**What is assembly language?**

* Low level programming language, **specific to a particular computer architecture** or CPU
* Second generation programming language, first being the machine code which is basically a binary code (representation of any data using base-2 numeric, example data values, machine instructions, etc) instructions that CPU understands and executes directly.
* Using assembly language we write **human readable machine instructions** for a specific CPU
* It follows **Mnemonic representation** of code (human-readable symbolic representation of machine code instructions, For example, "MOV" might represent an instruction that moves data from one memory location to another.)
* Programs written in assembly language are **highly hardware dependent**
* assembly language can be **highly efficient** because they allow programmers to optimize code for specific hardware.
* **Lacks abstraction**, Programmers need to manage memory, registers, and control flow explicitly
* **typically used** in situations where **precise control over hardware is required**, such as in operating system development, embedded systems, real-time systems, and some high-performance computing applications.
* To convert assembly language code into machine code, **you need an assembler**. An assembler is a software tool that translates human-readable assembly code into binary machine code that can be executed by the CPU.
* **Examples:** Common assembly languages include x86 assembly (used in many Intel and compatible CPUs), ARM assembly (common in mobile devices and embedded systems), and MIPS assembly (used in some embedded systems and educational contexts).

**Some computer architecture concepts:**

**Registers**

* **Fastest storage elements** in computer
* Used for temporarily holding the data that CPU needs to process
* **Limited storage capacity**! In terms of both, count of registers and numbers of bit each register can hold. Common register sizes are 8bit, 16bit, 32bit, 64bit.
* **Various types of registers**- general purpose, special purpose (example program counter(PC), stack pointer(SP), instruction register(IR) ) , data registers, address registers, control registers.
* **Context Switching**: Registers are crucial for context switching, which is the process of saving and restoring the CPU's state when switching between different tasks or processes. Saving the contents of registers allows a CPU to resume execution of a process from where it left off.

**Stacks**

* linear data structure that follows the Last-In-First-Out (LIFO) principle.
* Basic operations supported are **push** and **pop.**
* **Function Calls:** Stacks are often used in CPUs to **manage function calls**. When a function is called, its return address and local variables are pushed onto the stack. When the function returns, the values are popped from the stack.
* **Memory Allocation:** Stacks can be used for managing memory allocation. In some CPU architectures, a stack pointer keeps track of the top of the stack, and memory is allocated and deallocated by adjusting this pointer.
* It can be used for evaluating postfix expressions.
* Stack overflow ( when element is pushed on full stack)
* Stack underflow (popping element from empty stack)

We will focus on x86 assembly language!

**Example 1**

**Terminal commands to execute assembly program ‘example1.asm’ using netwide assembler :**

*To install netwide assembler*

$ sudo apt install nasm

$ nasm -f elf32 example1.asm -o example1.o

$ ld -m elf\_i386 example1.o -o example1

The two commands above are used to assemble and link an x86 assembly code file (`example1.asm`) into an executable binary on a Linux system.

**1. `nasm -f elf32 example1.asm -o example1.o`**

- `nasm` is the Netwide Assembler, which is used to assemble assembly language source code into object files.

- `-f elf32`: This option specifies the output format as 32-bit ELF (Executable and Linkable Format). ELF is a common format for executable files on Unix-like systems.

- `example1.asm`: This is the source code file you want to assemble.

- `-o example1.o`: This option specifies the output file name as `example1.o`, which will be the resulting object file. Object files contain the compiled code in a format that can be linked to create an executable.

So, this command assembles `example1.asm` into a 32-bit ELF object file named `example1.o`.

**2. `ld -m elf\_i386 example1.o -o example1`**

- `ld` is the GNU linker, which is used to link object files into executable programs.

- `-m elf\_i386`: This option specifies the target architecture as 32-bit x86 (i386) ELF.

- `example1.o`: This is the input object file produced by the previous `nasm` command.

- `-o example1`: This option specifies the output file name as `example1`, which will be the resulting executable.

