Lab 2

`Objective: Practice RISC-V Assembly Language`

In this lab, you will practice writing and debugging programs that are written with the RISC-V assembly language. You will test these programs with an RISC-V assembler called `rars`. The RISC-V assembly language is a language that resembles the native RISC-V machine instruction set with some additional *pseudo instructions* as well as special assembler directives to provide additional information about the code that will be run. NOTE: since there is not a single universal RISC-V assembler definition, the list of supported directives and some calling conventions can vary between assemblers. In this lab, you will be using those supported by the `rars` program.

1 Getting the Code

You may find the source files needed for this lab from:

`https://www.eee.hku.hk/~elec3441/sp24/handout/elec3441lab2.zip`

2 Getting the Software

`rars` is an open source full feature RISC-V assembler that is used by many to learn able the RISC-V processor. You can find the `rars` software from:

`https://github.com/TheThirdOne/rars`

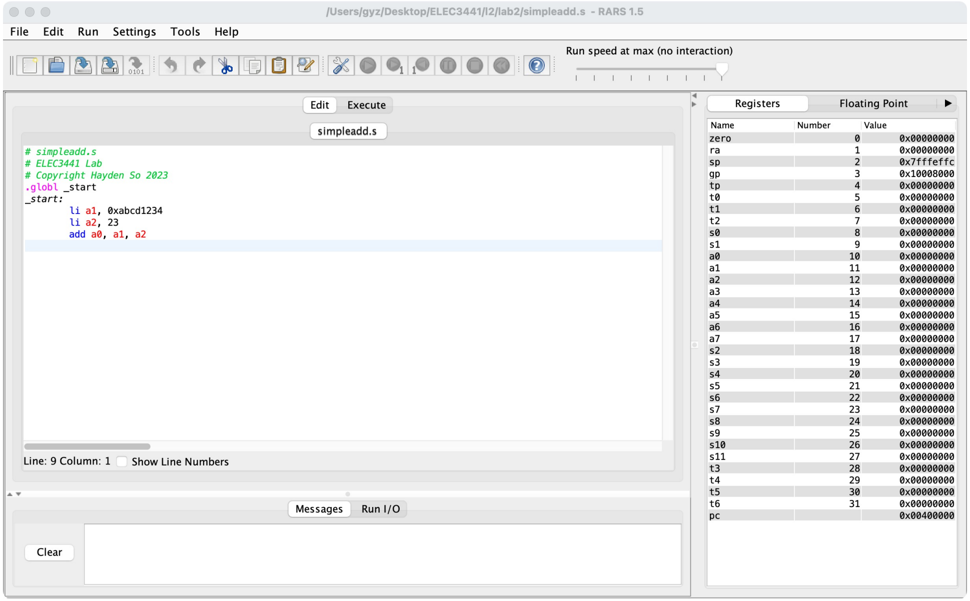
The easiest way to start using the software is to download the pre-built `jar` file from the release page:

`https://github.com/TheThirdOne/rars/releases/tag/continuous`

You will need the Java run time to execute the `jar` file. Download it here `https://www.java.com/en/download/`. Once installed, double-click the `rars.jar` icon to start the GUI.

