Steven Reeves

CST 250

5/18/18

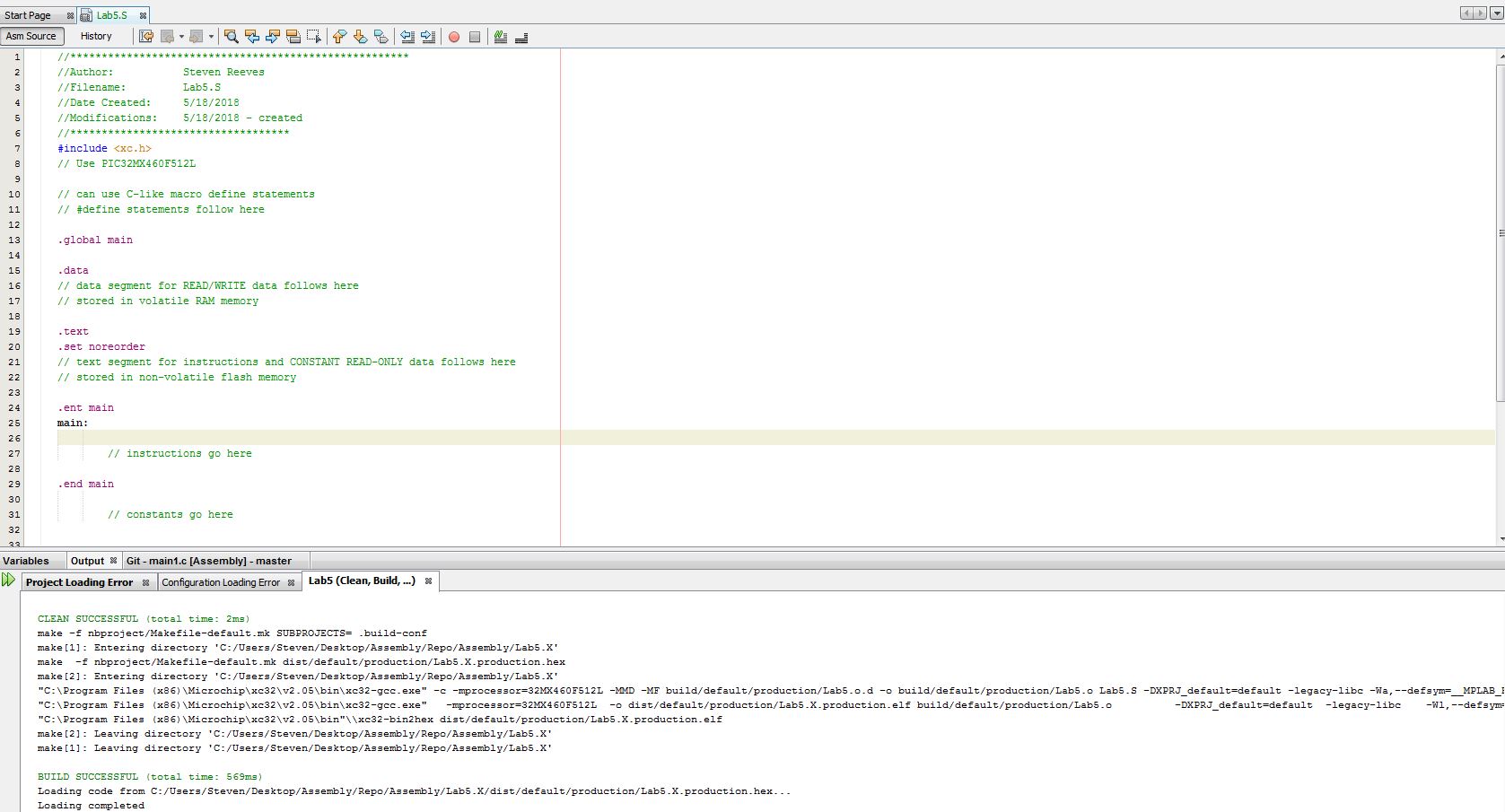
Lab 5 Report

**Introduction:**

The objective of this lab is to parse a string, evaluate the digits, keep a tally of different digits, and output this total. Creation and manipulation of the stack frame during function calls is another big objective here.

**Part 1:**

Part 1 of this lab had us review a different lab file, “CST250 - Lab 05”. After careful review, I first created a new .S project in the MPLAB IDE. After re-opening the IDE, I built and ran the project to make sure everything was working well.



