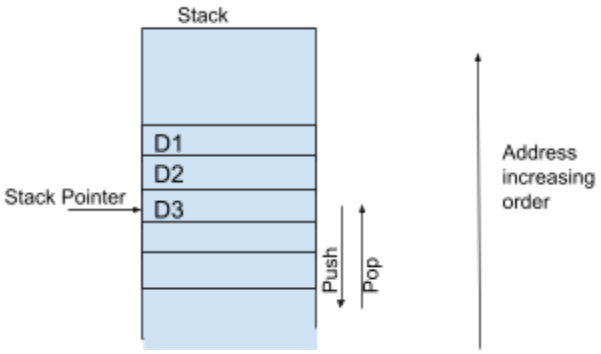
Caller-callee convention for gcc

We all might have used functions in C. In this part, we will learn some internal details of how a function call is actually handled by a compiler. Different compilers might handle it in different ways and we will focus on the gcc compiler, which is what we would also use in our assignments. Also, our focus would be on Intel's x86 architecture.

Let's revise some basic concepts first.
x86 registers - We would use Intel's x86 architecture to implement this lab. In x86, there are many general purpose registers as well as special registers. We consider four general purpose registers named as eax, ebx, ecx, and edx while special registers are eip and esp. eip is the instruction pointer or the program counter, while esp is the stack pointer. eip points to the next instruction to be executed by the processor while esp points to the top of the stack.

Stack - In processors, stack is implemented as a portion of memory and the top of the stack is pointed by a register called Stack Pointer (in x86, the stack pointer register is called esp). Adding a new data is called a "push" operation and it moves the stack pointer one location next. Removing any data is called a "pop" operation and it moves the stack pointer to one previous location. Typically, the data is accessed relative to the stack pointer, as shown in the following figure. Here, the stack contains 3 entries D1, D2, and D3. D1 was first added to stack and D3 was added last. The stack pointer in x86 always points to the last valid position.



In x86, the stack always grows downwards starting from a higher address. Therefore, the push operation decreases the stack pointer value while pop operation increases the same. So, if we want to push register eax on the stack, it would be equivalent to:

esp = esp-4 followed by *esp=eax

The same thing can be written in assembly language syntax as:

```
pushl %eax          subl $4, %esp
                    movl %eax, (%esp)
```

Similarly, a pop operation on eax could be equivalent to:

eax=*esp and then esp = esp+4

In assembly language syntax, a pop is equivalent to:

```
popl %eax          movl (%esp), %eax
                   addl $4, %esp
```

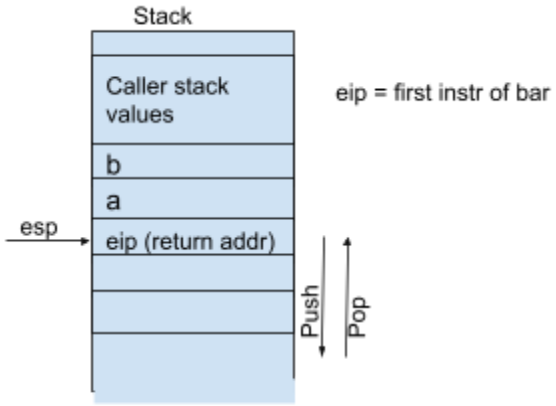
Function call - In C language, a function **foo** can call another function **bar**. We call function foo as the caller function and bar as the callee function. When we call a function, the execution moves from the caller to the callee function. Internally, the current value of the instruction pointer (eip) is saved on the stack and eip is updated with the address of the callee function. Once the execution is complete, the eip pushed on the stack is restored through a pop operation and the execution continues from the previous point. Conceptually, a call would do something like the following:

```
call 0x12345       pushl %eip (*)
                   movl $0x12345, %eip (*)
```

Now, let's consider a function foo which calls bar. The function bar takes two integer arguments and returns an integer, as shown below.

```
foo () {
        int a, b, x;
        x = bar (a, b);
        …
}

int bar (int a, int b) {
        int r;
        r = a + b;
        return r;
}
```

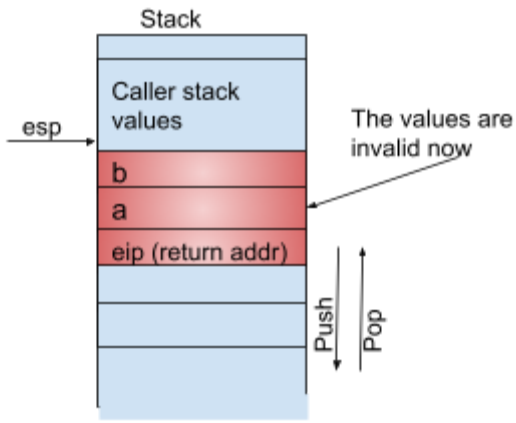


Stack status at the begin of bar (callee)

Internally, the function bar would be implemented as follows (as per the figure above):

Read a from stack into ebx register: **ebx = *(esp+4)**
Read b from stack into ecx register: **ecx = *(esp+8)**
Perform addition: **eax = ebx + ecx**
Return the result: The return value is always returned in the eax register. Since the eax register already contains the result, we need not do anything.

Now, we need to go back to the caller function, which is done through a ret instruction. Conceptually, the ret instruction restores the stored eip from the stack. (popl %eip)
The stack status after return is shown below.



Stack status after return to caller

Until now, we got an idea of how function call and return works in x86 and gcc. But, there are finer details. We used ebx and ecx in the callee function (bar). What if those registers were already having some useful values from the caller function (foo), they would get overwritten. What should be done to prevent it? Obviously, we would save them to stack. Now how many registers should the caller save? Ideally, all registers, because the callee could use any of them internally. But if the callee does not use these registers, then we are wasting CPU instructions in unnecessarily saving and restoring these registers. **Can we do better?**

gcc divides the registers into **caller-save** and **callee-save** registers. Caller-save registers are saved by the caller before jumping to the callee function. Callee-save registers are saved by callee, if it requires modification inside the callee function. In gcc, eax, ecx, edx are caller-save registers and ebp, ebx, esi, edi are callee-save registers. (We did not discuss ebp, esi, edi registers, but for now, just consider them as some registers).

If you are interested to know more, you can explore by writing a program (test.c) yourself like the below and execute the commands given.

```
#include <stdio.h>
int bar (int a, int b) {
        int r;
            r = a + b;
            return r;
}
void foo () {
            int a, b, x;
            x = bar (a, b);
        printf("%d", x);
}
int main(){
    foo();
}
```

Commands (we use -m32 with gcc to compile for 32-bit architecture):

1. gcc test.c -O0 -m32
2. objdump a.out -d > assembly.S

See the generated assembly for foo and bar. It is okay if you don't understand everything, but getting an idea would be useful. You might change the number of arguments and see how things change.

System calls

System calls are used whenever a user process requests for a service from the operating system. System calls could be like "open" to open a file, "fork" to spawn a child process, "write" to write to a file, "exec" to load a program and execute it, etc. The processor switches from the user mode to kernel mode during a system call, completes the request raised by the user process, and then goes back to the user mode. Similar to a function call that we studied earlier, a system call can also take arguments from user to kernel mode or return value from kernel to user mode. Each system call is identified by a unique number identifier and used by the kernel to accordingly process the request from the user.

xv6

xv6 is a Unix based academic OS from MIT designed for learning various implementation aspects. We would use xv6 for some simple exercises. The details about downloading, compiling and running it are given in the assignment statement. xv6 also implements system call like Unix and we would learn more about it.