CMPE230 Project 2 - Postfix Translator

Bahadır Kuşcan, Yusuf Akdoğan

1) Purpose and Objectives

The aim of the project was to write an assembly program that interprets a single line of postfix expression involving decimal quantities and outputs the equivalent RISC-V 32-bit machine language instructions.

2) Implementation Process

After inspecting the project description in detail, we determined that we could set premade strings for different kinds of outputs. Only variable part in the outputs was the 12-bit binary representation of the number popped from the stack so we left that one out. We decided to process the input line character-by-character and we divided the code into blocks. We used jumps to different blocks for different processes (main input reading block, one block for each operation and a block for multi-digit number processing) . We evaded writing many functions since we needed to do lots of operations on the stack (which function calls also use) and we only wrote a function for getting the 12-bit binary representation of a number which didn't need stack operations. We implemented that function by iteratively shifting the number to right and using the and operator with 1 to get the least significant digit at each loop and construct the string binary representation in a buffer. We also evaded using registers like rax, rbx, rcx etc. and used r8 to r14 registers for data storage and processing in order to prevent unexpected behaviour. Here are the registers we used and their purposes:

%r8 : Storage of the first number popped from the stack after encountering an operator. Used as an operand to perform the operation.

%r9 : Storage of the second number popped from the stack after encountering an operator. Used as an operand to perform the operation and as a storage for the result to push it to the stack.

%r10 : Temporary storage for a number to get its 12-bit binary representation. This register is shifted right in a loop in the conversion function to get the next bit.

%r11 : Temporary storage for the number in %r10. This register is "and"ed with 1 to get the least significant bit.

%r12 : Storage of the current character of the input line that is being read during input processing. Used for comparisons.

%r13 : Used for reading multi-digit numbers. Every time after a new succesive digit is read, %r13 is multiplied by 10 and added with that digit to build the number.

%r14 : Stores the input line since %rsi changes during outputs.

3) Challenges

First challenge we encountered was to find a way to get the 12-bit binary representation of a number. After some brainstorming we were able to find a way that is described above.

Another challenge was we weren't cautious enough to evade using registers like eax and edx in the beginning. We needed to change the registers we used in some operations. Register size compatibility in operations was also a reason for us to change the registers we used.

In the beginning for some reason we didn't think of multi-digit numbers so we needed to change the way we processed the input. We needed to divide our program into more blocks.

4) Usage and Examples

User inputs a syntactically correct postfix expression. Each number encountered is pushed to the stack. When an operator is encountered, 2 numbers at the top of the stack are popped and used as operands, then the result is pushed to the stack. Every time an operator is encountered, operands are loaded to x1 and x2 registers of RISC-V assembly, the machine codes for loading the operands to those registers are outputted by our program. The machine code for RISC-V commands to perform the operation is also outputted.

Input 1:

2 3 + 4 5 + *

Output 1:

```
000000000011 00000 000 00010 0010011
000000000010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
 000000000101 00000 000 00010 0010011
 000000000100 00000 000 00001 0010011
 0000000 00010 00001 000 00001 0110011
 000000001001 00000 000 00010 0010011
 000000000101 00000 000 00001 0010011
 0000001 00010 00001 000 00001 0110011
```

Input 2:

2 3 1 ^ & 9 -

Output 2:

```
000000000001 00000 000 00010 0010011
 000000000011 00000 000 00001 0010011
 0000100 00010 00001 000 00001 0110011
 000000000010 00000 000 00010 0010011
 000000000010 00000 000 00001 0010011
 0000111 00010 00001 000 00001 0110011
 000000001001 00000 000 00010 0010011
 000000000010 00000 000 00001 0010011
 0100000 00010 00001 000 00001 0110011
```