Lab 1: Assembly Lab

Overview

Convolution is an important primitive widely used in many applications including computer vision (CV), natural language processing (NLP) and signal processing. One of the most popular algorithms is convolutional neural networks (CNNs). In this lab, we will implement in the RISC-V assembly language a simple convolution program that uses a Gaussian-like filter, and optimize its performance using a cycle-accurate CPU simulator.

Before starting this lab, you are supposed to be equipped with the following:

- Access to a Linux server (suggested approach, set at Lab 0)
- Basic Linux operation (refer to Linux cheat sheet)
- Basic C programming language
- Basic RISC-V assembly

After this lab, you will understand the following:

- Concepts of image processing, particularly smooth filtering
- Concepts of simulation, especially the use of Ripes, a RISC-V simulator
- Concepts of code optimization and simple performance analysis of your own program

During this lab, you are suggested to refer to the following materials:

- Linux cheat sheet

  https://cheatography.com/davechild/cheat-sheets/linux-command-line/

- Image convolution & Gaussian smooth filter

  https://en.wikipedia.org/wiki/Kernel_(image_processing)

  https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

- RISC-V ISA

  Quick reference card

  RISC-V reference book

  Specification, Volume 1

- Code optimization

  https://en.wikipedia.org/wiki/Loop_nest_optimization

  https://www.agner.org/optimize/optimizing_assembly.pdf

Provided Infrastructure

Inside this repository, we provide the following files:

```
+-- lab1-handout.pdf --- This document
+-- X11 Forwarding Turotial.pdf --- The tutorial of ssh with X11
forwarding
+-- run --- Our workspace folder
|   +-- gen_pixel.py --- Convert an image into pixel file
|   +-- parse_pixel.py --- Convert a pixel file into image
|   +-- lab1-riscv-starter-code.S --- Starter code for RISC-V assembly
|   +-- lab1-c-starter-code.c --- Starter code for C
|   +-- simple_add.c --- Sample C code
|   +-- simple_add.S --- Sample RISC-V assembly code
|   +-- simple_add_data.txt --- Data for sample RISC-V assembly code
+-- sample --- Origin image and expected blurred results
|   +-- turing.jpg --- The original image that you work on.
|   +-- turing_before.pixel --- The pixel file of turing.jpg.
|   +-- turing_processed.jpg --- The processed image.
|   +-- turing_processed.pixel --- The pixel file of the processed image.
+-- submit --- Your final works to submit
+-- riscv-card.pdf --- A reference card of RISC-V instruction set
+-- RISC-V-Reader-Chinese-v2p1.pdf --- A reference RISC-V manual in
Chinese.
```

Tasks

1 Setup Environment

1.0 X11 Forwarding Tutorial

Before you set up environment, please view the X11 Forwarding Tutorial first.

1.1 Upload Lab 1 Package

First, log in to your server and create the root folder for our labs:

```
$ mkdir ~/yao-archlab-f22 && cd "$_"
```

Upload the Lab 1 package from your local machine to the server:

- For MobaXterm users, use the sidebar button.
- For other users, use `scp` tool on your local host: `scp -P <machineID> lab1.zip <username>@101.6.96.191:/home/<username>/yao-archlab-f22`

```
$ unzip lab1.zip -d ~/yao-archlab-f22
```

1.2 Build Ripes

```
$ GIT_SSL_NO_VERIFY=1 git clone --recursive https://git.tsinghua.edu.cn/yao-archlab-f22/Ripes.git
```

Notice: You will be asked to input the username and password when cloning Ripes.

Username: yao-archlab-f22

Password: 7U6-zJHEn-_6sj8iiHc8

```
$ git -C ./Ripes/ checkout lab1-v1.1
$ cd ./Ripes
$ mkdir build && cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ../
$ make -j4
```

Note: these steps may not work at the first time, especially on other Linux environments (e.g., your personal laptop). Be aware of the output and check if any error occurs, which is usually due to lack of necessary dependencies. Contact the TA if you don't know how to handle those dependencies.

Notice that Ripes depends on Qt, which is a widget toolkit for creating graphical user interfaces (GUI). We have installed Qt on the servers. Thus we recommend you to run your labs on the servers provided (You can do the coding on other platforms of course). If you want to run your labs on other platforms, you will need to install Qt (version must **be Qt5 and later than 5.10**) and RISC-V compilers by yourself, which may be troublesome. If you really want to do so, you can find some instructions here or ask TAs for helps.

