**MIPS Assignment 2**

Jonathan Isely

ECE-4612

Due: 10/12/23

**Objectives**

The objective of this assignment was to utilize the MARS IDE and MIPS instruction set to develop a tool for evaluating the validity of an expression. These expressions, which are input by the end-user, are checked for validity using the same rules that MATLAB uses. Outputs corresponding to the determination of validity are then displayed to the end-user and the program is to be cycled back to the start. The valid characters are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, /, \*, (, ), and spaces.

**Tools/Equipment**

The primary development tools utilized in this assignment are the MARS 4.5 IDE, which is used to compile MIPS assembly code. While not an explicit requirement, we also utilized Lucid Chart to develop a system diagram that helped to plan out our code structure and keep development on track.

**Analysis/Algorithms/Procedure**

The first step we took in completing the objective laid out in the assignment was to develop a system diagram, as seen in Figure 1.

A diagram of a company

Description automatically generated

Figure - System Diagram

While this was not a comprehensive diagram, and the ultimate structure of the code did change significantly, it still served as a great tool to develop an initial algorithm that solved some of the validity checks. As additional requirements were found, new subroutines were either added to the main loop or nested within a main subroutine. For our case main subroutines are defined as the subroutines that the main program loop calls at each step and is representative of a general requirement. Main subroutines are the ones shown being iterated through in the block diagram. The nested subroutines are called from within each main subroutine but are never called outside of the scope of that main subroutine. The nested subroutine’s primary function is to enable branching condition checks within each main subroutine.

One of the main benefits of this structure of code was that there were only a few registers that we had to keep track of throughout the execution of the main loop. Any additional registers existed only within the scope of their main subroutines and were reset upon exiting that main subroutine. Figure 2 shows the list of registers and highlights those that were used throughout the program, and thus blocked off from individual use within subroutines.

A screenshot of a table

Description automatically generated

Figure - Key Registers

Registers $a0 and $a1 were important for loading in syscall parameters and temporarily storing input, thus they were only used when necessary for those components. Register $t0 was a counter, which was reset before entering each new subroutine. Registers $t0 and $s0 were added together and stored in $t1, which allowed us to point to an individual character within the input string and increment through it. Finally, $t2 was where we loaded the byte corresponding to the address located in $t1. Other registers were used for items such as counting parentheses or operating as flags for certain conditions but were not common throughout the code and were therefore not annotated in Figure 2.

Utilizing this method of looping through the entire input and checking for validity for each subroutine may not be the most computationally efficient method, however it allowed for readable code and provided a measure of encapsulation that was helpful when debugging certain incorrect behaviors and errors. It also made it easier to add additional checks in to the main loop as more conditions of a valid expression were found.

**Test/Results/Observations**

Testing was done on a large variety of expressions using a systematic approach. However, attempting to test every permutation would have required an additional script to pipe in/generate inputs. Therefore, it is possible that there are still some errors that exist within the code that result in incorrect valid/invalid determinations. However, at the time of this writing there are none that we are currently aware of. If an error were to become apparent, debugging the issue is a relatively simple task. For a valid expression that shows as invalid we can set a breakpoint at the invalid output and step back one line at a time to see what triggered this behavior. For an invalid expression that shows as valid, we can determine where the invalid aspect is and then set a corresponding breakpoint in the main subroutine that should catch it. From there we can step through and see what check fails.

In terms of testing examples, Figures 3, 4, and 5 show three different valid expressions and their resultant output.

A white background with black text

Description automatically generated

Figure - Valid Expression

A white background with black text

Description automatically generated

Figure - Valid Expression

A black text on a white background

Description automatically generated

Figure - Valid Expression

Please note that for each of the valid expressions shown above, the program resets upon displaying the output and allows for another expression to be entered. These expressions were shown because they display a variety of different behaviors that are all valid within MATLAB. Figure 3 shows a variety of operators being used, while Figure 4 shows parentheses and spaces between operators and operands. Figure 5 shows another behavior that is in fact valid within MATLAB, and that is the ability to have multiple + or – operators in sequence.

Conversely, Figures 6, 7, and 8 show various invalid expressions and the resultant output.

A black text on a white background

Description automatically generated

Figure - Invalid Expression

A black and white text

Description automatically generated

Figure - Invalid Expression

A black and white text

Description automatically generated

Figure - Invalid Expression

As with the valid expressions, the invalid expressions reset upon outputting the resultant message and allow for additional input. Figure 6 provides an example of an expression with a mismatched number of parentheses. Meanwhile, Figure 7 is the result of the invalid combination of two operators. Finally, Figure 8 shows what happens when you have a space between two operands without an operator between them. As with the valid expressions, all these expressions can be copied into MATLAB with the same result.

**Conclusion**

This project provided an opportunity to grow in our knowledge of assembly and it was successful in that. Looking back at the developed algorithm there are some definite improvements that could be made. Namely, if we had developed a list of all valid/invalid conditions at the start, we could have made a more accurate flow chart that would have led to less need for debugging. It also would have resulted in more readable code. However, given the algorithm that was developed, it still allowed for relatively easy modification and addition of new validity checks. Regarding the results, they were in line with what was expected and, as of writing, no incorrect results have been observed. If they were to crop up, it would be a relatively simple task of debugging and/or adding in a new main subroutine as mentioned previously.