MIPS Assignment 1

Yacouba Bamba
Leomar Durán
Moussa Fofana
Tairou Ouro-Bawinay

ECE 4612

2020-09-30

Objectives of the assignment

1. Create a program that accepts a string and copies it into another buffer.

2. Create a program that accepts an integer from the user and prints its factorial.

3. Create a program that validates a Matlab expression, given the following criteria.

Error Check List

In the expression entered by users,

1. Check for allowed symbols
2. Check for space(s) between digits of a number
3. Check for uneven number of parentheses
4. Check for errors:
   - 4.1 (/
   - 4.2 (*
   - 4.3 ()
   - 4.4 No operator between operand and open parenthesis. Ex: 2(
   - 4.5 No operator between close parenthesis and operand. Ex:)2
   - 4.6 /)
   - 4.7 //
   - 4.8 /*
   - 4.9 +/
   - 4.10 +*
   - 4.11 +)
   - 4.12 -/
   - 4.13 -*
   - 4.14 -)
   - 4.15 **
   - 4.16 */
   - 4.17 *)
   - 4.18 */
   - 4.19 *)

Tools/equipment

We used the MARS MIPS Simulator 4.5 as our tool for analysis. The Data Segment window was very useful. For the last objective, breakpoints and the single step function in the Text Segment window and the Registers tab were especially important for debugging.

Analysis/algorithm/procedure

For objective 1, it was assumed that strings are null terminated.

The analysis involved using a test string for the input string and testing if the same string was found in the copy string.

To find the algorithm, we started by writing out the program in C first, then translating the result to MIPS. This was not too difficult because the program was for the most part a `for` loop in the `strcpy` procedure.

The procedure is as in Table 1.

Table 1. INPCPY *procedure.*

```
10 Display the prompt.
20 Accept a string from the console.
30 Set index to 0.
40 Find the character at index.
50 Copy the character at index over to the buffer at index.
60 If the character is null, exit the loop to line 90.
70 Increase the index by 1.
80 Repeat from line 40.
90 Repeat from line 10.
```

For objective 2, it was assumed that all values were 32 bits (4 bytes) including both the input and the answer. This allows for factorials of natural numbers from 0 to 12. It was also assumed that numbers are unsigned.

The analysis is to take the factorial of select numbers such as 12, 7 and 5, see if it computed the expected factorial.

This experiment was also a for loop, so finding the algorithm was not too difficult either.

The main procedure is in Table 2.

Table 2. The INPFCT *procedure.*

```
110 Display the prompt.
120 Accept an integer from the console.
130 Call the factorial procedure returning a result.
140 Display "Ans:", then the answer.
150 Repeat from line 110.
```

The factorial procedure is as in Table 3.

Table 3. The FACTORIAL *procedure.*

```
210 Start with f := 1.
220 Let n := argument $a0.
230 If n is zero, exit the loop to line 270.
240 Multiply f by n.
250 Decrease n by 1.
```

```
260 Repeat from line 230.
270 Return to the caller.
```

For objective 3, it was assumed that the Matlab expression can end on a newline character, and that an empty string was vacuously valid. It was also assumed that only the characters in the class ()*+= and the character - were operators, as well as that characters in the classes 0-9, a-z, and A-Z were operands, and that parentheses were not in any of these two classes.

The analysis is to try various expressions that would follow or break the rules. For example:

- (256-5)*39
- ((256-5)*39
- (256-5)*39)
- 256-5)*39
- (256-5*39
- 5&4
- ABC=3/4*(xyz+29)-10
- 5(22)
- (22)5
- (5)(22)
- (5)*(22)
- the different digraph combination errors in Error Check List.

This program is more complex. So to find the procedure, we used an iterative process documented in the changelog. We started with the simplest task, which was to throw out all invalid characters.

The procedure for that is in Table 4.

Table 4. The MATCHK *procedure.*

```
1010 Let k := 0.
1020 Get character at index k.
1030 If the character is a null terminator, or a newline, then go to line 1400.
1040 If the character is in the range '(' to '+', then go to line 1200.
1050 If the character is '-', then it is valid.
1060 If the character is in the range '/' to '9', then go to line 1200.
1070 If the character is '=', then it is valid.
1080 If the character is in the range 'A' to 'Z', then go to line 1200.
1090 If the character is in the range 'a' to 'z', then go to line 1200.
1100 Otherwise, go to line 1300.
1200 So this character is valid.
1210 Increase k by 1.
1220 Go to line 1020.
1300 So the entire string is invalid.
```

```
1310 Clear the valid flag.
1320 Return to caller.
1400 So the entire string is valid.
1410 Set the valid flag.
1420 Return to caller.
```

To check the ranges is a little more involved than in C for example. For example, line 1040 may be broken down to the steps shown in Table 5.

