Computer Organization 2025 Programming Assignment I

Computer Organization 2025 Programming Assignment I

Due Date: 23:59, April 10, 2025

Overview

This assignment aims to get familiar with the hardware/software interface: instruction set architecture (ISA), as well as the development environment and tools (e.g., compiler and RISC-V simulator) for RISC-V programming. In particular, this programming assignment is to **write inline assembly code** and run the developed code on a RISC-V ISA Simulator, *Spike*, to obtain the simulation results.

The assignments of this course are based on the RISC-V Spike simulator running on Ubuntu Linux 22.04. As our assignments are built upon open source projects, there will be unexpected compatibility issues if you choose a different development environment.

This assignment consists of three exercises, with a total score of **100 points**. You will be given some test cases, and you must ensure that your code passes all these test cases, referred to as **public test cases**. Please note that there will be **hidden test cases** as well. Try to make your program handle special inputs properly. For details on how points are assigned, see **Scoring Criteria**.

Suggested Workflow

1. Read this document carefully to understand the assignment requirements.
2. Complete the inline assembly code for Exercise 1, 2, and 3.
3. Test your code using `local-judge` to check all public test cases. Think about possible hidden test cases and consider common edge cases. Manually test any additional cases you think are important. For more details, see **Test Your Assignment**.
4. After completing Exercise 1, 2, and 3, follow the instructions in **Submission of Your Assignment** to submit your work.

Prerequisite

Environment Setup

You have to follow the procedures listed in the **HW0** document to set up your development environment.

1. Assignment

In this assignment, you will complete three exercises by implementing **inline assembly** in a C program. The goal is to practice low-level programming and understand how assembly interacts with C.

- **Exercise 1 (Array Bubble Sort, 20%)**: Implement the **bubble sort** algorithm using inline assembly to sort an array of integers.
- **Exercise 2 (Array Search, 40%)**: Implement the **search** algorithm using inline assembly to find the index of a target value in an array.
- **Exercise 3 (Linked-List Merge Sort, 40%)**: Implement **merge sort** for a linked list using inline assembly. You will need to split the list, merge sorted lists, and move between nodes.

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**.
- The following C code contains only essential parts for the explanation of this assignment. Please use the full version downloaded from NCKU Moodle as the primary reference.
- The following are the details of both public and hidden test cases.
  - All input values are positive integers of type `int` (32-bit), within the range of C language integers.
  - The length of the input array or list ranges from `0` to `10,000`.

Exercise 1. Array Bubble Sort (20%)

Please complete the array bubble sort algorithm and sort the array. You need to add your inline assembly code in the C file.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }
```

```
    FILE *input = fopen(argv[1], "r");
    if (!input) {
        fprintf(stderr, "Error opening file: %s\n", argv[1]);
        return 1;
    }
    int arr_size;
    fscanf(input, "%d", &arr_size);
    int arr[arr_size];

    // Read integers from input file into the array
    for (int i = 0; i < arr_size; i++) {
        int data;
        fscanf(input, "%d", &data);
        arr[i] = data;
    }
    fclose(input);

    int *p_a = &arr[0];

    for (int i = 0; i < arr_size - 1; i++) {
        for (int j = 0; j < arr_size - i - 1; j++) {
            asm volatile(
                // Your code
                "");
        }
    }
    p_a = &arr[0];
    for (int i = 0; i < arr_size; i++)
        printf("%d ", *p_a++);
    printf("\n");
    return 0;
}
```

The following is an example of a test case.

Input:

```
10
9 5 8 3 4 10 1 6 2 7
```

Expect output:

```
1 2 3 4 5 6 7 8 9 10
```

Input Format:

- The first line represents the array size.
- The second line contains the array elements, and the number of elements always matches the number in the first line.

Expected Output Format:

- A single line contains the sorted array.

Program Execution Steps:

- Read the input: The first line (`10`) is the array size, and the second line (`9 5 8 3 4 10 1 6 2 7`) is the array.
- Execute the bubble sort algorithm using inline assembly to sort the array.
- Print the sorted array: `1 2 3 4 5 6 7 8 9 10` and end the program.

Exercise 2. Array Search (40%)

Please finish the following code. You need to add your inline assembly code in the C file.

