Computer Structure
Lab 1 Report

Alex Zaharia & Skyler Edwards
100500012@alumnos.uc3m.es & 100500024@alumnos.uc3m.es
Group 121

Universidad Carlos III de Madrid

Table of Contents

Exercise 1: Pseudocode

```
int string_compare(char A[], char B[]){

Put 0 into t0, the temporary register;
Load the address of A into a1;
Load the address of B into a2;

if(no address is inputted into a1 or a2){
Jump to error case;
}

if(a1 and a2 both contain addresses){
Load first characters of the strings into the t3 and t4 registers using the addresses of the strings
previously stores in t1 and t2;
}

for(loop to iterate through arrays){
if(t1 != t2){
Jump to ‘different’ case;
}
Else if(t1 == t2){
Add 1 to the address of the strings to move to the next letter;
Load letters into registers t3 and t4;
When end of string is reached without error or differences, jump to ‘same’ case;
}}

Case 1 (same):
Return 1 into a0;

Case 2 (different):
Return 0 into a0;

Case 3 (error):
Return -1 into a0;
}
```

Exercise 1: Description and Test Cases

This function takes in two parameters - two strings A and B in the form of character arrays. The first part of the function compares the two, first checking if they both contain addresses. In the case where they do not, the program moves to error, which returns -1 into a0. The next set of instructions loads the first characters of A and B into t3 and t4, using addresses of the strings stored in t1 and t2. At this point in the program, since we have ensured that A and B have addresses are are not null, we can begin the loop. The loop iterates through the strings, checking to see if the individual characters are the same. If two characters in the same position differ, then the program jumps to the 'different' case, in which 0 is returned into a0, indicating that the strings are different. As the loop iterates through the characters, if they are the same then 1 is added to the address of the strings in order to move to the next letter. Then the letters are loaded into t3 and t4, and the loop continues. If the loop reaches the end of the strings, the program jumps to the 'same' case, in which 1 is returned into a0, indicating that the strings are the same.

Input data:	**Test description:**	**Expected output:**	**Actual output:**
strA: "computer", strB: "computer"	This is the case where the values for A and B are the same	1	1
strA: "green", strB: "rainy"	This is the case where the values for A and B are different, but have the same length	0	0
strA: "computer", strB: ""	This is the case where one of the strings is empty	-1	-1
strA: "green", strB: "blue"	This is the case where the strings have different values and different lengths	-1	-1

Exercise 2: Pseudocode

Refer to Exercise 1 for string_compare function

```
void study_energy(char[] password){

Make space in stack to store return address;
Store password into a1;
Store string into a1;

for(loops until t6 == t5 (letter: ‘{‘ → The ascii value after ‘z’)) {
Print current letter being processed
Print ‘: “ from .data
int var = number of cycles currently completed;
Store var into temporary register;
Call the string_compare function;
string_compare.get();
Add to number of cycles;
Print letter followed by var;
Iterates to next letter in the alphabet
Print number of cycles
Print new line;
}

Load return address stored in the stack;
}
```

Exercise 2: Description and Test Cases

First, the string_compare function from Exercised 1 is used to check if the user's password is valid by comparing the password entered to the password stored in memory. Next, space is made in the stack to add and store a return address. The loop prints a char letter followed by the number of cycles needed by the processor to utilize the string_compare function. This number is tracked by the program as it uses the function, and stored in the t0 (temporary) register. A new line is printed and then the loop iterates through the entire alphabet, starting with 'a'. Finally, the return address is loaded into ra and stores it in the stack so that we are able to return to the main.

Input data:	Test description:	Expected output:	Actual output:
password: "password"	This test tests what happens when any string, the string "password" in this case, is inputted into the function study_energy. It compares each letter of the alphabet to the first letter of the inputted password and returns a list of every letter along with the number of cycles for each letter. A match is detected when a the comparison is true (one letter has a greater number of cycles than the rest of the alphabet)	a: 10 b: 10 c: 10 d: 10 e: 10 f: 10 g: 10 h: 10 i: 10 j: 10 k: 10 l: 10 m: 10 n: 10 o: 10 p: 17 q: 10 r: 10 s: 10 t: 10 u: 10 v: 10 w: 10 x: 10 y: 10 z: 10	a: 10 b: 10 c: 10 d: 10 e: 10 f: 10 g: 10 h: 10 i: 10 j: 10 k: 10 l: 10 m: 10 n: 10 o: 10 p: 17 q: 10 r: 10 s: 10 t: 10 u: 10 v: 10 w: 10 x: 10 y: 10 z: 10