After noting where the string to be parsed would be stored (bottom of the .text section), I followed the steps in the separate lab file to set up the UART output window.

- Open the Project Properties window

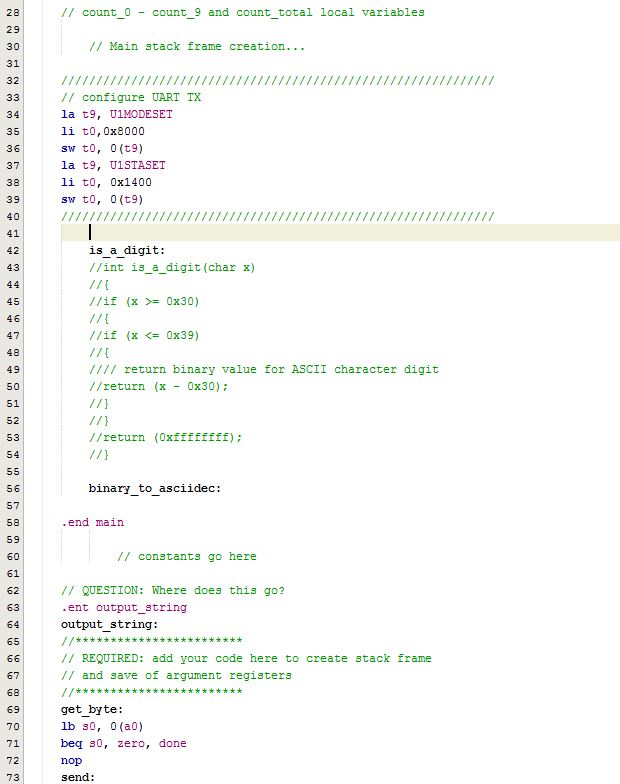
- Navigate to Conf:[default] -> Simulator -> Option categories

- Use dropdown menu to choose “Uart1 IO Options”

- Click the box to activate “Enable Uart 1 IO”

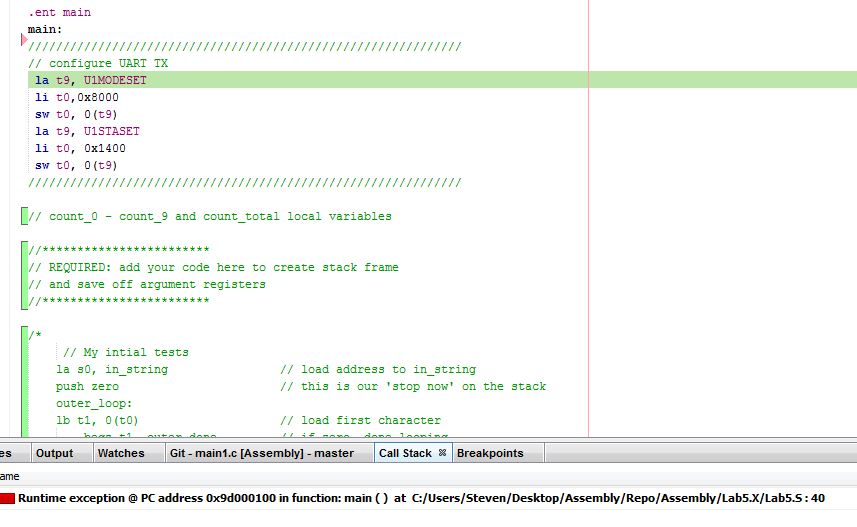
- Apply changes and exit window

I then added the verbatim code from the lab document for configuring the UART TX and the output\_string function. I also added the labels for is\_a\_digit and binary\_to\_asciidec. A clean and build showed that I was good to go thus far.