Now you have successfully set up the Lab 1 environment.

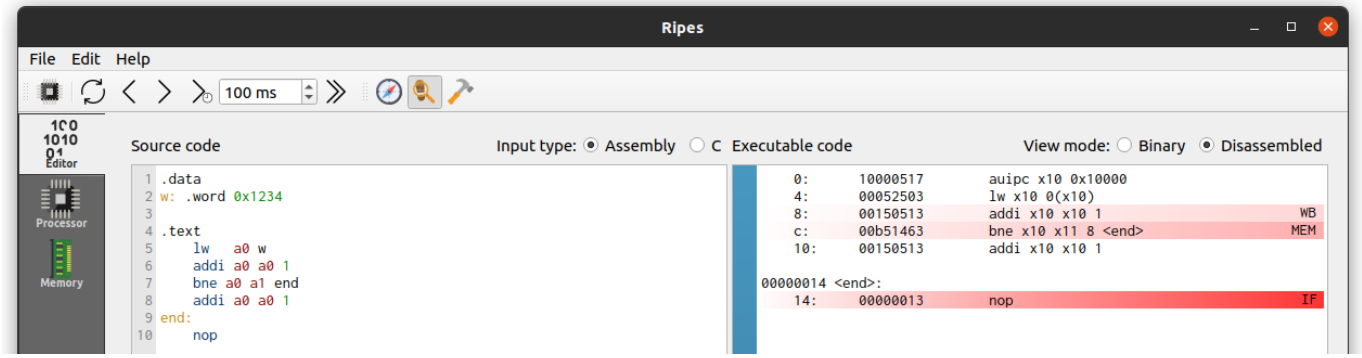
1.3 Ripes Basic Tutorial

Now let's take a look at Ripes, the simulator we are going to use in all three labs. Ripes is an open-sourced RISC-V simulator, where your RISC-V assembly gets simulated on an x86 machine, instead of being executed by a real RISC-V processor which is uncommon today. For example, Ripes uses a bunch of `int` variables to imitate the functionality of processor registers. And Ripes uses `switch... case...` structures to imitate the instruction decoding. Ripes supports multiple processor models to execute the assembly with different performance. In the next lab, we will learn more about this, and also implement our own processor model!

In our lab, we are going to use a slightly modified version of it that better matches our purpose. So you need to download Ripes from the link above, rather than from the public repository.

There are 3 tabs in Ripes:

