RP Stack Layout

When a program function executes, the RP Compiler maps the list of instructions to somewhere in memory. When a function inside the program is called, its contents such as function parameters, local variables, and return address are stored into a region of memory called a **stack.**

A stack behaves as follows:

- To store content onto the stack, we **PUSH** them to the top of the stack, like stacking pancakes
- To remove content from the stack, we **POP** them from the top of the stack, like eating the pancake on the top first (aka Last In First Out – LIFO)
- The memory address of the top of the stack at any time during execution is stored in the ESP register
- The stack grows towards lower memory address, by decrementing the stack pointer

When a function is called, it allocates a portion of the memory on the stack to use called a **frame.** Each frame consists is laid out as:

High Memory Address		**Description**
|	Function Parameters	
|	Return Address	Where to return
|	Previous Frame Pointer	
|	Local Variables	
V	…	
Low Memory Address		

Memory is pushed onto the stack via PUSH, and popped via POP

The steps are as follows:

1. Before we issue a CALL instruction we PUSH the parameters onto the stack, and we decrement the stack pointer
2. When we issue a CALL instruction, it will save the return address by pushing current value of EIP register onto the stack, and change the EIP register to the location of the new start of new function
3. We the push EBP on to the stack to save the frame pointer as well
4. Finally we push all the local variables
5. When the function returns,

A simple program that calls a function will have instructions like this:

```
Main:
PUSH 1
PUSH 2
CALL 5000 //function foo
…
```

```
Foo:
PUSH EBP
MOV ESP, EBP
ADD -8, ESP
ADD -16, ESP
…
MOV ESP, EBP
POP EBP
RET
```

All RP memory stack are 3-byte aligned, which means you see memory allocated in chunks of 3 bytes.