Introduction to Lab #2

José Nelson Amaral

General Intro to 229 Labs

- In 229, a “lab” is a programming assignment:
  - A lab requires many more hours of work than the time allocated for lab sessions.
  - Lab sessions are “consulting hours” when TAs are available to answer questions and to help.
  - Reading/work prior to the lab date/time is essential.
  - The lab assignments will be progressively more difficult, and will require more time as the term advances.
- A CMPUT 229 lab is not a “lab” in the sense of a chemistry lab.

The `calculator` Program

Write a RISC-V assembly program that acts as a calculator for reverse Polish notation/postfix expressions.

Infix notation:

```
(1 + 2) * 3 = 9
```

Postfix notation:

```
1 2 + 3 *  = 9
```

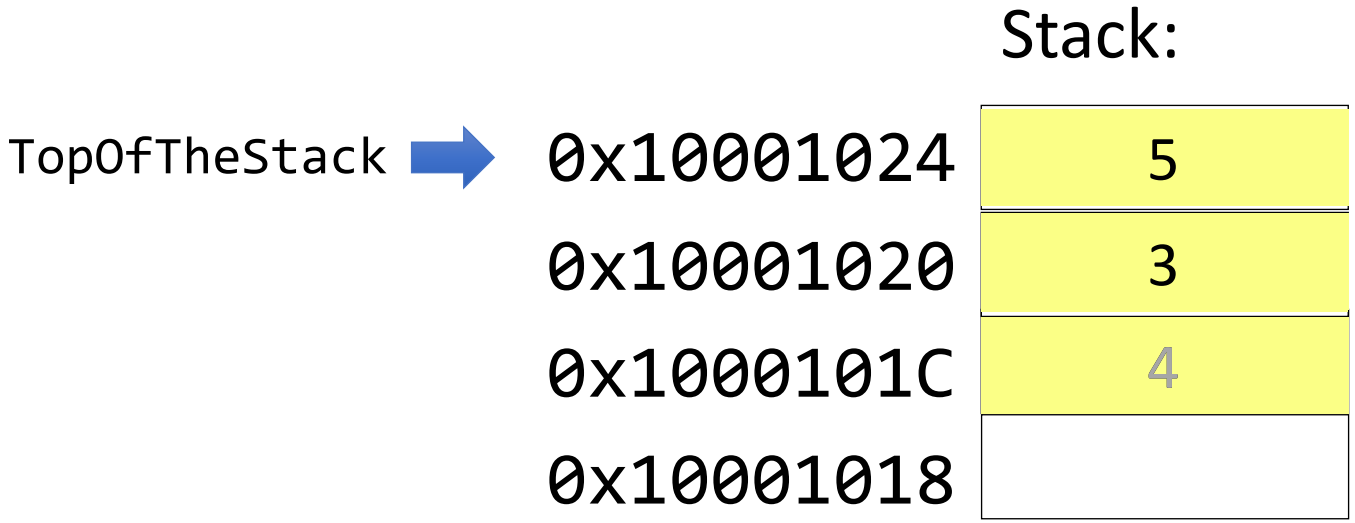
Types of input tokens for `calculator`

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

Operation of the `calculator`

- read a token from the input list
- if token == OPERAND (a non-negative value)
  - push value into the stack
- if token == PLUS or token == MINUS
  - pop two topmost values from the stack
  - perform operation
  - push result into the stack
- if token == TERMINATION
  - print out the value that is on top of the stack
  - terminate the program

How does the stack grow?



