CMPUT 229 - Computer Organization and Architecture I

Lab 2: RPN Calculator

Writing A Simple Program

In this lab you will write your first RISC-V assembly program. Your task is to write a Reverse-Polish-Notation calculator. Reverse polish notation - also commonly known as *postfix order* - simply refers to a particular ordering of a sequence of tokens delimited by spaces. The tokens either represent operands or operators. A notation that you are likely more familiar with is called *infix notation*. For example.\, in this infix ordered statement `"1 + 2"` , 1 and 2 are operands and + is the operator. In postfix notation the operator always follows its operands. For example the infix notation statement `"1 + 2"` can be written in postfix notation as `"1 2 +"` .

The input to this assignment is a sequence of tokens for the calculator, and some memory space to be used as a *stack*.

A *stack* is an abstract data type, in other words, it is a space where data can be stored. Your function will use the *stack* to store both the operands and the results of applying operators to operands. When data is added to the stack it is said to be *pushed* onto the stack, and when data is removed from the stack it is said to be *popped* from the stack. Because data in memory has no inherent meaning, *popping* a value from a stack is purely conceptual. *Popping* a value from the stack does not actually remove the value from memory, it simply removes the value from our current representation of the stack. The *top* of the stack refers to the latest element that was pushed onto the stack.

Stacks can be implemented to *grow* in one of two different ways. A stack can *grow* either by adding to or by subtracting from the address of the stack. In this lab we will be asking you to *grow* the stack by subtracting from the provided stack address, in other words, as you push items onto the stack you will have to decrement the value we provide to you in `a1` . For example, if the stack is to contain words, and the stack starts at address `0x10001024` , then the first item pushed onto to the stack would be stored at address `0x10001024` and the second element pushed onto the stack will be stored at address `0x10001020` .

Write a RISC-V assembly program that acts as a Reverse-Polish-Notation calculator. Your program will read tokens from a list of tokens provided. Each token occupies one word in memory. Therefore, you can read a token from the list using the `lw` instruction.

There are four types of tokens. The action that your program needs to take for each type token is:

- OPERAND: push into the stack
- PLUS: pop the two top values off the stack, add their values, and push the result into the stack
- MINUS: Let `A` be the value on the top of the stack. Let `B` be the value immediately below `A` in the stack. Pop `A` and `B` off the stack. Compute `C=B-A` . Push `C` into the stack
- TERMINATION: print the value on top of the stack using a `PrintInt` system call

Each token is represented by a 32-bit word. The values allowed for the four types of tokens are as follows:

Token	Value
OPERAND	non-negative integer
PLUS	-1
MINUS	-2
TERMINATION	-3

When your program starts executing, the list of input tokens is already stored in memory. The memory space that is needed to store the stack is also already allocated. Your program will receive these as *arguments* in registers `a0` and `a1` . These arguments are memory addresses that can be used to access memory. You will be implementing a subroutine called `calculator` . Thus, you need to use the label `calculator:` at the start of your code as the **very first** label under the `.text` directive. Here is the specification for `calculator` :

- `calculator:`
  - **Arguments:**
    - `a0` : address of the memory where the list of tokens is stored.
    - `a1` : address of the memory that is to be used as a stack.
  - **Effect:**
    - On reaching the termination token, print the top value on the stack using the `PrintInt` system call. **Do not** print a new line after printing the value on top of the stack so that your solution works well with the automated grading scripts.
  - **Note:**
    - In RISC-V assembly every subroutine must have a return statement, this return statement is the instruction: `jr ra` Thus, make sure to include this instruction at the end of your subroutine so that it can work well with the automated grading scripts.

When RARS starts executing, it looks for a `main:` label where it will find the user code. This common.s file, that is included in your solution template, contains a `main` function that sets up the list of tokens in memory, allocates memory for the stack, and then calls your `calculator` function. You are encouraged to read the code in the `common.s` file to understand how the whole program works.

