Computer Architecture

Exercise 8
(stack)

Peter Stallinga
Universidade do Algarve 2024-2025

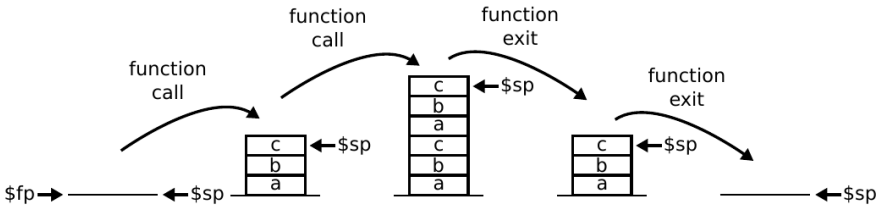
The problem with functions is that they must be independent of the main code. It means that the **function calls must not permanently destroy the information that is in the registers!** That is, there must be information that is guaranteed to be unaltered when coming back from the function call (the 'callee'), so that the calling program (the 'caller') can continue with processing the **intact** data.

We might simply save the registers on the heap, at a static place in memory, as we have done until now. This was the method in earlier days. The problem with this is that functions cannot be recursive, because then the registers saved in memory will be overwritten by the new function call, and the values lost forever.

To avoid this problem and to allow for recursivity, the **stack** was invented. Imagine in C we have the code

```
int sum(){
    int a;
    int b;
    int c;
     <...code...>
}
```

These values a, b, and c are not placed in the heap, but rather on the stack. Now, if the function calls itself, a new set of three variables is placed on the stack. That way, each level of depth of the function call has its own set of variables. The stack looks like this:



To implement this in Assembly we have to our disposition the stack pointer (**`$sp`**). If we want to temporarily to save a register we '**push**' it onto the stack. Then later, when we want to recover it, we '**pop**' it off the stack. Many other architectures have single instructions for this, but in MIPS they consist of two actions: 1) store or load the value and 2) adjust the stack pointer. Note that in MIPS the stack is growing downwards, to lower addresses.

To push the value of (target) register **`$rt`** on the stack

```
addi $sp, $sp, -4
sw $rt, ($sp)
```

To pop a value from the stack and assign it to (target) register **`$rt`**

```
lw $rt, ($sp)
addi $sp, $sp, 4
```

We now understand the difference between the temporary registers (**$t**) and saved registers (**$s**). The former are the responsibility of the caller, the latter are the responsibility of the callee. They must temporarily save the registers somewhere, before doing the processing in the function itself, and then restore these registers when going back to the caller.

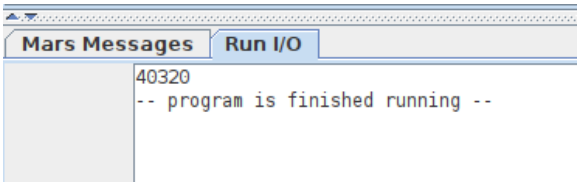
Thus, **$s** registers have to dealt with by the callee (function). It means that it must save all **$s** registers *that it is going to use* on the stack, and pop them off the stack just before leaving the function. It does not have to care about the **$t** registers and can do with them what it wants. The **$t** registers are the responsibility of the caller. It does not have to care at all about the **$s** registers, but if it is going to need the value of a **$t** register after the function call, it must save it on the stack before calling the function and pop it from the stack immediately after coming back from the function call.
Note that **the return address also has to be saved on the stack**, otherwise the program will never be able to find its way back to the original function call in 'main'!

Note: we can here now use macros for the popping and pushing code. This will create so-called pseudo-instructions (one which we have already seen in the beginning, namely **move** and **li**, move and load immediate, resp.). A macro is written in the following way

```
.macro macroname (%parameter1, %parameter2, ...)
   MIPS code here
.end_macro
```

Write macros for push e pop

Write a **recursive** MIPS function that calculates the factorial (*n*!) of the argument *n*. Example for 8!:



Take care of:
- Passing the argument in an **$a** register
- Return the value in **$v0**
- Save the relevant registers on the stack
- Save also the return address on the stack!
- Make sure you have matching pop and push instructions

The relevant (new) instructions for today are:

pop	Pop from stack
push	Push to stack