Table 5. The range '(' – '+' decomposed.

```
1043 If the character is less than '(', then go to 1300.
1047 If the character is less than ('+' + 1), then go to 1200.
```

It all has to be expressed in terms of the less than operation.

It followed that spaces were invalid characters, so a separate check for them was unnecessary.

In the second iteration, parentheses checking was implemented. This is done by keeping a counter `$t5` that is incremented on each open parenthesis and decremented on each close parenthesis.

To do that, the parentheses must each be treated as special characters, each with their own subroutines. And if counter `$t5` was nonzero at the end of the loop, then this meant that the number of parentheses are uneven, so the expression was invalid.

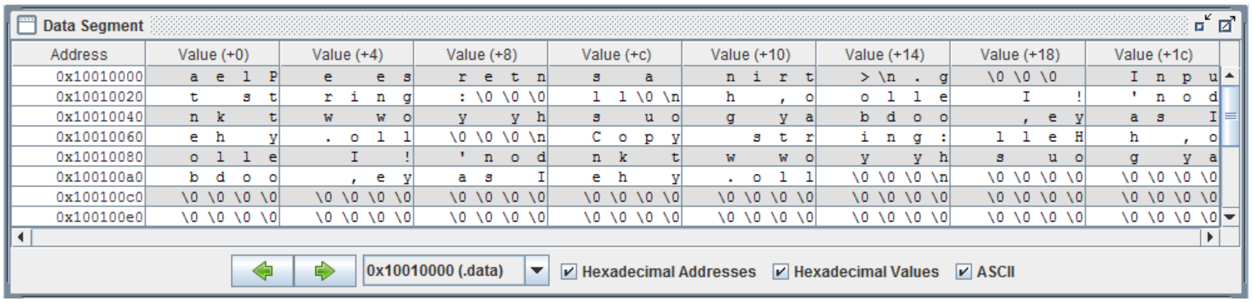
The case with "no operators since" errors, which require that there be operators between an operand and an opening parenthesis, as well as between a closing parenthesis and an operand, was effectively its own special task. A vector of flags was created `$t6`, with flags for "no operator since last operand" and "no operator since last closing parenthesis". When an operand is hit, the first flag is set, and it is cleared on the next operator. If the flag is still set when an opening parenthesis is hit, then this is an error. Likewise, for the second flag, it is set whenever a closing parenthesis is hit, and cleared on the next operator. However, if the second flag is still set when hitting an operand, this is an error that results in an invalid expression.

For the remaining task, which was to check for the diagraph errors, first, the character was initialized to `'\0'` before the loop started. Then, a copy of the current character `$t7` was maintained, being updated immediately before the current character was fetched. When one of the ending characters '/', '*', or ')' was hit, it would result in a test for the previous character, and if it was '(', '/', '+', '-' or '*', then this resulted in an invalid string.

This could have been done using flags to save on the last check which is a little more expensive in terms on instructions being performed, but this way seemed like the easier way at the time.

Test/Results/observations

For objective 1, the analysis was using a test string, pausing the simulation, and verifying that the copy string is identical through inspection.



Data Segment

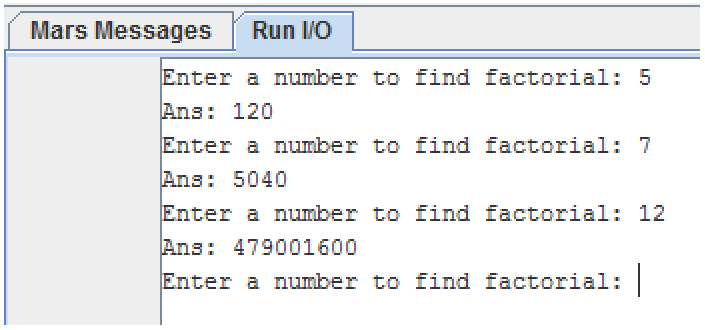
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	a e l P	e e s	r e t n	s a	n i r t	> \n . g	\0 \0 \0	I n p u
0x10010020	t s t	r i n g	: \0 \0 \0	l l \0 \n	h , o	o l l e	I !	' n o d
0x10010040	n k t	w w o	y y h	s u o	g y a	b d o o	, e y	a s I
0x10010060	e h y	. o l l	\0 \0 \0 \n	C o p y	s t r	i n g :	l l e H	h , o
0x10010080	o l l e	I !	' n o d	n k t	w w o	y y h	s u o	g y a
0x100100a0	b d o o	, e y	a s I	e h y	. o l l	\0 \0 \0 \n	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ✓ ASCII

Figure 1. Data segment result of objective 1.