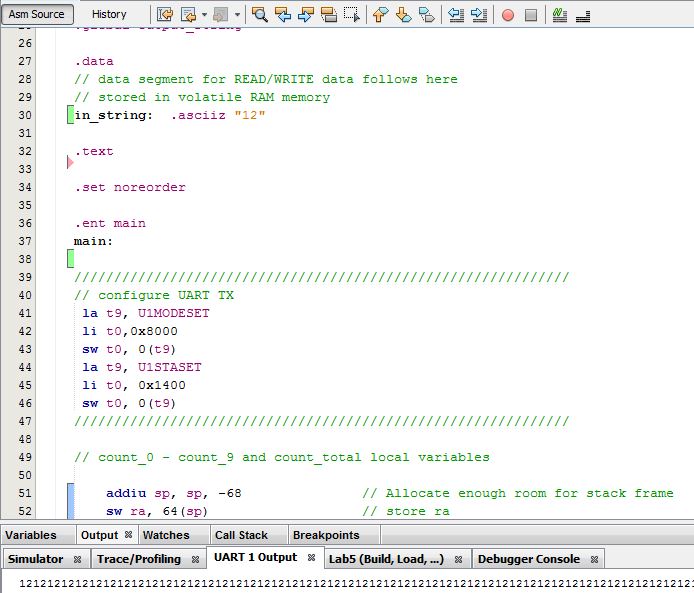


**Part 2:**

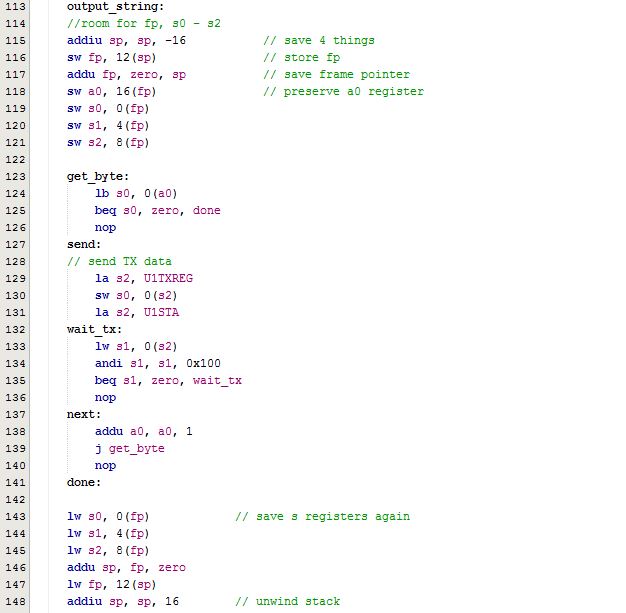
The first things I did in this part was setup my push and pop macros. I then made sure I could traverse through the string and send the characters out over UART. After writing some basic traversing logic, I realized I was having a runtime exception in the provided code.



After putting the variable in the .data segment, everything seemed to be working again.



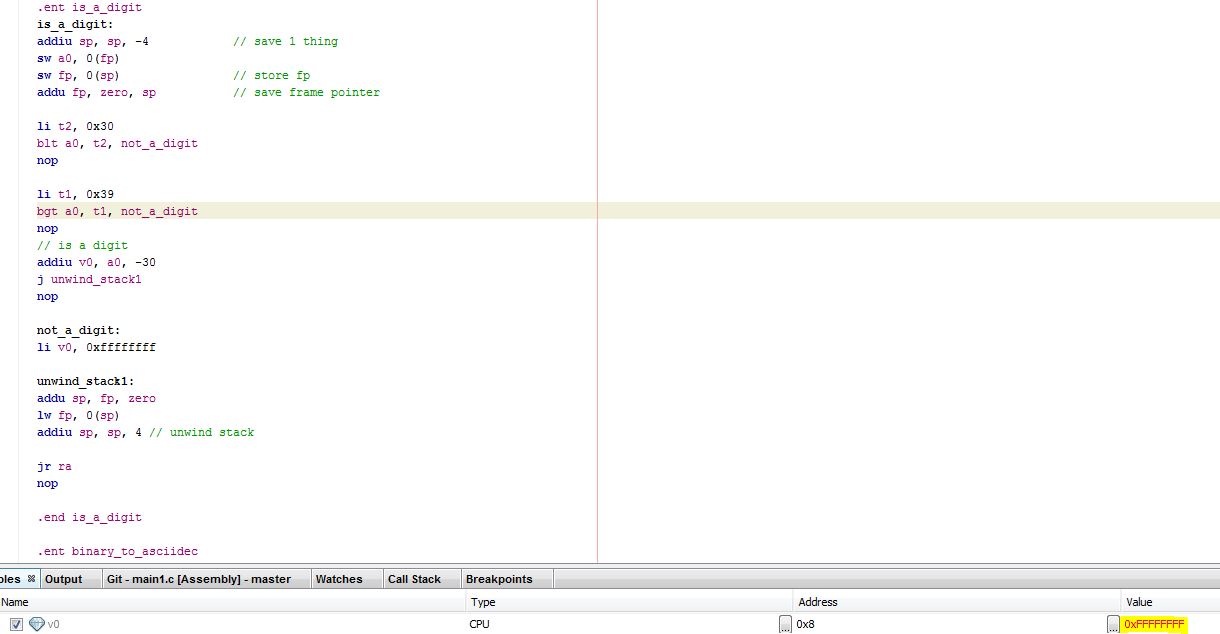
My next step was to copy over the code for output\_string and create a stack frame for it. I also included the code for unwinding the stack after the function was over.