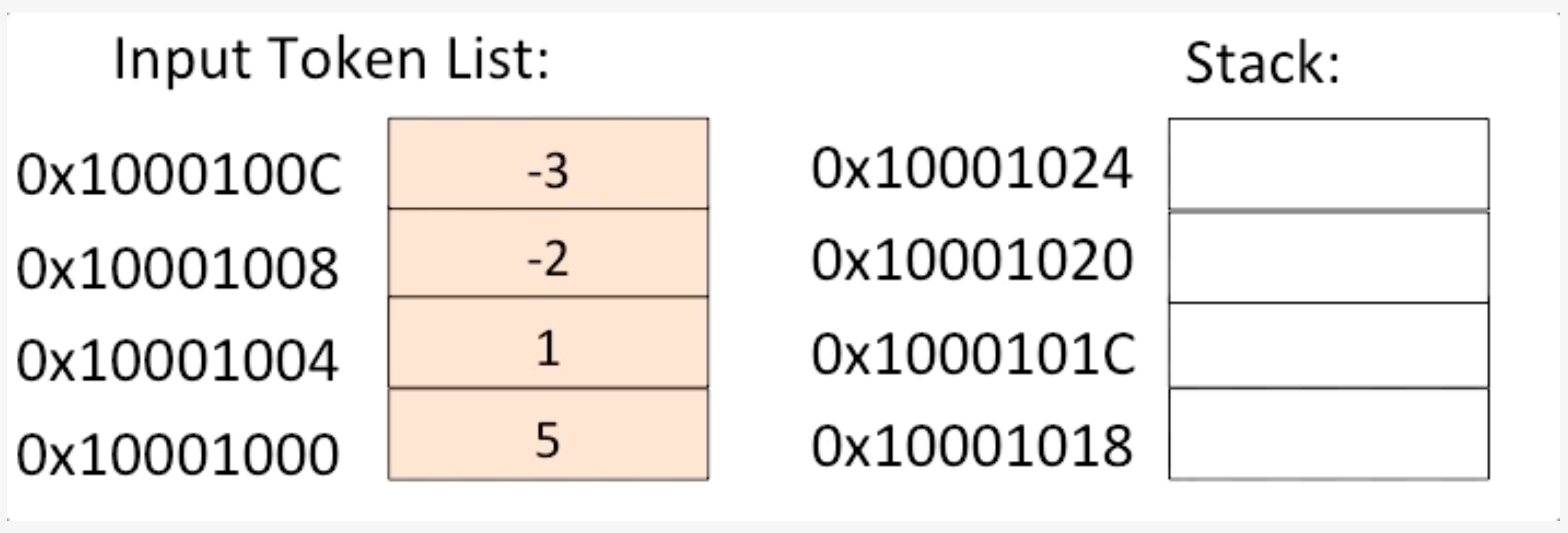
Here are some instructions to aid you in running and testing the program using the files that were provided in the original git repo.

To illustrate how the calculator operates, let's consider an execution with the following sequence of tokens:

OPERAND OPERAND MINUS TERMINATION

The animation below illustrates the reading of the tokens from the input list and the operations that occur in the stack. Each of the bold red rectangles on the input token list indicates that the program is reading the token for processing. The corresponding operation on the stack is showed simultaneously. As the animation shows the following sequence occurs:

- the first two OPERAND tokens are pushed into the stack;
- the MINUS token causes the two top values of the stack to be popped, the subtraction operation is performed and the result of the operation is pushed into the stack;
- the TERMINATION token should cause the value 4 to be printed.



For your program we suggest the following algorithm, though others are also correct:

- Load a word from the input list, `a0` .
- If the element is an operand, push it onto the stack.
- If the element is an operator, pop the two topmost values off of the stack and perform the operation specified on them. Then push the result on the new top of the stack.
- Increment `a0` by 4 in order to load the next value in the input list.
- Repeat the above process until you encounter the termination sentinel -3.
- Print out the result (the topmost value on the stack) using the `PrintInt` system call

Guarantees:

- All tokens are valid. Negative values that do not represent one of the tokens will not appear in the list.
- The input list will contain at least two operands before the first operator.
- There will be just enough operators and operands in a valid ordering so that your stack will only contain one value when a proper implementation reaches the end of the input list.

Check My Lab

Link to CheckMyLab

This lab is supported in CheckMyLab. To get started, navigate to the RPN Calculator lab in CheckMyLab found in the dashboard. From there, students can upload their test cases in the *My test cases* table (see below). Additionally, students can also upload their `calculator.s` file in the *My solutions* table, which will then be tested against all other valid test cases.

Resources

- An example demonstrating how to format your code is here: example.s This code contains a main label, but your `calculator.s` file should **not contain a main label**. This file is only provided to demonstrate clean and readable coding practices.
- Slides used for in-class introduction of the lab (.pptx) (.pdf)
- Slides used for in-lab introduction of the lab (.pdf)
- Instructions to Generate test cases and to run your code

Marking Guide

- 80% for correct program functionality
- 20% for program style
- Here is the mark sheet used for grading

Submission

There is 1 file to submit for this assignment:

- File `calculator.s` , should contain the assembly program you wrote in the last step.

For the file `calculator.s` , make sure to keep the line "`.include "common.s"`" and make sure that you do **not** add a `main` label.