercise, two arrays use the **double floating-point (double)** data type to store the input data. After re-writing the program, you should collect the performance data and derive related performance statistics as did in the previous exercise.

- As shown in the `arraymul_double()` function, you are responsible for writing the assembly to multiply all the elements in both arrays together to produce the final result. The execution will be done by a for loop like: `for(...) result = result * h[i] * x[i];` .
    - **NOTE:** You should put your assembly code within the `arraymul_double.c` file, as indicated in `asm volatile( #include "arraymul_double.c" ...);` within the `arraymul_double()` function in `exercise3_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` , but **it must be the power of 2 and 2 < `arr_size` <= 16**.
    - **NOTE:** Please do not modify the rest of the program.

```
void arraymul_double(){
    double *p_h = u;
    double *p_x = v;
    int arr_length = arr_size;
    /* original C code
    for (int i = 0; i < arr_size; i++){
        double_floating_result = double_floating_result * p_h[i] * p_x[i];
    }
    */
    asm volatile(
        #include "arraymul_double.c"
        : [h] "+r"(p_h), [x] "+r"(p_x), [result] "+f"(double_floating_result), [a
        :
        : "t0", "f0", "f1", "f2", "f3"
);
}
```

- Input
    - This assignment reads data from `arraymul_input2.txt` .

- The input values for this exercise consist of floating-point numbers accurate to six decimal places, with values ranging from `0.0` to `10.0`.

```
7.22 1.0 5.08 7.3 8.48 7.58 3.61 6.74 1.83 9.82 1.72 7.49 0.03 6.13 8.96 ...
```

- This exercise reads two arrays, each with `arr_size` elements, from `arraymul_input.txt`. Assume `arr_size = 4`, then `h[]` and `x[]` will be:

```
h[4] = {7.22, 1.0, 5.08, 7.3}
x[4] = {8.48, 7.58, 3.61, 6.74}
```

- The value of `arr_size` ranges from 2 to 16, and it should be a power of 2.

- Output

  - An example output when the `arr_size` sets to four.

```
===== Question 3-2 =====
array size = 4
output:  418751.066664454585407

add counter used: 12
sub counter used: 0
mul counter used: 0
...
```

- The performance probes should be inserted into your code to collect the performance data. A complete list of the to-be-collected performance counters is listed in Scoring Criteria below.

  - You can refer to the table in *1. Performance Modeling* and the scoring criteria below.

- You also need to **compute** the total cycle count (`arraymul_double_cycle_count`) and the CPU execution time (`arraymul_double_cpu_time`).

- Variables/Constants defined in the header files used in this exercise.

Var./Cons. Name	Definition
`arr_size`	Size of the array
`h[ ]`	Input array 1 in `arraymul.h`
`x[ ]`	Input array 2 in `arraymul.h`

`double_floating_result`	Output array in `arraymul.h` which is initialized to `1.0`
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz **Do Not Modified**
`arraymul_double_cycle_count`	The total clock cycle in `arraymul_double.c`, you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_double_cycle_count`
`arraymul_double_cpu_time`	The CPU time in `arraymul_double.c`, you need to calculate and define the formula in `macro_define.h` by modifying `macro_arraymul_double_cpu_time`

- The files you will modify in this exercise:
  - `arraymul_double.c`
  - `macro_define.h`

- **Scoring Criteria**: Your obtained scores of this exercise is detemined by the correctness of your reported performance data.
  - **NOTE:** When judging the exercise, we will change the array size and use hidden test cases to verify your code.

  1. The computation result for the multiplications on the two arrays. (2%)
     - **Note:** If the `result` is incorrect, you won't get the scores below.
  2. The values of the counters. (13%)
     - `add_cnt` (1%)
     - `sub_cnt` (1%)
     - `mul_cnt` (1%)
     - `div_cnt` (1%)
     - `lw_cnt` (1%)
     - `sw_cnt` (1%)
     - `dadd_cnt` (1%)
     - `dsub_cnt` (1%)
     - `dmul_cnt` (1%)
     - `ddiv_cnt` (1%)

        - `dlw_cnt` (1%)
        - `dsw_cnt` (1%)
        - `others_cnt` (1%)
    3. The total cycle count (`arraymul_double_cycle_count`). (1%)
    4. The CPU time (`arraymul_double_cpu_time`). (1%)
    5. Slowdown of exercise 3-2 (`double`) compared to exercise 3-1 (`float`). (3%)
        - The slowdown will be calculated automatically if you provide the correct data above (total cycle count and the CPU time).
        - If Slowdown < 1, get 0%.
        - If Slowdown > 1, get 3%.
    6. Relative error of the obtained results in the exercise 3-2 (`double`) and exercise 3-1 (`float`) versions. (4%)
        - The error will be calculated automatically if your code perform correctly to produce the result.
        - If relative error == 0, get 0%.
        - Otherwise, get 4%.

    - **NOTE:** For *5. Slowdown* and *6. Relative error*, you should choose an `arr_size` that **must match with the `arr_size` used in exercise 3-1** to calculate the slowdown and error, caused by the *double* version over the *float* version.

5. About Hidden Test Cases

There are two hidden test cases for each exercise. For each exercise, the public test cases (e.g., `fft_input.txt` , `arraymul_input.txt` and `arraymul_input2.txt`) account for 60% of the total score, and **the hidden test cases account for 40%**.

For example, if your code passes Exercise 1 with the public test case `fft_input.txt` , you will earn 24 points. If it also passes the two hidden test cases, you will earn the full 40 points.