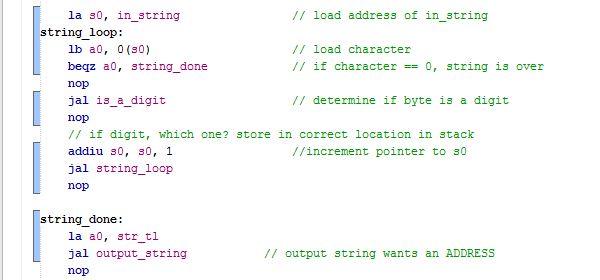
Running a simple string with this function did confirm its functionality.



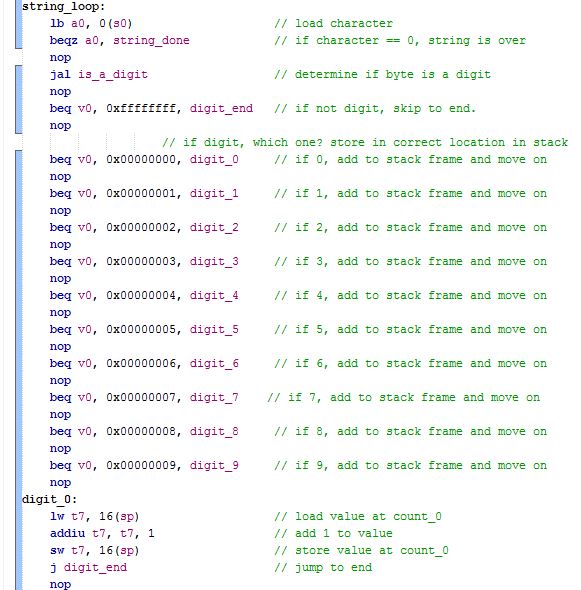
Next, I created a stack frame (with unwinding capability) and logic for the is\_a\_digit function. Testing the first character of “string here” shows that the function is working correctly by returning a value of 0xffffffff in register v0.



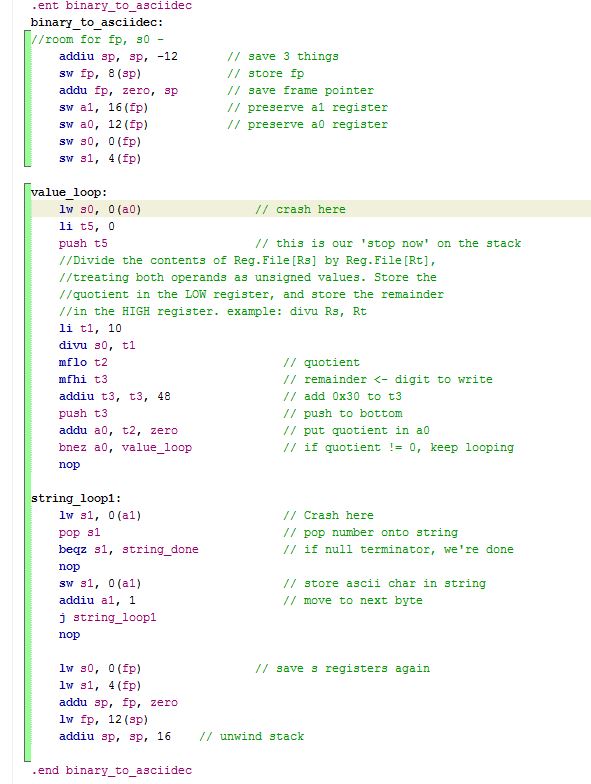
My next step was to make sure I can read through an entire string. I did this by implementing a loop. Walking through a breakpoint showed that my loop did in fact work for a test string.



