ENCM 369 Winter 2023 Lab 4 for the Week of February 6

Steve Norman
Department of Electrical & Software Engineering
University of Calgary

February 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 Monday, February 13. The Late Due Date is 6:00pm Tuesday, February 14. You being given a little extra time this week due to Midterm 1.

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 not be marked.

Marking scheme

C	6 marks
E	6 marks
total	12 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.

Getting `.c` and `.asm` files for this lab

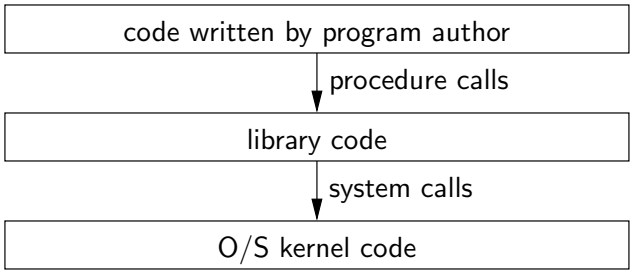
The files you need for all six exercises are in a folder called `encm369w23lab04`, which has been uploaded to D2L in `.zip` format. Before starting work, you must download the `.zip` file, and extract its contents in a suitable place in the file system of your computer.

Exercise A: RARS system calls and terminal output

Read This First, Part I: System calls

If you write a C program and run it on an operating system such as Linux, Microsoft Windows or Mac OS X, three collections of instructions are usually needed for the program to do its job: (1) instructions generated from the C code you wrote yourself; (2) instructions belonging to the library; and (3) instructions belonging to the operating system kernel. The kernel is a program that allows other programs to run; the kernel manages such things as input/output devices, file systems, and allocation of physical memory to other programs.

Instructions in groups (1) and (2) are located in the program's text segment. Instructions in group (3) are found in the kernel's text segment. Typically, applications interact with the kernel through the library, as shown below:

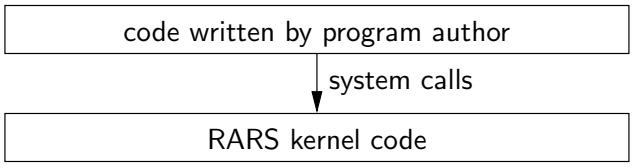


For example, if your program has to open a file, it calls `fopen`, which is a library function, and `fopen` makes a *system call* to get the kernel to open the file for the program.

A system call works according to the following general scheme:

- A value is copied into some register to indicate which kernel service is required.
- More values may copied into other registers as "arguments" for the kernel to use.
- A `ecall` instruction is executed, which suspends execution of the user's program and starts execution of kernel code. ("Environment call" is the RISC-V name for "system call".)
- The kernel does what it is asked to do. Sometimes this may involve the kernel using registers to return information to the user program.
- The kernel then transfers control back to the user's program, which resumes execution with the instruction after the `ecall` instruction.

