uc3m | Universidad Carlos III de Madrid

Degree in Applied Mathematics and Computing

Computer Structure

2022-2023

Assignment 1

Introduction to Assembly Programming

Ilan Halioua Molinero | 100472908
100472908@alumnos.uc3m.es

Alejandro Fernández Paniagua | 100473072
100473072@alumnos.uc3m.es

Both: Group 121

TABLE OF CONTENTS

1. EXERCISE 1

1.1. Behavior of the program, design decisions and pseudocode.

This exercise consists of coding a string compare function (where 2 strings will be compared). Its inputs are the address of the first byte of str1 and the address of the first byte of str2 which are loaded in a0 and a1 respectively. The output (stored in a0) will be one of the following possibilities: *-1* on error (Address of A or B is NULL), *0* if the strings introduced are different and *1* if they are the same.

In order to discard the errors from the beginning, we have done 2 *if statements.* The first one makes the function return *-1* if **t3** (the temporal register that stores the first string) is equal to zero. The second one makes the function return *-1* if **t4** (the temporal register that stores the second string) is equal to zero. If none of these conditions are satisfied, then the program continues.

The idea from this point was to set **a0** to *1* (the output of the function), i.e. input strings are the same string and then check iteratively through a loop each of the letters. Inside the i-th iteration, when the i-th letters of both differ, then set **a0** to *0* and return (*jr ra*). In this way, if throughout the loop no letter differences are found, then **a0** would not be changed and would stay as a 1(equal). At this stage yet, both strings could still be different (maybe they have different lengths but the one with shorter size shares the same letters as the longer one). Thus, to determine their equivalence, we will verify that they are the same length, which will be the case if their last letter after finishing the loop is the 0-letter ie: 00000000.

The pseudocode for the loop that checks letter by letter if there is a different one is the following:

```
While the letter of the first string is different from zero:   # (Could be of the str2)
.
.       If letter number 'k' checked is equal in both strings:
.       .
.       .       - Go to address of next letter (for both strings)
.       .       - Load the corresponding letter
.       .       - Go to check again the condition of the while loop
.       .
.       Else:           # different letter is found
.       .
.       .       Return a0 -> 0
.       .
.       End
.
End
```

*Continues in next page

If actual letter of string 1 is equal to the one of string 2

.

. Return (*jr ra*)

.

Else (If actual 'letter' of one of the strings is *0* and the other one is different from 0 (this string has more letters)): ## strings end with */0*

.

. Return a0 -> 0

.

End

1.2. Test battery table.

Input Data:	Test description:	Expected Output:	Actual Output:
"hello" / "hello"	Exactly same strings	1	1
"help" / "hell"	Different strings of same sizes (number of letters)	0	0
"hack" / "hacking"	Strings of different sizes (number of letters)	0	0
" run" / "run"	SPACE before one of the strings (All though they mean the same, they are not the same string)	0	0
"test" / "Test"	Strings that differ on the case of their letters (All though they mean the same, they are not the same string)	0	0
"" / "seven"	Error: Address of A is NULL	-1	-1
"ten" / ""	Error: Address of B is NULL	-1	-1
"" / ""	Error: Address of A and B are NULL	-1	-1

2. EXERCISE 2

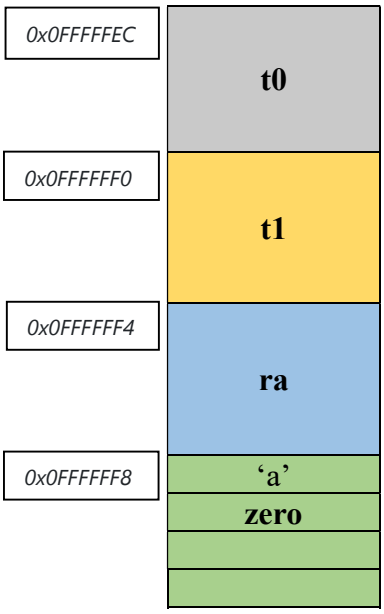
2.1. Behavior of the program, design decisions and pseudocode.

This exercise consists of coding a function that prints the number of cycles it takes to call *string_compare* and compare the string password with a certain letter. Its inputs are the address of the first letter of the password and the address of the possible key which are loaded in a0 and a1 respectively. There is no output as it just prints the number of cycles.