My next step was to determine if the returned value of is\_a\_digit was a digit or not, and if so, which one? I did hit a bit of a wall here with subtracting the 0x30 from the a0 register. 0x30 = 48 decimal. After figuring that out, I did confirm that digit ‘1’ was working. I copied the code for that logic for each digit.

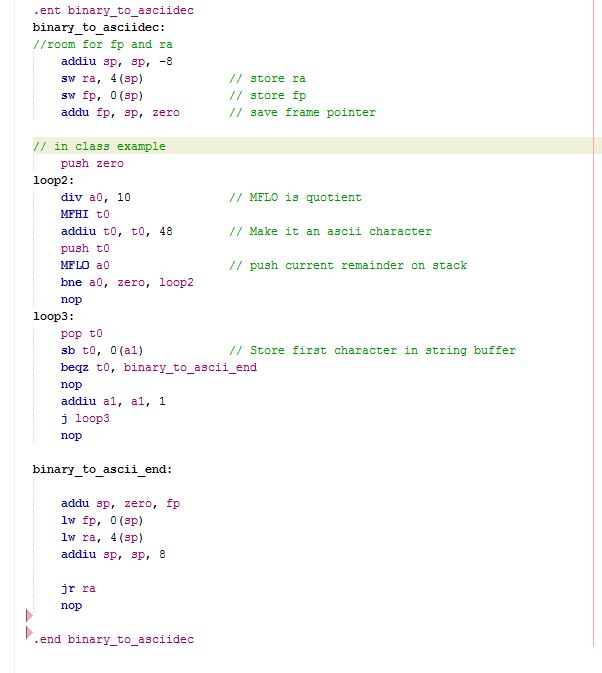


The next step I took was to implement the binary\_to\_asciidec function. Unfortunately, here is where I couldn’t get the logic to work.

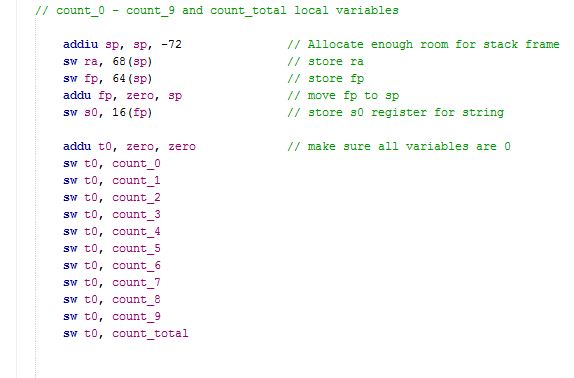
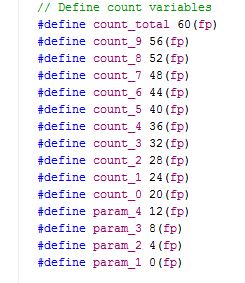


My basic idea was to divide the number by ten, push the remainder (adding 48 decimal) on to the stack, and divide the quotient again. The stack would then end up being just the number itself.

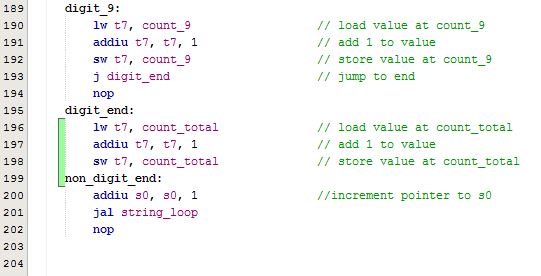
After an in-class example, I did get the binary\_toasciidec function working again.



Part 2 also had me set up my stack frame for main(). The allocation for that is below.



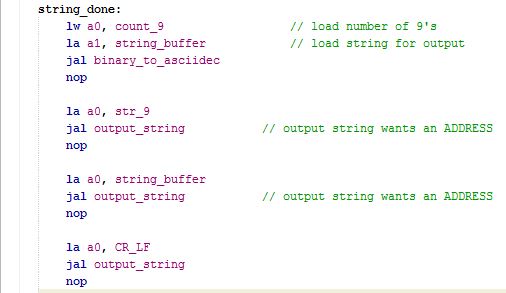
As for tallying the counts of different digits, I just branched off to an incrementing branch for each different digit. An example of the ‘9’ digit with the incrementing of the total number of digits is below.



