Computer Architecture
Lab 4

In this lab, we'll work with calling a function and returning from a function. We'll also explore the stack data structure at the heart of the physical implementation of procedure calls.

You should turn in the source file for the program you're asked to write. It should be submitted via the tool available on Sakai for this lab. It is due by the start of the next class period.

Procedures

Among the most important high-level language concepts is that of the **procedure**. It's essential to **encapsulation,** and organizing code into functions is second-nature to experienced programmers. The architecture of a computer is designed to support programmers. Since functions are so core to programming, designing the hardware with that in mind can lead to efficiencies. This is an example of the software influencing the design of the hardware.

Read all about it in the text: sections 2.8 and A.6.

In this lab we'll use the MIPS `jal` instruction to call a function and the `jr` instruction to return from a function. The syntax is the following:

```
jal procedureAddress
jr $ra
```

For `procedureAddress`, we can use a label standing in for the name of the function, just as we've used labels as names of loop targets, arrays, etc. The `$ra` is a specific register in the MIPS architecture that the `jal` instruction automatically puts the return address into. So, while in principle `jr` can be used with any register operand, to use it as a return statement we typically need to write `jr $ra`.

Just like a branch, a function call changes the next instruction to be executed. Therefore, `jal` works fundamentally the same as `bne` and `beq`: it adds a number to the PC so that the next instruction to be executed is the first instruction of the function. What makes a function call different than a normal branch instruction is that the return address needs to be saved. The `jal` instruction is a hardware assist in doing this: it automatically stores the return address in register `$ra`. By convention, we typically use the registers `$a0`-`$a3` for function arguments and registers `$v0`-`$v3` for return values. (Note that this is how the `syscall` routines work, too.)

Let's write a MIPS program that uses functions. As a bonus, let's (start to) build the stack data structure that the hardware helps the compiler implement when we write procedures in our high –level code.

Lab Exercise

Write the following C++ program in MIPS:

```
//Our first stack

#include<iostream>
using namespace std;

const int STACK_SIZE = 5;

void push(int a[], int v, int& sp)
{
	a[++sp] = v;
}

int pop(int a[], int& sp)
{
		return a[sp--];
}

void add(int a[], int& sp)
{
	push(a, pop(a, sp) + pop(a, sp), sp);
}

void print(int a[], int sp)
{
	for (int i = 0; i < STACK_SIZE; i++)
		cout << a[i] << " ";
	cout << "sp = " << sp << endl;
}

int main()
{
	int stack[STACK_SIZE] = { 0, 0, 0, 0, 0 };
	int sp = -1;

	print(stack, sp);

	push(stack, 1, sp);
	push(stack, 5, sp);
	push(stack, 3, sp);
	push(stack, 4, sp);
	print(stack, sp);

	add(stack, sp);
	print(stack, sp);

	add(stack, sp);
	print(stack, sp);

	pop(stack, sp);
	print(stack, sp);
}
```