CS2110 Spring 2014
Homework 7

Version: 1

The goal of this assignment is to get you familiar with the LC-3 calling convention using assembly recursion. This will involve using the stack to save the return address and the old frame pointer.

Overview

A Few Requirements

1. Your code must assemble with NO WARNINGS
2. Comment your code! This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad left notes to let you know sections of code or certain instructions are contributing to the code. Comment things like what registers are being used for and what not so intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semi-colon (;), and the rest of that line will be a comment. Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

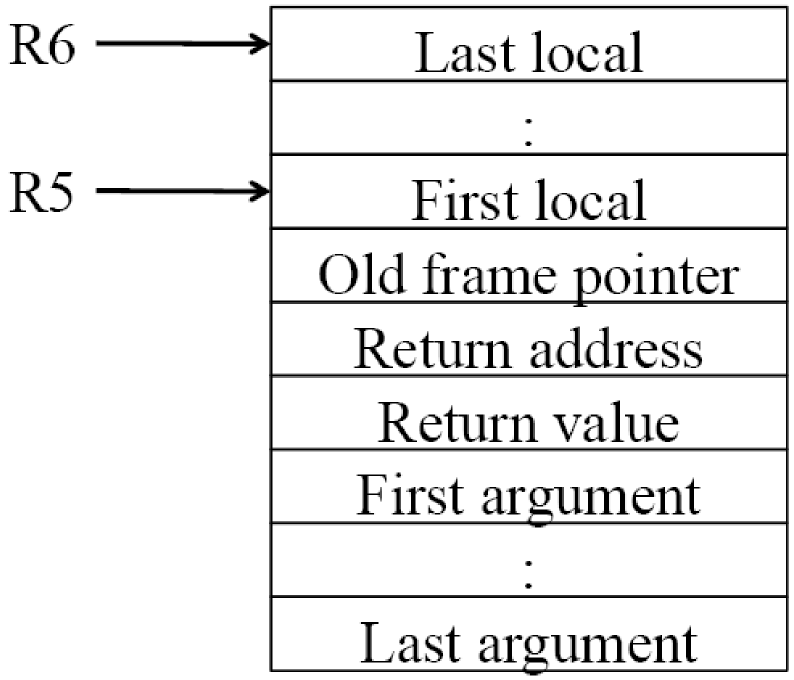
   Good Comment
   ```
   ADD R3, R3, -1 ;counter--
   BRp LOOP ;if counter == 0 don't loop again
   ```

   Bad Comment
   ```
   ADD R3, R3, -1 ;Decrement R3
   BRp LOOP ;Branch to LOOP if positive
   ```

3. DO NOT assume that ANYTHING in the LC-3 is already zero. Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.
4. Following from 3. You can randomize memory by using the menu option State → Randomize. And then load your program by saying File → Load Over
5. Use the LC-3 calling convention. This means that all local variables frame pointer, etc must be pushed onto the stack. Our autograder will be checking for this. (SEE NEXT PAGE)
6. Do this assignment using recursion, else you will receive no credit.
7. Start the stack at xF000
8. Do not do bounds checking when pushing and popping
9. Stack pointer points to the last used stack location (Which means allocate positions first, then store onto the stack pointer).
10. Test your assembly. Don't just assume it works and turn it in.
11. For all problems you will store your final answer at the label named ANSWER. You may not do this in your subroutines. Your subroutines are only responsible for following the calling convention and storing the answer in the return value slot in your stack frame.

LC-3 Calling Convention Overview

Since you will be using subroutines, you need a way of preserving the old return addresses and arguments before calling another subroutine which will trash these values. The stack is a convenient place to keep them. It would make sense to use the stack pointer for figuring out where to push the next value, but it is extremely inconvenient for loading a particular value into a register if you have pushed a whole bunch of things into the stack. You would have to keep track of how many values got pushed into the stack to find the exact address since you will not necessarily be using those values in the order they were pushed into the stack. This is where the frame pointer comes in handy. In LC-3 calling convention, the caller pushes all the arguments into the stack before calling the subroutine. Then the callee reserves space for the return value, pushes the return address (R7), and pushes the old frame pointer (R5) into the stack. The frame pointer should then be modified to point to the address right above where the old frame pointer was stored on the stack. You now know precisely where the old frame pointer, return address, and arguments are stored in relative to the frame pointer regardless where the stack pointer is pointing at. Using this will make debugging and cleanup much easier.



