Project Overview

In this project, you are required to implement a GNU assembly language program that interprets a single line of postfix expression involving decimal quantities and outputs the equivalent RISC-V 32-bit 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 ^ & 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
^	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.

Purpose

The main goal of this project is to develop a program that can change postfix expressions into RISC-V 32-bit machine code using GNU assembly language.

Design

The design of the postfix translator is centered around a stack-based approach for processing postfix expressions into RISC-V machine language instructions. It begins with input handling, where the program accepts a validated single line of input containing numbers and operators. A stack is utilized to manage operands; numbers are pushed onto the stack until an operator is encountered, which then triggers the popping of operands for operation execution. The program includes a printing module that prints the final RISC-V machine language instructions as output.

Implementation Details

In the code we have implemented, we have some main parts. These are process, operation_modules, and bit_printers. Basically, the first one takes only one character from the provided input, and checks whether we are considering a number, a whitespace, or an operator sign. if number, this is the tricky part where we use the stack, we pop the stack. If the popped information is a dot, we push a dot and the number we are considering. Else if the popped thing is another number we multiply it by ten and then add the current number from the input line. Other possibilities occur mostly when the current char is an operation sign. In this case we forward the program to the current operation modules. The last possibility is having a whitespace. If whitespace it means that we considered an operation sign beforehand or finished considering a previous number. So, we should separate the previous number from the next one with dot. We pop the stack; if dot we push it again and continue, if not we push a

dot. This method ensures that the stack includes numbers separated by dots. Hence when forwarded to opeartion_modules we can have a stack full of effective information. Here comes the second part where we are in operation_modules. The thing we do is that popping two things from stack for each operand (since there will be dots with them) and make the associated operation and push the result to the stack again with a dot. In the meantime (after popping each information from the stack) we convert its form from decimal to binary and print it. This printing modules will print the number in a 12-digit binary number. The simple logic is the following: take the number in a temporary register and have an additional register to manipulate the data. In each iteration we will decrease the counter which is initially 12 and shift left the manipulative register and then AND it with 2048 (2^12), shifting it right by 11 and printing the result will give us the binary representation at the end. After printing the binary representation, we print the registers (0 and 2; 0 and 1) and addi RISC-V machine codes. The thing that we do last for each operation is printing out the RISC-V instruction machine code of the current operation. The program's logic is it basically.

How to Run the Program

1-Write ;make to the command prompt.

2- ./postfix_translator to reach the executable

3- Then provide the postfix expression as input.

Difficulties Encountered

One of the biggest challenges we faced with this project was really getting the hang of GNU Assembly language. We had hard times learning it due to the lack of resources, except for 15-year-old YouTube lectures by Hindu lecturers. The process of printing some text, which is super easy in high-level languages where you just write one line of code, turned into a bit of an adventure here. Also, due to the strict and complex syntax of GNU Assembly language, we frequently made errors.

Example Inputs & Outputs