1 Basics of functions/procedures in x86

Assembly program:

```
section .data

section .text
        global main

        addTwo:
                ADD eax, ebx
                RET

        main:
                MOV eax, 4
                MOV ebx, 1

                CALL addTwo

                MOV ebx, eax

                MOV eax, 1
                INT 80h
```

addTwo is a procedure/function that basically **ADD**s two registers and **RET**urns the sum.

Debugging the program using GDB and observing the stack.
ESP → **E**xtended **S**tack **P**ointer. This register contains the address of the top of the stack.

The control flow is very simple in this program. We have two registers **ESP** and **EIP**(the instruction pointer). The **EIP** always stores the address of(or points to) the next instruction to be executed. It's like the program counter(**pc** register) in MIPS.

Now, when **addTwo** is **CALL**ed i.e when the instruction **CALL addTwo** is executed, then **EIP** was pointing to the next instruction that is **MOV ebx, eax**. Now, **ESP** was decremented(**stack grows downwards**) and the address that **EIP** was holding gets pushed onto the stack. So, the address that **EIP** was holding is the new top of the stack and **ESP** points to this new top. Now, since the address held by **EIP** gets stored in the stack, the address of the instruction **add eax, ebx** gets stored in **EIP**. Now, the body of **addTwo** starts getting executed. When **addTwo RET**urns then **ESP** is incremented, that is the the address of **MOV ebx, eax** is popped from the stack and restored into **EIP** and so the we see the lines after the **CALL** instruction execute normally.

Now, we just executed the first line inside our **main** procedure(rather label).

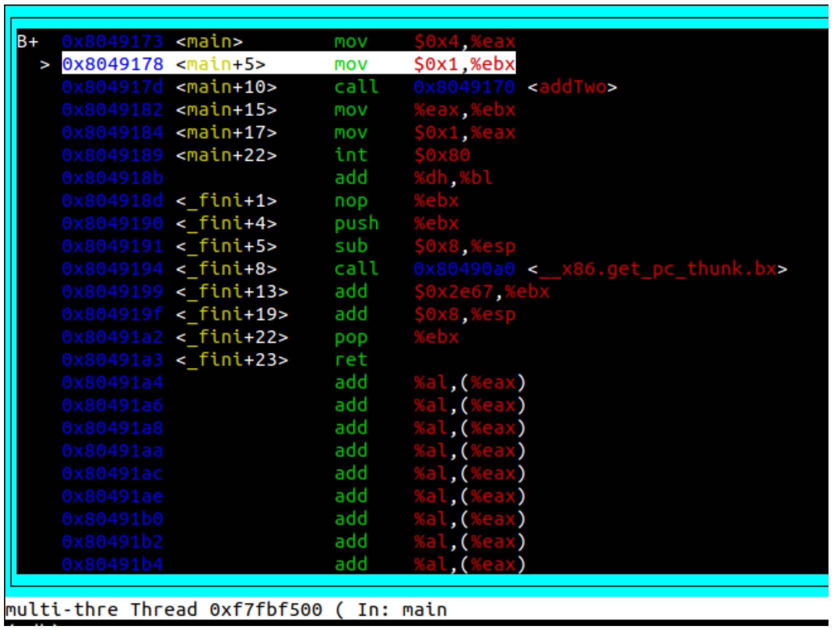


Figure 1: Executing the first line

If we inspect the **EIP** we will see that it stores the address of the next instruction to be executed(the highlighted instruction in the image above).