The Editor Tab



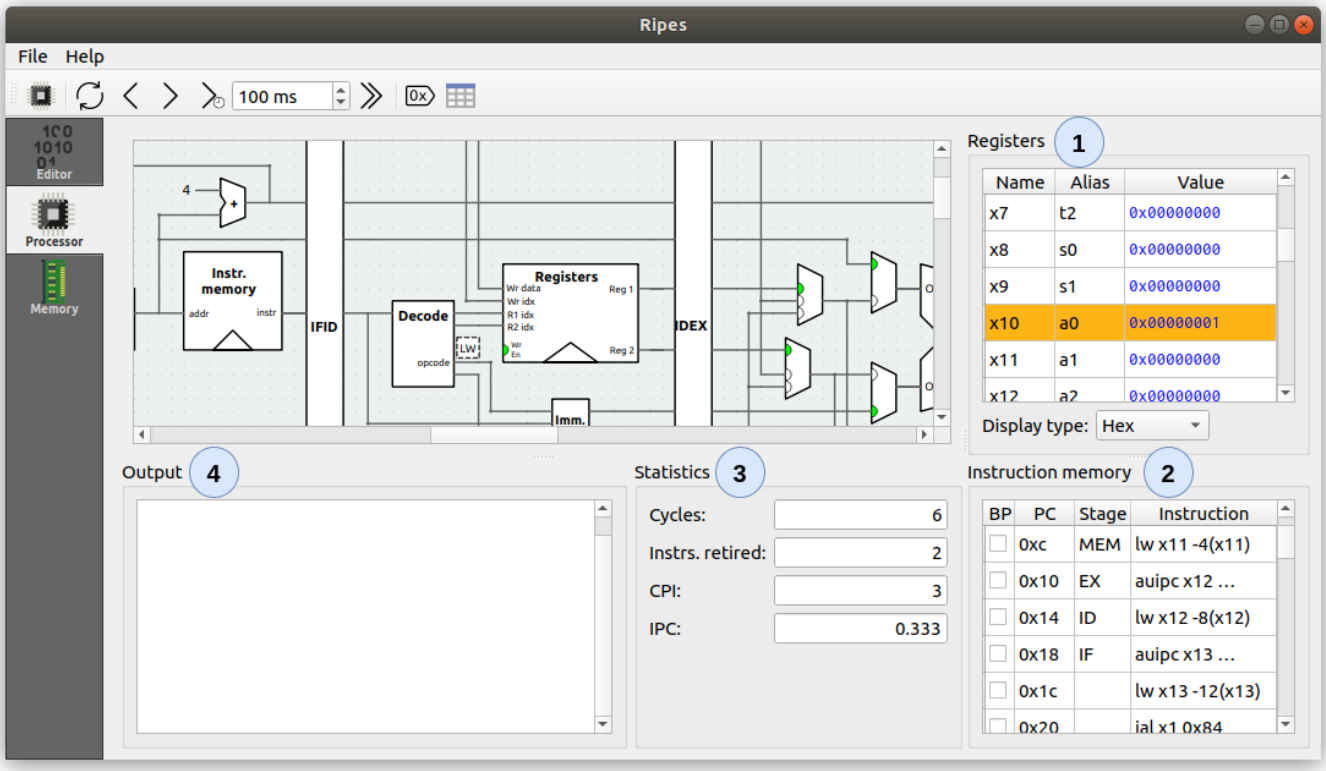
The editor tab shows two code segments. On the left side, you can write an assembly program using the RISC-V RV32 (I/M) instruction sets. Whenever edits are performed in this assembly program - and no syntax errors are found - the assembly code will automatically be assembled and inserted into the simulator. If a C compiler has been registered, the `input type` may be set to `C`. It is then possible to write, compile and execute C programs within Ripes. We will talk about how to load and execute C programs in Chapter [1.4.1](#1.4.1 Load C Program).

Next, on the right side, a second code view is displayed. This is a non-interactive view of the current program in its assembled state, denoted as the *program viewer*. We may view the assembled program as either disassembled RISC-V instructions, or as the raw binary code. Pressing the compass icon will bring up a list of all symbols in the current program. Through this, it is possible to navigate the program viewer to any of these symbols.

Ripes is bundled with various examples of RISC-V assembly programs, which can be found under the `File -> Load Examples` menu.

The Processor Tab

The processor tab is where Ripes displays its view of the currently selected processor, as well as any additional information relevant to the execution. You can find the contents of registers (1) and instruction memory (2), as well as the statistics of the execution (3) here. The output console (4) shows the output of the executed program.



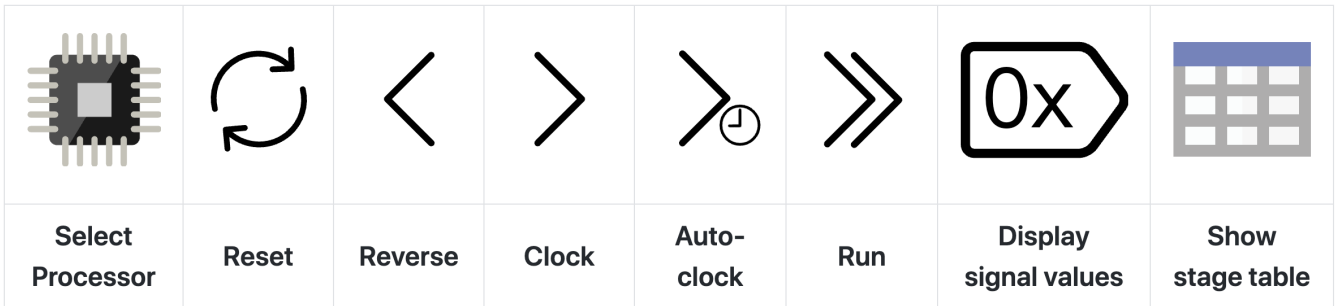
The Memory Tab

The memory tab provides a view into the entire addressable address space of the processor, as well as access to Ripes' cache simulator, which we won't use until Lab 3.

Controlling the Simulator

The control of the simulator is done by a set of buttons as described blow.