Initial State: Stack only contains value 5

Action: Push 1 on top of stack

Action: Push 4 on top of stack

Action: Pop 4 from top of stack

Action: Pop 1 from top of stack

Action: Push 3 on top of stack

Input Token List:

0x1000100C	-3
0x10001008	-2
0x10001004	1
0x10001000	5

Stack:

0x10001024	4
0x10001020	1
0x1000101C	
0x10001018	

Token Type:

Pop 1 and 5 from stack, execute the operation 5-1 and push result into the stack

Pop 4 from stack and write it to output

Formatting and Style

- Check the provided `example.s` file
- Check the lab grading marksheet

Assembler Syntax

comments begin with a sharp sign (#) and run to the end of the line.

identifiers are alphanumeric sequences, underbars (_), and dots (.) that do not begin with a number.

labels are identifiers placed at the beginning of a line, and followed by a colon.

```
        .data
item:
        .word 1

        .text
        .globl  main
main:
        lw      s3, item
Loop:
        add     t1, s3, s3      # t1 <- 2 * i
        add     t1, t1, t1      # t1 <- 4 * i
        add     t1, t1, s6      # t1 <- Addr(save[i])
        lw      t0, 0(t1)       # t0 <- MEM[save[i]]
        bne     t0, s5, Exit    # if save[I] != k goto Exit
        add     s3, s3, s4      # i <- i + j
        jal     zero, Loop      # goto Loop
Exit:
```

Assembler Directives

`.data` identifies the beginning of the data segment (in this example this segment contains a single word).

`.word 1` stores the decimal number 1 in 32-bits (4 bytes)

`.text` identifies the beginning of the text segment (where the instructions of the program are stored).

`.globl main` declares the label main global (so that it can be accessed from other files).

```
        .data
item:
        .word 1

        .text
        .globl  main
main:
        lw      s3, item
Loop:
        add     t1, s3, s3      # t1 <- 2 * i
        add     t1, t1, t1      # t1 <- 4 * i
        add     t1, t1, s6      # t1 <- Addr(save[i])
        lw      t0, 0(t1)       # t0 <- MEM[save[i]]
        bne     t0, s5, Exit    # if save[I] != k goto Exit
        add     s3, s3, s4      # i <- i + j
        jal     zero, Loop      # goto Loop
Exit:
```

Pseudo Instructions

pseudo instruction that loads the immediate value in the register

pseudo instruction that loads the address of specified label into register

```
# What's going on here ?
        .data
val:
        .word 12, 34, 56, 78, 90
outputMsg:
        .asciz "\n Result = "
newln:
        .asciz "\n\n"

        .text
main:
        li    a1, 5
        la    t0, val
        xor   t1, t1, t1
        xor   t2, t2, t2

loop:
        sub   t3, a1, t2
        blez  t3, exit
        lw    t4, 0(t0)
        add   t1, t1, t4
        add   t2, t2, 1
        addu  t0, t0, 4
        jal   zero, loop
```

```
exit:
        div   t5, $t1, $a1
        li    a7, 4
        la    a0, outputMsg
        ecall

        li    a7, 1
        add   a0, 0, t5
        ecall

        li    a7, 4
        la    a0, newln
        ecall

        li    a7, 10
        ecall
```

OS-style call to obtain services from RARS:

`a0-a2:` arguments

`a7:` system call code

`a7:` return value

Using GitHub

- While you can either type directly or copy and paste into an editor provided by github, this is not recommended.
- Learn to use basic command-line commands for git such as:
  - `clone`
  - `pull`
  - `commit`
  - `push`
- When you initially clone the repository provided, you will see a `Code` folder.
- In this folder there will be a `calculator.s` file.
  - Your solution goes at the bottom of this file.
  - Your code must start under the label ‘calculator’.

CMPUT 229 Student Submission License

- Carefully read the text of the CMPUT 229 Student Submission License to understand what you are allowed to do with your code before and after submission.
- After reading the license, complete the following information, in the `calculator.s` file, to acknowledge that you have read and understood the license:

```
#-------------------------------------------------------------
# CCID:
# Lecture Section:
# Instructor:           J. Nelson Amaral
# Lab Section:
# Teaching Assistant:
#-------------------------------------------------------------
```

common.s

- Every lab will have a common.s file that performs some actions and then calls the function written by the student.
- Read carefully and try to understand the common.s file as a way to learn.