Postfix Translator

About

Project 2 for CMPE230 - Spring 2024 Course.

An assembly program that reads a postfix expression from the user, evaluates it step by step and prints the RISC-V assembly code for each step.

For each operation between two numbers, the program prints the RISC-V assembly code that loads the values into the registers r1 and r2 and the corresponding operation between them.

The program supports the following operations:

- Addition
- Subtraction
- Multiplication
- Bitwise AND
- Bitwise OR
- Bitwise XOR

The program reads the postfix expression from the user and prints binary representation of RISC-V assembly instructions for each step.

Usage

`make` to create executable file from assembly. `./postfix_translator` to run the program.

`make run` to compile and run the program. The program will prompt for a postfix expression.

`make clean` to delete the executable file.

`make grade` to test the program with 4 test cases under `/test-cases`

`make debug` to debug the program via debugger GDB (GNU Debugger).

Example Input - Outputs

Input 1:

```
1 2 +
```

Output 1:

```
000000000010 00000 000 00010 0010011
000000000001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
```

Input 2:

```
42 54 + 13 - 4 *
```

Output 2:

```
000000110110 00000 000 00010 0010011
000000101010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000001101 00000 000 00010 0010011
000001100000 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000000100 00000 000 00010 0010011
000001010011 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
```

Input 3:

```
0 2047 - 1 - 3 *
```

Output 3:

```
011111111111 00000 000 00010 0010011
000000000000 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
100000000001 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000000011 00000 000 00010 0010011
100000000000 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
```

Input 4:

```
85 47 + 6 | 55 11 ^ 1 72 & * +
```

Output 4:

```
000000101111 00000 000 00010 0010011
000001010101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000110 00000 000 00010 0010011
000010000100 00000 000 00001 0010011
0000110 00010 00001 000 00001 0110011
000000001011 00000 000 00010 0010011
000000110111 00000 000 00001 0010011
0000100 00010 00001 000 00001 0110011
000001001000 00000 000 00010 0010011
000000000001 00000 000 00001 0010011
0000111 00010 00001 000 00001 0110011
000000000000 00000 000 00010 0010011
000000111100 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
000000000000 00000 000 00010 0010011
000010000110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
```

Input 5:

```
1 2 & 4 | 64 ^
```

Output 5:

```
000000000010 00000 000 00010 0010011
000000000001 00000 000 00001 0010011
0000111 00010 00001 000 00001 0110011
000000000100 00000 000 00010 0010011
000000000000 00000 000 00001 0010011
0000110 00010 00001 000 00001 0110011
000001000000 00000 000 00010 0010011
000000000100 00000 000 00001 0010011
0000100 00010 00001 000 00001 0110011
```

Input 6:

```
0 2047 - 1 - 4 +
```

Output 6:

```
011111111111 00000 000 00010 0010011
000000000000 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
```

```
100000000001 00000 000 00001 0010011
0100000 00010 00001 000 00001 0110011
000000000100 00000 000 00010 0010011
100000000000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
```

Input 7:

```
4 3 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
```

Output 7:

```
000000000011 00000 000 00010 0010011
000000000100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000010 00000 000 00010 0010011
000000000111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000001111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000010110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
```

```
000000010111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000011111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000100111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101011 00000 000 00001 0010011
```

```
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000101111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000110111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111110 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000000111111 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
```

```
000000000001 00000 000 00010 0010011
000001000000 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000001000001 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000001000010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000001000011 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000001 00000 000 00010 0010011
000001000100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000001100 00000 000 00010 0010011
000001000101 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
```

Program Structure

```
src/
-> postfix_translator.s
ascii.sh
Makefile
```

There is only one source code, namely `postfix_translator.s`. `ascii.sh` is a shell script to quickly get hex and dec values of a given number

---

Program Design and Rationale

The program is designed to read a postfix expression from the user and evaluate it step by step. The program uses a stack to store the numbers. They are pushed to the stack upon generating the decimal value from the byte-by-byte digits, and popped from the stack after encountering an operation.

The program reads the input character by character and decides what to do based on the character.

- If the character is a number, the program stores the number in a buffer. Until it gets into a space, it builds up a number value. It does so via a number_buffer that keeps track of the digits of a number.
- If the character is an operator, the program pops the necessary number of operands from the stack, performs the operation and pushes the result back to the stack. (It's important to note that the program increments the input buffer pointer by 2 to skip the space character for simplicity. In order to avoid any collisions in between data segments, a set of 2 hard-wired 0 bytes are places right after the input buffer)
- If the character is a space, the program converts the number in the buffer to a decimal value and pushes it to the stack. This is done by a loop that multiplies the existing value by 10 and adds the new digit to the value. (Note that the program assumes that it can only get into this state after a number)
- If the character is a newline, the program knows that the input is over and calls the syscall to exit the program.

Logging and Debugging

The program does not have an inherent logging/debugging mechanism. However, the program was run with `gdb` for debugging purposes. Additionally, when the program was in an immature state, the buffer values were printed to the console to check the correctness of the program at a given point.

Some Challenges

- One of the main challenges was to implement the decimal to binary conversion. The program uses a loop to convert the decimal number to binary. The program uses the `and` operation with 1 to get the rightmost bit of the number. The program then shifts the number to the right by 1 to get rid of the rightmost bit. The program repeats this process until the counter is zero.
- Another challenge was to implement the printing of the RISC-V assembly code for each step. The program uses the `print_func` function to print the output buffer to the console.
- Creating a generic procedure/label for operators was also a challenge. Because the routine makes use of popping the stack, using call/ret structure was simply not possible. Therefore, the program does not use a generic operator routine but instead copies the same routine for each operator.
- Having the awareness of clearing and resetting unused registers and buffers was quite crucial for the program to work correctly. Just because we had forgotten to clear the value in the address stored in the `number_buffer` (`%r13`), we had to deal with a bug that caused the program to mess up the decimal number values.
- The fact that the registers were used by the caller/calle routines of the GNU Assembly was a big challenge during the process of debugging. Therefore register selection was something to pay really attention to.

Contributors

- Eray YÜKLÜ - 2021400273
- Ümit Can EVLEKSİZ - 2020400114