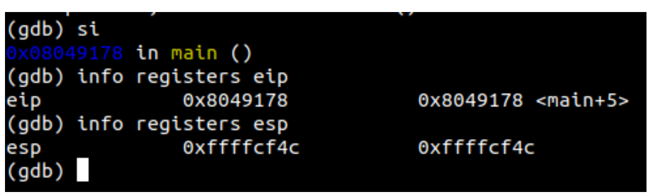
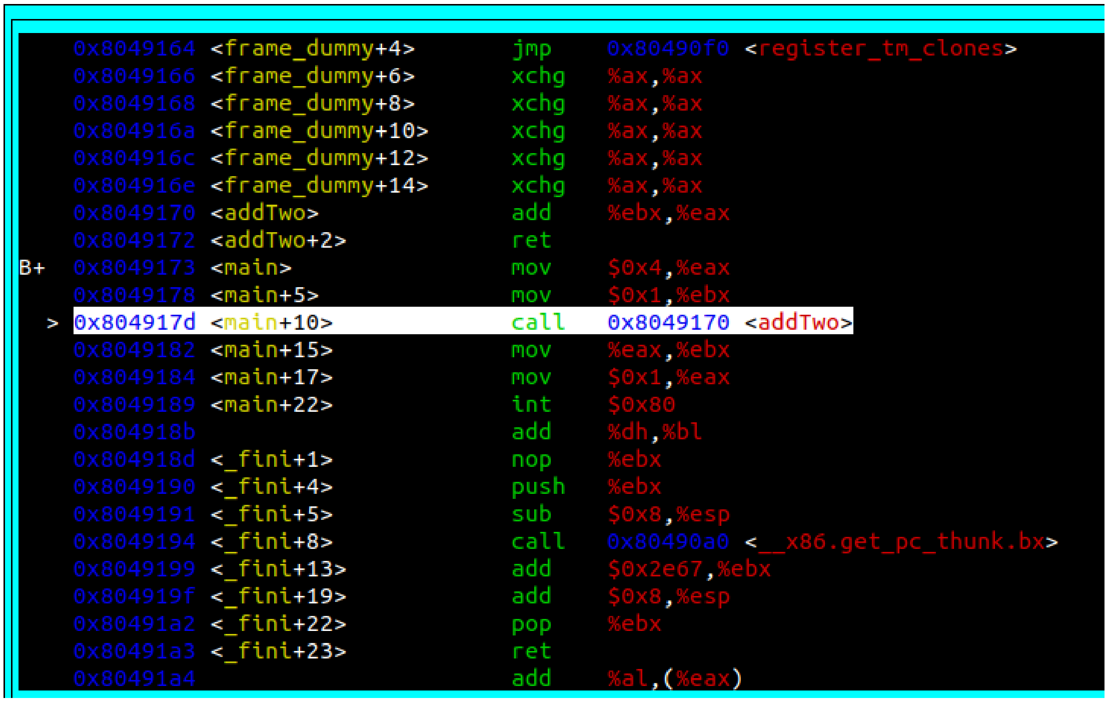


Figure 2: Address pointed to by **EIP**

Also, **ESP** is currently storing some address(which is not known to us), that is basically it is pointing to whatever is in the top of the stack at the moment.

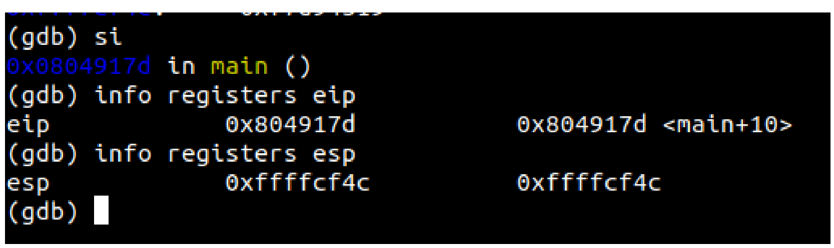
Now, we executed the next instruction:



```
   0x8049164 <frame_dummy+4>       jmp    0x80490f0 <register_tm_clones>
   0x8049166 <frame_dummy+6>       xchg   %ax,%ax
   0x8049168 <frame_dummy+8>       xchg   %ax,%ax
   0x804916a <frame_dummy+10>      xchg   %ax,%ax
   0x804916c <frame_dummy+12>      xchg   %ax,%ax
   0x804916e <frame_dummy+14>      xchg   %ax,%ax
   0x8049170 <addTwo>              add    %ebx,%eax
   0x8049172 <addTwo+2>            ret
B+ 0x8049173 <main>                mov    $0x4,%eax
   0x8049178 <main+5>              mov    $0x1,%ebx
 > 0x804917d <main+10>             call   0x8049170 <addTwo>
   0x8049182 <main+15>             mov    %eax,%ebx
   0x8049184 <main+17>             mov    $0x1,%eax
   0x8049189 <main+22>             int    $0x80
   0x804918b                       add    %dh,%bl
   0x804918d <_fini+1>             nop    %ebx
   0x8049190 <_fini+4>             push   %ebx
   0x8049191 <_fini+5>             sub    $0x8,%esp
   0x8049194 <_fini+8>             call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x8049199 <_fini+13>            add    $0x2e67,%ebx
   0x804919f <_fini+19>            add    $0x8,%esp
   0x80491a2 <_fini+22>            pop    %ebx
   0x80491a3 <_fini+23>            ret
   0x80491a4                       add    %al,(%eax)
```

Figure 3: Executing

We can see that the value of **EIP** is the address of the next instruction(the highlighted one) and **ESP** is still unchanged.



```
(gdb) si
0x0804917d in main ()
(gdb) info registers eip
eip            0x804917d           0x804917d <main+10>
(gdb) info registers esp
esp            0xffffcf4c          0xffffcf4c
(gdb)
```

Figure 4: **EIP** and **ESP**

Let's execute the next line:

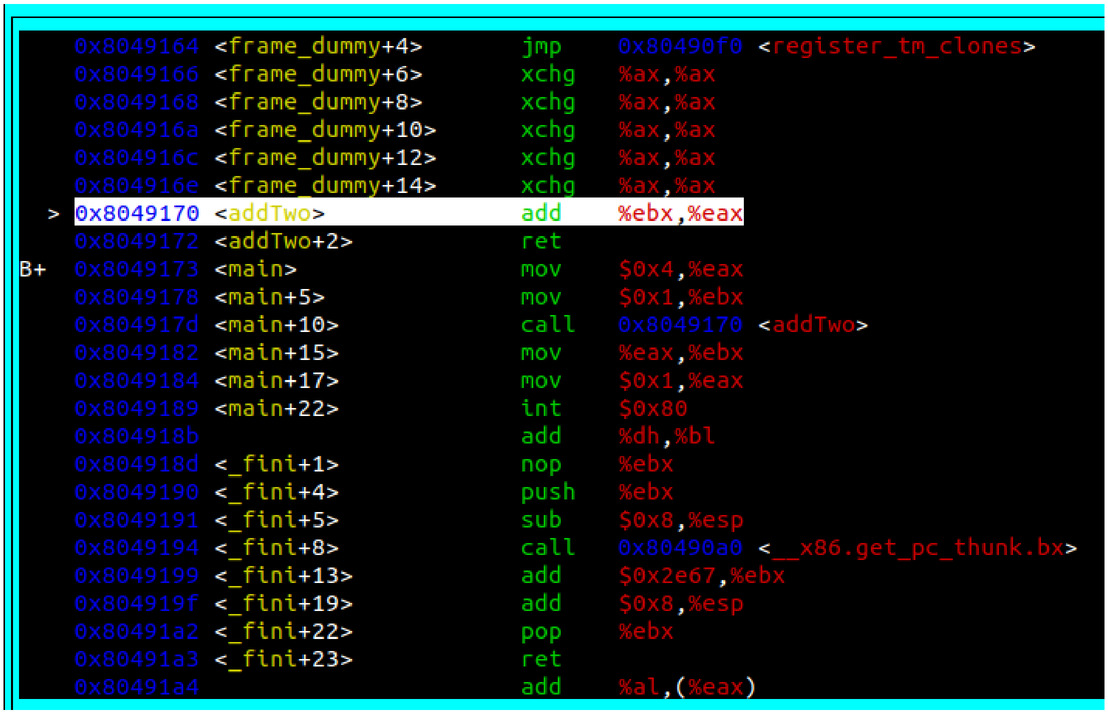


Figure 5: **addTwo** about to be executed

We can see that the first instruction of **addTwo** is about to get executed. Let's see what has changed in **EIP** and **ESP**:

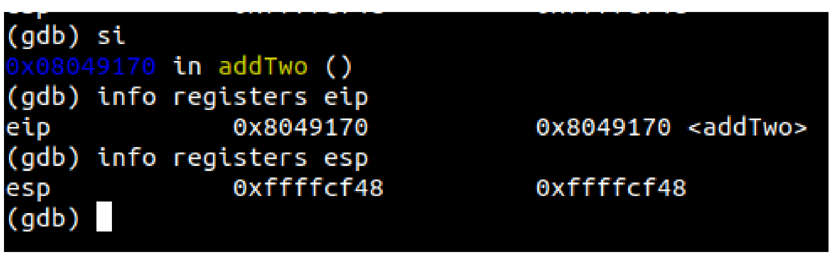
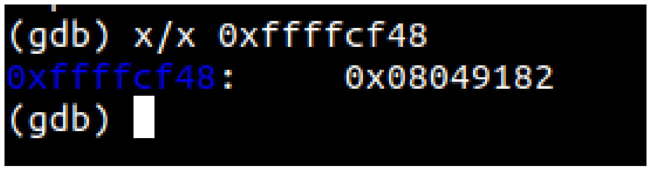


Figure 6: Changes in **EIP** and **ESP**

We can see that **EIP** stores the address of the first instruction of **addTwo** procedure and **ESP** stores **0xffffcf48**. So the value in **ESP** decreased by 4.

Now, if we try to view what is in this address pointed to by **ESP** then we will get:



We can see that the address that we get after using the **x/x** command is the address of the instruction **mov ebx, eax** which is the instruction right after **CALL addTwo** instruction as seen from Figure 5.

