Project 1

Overview

In Project 1, our objective is to familiarize you with some fundamental concepts within RISC-V and enable you to write basic programs using RISC-V assembly language. In this assignment, your first task is to implement some ALU (Arithmetic Logic Unit) functions that produces correct outputs based on the provided test inputs. Following that you are required to optimize a given RISC-V assembly program and only modify on restirced part.
Before you start working on this project, you need to:

- Install the docker environment (follow the **Tutorial1 slides**, which will be uploaded beforehand)
- Review the content of RISC-V on blackboard

Contact: This project is mainly modified and designed by USTF Wenjie Feng (122090112@link.cuhk.edu.cn). If you have any questions with regard to the design or purpose of this project, feel free to contact him and go to his office hours (Every Thursday 13:00-14:00 outside A Cup of Tea (一瓯茶) during Project 1. TA Wenhai Lai (wenhailai@link.cuhk.edu.cn) will also help with this project.

Task

Task 1: ALU Functions

In general, the ALU here is a rather simple Arithmetic Logic Unit. There is no need for instruction parsing. Instead, you are only required to implement various ALU functions corresponding to different instructions, such as add, sub and so on. In plain words, you just need to use assembly code to implement functions like add, sub and so on. It's like simulating RISC-V using RISC-V... ;)

1.1 ALU Functions

There are 2 parts for this task. First, you are required to implement the following instructions in you ALU. (They are "missing" instructions)

- not
- subi
- bg (branch greater than)

Second, you need to select one instruction from each case below:

- case1: add/sub

- case2: beq/bnq/slt
- case3: and/or/nor/xor
- case4: sll/sra/srl

Notice: the `add` and `sub` in RISC-V won't handle overflow due to its design, you shouldn't implement overflows in `add.s` or `sub.s`. Instead, the implementation of overflow will be handled as *extra credits* and you should do it in `add_extra.s` or `sub_extra.s`. Please make sure your implementations are different or there will be severe consequences for the suspicion of plagiarism.

You will be given 13 template files, where one (`required.s`) is for the first part and 12 other files named after their function names for you to slelect, like `add.s` . We seperate all the files so you can test your selected ones independently. The extra credits files are `add_extra.s` and `sub_extra.s` .
The inputs for functions (add, sub, and, nor, or, xor, slt) come in single forms: rs1, rs2, rd which specify the locations of registers of inputs and output. Then, the inputs of functions(bg, beq, bne) are rs1, rs2. The inputs of functions(subi, sll, srl, sra) are rs1, imm, rd, where imm means the immediate number we need. The inputs of function `not` is rs1, rd.
For example,

```
int ADD(int rs1, int rs2, int rd)
        ......
        return 0

int BEQ(int rs1, int rs2)
        result = ...
        ......
        return result
        # store the result in a0 since there is no return register
        # return 1 if branch; return 0 if not.

int SLL(int rs1, int shamt, int rd)
        ......
        return 0

int NOT(int rs1, int rd)
        ......
        return 0
```

(Code example above is just to show you the signature and output of different functions)

Notice: We define a0 to represent the value of the first input, a1 for the second input, and a2 to represent the value of the third input. You should do this in your

program and we will check it randomly. If you don't follow the pattern, you will miss 50% points related to the alu task.

1.2 Function example

This part will provide you with a template on how to write functions.

```
AND:
        lui a4,%hi(registers)
        addi a3,a4,%lo(registers)
        slli a0,a0,2
        slli a1,a1,2
        slli a2,a2,2
        add a0, a3, a0
        add a1, a3, a1
        add a2, a3, a2
        lw a0,0(a0)
        lw a1,0(a1)
        and a1,a0,a1
        sw a1,0(a2)
        li a0, 0
        jr ra
```

- Load the Base Address of the Registers:
  - `lui a4, %hi(registers)` : Load the high 20 bits of the address of registers into a4 .
  - `addi a3, a4, %lo(registers)` : Add the low 12 bits of the address of registers to a4 , storing the complete address in a3 . Now, a3 contains the base address of registers.
- Shift and Scale Indices (Assuming 4-Byte Elements):
  - `slli a0, a0, 2` : Scale the value in a0 (index for the first element) by 4 to account for 4-byte (32-bit) elements.
  - `slli a1, a1, 2` : Scale the value in a1 (index for the second element) by 4.
  - `slli a2, a2, 2` : Scale the value in a2 (index for storing the result) by 4.
- Calculate Addresses of Elements:
  - `add a0, a3, a0` : Add the scaled offset to the base address to get the actual address of the first element.
  - `add a1, a3, a1` : Calculate the address of the second element.
  - `add a2, a3, a2` : Calculate the address where the result will be stored.
- Load and Perform the Bitwise AND Operation:
  - `lw a0, 0(a0)` : Load the value of the first element into a0 .
  - `lw a1, 0(a1)` : Load the value of the second element into a1 .