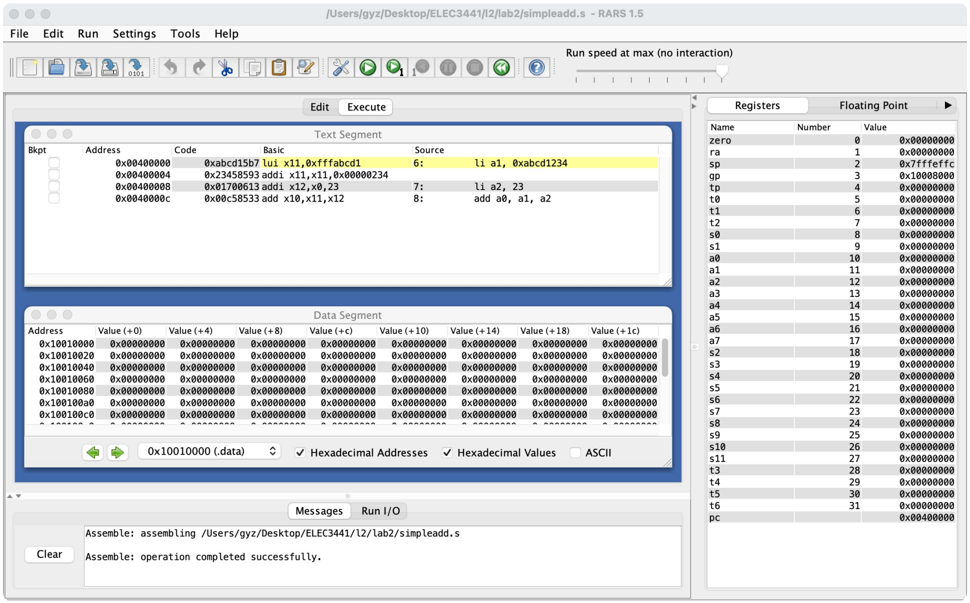
3 Your First Program

3.1 Starting `rars` Run the `rars` program. You should see the initial UI similar to the following:



Load the file lab2/simpleadd.s by using the file open button ( ) on the top left. You should see the content of the file is now loaded under the main **Edit** pane.

3.2 Running the Code To run the code, click the Assemble ( ) button. A new **Execute** pane will show up that shows the current running program:



Within the **Execute Pane**, there are two windows corresponding to the **Text Segment** and the **Data Segment** of the memory. The two windows show the content of the system memory. Program instructions will be shown in the Text Segment, while program data are stored under the Data Segment.

3.3 Tracing Program Execution Now that the program is assembled and loaded into the main memory, you may execute the program in the assembler. You can run the whole program to the end with the **Run** button ( ). Alternatively, and most often, you will want to step through the program one instruction at a time by using the Step function ( ).

Now, step through the program one instruction at a time. After executing each line, make sure you understand how the register values are affected, including the PC register.

3.4 Check Yourself

Run the program and make sure you know the answer to these questions:
(i) What value is stored in the register `a0` when the program is first loaded.
(ii) What is the value of `a0` after the program finishes running?
(iii) What is the hardware register number of `a1` (0 to 31)?
(iv) What is the memory address of the first instruction in your program?

4 Basic RISC-V Assembly Programming

4.1 Pseudo-instruction Load the program `lab2/li.s` and assemble the file.

4.2 Load Immediate Line 6 and 7 of `li.s` uses a load immediate (`li`) instruction that is a *pseudo-instruction* or sometimes also called an *assembly instruction*. In this case, `li` instruction loads a constant value into the target register `rd`:

```
li rd, <value>
```

Pseudo-instructions are instructions that are very similar to the machine instructions we described in class, but are supported only by an assembler. Before being executed in a processor, it must first be assembled and translated into their corresponding machine instructions.

4.3 Examine the content of the **Text Segment** to see how the original `li` instructions are being assembled.

Consider line 6. When the constant immediate value is less or equal to 12 bits, the assembler translates the `li` instruction i it into one `addi` machine instruction (`addi x12, x0, 23`). Note that `x0` register always contains the value 0. So it will be `x12=0+23=23` in the end.

4.4 Larger constant If you try to load a constant larger than 12 bits, the assembler will translate the `li` instruction differently.

4.5 Checkoff 1

Trace through the execution of the code and compare the machine instructions and their corresponding source assembly instructions. Then answer the following:
(i) How many machine instructions have the `li` instruction been assembled into?
(ii) Experiment with loading different constants. How are the `li` instructions being assembled differently with different constant values?

4.6 Jump Load the file `jump.s`.

4.7 `jump`, or `j`, is another commonly used pseudo instruction that jumps to a label (similar to `goto` in C language):

```
j, <label>
```

This pseudo-instruction is usually translated into `jal` (*jump-and-link*) machine instruction as:

```
jal, x0, <offset>
```

where *offset* is the memory address offset between the current PC and the `<lable>` code segment. Since `x0` register always contains the value zero, it basically means the return address can be discarded with this jump.

4.8 Check Yourself

1. Examine the file. How are the branches and jump instructions translated?
2. Trace through the code and make sure you understand how the code is executed with the branch and jump instructions.
3. Consider the offset in the jump *machine* instruction, try to see if you can compute that offset from the relative difference between the jump instruction and the target label.

