Function call and calling convention

Let's not reinvent the wheel

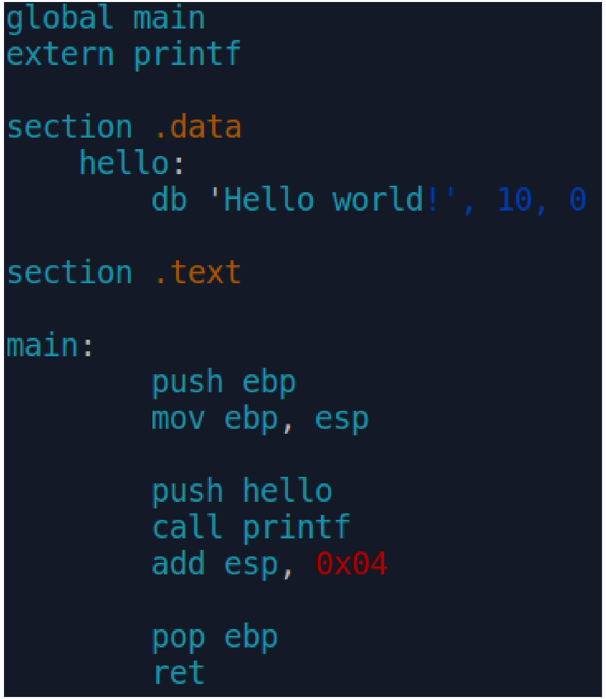
So, what exactly is a calling convention?

cdecl - Highlights

- **Parameters** are passed on the **stack** in **reverse order**
- **Caller cleans** the stack
  - This means, space made on stack for parameters is cleaned up by the caller, not the callee
- eax, ecx and edx
  - **Available** for use to **the callee**
  - This means if values in these registers are relevant before call, caller is responsible for saving these values
- All **other** registers need to be **saved by** the **callee** if used by the callee

- Follow the rules and x86 shall reward you!

Hello assembly!



```
global main
extern printf

section .data
    hello:
        db 'Hello world!', 10, 0

section .text

main:
        push ebp
        mov ebp, esp

        push hello
        call printf
        add esp, 0x04

        pop ebp
        ret
```

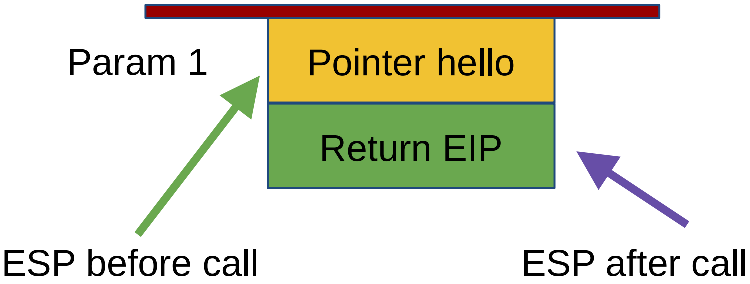
the keyword “extern” allows to declare a symbol as “external” to a given file. In other words, in this case, it allows us to use a C function from assembly.

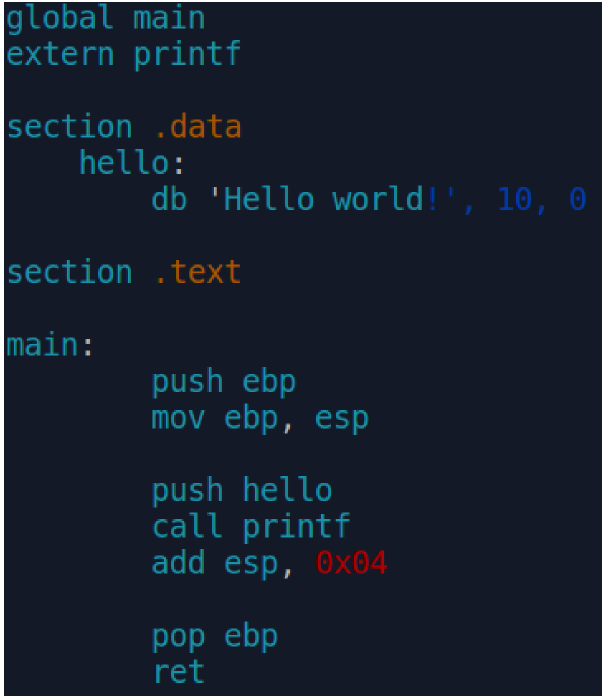
Let’s try this code and remove this line:
add esp, 0x04

What will happen?

what does the stack looks like at the moment of running the first instruction from the printf function?

Looking at the stack





```
global main
extern printf

section .data
    hello:
        db 'Hello world!', 10, 0

section .text

main:
        push ebp
        mov ebp, esp

        push hello
        call printf
        add esp, 0x04

        pop ebp
        ret
```

Pointer hello can be accessed as [ebp + 0x8]

Call cause the address of the instruction located after the call to be pushed on the stack.

The return EIP value is the value historically overwritten in classic stack buffer overflow situation.

When "ret" is executed, execution will resume at the address located in "return EIP"

Let’s write some code!