

Lab 1: MIPS

Arrays, Strings, Loops and Subroutines

Purpose

Learn the basics of the MIPS assembly language by writing a few programs (from simple to complex). After completing this assignment you should have a basic understanding of:

- Arrays (Data stored one after another in memory)
- Strings (Arrays of characters)
- Loops
- Subroutines

Method

We will provide you with a skeleton program `---string_functions.s---` which you will fill in and complete. You will debug and run your program using one of the available MIPS simulators, SPIM or MARS (the latter is recommended).

Preparation

If you need help with MIPS and SPIM/MARS in order to get started with this lab, you can go through the respective tutorials available in on the student portal --- you don't need step 5 for SPIM assignment. It is not mandatory to run the example files mentioned in the tutorials. The tutorial is only meant to give you an overview on how to program in MIPS and use the SPIM or MARS simulators, as

well as providing some practice and hands-on experience. After reading the tutorials, you can work on the main skeleton file for this lab ---stirng_functions.s. You can skip the tutorials in part or completely if you are already familiar with MIPS programming and using SPIM or MARS simulators. You should also be familiar with the basic building blocks of a MIPS CPU:

- Registers
- Program Counter (PC)
- The Stack
- Stack Pointer (SP)
- The Return Address Register (RA)

You should know what a **label** is. (This is a name for a particular location in memory, used to make it easier to read and write assembly code.)

You should also know the difference between the **registers** and the **main memory** and how to address **bytes** and **words** of memory.

Regarding the **stack**, you should know how to put items on the stack (push) and remove them (pop) in MIPS assembly language.

SPIM or MARS allow you to use a set of **syscalls**. Syscalls are operating system routines that interact with the rest of the system. We will be using syscalls to print the program's results to the console.

All code must adhere to the **MIPS Calling Convention**.

File to Use

string_functions.s

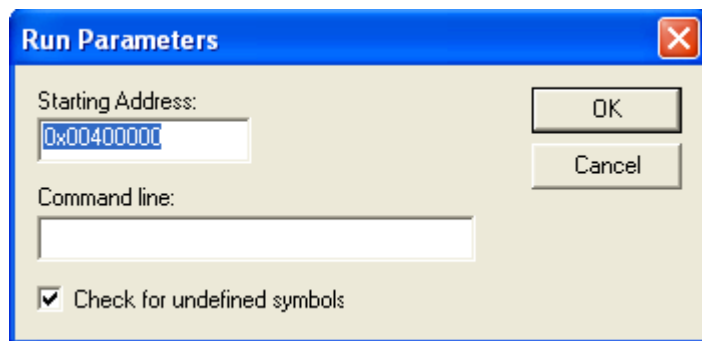
What to Hand In

No hand-ins are required. Demo your completed lab to the assistants during the lab sessions.

Getting started

On the Unix system, start SPIM or MARS---please refer to the tutorials; the commands in this manual are based on SPIM simulator, however you can achieve the same results in MARS as well. Load the **string_functions.s** program. To run the program in SPIM, click on the run button (on Windows you can also press F5). When you run the program the following popup appears:

Here you can choose at which address to start the execution. Normally you should not alter this address. Continue by pressing the OK button.



Output from the program appears in a separate console window, you should see something similar to this:

Have a look at the source file **string_functions.s** You will see it has several sections.

The first section starts with a `“.data”` which indicates that the contents are data you will use in your programs. In the first `.data` section you have several labels

