ENCM 369 Winter 2020 Lab 2 for the Week of January 27

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

January 2020

Lab instructions and other documents for ENCM 369 can be found at
`people.ucalgary.ca/~norman/encm369winter2020/`

Administrative details

Each student must hand in their own assignment

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

Due Dates

The Due Date for this assignment is 3:30pm Friday, January 31.
The Late Due Date is 3:30pm Monday, February 3.

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 returned unmarked.

Marking scheme

A	4 marks
C	4 marks
D	10 marks
total	18 marks

How to package and hand in your assignments

Please follow the instructions given in Lab 1.

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 "R-type" instructions like `add` is described in Section 6.3.1 of the course textbook, starting on page 305. The first six bits are `000000`, and the last 11 bits are `00000_100000`, following an example in Figure 6.6. Table 6.1 (page 300) says `$s0` is register 16 (`10000`), `$t0` is register 8 (`01000`) and `$t2` is register 10 (`01010`). So the bit pattern for the overall instruction is

```
000000_01000_01010_10000_00000_100000
```

What to Do

Use information in Sections 6.3.1 and 6.3.2—and possibly also Appendix B—of the textbook to find machine code for the following instructions:

```
sub     $s1, $s1, $t5
sw      $s4, ($t8)
lw      $t6, 72($s3)
addi    $s7, $s6, -16  # Hint: Involves 16-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 Hand In

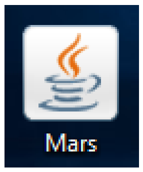
A list of bit patterns for each of the above instructions, along with brief explanations of how you found the bit patterns.

MARS

In ENCM 369 we'll use a program called MARS (MIPS 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 MARS:

https://courses.missouristate.edu/KenVollmar/MARS/

MARS should be installed on the Windows 10 machines in ICT 320. To start it up, double-click on the MARS icon located among the icons on the left edge of the Windows desktop:



If you find MARS missing from a Windows 10 machine, you can download the file `Mars4_5.jar` from the MARS home page, then launch MARS

If you would like to install MARS on your own computer, use a "download" link on the MARS home page. MARS is easy to set up on any reasonably recent Windows, Mac, or Linux platform. Most Linux distributions already have the necessary Java software installed. Installing a Java "SDK" (software development kit) on Mac OS X or Windows is quite straightforward. Here is a link to the download page for Java SDKs and related software:

https://www.oracle.com/technetwork/java/javase/downloads/index.html

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

MARS 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 MARS, the text segment starts at address `0x0040_0000`. The PC is initialized to `0x0040_0000` when a MARS 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 MARS, 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 MIPS, a user program makes a system call this way: the type of system call is specified in the `$v0` register; sometimes more information is specified in the `$a0` register; finally, a `syscall` instruction is run.

Exercise B: A first MARS 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
```

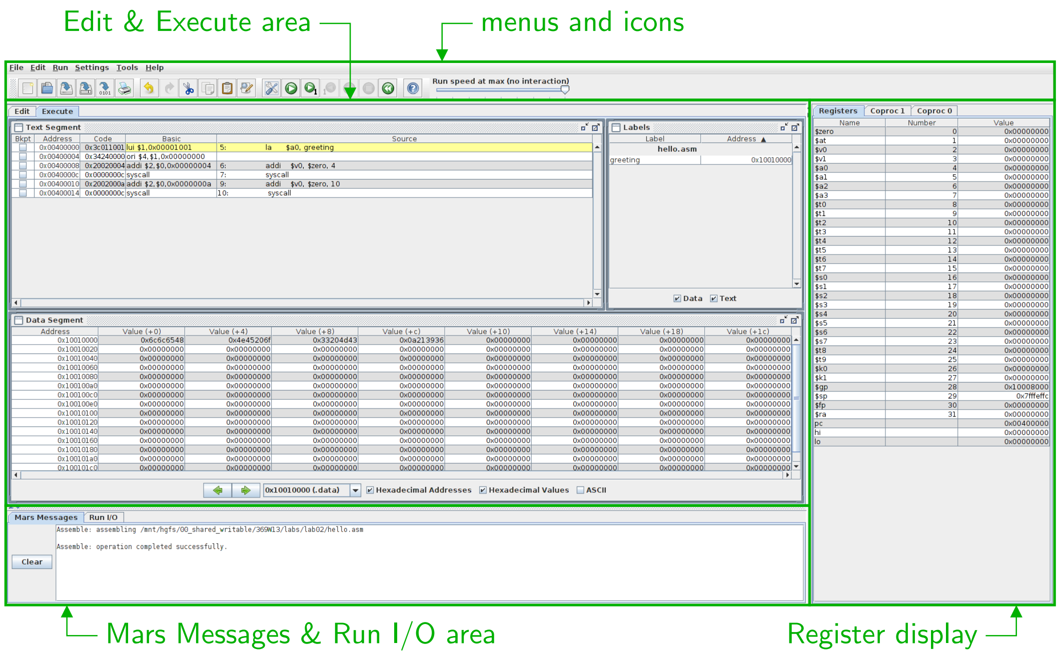
... so that is what you will do with MARS. Doing so will help you get acquainted with some of the basics of using MARS.

