# Function Calls & Stacks in Assembly x86

🤔 **What is a Function Call?**
If you're writing a function is_prime(x), when you call it, the computer needs to remember where to return after executing the function.

**Example:**

```
int main() {

int x = 3;

int y = 5;

int res = is_prime(x); // Function call

x = x + y;

res = is_prime(x);

return 0;

}
```

---

When is_prime(x) is called:
✅ The program must remember where to return in main() after execution.
✅ The values of registers at the current location must be preserved to prevent data loss when returning from the function.

---

🛠️ **Stack Registers: ESP & EBP**
💡 The stack is controlled using two special registers:

- **ESP (Stack Pointer)** – Points to the last element of the stack.
- **EBP (Base Pointer)** – Holds a fixed reference point for accessing function values.

🔍 **Stack memory grows downward:**

- **ESP moves down** when storing new values (`sub esp, 4`).

- **ESP moves up** when retrieving values (`add esp, 4`).

---

🛠️ **Function Call Backend (Assembly)**
When calling the `is_prime(x)` function in C++, the following assembly instructions execute in the background:

🔹 **Function Call (`call is_prime`)**

push eip ; Store return address

push eax ; Save register eax

push ebx ; Save register ebx

push ecx ; Save register ecx

push edx ; Save register edx

jmp is_prime ; Jump to function address

---

🔍 **What's happening?**

✅ `push eip` – Saves the return address to resume execution after the function call.

✅ `push eax, ebx, ecx, edx` – Saves register values to prevent data loss.

✅ `jmp is_prime` – Jumps to the function for execution.

---

🔹 **Function Return (`ret`)**

pop edx ; Restore registers

pop ecx

pop ebx

pop eax

pop eip ; Load return address -> Resume execution

---

🔍 **What's happening?**

✅ `pop edx, ecx, ebx, eax` – Restores saved register values.

✅ `pop eip` – Loads the return address, resuming execution in `main()`.

---

🎯 **Example:**

***C Code:***

```
#include <stdio.h>

void funcB() {

printf("Inside funcB\n");

}

void funcA() {

printf("Inside funcA\n");

funcB();

}

int main() {

printf("Inside main\n");

funcA();

return 0;

}
```

---

**Step-by-Step Breakdown (Function Calls & Stack Execution):**

Understand the Stack Execution:

1. main() start hota hai, stack pe return address push hota hai.
2. main() se funcA() call hota hai, toh funcA ka return address stack pe jata hai.
3. funcA se funcB() call hota hai, toh funcB ka return address bhi stack pe jata hai.
4. funcB complete hota hai, stack se pop hota hai aur funcA wapas chalta hai.
5. funcA complete hota hai, stack se pop hota hai aur main wapas chalta hai.

---

### 1️⃣ When `main()` calls `funcA()`:

- **Current Instruction Pointer (EIP)** (which stores the next instruction of `main()`) **is pushed onto the stack**
- Jumps to `funcA()`

```
push eip ; Push main's return address onto the stack

jmp funcA ; Jump to funcA
```

---

### 2️⃣ When `funcA()` calls `funcB()` inside it:

- **Return address of funcA is also pushed onto the stack**
- Jumps to funcB()

```
push eip ; Push funcA's return address onto the stack

jmp funcB ; Jump to funcB
```

---

### 3️⃣ When `funcB()` completes (`ret` executes):

- **Stack pops `funcA`'s return address back into `EIP`**
- Execution resumes in `funcA`

```
ret ; Pop EIP (return to funcA's return address)
```

---

### 4️⃣ When `funcA()` completes (`ret` executes):

- **Stack pops main's return address back into EIP**
- Execution resumes in main

```
ret ; Pop EIP (return to main's return address)
```

---

## 🔥 Final Stack Execution:

| Stack (Top → Bottom) | Action |
|---|---|
| funcA return addr | Pushed before funcA() was called |
| funcB return addr | Pushed before funcB() was called |
| (Executing funcB) | (funcB is running) |
| (Executing funcA) | (funcA continues after funcB returns) |
| (Executing main) | (main continues after funcA returns) |

---