CMPE230, Assignment 2

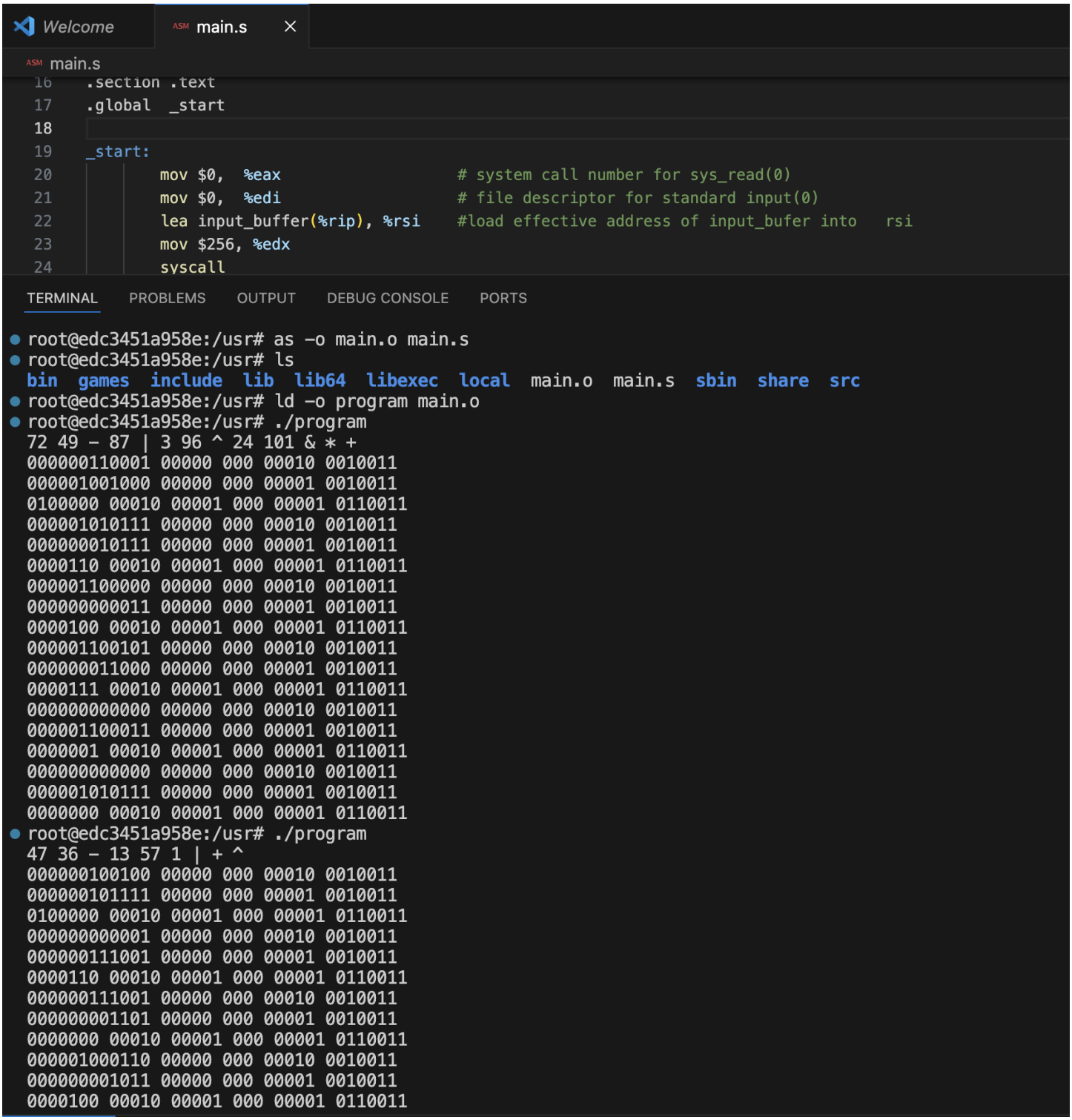
This is the documentation paper for the CMPE230 class, 2nd assignment. The purpose of the program that we have implemented is a translator that maps a postfix mathematical expression to RISC-V assembly code. We have used the GNU-Assembly (AT&T syntax) language for this project.

The procedure of the program is very straightforward. After the compilation of the assembly file, our executable is ready to be run. The user is expected to write a postfix expression via the standard input and our executable correctly converts the post-fix expression to the RISC-V Assembly (32-bit) syntax. It translates the expression by evaluating the intermediate steps of the calculation and printing the operations RISC-V equivalent to the standard output.

When it came to our implementation, we again went with the simplest way we could think of. Since the most basic way of evaluating a postfix expression is by using a stack structure, we thought that we could use the internal stack of our machines. We pushed the numbers to the stack and when we saw an operation character in the input, we popped the last 2 elements from the stack and performed the operation. We also didn't forget to push the answer of the operation onto the top of the stack.

Since we are using the assembly language, our program works like this:
We wait for the user input and store it in a buffer once the user enters. After taking the input, we read it character by character by loading a single byte each time from the buffer until we reach the "\n" symbol which indicates the input is finished. Also, while reading each byte, we make the necessary comparisons which are if the character is an operator symbol (+, -, *, ^, &, |), if it is a blank space, or if it is a number. Since our numbers in the input can have multiple digits, we store the digit read in a register while reading the numbers until a blank space is reached. By doing so, we can multiply the current number in that register by 10 and add the recent read value, getting the real value of the input number correct every time. This method can be used to evaluate numbers which have multiple digits.

The most challenging thing in this assignment for us was keeping track of our stack. Since function calls are also pushed onto the stack, values in our registers were not stable and they kept changing out of our control. So, to overcome this issue we had 2 choices: Either follow the conventions of the registers tightly or don't make any function calls. We have chosen the 2nd one since we didn't want to cause any reliability issues. We added labels to lines that we wanted to call a function and returned afterward, which allowed us to make use of functions without actually calling them. We also had a few problems while printing the values in our registers since converting them to their ASCII values while iterating over their binary representation was a little challenging. We resolved that issue again by using the stack, dividing the value by 2, and pushing the remainder to the stack. Mapping to RISC-V was rather easy since we didn't have much variety in our operations. We have defined template strings in our .data section and print the necessary output depending on the operation.



This homework is done by Cengiz Bilal Sarı and Berkay Buğra Gök (2021400201 and 2021400258)