The toolbar within Ripes contains all of the relevant actions to control the simulator.



- **Select Processor**: Opens the processor selection dialog. In Lab 1, you should always choose the 5-stage processor.
- **Reset**: Resets the processor, sets the program counter to the entry point of the current program, and resets the simulator memory.
- **Reverse**: Undoes a clock-cycle.
- **Clock**: Clocks all sequential elements in the circuit for one cycle and updates the states.
- **Auto-clock**: Clocks the circuit with the given frequency specified by the auto-clock interval. Auto-clocking will **stop** once a breakpoint is hit.

- **Run**: Executes the simulator **without** performing GUI updates, to be as fast as possible. Any print `ecall` functions will still be printed to the output console. Running will **stop** once a breakpoint is hit or an exit `ecall` has been performed.

If you want to know more about Ripes, you can find an introduction of Ripes here. But the information above is enough for our labs.

1.4 Load Programs

Ripes can be used to run both C programs and RISC-V assembly programs. First we start Ripes:

(When running Ripes, remember to user X11 forwarding when accessing to server)

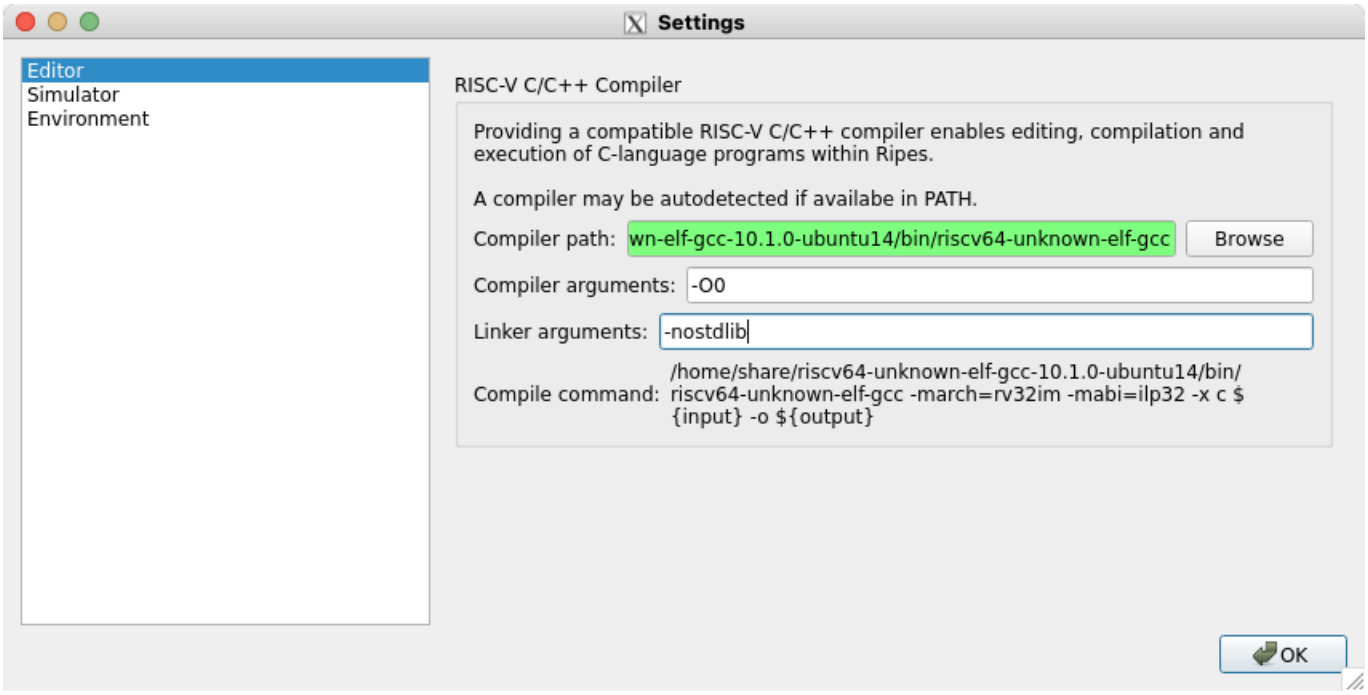
```
$ ./Ripes
```

1.4.1 Load a C Program

Now we test a `simple_add` C program. Choose `File -> Load Program` on the RIPES window. Choose `Source File` as the file type and open `run/simple_add.c`. **You need to specify your own location of the input file (the location of simple_add_data.txt).** The C program will be shown on the left.