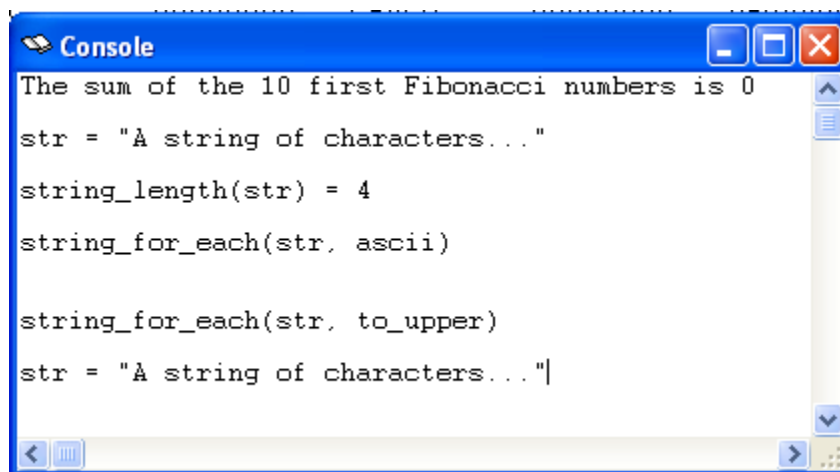
that point to the data. In this case there is `ARRAY_SIZE` (which is a word with the value 10), the `FIBONACCI_ARRAY` (which is 10 words), and `STR_str`, which is a string.

Under that you have several subroutines. The first one is `integer_array_sum`. These are the parts you will fill in for the lab.

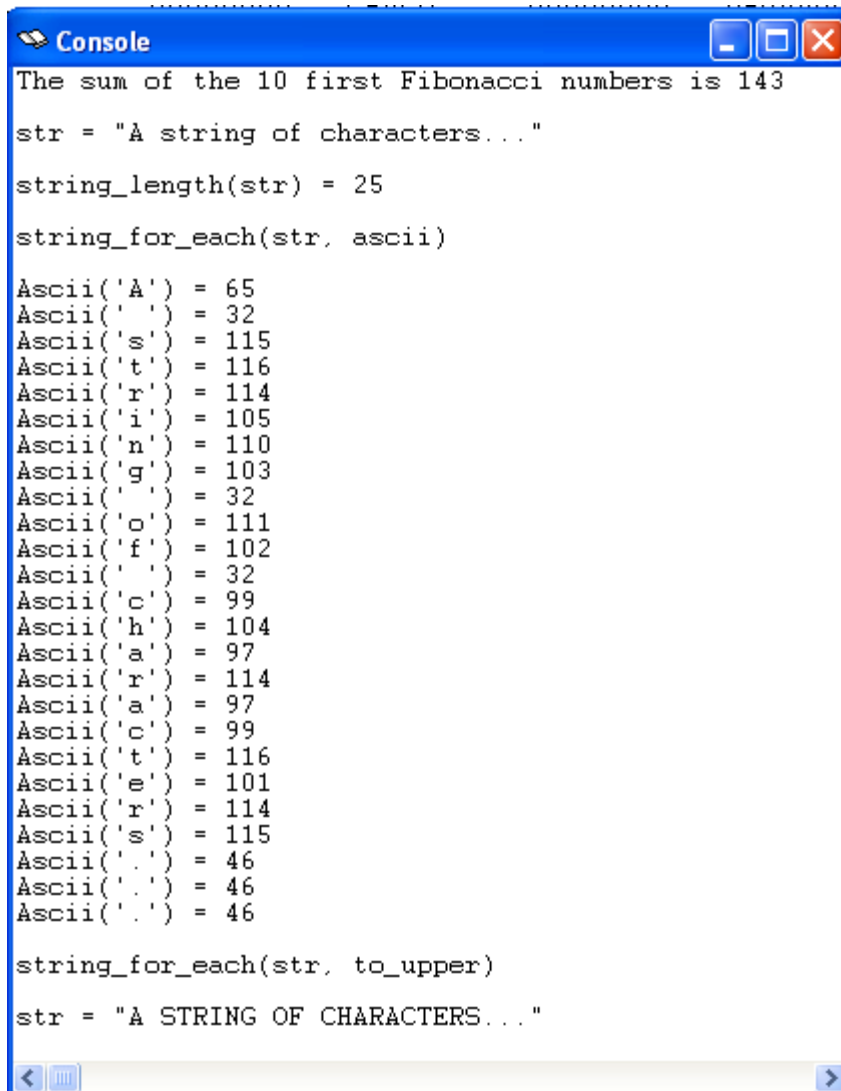
If you scroll down, you will find the `main` subroutine. This is where the program starts. You can see that the first thing it is put various string label addresses into the `$v0` register and do a `syscall` to print them out to the console. Then it puts the `FIBONACCI_ARRAY` and `ARRAY_SIZE` addresses into the `$a0` and `$a1` registers and calls the `integer_array_sum` subroutine with a `jal`. The rest of the main function is similar. Make sure you've read through it and understand how it is producing the output you saw when you ran the program.