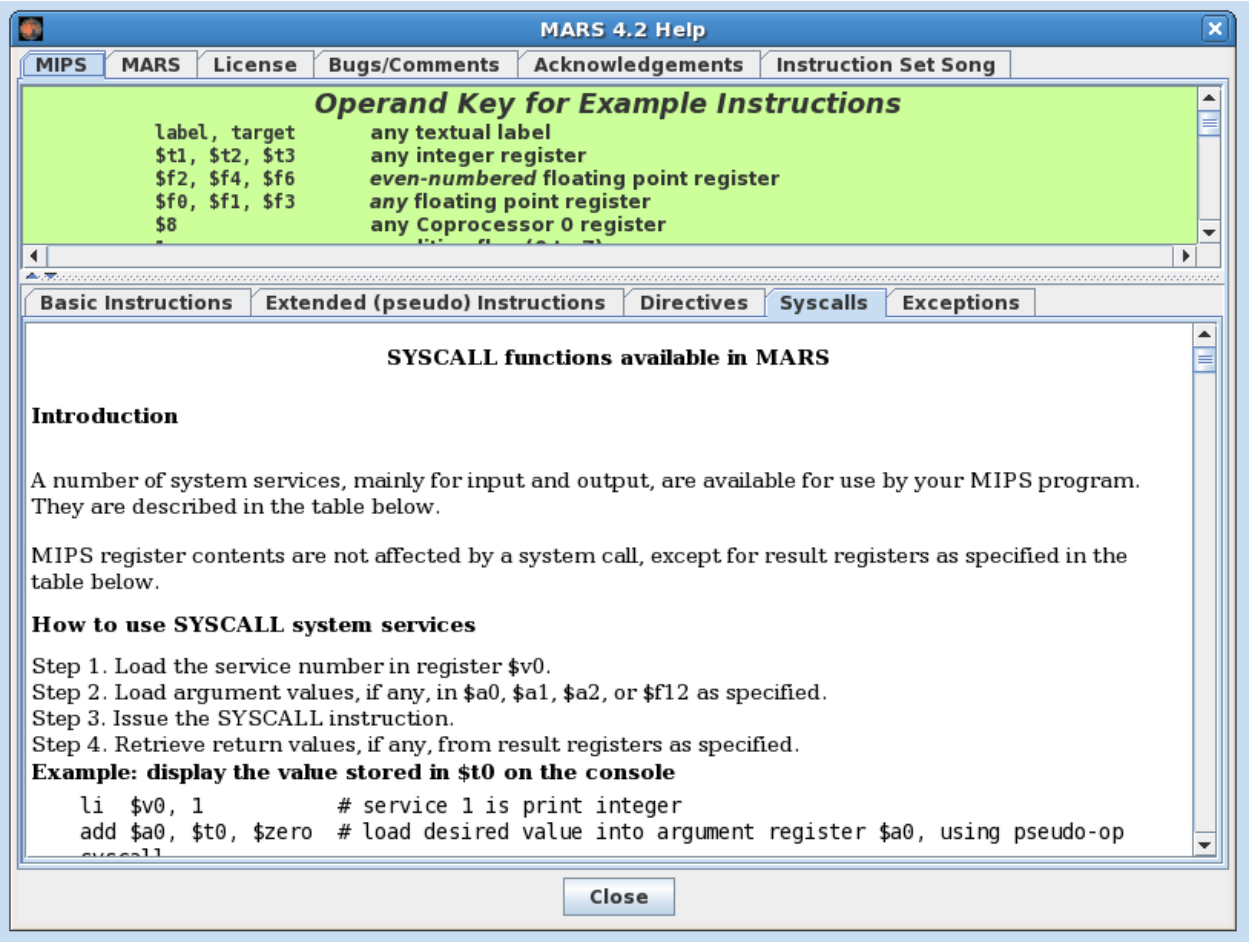
RARS, unlike most C development environments, has no library at all. So your RARS programs can't call functions to do things like input and output—instead your programs must make system calls, as follows:



The system call services provided my RARS are very nicely documented within RARS—see Figure 1 as a guide to getting started with that.

It is worth noting that RARS "fakes" execution of system calls, in the sense that you won't find actual instructions in the kernel text area to handle writing

Figure 1: Screenshot of RARS Help window with the Syscalls tab selected. Bring this up in RARS, then scroll down to see detailed documentation of all the RARS system call services.



and reading numbers and strings to and from input/output devices. It would be interesting to be able to step through the code that did that, but unfortunately you can't.

What to Do

Copy the file `encm369w23lab04/exA/do_output.asm`, and predict the output. Use RARS to test your prediction—look for output in the Run I/O tab.

What to Include in Your PDF Submission

There is nothing to submit for this exercise.

Exercise B: String-copy demonstration

Read This First

The `.asciz` assembler directive is a convenient way to allocate a zero-terminated sequence of ASCII codes in an array of bytes in the `.data` area of a RARS program. (The `z` in `.asciz` stands for *zero*; keep in mind that the code for the `'\0'` character used to terminate C strings is zero.) For example, the directive

```
S1:  .asciz "hello"
```

would allocate an array of 6 bytes containing the ASCII codes for `'h'`, `'e'`, `'l'`, `'l'`, `'o'`, and `'\0'`, and set up the symbol `S1` to represent the address of the byte containing the `'h'`.

One important use of `.asciz` directives has to do with string constants, both as initializers of variables, as in

```
char my_array[ ] = "ENCM 369";
```

and as used in various C expressions, such as

```
strcpy(buffer, "Winter 2023 Lab 4");
```

In the latter case, `"Winter 2023 Lab 4"` would not have a name in the C code, but would need a name in assembly language, something like this:

```
      .data
S2:   .asciz "Winter 2023 Lab 4"
```

Then the second argument of the procedure call to `strcpy` could be set up with this:

```
      la    a1, S2
```

The `.asciz` directive was used in some lecture examples. Two other directives haven't been mentioned at all yet in ENCM 369 but are used in this exercise:

- `.space` n

  allocates n bytes of data that will be initialized to zero.

- `.align` n

  ensures that the next item of data will be given an address that is a multiple of 2^n.