Then we need to select the compiler to compile the C program to executable machine code. We provide you a compiler in `/home/share/riscv64-unknown-elf-gcc-10.1.0-ubuntu14/`. Now navigate to `Edit -> Settings -> Editor`. It should look like the picture below. If the compiler path haven't been set, you can set the compiler manually. Click `Browse` and choose `/home/share/riscv64-unknown-elf-gcc-10.1.0-ubuntu14/bin/riscv64-unknown-elf-gcc`.

Notice that the compiler arguments should be `-O0` and the linker arguments should be `-nostdlib`.



Next, click the `Compile C Program` button above, which looks like a hammer. You will see the compiled result on the right.

Now switch to the processor tab. Click the `run` button. You will see the output result in the console.

1.4.2 Load a RISC-V Program

To load programs, you need to **switch to the editor tab** first. Choose `File -> Load Program` on the Ripes window. Choose `Source File` as the file type and open `run/simple_add.S`. **You need to specify your own location of the input file (the location of simple_add_data.txt).** The RISC-V program will be shown on the left.

This time you do not need to click the compile button.

Switch to the processor tab and `run`. You can see the output result in the console. (**If the Ripes program crashes**, please check the terminal where you type `./Ripes`, you may see it complains about the error. In most cases it is because of the invalid path of input files.)

2 Generate Pixel File

To make it simpler to work on the image data, we will extract only the pixels from a `.jpg` image into a `.pixel` file. Use the Python script we provide as follows.

```
$ cd lab1/run
$ python3 gen_pixel.py ../sample/turing.jpg turing.pixel
```

Note: If you encounter errors, you can either install the missing modules, or just use the pixel file `turing_before.pixel` provided in the `sample/` folder. For example, when you see the error `ImportError: No module named Image`, do `$ python3 -m pip install pillow numpy --user` first, and then try the above command again.

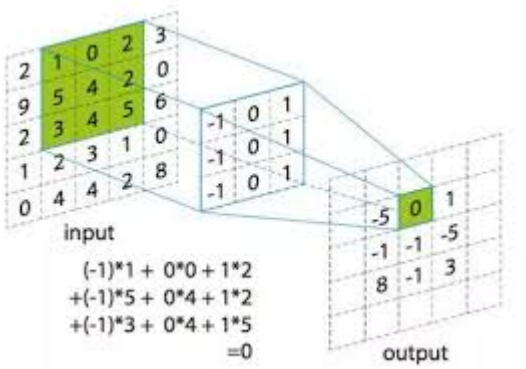
Note: `.pixel` file format description:

```
N M : for the first line, height and width of image
img(i,j) ... : for the next N lines, M numbers each line, img(i, j)
representing grayscale of each pixel: 0-255
```

Check if the output file meets this format.

3 Write Gaussian-like Filter in C Programming Language

3.1 Implementation Discription



There are many existing algorithms for image processing. Most of them apply convolution operations to the image. We would like to implement a simple gauss-like image filter which uses a 3 * 3 kernel matrix `K`:

```
1 2 1
2 4 2
1 2 1
```

We have provided you a basic framework to start with: `lab1-c-starter-code.c`. It contains the following global variables:

- `n` and `m`: The height and width of the image.
- `img`: A one-dimension array that contains the original image. The first `n * m` elements of `img` contain the image pixel data. For example, `img[i * m + j]` is the pixel on the `i-th` row and `j-th` column (`0 <= i < n, 0 <= j < m`). **You do not need to read the image data into `img`.** We have done this in the function `image_input`.
- `result_img`: A one-dimension array that contains the result image after processing. **You should store your processed result in `result_img`** so that the function `image_output` can output it.
- `read_path`: The path to the input `.pixel` file. **You need to change it to specify your own location of the input pixel file.**
- `write_path`: The path to the output `.pixel` file. **You need to change it to specify your own location of the output pixel file.**

We have implemented the following functions to handle read/write data from/to pixel files.

