# ENCM 369 Winter 2023 Lab 2 for the Week of January 23

Steve Norman
Department of Electrical & Software Engineering
University of Calgary

January 2023

## Administrative details

### Each student must hand in their own assignment

Later in the course, you may be allowed to work in pairs on some assignments.

### Due Dates

The Due Date for this assignment is 6:00pm Friday, January 27.
The Late Due Date is 6:00pm Saturday, January 28.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be not be marked.

### Marking scheme

| | |
|---|---|
| A | 4 marks |
| C | 4 marks |
| D | 10 marks |
| total | 16 marks |

### How to package and hand in your assignments

You must submit your work as a *single PDF file* to the D2L dropbox that will be set up for this assignment. The dropbox will be configured to accept only file per student, but you may replace that file as many times as you like before the due date.

See the Lab 1 instructions for more information about preparing and uploading your PDF file.

## Exercise A: Finding machine code for instructions

### Read This First

Before the introduction of assemblers, programmers had to use programming manuals to determine bit patterns for instructions. That's essentially what you will do in this exercise. The main benefits are (a) some insight into the relationship between assembly language and machine code and (b) practice looking up instruction descriptions in the textbook.

Example problem:

> Find machine code for the instruction `add s0, t0, t2`

Solution:

> The machine code format for an `add` instruction is described in Section 6.4.1 of the course textbook, starting on page 333. Bits 31:25 are `0000000`, bits 14:12 are 000, and bits 6:0 are `0110011`, following an example in Figure 6.13. Table 6.1 (page 305) says `s0` is register 8 (`01000`), `t0` is register 5 (`00101`) and `t2` is register 7 (`00111`). So the bit pattern for the overall instruction is
>
> `0000000_00111_00101_000_01000_00000_0110011`

## What to Do

Use information in Sections 6.4.1–6.4.3—and possibly also the inside covers—of the textbook to find machine code for the following instructions:

```
sub     s1, s1, t5
sw      s4, (t3)
lw      t6, 72(s3)
addi    s7, s6, -16  # Hint: Involves 12-bit two's-complement.
```

Write brief explanations of how you found the bit patterns for each instruction. (Your explanations do not have to be as detailed as the one I gave in the example.)

## What to Put in Your PDF Submission

A list of bit patterns for each of the above instructions, along with brief explanations of how you found the bit patterns.

# RARS

In ENCM 369 we'll use a program called RARS (RISC-V Assembler and Runtime Simulator) to learn about assembly language programming and relationships between C code and machine instructions. Here is the Web link to the home page for RARS:

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

Please see the D2L module called "Information about running RARS" for information about installing and running RARS.

Figure 1 on page 4 provides an overview of different parts of the RARS graphical user interface. Figure 2 on page 5 show more detail of the parts of RARS that display the state of a program running in RARS. Figure 3 on page 6 explains the use of some of the icons in the RARS "menus and icons" area.

RARS simulates running a program in a Unix-like environment. (Linux is an example of a Unix-like environment.) In that kind of environment a running program has access to at least three main regions of address space:

- A *text segment*, where the program instructions are located. In RARS, the text segment starts at address `0x0040_0000`. The PC is initialized to `0x0040_0000` when a RARS program runs, so a program starts by fetching and executing the instruction located at address `0x0040_0000`.

- A *data segment*, containing global variables (as described in Lab 1 Exercise B), string constants, and other statically allocated data. In RARS, the text segment starts at address `0x1001_0000`.
- A *stack segment*. We won't use the stack until Lab 3.

User programs running on most modern operating systems generally do not have permission to read or write directly to or from terminal windows, access the file system or network hardware, or do other similar things that involve direct control of I/O devices in the computer. All a user program can do is access registers, and read or write the memory allocated to the program.

To get something done with an I/O device, a user program makes a *system call*, which suspends the user program, and requests that another program called the operating system *kernel* take over and provide some sort of service for the user program. So, for example, to print a message in a terminal window, a user program would have to make a system call. Once the kernel has taken care of the system call, it allows the user program to continue (unless the system call was a request for termination of the user program).