We have chosen to store in stack the following contents:

- A word for the strings. At the beginning: "a" ('a' followed by zero). At the end: "z" ('z' followed by zero). For this, the first byte of **t4** (*t4 = 'a' initially*) and **x0** (zero) are stored in this word at the stack.
- Return address (**ra**) of the call of *study_energy* → 1 word (4 bytes)
- Content of register **t0**: Iteration variable for the loop (i = from 0 to 26) → 1 word (4 bytes)
- Content of register **t1**: 25 (when t0 > t1 (*i > 25*) end loop) → 1 word (4 bytes)

Simplified Stack Idea



The idea behind storing in stack **ra**, is that each time the function *string_compare* is called inside *study_energy*, the original content of **ra** (the return address to the *study_energy* call) is lost, so when *study_energy* ends, if you jump to **ra**, it will go to the next line of the call to *string_compare* instead of *study_energy*, causing an infinite loop error.

So, we first store **ra** content in the stack, and when *study_energy* ends, we load it back to the **ra** register and *jr ra* (resetting **sp** to its original address afterwards).

The content of registers **t0** and **t1** are stored in stack as their content is modified on *string_compare*, and we don't want their values to be lost after calling this function.
Hence, we store them inside the stack before the call, and load them after too.

If we add the number of bytes needed in stack to store the necessary content, it gives 16 bytes (4 bytes * 4). Therefore, we start *study_energy* by leaving space in the stack for 16 bytes (*addi sp sp -16)*.

A general idea of how the function works is explained below:

Using the "a" we have stored inside the stack, we can easily obtain the next letter by adding 1 to the char 'a' (ASCII Code) stored at address sp+12 and input that as the address for the possible_key in the *string_compare* function.

In this way, we make sure we are taking as inputs: password together with "a","b",...,"z" in each respective iteration by summing 1 by 1 25 times.

In order to count the number of cycles required to call *string_compare* and compare the string password with a certain letter, we use the *rdcycle* command before and right after calling *string_compare*. By subtracting the cycles before the call by the ones right after, we assure to have the exact number of cycles taken by the processor to execute the string_compare function call.

Further details are presented in this pseudocode:

- Make a space of 16 bytes on the stack.
- Let **t4** = 'a'
- Store at bottom of stack a word with byte3: content of t4, and byte2: zero value

- Let **a0** = address of password

- Store a second word in stack, on top of the previous one that stores **ra** (return address of the call of *study_energy*)

- Let **t0** = 0 ('i' for the loop [from 0 to 26])
- Let **t1** = 25 (limit for t0 → end loop)

- Let **a1** = address where the possible keys are stored (address of byte3 of bottom word at stack, which contains the ASCII of character 'a')

While (**t0** < **t1**): # (*i* < 25)
.
. - Store **t0** at stack (a new word in top of stack)
. - Store **t1** at stack (second word in stack)
.
. - Store cycles of *string_compare(*password*,* current poss_key*)* in **t3**
.
. - print: letter (a,b,…,z) → ':' → space (' ') → # cycles → '\n' (new line)
.
. - a0 = address of password (The value of a0 was lost due to the prints)
.
. - Load top word of the stack to **t0** (the *i*)
. - Load second word of the stack to **t1** (the limit of *i (25)*)
.
. - Load first byte of the bottom word at stack to **t4** (the character of the current string)
. - **t4** = **t4** + 1 (example: from 'a' → 'b' ~ ASCII Code)
. - Store content of **t4** to byte3 of the bottom word at stack (the character of the next string)
.
. - **t0** = **t0** + 1 (*i*++)
.
End:

- Load **ra** (return address of *study_energy* call) from stack
- Restore stack pointer to original position (*addi sp sp 16*)
- Return (*jr ra*)

2.2. Test battery table.

Input Data: (“password” / “poss_key”)	**Test description:**	**Expected Output:** **printed cycles*	**Actual Output:** **printed cycles*
(“run” / “a”)	All wrong keys except “r”	13	13
(“run” / “b”)		13	13
…		…	…
(“run” / “r”)		20	20
…		…	…
(“run” / “z”)		13	13
(“” / “a”)	Address of password is NULL	11	11
…		…	…
(“” / “z”)		11	11

3. EXERCISE 3

3.1. Behavior of the program, design decisions and pseudocode.

Exercise 3 wraps up the ideas that underlie the previous exercises proposed, through the design of a void function (no output) named attack that receives as input two sources of data: the starting byte address of a string of length 8 called password and another one for a string called dummy. The name used as one of the inputs together with the one of the function already gives us an idea of what to expect from it.

The objective of the exercise is based on developing code that will crack an 8-Byte password string by exploiting the information given by the cycle counting when calling the string_compare function. Note that this function counts with the password already to be able to use the command ***rdcycle*** together with the call of the *string_compare* function to obtain the number of cycles.