password: “”	This test case tests what happens when nothing is inputted into the password parameter. Returns the same number of cycles for every letter since there is no letter to compare it to.	a: 10 b: 10 c: 10 d: 10 e: 10 f: 10 g: 10 h: 10 i: 10 j: 10 k: 10 l: 10 m: 10 n: 10 o: 10 p: 10 q: 10 r: 10 s: 10 t: 10 u: 10 v: 10 w: 10 x: 10 y: 10 z: 10	a: 10 b: 10 c: 10 d: 10 e: 10 f: 10 g: 10 h: 10 i: 10 j: 10 k: 10 l: 10 m: 10 n: 10 o: 10 p: 10 q: 10 r: 10 s: 10 t: 10 u: 10 v: 10 w: 10 x: 10 y: 10 z: 10

Exercise 3: Pseudocode

Refer to Exercise 1 for string_compare function

void attack(char password[], char dummy[]){

Make space in the stack to add the return address;
Load password into a1;
Load dummy into a2;

for(loop to calculate the number of cycles){
Use rdcycle to calculate the current number of cycles - store in t3;
Call string_compare function;
Store the number of cycles in t3;

Store current letter in t1 in case t3 contains the correct letter;
Iterate to the next letter;
Store the next letter back into the dummy variable at the same index;

Calculate the current number of cycles - store in t3;
Subtract t3 from t4 to get number of cycles and store value in t5;

if(the original letter has more cycles than the next letter (t3>t5)){
Go to next1;
}
Else if(t3 and t5 have the same number of cycles but aren't the letter(s) we are looking for){
Continue loop;
}

next1:
Store original letter back into dummy at the correct index;
Move to next2;

next 2:
Iterate to the next index of the password and dummy;
Move to done;

done:
Load ra into the return address stored into the stack so that we can return to the main;
}}

Exercise 3: Description and Test Cases

The bulk of this exercise follows a similar approach as was used to complete Exercise 2. We begin by passing the preset password and a dummy variable as parameters. The dummy variable will eventually be used to store the key that is discovered by the program after, in the worst case scenario, iterating through 27*8 possible letter combinations. To begin, the number of cycles used by each letter is calculated using the string_compare function. The program processes two characters at a time, tracking their number of cycles and comparing them. Once there is a character match, it iterates to the next index of the string until the function reaches '0', indicating the end of the string. This string is stored in the variable 'dummy' in memory.

Input data:	Test description:	Expected output:	Actual output:
password: "password" dummy: " "		"password"	"112, 97, 115, 115, 119, 111, 114, 100"

Exercise 4: Response

How should the string_compare function be implemented to avoid attacks of this type?

The most effective way to avoid attacks of this type would be to store the letters and number of cycles in random places in memory and store the memory addresses in a stack such that the data isn't lost. Doing so would make it more difficult to access the number of cycles since the stack would have to be accessed. This would also increase the amount of time it would take someone to perform a successful attack on the function. Another way to avoid attacks of this type, I would write the program in another an Object Oriented language such as Java or Python and implement an encryption algorithm such as SHA-256 or even a simpler one such as a Caesar Shift as this will deter any attacks on the functionality of the program. We say to implement this program in another language as it would be easier to implement an encryption algorithm in Java or Python (programs we are familiar with and know how to implement such an algorithm in). This may not be the most effective option since it doesn't use assembly and doesn't optimize the time it takes to

run a program (Java and python aren't as optimized since they are higher level languages and we aren't able to access everything the computer is doing in the background).

Conclusion

Overall, the exercises in the lab were useful to the learning process as it provided an opportunity to utilize the knowledge of assembly language and functions that were introduced during lectures in a practical, more applicable manner. While a bit difficult to decipher, each exercise tested a different skill set and prompted new ways of thinking and approaching problems. For example, finding a way to implement the number of cycles into Exercise 2 was confusing, as was attempting to provide proper parameters for the functions in the exercises. However, once figured out, it provided a deeper and more complex understanding of the task at hand. In the future, we think it would be wise to ensure that the material is fully understood before attempting to accomplish the lab while lacking a complete understanding of the fundamental concepts being tested. Another issue we ran into and weren’t able to resolve was in function ‘attack’: we couldn’t figure out how to store characters back into memory, only their hexadecimal values. This can be seen in the test cases for Exercise 3 and can be seen when running our program.