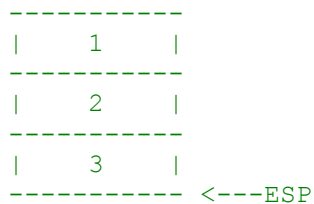


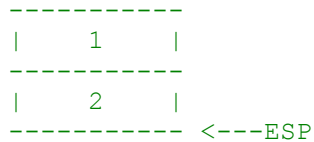
## The Stack in Detail

The stack is a Last In, First Out (LIFO) data structure. Imagine that you are stacking plates; the first plate you put on the stack will be on the bottom; the second plate will be on top of the first plate, and the third plate will be on top of the second. When you start taking plates off of the stack, the third plate will come off first, then the second, and finally, the first. The stack section in memory operates the same way: data can be placed on the stack, but if you place three pieces of data on the stack, you will first have to remove the last two in order to access the first piece of data.

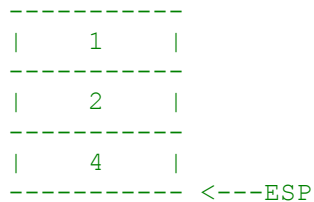
There are two types of stack operations: push and pop. When you want to place data onto the stack, you "push" it; when you want to remove data from the stack, you "pop" it. So, if you push the numbers 1, 2 and 3 in order onto the stack, when you pop the stack, you will get the number three; pop it again, and you will get the number two; pop it a third time and you will get the number one. To help visualize this, after pushing the numbers, the stack would look like:



If we then pop the stack, it will look like:



If we push the number 4 onto the top of the stack, it will look like:



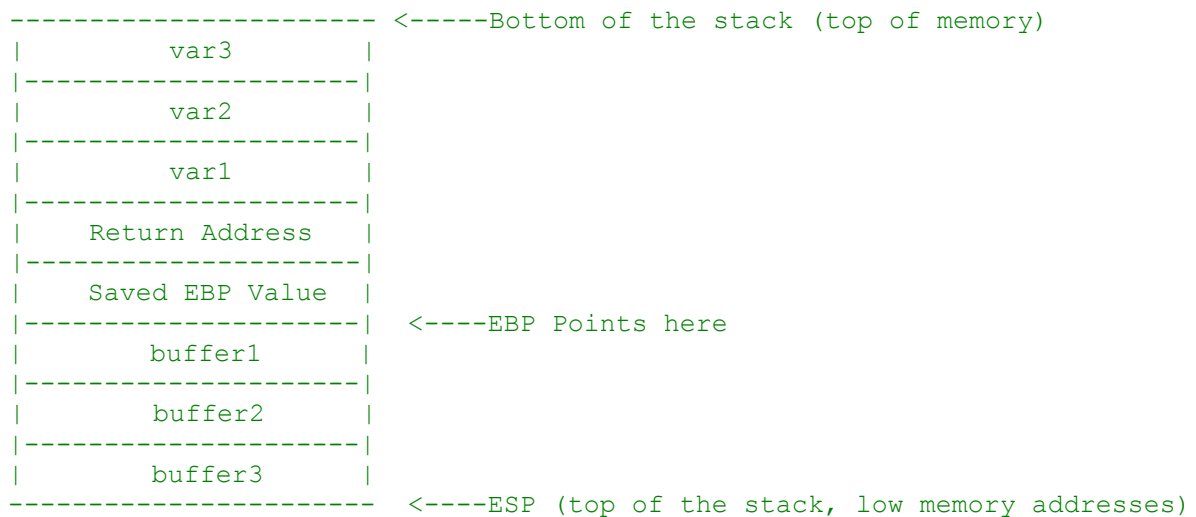
Don't be confused by the arrangement of the "top" and "bottom" of the stack; remember that the stack grows downwards, so data at the bottom of the stack (in this case, the number 1) is actually at the highest memory address, and the top of the stack (the number 4) is at a lower memory address. This is analogous to stacking plates on the ceiling.

Recall that the ESP register always points to the top of the stack. This means that whenever you push data onto the stack, the address stored in ESP is decremented by the number of bytes placed onto the stack; when you pop the stack, ESP is incremented by the number of bytes removed from the stack.