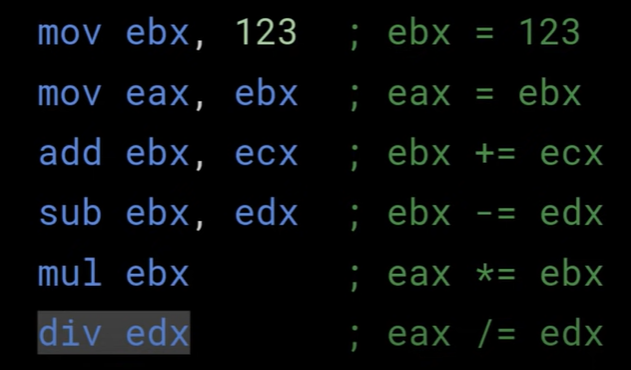
This command takes the 32-bit ELF object file `example1.o` and links it into a 32-bit x86 ELF executable named `example1`.

After running these two commands, you should have an executable file named `example1` that you can run in your terminal to execute the assembly code in `example1.asm`.

*$ ./example1*

*$ echo $? //outputs 42*

**Some common operation & operands**



Note: multiplication and division in x86 is done on particular register ebx and edx respectively. Actually, MUL and DIV also affect EDX. The MUL instruction stores the higher half of the result in EDX, while DIV stores the remainder from division there. If one dosn't know about that, one can be very surprised that suddenly their EDX is getting clobbered with "random" numbers after division/multiplication.

How is storing higher half of the result useful?

When you multiply a 32-bit number by a 32-bit number, the result might need as much as 64 bits to be stored. If you stored just the 32-bit result, you wouldn't get the actual result, just its remainder modulo 2³², which most likely is not what you want. So the CPU must store these higher bits somewhere, and its designers chose them to be stored in EDX. So the full result of multiplication is the 64-bit pair of registers, EDX:EAX.

**Instruction pointer**

The Instruction Pointer (IP) is a special-purpose register that holds the memory address of the next instruction to be executed by the CPU. It is also sometimes referred to as the Program Counter (PC) in other architectures.

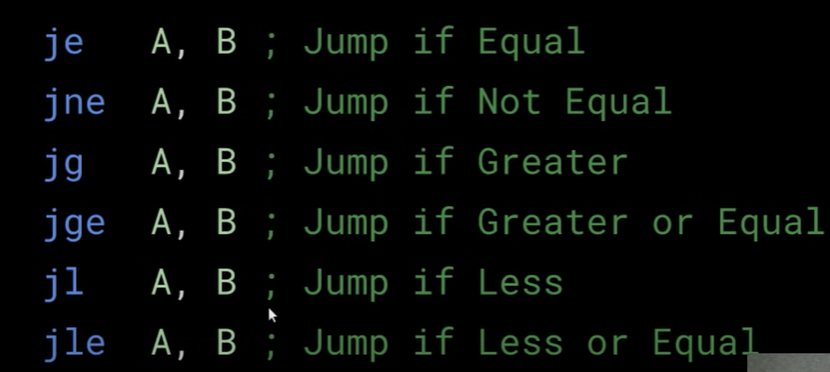
**How instruction pointer works?**

1. Fetching: The **CPU follows a fetch-decode-execute cycle**. During the fetch stage, the CPU reads the instruction at the memory address pointed to by the Instruction Pointer (IP).
2. Incrementing: After fetching the instruction, the CPU increments the Instruction Pointer to point to the next memory address where the next instruction is located. This ensures that the CPU proceeds to the next instruction in memory during the next cycle.
3. Decoding and Execution: Once the instruction is fetched and the IP is updated, the CPU decodes the instruction and executes it. The specific operation performed by the instruction depends on its opcode and operands.
4. Repeat: The CPU repeats this fetch-decode-execute cycle, continuously fetching and executing instructions one after the other, until it encounters a branching or jump instruction that modifies the IP to jump to a different memory address.

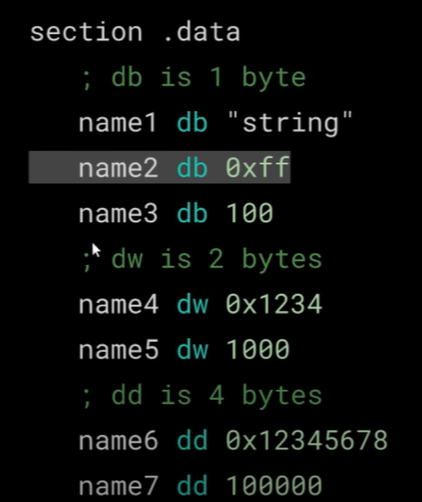
In x86 assembly language, the IP register is combined with the Code Segment (CS) register to form the Effective Address of an instruction. Together, they specify the exact memory location of the instruction to be executed.