In RISC-V, a user program makes a system call this way: the type of system call is specified in the `a7` register; sometimes more information is specified in the `a0` register; finally, an `ecall` instruction is run.

# Exercise B: A first RARS program

## Read This First

It seems to be an important tradition when learning a new programming language to start with a program that prints a message like ...

```
hello, world
```

... so that is what you will do with RARS. Doing so will help you get acquainted with some of the basics of using RARS.

## What to Do

Create a folder to work in and start up RARS.

In the Edit & Execute area, make sure the Edit tab is selected. Then use File→New to create a new file. RARS will give this a name something like `risvc1.asm`. Before typing in any code, use File→Save as ... to save the file as `hello.asm` *within the folder you just created.* If you have done this correctly, `hello.asm` should appear as the name of the file you are editing.

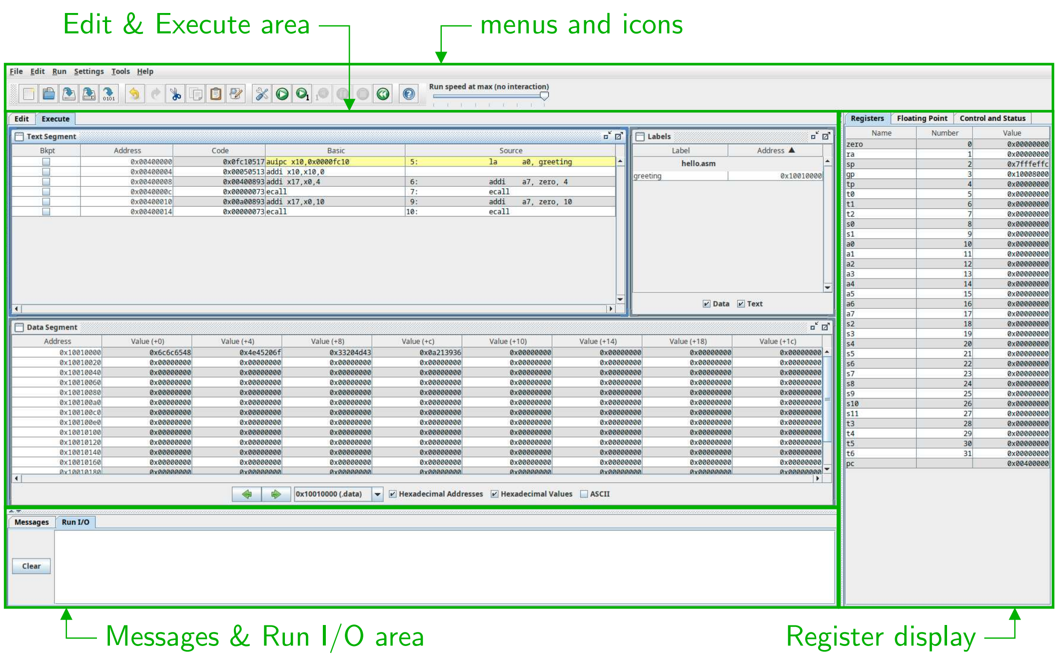
Into the window for `hello.asm`, type in the text inside the box in Figure 4. Read what you have typed to make sure you have everything correct—this is computer programming, so every little punctuation character matters, as does using lowercase instead of uppercase, or vice versa. (Fortunately, though, RARS does not care much whether you use lots of space characters or a smaller number of tab characters to put "white space" in between pieces of source code on a single line.)

Use the Assemble icon (see Figure 3 if you don't remember what that is) to try to assemble your code. (Note: This will automatically save the file you are editing.)

If you have made any significant typing mistakes, there will be be error messages in the Messages tab near the bottom of the screen. If that happens, fix the mistakes, then try to assemble again.

