CS61C Summer 2015 Lab 4 - Function Calls and Pointers in MIPS

Goals

These exercises are intended to give you more practice with function calls and manipulating pointers in MIPS.

Setup

Copy the directory `~cs61c/labs/04` to an appropriate directory under your home directory.

Exercises

Exercise 1

This exercise uses the file `listmanips.s`.

We might have left Python behind with CS61A, but we definitely want to bring our friends `map` and `reduce` along with us! In this exercise, you will complete an implementation of `map` in MIPS. In general, `map` takes a function and a list as arguments and applies the function to each element in the list. Our function will be simplified to mutate the list in-place, rather than creating and returning a new list with the modified values.

Our `map` procedure will take two parameters; the first parameter will be the address of the head node of a singly-linked list whose values are 32-bit integers. So, in C, the structure would be defined as:

```
struct node {
  int value;
  struct node *next;
};
```

Our second parameter will be the address of a function that takes one int as an argument and returns an int. We'll use `jalr` (see below) to call this function on the list node values.

Our `map` function will recursively go down the list, applying the function to each value of the list nodes, storing the value returned in that node. In C, this would be something like this:

```
void map(struct node *head, int (*f)(int))
{
  if(!head) { return; }
```

```
  head->value = f(head->value);
  map(head->next,f);
}
```

If you haven't seen the `int (*f)(int)` kind of declaration before, don't worry too much about it. Basically it means that `f` is a pointer to a function, which in C is used exactly like any other function.

You'll need to use an instruction you might not have learned before to implement this: `jalr`. `jalr` is to `jr` as `jal` is to `j`. It jumps to the address in the given register and stores the address of the next instruction (i.e., `PC+4`) in `$ra`. So, if I didn't want to use `jal`, I could use `jalr` to call a function like this:

```
# I want to call the function garply, but not use jal.
la $t0 garply    # so I use la to load the address of garply into a register ($t0)
jalr $t0         # and then use jalr to jump and link to it.
```

There are 7 places (6 in `map` and 1 in `main`) in the provided code where it says "`#### YOUR CODE HERE ####`". Replace these with instructions that perform as indicated in the comments to finish the implementation of `map`, and to provide a sample call to `map` with `square` as the function argument. The sample list is already created for you in `create_default_list`. When you've filled in these instructions, running the code should provide you with the following output:

```
List Before: 9 8 7 6 5 4 3 2 1 0
List After: 81 64 49 36 25 16 9 4 1 0
```

Checkoff

- Show your TA your test run.

Exercise 2

Add the prologue and epilogue to the code in [nchoosek.s](#) so that it computes "n choose k", the number of combinations of `n` distinct elements taken `k` at a time. (This is also the `(n,k)` entry in Pascal's triangle.)

Checkoff

- Show your TA your code and its test run.

Exercise 3

Write two versions of a function named `first1pos` (starting from `first1pos.s`) that, given a value in `$a0`, returns in `$v0` the position of the leftmost bit that is set to 1 in the word in `$a0`. If `$a0` contains 0, store -1 in `$v0`. You are allowed to modify `$a0` in the process of finding this position. Positions range from 0 (the rightmost bit) to 31 (the sign bit).

The first version repeatedly shifts `$a0` to the left, checking the sign bit at each shift. The second version starts a mask at `0x80000000` and repeatedly shifts it right to check each bit in `$a0`.

Checkoff

- Show your TA both versions of the function and its test run.