NOTE: You do not have to make any changes to main.

When you are done with the assignment, the output from the program should look similar to the following:

A screenshot of a Windows-style console window titled "Console". The window has a blue title bar with standard minimize, maximize, and close buttons. The main area is white and contains the following text: "The sum of the 10 first Fibonacci numbers is 0", "str = \"A string of characters...\"", "string_length(str) = 4", "string_for_each(str, ascii)", "string_for_each(str, to_upper)", and "str = \"A string of characters...|\"". On the right side of the text area, there are vertical scroll arrows. At the bottom, there is a horizontal scrollbar with a slider and arrow buttons.

```
The sum of the 10 first Fibonacci numbers is 0
str = "A string of characters..."
string_length(str) = 4
string_for_each(str, ascii)
string_for_each(str, to_upper)
str = "A string of characters...|"
```

A screenshot of a console window titled "Console". The window has a blue title bar with standard Windows window controls (minimize, maximize, close). The text inside the console is as follows:

```
The sum of the 10 first Fibonacci numbers is 143  
str = "A string of characters..."  
string_length(str) = 25  
string_for_each(str, ascii)  
Ascii('A') = 65  
Ascii(' ') = 32  
Ascii('s') = 115  
Ascii('t') = 116  
Ascii('r') = 114  
Ascii('i') = 105  
Ascii('n') = 110  
Ascii('g') = 103  
Ascii(' ') = 32  
Ascii('o') = 111  
Ascii('f') = 102  
Ascii(' ') = 32  
Ascii('c') = 99  
Ascii('h') = 104  
Ascii('a') = 97  
Ascii('r') = 114  
Ascii('a') = 97  
Ascii('c') = 99  
Ascii('t') = 116  
Ascii('e') = 101  
Ascii('r') = 114  
Ascii('s') = 115  
Ascii('.') = 46  
Ascii('.') = 46  
Ascii('.') = 46  
string_for_each(str, to_upper)  
str = "A STRING OF CHARACTERS..."
```

At the bottom of the console window, there is a horizontal scrollbar with a left arrow, a vertical bar, and a right arrow.

Run

To run the program again, you just press the run button once more.

Reload

After editing the MIPS assembler source file (**string_functions.s**) you must press reload for the updated file to be loaded into SPIM.

Windows: From the top menu, choose simulator→reload.

Step

To be able to follow the execution in more detail, the simulator allows you to step through the instructions one by one.

Click on step.

You will be prompted by a dialog box where you can choose "number of steps". Leave this to the default value of 1.

Now, click on **step** (the button in the newly opened dialog). This will execute the instruction pointed to by the program counter (**PC**) and increment the **PC** by four (next instruction). If the instruction is a branch, the program counter may be updated to another value according to the branch target.

Continue clicking on step and you will be able to run the program one instruction at the time.

By stepping through the program you can easily follow what's happening. For each instruction you can see the changes in the contents of the registers and memory.