NOTE: You may modify `fft_input.txt` , `arraymul_input.txt` or `arraymul_input2.txt` to verify the correctness of your code for every exercise.

NOTE: The hidden test cases follow the same value range as specified in the input description of each exercise.

6. 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 with public test cases.

- Test your code in `exercise1.c` with public test cases

```
$ make test_exercise1
```

- Test your code in `exercise2_1.c` with public test cases
  - You should modify the `student_id` in `arraymul.h` to get the correct result.

```
$ make test_exercise2_1
Input the array size: # input the number of array size
```

- Test your code in `exercise2_2.c` with public test cases
  - You should modify the `student_id` in `arraymul.h` to get the correct result.

```
$ make test_exercise2_2 with public test cases
Input the array size: # input the number of array size,
                      # it must be the same as exercise2_1
```

- Test your code in `exercise3_1.c` with public test cases

```
$ make test_exercise3_1
Input the array size: # input the number of array size
```

- Test your code in `exercise3_2.c` with public test cases

```
$ make test_exercise3_2
Input the array size: # input the number of array size,
                      # it must be the same as exercise3_1
```

> 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:***

```
$ make test_exercise1
---------------------Exercise1---------------------
X[0] = 4.000000 + 0.000000j
X[1] = 1.001207 + -2.414567j
X[2] = 0.000000 + 0.000000j
X[3] = 1.000914 + -0.413859j
```

```
X[4] = 0.000000 + 0.000000j
X[5] = 0.999793 + 0.414567j
X[6] = 0.000000 + 0.000000j
X[7] = 0.998086 + 2.413859j


add counter used: 2007
sub counter used: 0
mul counter used: 1000
div counter used: 0
fadd counter used: 546
fsub counter used: 546
fmul counter used: 88
fdiv counter used: 1000
lw counter used: 1066
sw counter used: 0
others counter used: 2708
The total cycle count in this program: 51453
CPU time: 19757952.0 us
Exercise Result:  This program is a CPU bound task.
------------------------result-------------------------
student_fft: V
student_add_cnt: V
student_sub_cnt: V
student_mul_cnt: V
student_div_cnt: V
student_fadd_cnt: V
student_fsub_cnt: V
student_fmul_cnt: V
student_fdiv_cnt: V
student_lw_cnt: V
student_sw_cnt: V
student_others_cnt: V
student_cycle_count: V
student_CPU_time: V
student_task_type: V
Obtained/Total scores: 24.00/24.00
```

- ***Error:***

```
$ make test_exercise1
---------------------Exercise1---------------------
X[0] = 4.000000 + 0.000000j
X[1] = 1.001207 + -2.414567j
X[2] = 0.000000 + 0.000000j
X[3] = 1.000914 + -0.413859j
X[4] = 0.000000 + 0.000000j
X[5] = 0.999793 + 0.414567j
X[6] = 0.000000 + 0.000000j
X[7] = 0.998086 + 2.413859j


add counter used: 2007
sub counter used: 0
mul counter used: 1000
div counter used: 0
fadd counter used: 534
```

```
fsub counter used: 546
fmul counter used: 88
fdiv counter used: 1000
lw counter used: 1066
sw counter used: 0
others counter used: 2708
The total cycle count in this program: 51393
CPU time: 19734912.0 us
Exercise Result:  This program is a CPU bound task.
------------------------result-------------------------
student_fft: V
student_add_cnt: V
student_sub_cnt: V
student_mul_cnt: V
student_div_cnt: V
student_fadd_cnt: X
student_fsub_cnt: V
student_fmul_cnt: V
student_fdiv_cnt: V
student_lw_cnt: V
student_sw_cnt: V
student_others_cnt: V
student_cycle_count: X
student_CPU_time: X
student_task_type: V
Obtained/Total scores: 19.20/24.00
```

7. Submission of Your Assigment

Your developed codes should be put into the folder: `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 `CO_StudentID` with your student ID number, e.g., F12345678_HW2.zip**

```
CO_StudentID_HW2.zip
├── README.md
├── macro_define.h
├── complex_add.c
├── complex_sub.c
├── complex_mul.c
├── bit_reverse.c
├── log2.c
├── pi.c
├── arraymul_baseline.c
├── arraymul_improved.c
```

```
├── arraymul_float.c
└── arraymul_double.c
```

- Do not submit any files that are not listed above.
- In addition to your code, you must also submit a file named `README.md`. This document should record your development process to prove that the submitted code is your own work. `README.md` is only accepted in **Markdown** or plain text format. You can use HackMD to edit your `README.md` file. `README.md` can be written in Chinese or English.
    - `README.md` is **mandatory**. Although it does not contribute to your score, failing to submit `README.md` will result in a score of zero.
    - You can write anything in `README.md`, and there is no length requirement, as long as it proves that you completed the assignment yourself. If you are unsure what to write, consider documenting your development process, algorithm explanations, debugging steps, and testing process.
    - A plagiarism checking process will be performed on your submitted code. A high similarity score will result in a score of zero.

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

!!! A 30% penalty will be applied for late submissions within seven days (from 00:00, May 15 to 23:59, May 21, 2025) after the deadline. !!!

!!! Do not modify the `Makefile`, as this may cause the judge program to fail, resulting in a score of zero. !!!

8. References

- Cooley-Tukey DIF FFT Introduction
- Cooley-Tukey DIT FFT Introduction
- RISC-V V Extension
- RISC-V ISA List
- Extending RISC-V
- RISC-V-Spec