```c
#include <stdio.h>

int arraySearch(int *p_a, int arr_size, int target)
{
    int result = -1;

    asm volatile(
        // Your code
        "");

    return result;
}

// Main function to test the implementation
int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *input = fopen(argv[1], "r");
    if (!input) {
        fprintf(stderr, "Error opening file: %s\n", argv[1]);
        return 1;
    }
    int arr_size;
    fscanf(input, "%d", &arr_size);
    int arr[arr_size];

    // Read integers from input file into the array
    for (int i = 0; i < arr_size; i++) {
        int data;
        fscanf(input, "%d", &data);
        arr[i] = data;
    }
    int target;
    fscanf(input, "%d", &target);
    fclose(input);

    int *p_a = &arr[0];
```

```
    int index = arraySearch(p_a, arr_size, target);

    // Print the result
    printf("%d ", index);
    printf("\n");

    return 0;
}
```

The following is an example of a test case.

Input:

```
10
6 5 7 3 10 9 1 4 2 8
8
```

Expect output:

```
9
```

Input Format:

- The first line represents the array size.
- The second line contains the array elements, and the number of elements always matches the number in the first line.
- The third line contains the target value to search for in the array.

Expected Output Format:

- A single integer representing the index of the target value in the array, or `-1` if the target is not found.

Program Execution Steps:

- Read the input: The first line (`10`) is the array size, the second line (`6 5 7 3 10 9 1 4 2 8`) is the array, and the third line (`8`) is the target value.
- Execute the search algorithm using inline assembly to find `8` in the array.
- Print `9` as the index of `8` in the array and end the program.

Exercise 3. Linked-List Merge Sort (40%)

Please finish the following code. You need to add your inline assembly code in the C file.

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

```c
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Split the linked list into two parts
void splitList(Node *head, Node **firstHalf, Node **secondHalf)
{
    asm volatile(
        /*
        Block A (splitList), which splits the linked list into two halves
        */
        "");
}

// Merge two sorted linked lists
Node *mergeSortedLists(Node *a, Node *b)
{
    Node *result = NULL;
    Node *tail = NULL;

    asm volatile(
        /*
        Block B (mergeSortedList), which merges two sorted lists into one
        */
        "");

    return result;
}

// Merge Sort function for linked list
Node *mergeSort(Node *head)
{
    if (!head || !head->next)
        return head; // Return directly if there is only one node

    Node *firstHalf, *secondHalf;
    splitList(head, &firstHalf,
              &secondHalf); // Split the list into two sublists

    firstHalf = mergeSort(firstHalf);   // Recursively sort the left half
    secondHalf = mergeSort(secondHalf); // Recursively sort the right half

    return mergeSortedLists(firstHalf, secondHalf); // Merge the sorted sublists
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *input = fopen(argv[1], "r");
    if (!input) {
        fprintf(stderr, "Error opening file: %s\n", argv[1]);
        return 1;
```

```
    }
    int list_size;
    fscanf(input, "%d", &list_size);
    Node *head = (list_size > 0) ? (Node *)malloc(sizeof(Node)) : NULL;
    Node *cur = head;
    for (int i = 0; i < list_size; i++) {
        fscanf(input, "%d", &(cur->data));
        if (i + 1 < list_size)
            cur->next = (Node *)malloc(sizeof(Node));
        cur = cur->next;
    }
    fclose(input);

    // Linked list sort
    head = mergeSort(head);

    cur = head;
    while (cur) {
        printf("%d ", cur->data);
        asm volatile(
            /*
            Block C (Move to the next node), which updates the pointer to
            traverse the linked list
            */
            "");
    }
    printf("\n");
    return 0;
}
```

The following is an example of a test case.

Input:

```
10
9 5 8 3 4 10 1 6 2 7
```

Expect output:

```
1 2 3 4 5 6 7 8 9 10
```

Input Format:

- The first line represents the number of elements in the linked list.
- The second line contains the linked list elements, and the number of elements always matches the number in the first line.

Expected Output Format:

- A single line contains the sorted linked list.

Program Execution Steps:

- Read the input: The first line (`10`) is the number of elements, and the second line (`9 5 8 3 4 10 1 6 2 7`) represents the linked list.
- Execute the merge sort algorithm using inline assembly to sort the linked list.
- Print the sorted linked list: `1 2 3 4 5 6 7 8 9 10` and end the program.

Hint:

- There are three blocks where you need to implement inline assembly: **Block A (splitList)**, which splits the linked list into two halves; **Block B (mergeSortedList)**, which merges two sorted lists into one; and **Block C (Move to the next node)**, which updates the pointer to traverse the linked list.
- It is recommended to start with Block C first, because you need to understand how to move to the next node. This is related to **C memory alignment** and the fact that the simulator uses **RISC-V 64-bit** architecture.
- The comments in the code are just for reference. You do not have to follow the exact program logic. You can implement your own approach.

2. Test Your Assignment

We use local-judge to evaluate your program. For installation instructions, please refer to the **HW0** document.

The source files of the aforementioned exercises are included in the zip file `CO2025HW1.zip` that can be downloaded from the course website on NCKU Moodle.
Please unzip and move these files into the project folder (e.g., `$HOME/riscv`). The file `judge*.conf` should be placed in the same folder as your code (e.g., `$HOME/riscv/CO_StudentID_HW1`). The directory tree of the source code for this assignment is shown below.

```
.
├── CO_StudentID_HW1
│   ├── Makefile
│   ├── array_search.c
│   ├── array_sort.c
│   ├── judge1.conf
│   ├── judge2.conf
│   ├── judge3.conf
│   └── linked_list_sort.c
└── testcases
    ├── expected
    │   ├── 1_1.out
    │   ├── 1_2.out
    │   ├── 1_3.out
    │   ├── 1_4.out
    │   ├── 2_1.out
    │   ├── 2_2.out
    │   ├── 3_1.out
```

```
    │   └── 3_2.out
    └── input
        ├── 1_1.txt
        ├── 1_2.txt
        ├── 1_3.txt
        ├── 1_4.txt
        ├── 2_1.txt
        ├── 2_2.txt
        ├── 3_1.txt
        └── 3_2.txt
```

HW1 Folder Structure

`CO_StudentID_HW1/` : This directory contains **source code and configuration files** for HW1.

- `array_search.c`
  Implementation of **search on an array**.
- `array_sort.c`
  Implementation of **sorting an array**.
- `linked_list_sort.c`
  Implementation of **sorting a linked list**.
- `judge1.conf` , `judge2.conf` , and `judge3.conf` are configuration files for judging HW1 test cases: **array sort**, **array search**, and **linked list sort**, respectively.

`testcases/input/` : This folder contains **test input files**, the public test cases. Each file corresponds to a specific **test case**.

- `1_1.txt` , `1_2.txt` , `1_3.txt` , `1_4.txt`
  Input files for **array sorting (** `array_sort.c` **)**.
- `2_1.txt` , `2_2.txt`
  Input files for **array search (** `array_search.c` **)**.
- `3_1.txt` , `3_2.txt`
  Input files for **linked list sorting (** `linked_list_sort.c` **)**.

`testcases/expected/` : This folder contains **expected output files** for each test case.

- `1_1.out` , `1_2.out` , `1_3.out` , `1_4.out`
  Expected outputs for **array sorting**.
- `2_1.out` , `2_2.out`
  Expected outputs for **array search**.
- `3_1.out` , `3_2.out`
  Expected outputs for **linked list sorting**.

Use judge program to test your code

Now, you can use the judge program to get the score of your developed code with the following commands.

