CMPE 230 SPRING 2024 PROJECT 2 - POSTFIX TRANSLATOR REPORT

Berk Göktaş 2022400039
Bora Aydemir 2021400054

May 2024

1 Purpose

Our main purpose was to implement the project with a simple and plain concept to maintain the code's reusability and reduce the effort needed while implementing and optimizing the code. We had a well-thought-out, structured, and organized design process to achieve this.

2 Design

As stated in the purpose part, the design concept was crucial for the project's maintainability. Before starting to code, we thoroughly thought about the kind of implementation that would make our work simpler. Most of the thought process was on how to convert integers to binary and which registers to use in order to maintain a stable run flow. At first, we wanted to use a set of registers that didn't have a specific role in syscalls as we needed their addresses to stay constant. Then we designed our calls and loops for a smooth flow. We also introduced some output buffers and custom strings to ease our way through the printing process. We introduced some calls to reduce repetitive operations. After having a clear design process, we split the tasks and began implementing the project merging the changes we made on our code in a shared repository and effectively communicating with each other in key aspects that might affect the integrity of the project.

3 Implementation

We implemented reading the input character by character. We encountered some segmentation faults during that process. We used GDB debugger to detect bugs and effectively fixed them by examining the unwanted register behavior. So the first part implemented was the input reading. In parsing, we first cleared a register and used that register to accumulate the value so far, then used a loop multiplying that value with 10 and adding it to every other digit until a whitespace is encountered. When a character is neither digit nor whitespace we considered that an operation operator and checked which operator it was. We also introduced a condition check that ensures the program terminates when a newline character is encountered. After we made sure that the input was read correctly, we began to implement the operations. For the operations part, we arranged different labels for different operations. Then we introduced the RISCV machine code for those operations as strings in the data part as they are independent of the values. Then, we implemented pull-push routines and registers for the operations. We faced some issues there as some register addresses were affected by system calls. We aimed to use general-purpose registers to resolve that issue. As we were making calculations with 8-bit registers, we used 64-bit counterparts to push to the stack and pop from it. We ensured that the values were correct via GDB. The most challenging part was to convert the integer value to binary for push and pop operations and print them before the constant part of the string that was declared for the RISCV machine code for addi. To convert the integer value to binary, we start from the most significant bit and basically and all the binary digits of the integer with 1. To achieve this we need to shift 1 by 12, then 11, then 10, etc. for each digit. Instead of doing this, we start from 2048 which is 2^{11}. Then we shift 2048 right by one hence dividing it by 2 every step to skip to the next digit of the 12-bit number. Hence we have found the binary representation in the order of the most significant digit to the least. We append all of those digits to a buffer to print it before the addi string. As negative numbers are automatically represented by their 2's complement values in assembly, there was no need to implement them separately. We used a custom print call between the operations to make them look tidy.

4 Examples and How To Use The Program

After typing "make" into the console, the program will compile and the postfix_translator executable will be created. After running the executable, the postfix expressions will be fed into the console as input. Here are some examples:

```
root@6800fe629c77:~/homework2-2021400054-2022400039# make
as -o postfix_translator.o src/postfix_translator.s
ld -o postfix_translator postfix_translator.o
root@6800fe629c77:~/homework2-2021400054-2022400039# ./postfix_translator
3 4 + 5 6 7 8 ^ & | *
000000000100 00000 000 00010 0010011
000000000011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000001000 00000 000 00010 0010011
000000000111 00000 000 00001 0010011
0000100 00010 00001 000 00001 0110011
000000001111 00000 000 00010 0010011
000000000110 00000 000 00001 0010011
0000111 00010 00001 000 00001 0110011
000000000110 00000 000 00010 0010011
000000000101 00000 000 00001 0010011
0000110 00010 00001 000 00001 0110011
000000000111 00000 000 00010 0010011
000000000111 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
root@6800fe629c77:~/homework2-2021400054-2022400039# ./postfix_translator
85 67 33 - | 9 ^ 8 + 10 11 * &
000000100001 00000 000 00010 0010011
000001000011 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000100010 00000 000 00010 0010011
000001010101 00000 000 00001 0010011
0000110 00010 00001 000 00001 0110011
000000001001 00000 000 00010 0010011
000001110111 00000 000 00001 0010011
0000100 00010 00001 000 00001 0110011
000000001000 00000 000 00010 0010011
000001111110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000001011 00000 000 00010 0010011
000000001010 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
000001101110 00000 000 00010 0010011
000010000110 00000 000 00001 0010011
0000111 00010 00001 000 00001 0110011
```