Programs and operating systems use the IP register to manage control flow, loops, conditional statements, and function calls, making it a fundamental component of computer architecture and low-level programming.

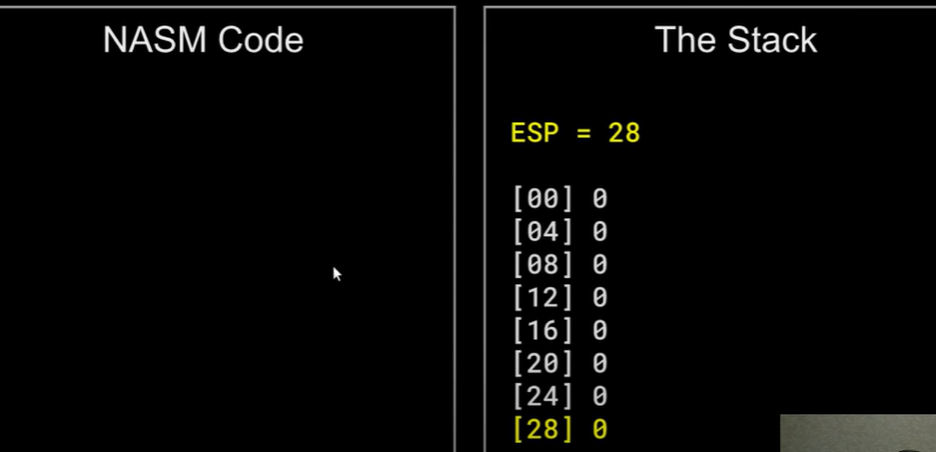
Different types of jump Mnemonic representation in x86 are:



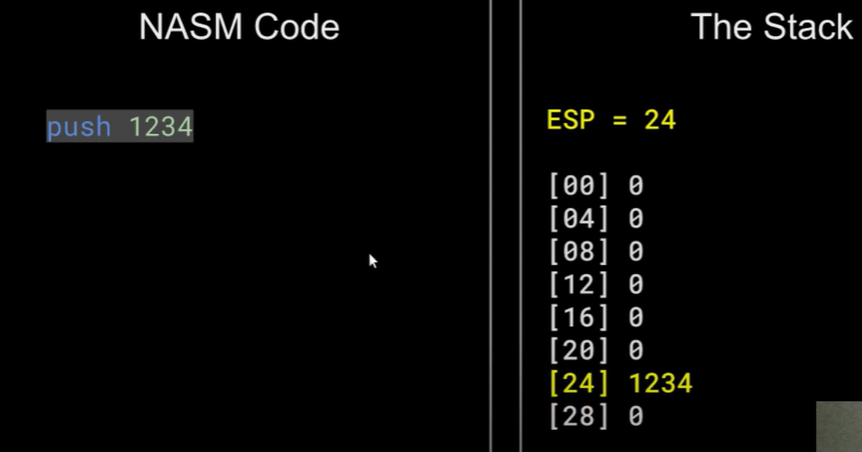
Some data types with the storage capacity

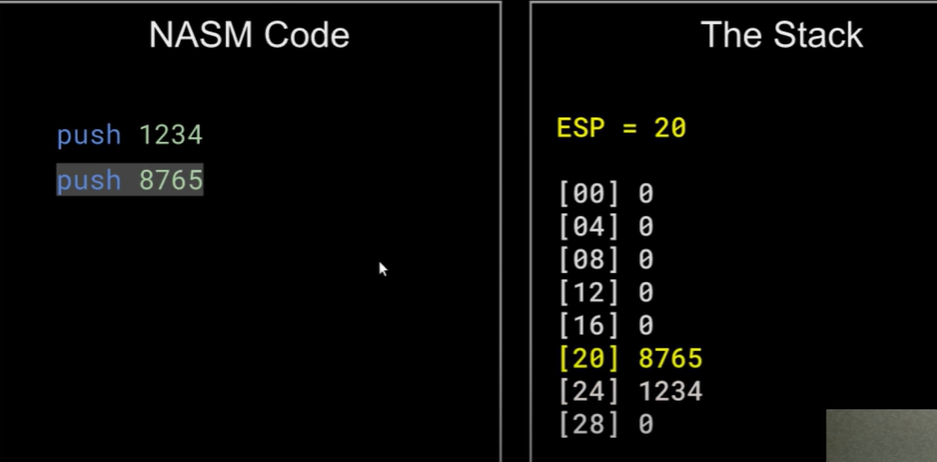


Demonstration of how stack works:

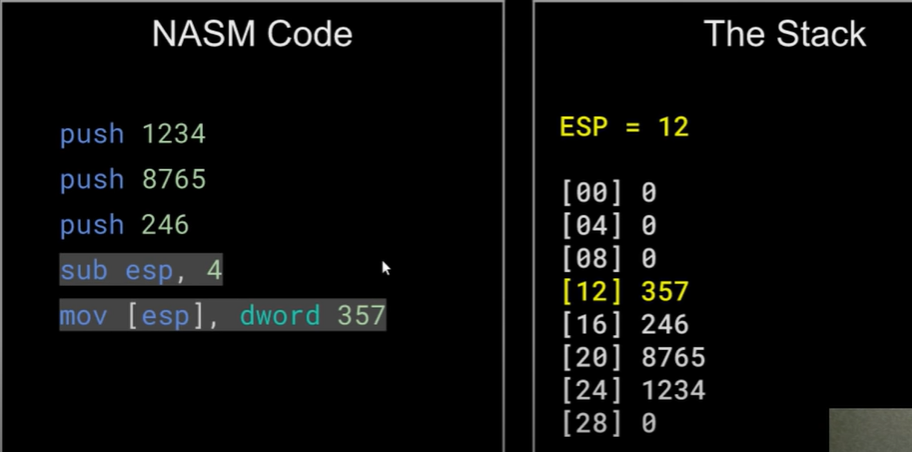


Stack pointer i.e ESP points at higher address 28 as soon as a new data(<=4 byte) is pushed the pointer next points to bit lower address 24 (lowering by 4 byte).





Instead of push, it can be represented as (the push operation for 357)



**Demonstration of function call in assembly:**

In assembly language, function calls are typically managed using the call and ret instructions. Here's an overview of how function calls work in assembly language:

1. Call Instruction (`call`):

- When a function is called, the `call` instruction is used to transfer control to the called function.

- Before executing the `call` instruction, the current value of the `EIP` (Instruction Pointer) is pushed onto the stack. This allows the program to return to the correct instruction after the function call is complete.

- The `call` instruction specifies the target function by its label or memory address. For example, `call my\_function` will call a function labeled `my\_function`.

2. Function Prologue:

- Inside the called function, a typical sequence of instructions, known as the "function prologue," is used to set up the function's stack frame.

- This often involves saving the values of certain registers, such as `EBP` (Base Pointer), on the stack. `EBP` is commonly used as a reference point for addressing local variables and function parameters.

- The function prologue may also allocate space on the stack for local variables by adjusting the `ESP` (Stack Pointer) register.

3. Function Body:

- The main body of the function performs its operations, which may include manipulating data, performing calculations, or calling other functions.

4. Function Epilogue:

- After completing its tasks, the function prepares to return to the caller by executing a "function epilogue."

- This often involves restoring the values of registers saved in the prologue, deallocating any stack space used for local variables, and cleaning up the stack.

- The `ret` instruction is used to return control to the calling function.

5. Return Instruction (`ret`):

- The `ret` instruction is used at the end of the called function to return to the caller.

- When executed, `ret` pops the previously saved value of `EIP` from the stack and transfers control to that address.

- This effectively resumes execution at the instruction immediately following the `call` instruction that initiated the function call.