Part 1 nCr

You will start by implementing a subroutine that returns n choose r recursively. We are providing you with the ncr.asm file which already calls the subroutine to show you how to properly pass in arguments using the LC-3 calling convention.

Possible inputs for Combination: n is an integer between 0 and 16 inclusive, r is an integer between 0 and 10 inclusive.

Here is the C version of the subroutine:

```
int nCr(n, r) {
        if (k > n)
                return 0;
        if (r == n || r == 0)
                return 1
        return nCr(n-1, r-1) + nCr(n-1, r);
}
```

Part 2 GCD

Here is another simple subroutine that you will be converting to assembly. This subroutine calculates the GCD of the two given numbers, just like the last homework, but it does it a little bit differently. This time, you will be following the LC-3 calling convention to pass in the arguments yourself. When you finish executing the subroutine, make sure to pop the value off the stack like we did in ncr.asm and store it into the label named ANSWER. Remember, you can change the values in m and n to test your program.

Possible inputs for GCD: m and n are integers between 1 and 32767 inclusive.

Here is the C version of the subroutine:

```
int gcd(int m, int n) {
  if(m == n)
    return m;
  else if (m > n)
    return gcd(m-n, n);
  else
    return gcd(m, n-m);
}
```

Part 3 BST Height

For this part, you will calculate the height of a binary search tree.

Like the previous two problems, we will be providing you with a template file. You will have to call the subroutine with the proper inputs using the LC-3 Calling Convention.

Both inputs for this subroutine are supplied in labels ARRAY and INDEX, and the result should be stored in ANSWER.

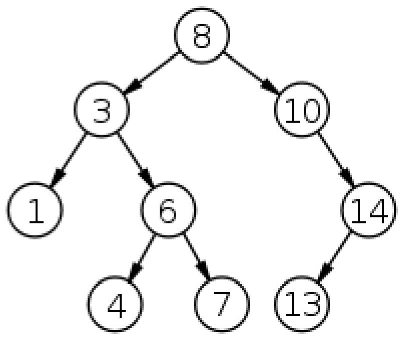
Here is the C code for binary search tree height:

(Notice left and right here are LOCAL VARIABLES and must be saved to the stack in its proper location failure to do this will result in a loss of points. We are testing if you can follow the calling convention and handling the stack here!)

```
/**
 * Calculates the height of a binary search tree
 * param  int* arr – location of the array, specified at label ARRAY.
 * param  int index – index of the node for which to calculate
                 height. The starting node is at index 1.
 */
int height(int* arr, int index){
        if ( *( arr + index ) == 0 )
                return 0;
        int left = height( arr, 2*index );
        int right = height( arr, 2*index + 1 );
        if ( left > right )
                return left+1;
        else
                return right+1;
}
```

How does this work?

Below is an example Binary Search Tree, with height 4. All you need to know about Binary Search Trees is the following:



- Binary search trees are made of nodes (a circle), which have data (a number) and two children nodes (left and right arrows).
- Binary search trees can be implemented in arrays very easily. If a node is in an array at index i, then the left child node is found at 2*i, and the right child node is found at 2*i+1.
- The first (head) node in a binary search tree implemented in an array is at index 1.
- The height of a node is equal to the maximum height of its children, plus one.

More about Binary Search Trees: http://en.wikipedia.org/wiki/Binary_search_tree

NOTE: Since you are to follow the C code on the previous page, you do not need to know exactly how binary search trees work!

Deliverables

Remember to put your name at the top of EACH file you submit.

ncr.asm

gcd.asm

height.asm