Postfix Translator

Systems Programming

)

Project Overview

In this project, the aim is implement a GNU assembly language program that interprets a single line of postfix expression involving decimal quantities and outputs the equivalent RISC-V 32bit machine language instructions. This project will help you understand the mechanics of converting high-level operations into machine code, using the RISC-V architecture as a basis.

Introduction to RISC-V

RISC-V is an open-source instruction set architecture (ISA) that is based on established reduced instruction set computing (RISC) principles. Unlike proprietary ISAs, RISC-V can be freely used for any purpose, allowing hardware designers and software developers to create more customized and optimized computing systems. In this project, you will be utilizing the RISC-V 32-bit ISA, focusing specifically on I-type (Immediate) and R-type (Register) instructions, which are essential for performing arithmetic and logical operations directly on the processor.

Technical Specifications

Your program should convert the postfix expressions to the corresponding RISC-V machine language code. You must only use I-type and R-type instruction formats as outlined in the RISC-V user-level ISA manual, which you can reference here: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf. Additional resources on RISC-V programming can be found at:

https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture7.pdf.

Examples

Here are some sample inputs with the expected outputs that your program should generate:

input	output

2 3 + 4 5 + *	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	output

2 3 1 ˆ & 9 -	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

Handling Postfix Expressions

To process postfix expressions, your program should implement a stack-based approach:

- Number Handling: Push each numeric value onto the stack until an operator is encountered.
- Operator Encounter: Upon encountering an operator, pop the top two elements. Assign the first popped element to the x2 register and the second to the x1 register.
- Operation Execution: Execute the operation with x1 as the destination register.
- Immediate Loading: Utilize the addi rd, x0, imm instruction format for loading values into registers, where x0 is a constant zero register.

Mapping RISC-V Assembly to Machine Code

The table below illustrates the correspondence between RISC-V assembly instructions and the machine code outputs for the given examples:

RISC-V Assembly	Machine Code
addi x2, x0, 3	000000000011 00000 000 00010 0010011
addi x1, x0, 2	000000000010 00000 000 00001 0010011
add x1, x1, x2	0000000 00010 00001 000 00001 0110011
addi x2, x0, 5	000000000101 00000 000 00010 0010011
addi x1, x0, 4	000000000100 00000 000 00001 0010011
add x1, x1, x2	0000000 00010 00001 000 00001 0110011
addi x2, x0, 9	000000001001 00000 000 00010 0010011
addi x1, x0, 5	000000000101 00000 000 00001 0010011
mul x1, x1, x2	0000001 00010 00001 000 00001 0110011

Supported Operations

The program will support a variety of arithmetic and bitwise operations as detailed in the table below. Each operation corresponds to a specific symbol that should be recognized and processed by your program.

Operator	Meaning
+	addition
-	subtraction
*	multiplication
ˆ	bitwise xor
&	bitwise and
|	bitwise or

Instruction Formats for Supported Operations

Each operation symbol corresponds to specific RISC-V instruction codes. Below is a description of how each operation is implemented in RISC-V using the respective instruction formats:

RISC-V assembly code	RISC-V machine instructions

add rd, rs1, rs2	Function (funct7): 0000000 (Addition) rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 (Addition) rd (Destination Register): 5 bits Opcode: 0110011 (same for all R types)
sub rd, rs1, rs2	Function (funct7): 0100000 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
mul rd, rs1, rs2	Function (funct7): 0000001 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
xor rd, rs1, rs2	Function (funct7): 0000100 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
and rd, rs1, rs2	Function (funct7): 0000111 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
or rd, rs1, rs2	Function (funct7): 0000110 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
addi rd, rs1, 5	Immediate (12 bit binary number): 000000000101 rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0010011 (I format)

Assumptions

- The input tokens are separated by a blank character.
- An input will not be longer than 256 characters.
- The provided postfix expression is syntactically correct.
- All numerical values and results of operations will be 12 bit values at most.
- There will not be any trailing spaces at the end of the input.

Execution

The example below illustrates a sample execution of the program, including producing the executable, running it, taking input, and displaying the output.

```
$ make
$ ./postfix_translator
2 3 + 4 5 + *
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
```