- `image_input`: A function to read `n` and `m`, as well as the entire remaining pixels into `img` from a `.pixel` file. **Please do NOT modify this function.**
- `image_output`: A function to print `n` and `m`, as well as the entire processed result `result_img` to the console and to a `.pixel` file. **Please do NOT modify this function.**

You need to implement one critical function marked as `TODO`:

- `image_process`: For each pixel `P` in `img`, we define its 8 neighboring pixels, as well as itself, as the 3 * 3 neighboring matrix `N(P)`. Then we do element-wise multiplications of `N(P)` and the kernel `K`. Finally we accumulate all the nine products and normalize it by dividing with 16, to generate the new pixel `P'` and store it in `result_img`. This entire image gets blurred because the grayscale of each pixel has *diluted* to its neighboring pixels. **We leave the image borders (the pixels in the first and last rows and columns) unchanged.** Therefore, the size of the output image should be the same as the input.

The processing formula is like this:

$$result\_img[x,y] = \begin{cases} \frac{1}{16}\sum_{i=0}^{2}\sum_{j=0}^{2} img[x+i-1, y+j-1] * K[i,j] & \forall x,y > 0 \wedge x < N-1 \wedge y < M-1 \\ img[x,y] & \forall x = 0 \vee x = N-1 \\ img[x,y] & \forall y = 0 \vee y = N-1 \end{cases}$$

3.2 Test Correctness

To test the correctness of your C program, enter the folder where your C program resides. Then run

```
$ gcc -DCOMPILE_X86 lab1-c.c -o lab1-c
$ ./lab1-c
```

Your program will print the processed `.pixel` format both to the console and to the output pixel file. Check the result and see whether the result is correct. We have provided you a correct processed pixel file named `turing_processed.pixel` in `lab1/sample`.

Hint: Your program can also run in Ripes. If you want to run this C program in Ripes, please refer to Chapter [1.4.1](#1.4.1 Load a C Program). (In Ripes, you will be able to see the cycle count and other information directly)

Hint: You can literally view the processed image through the output `.pixel` file by typing `python3 parse_pixel.py [output file path]` in the terminal.

Hint: A good way to test the correctness of your code is to design some small test pixel files and see whether your code outputs the correct answer.

4 Write Gaussian-like Filter in RISC-V Assembly

Sometimes the compiler is not able to generate the most efficient assembly code. That is the time when we prefer to write in assembly languages directly. Let's see which implementation performs better.

Again, we provide you a basic framework to start with. See `lab1-riscv-starter-code.S`. Complete all `TODO`s. Finish your implementation and name the file as `lab1-riscv.S`.

To test the correctness of your RISC-V code, use the same way in Chapter [1.4.2](#1.4.2 Load a RISC-V Program) to load and run it.

Your program will print the processed `.pixel` format both to the console and to the output pixel file. Check the result and see whether the result is correct.

Hint: The output results between your C and RISC-V implementations should match. Otherwise there must be bug(s) in either (or both!) of them. That is the reason why you should start with simpler C programming.

5 Optimize RISC-V

You may (or may not) see the RISC-V implementation outperform the C implementation. But in any case, let's not just call it a day here. You can do better with RISC-V! Your next task is to optimize your RISC-V implementation. The faster, the better. Name your optimized implementation as `lab1-riscv-opt.S`.

Hint: Some tips for code optimization that you may find useful:

- Use shift instructions (e.g., `srai`, `slli`) instead of multiply/division. Notice the kernel elements are all power-of-2.
- You may notice large overheads from memory accesses. Try to reduce memory accesses by reusing as many registers as possible.

- Another way to optimize memory accesses is through loop optimization techniques, such as loop unrolling and loop blocking to fully utilize the cache. For more information, refer to the wiki page at the top.

Note: You can only use RV32I, RV32M and some pseudo instructions in this lab. SIMD extensions are not allowed or supported. Please refer to the RISC-V reference & card.

6 (Optional) A Faster Way to Test Assembly Programs

There are two ways to run your assembly programs.

The first way, as we introduced above, is to load programs in the editor tab. You can debug using the information provided in Ripes in this way. However, it is kind of slow to run programs with a GUI.

We also provide another way to test your assembly programs. This is done by running a test program we provided. First, you need to modify the paths in `~/yao-archlab-f22/Ripes/test/tst_lab1.cpp`. You need to specify the following 3 paths:

- testDir: the path to your assembly code.
- resultDir: the path to the output pixel file.
- correctResultDir: the path to the correct pixel file (e.g., for turing.jpg, it is turing_processed.pixel).