    - `and a1, a0, a1` : Perform the bitwise AND operation between the two loaded values, storing the result in a1 .
- Store the Result:
    - `sw a1, 0(a2)` : Store the result of the AND operation into the third element of the registers (location calculated earlier).
- Return and End of Function:
    - `li a0, 0` : Set the return value to 0 (optional, depends on calling convention).
    - `jr ra` : Return from the subroutine.

1.3 compile and run file

```
riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 add.s -o add
./add    #if it does not work, use qemu-riscv32 add
```

1.4 Extra Credits 5'

The implementation of overflow in `add` or `sub`. Do the modification in `add_extra.s` or `sub_extra.s`. The structure will be different. Take `add` as an example,

```
int ADD(int rs1, int rs2, int rd)
        ......
        if overflow:
                return 1
        return 0
```

`sub` 's overall structure should be the same. As long as the return value is correct and the value in `rd` is the value that overflowed, you will pass the test.

Task 2: Optimize given RISC-V Program

This part aims to let you have some hands-on experience on assembly code programming. You are supposed to read through the whole assemblly code and understand what this program wants to do. Once you understand it, it will be very clear how you should improve the algorithm (actually very simple).

2.1 Requirement

You are provided with an template RISC-V assemble code file named `naive.s` . And there are 2 things you need to do. First, optimize the code to reduce its execution time. Second, expand the coverage of numbers by 10 times. For example, if originally the program is operating on an integer sequence from 1 to 1000, now you should make it operate on the sequence from 1 to 10000. For optimization, you can **can only modify** the function `judge`

part (between line 10 to line 46). For range expansion, there are at most 2 lines need changes. The modification room we left is more than enough, so don't be scared. Submission with modification outside the range will be **directly graded with 0**. So please pay attention.

> Hint: You can optimize the program by improving its algorithm's time complexity. One very possible (and easiest) way is to improve it from O(n^2) to O(n*squr(n)). And we will measure your optimization by comparing program execution time.

2.2 Compile and run file

To run the assembly code file, you must be equipped with `riscv64-gcc` toolchain. Inside a Linux environment, like docker we provided:

```
1. riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32  naive.s -o naive
# compile the program
2. ./naive     # run the program, it will output everything in terminal
```

2.3 Grading

We will grade your code based on its execution time. If you used our hint and have improved its algorithm correctly, the reduction will be significant.
You should first compare the optimzed code's execution time with the original one. Then, you shall expand the sequence coverage and also compare its performance before and after optimization. Also, submit your code on an expanded range version.
You can check the execution time yourself by typing `time ./naive`.
You can also use `./naive > output.txt` to store your output in a text file instead of printing all of them in terminal.
You can combine them by `time ./naive > output.txt`

Report

The report of this project should be no longer than 5 pages. Keep your words concise and clear. In your report, you should include:

1. Your big picture thoughts and ideas, showing us you really understand RISC-V assembly.
2. For Task1, state instructions you have chosen and explicitedly show us whether you are doing the *extra credits problem* or not. Show your implementation details. i.e. explain some special tricks used. Attach the screenshots of running results.
3. For Task2, state what the code is trying to do and how you improve it. Attach the screenshots of your code's execution time before and after optimization on original sequnce range and on expanded sequence range. But **submit your code on an expanded range version**.

Submission and Grading

1. Submission

You should put all of your source files (`naive.s` , `required.s` , 4 other assembly code files you choose, one extra credits .s file if you did it, report.pdf) in a folder and compress the folder in a .zip. Name it with your student ID, i.e. 122090xxx.zip. Submit it through BB. The ddl of this project is 2025/02/23. **We eliminate the Spring break, so no need to panic**. If you fail to submit your assignment by the deadline, 10 points will be deducted from your original mark for each day you are late, up to a maximum of 30 points. Assignments submitted late more than 3 days after the deadline will not be considered valid and will be marked as 0 points.

2. Grading Details

- Task1: ALU - 49% (7%*7)
- Task2: Optimize the assembly code - 41% (Sequence coverage expansion: 10%, Optimization: 31%) The coverage expansion points will only be granted if you optimized the code. Only if there is significant reduction on execution time, you will get full optimization points. Or you will only get partial points ranging from 30% to 60%.
- Extra credits: Overflow check - 5%
- Report - 10% (A single report without code or gibberish code will also not be graded)

3. Honesty

We take your honesty seriously. If you are caught copying others' code, you will get an automatic 0 in this project. Please write your own code.