Project 2 - Postfix Translator

CmpE 230, Systems Programming, Spring 2024

Instructor: Can Özturan
TAs: Gökçe Uludoğan, Goshgar İsmayilov
SA: Deniz Bilge Akkoç

Due: May 5, 2024, 23:59 (Strict)

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.

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
```

Submission

Your project will be tested automatically on the provided docker image. Thus, it's important that you carefully follow the submission instructions. Instructions for setting up the development environment can be found at this page. The root folder for the project should be named according to your student id numbers. If you submit individually, name your root folder with your student id. If you submit as a group of two, name your root folder with both student ids separated by an underscore. You will compress this root folder and submit it with the `.zip` file format. Other archive formats will not be accepted. The final submission file should be in the form of either `2020400144.zip` or `2020400144_2020400099.zip`.

You must create a Makefile in your root folder which creates a `postfix_translator` executable in the root folder, do not include this executable in your submission, it will be generated with the `make` command using the Makefile. The `make` command should not run the executable, it should only compile your program and create the executable.

Additionally, submit your project to the Github Classroom assignment for this project to verify its structure and basic functionality. The link to join this assignment will be provided shortly.

Late Submission

If the project is submitted late, the following penalties will be applied:

Hours late	Penalty
0 < hours late <= 24	25%
24 < hours late <= 48	50%
hours late > 48	100%

Grading

Your project will be graded according to the following criteria:

- **Code Comments (8%):** Write code comments for discrete code behavior and method comments. This sub-grading evaluates the quality and quantity of comments included in the code. Comments are essential for understanding the code's purpose, structure, logic, and any complex algorithms or data structures used. The code should be easily readable and maintainable for future developers.
- **Documentation (12%):** A written document describing how you implemented your project. This sub-grading assesses the quality and completeness of the written documentation accompanying the project. Good documentation should describe the purpose, design, and implementation details, as well as any challenges encountered and how they were addressed. The documentation should also

include examples of input/output and how to use the program. Students should aim to write clear, concise, and well-organized documentation that effectively communicates the project's functionality and design decisions.

- **Implementation and Tests (80%):** The submitted project should be implemented following the guidelines in this description and should pass testing. This sub-grading assesses the quality and correctness of the implementation. Grading the test cases will be straightforward: your score from the test cases will be determined by the formula:

  Total Correct Answers to Lines / Total Lines Given

Warnings

- You can submit this project either individually or as a group of two.
- All source codes are checked automatically for similarity with other submissions and exercises from previous years. Make sure you write and submit your own code. Any sign of cheating will be penalized with an F grade.
- Do not use content from external AI tools directly in your code and documentation. Doing so will be viewed as plagiarism and thoroughly investigated.
- Project documentation should be structured in a proper manner. The documentation must be submitted as a .pdf file, as well as in raw text format.
- Make sure you document your code with necessary inline comments and use meaningful variable names. Do not over-comment, or make your variable names unnecessarily long. This is very important for partial grading.
- Do not start coding right away. Think about the structure of your program and the possible complications resulting from it before coding.
- Questions about the project should be sent through the discussion forum on Piazza.