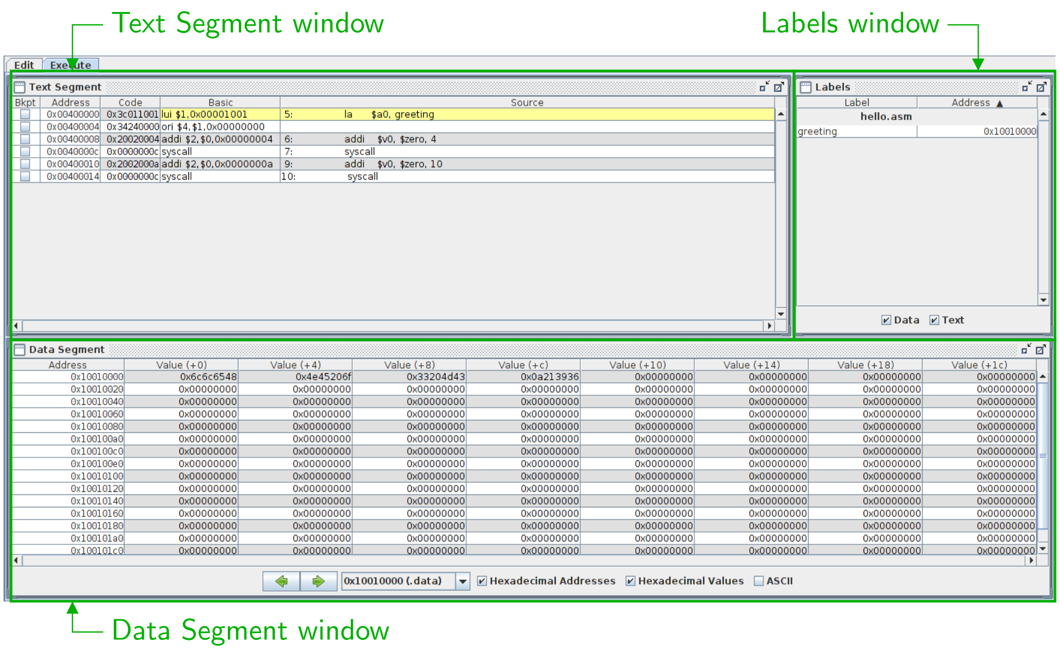
When Assemble succeeds, you will see the Execute tab. Before you run the program, here are some things you should look at:

**Figure 1:** Overview of the RARS user interface—screenshot taken on a Linux machine, with some annotations added in green. Don't bother trying to read the tiny characters in the screenshot; just get an idea of what each of the major areas are used for.



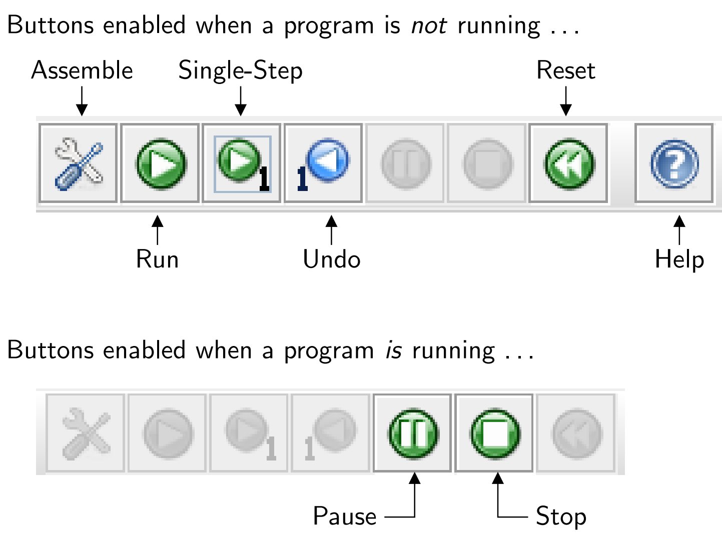

| **Area** | **Remarks** |
|---|---|
| menus and icons | Used to control the editor, assembler, and RISC-V simulator. |
| Edit & Execute | The Edit tab is used to write and modify assembly language source code files. The Execute tab is used to view machine code, run programs, and inspect contents of data memory. |
| Messages & Run I/O | Messages displays assembler error messages, and also various messages about what state the simulator is in as you run a program. Run I/O is for input and output done with `ecall` ("environment call") instructions. |
| Register display | The Registers tab displays contents of the GPRs and the PC. Early in the term, you won't have any use for the Floating Point and Control and Status tabs. |