What to Do

Create a folder named something like `E:\encm369\lab02\exB` to work in.

Start up MARS. If you are using a Windows 10 machine in ICT 320, do that with the desktop icon, as described a page or two previously. If you are using some other kind of computer, and you have downloaded MARS to that computer, it's up to you to figure out what to do, but most likely it involves double-clicking on the icon for the `.jar` file you downloaded from the MARS home page.

Figure 1: Overview of the MARS 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.


Edit & Execute area
menus and icons
Mars Messages & Run I/O area
Register display
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
Data Text

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	5: la $a0, greeting
☐	0x00400004	0x34240000	ori $4,$1,0x00000000	
☐	0x00400008	0x20020004	addi $2,$0,0x00000004	6: addi $v0, $zero, 4
☐	0x0040000c	0x0000000c	syscall	7: syscall
☐	0x00400010	0x2002000a	addi $2,$0,0x0000000a	9: addi $v0, $zero, 10
☐	0x00400014	0x0000000c	syscall	10: syscall

Labels

Label	Address ▲
hello.asm	
greeting	0x10010000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x6c6c6548	0x4e45206f	0x33204d43	0x0a213936	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

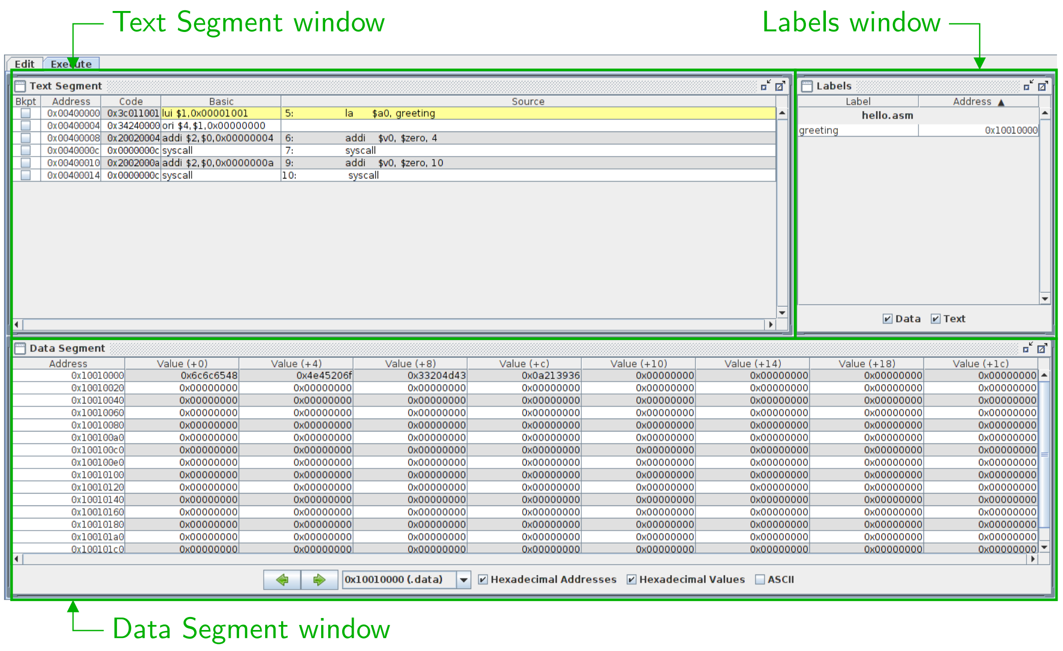
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

```
Assemble: assembling /mnt/hgfs/00_shared_writable/369W13/labs/lab02/hello.asm

Assemble: operation completed successfully.
```

Area	Remarks
menus and icons	Used to control the editor, assembler, and MIPS 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.
Mars Messages & Run I/O	Mars 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 `syscall` instructions.
Register display	The Registers tab displays contents of the GPRs, the PC, and Hi and Lo. (Hi and Lo are used for results of integer multiplication and division instructions.) Early in the course, you won't have any use for the Coproc 1 and Coproc 0 tabs.

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



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	5: la $a0, greeting
☐	0x00400004	0x34240000	ori $4,$1,0x00000000	
☐	0x00400008	0x20020004	addi $2,$0,0x00000004	6: addi $v0, $zero, 4
☐	0x0040000c	0x0000000c	syscall	7: syscall
☐	0x00400010	0x2002000a	addi $2,$0,0x0000000a	9: addi $v0, $zero, 10
☐	0x00400014	0x0000000c	syscall	10: syscall

