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 which you will fill in and complete. You will debug and run your program using the MIPS simulator SPIM (or QTSPIM) or MARS. The instructions in this document are focused on the CMD environment for SPIM and the GUI environment that is QTSPIM. If you want to learn how to use MARS you can do so at the following link http://courses.missouristate.edu/KenVollmar/MARS/.

You can find additional information about the CMD environment for SPIM at the following links: https://cs.gmu.edu/~dnord/cs265/spim_intro.html , https://www.cs.umd.edu/~meesh/cmsc411/website/projects/spim/intro.html , https://manpages.ubuntu.com/manpages/bionic/man1/spim.1.html

Note that for the two links below, you have to make sure to be on the HTTP version of the page not HTTPS. Try using a different browser if you get a URL not found message.

http://pages.cs.wisc.edu/~larus/spim.html

http://pages.cs.wisc.edu/~larus/spim.pdf

You can find additional information about QTSPIM (GUI) at the following links: https://ecs-network.serv.pacific.edu/ecpe-170/tutorials/qtspim-tutorial , http://spimsimulator.sourceforge.net/

Preparation

Before you start with this assignment you should go through this entire document. 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 (not MIPS) allows 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

Completed file **string_functions.s**

Upload the files to the **Lab 1 Submission** on Studium.

Getting started

With spim, you have two environments, the CMD environment and the GUI environment.

To install the CMD environment, you need access to the Linux shell. If you are running Ubuntu, you can install it using 'sudo apt install spim'. For other Linux distributions, use the appropriate package manager.

If you want to run SPIM using the GUI based option, you can download it at http://spimsimulator.sourceforge.net/.

Open SPIM and load the **string_functions.s** program.

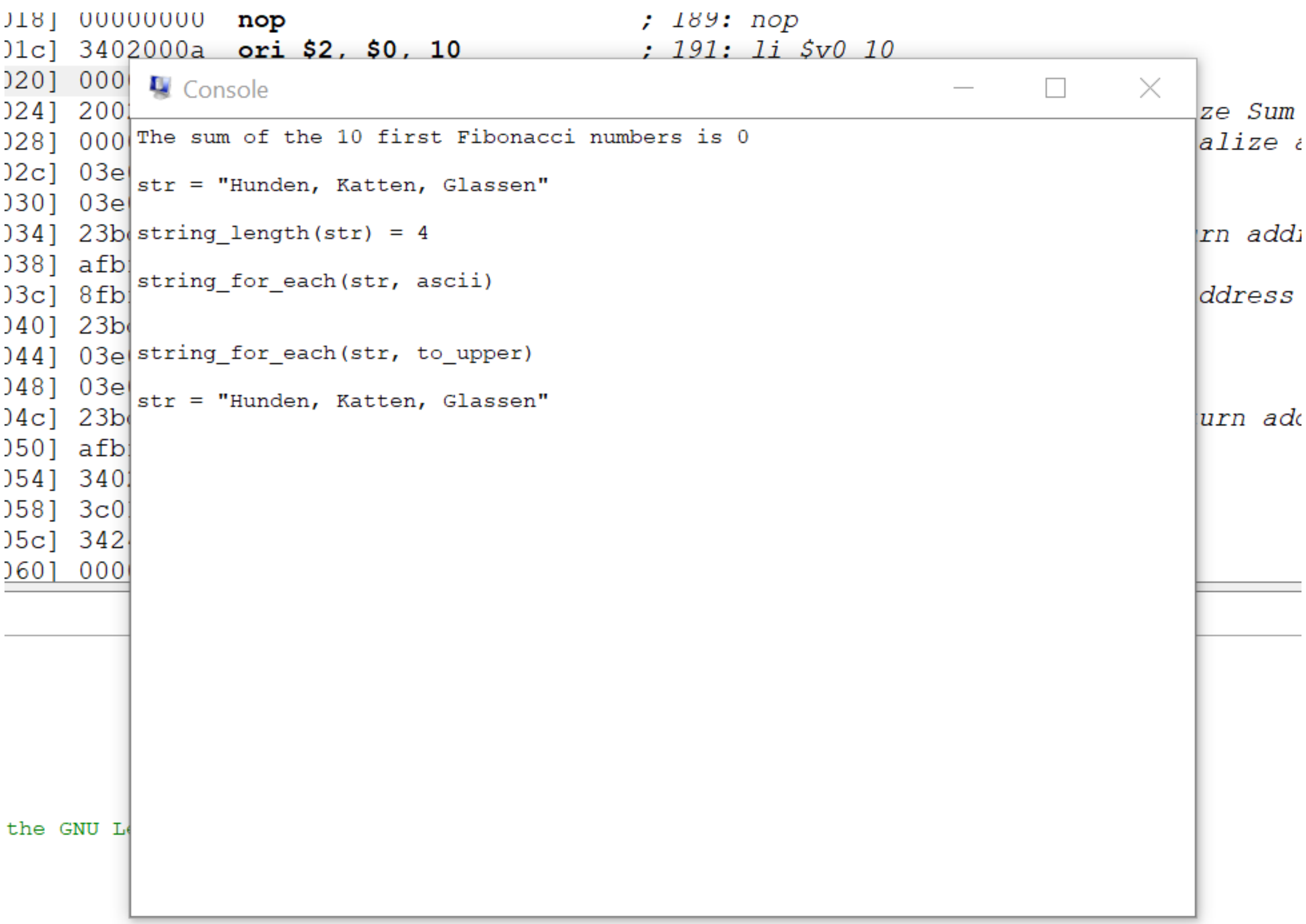
To load and run the program using the CMD, enter 'spim -file FILENAME' to load a particular assembly code file, with FILENAME corresponding to the path of the file.