In reality, attackers use **timing and signal measurements** to analogously get the "cost" of these calls and find the anomaly they look for. What is an anomaly here though? Why can the cycles be used against the processor to crack the password? Well, in the case the processor verifies the authenticity of a string to be the password (stored in memory at some point in the past), using the *string_compare* function, the attackers could take on a better approach to guess it instead of the brute force one (which will consist of testing with all possible 8 character permutations: 27^8 different ones) that might be computationally unaffordable. This approach only works whenever the algorithm that defines the function *string_compare* is programmed in such a way that for all the strings of a fixed length when compared to the password take up more or less cycles, depending on whenever the guess is a closer one or not. To understand this, it is better to view it with an example.

Example:

Let password = "aefghtew"
On the way the function *string_compare* is programmed, one can see that
#Cycles to perform *string_compare*(password,"a") : 20
#Cycles to perform *string_compare*(password,"b") : 13

Notice that both strings are different from the string *password,* but since the comparison is done character by character, for the first case, since the first character of *password* coincides with the first one (and only one) from the string "a", the computer takes the next step to the second character to then discard that they are equal. Opposite to it, with the string "b", equality is declined earlier since they differ right at the beginning. In fact, all strings of one character different from "a" will share the same cycles since the inequality is spotted at the same time for all. And this is the approach that an attacker could take, extending the idea until the last (8th) character is found. In this way, the attacker can guess the password by chaining the right character guess for each place in the string. This reduces the 27^8 different guesses to only having to take 27*8 guesses in the worst case.

To sum up, the algorithm for the *attack* function, that defines how a side channel attack works is the following:

Initialize dummy = "_____"

For i = 1 : length(password)
.
. Let dummy[i] = 'a'
. Let t5 = #Cycles(*string_compare*(password,dummy))
.
. Let dummy[i] = 'b'
. Let t6 = #Cycles(*string_compare*(password,dummy))
.
. While t5 == t6
. #both characters have same cycle value and as explained in the example above and therefore the i-th character of password is yet unknown
.
. . Get the next one (using Ascii) with highest cycle value
. . Let dummy[i]= dummy[i] +1
. . Modify t6 = #Cycles(*string_compare*(password,dummy))
. .
. End
.
. If highest == t5 #then the i-th character of the password is 'a' so:
. . dummy[i] = 'a'
. End
. #otherwise, we already have stored at dummy[i] the right character
. #Now that the i-th character has been guessed, if i<8, we look for the i+1-th
. #Increment i
.
. ++i
.
End

3.2. Test battery table.

Input Data:	**Test description:**	**Expected Output:** **guess inside dummy[]*	**Actual Output:** ** guess inside dummy[]*
"acbcaakl" / "aaaaaaaa"	Crack password of 8-chars: "acbcaakl"	acbcaakl	acbcaakl
"zklm" / "aaaaaaaa"	Crack password of less than 8-chars: "zklm"	zklm	zklm
"" / "aaaaaaaa"	Address of password is NULL (Error)	0	0

4. EXERCISE 4

4.1. *string_compare* function implementation to avoid Side Chanel Attacks.

(View answer at ***exercise4.txt)***

5. CONCLUSION

5.1. Problems encountered.

Throughout, the process of coding the three desired functions, several problems were encountered. One of the most important ones, came from a misconception of the stack pointer functioning. We had the idea that in order to not have a corrupted stack pointer, each time that we left space for storing a byte or a word in stack, we had to reset it to original position. This was a fatal error as by doing it, without knowing, we were erasing everything we had stored on that space in stack. Once we realized, many compilation errors disappeared as now we were letting the necessary space left in stack at the beginning, and just resetting **sp** at the very end (when stack was not necessary anymore).

Furthermore, for some time we were also stuck for having infinite loop errors. This came from the fact that at a first glance we did not realize that many of the content stored in temporal registers inside a function, were lost when calling another (also happened to **ra**'s original content). This made that for example the iteration counter ('i') instead of increasing constantly until its limit to end the loop, it changed completely each time the function was called. This was solved by storing the content of the registers that were going to loose its original value at stack, and after the function call, when the original values of this registers were necessary again, they were loaded and modified.

5.2. Conclusions.

As a first taste of what a processor and assembly language is like, we have definitely learned and extended our knowledge about it. Throughout the project elaboration, we have had the chance to collaborate and discuss on different ideas, which has enhanced our ability to design and implement various algorithms from a combination of non-personal but rather collective approaches (this led to the invention of efficient and useful algorithms). Although the level of programming inside the exercises was not too complicated (viewing it from a high level programming language perspective), assembly is a completely different world, acting as a double edged sword if not treated correctly. Lots of hours stuck at error debugging has clearly shown us that the crucial part for a successful assembly program development has been at the initial state, where it was essential to clear up all the ideas behind the uses of certain registers and the memory details, rather than on the "coding aspect". For this, sketches and pseudocodes have been our greatest ally, allowing us to visualize everything before getting our hands dirty with all the code after. In assembly programming, we really feel identified with the following saying: First plan, then review the plan and finally, execute!