Steven Reeves

CST 250

4/28/18

Lab 3 Report

**Introduction:**

The objective of this lab is to familiarize us memory allocation and reading/writing using labels. Indirect addressing and pointer arithmetic is also an objective of this lab. The pointer arithmetic is used to access values of an array and manipulate a stack. Conditional branching is to be practiced to perform iterations and nested loops. The use of a stack requires us to use macros to define push and pop. In short, this lab has us building the functionality for a very simple Reverse Polish Notation calculator.

**Part 1:**

The first part of the lab has us lay down the variables to be manipulated. A static RPN expression is stored in a word “array”, named EXPR.

For example, 3 4 \* 5 6 - + = is stored as the following:

// Expression for (3\*4) + (5-6) =

// RPN: 3 4 \* 5 6 - + =

EXPR: .word 3

.word 4

.word 0x80000000 + '\*'

.word 5

.word 6

.word 0x80000000 + '-'

.word 0x80000000 + '+'

.word 0x80000000 + '='

RESULT: .word 0 // should be result of entire expression

The RESULT variable is the result of the entire expression. It’s worth noting that for this lab, the expression given is assumed to be correct RPN notation and only contain the ‘+’, ‘-‘,’\*’, and ‘=’ operators.

The last declarations are global static values, used for the operators. They are as follows and will be used in the logic of .main:

.text

ADD\_OP: .word 0x80000000 + '+'

SUB\_OP: .word 0x80000000 + '-'

MULT\_OP: .word 0x80000000 + '\*'

EQU\_OP: .word 0x80000000 + '='

**Part 2:**

This part of the lab has us use loop and branching statements to evaluate all the items in the EXPR sequesnce. At first, this algorithm was used:

// Loop

// if digit, push

// Add 4 bytes to EXPR

// Jump to read EXPR

// if expression not equal

// Pop 2 values from stack and manipulate based on operator value

// Add 4 bytes to EXPR

// Jump to read EXPR

// if '=' pop last thing off and store value in RESULT

I started this process in a test driven development style with steps as follows:

* Load values of global variables into s registers and verify contents

la t0, ADD\_OP // Load value of ADD\_OP

lw s1, 0(t0)

la t1, SUB\_OP // Load value of SUB\_OP

lw s2, 0(t1)

la t2, MULT\_OP // Load value of MULT\_OP

lw s3, 0(t2)

la t3, EQU\_OP // Load value of EQU\_OP

lw s4, 0(t3)

* Define main loop and confirm functionality of pointer arithmetic

loop\_begin:

lw s7, 0(s0) // Load value of EXPR[i] into s7

addiu s0, s0, 4 // EXPR[i++]

j loop\_begin

nop

* Define Push and Pop Macros (in .text section), confirm stack is correct.

.macro push reg

addiu sp, sp, -4 // Move stack pointer "up" one.

sw \reg, 0(sp) // Store "reg" at new location

.endm

.macro pop reg

lw \reg, 0(sp) // Get value from sp

addiu sp, sp, 4 // Move sp

.endm

* Include check for ADD\_OP, SUB\_OP, MULT\_OP, and EQU\_OP in main loop. Jump to subroutines if found. Confirm subroutines are jumped to.

beq s7, s1, ADD\_THINGS // Go to ADD\_THINGS

nop

beq s7, s2, SUB\_THINGS // Go to SUB\_THINGS

nop

beq s7, s3, MULT\_THINGS // Go to MULT\_THINGS

nop

beq s7, s4, EQUALS // Go to EQUALS

nop

* Write and test subroutine for ADD\_THINGS

ADD\_THINGS:

pop t0

pop t1

addu t3, t1, t0

push t3

addiu s0, s0, 4 // EXPR[i++]

j loop\_begin

nop

* Write and test subroutine for SUB\_THINGS

SUB\_THINGS:

pop t0

pop t1

subu t3, t1, t0 // Note order of operands, due to stack functionality

push t3

addiu s0, s0, 4 // EXPR[i++]

j loop\_begin

nop

* Write and test subroutine for MULT\_THINGS (lab 2)

MULT\_THINGS:

pop t0

pop t1

//check to see if X or Y are zero, if so, we’re done.

beqz t0, inner\_loop\_end //if X is zero

nop

beqz t1, inner\_loop\_end //if Y is zero

nop

addu t9, zero, zero //clear the temp

inner\_loop:

addu t9, t9, t0 //temp = temp + X

addiu t1, t1, -1 //y--

//if y == 0, then the multiplication operation is complete

bnez t1, inner\_loop

nop

inner\_loop\_end:

push t9

addiu s0, s0, 4 // EXPR[i++]

j loop\_begin

nop

* Write and test subroutine for EQUALS

EQUALS:

pop t0

sw t0, 0(s5) // Put value at t0 into address at s5 (RESULT)

j endless // Expression is done

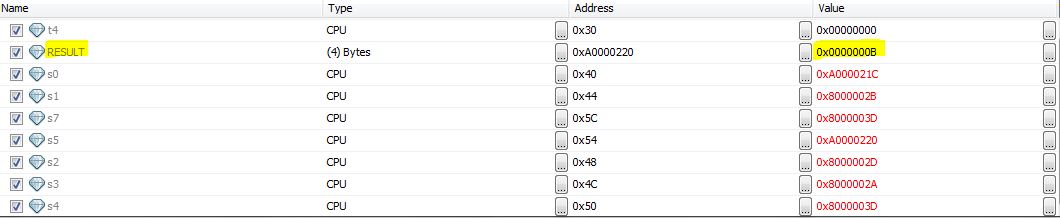
nop

**Part 3:**

The last part of this lab has us store the final item on the stack into the RESULT address and enter an endless loop. This way the RESULT can be reviewed at the end of the program.

I stored the RESULT and jumped to a simple endless loop in my EQUALS subroutine shown above.

After running my program with the above RPN expression I confirmed the contents of RESULT in the IDE.



(3\*4) + (5-6) = 11 = 0x0000000B

**Conclusion:**

I gained a lot of practice in using conditional branching and iterations in this lab. Jumping back and forth from the main loop and even looping in a subroutine was very helpful. Walking through those loops one at a time helped a lot too. Using the debugger in the IDE was something I gained much more experience with. Allocating variables and memory was also practiced in this lab. Indirect addressing and pointer arithmetic was used to traverse the stack and expression array. The definition and use of macros made this lab much more efficient and easy to write as well. I also gained some practice on stack use.

The biggest challenge for me was wrapping my mind around the stack. I’ve used them before, but using pointer arithmetic took a bit practice to fully understand. I feel like once I made and tested the macros, the need for this understanding was abstracted away. With the macros defined, it was just a basic use of a stack to do arithmetic. Traversing an array was much easier this time as well, practice from lab 2 helped a lot there.