You should get something similar to the following text output:

```
SPIM Version 8.0 of January 8, 2010

Copyright 1990-2010, James R. Larus.

All Rights Reserved.

See the file README for a full copyright notice.

Loaded: /usr/lib/spim/exceptions.s

The sum of the 10 first Fibonacci numbers is 0

str = "Hunden, Katten, Glassen"

string_length(str) = 4

string_for_each(str, ascii)

string_for_each(str, to_upper)

str = "Hunden, Katten, Glassen"
```

To load the program using the GUI, click on ‘File’ in the top left, and choose load file. To run the program, click on the run button (on Windows you can also press F5). When you run the program the following popup appears:

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



```
]18] 00000000  nop                        ; 189: nop
]1c] 3402000a  ori $2, $0, 10             ; 191: li $v0 10
]20] 000
]24] 200                                                              ze Sum
]28] 000                                                              alize a
]2c] 03e
]30] 03e
]34] 23b                                                              rn addi
]38] afb
]3c] 8fb                                                              ddress
]40] 23b
]44] 03e
]48] 03e
]4c] 23b                                                              urn add
]50] afb
]54] 340
]58] 3c0
]5c] 342
]60] 000

the GNU L
```

```
Console

The sum of the 10 first Fibonacci numbers is 0

str = "Hunden, Katten, Glassen"

string_length(str) = 4

string_for_each(str, ascii)


string_for_each(str, to_upper)

str = "Hunden, Katten, Glassen"
```

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, except for task five and the bonus task.

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

```
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
The sum of the 10 first Fibonacci numbers is 143

str = "Hunden, Katten, Glassen"

string_length(str) = 23

string_for_each(str, ascii)

Ascii('H') = 72
Ascii('u') = 117
Ascii('n') = 110
Ascii('d') = 100
Ascii('e') = 101
Ascii('n') = 110
Ascii(',') = 44
Ascii(' ') = 32
Ascii('K') = 75
Ascii('a') = 97
Ascii('t') = 116
Ascii('t') = 116
Ascii('e') = 101
Ascii('n') = 110
Ascii(',') = 44
Ascii(' ') = 32
Ascii('G') = 71
Ascii('l') = 108
Ascii('a') = 97
Ascii('s') = 115
Ascii('s') = 115
Ascii('e') = 101
Ascii('n') = 110

string_for_each(str, to_upper)

str = "HUNDEN, KATTEN, GLASSEN"

string_reverse

str = "NESSALG ,NETTAK ,NEDNUH"
```

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 *File then Reinitialize and load file*.

In the CMD, you must launch the program again for the updated file to be loaded.

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*.

A dialog box where you can choose *number of steps* will prompt you. 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.

To step using the CMD, open spim by typing the ‘spim’ command. Then load the file using the command ‘load “FILENAME”’ Note that the quotations marks are necessary for the function. Then you can type ‘step’ to step through the instructions one by one.

You should get an output similar to the following for the first step.