Then, run

```
$ cd ~/yao-archlab-f22/Ripes/build && make -j4
$ ./test/tst_lab1
```

You will probably see the following information:

If you pass:

```
PASS   : tst_LAB1::initTestCase()
Test Pass! Total cycle: 34
PASS   : tst_LAB1::testAssemblyImpl()
PASS   : tst_LAB1::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 6ms
********* Finished testing of tst_LAB1 *********
```

If you fail:

```
PASS   : tst_LAB1::initTestCase()
FAIL!  : tst_LAB1::testAssemblyImpl() False result
   Loc: [/home/tianby/s22/Ripes/test/tst_lab1.cpp(131)]
PASS   : tst_LAB1::cleanupTestCase()
Totals: 2 passed, 1 failed, 0 skipped, 0 blacklisted, 7ms
********* Finished testing of tst_LAB1 *********
```

The `tst_LAB1::testAssemblyImpl()` is the result of your code. And the `Test Pass! Total cycle: xx` line outputs the total cycles used.

As you can see, you can only get the correctness and the performance (cycle count) in this way, but without any debug information. So it is more useful to test your programs when you are basically sure that the program is correct.

Write-up

You should write a design document for Lab 1. There is no specific format required, but you should demonstrate how your program works, in a clear way. You should at least include the following parts:

Methodology

How did you finish your lab. Explain how your assembly program is organized. Show your debugging process if there is anything interesting.

Summarize the optimizations you have used to improve the performance of the RISC-V implementation. For each optimization, include a short summary of the code changes, whether it helps reduce instruction count and/or CPI and why, how much percentage reduction in the number of cycles it results in, etc.

Results

In your design document, record the performance of your programs by using the total cycles of: 1) the C implementation; 2) the unoptimized RISC-V implementation; 3) the optimized, final RISC-V implementation.

Question Answering

- Compile the your C code into RISC-V assembly using GCC: `/home/share/riscv64-unknown-elf-gcc-10.1.0-ubuntu14/bin/riscv64-unknown-elf-gcc lab1-c.c -S -nostdlib -O0 -o lab1-c.S`, and vary the optimization level `-O0` to `-O3`. Check out the output assembly file `lab1-c.S`, try to explain what optimization techniques are used by GCC O3; you only need to list two techniques.
- What are the limitations still in your optimized code? Any ideas on how to overcome them? Describe the ideas; no need to implement.

Submission

Create a folder with the name of `lab1-<student ID>-<first name>-<last name>`, e.g., `lab1-2020123456-xiaoming-wang`. Put the following files in it.

```
+-- lab1-design-document.pdf --- Your design document
+-- lab1-c.c --- C version of your program
+-- lab1-riscv.S --- unoptimized RISC-V version of your program
+-- lab1-riscv-opt.S --- optimized RISC-V version of your program
```

Compress the folder as a `.zip` package by:

```
$ zip -r lab1-<student ID>-<first name>-<last name>.zip lab1-<student ID>-<first name>-<last name>/
```

Upload the zip file to `learn.tsinghua.edu.cn` (网络学堂) by **October 25 (Thuesday)**. You may submit for multiple times, and we will grade based on the latest submission.

Grading Policy

Plagiarism is **strictly** forbidden. Peer discussion is **not** suggested. Contact the TAs if you are in trouble.

- 20% - Correct C implementation.
- 20% - Correct unoptimized RISC-V implementation.
- 20% - Correct optimized RISC-V implementation.
- 20% - Design document: the description of the optimization methods used and why they help improve the performance.
- 10% - Design document: question answering
- 5% - Design document: others.
- 5% - Performance of the optimized RISC-V implementation. Score by given performance bars (see below).

Correctness

We will run your program to generate an output pixel file, and then compare it with a pre-processed ground truth. More matches mean higher correctness score. Notice that **we will use different and multiple input images for grading**! (All smaller than `turing.jpg`). So you should thoroughly test your programs before submitting.

Performance

Measured by the total cycles of your programs from the output of Ripes. We set the following grading bars based on the distribution of performance results from the last year. The full score is 5 points.

Cycles (10^4)	Score	Number of submissions in this range last year
[0, 168)	5	12
[168, 195)	4	17
[195, 243)	3	13
[243, 270)	2	10
[270, ∞)	1	5