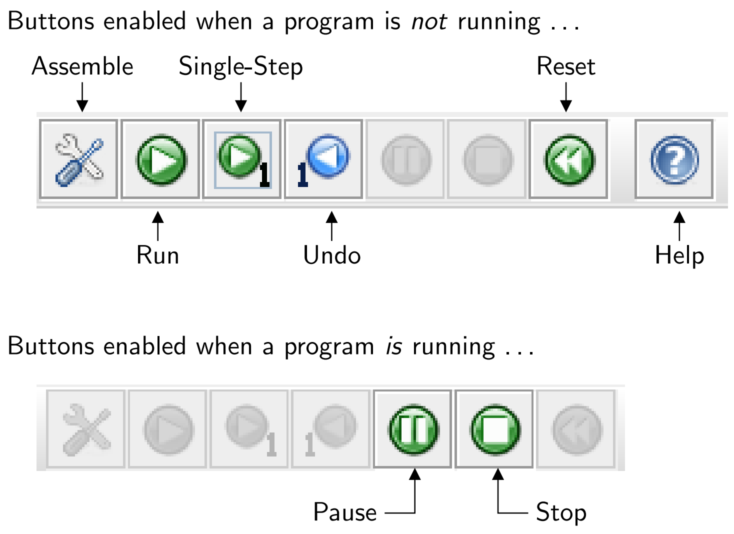
Label	Address ▲
hello.asm	
greeting	0x10010000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x6c6c6548	0x4e45206f	0x33204d43	0x0a213936	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

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 MARS user interface.


Buttons enabled when a program is *not* running . . .
Assemble
Single-Step
Reset
Run
Undo
Help
Buttons enabled when a program *is* running . . .
Pause
Stop

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 MARS *and* general help about MIPS 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 constant allocated in the data segment. Lines 5 to 7 are instructions to print the string constant, and Lines 9 and 10 are instructions the program uses to terminate itself.

```
 1                      .data
 2  greeting:           .asciiz "Hello ENCM 369!\n"
 3
 4                      .text
 5                      la      $a0, greeting
 6                      addi    $v0, $zero, 4
 7                      syscall
 8
 9                      addi    $v0, $zero, 10
10                      syscall
```

In the Edit & Execute area, make sure the Edit tab is selected. Then use File→New to create a new file. MARS will give this a name something like `mips1.asm`. Before typing in any code, use File→Save as ... to save the file as `hello.asm` *within the `exB` directory 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, MARS 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 Mars 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:

- 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 MIPS 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 `lui` and `ori`—those instructions 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 Hand In

Nothing.

Remark

This exercise demonstrates how to get simple output from a program. However, to find out whether your code works in MARS, 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.

How to download files for ENCM 369 labs

[This is copied and pasted in from the Lab 1 instructions, with a few minor edits, to serve as a convenient reminder.]

Most lab exercises in this course will require you to download one or more files containing C code, assembly language code, or data.

Links for downloading these files can be found on the same Web page where you find lab instructions. You have the option of clicking through links to get files one at a time or downloading all the files you need for one lab in a single `.zip` archive.

When you are told to make a copy of, for example, of the files in the directory

`encm369w20lab02/exC`,

you should look for them starting on the ENCM 369 Lab Information web page.

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

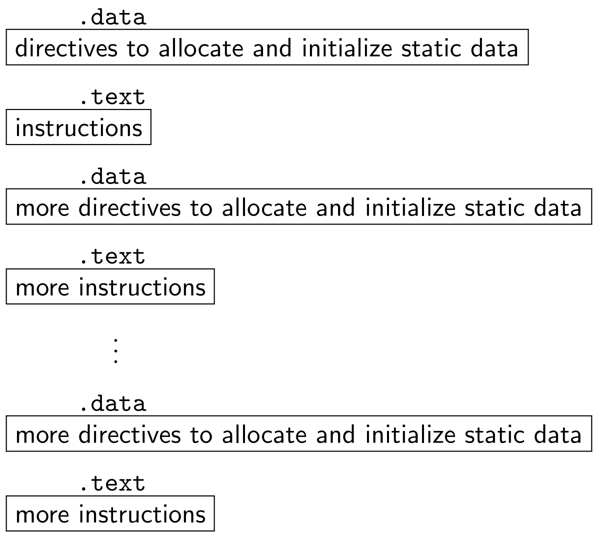
Read This First: Start-up and clean-up code

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.

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

What to Do

Part 1

Make a directory to work in, and copy in all of the files from `encm369w20lab02/exC`.

Start up MARS and open the file `stub1.asm` 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 have a look at the files `add-ints.c` and `add-ints.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 MARS, open `add-ints.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 `0x00400060`, 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 `0x00400060` is highlighted, and also that the value of the PC (shown in the Registers tab) is `0x00400060`.

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.

Open the file `array-sum.asm` in the MARS 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-sum.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—the only two kinds of branch instructions you are allowed to use are `beq` and `bne`; you will probably need to use an `slt` instruction as well. Document each instruction you add with a C-like comment.

What to Hand In

Hand in a printout of your completed `array-sum.asm` file.

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

What to Do

Copy the file `encm369w20lab02/exD/exD.c`. Follow the instructions in the C source file `exD.c`.

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

Hand in a printout of your completed file `exD.asm`.