**Figure 2:** Detail of the Execute tab—another screenshot taken on a Linux machine, with more annotations added in green.




| Window | Remarks |
|---|---|
| Text segment | Displays instructions of your program in machine code and assembly language format; allows you to set and clear *breakpoints*, which are very handy things. The yellow highlight indicates the next instruction to be fetched and executed. |
| Data segment | Shows the content of data memory, eight words per row. Can be used to inspect the `.data` section and *also the stack*. (We'll start using the stack in Lab 3.) |
| Labels | Lists all the labels in your program along with the corresponding addresses. Double-clicking on a label finds you the matching instruction or data item in the Text Segment or Data Segment. To see this window, pull down the Settings menu and check the appropriate box. |

**Figure 3:** Important icons in the RARS user interface.




| Icon | Remarks |
|---|---|
| Assemble | Usually used in the Edit tab. Requests translation of assembly-language code into machine code, so the machine code can be run in the Execute tab. |
| Run | Starts a program that has just been assembled *or—this is important*—continues a program that has been paused by a breakpoint or by use of the Pause icon. |
| Single-Step | Runs the next instruction, then pauses execution. |
| Undo | Undoes most recently executed instruction, reversing changes to registers and memory. Very useful in a simulator, generally not available on real processors! |
| Reset | Reverts registers and memory to their initial states for the program in the Execute tab. |
| Help | Opens a window with help on using RARS *and* general help about RISC-V assembly language. |
| Pause | Freezes a running program. Very useful if you think your program might have entered an infinite loop. |
| Stop | Terminates a running program. |

**Figure 4:** Source code listing for Exercise B. Line 2 sets up a string literal allocated in the data segment. Lines 5 to 7 are instructions to print the string literal, and Lines 9 and 10 are instructions the program uses to terminate itself.

```
1                    .data
2   greeting:        .asciz         "Hello ENCM 369!\n"
3
4                    .text
5                    la      a0, greeting
6                    addi    a7, zero, 4
7                    ecall
8
9                    addi    a7, zero, 10
10                   ecall
```

- The machine instructions of your program are shown in hexadecimal in the Code column of the Text Segment window.
- The assembly language instructions you typed into the editor are shown in the Source column.
- Note that the `la` instruction is not a real RISC-V instruction; instead, it's a *pseudoinstruction* that tells the assembler to generate two real instructions to do the work of putting the address corresponding to a label into a GPR. (For now don't worry about the details of `auipc`—that instruction will be covered in lectures early in February.)
- In the Settings menus, check Show Labels Window—you should now see a Labels window, which tells you that the address corresponding to the label `greeting` is `0x10010000`.
- In the Data Segment window, check the box for ASCII display. You should now see the characters of `"Hello ENCM 369!\n"`, apparently in a somewhat scrambled order, starting at address `0x10010000`. (For now, don't worry about why the order of the characters appears to be scrambled.) Uncheck the ASCII box to once again display the data segment contents as words in hexadecimal format.
- Look again at the Text Segment. The instruction at the very beginning of the Text Segment, at address `0x00400000`, is highlighted in yellow. When you run the program, it will start with that instruction.

Run the program by clicking on the Run icon, then check the Run I/O tab to see what output was produced.

## What to Include in Your PDF Submission

Nothing.

## Remark

This exercise demonstrates how to get simple output from a program. However, to find out whether your code works in RARS, it turns out to be much easier to simply look at memory and registers than it is to have your program print its results. Until Lab 4, you won't have to write any more instructions to generate output.

**Figure 5:** Outline of a typical assembly language file. Directives `.data` and `.text` can be used repeatedly to switch back and forth between specifying data for the data segment and specifying instructions for the text segment.

```
    .data
directives to allocate and initialize static data

    .text
instructions

    .data
more directives to allocate and initialize static data

    .text
more instructions

      ⋮

    .data
more directives to allocate and initialize static data

    .text
more instructions
```

# Exercise C: Translating C code that has a `main` function

## Read This First: Start-up and clean-up code

When you program in C, typically you think of a function called `main` as the starting point for your program, and that when your program runs, the last thing it does is return from `main`.