## Function Arguments and Local Variables

The stack is used to store a function's arguments and local variables; to understand how assembly instructions reference these variables, let's see how that data is arranged on the stack. Take a look at the following function and what the resulting stack layout would be:

```
int myFunction(int var1, int var2, int var3)
{
    char buffer1;
    char buffer2;
    char buffer3;
}
```



For the moment, we will ignore the return address and saved EBP value, and concentrate on how the arguments and variables get placed onto the stack. Before a function is called, all of its arguments must first be placed on the stack. These arguments are pushed onto the stack in reverse order; that is, in our example, `var3` would be pushed first, `var2` second, and finally `var1`:

```
push var3
push var2
push var1
call myFunction
```

The `call` instruction will automatically place the return value onto the stack, and the saved EBP value is pushed immediately afterwards by `myFunction` (again, we are ignoring these values for now - more on them later). Then, the local variables are pushed onto the stack in the order which they are declared; first `buffer1`, then `buffer2`, and lastly `buffer3`. When you look at the assembly code of a disassembled program however, you won't have nice names for variables like `var1` or `buffer1`; instead they will be indicated by memory addresses, or as offsets from EBP (recall that the purpose of EBP is to reference variables on the stack). Since the function arguments are located at higher memory addresses than the address pointed to by EBP, they will be referenced as positive offsets from EBP (example: `'ebp+8'`); local variables, being located at lower memory addresses, will be referenced as negative offsets from EBP (example: `'ebp-4'`). So, whenever you see something referenced as an offset from EBP, you know that you are dealing with a local variable.

## Return Addresses and the Prologue

Besides storing data and function arguments, the stack is also used for storing critical values when calling functions. Recall that the EIP always points to the next instruction to be executed; however, the EIP has no way of storing old instruction addresses, so when a function returns, the EIP needs a way to determine where to return to. Whenever a function is called, the memory address of the next instruction in the calling function is pushed onto the stack. When the called function finishes, this address is popped off the stack and placed into the EIP register so that the CPU can return to the next instruction in the calling function. Take the following pseudocode as an example:

```
functiona()  
    var x = 1  
    call functionb()  
    x=0  
return
```

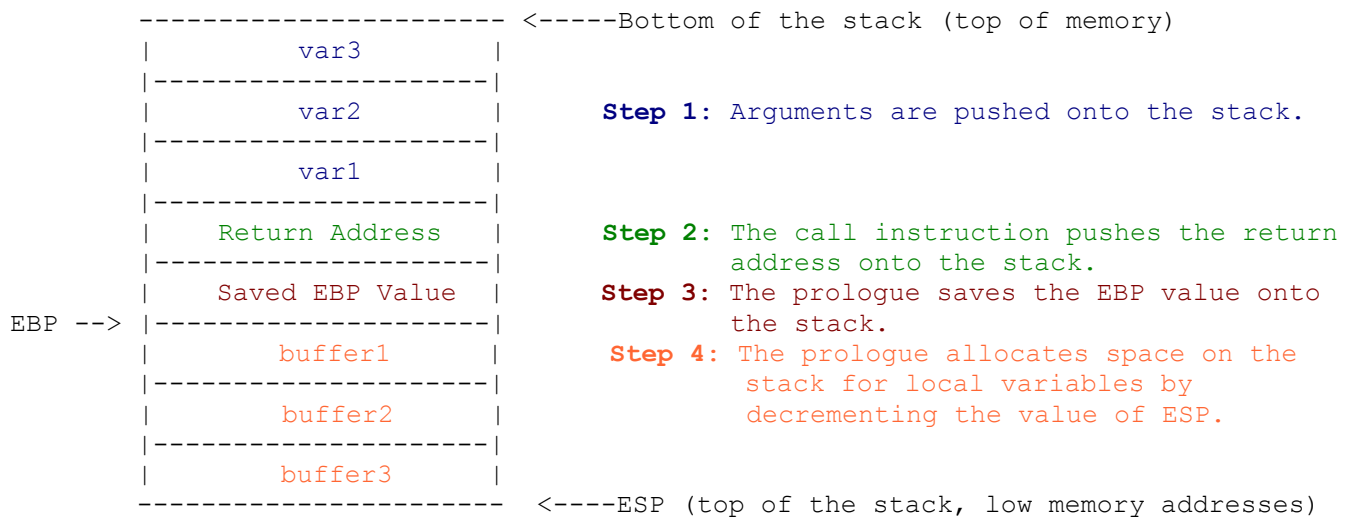
When functiona calls functionb, the memory address that contains the 'x=0' assignment is pushed onto the stack. When functionb finishes, that address is popped off the stack and placed into the EIP, so the processor then knows that the next instruction it has to perform is to set the variable x equal to zero.