4.9 Load/Store Load the file `laodstore.s` into `rars`.

4.10 `riscv` uses dedicated load/store instructions to load/store values from/to memory.

```
l{b|h|w}, rd, offset(rs1)
s{b|h|w}, rs2, offset(rs1)
```

`b|h|w` refer to *byte*, *half-word*, and *word*. In a load/store instruction, the value is rs1 register is considered as an absolute base address in the memory. Adding the offset immediate value onto the base address will give you the final address in the memory.

4.11 Assemble the file `loadstore.s`. The program loads data stored in the label `arrayPrime`. The address of this label is hard coded in the file. You will explore how the address can be found automatically later in this lab.

4.12 Checkoff 2

Trace through the execution of the file `loadstore.s` and answer the following questions:

1. What are the values in the array `arrayPrime` after the code has completed?
2. What is the base address of `arrayPrime`?
3. If you insert another `.word` *before* the label `arrayPrime` and assemble the file again, where would the values of `arrayPrime` be moved to in memory?

5 Mysterious Program

Load the program `lab2/mystery.s` into `rars`. It contains a fully functional program with a mysterious function. There are a number of assembler directives that you may not be familiar with. Go through the following steps to learn about how they are used.

5.1 Text vs Data In the file `lab2/mystery` you will notice two new directives: `.text` and `.data`.

The directive `.text` tells `rars` (and any other RISC-V machine) that the following lines are **program instructions** and should be loaded into the *Text Segment* of the main memory.

The directive `.data` tells `rars` that the lines followed should be loaded into the *Data Segment* of the main memory.

In real world systems, the operating system kernel will designate areas of the main memory for different purposes. Some of them will be for instructions, while others will be for data. For your interest: common segments include: text, data, stack, extern, etc.

For this program, we will simply use the data and text segments.

5.2 The data label `anArray` and `anArrayLen` are defined in line 42 and 49 respectively. Note, each of the directive `.word` stands for a 32-bit word and its value. In other word, `anArray` consists of six words, and the next word in memory contains the value of `anArrayLen`.

Now **Assemble** the file and look into the **Execute** pane. Look in the **Data** segment. Can you find the data being loaded there? What is the actual address that `anArray` is stored at? What is the address of `anArrayLen`?

5.3 Look at line 9 and line 10 of the original `mystery.s`. There is a new pseudo instruction `la`, which has the format:

```
la <rd>, <label>
```

The function of the **l**oad the **a**ddress instruction (`la`) is to load the memory address of the given label into the target register `rd`.

When we write position independent code (PIC), the programmer do not explicitly define the location where the data is stored. Instead, the location where data is stored is determined by the operating system, and in this case, the `rars`. Therefore, you must use pseudo instructions like `la` to load the address.

Now, look into the Execute pane. See how the pseudo instruction has been translated into two instruction sequence with `auipc` and `add`.

The new instruction `auipc` behaves similar to the `lui` instruction by loading a value to the upper 20-bit of the target register `rd`. However, in the case of `auipc`, the value to be loaded is relative to the PC of the current instruction:

```
rd = pc + imm << 12
```

5.4 Look in the Execute pane. What machine instructions are produced from the two load address instructions in line 9 and 10? Why are they produced that way?

Hint: They are meant to find the address of the array `anArray` and the variable `anArrayLen`. You can look for them in the Data Segment and find out their addresses.

5.5 Now, trace through the code. How many times the function `chk_number` is being called? What is the pseudo instruction `call` translated into? What instruction is used to implement the `ret` return pseudo instruction?

5.6 ecall The `ecall` In line 24 and 27 is the Environment call. Environment Call (or System Call) provides interfaces between a program and the operating systme (in our case, it's `rars`). `rars` support different types of system call such as `PrintInt`, `PrintString`, which can be specified using register `a7`.

The argument of the system call will be placed in `a0-a6` registers. More details in the system call are specified in `https://github.com/TheThirdOne/rars/wiki/Environment-Calls`.

In the `mystery.s` program, it uses `ecall` to print the final results in the **Run I/O** pane.

5.7 Checkoff 3

(i) What is the value printed at the **Run I/O** pane after running the program?
(ii) In word, describe what is the function of the program `mystery.s`?

5.8 Submission Submit your answers to Checkoff 1, 2, and 3 above.