```
$ cd ~/riscv/CO_StudentID_HW1
$ make judge
riscv64-unknown-linux-gnu-gcc -static -o 1 array_sort.c
riscv64-unknown-linux-gnu-gcc -static -o 2 array_search.c
judge -c judge1.conf
local-judge: v2.7.2
=======+=================================================================================
Sample | Accept
=======+=================================================================================
   1_1 | ✘
=======+=================================================================================
   1_2 | ✘
=======+=================================================================================
   1_3 | ✔
=======+=================================================================================
   1_4 | ✔
=======+=================================================================================
Correct/Total problems:        2/4
Obtained/Total scores: 10/20

`judge -c judge2.conf
local-judge: v2.7.2
=======+=================================================================================
Sample | Accept
=======+=================================================================================
   2_1 | ✘
=======+=================================================================================
   2_2 | ✘
=======+=================================================================================
Correct/Total problems:        0/2
Obtained/Total scores: 0/20

`judge -c judge3.conf
local-judge: v2.7.2
=======+=================================================================================
Sample | Accept
=======+=================================================================================
   3_1 | ✘
=======+=================================================================================
   3_2 | ✘
=======+=================================================================================
Correct/Total problems:        0/2
Obtained/Total scores: 0/20
```

If you want to judge a specific exercise, you can use the command `judge -c [CONFIG]`. For example, you can check the result of the first exercise 1 via the command `judge -c judge1.conf`.

```
$ cd ~/riscv/CO_StudentID_HW1
$ judge -c judge1.conf
local-judge: v2.7.2
```

```
=======+=================================================================================
Sample | Accept
=======+=================================================================================
   1_1 | ✘
=======+=================================================================================
   1_2 | ✘
=======+=================================================================================
   1_3 | ✔
=======+=================================================================================
   1_4 | ✔
=======+=================================================================================
Correct/Total problems:        2/4
Obtained/Total scores: 10/20
```

If your code generates wrong output, you can use the parameter `-v 1` while running the judge command to check the differences between your output and the correct answer. The example message is shown as below.

```
Example:
standard answer is 74.
your assignment output is -1.
```

```
$ cd ~/riscv/CO_StudentID_HW1
$ judge -c judge2.conf -v 1
local-judge: v2.7.2
=======+=================================================================================
Sample | Accept
=======+=================================================================================
   2_1 | ✔
=======+=================================================================================
   2_2 | ✘
-------+---------------------------------------------------------------------------------
--- /home/CompOrg/projects/CompOrg2025_HW1/testcases/expected/2_2.out        2025-
+++ ../output/2_2local_1741138212.out  2025-03-05 01:30:12.811641395 +0000
@@ -1 +1 @@
-74
+-1

=======+=================================================================================
Correct/Total problems:        1/2
Obtained/Total scores: 10/20
```

Manually test possible hidden test cases

If you want to test a different test case, modify the files in `testcases/expected` and `testcases/input` according to the instructions for each exercise.

3. Submission of Your Assignment

- Compress your source files into a `zip` file.

- Submit your assignment to NCKU Moodle.
- The file organization of your zip file should be as follows.
  - **NOTE: Change all `CO_StudentID` to your student ID number, e.g., F12345678.zip.** The example zip file for the student with the student ID, F12345678, is shown below.

```
F12345678_HW1.zip/
└── F12345678_HW1/
    ├── README.md
    ├── Makefile
    ├── array_search.c
    ├── array_sort.c
    └── linked_list_sort.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, April 11 to 23:59, April 17, 2025) after the deadline. !!!

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

4. Scoring Criteria

- Exercises 1, 2, and 3 have a total of 100 points.
- Exercise 1 has 4 public test cases, each worth 5 points, for a total of 20 points.
- Exercise 2 has 2 public test cases and 2 hidden test cases, each worth 10 points, for a total of 40 points.

- Exercise 3 has 2 public test cases and 2 hidden test cases, each worth 10 points, for a total of 40 points.
- Public test cases account for 60 points, while hidden test cases account for 40 points.
- `README.md` is **mandatory**. Although it does not contribute to your score, failing to submit `README.md` will result in a score of **zero** for this assignment.
- A high similarity score between your code and someone else's will result in a score of zero for this assignment.

5. Reference

- GCC-Inline-Assembly-HOWTO
- Extended Asm - Assembler Instructions with C Expression Operands
- RISC-V Assembly Programmer's Manual
- The RISC-V Instruction Set Manual Version 2.2