Now, we are in the **RET**urn instruction of **addTwo**:

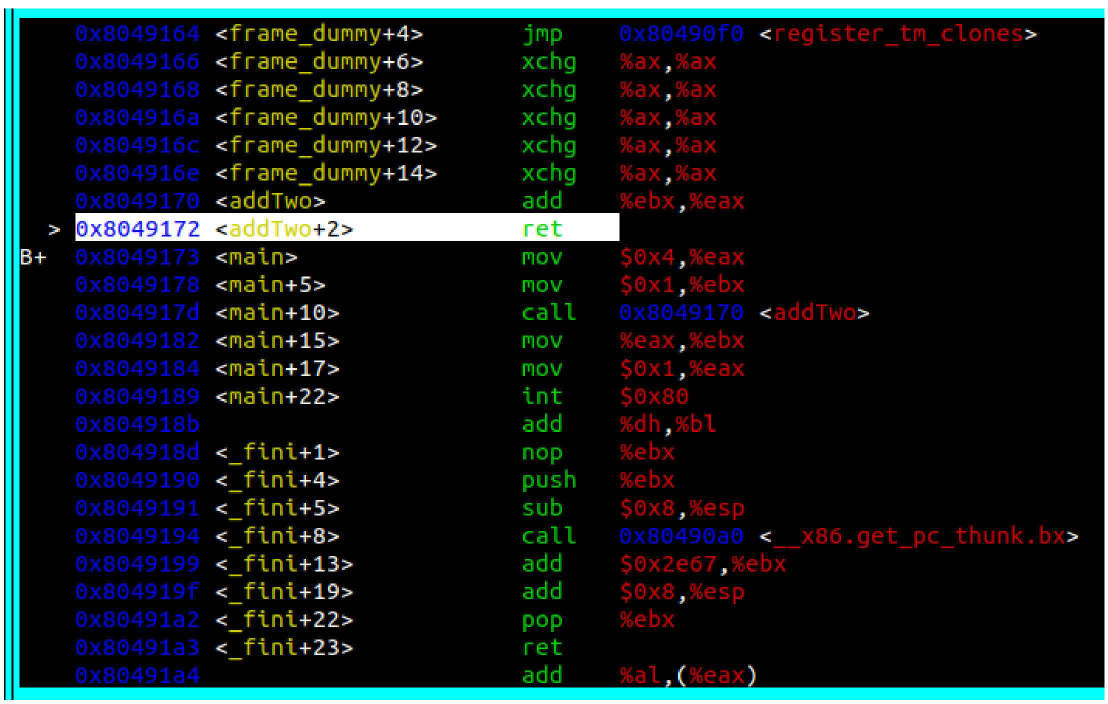


Figure 7: **RET**

We execute this line. After that, we can see that the instruction **mov ebx, eax** is the next instruction to be executed.

```
    0x8049164 <frame_dummy+4>      jmp    0x80490f0 <register_tm_clones>
    0x8049166 <frame_dummy+6>      xchg   %ax,%ax
    0x8049168 <frame_dummy+8>      xchg   %ax,%ax
    0x804916a <frame_dummy+10>     xchg   %ax,%ax
    0x804916c <frame_dummy+12>     xchg   %ax,%ax
    0x804916e <frame_dummy+14>     xchg   %ax,%ax
    0x8049170 <addTwo>             add    %ebx,%eax
    0x8049172 <addTwo+2>           ret
B+  0x8049173 <main>               mov    $0x4,%eax
    0x8049178 <main+5>             mov    $0x1,%ebx
    0x804917d <main+10>            call   0x8049170 <addTwo>
  > 0x8049182 <main+15>            mov    %eax,%ebx
    0x8049184 <main+17>            mov    $0x1,%eax
    0x8049189 <main+22>            int    $0x80
    0x804918b                      add    %dh,%bl
    0x804918d <_fini+1>            nop    %ebx
    0x8049190 <_fini+4>            push   %ebx
    0x8049191 <_fini+5>            sub    $0x8,%esp
    0x8049194 <_fini+8>            call   0x80490a0 <__x86.get_pc_thunk.bx>
    0x8049199 <_fini+13>           add    $0x2e67,%ebx
    0x804919f <_fini+19>           add    $0x8,%esp
    0x80491a2 <_fini+22>           pop    %ebx
    0x80491a3 <_fini+23>           ret
    0x80491a4                      add    %al,(%eax)
```

Figure 8: After **CALL**

We can also see that **ESP** is now restored and **EIP** stores the address of the instruction **mov ebx, eax**.

```
(gdb) info registers eip
eip            0x8049182           0x8049182 <main+15>
(gdb) info registers esp
esp            0xffffcf4c          0xffffcf4c
(gdb)
```

Figure 9: **ESP** and **EIP**

This video helped me a lot in understanding the stack.

2 Sources

- https://en.wikibooks.org/wiki/X86_Disassembly/The_Stack
- https://wiki.osdev.org/Stack
- https://www.youtube.com/watch?v=vcfQVwtoyHY
- https://youtu.be/F58WAnf2gr0?si=Bj4B0OpnJ7N0rG5v
- https://stackoverflow.com/questions/40324514/what-is-the-difference-between-esp-and-eip-registers
- https://stackoverflow.com/questions/3699283/what-is-stack-frame-in-assembly