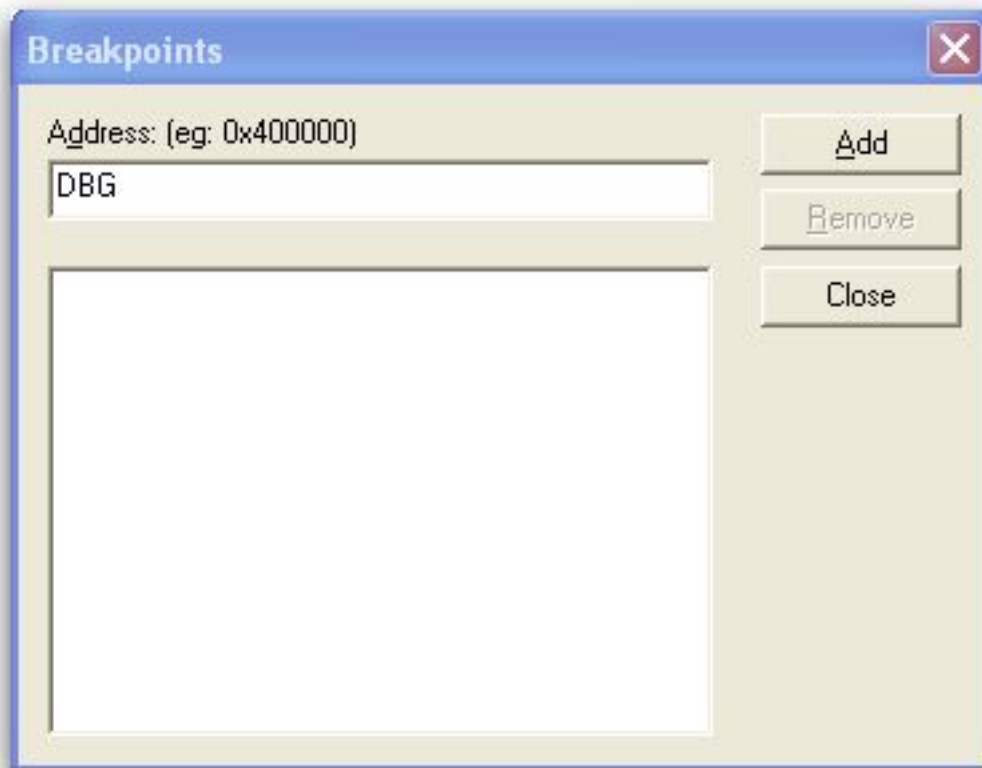
Windows: To single step, press the F10 key.

Breakpoints

A good way to test your programs is to step through the program as explained above. For small programs this might be sufficient but for larger programs this quickly becomes tedious.

A better way is to set a breakpoint in your program. After you added your breakpoint you can run your program and SPIM will stop at the breakpoint. Once SPIM has stopped at the breakpoint you can switch to single stepping to investigate the execution in more detail.

Reload the program. Click on the "breakpoints" button. You will be prompted by a dialog box. In the address field, type DBG:



Windows: You can set a breakpoint by clicking on the  button.

Click on "add".

Run the program.

SPIM will execute the program and stop after a while asking if you want to continue the execution or not. Answer No.

Now you can continue to execute the program one instruction at a time using single stepping as explained before.

DBG is nothing more than a **label**. But to be able to function as a breakpoint, the label must be declared global.

In the beginning of the source file **string_functions.s**, you will see how the **assembler directive .globl** is used to declare the label DBG to be a global label.

```
.globl DBG
```

TASK ONE: `integer_array_sum`

Have a look at the subroutine **`integer_array_sum`**. This routine should iterate through a set of numbers (words, so 4 bytes each) and add them up. The result should be returned to the caller. Remember that this is a subroutine (function) so you will have to obey the MIPS calling convention in how you use registers!

As you can see, the label `DBG` is put here. If we use it as a breakpoint we can run up to this point in the program and switch to stepping.

It's important to write good comments that describe your program. A common mistake is to write comments that do not add anything new. (E.g., writing “`i=i+4 ;` this line increments `i` by 4” is not helpful. Instead you should write “this line increments the index for the string array to the next letter.”) Your comments should describe what your code is doing in terms of the problem you are solving.

As an example of how such "good comments" may look like, each line of the subroutine already have one comment. There are also some labels defined to help you insert branches. Giving good names to labels is also important to make your program easy to understand (and debug).

Your first task is to translate each comment to one line of MIPS assembly. Finish the subroutine **`integer_array_sum`**

Make sure your solution allows for different values for **`ARRAY_SIZE`**, especially the case **`ARRAY_SIZE = 0`** is interesting.

Hints

By a number, we mean a 32 bit number (a word). An array is a sequence of such numbers stored in the data segment.