  In this exercise, `.align` is used to get two arrays to start at addresses that are multiples of 32. That is not needed to make the program work—it just makes it easier to use the RARS Data Segment window to see what happens when the program runs.

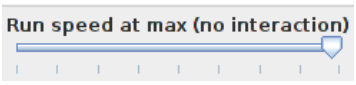
What to Do

Copy the file `encm369w23lab04/exB/ex4B.asm`, and bring it into the RARS editor. Read the definitions of `my_strcpy` and `main`. Everything should make sense to you—if not, please ask questions in your lab period.

Set a breakpoint on the `lbu` instruction at the beginning of `my_strcpy`, and run the program to that breakpoint.

Make sure ASCII display is checked in the control near the lower right of the Data Segment window, then single-step through the loop in `my_strcpy` a few times so you can observe characters being copied one at a time.

Get rid of the breakpoint (by unchecking its box), change



to


,

and click on the Run icon to watch the rest of the string-copy operation happen in "slow motion".

What to Include in Your PDF Submission

Nothing.

Exercise C: Writing string-handling functions

Read This First

This exercise is intended to help you practice processing of C strings using the `lbu` and `sb` instructions.

It's strongly recommended that you do Exercises A and B before this one, as they provide useful examples of output with system calls and C string operations with `lbu` and `sb`.

What to Do

The files you need for this exercise are in `encm369w23lab04/exC`.

Read `string-funcs.c` to get an idea of what it does, then build and run an executable to see if the output is what you expect.

Edit the file `string-funcs.asm` so that the assembly-language definitions of `copycat` and `lab4reverse` match their C definitions. If you have done this correctly, you should find that the RARS program output matches the C program output.

What to Include in Your PDF Submission

Include listings of the code for `copycat` and `lab4reverse` from your completed `string-funcs.asm` file.

Exercise D: Logical instructions

What to Do

The file you need for this exercise is `encm369w23lab04/exD/ex4D.asm`.

Read the program, and, without running it, determine the values that registers `t0`–`t5` will get when the program runs.

What to Include in Your PDF Submission

Nothing. You can check your answers by running the program in RARS.

Exercise E: Programming with logical instructions

Read This First

Section C.5 of your textbook has a nice summary of all of the C *operators*. Here is some more information about the "bitwise shift", "bitwise", and "logical" operators:

- *You must know the differences between* `||` *and* `|`, *between* `&&` *and* `&`, *and between* `!` *and* `~`. For example, suppose the value of `a` is `0x00ff0000` and the value of `b` is `0x0000ff00`. Then
  - `a || b` has a value of `0x00000001` but `a | b` has a value of `0x00ffff00`;
  - `a && b` has a value of `0x00000001` but `a & b` has a value of `0x00000000`;
  - `!a` has a value of `0x00000000` but `~a` has a value of `0xff00ffff`.

  So for RISC-V, a C expression with one of `|`, `&`, or `~`, would be implemented with `or`, `ori`, `and`, `andi`, or `xori`, as appropriate. But it would be *incorrect* to try to implement `||`, `&&`, or `!` with that kind of instruction.
- *In expressions involving* `<<` *and* `>>`, *the data being shifted is on the left and the shift count is on the right.* For example, the value of `0x2fa5_6789 << 4` is `0xfa56_7890`.
- *The C standard specifies that a left shift fills with zero bits on the right, but does not fully specify how filling on the left is to be done in a right shift of an* ***`int`***. In a right shift there are two possibilities: always fill with zero, or fill with copies of the sign bit of the data being shifted. For example, consider this code:

```
int j, k;
j = (int) 0x80000010;
k = j >> 4;
```

  On some C implementations with 32-bit `int`s, `k` would get a bit pattern of `0xf8000001`, but on others, `k` would get a bit pattern of `0x08000001`. On the other hand, a right shift of an `unsigned int` is guaranteed to fill with zero bits. The difference between a plain `int` and an `unsigned int` will be covered in detail in lectures before Winter Break Week.

What to Do

The files for this exercise are in `encm369w23lab05/exE`

The program `bin_and_hex.c` shows some practical applications of shift and logical AND operations. Study the C program for a while, then compile it and run it to see what the output is.

Complete the RARS version of the program in `bin_and_hex.asm`. The functions `main`, `test` and `write_in_hex` are complete and correct, but `write_in_bin` is not. As in exercises in Lab 3 and Exercise C of this lab, your code must follow all the calling conventions introduced so far in ENCM 369, and your instructions should *closely* match the given C code.

What to Include in Your PDF Submission

Include a listing of the code for `write_in_bin` from your completed `bin_and_hex.asm` file.