However, in fact, when you build an executable file from C code, that executable will probably contain not only instructions for all of the C functions you wrote, but also some instructions that run before `main` starts and some more instructions that run after `main` is finished. These extra instructions sometimes go by the name of "start-up and clean-up code". The start-up code is located at the very beginning of the text segment for the program.

Starting with this exercise you will write programs by copying a file containing start-up and clean-up code, and also definitions for a `main` function and perhaps some other functions. To get a program to work, *you will never need to modify the start-up and clean-up code.* Instead you should edit `main`, perhaps some static data directives, and perhaps some other functions.

## What to Do

### Part 1

Make a directory to work in. Into that directory, download the file `stub1.asm`.

Start up RARS and open the file in the editor. Read the file. Do not worry about the details of the start-up and clean-up code. However, do pay attention to the use of `.data` and `.text` directives, as explained in Figure 5.

Assemble and run the program, and note that the clean-up code prints a message including the return value generated by `main`.

**Part 2**

Now download and take a look at the files `add-ints2C.c` and `add-ints2C.asm`. The assembly language file contains start-up and clean-up code copied from `stub1.asm`, and translations of the code in the C file. Note the use of the `.word` directive to allocate and initialize the variables `foo`, `bar`, and `quux`.

In RARS, open `add-ints2C.asm` in the editor and assemble it. In the Execute tab, make sure the Labels window is visible, using the Settings menu. In the Labels window, double-click on `foo`—that will show you that `foo` has an initial value of `0x15`, and that its neighbour `bar` has an initial value of `0x2d`.

In the Text Segment window, scroll down to address `0x00400050`, where the first instruction of `main` is located. For that instruction, check the box in the Bkpt (breakpoint) column—that will cause the program to pause when it gets to that instruction.

Click the Run icon. Notice that the instruction at address `0x00400050` is highlighted, and also that the value of the PC (shown in the Registers tab) is `0x00400050`.

Now click the Single-Step icon. Note the updates to the PC and to `t0`.

Click the Single-Step icon again and again; after each click note changes to the PC and t-registers. (If you think you have missed something, click on the Undo icon to back up.) Keep doing this until you have seen the value of `quux`—at address `0x10010034` in the data segment—get updated. Once you have seen that, click the Run icon to let the program run to completion.

**Part 3**

*If you have skipped Parts 1 and 2, or Exercise B, because you have figured out that there are no marks for any of them, please go back and do them—they teach concepts and skills you will need to know.*

Download the file `array-sum2C.asm`, load it into the RARS editor and carefully read the last 30 lines or so (starting with the comment `# Global variables`).

Assemble and run the program. Here are some things to check: look for the array elements in the Data Segments window; make sure the values in the local variable registers `s0`–`s2` are what you expect when the program is finished.

Modify the code in `array-sum2C.asm` so that in addition to putting the sum of the array elements in the local variable `sum`, it also updates the variable `max` with the maximum value from the array.

Get the logic right—don't just write something that puts a value of $-4$ into the `s3` register! To create an `if` statement you will need a branch instruction. Document each instruction you add with a C-like comment.

## What to Include in Your PDF Submission

Include a listing of your completed `array-sum2C.asm` file.

# Exercise D: Practice with arrays, loops, and `if` statements

## What to Do

Download the file `lab2exD.c`. Follow the instructions given in comments in the file.

*Hint #1:* Start by making a copy of `stub1.asm` from Exercise C, and renaming it `lab2exD.asm`.

*Hint #2:* This is one good way to set up your two global arrays, each with eight `int`s:

```
        .data
        .globl  alpha
alpha:  .word 0xb1, 0xe1, 0x91, 0xc1, 0x81, 0xa1, 0xf1, 0xd1
        .globl  beta
beta:   .word 0x0, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70
```

## What to Include in Your PDF Submission

Include a listing of your completed `lab2exD.asm` file.