If you look at the top of the source file **`string_functions.s`** you will see how ten numbers are stored in the data segment using the directive **`.word`**

By means of the label **`FIBONACCI_ARRAY`** we can refer to the address of the first number in the array later in our code. (E.g., wherever you use `FIBONACCI_ARRAY` in your code it will be replaced with the address of that data.)

Since each number is a word (**32 bit** long or **4 bytes**), each number is four bytes apart. Hence the second number is stored at the address given by the following calculation: **FIBONACCI_ARRAY + 4**.

The third number is stored at address **FIBONACCI_ARRAY + 8**. Therefore, since each word is 4 bytes further along than the previous one, we can calculate the address of the *Nth* number with **FIBONACCI_ARRAY + n*4**. To be able to sum all the numbers in the array you will need to write a loop that loads each value in the array and adds them up into a register.

TASK TWO: string_length

For this task you will write a subroutine that takes the memory address of a string and returns the number of characters in that string.

A **string** is nothing more than an **array of bytes** (8 bits, or 1/4 of a word) where each byte encodes one character using the ASCII encoding. To mark the end of the string, the special ASCII-value NULL (0x00) is used to terminate the string.

The length of a string is the number of characters before the terminating NULL. To calculate the length of a string we can loop through the characters and keep count of the number of characters until a terminating NULL is found.

Because each character is one byte, we must only increase the address by 1 byte each time in the loop. Finish the subroutine **string_length** and be sure to include clear comments.

TASK THREE: string_for_each

This routine will take the memory address of a string and the memory address of another subroutine. It will go through every letter in the string and for each letter call the subroutine. We will use this to convert every letter in the to upper case later by providing a string and a function that converts letters to uppercase.

High-level languages can take a string and print out each character on a separate line. For example, the string "ABCabC" would be print out as:

Ascii('A') = 65 (0x41)

Ascii('B') = 66 (0x42)

Ascii('C') = 67 (0x43)

Ascii('a') = 97 (0x61)

Ascii('b') = 98 (0x62)

Ascii('c') = 99 (0x63)

We would like to have something similar to this in MIPS assembly.

We can achieve this by writing a subroutine **string_for_each** taking the address to a string as the first input parameter and the address to a callback subroutine as the second input parameter. The subroutine then loops through all characters in the string and for each character, calls the callback subroutine with the address of the characters as input. In pseudo code it looks like this:

```
string_for_each(string, callback) {  
    for each character in string {  
        callback(address_of(character))  
    }  
}
```

Translating this to MIPS assembly, the input to the callback, the address to a character is put in the **\$a0** register.

In the source file **string_functions.s** there is a subroutine called "print_test_string" that takes an address to a character in memory and writes out the ASCII value as shown above. (E.g., "Ascii('A') = 65 (0x41)" if we give it the

address of the value 'A'.)

Your task is to complete the subroutine **string_for_each**. When you're done, the main program should be able to write out all the ASCII values in the string correctly. Be sure to test with various strings, especially the empty string "". You can test with different strings by either modifying the string used by the main subroutine or by creating your own strings.

TASK FOUR: string_to_upper

Here you will write a subroutine that takes an address to a letter, determines the upper case version of that letter, and stores it back into the same memory location. (E.g., it changes the data in memory to uppercase.) We will then use this in the string_for_each subroutine to convert a whole string to uppercase.

HINT: Have a look at the ASCII table below... What is the difference between a lower case 'a' and the upper case 'A'? When do you need to change the character and when not? (To read the table, you can see that 'A' = 0x41 and 'a' = 0x61.)

TASK FIVE: reverse_string

Write a subroutine to reverse a string. When given a string such as "ABC" as input it should produce "CBA" as the output.

Write a new subroutine called "reverse_string" from scratch. This subroutine should modify the characters in the original memory space of the string and not create a new one.

You will need to add new code to "main" to print out the results on the console.

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Hint: How can you use the address of the string and the string_length subroutine you already wrote to swap the first and last letters? Can you then write a loop that on the next iteration swaps the 2nd and 2nd-to-last letters? When would this loop end? What about even-length and odd-length strings?