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                ; 183: lw $a0 0($sp)            # argc
```

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. Right click on an instruction and select ‘set breakpoint’ to set a breakpoint. To clear a breakpoint, right click on an instruction that currently has a breakpoint and select ‘clear breakpoint’.

Run the program.

SPIM will execute the program and stop after a while asking if you want to continue the execution, step or abort.

To add a breakpoint using the CMD, you have to insert a label into the code where you want the breakpoint to be. This label has already been inserted as ‘DBG’. 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

To run until the breakpoint, open SPIM, load the file. And type the following instruction ‘breakpoint DBG’ and type ‘run’ to run the program until the breakpoint is reached.

You should get something similar to the following:

```
The sum of the 10 first Fibonacci numbers is Breakpoint encountered at 0x00400024
```

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 then 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 **ascii** 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.

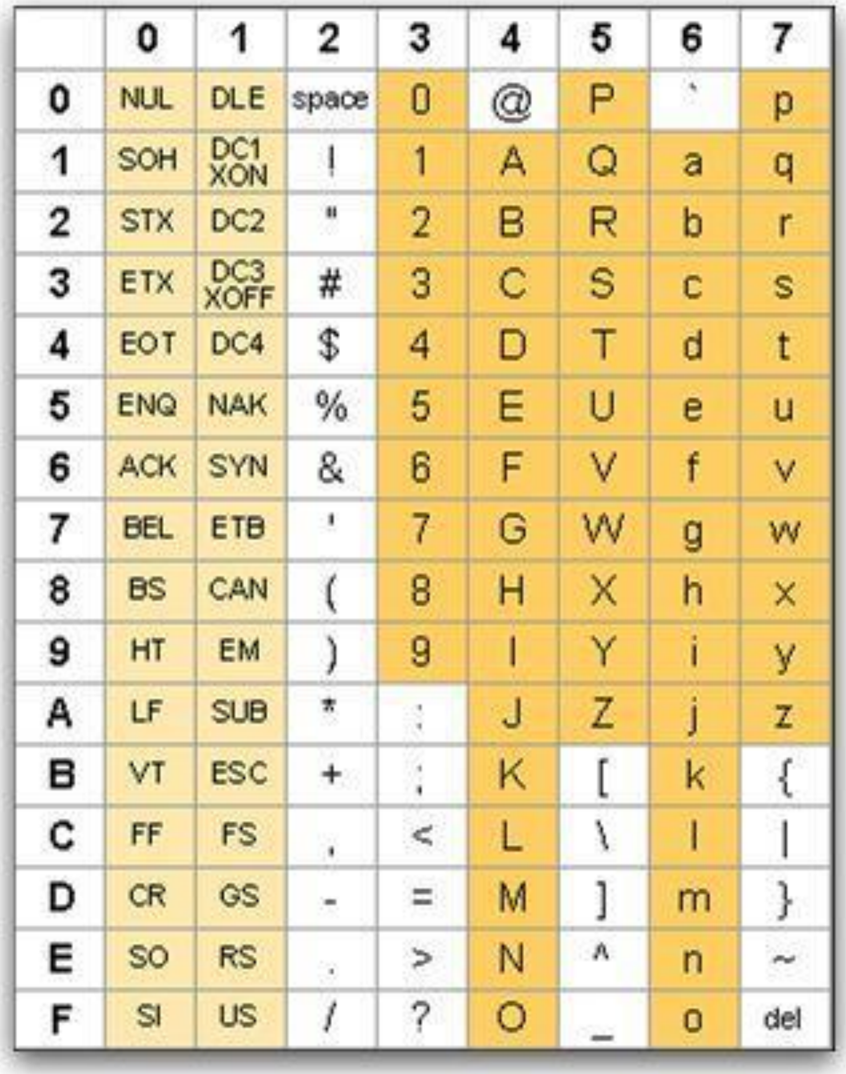
HINT: Have a look at the ASCII table below... What is the difference between a lower case 'a' and the upper case 'A'? When do you need to change the character and when not? (To read the table, you can see that ‘A’ = 0x41 and ‘a’ = 0x61.)

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?

CHALLENGE TASK: camelcase

Note: The challenge task is optional and it is for people looking for more of a challenging problem. It is not graded.

Write a new subroutine called **camelcase** from scratch, that takes a string as input, and capitalizes only every other letter, setting the rest to be lower case.

Examples:

camelcase(tree) TrEe

camelcase(verylongword) VeRyLoNgWoRd

camelcase(AmbusH) AmBuSh

camelcase(aMbUsH) AmBuSh

camelcase(UNIVERSITY) UnIvErSiTy

The 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 both call the function, and print out the results on the console.

Hint: take a look at how the previous functions were called within the **main** *function. Make sure to give a string as input argument.*

Grading and Evaluation:

The lab is graded as either pass or fail. You have to upload your code to Studium before the deadline. For evaluation, you will have a session with a TA at designated time slot on Zoom. You will have to select a time slot (more information on time slot sign up later through email). During evaluation, the TA will ask you to run the code and then ask you to explain some sections in the code. You should be able to explain your code and underlying assumptions and logic behind the code.