In addition, the value of the EBP register needs to be saved and appropriately changed when a new function is called, such as in our above example. By now you may be wondering why the EBP is used at all; why not just reference variables from the stack pointer? The base pointer is used because as data is added to and removed from the stack, the position of the stack pointer (ESP) will be constantly changing, making it difficult to use it as a reference point for locating stack variables. However, it is impractical to use the same EBP value for every function, especially in more complex programs where you have functions inside of functions inside of functions, ad infinitum. But, just like the EIP, when a function finishes and returns control to its parent function, that parent function will need to have its original EBP value restored into the EBP register so that it can continue to reference its own variables and arguments. And, just like the EIP value, the calling function's EBP value is also placed on the stack.

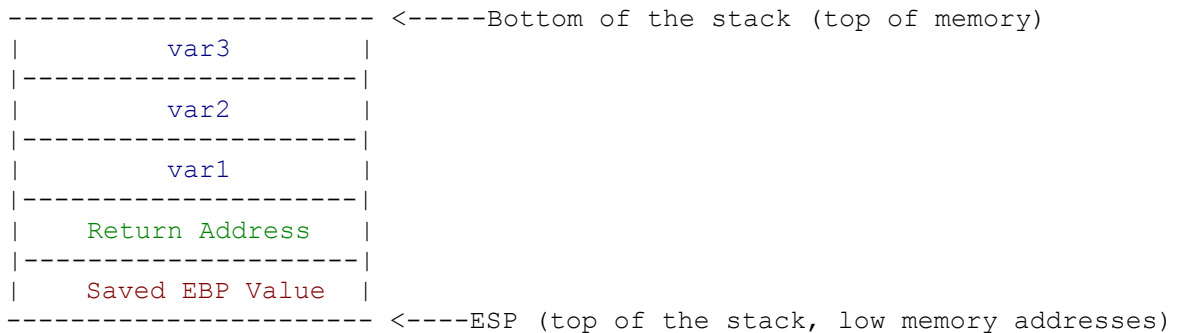
However, the EBP value is not pushed onto the stack automatically; this job is up to the child function, and the process of doing so is called the prologue. Basically what the prologue does is save the parent function's EBP value onto the stack, then gives the child function its own EBP value. Finally, the prologue allocates enough room on the stack to hold all of the local variables. The resulting assembly code looks like this:

```
push ebp  
mov ebp, esp  
sub esp, 0x24
```

Let's take this one line at a time, shall we. The first instruction is very simple; it pushes the value in EBP (i.e., the EBP value of the calling function) onto the stack. The second instruction copies the value in ESP into the EBP register (thus giving the child function its own EBP value). Finally, the third instruction decrements the stack pointer by 36 bytes (0x24 in hexadecimal); the actual value that is subtracted from ESP will of course depend on the size and number of local variables present in the function. But what is the second instruction really doing? Why copy the stack pointer value into EBP? To see why, look again at our sample stack layout; note the steps that have been added, and which parts of the stack are affected by them. Make particular note of where the EBP value is pointing to as well:



However, when the second instruction (`mov ebp, esp`) is executed, only steps one through three have been performed – no space has been allocated on the stack for local variables yet. So when the ESP value is copied into the EBP register, the stack actually looks like this:



Note that ESP is pointing exactly where the EBP needs to be. This makes setting the new EBP value simple; before allocating space for the local variables (step four), simply copy the value of ESP into EBP.

### Some Minor Details...

The above examples have been portrayed as layouts of the program's stack; in reality, they are really just sections of the stack known as stack frames. Since each function has its own arguments and variables, each function has its own frame. A function will clean up its frame before returning, but if you have functions called inside of other functions, you will have multiple frames on the stack. For instance, if `functiona()` calls `functionb()` which calls `functionc()`, an overall view of that program's stack would look like:

