CSIT 5740 Introduction to Software Security

Note set 3B

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

Addressing memory

Addressing Memory

- Memory access can be composed of
  width, base, index, scale, and displacement

  - Base: starting address of reference
  - Index: offset from base address
  - Scale: constant multiplier of index
  - Displacement: constant base
  - Width: size of reference (byte, word, dword, qword → 8, 16, 32, 64 bits)
  - Address = base + index*scale + displacement

- Example
  - `mov dword ptr [eax+ecx*4+0x20], edx`

Instruction Classes

- Data transfer/memory related
  - `mov, xchg, push, pop, lea`
- Binary arithmetic
  - `add, sub, imul, mul, idiv, div, inc, dec`
- Logical
  - `and, or, xor, not`
- Stack handling
  - `push <register>` → decreases the stack pointer (esp/rsp) and saves the content of `<register>` in the newly pointed location
  - `pop <register>` → saves the content pointed by the stack pointer (esp/rsp) in `<register>` and increases the stack pointer

Instruction Classes

- Control transfer/function call
  - `jmp, call, ret, int, iret`
- Values can be compared using the `cmp` instruction
  - `cmp dest, src`

```
    if dest-src<0, set ZF=0, CF=1
    if dest-src==0, set ZF=1, CF=0
    if dest-src>0 set ZF=0, CF=0
```

> Recall from slide 34:
> If dest < src, then the flags will be **ZF = 0**, **CF = 1**
> If dest == src, then the flags will be **ZF = 1**, **CF = 0**
> If dest > src, then the flags will be **ZF = 0**, **CF = 0**

- Various eflags bits are set accordingly
  - `jne (ZF=0), je (ZF=1), jae (CF=0),...`

Instruction Classes

- Control transfer can be direct (destination is a constant) or indirect (the destination address is the content of a register)
- In machine code jumps are encoded as relative addresses (e.g., `jmp +5`) → this has consequences when moving machine code in memory
- Misc
    - `nop (0x90)`

Endianess (and of Signed Integers)

- As we have discussed, Intel uses little endian ordering
  - For instance, if the value 0x03020100 is stored at address 0x00F67B40 the memory content is

address 00F67B43 →	0x30
address 00F67B42 →	0x20
address 00F67B41 →	0x01
address 00F67B40 →	0x00

- Signed integers are expressed in 2’s complement notation
- The sign is changed by flipping the bits and adding one
  - 0xFFFFFFFF is -1, 0xFFFFFFFE is -2, …

Invoking System Calls

- System calls effectively function calls to perform system level tasks (i.e. create a file, allocate memory space, etc). System calls are usually invoked through libraries (e.g., libc library of Linux)
- However, we can invoke them directly in assembly
  - Linux/x86 (32-bit): int 0x80
    - eax contains the system call number
    - https://syscalls32.paolostivanin.com/
  - Linux/x86_64 (64-bit): syscall
    - rax contains the system call number
    - https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

Some Tricks

- Use `nasm` to compile assembly directly
  - To create a 64bit program:
    - `nasm -f elf64 hello.asm && ld hello.o && ./a.out`

- `int3`
  (encoded as 1 byte: 0xCC)

  → it generates a “trap” interrupt, debuggers can catch it and stop the execution, useful to inspect assembly code

References

- Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture
- Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2: Instruction Set Reference, A-Z
- Wikipedia
  - https://en.wikibooks.org/wiki/X86_Assembly
- Online x86 / x64 Assembler and Disassembler
  - https://defuse.ca/online-x86-assembler.htm

Hello World! (64bit)

```
section .data
str: db "Hello world!"
section .text
global _start
_start:
        mov rax,1       ; write syscall
        mov rdi,1       ; stdout
        mov rsi,str     ; string address
        mov rdx,13      ; string length
        syscall
        mov rax,60      ; exit syscall
        mov rdi,0       ; exit code
        syscall
```

Hello World! (64bit)

```
section .data
str: db "Hello world!"
section .text
global _start
_start:
        :        :
        :        :
        :        :
        :        :
        :        :
```

“section .data” is the nasm assembler directive, indicating a “data segment”

“section .text” is the nasm assembler directive, indicating a “text segment”

global directive to indicate the scope of the _start label (not very important now)

Hello World! (64bit)

```
section .data
str: db "Hello world!",0Ah
section .text
global _start
_start:
        :               :
        :               :
        :               :
        :               :
        :               :
```

"str" is the label name for the string "Hello world!",0Ah
0Ah is the new line character.
"db" indicates we are defining bytes (characters here)

Data types	Meaning
db	Byte datatype (1 byte)
dw	Word datatype (2 bytes)
dd	Doubleword datatype (4 bytes)
dq	Quadword datatype (8 bytes)

"_start" is the label name for the beginning of the program

Hello World! (64bit)

```
section .data
str: db "Hello world!",0Ah
section .text
global _start
_start:
        mov rax,1               ; write syscall
        mov rdi,1               ; stdout
        mov rsi,str             ; string address
        mov rdx,13              ; string length
        syscall
        mov rax,60              ; exit syscall
        mov rdi,0               ; exit code
        syscall
```

NR	syscall name	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
1	write	0x01	unsigned int fd	const char *buf	size_t count

Syscall 1, the write syscall to output the string

Write to the standard output (i.e. stdin=0, stdout=1,stderr=2)

Address of the string is in rsi register

string length is in rdx register

Do the syscall with the arguments in rax, rdi, rsi and rdx

Hello World! (64bit)

```
section .data
str: db "Hello world!",0Ah
section .text
global _start
_start:
        mov rax,1          ; write syscall
        mov rdi,1          ; stdout
        mov rsi,str        ; string address
        mov rdx,13         ; string length
        syscall
        mov rax,60         ; exit syscall
        mov rdi,0          ; exit code
        syscall
```

NR	syscall name	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
60	exit	0x3c	int error_code	-	-

Syscall 60, the exit syscall

Exit error code 0

List of linux syscalls can be found at
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

Stack and stack frame

The Stack

- The stack is a special memory region, used to store information (e.g., variables) of a specific function call
- Typically a “frame” of storage space is allocated on the stack for a specific function call, this frame is typically known as a **stack frame** (sometimes aka activation record of a function).
- In Intel processors
    - The beginning of the stack is pointed to by ebp/rbp register
    - The top of the stack is pointed to by the esp/rsp register
    - The stack grows towards lower memory addresses
    - The stack it is a last-in-first-out (LIFO) data structure

Recap: The "push" instruction

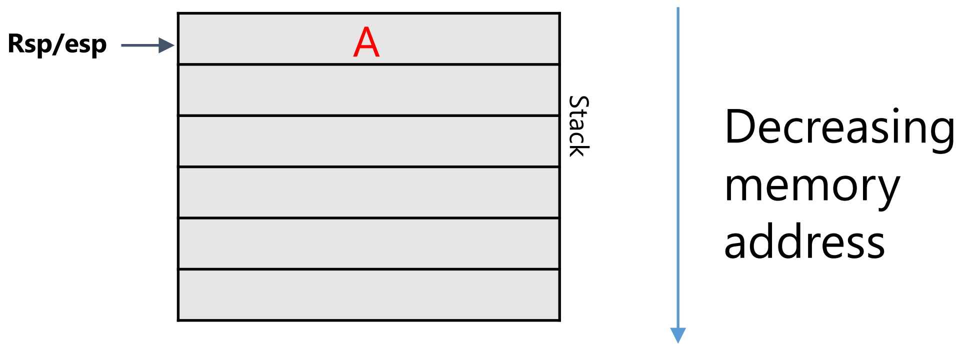
- The **`push`** instruction "pushes/stores" a piece of data to the stack and decreases the stack pointer
- **`push <register>`** → decreases the stack pointer (esp/rsp) and saves the content of **`<register>`** in the newly pointed location
- For example "**`push rax`**"
  - Will decrease the stack pointer (rsp/esp) by 8 to allocate 8 bytes of new space on the stack
  - Then it will put the 8-byte rax regiser value into the newly allocated 8-byte space on the stack
  - more on this with the help of a picture

Recap: The "pop" instruction

- The **pop** instruction "pops/retrieves" a piece of data from the stack and increases the stack pointer
- **pop <register>** → retrieves the last piece of data (i.e. top) from the stack and stores it to **<register>**, then it increases the stack pointer (esp/rsp) to delocate the data from the stack (i.e. the data will be out of the stack boundary)
- For example "**pop rax**"
  - Will copy 8-byte of data on the top of the stack to the 64-bit register
  - Then it will increase the stack pointer (rsp/esp) by 8 to de-locate 8 bytes of space from the stack
  - more on this with the help of a picture

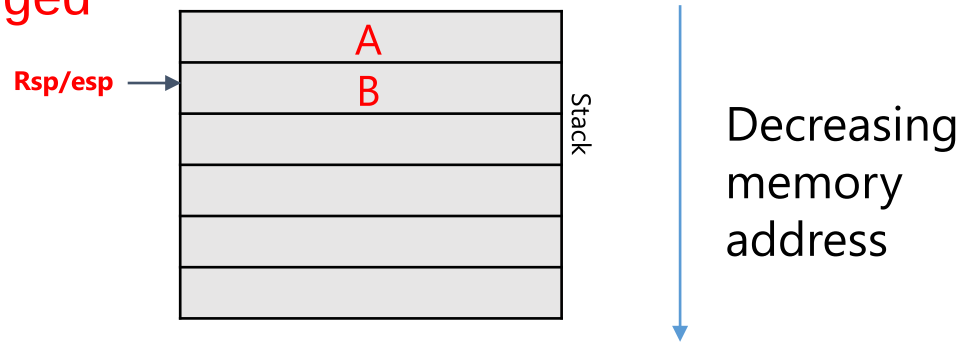
The Stack

- Original stack with 1 element, the only element “A” is pointed by the `rsp/esp` register to indicate it is the last element of the stack (i.e. the “**top**” in data structure term)



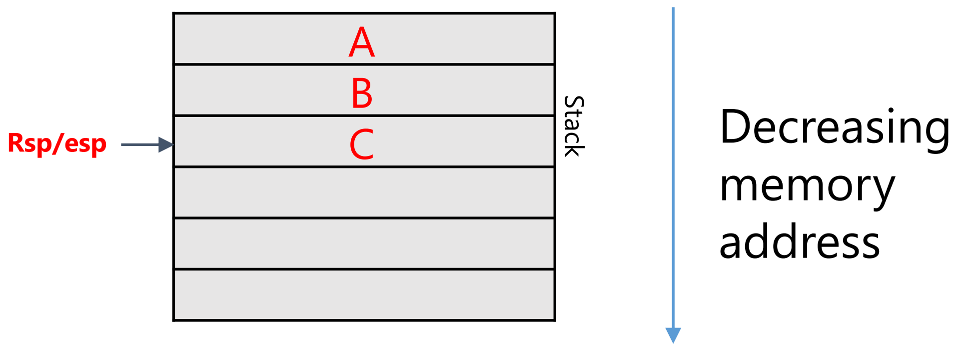
The Stack

- Now we push the data B (in register `rax`) to the stack
    - `push rax`
- The last element pushed, B, becomes the new top
    - data B is copied to the stack,
    - the rsp/esp is also changed



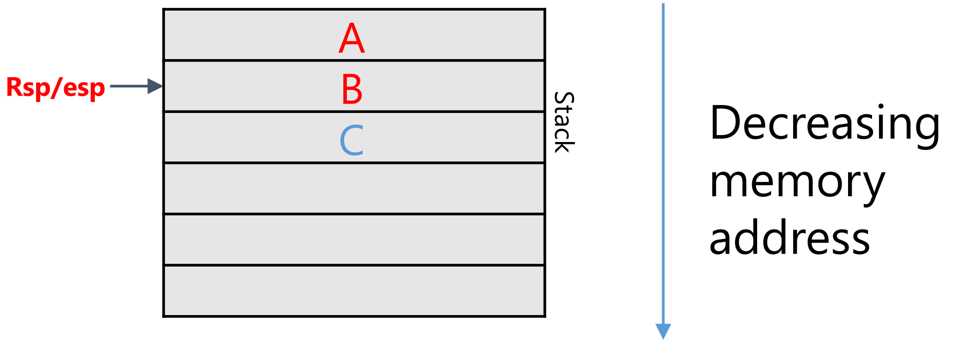
The Stack

- Now we push another piece of data C (in register `rdx`) to the stack
    - `push rdx`
- The last element pushed, C, becomes the new top



The Stack

- Now we pop the last data C a register, say the `rcx` register
    - `pop rcx`
- B becomes the new last element, C is still there but no longer considered part of the stack
- C is in the `rcx` register after the `pop` instruction has run
- It is very obviously **Last-In-First-Out** data structure



Stack Frames

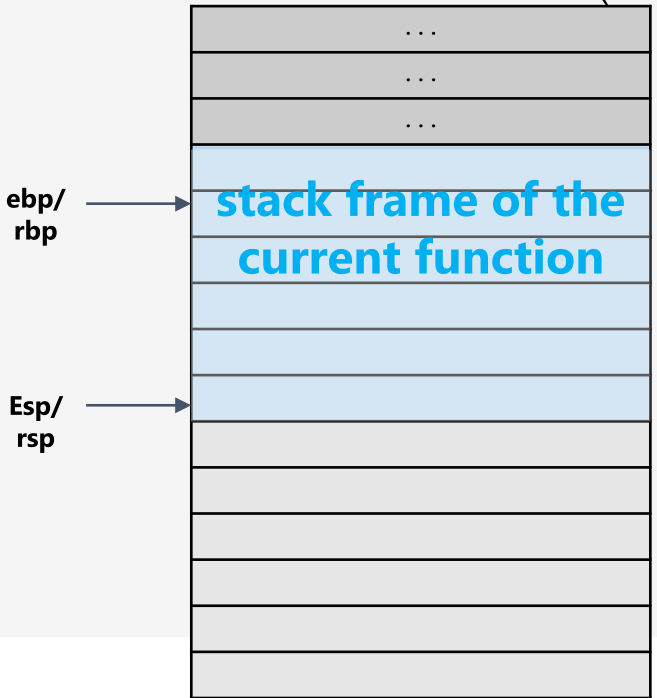
- Besides the last element we also need to know where the stack starts (i.e. where is the first element). The space from the first element to the last element of the stack is known as the **stack frame**. But sometimes, the stack frame could start slightly earlier (will see this in the future slides)

- When your program calls a function, space is made on the stack for local variables
  - This is the **allocated stack frame** for the function
  - The **allocated stack frame is de-located** once the function returns

- The stack starts at higher addresses. Every time your program calls a function, the stack makes extra space by growing downwards

Stack Frames

- Arguments and data are pushed on the stack as a consequence of function calls (function prologue)
- To maintain a stack frame, x86 uses two pointer registers
  - (ebp/rbp) points to the first element (i.e. beginning) of the stack frame. Note the stack frame here starts earlier than that. Because in the x64 function call convention we are following, the stack starts 8 bytes before ebp/rbp
  - (esp/rsp) points to the current last element (i.e. end) of the stack frame



Stack Frames

- Each frame contains
  - The function’s actual parameters
  - The return address to jump to at the end of the function
  - The pointer to the previous frame
  - The function’s local variables

Calling Conventions

x86 function call

```
void caller(){
        callee(1,2);
    }
```

The caller function, aka the "caller"

```
void callee(int x, int y){
        int local_var1=3;
        return 22
}
```

The function being called, aka the "callee"

Calling Conventions

- Calling conventions determine how the code would do tasks in the instruction level:
  - How to pass parameters between caller and callee
  - What registers need to be saved
- cdecl (stands for “C declaration” used by Linux 32 bit):
  - Caller pushes arguments on the stack (right to left)
  - eax, edx, ecx are caller-saved (callee can overwrite them with data)
  - Return value in eax
  - Caller cleans up the stack afterwards
  - Cons: Cleanup code needs to be replicated at each function call position

Calling Conventions

- stdcall (used by the Win32 API):
  - Caller pushes arguments on the stack
  - Callee cleans up the stack
  - Cons: no variadic functions (i.e. functions must have fixed numbers of arguments)

- SysV AMD64 **(used by Linux 64 bit)**
  - First six integer arguments are passed in registers (rdi, rsi, rdx, rcx, r8, r9)
  - Additional arguments are put on the stack
  - Return value in rax
  - Caller cleans up the stack afterwards

Calling Conventions

- Calling conventions are just that: conventions.
  - They are not enforced by the processor, but must be adhered to when communicating with external functions or libraries.

- Compilers can decide to ignore them
  - especially when optimizations are enabled (-O1, -O2, ...)

Function Calls

Before function call

```
int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    funct();

    return 0;
}
```

Caller

During function call

```
void funct() {
    int fb = 1;
    return;
}
```

Callee

After function returns

```
int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    funct();

    return 0;
}
```

Caller

The **caller** function (**`main`**) calls the **callee** function (**`funct`**).

The callee function executes and then returns (the control) to the caller function.

Calling Conventions

- In the following discussion, we will assume a 32-bit assembly program for easier illustration (64-bit will make the values too wide) ☺
- The key ideas remain the same

x86 Calling Convention

- We will be using 32-bit examples to explain, but the idea is the same for 64-bit program (usally we just need to change the registers to the 64-bit version)
- How to pass arguments
    - In the AMD64 convention the first 6 arguments are copied to the following registers in a non-syscall function call
        - rdi, rsi, rdx , rcx , r8 , r9
    - In the AMD64 convention, in a syscall function call, the first 6 arguments are copied to
        - rdi, rsi, rdx , r10, r8 , r9 (the only change is rcx->r10, because syscall will clobber rcx, destroying the argument passed)
        - Returned value of the syscall will be in rax
    - Further arguments (i.e. 7th argument and above) are pushed onto the stack in reverse order, so **`func(arg0, arg1,…,arg6,arg7,arg8)`** will place **`arg8`** at the highest memory address, then **`arg7`**, then **`arg6`**

x86 Calling Convention

- How to receive return values
  - Return values are passed in RAX/EAX
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it

x86 Calling Convention

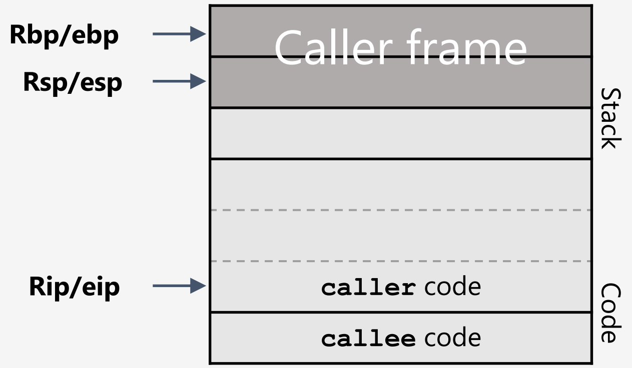
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it

Register	Usage	callee saved
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–%r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes
%r16–%r31	temporary registers	No
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%xmm16–%xmm31	temporary registers	No
%tmm0–%tmm7	temporary registers	No
%mm0–%mm7	temporary registers	No
%k0–%k7	temporary registers	No
%st0,%st1	temporary registers, used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	thread pointer	Yes
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes
tilecfg	Tile control register	No

These are the **callee saved registers**

Calling a Function in x86

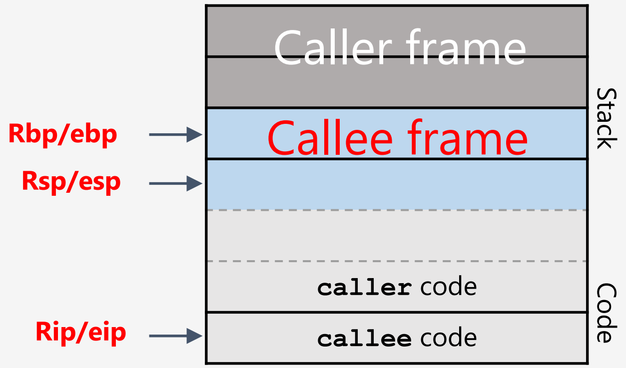
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



Before function call

Calling a Function in x86

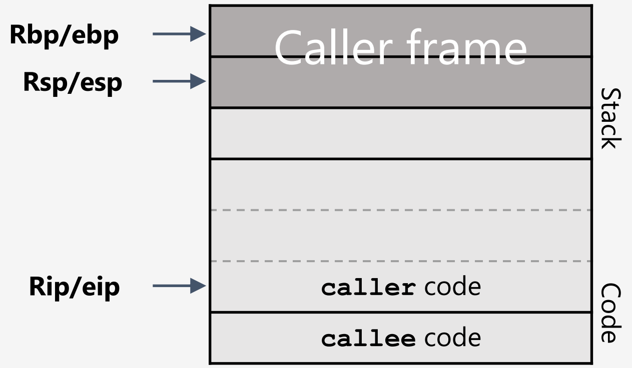
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



During function call

Calling a Function in x86

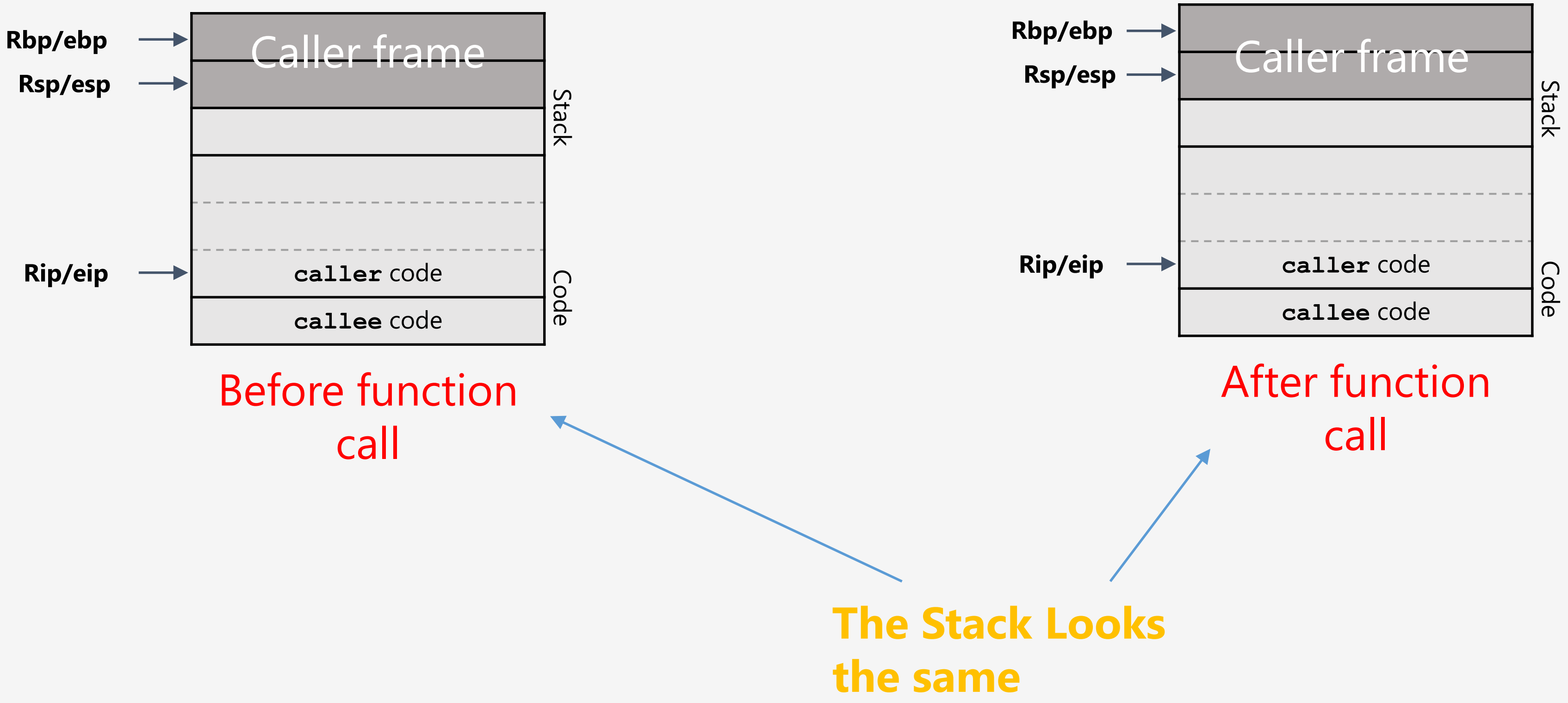
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



After function call

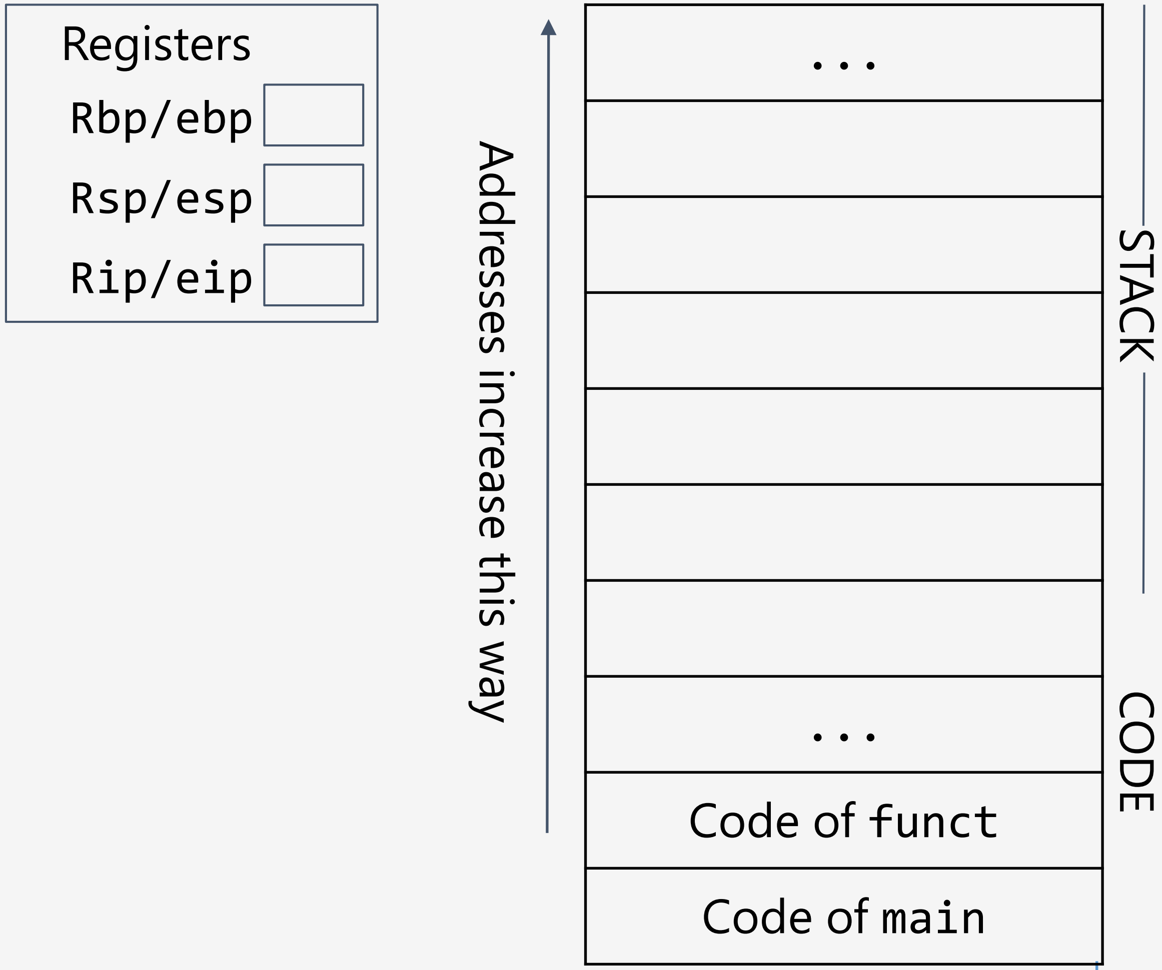
Calling a Function in x86

- Before and after a function call, the stack be the same to the caller, otherwise the caller will have trouble to find its data



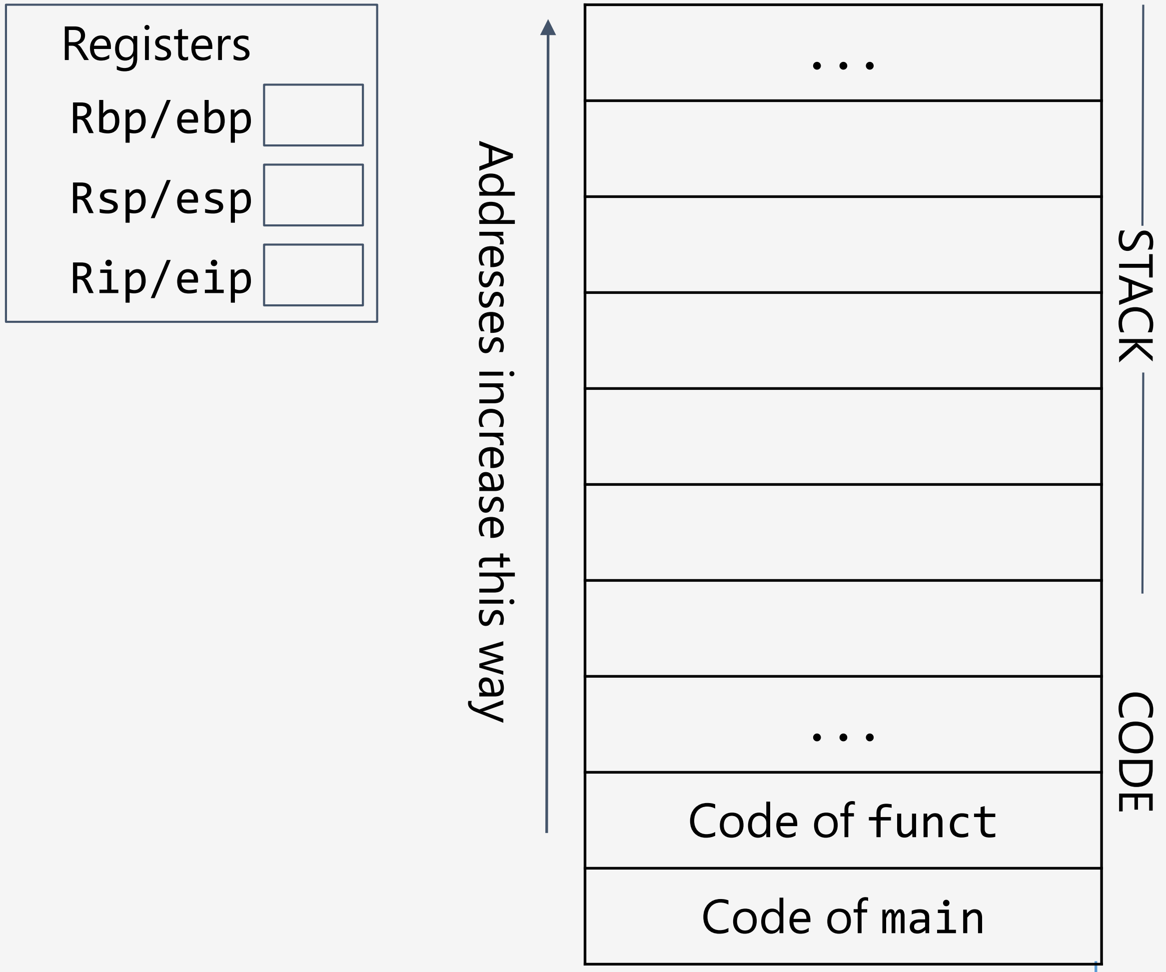
Review: The stack

- We are just showing the stack, and also the text segment holding the programs, as they are the most relevant part for a function call
- Each row of the diagram is 32 bits in width.
- Addresses increase to the top direction (i.e. north)



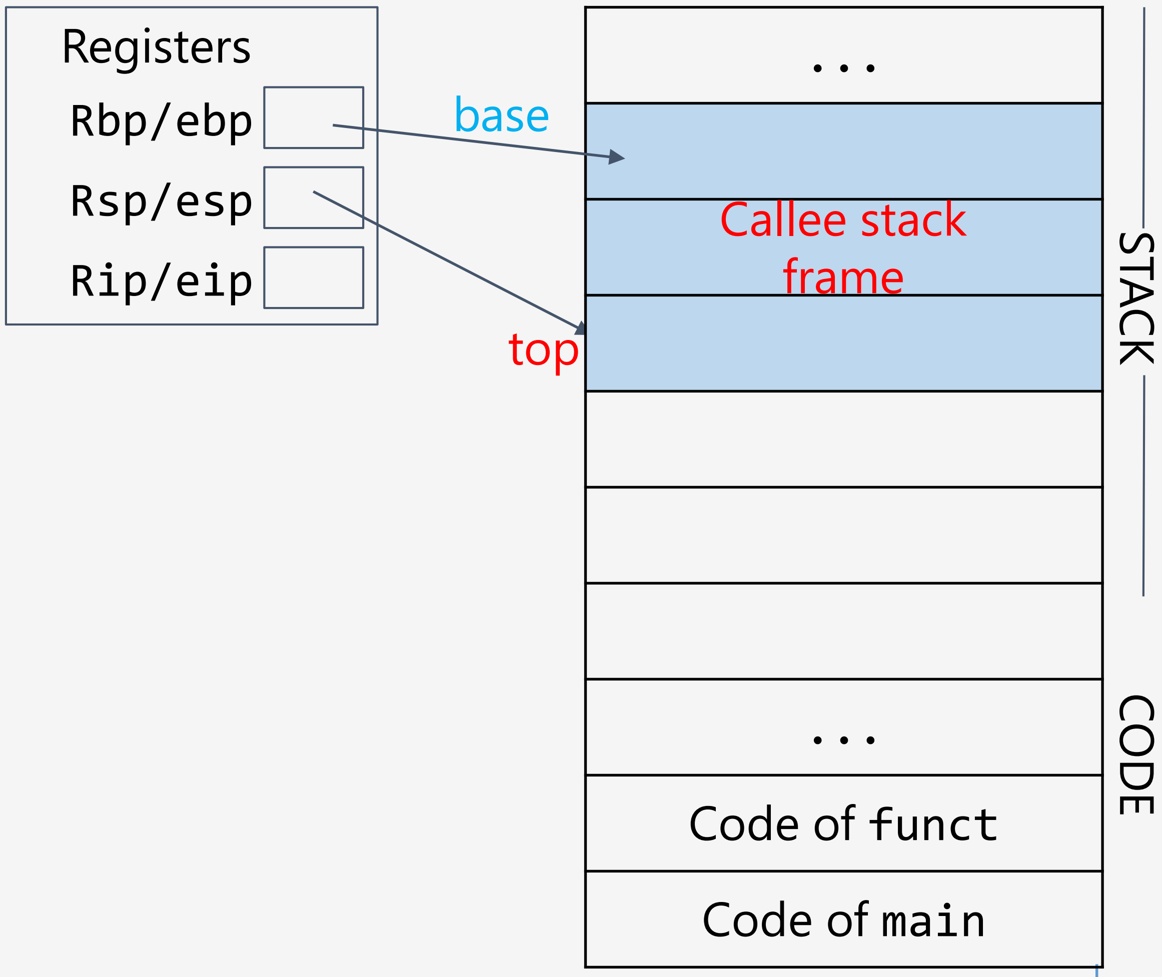
Review: The function stack frame

- Two pointers are use for indicating the part of the stack that is being used by the current function.
- As we have mentioned earlier, this part of the stack is called a **stack frame**.
- One stack frame corresponds to part of the stack allocated to a single function call.



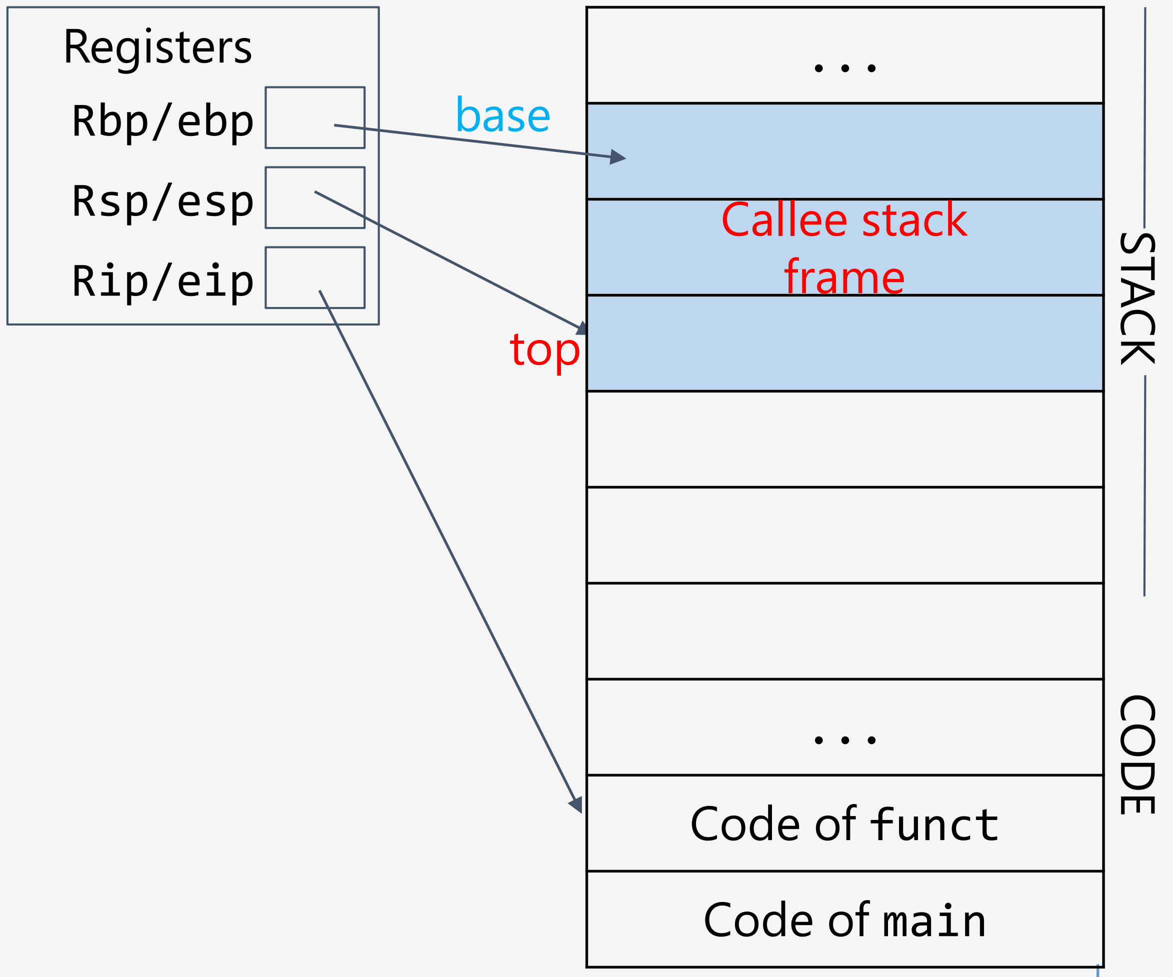
Maintaining the stack frame using rbp/ebp and rsp/esp

- We use registers to hold two pointers
- The two pointers show the start and end of the stack frame allocated to the current function
- rbp/ebp indicates the start/base of the stack frame, `rsp/esp` indicates the top of the stack frame (i.e. the last element of the current stack frame allocated to the function)
- If we **push** a new data onto the stack, rbsp/`esp` must move down to allocate space for the new data
- Each stack frame have space for local variables of the function (added through pushes).



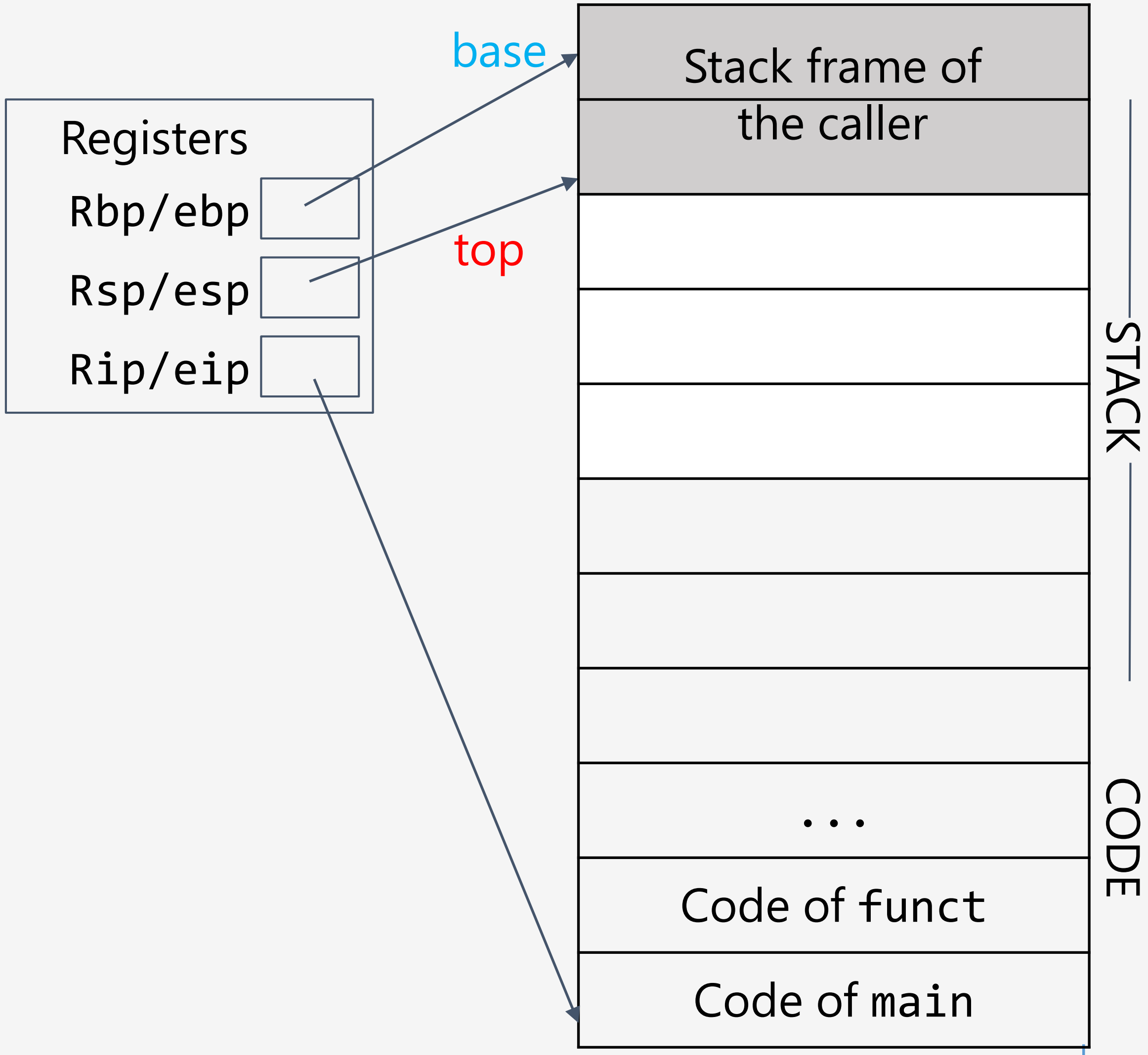
The instruction pointer rip/eip register

- It is also important to know which instruction of the function we are currently executing
- The instruction register pointer, rip/`eip` stores a pointer that points to the current instruction being executed



Designing the stack

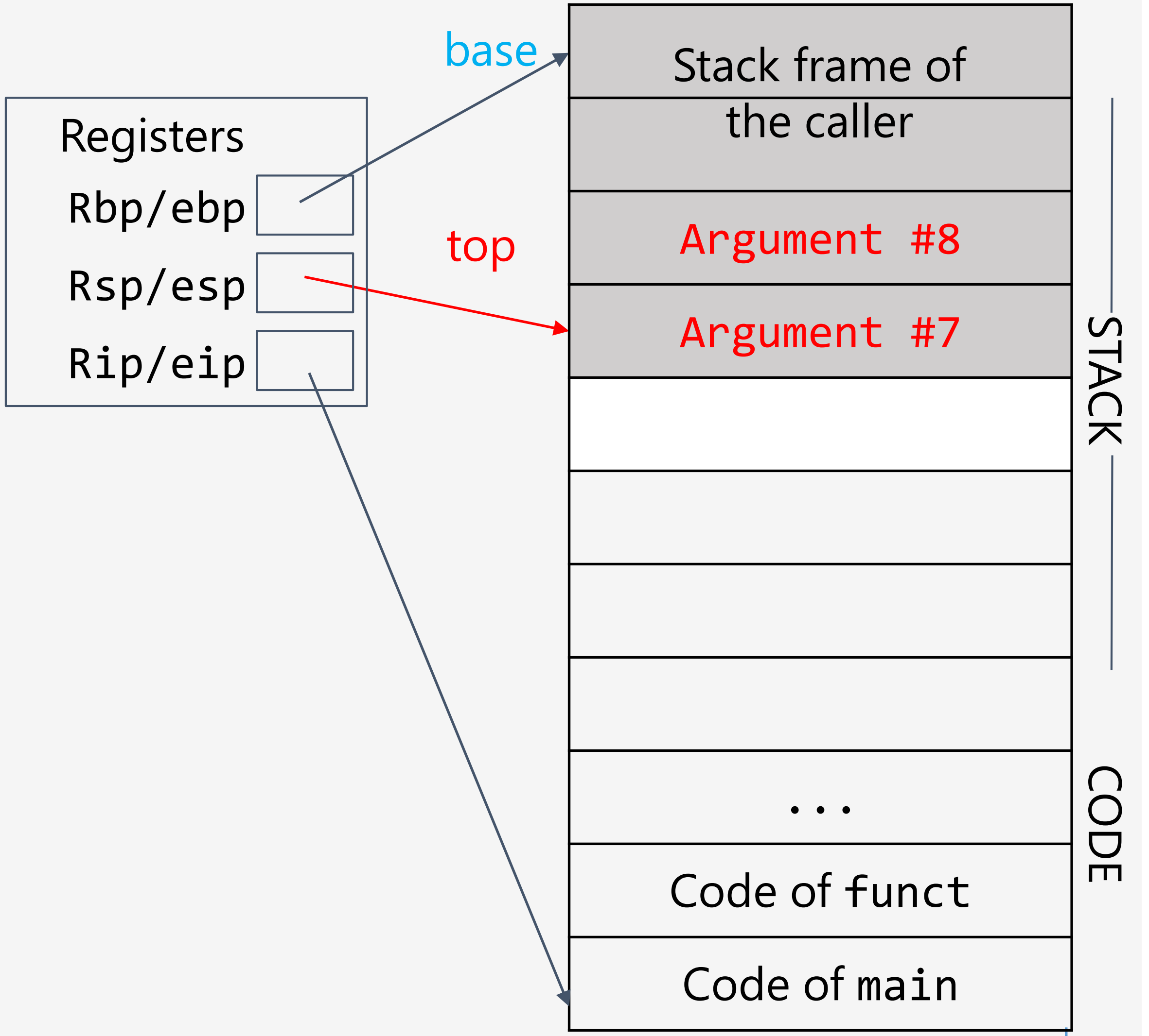
- Every time a function is called, a new stack frame is created for it
- **When the function returns, the stack frame allocated to the function must be discarded**. And the caller's stack frame is restored by updating rbp/ebp and rsp/esp registers
- The stack frame of a function is the place where the function's **local variables** are stored
- Arguments for the function call is also stored somewhere in the stack
- Also, we need to follow the function calling convention, if we overwrite a **callee saved register (aka saved register)**, we should remember its old value by putting it on the stack.



1. Function arguments

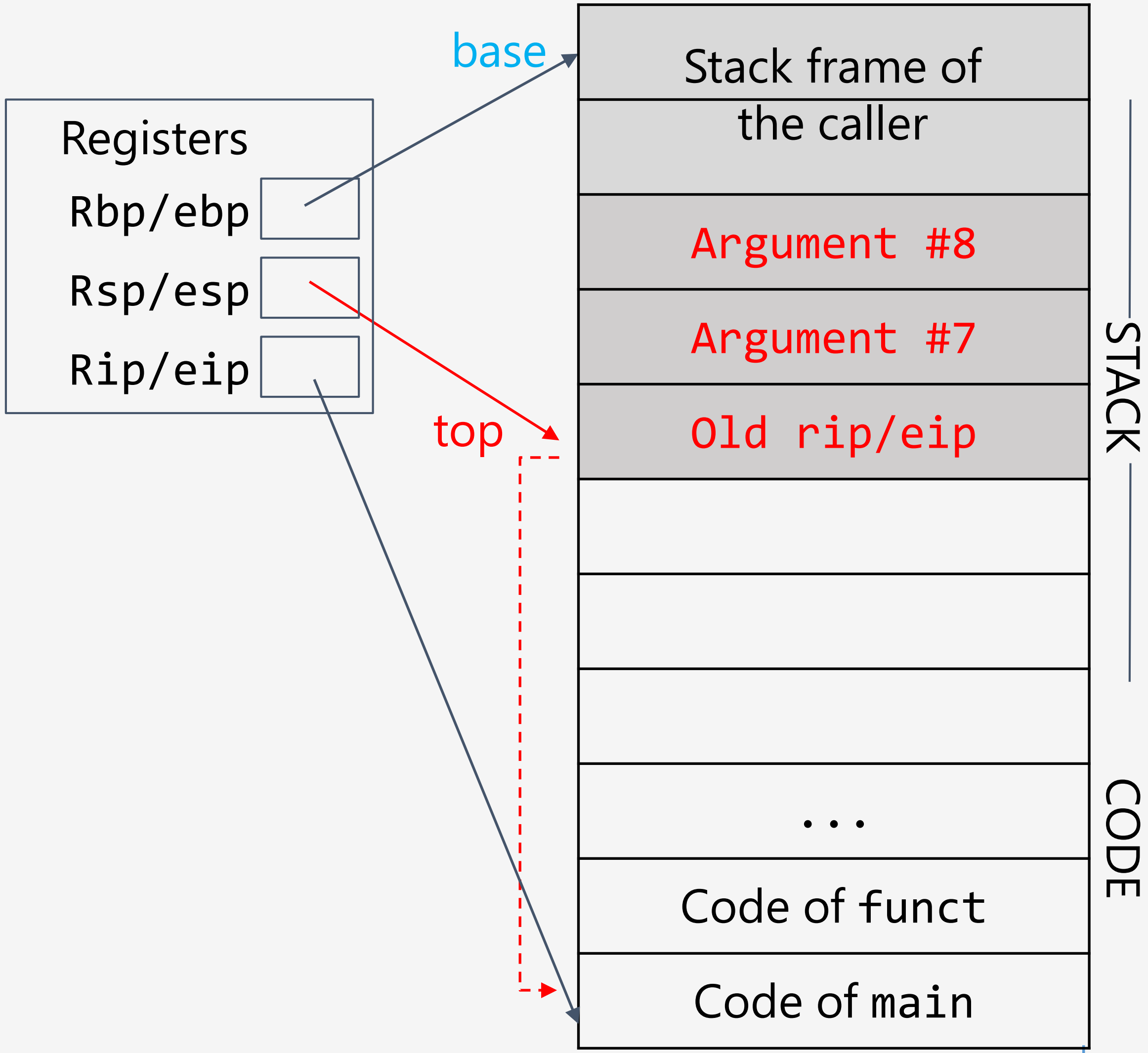
- The first 6 arguments of the function call is copied to the corresponding registers
  - In a non-syscall function call the first 6 arguments are copied to
    - rdi, rsi, rdx , rcx , r8 , r9
  - In a syscall function call, the first 6 arguments are copied to
    - rdi, rsi, rdx , r10, r8 , r9

- The 7th and later arguments are put to the stack. The 7th argument is put at the lowest address and then the 8th, 9th, 10th and so on (i.e. arguments are added to the stack in reverse order.). In the picture, we only assume 8 arguments being passed to the function in total



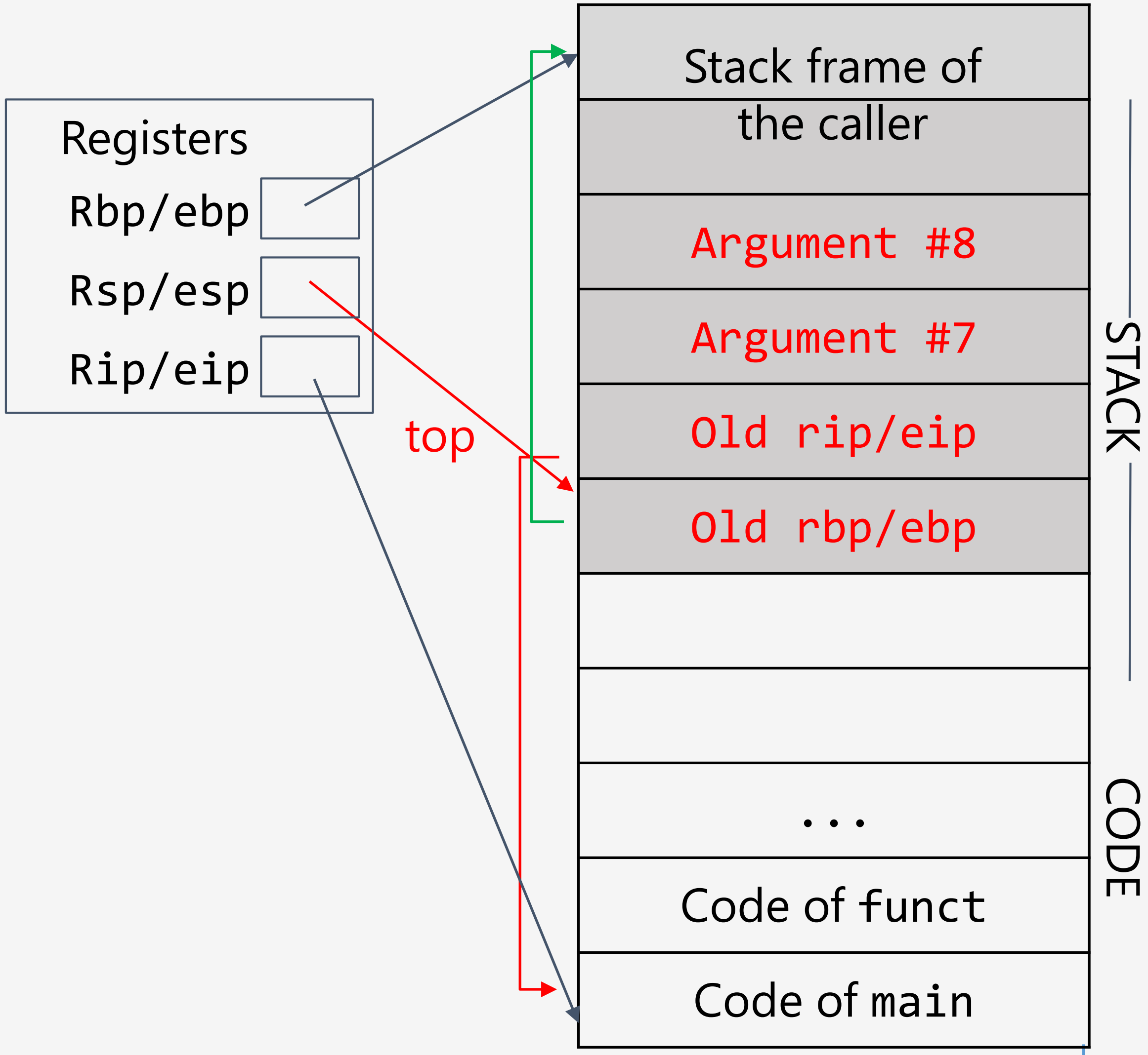
2. The return address in rip/eip

- Next, push the current value of rip/`eip` on the stack.
  - This tells us what code to execute next after the function returns
- Remember to adjust rsp/`esp` to point to the new lowest value on the stack.
- This value is also known as the `rip` (return instruction pointer), this pointer tells us which instruction in the caller to resume after finishing the function call.



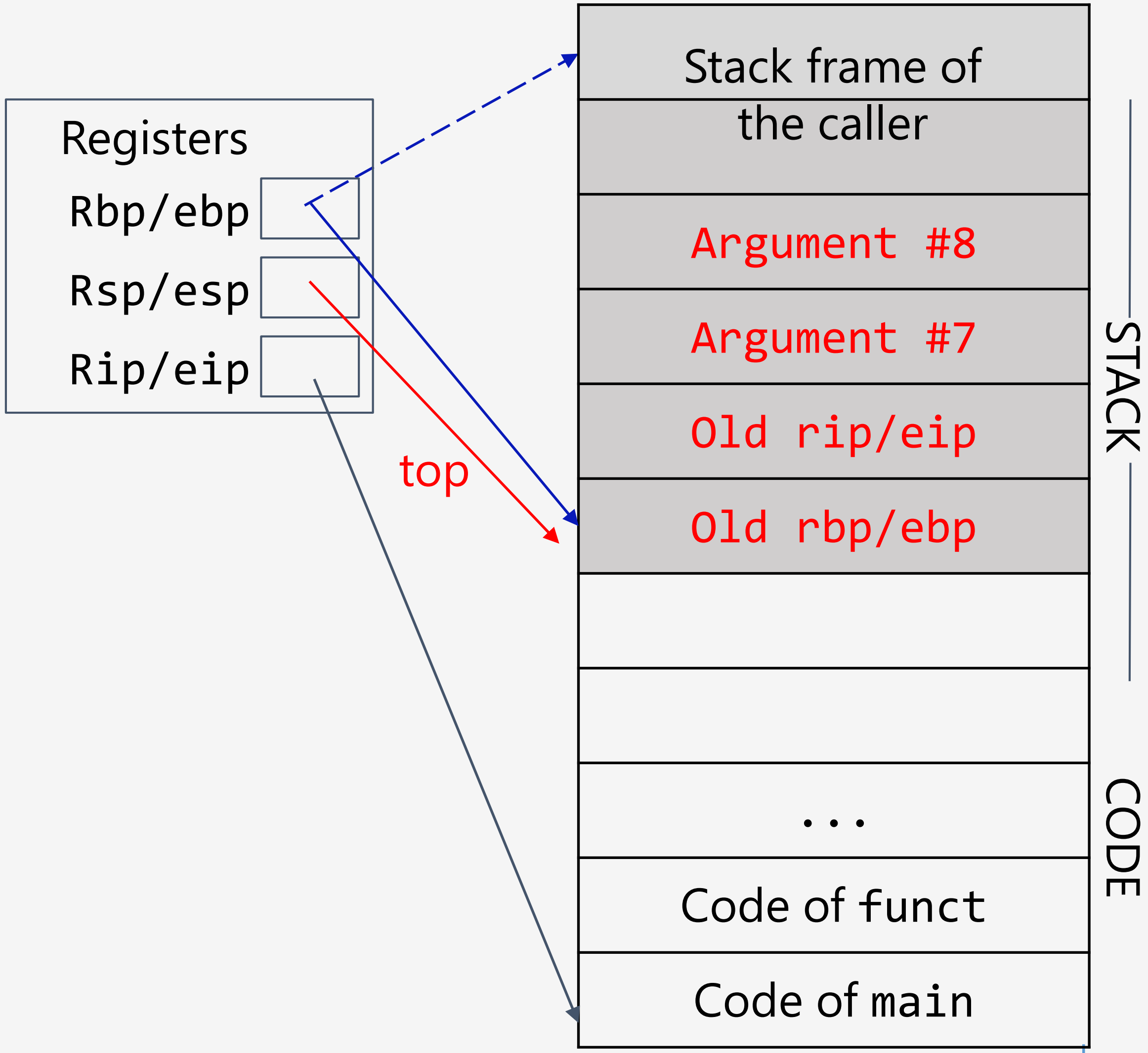
3. Remembering rbp/ebp

- Now, push the current value of rbp/ebp to the stack.
  - This will let us restore the top of the caller stack frame when we return
  - Alternate interpretation: rbp/ebp is a saved register. We store its old value on the stack before overwriting it.
- Mind that `rsp/esp` is also adjusted to point to the new top of the stack (i.e. the stack holds one more element).



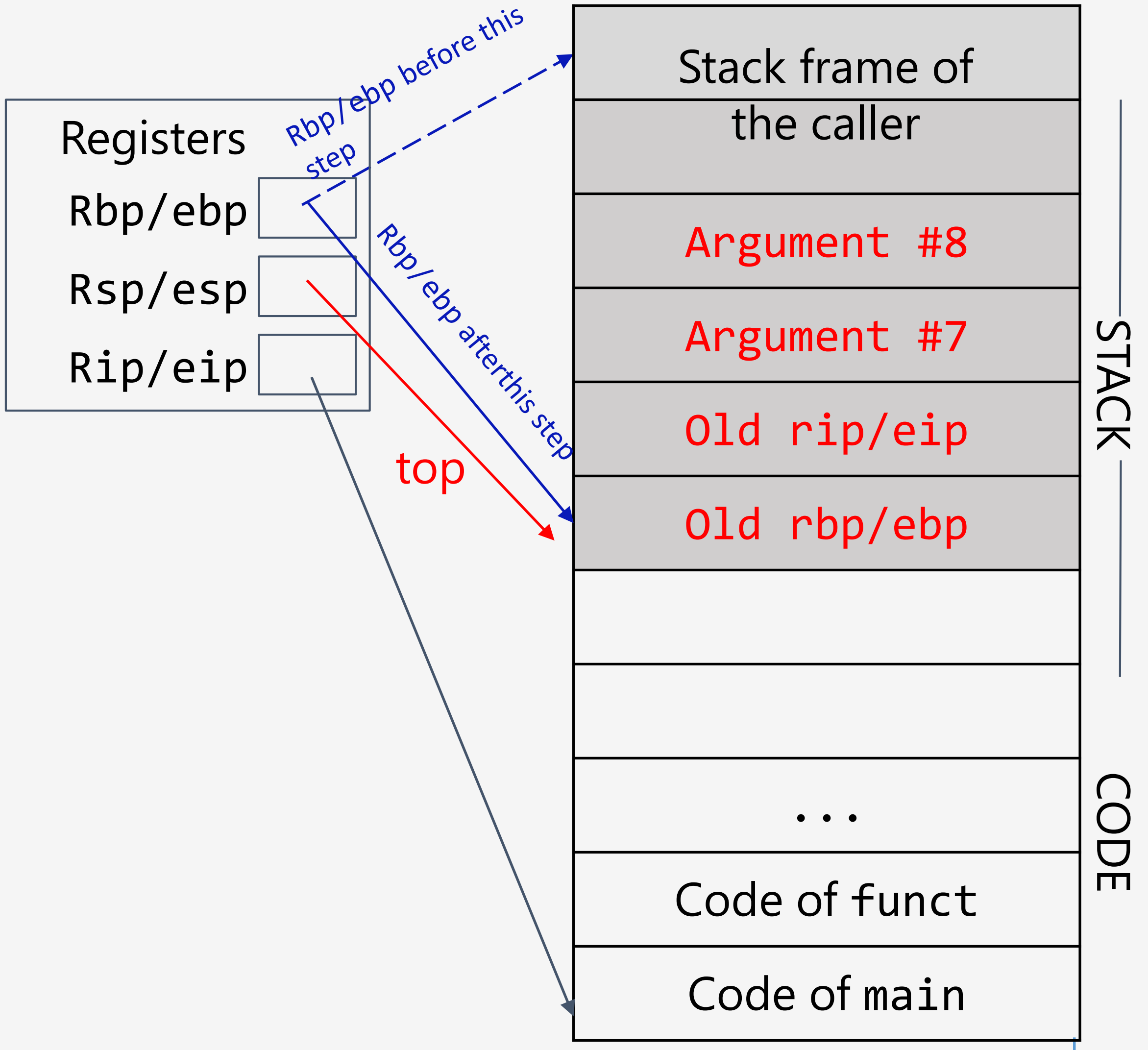
4. Adjust and allocate the stack frame to the callee

- To adjust the stack frame, we need to update all three registers.
- We can do this because we've just saved the old values of rbp/`ebp` and rip/`eip`. (rsp/`esp` will always be the bottom of the stack, so there's no need to save it).
- Rbp/ebp now points to the top of the current stack frame.
- What will happen if we haven't saved them but still update all the 3 registers?



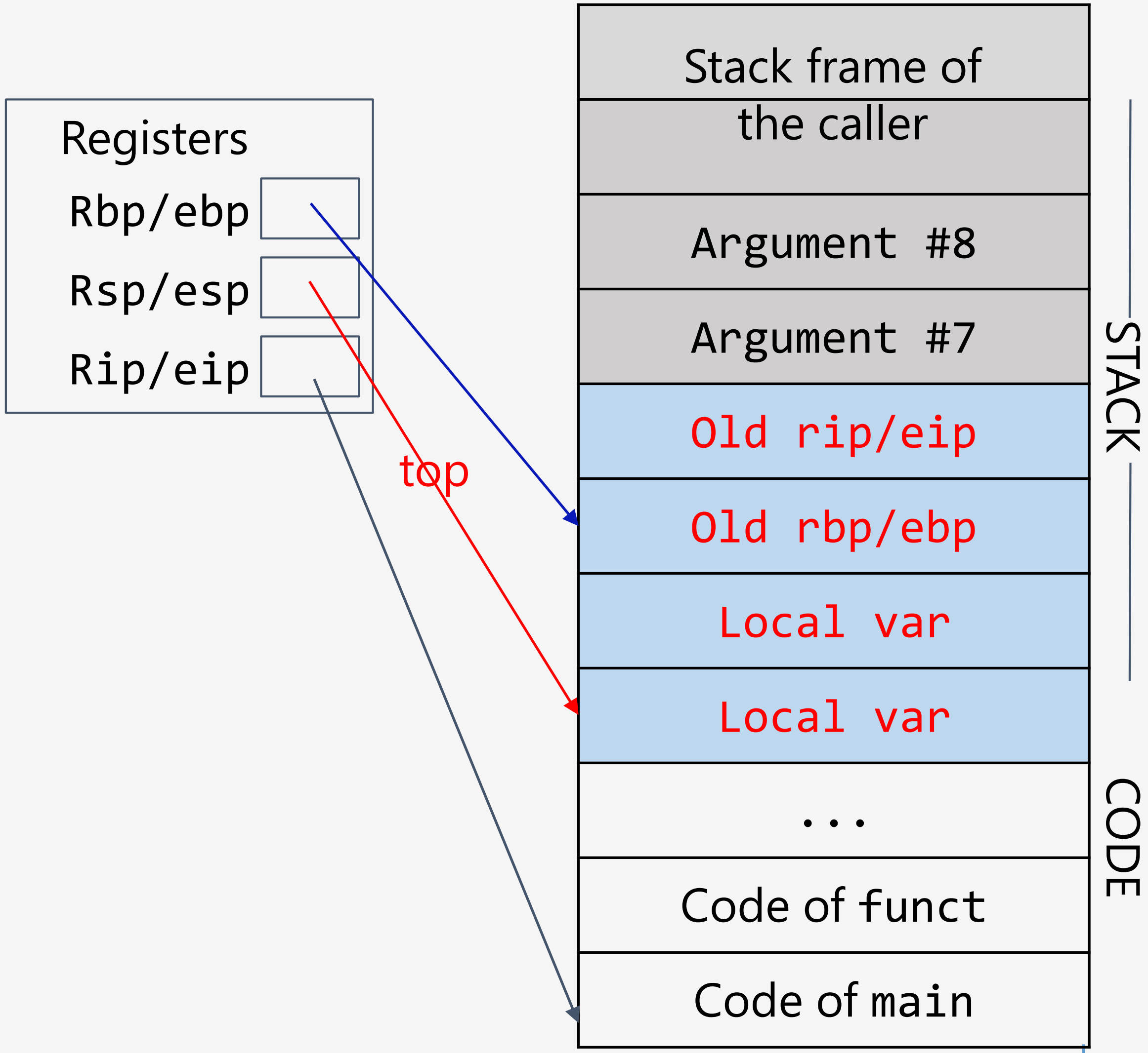
4. Adjust and allocate the stack frame to the callee

- To adjust the stack frame, we need to update all three registers.
- We can do this because we've just saved the old values of rbp/ebp and rip/eip. (rsp/esp will always be the bottom of the stack, so there's no need to save it).
- Rbp/ebp now points to the top of the current stack frame.
- What will happen if we haven't saved rbp/ebp and rip/eip but still update all the registers?



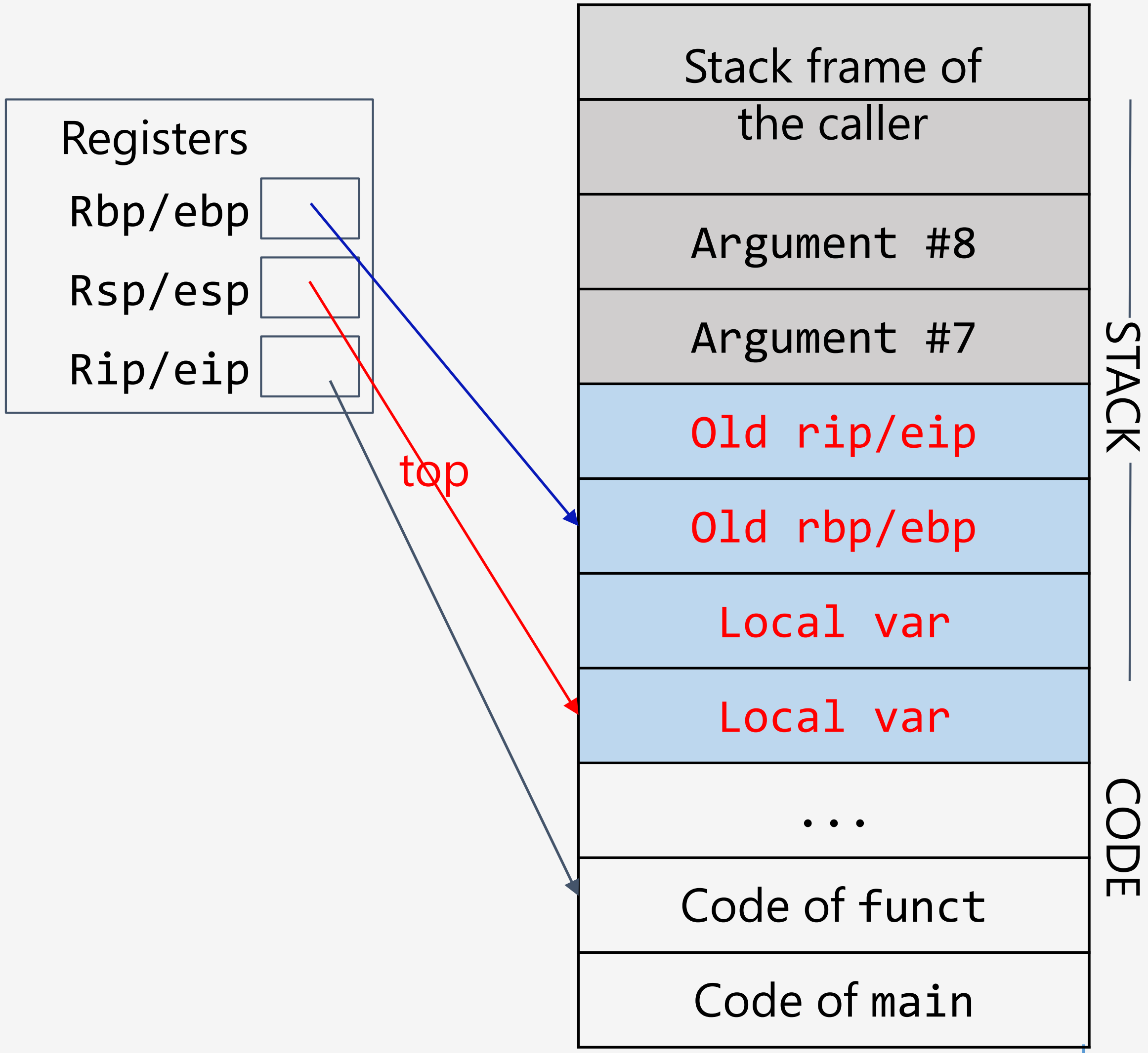
4. Adjusting the stack frame

- `Rsp/esp` now points to the bottom of the current stack frame.
- The compiler determines the size of the stack frame by checking how much space the function needs (how many local variables it has).



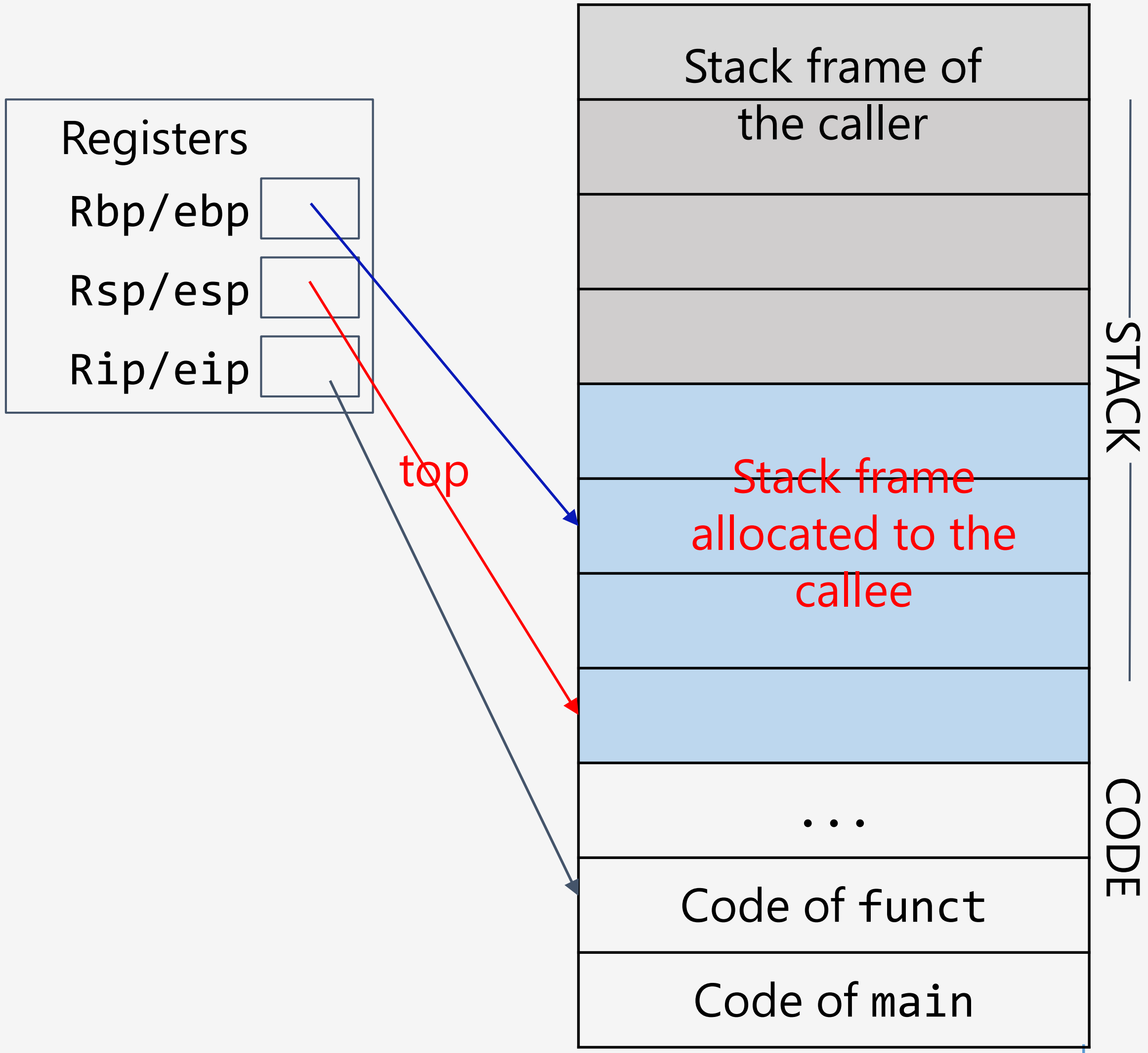
4. Adjusting the rip/eip instruction pointer

- `Rip/eip` now points to the first instruction of the function `funct`.



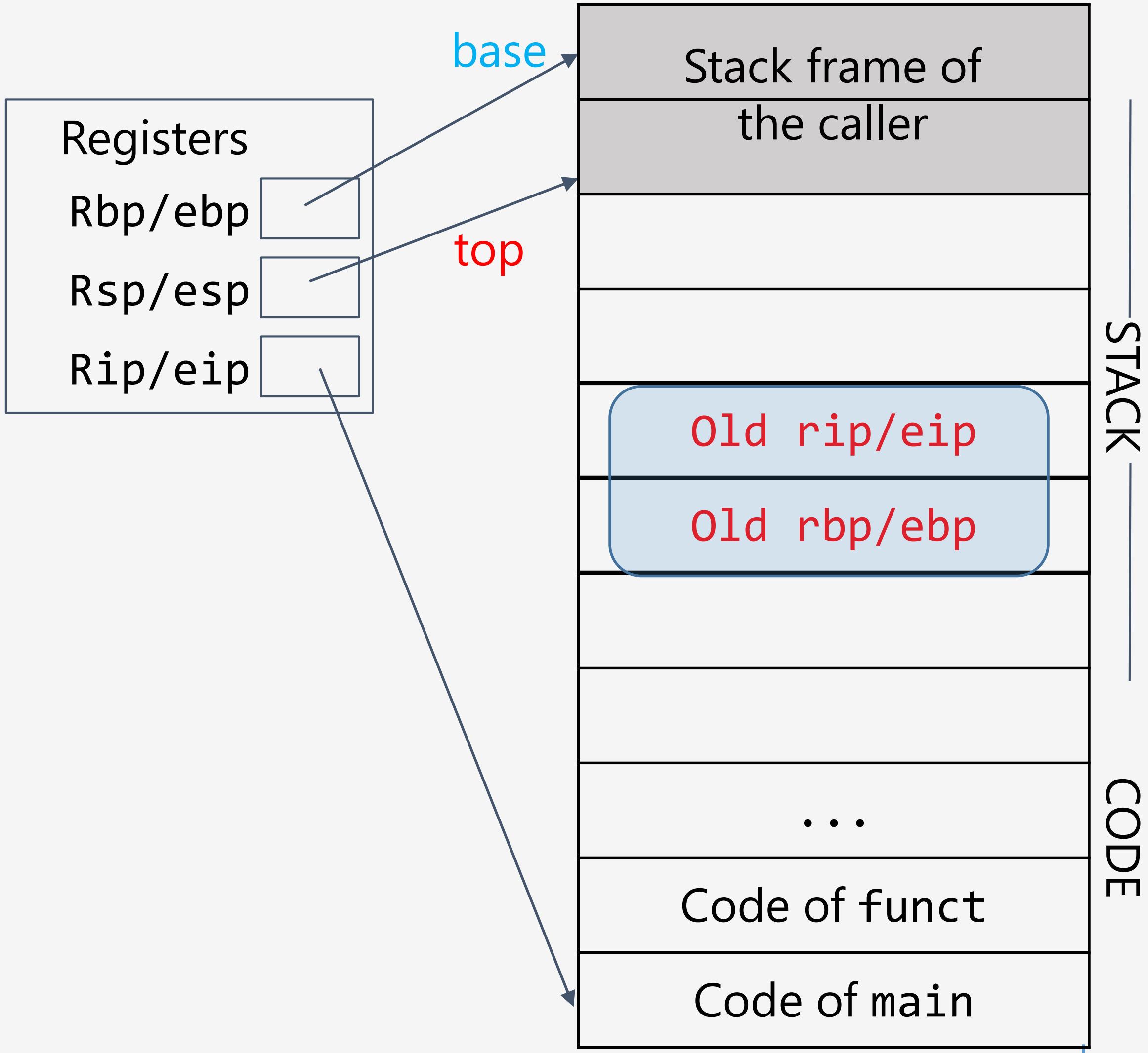
5. Execute the function

- Now we are ready to run the function
  - the stack frame is already allocated
  - The rip/eip points to the function
  - The local variables are created in the stack frame



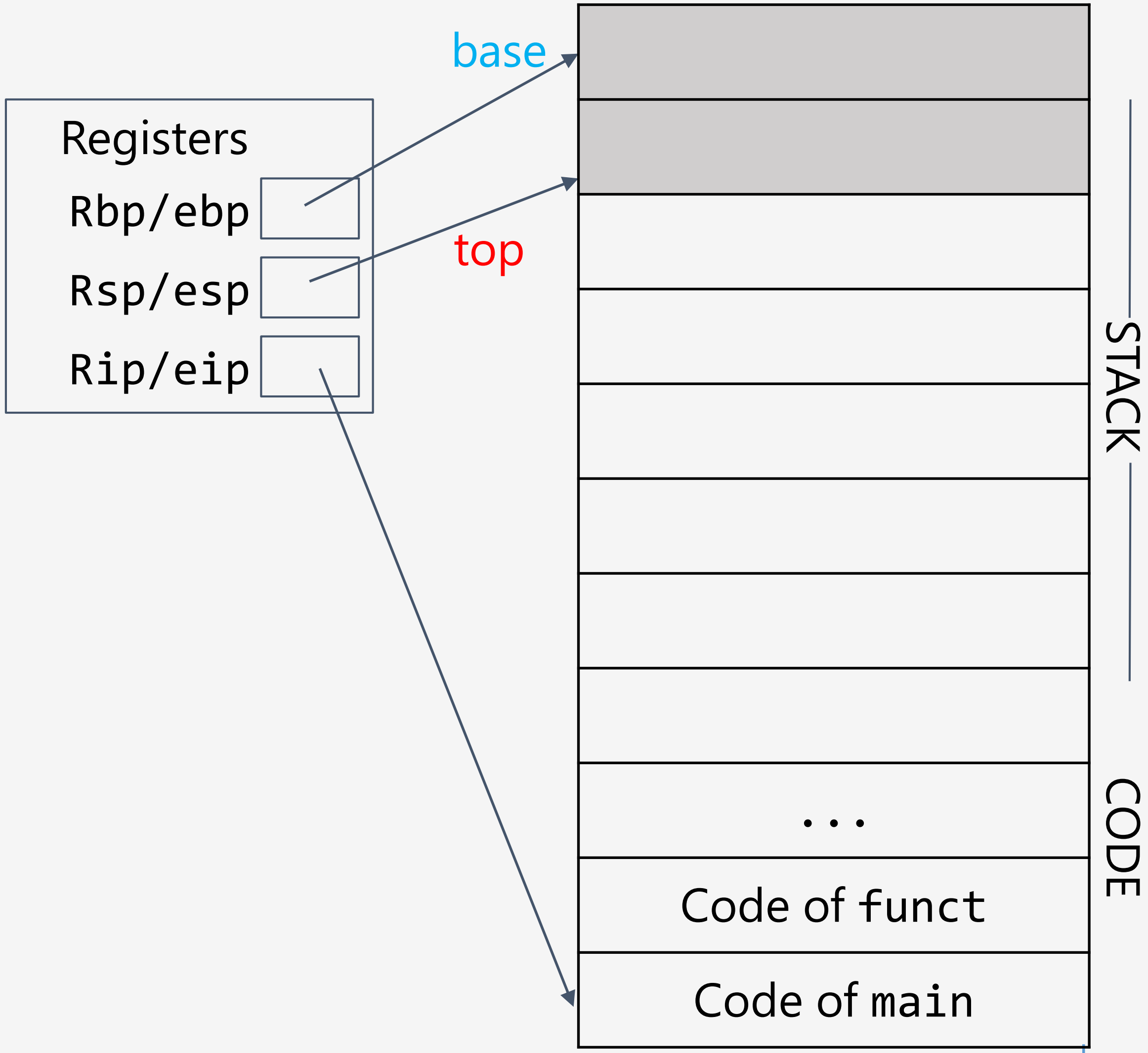
6. Restore everything after the function call

- After the function is finished, we put all three registers back where they were.
- We use the addresses stored in the stack to restore restore rip/eip and rbp/ebp to their old values, using the saved values we have on the stack



6. Restore everything

- `Rsp/esp` naturally moves back to its old place as we **pop** all the pushed values off the stack.
- Note that the values we pushed on the stack are still there (we don't overwrite them to save time), but they are below rsp/`esp` so we assume that they cannot be accessed.



Steps of a function call (simple)

1. Push arguments on the stack
2. Push old eip (rip) on the stack
3. Push old ebp (rbp) on the stack
4. Adjust the stack frame
5. Execute the function
6. Restore everything

Steps of a function call (complete)

1. Push arguments on the stack
2. Push old eip (rip) on the stack
3. Push old ebp (rbp) on the stack
4. Move ebp (rbp)
5. Move esp (rsp)
6. Execute the function
7. Move esp (rsp)
8. Restore old ebp (rbp)
9. Restore old eip (rip)
10. Remove arguments from stack

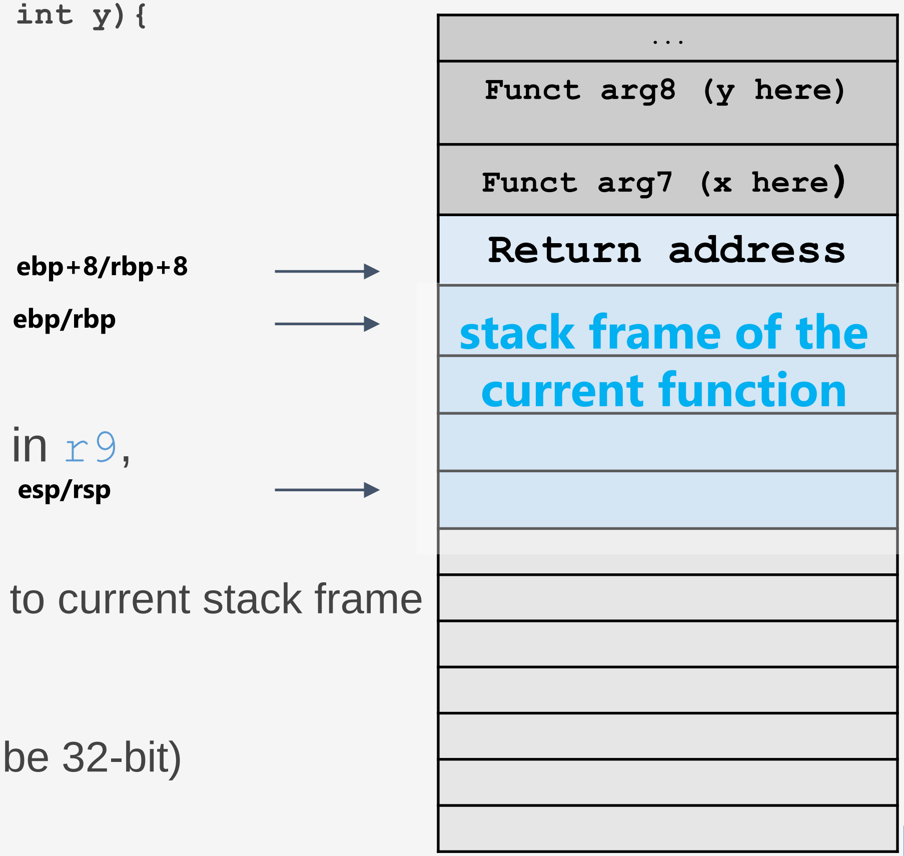
A function call stack layout example

- Consider the following C program, how the arguments and local variables are put to the stack according to the function call convention of AMD64?

```
#include <unistd.h>
#include <stdlib.h>

void funct(int a1, int a2, int a3, int a4, int a5, int a6, int x, int y){
        int local_var1=0x9;
        int local_var2=0xA;
        int local_var3=0xB;
        int local_var4;
}

void main(){
        funct(1,2,3,4,5,6,7,8);
}
```



- a1 stored in `rdi`, a2 in `rsi`, a3 in `rdx`, a4 in `rcx`, a5 in `r8`, a6 in `r9`, x in stack, y in stack

- In general, an earlier function argument is put at lower address, closer to current stack frame
  - x will be at address `rbp+16` (assuming 64-bit return address)
  - y will be at address `rbp+20` (assuming x to be 32-bit and y to be 32-bit)

-

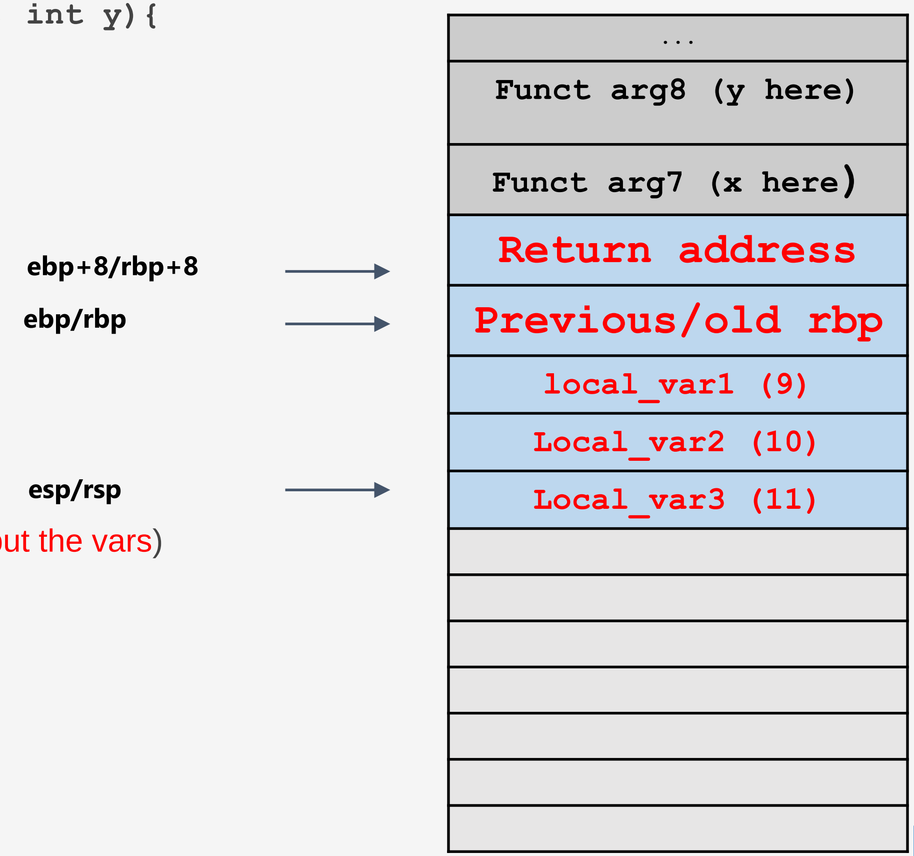
A function call stack layout example

- Consider the following C program, how the arguments and local variables are put to the stack according to the function call convention of AMD64?

```
#include <unistd.h>
#include <stdlib.h>

void funct(int a1, int a2, int a3, int a4, int a5, int a6, int x, int y){
        int local_var1=9;
        int local_var2=10;
        int local_var3=11;
        int local_var4;
}

void main(){
        funct(1,2,3,4,5,6,7,8);
}
```



- Local variables are put in the same order as their appearance

(different C compiler will put the local variables differently, the C standard does not mention how to put the vars)

  - local_var1 could be at rbp-4
  - local_var2 could be at rbp -8
  - local_var3 could be at rbp -12
  - Unused local_var4 not allocated any space in the stack