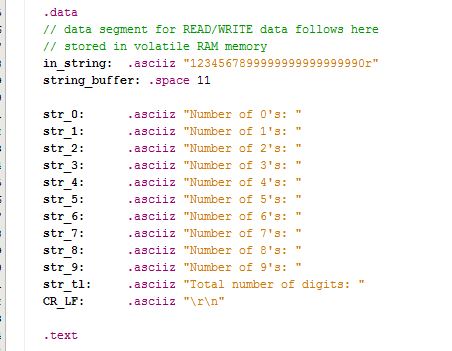
Basically, if the character in the string was a digit 0-9, an identical branch was made to increment the count of the digit. If the character wasn’t a digit, the s0 pointer was incremented and looping continues.

**Part 3:**

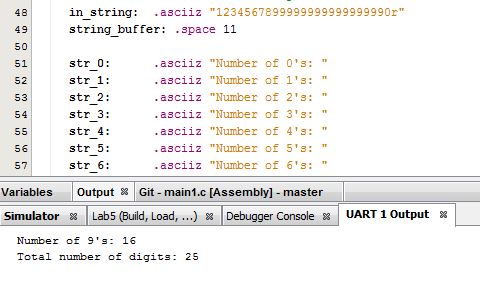
Output was first tested by using the binary\_to\_ascii function to turn the count of a digit into an ascii character. Output\_string was then used three times to construct the appropriate output. Example below.



Static strings were used for the first and last output.



Correct output was first verified below:



I next implemented a loop for ten iterations that prints out the message, count, and newline for each amount of digits. Pseudo-code and my actual code are below:

/\* output digit count messages including total \*/

for ( i = 0; i <= 10; i++ )

{

output\_string( i-message );

convert\_to\_ascii ( i-count );

output\_string( ascii-count );

output\_string( newline )

}

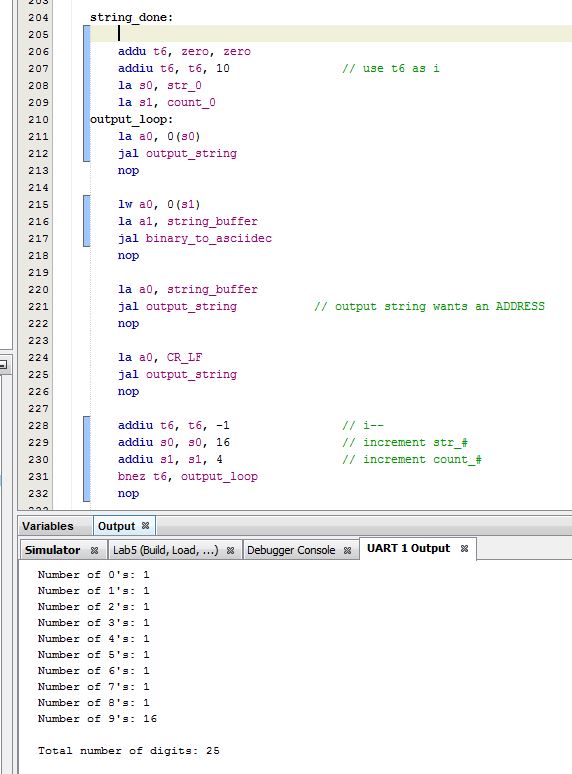
output\_string( newline ); // "double-space" before total count

output\_string( total-message );

convert\_to\_ascii ( total-count );

output\_string( ascii-total-count );

output\_string( newline );



The following string was used during testing and the amounts add up correctly: “Here is a test: 1234567899999999999999990r”

**Conclusion:**

This lab was again, the hardest I’ve encountered so far. Problems and solutions are included below.

- Where do functions go?

- Answered with example during lab time

- Runtime exception with UIMODESET

- Variables set before main after .text won’t work. Documentation provided was a bit misleading

- Sending byte to output string

- Looking into documentation requirements helped

- Decimal and hex conversion in addiu for asci

- Debugging and walking through watches helped here

- Binary to asci

- Lots of time trying different algorithms, in class example helped a lot.

- Printing out final output strings

- Logic in final loop took some work, but time and debugging helped.

Looking back at the objectives of this lab, I feel like I did complete the objectives:

- Parse a string of text -> shown in checking if each character is a digit

- Evaluate for digits -> shown in counting digits

- Keep tally of different digits -> shown in output

In summary, I learned a lot about allocating stack frames, string parsing, digit checking, and debugging through logic issues.