Subroutines

A **subroutine** in Nios II assembly can be thought of as analogous to a function in C. They allow us to break software tasks into pieces so that they are do-able/modular.

Consider a "subroutine" in C called `my_sub`:

```
int main(void) {
        int x, z;
        x = my_sub(3);
        z = my_sub(4);
}

int my_sub(int p) {
        return (p+p);
}
```

When writing this in assembly, we can think about a couple things:

1. How does the `return` statement know exactly which line to return to back in the `main` function?
2. How do the function arguments get passed into `my_sub`?

To answer the first question, we can consider two things:

- the processor has another 32-bit register `pc` called the program counter that contains the address of the next instruction to be executed. This can be seen on CPUlator when debugging
- just like how `r0` is reserved to access `0`, `r31` is reserved for the "return address" (`ra`) and holds the address to go back to for subroutines

Here's some Nios II assembly code for (most of) the above C code:

```
main: movi r4, 3
        call my_sub
        movi r4, 4
        call my_sub

done: br done

my_sub: add r2, r4, r4
        ret
```

Let's examine the new line we see in the above code: the `call` instruction. The `call` instruction puts the value of `pc` (the memory address of the next instruction) into `ra`. It then sets `pc` to the memory address labeled by `my_sub`, goes there, and starts executing code.

Another new instruction: the `ret` instruction simply tells the computer to go back to the address store in `ra` (that is, it stores the value of `ra` back into `pc).

We can see that the `call` instruction is similar to a `br` instruction, but `br` only changes the value of `pc`, and not the value of `ra`.

Notice another thing: we have been typically told to only use registers `r8` to `r15` because the others are reserved for different purposes. Here we see some of them:

- we use `r2` if we have a single answer to come back from a subroutine
- we will use `r4` to `r7` to pass values into a sub-routines

The obvious question here is what if we need to send more than 4 values into a subroutine? Or what if we call another subroutine in a subroutine since there's only one `ra` value? The answer: we use memory. However we will need to be careful about how to use it. For this we will use a concept know from C programming: the **stack**.

If subroutine calls another subroutine, `ra` will be overwritten. So, we must save it and restore it when appropriate. For that we need a last-in first-out (LIFO) data structure: aka a **stack**.

Stacks

A stack has two operations:

- **push:** put a word on the top of the stack
- **pop:** grab and take off the word on top of the stack