Here's a simplified example in x86 assembly language:

```assembly

global \_start

section .text

\_start:

call my\_function ; Call a function named 'my\_function'

; Continue with the rest of the program

my\_function:

; Function prologue (save registers, set up stack frame)

push ebp

mov ebp, esp

; Function body (perform operations)

; ...

; Function epilogue (clean up, return)

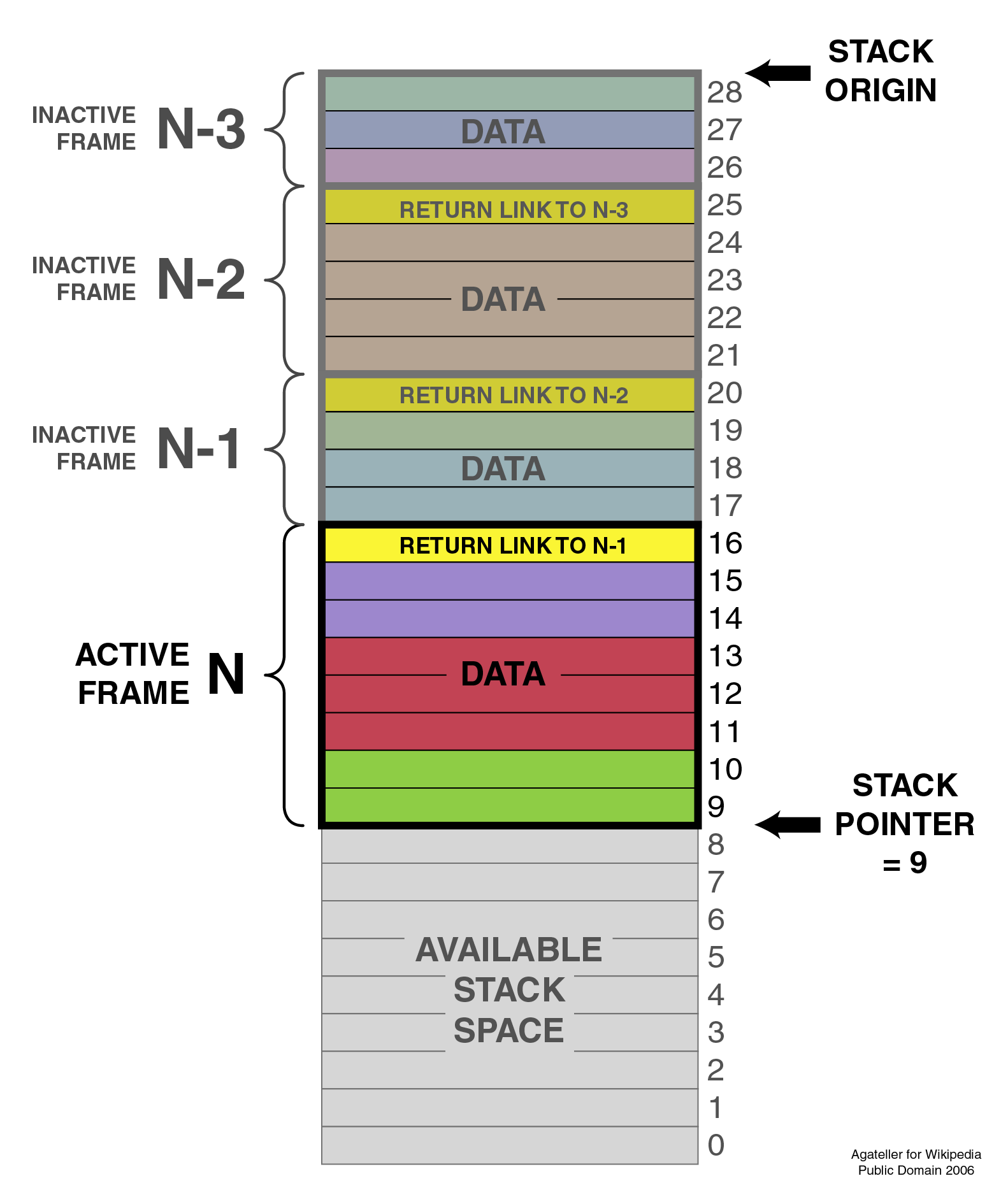
mov esp, ebp

pop ebp

ret

```

In this example, `\_start` calls `my\_function` using the `call` instruction, and `my\_function` performs its tasks before returning control to `\_start` using the `ret` instruction. The `EBP` register is used to reference the function's stack frame, and `ESP` is adjusted accordingly during setup and cleanup.



**Stack during function call**