Figure 1 shows the result of typing in the string "Hello, hello! I don't know why you say goodbye, I say hello." The input string from 0x1001002c to 0x10010068 is nearly identical to the copy string in 0x10010078 to 0x100100b4. The exception to this is that the input string has the character sequence "\n\0" for the first two characters instead of "He". However, if it were wrong, the copy string would not have come out correctly because it's copying each character. This may be something later that overwrote the first part of the string, possibly a register. To learn more, we will have to research how the memory is organized in MIPS.

For objective 2, testing is done by using several numbers. We will use 5, 7 and 12. For 5, we expect $5! = 5(4)3(2)1 = 120$; for 7, we expect $7! = 7(6)(5!) = 7(6)120 = 5{,}040$, and for 12, we expect $12! = 12(11)10(9)8(7!) = 12(11)10(9)8(5{,}040) = 479{,}001{,}600$.



```
Enter a number to find factorial: 5
Ans: 120
Enter a number to find factorial: 7
Ans: 5040
Enter a number to find factorial: 12
Ans: 479001600
Enter a number to find factorial:
```

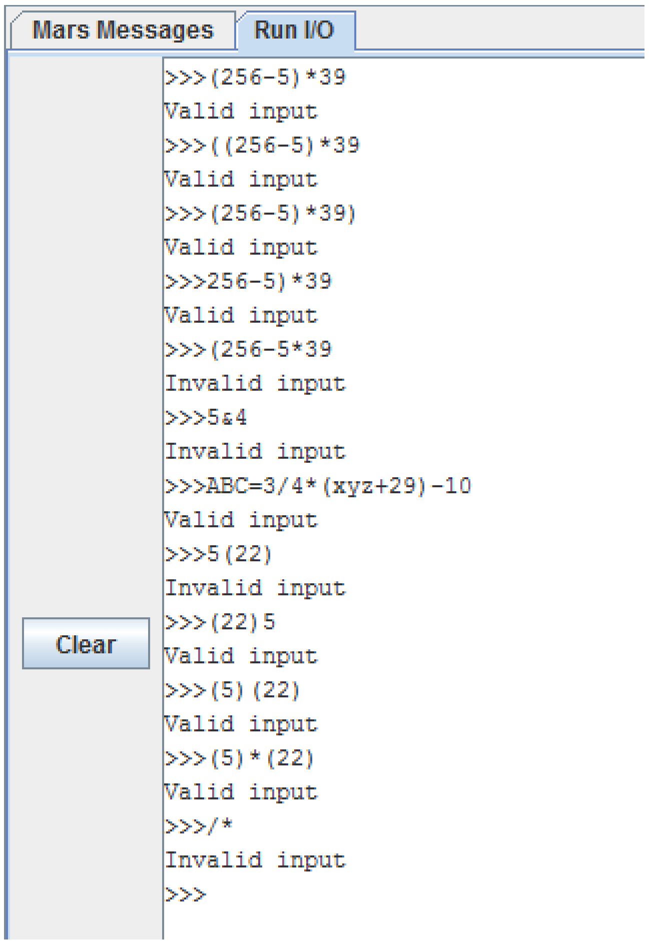
Figure 2. The results of objective 2, factorials of 5, 7, 12.

Figure 2 shows the exact expected results.

For objective 3, we will test with the following input.

- (256-5)*39
- ((256-5)*39
- (256-5)*39)
- 256-5)*39
- (256-5*39

- 5&4
- ABC=3/4*(xyz+29)-10
- 5(22)
- (22)5
- (5)(22)
- (5)*(22)
- the /* combination.



```
>>>(256-5)*39
Valid input
>>>((256-5)*39
Valid input
>>>(256-5)*39)
Valid input
>>>256-5)*39
Valid input
>>>(256-5*39
Invalid input
>>>5&4
Invalid input
>>>ABC=3/4*(xyz+29)-10
Valid input
>>>5(22)
Invalid input
>>>(22)5
Valid input
>>>(5)(22)
Valid input
>>>(5)*(22)
Valid input
>>>/*
Invalid input
>>>
```

Figure 3. Result of objective 3.

Figure 3 shows mostly the correct results. Something seemed to have broken in the last task. Namely, there seems to be an issue with the closing parenthesis. I traced my example, and it doesn't seem to go through the correct subroutines to return to the caller. This might be because we used a `jal` after another `jal` with no `jr` between.

Conclusions

Programming in MIPS is somewhat more complicated than the C code because you have to decompose the commands further than you would in C. We were able to make most of our requirements, but the stack is more involved in MIPS than in AVR for example, and we were not able to figure it out. For the first two objectives, it was not a problem because they involved simpler procedures with only one `jal` and `jr`.