x86-32 and x86-64 Assembly (Part 2)

(I know Kung-Fu !)

Emmanuel Fleury
<emmanuel.fleury@u-bordeaux.fr>

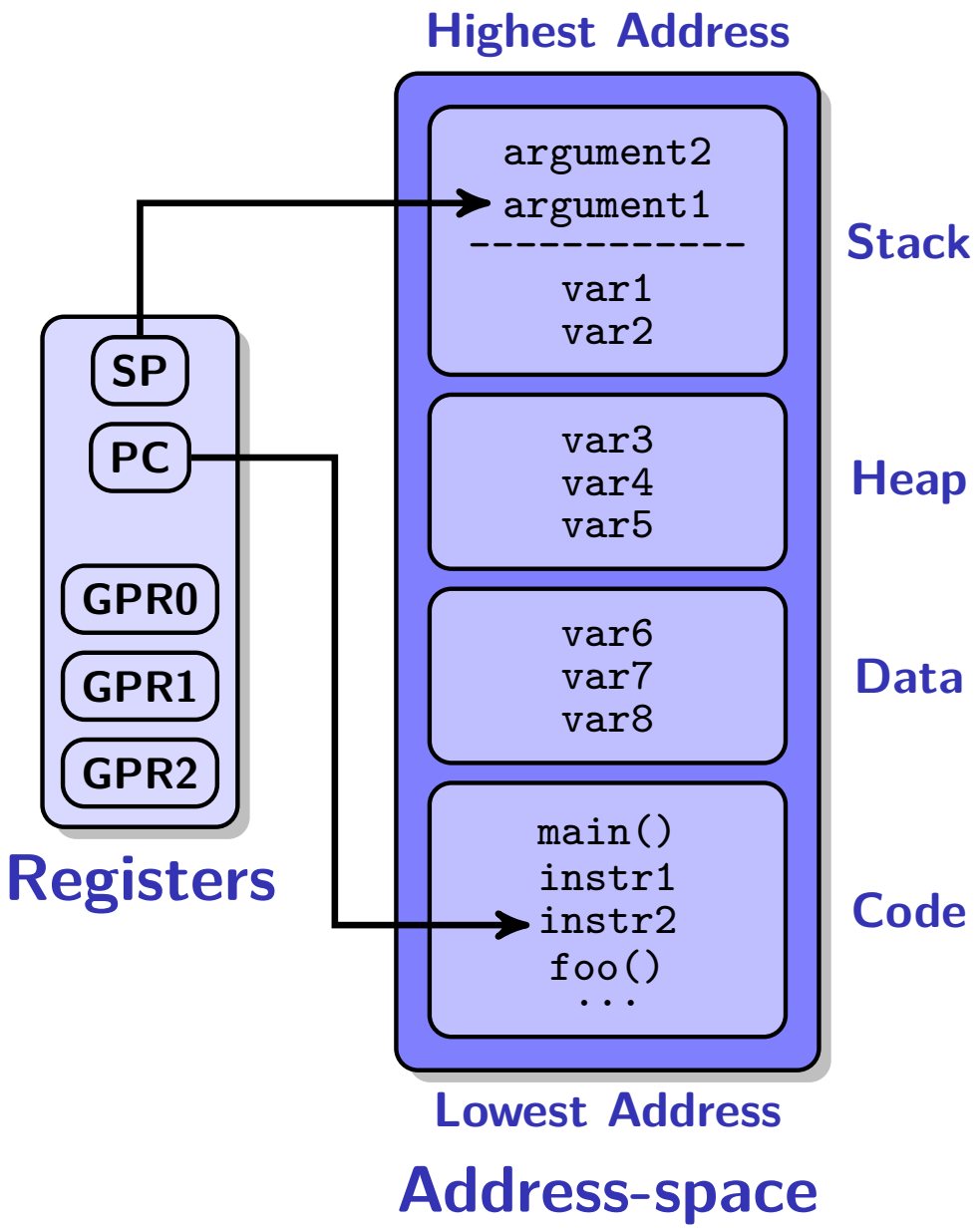
LaBRI, Université de Bordeaux, France

October 8, 2019

Overview

Overview

Program Overview



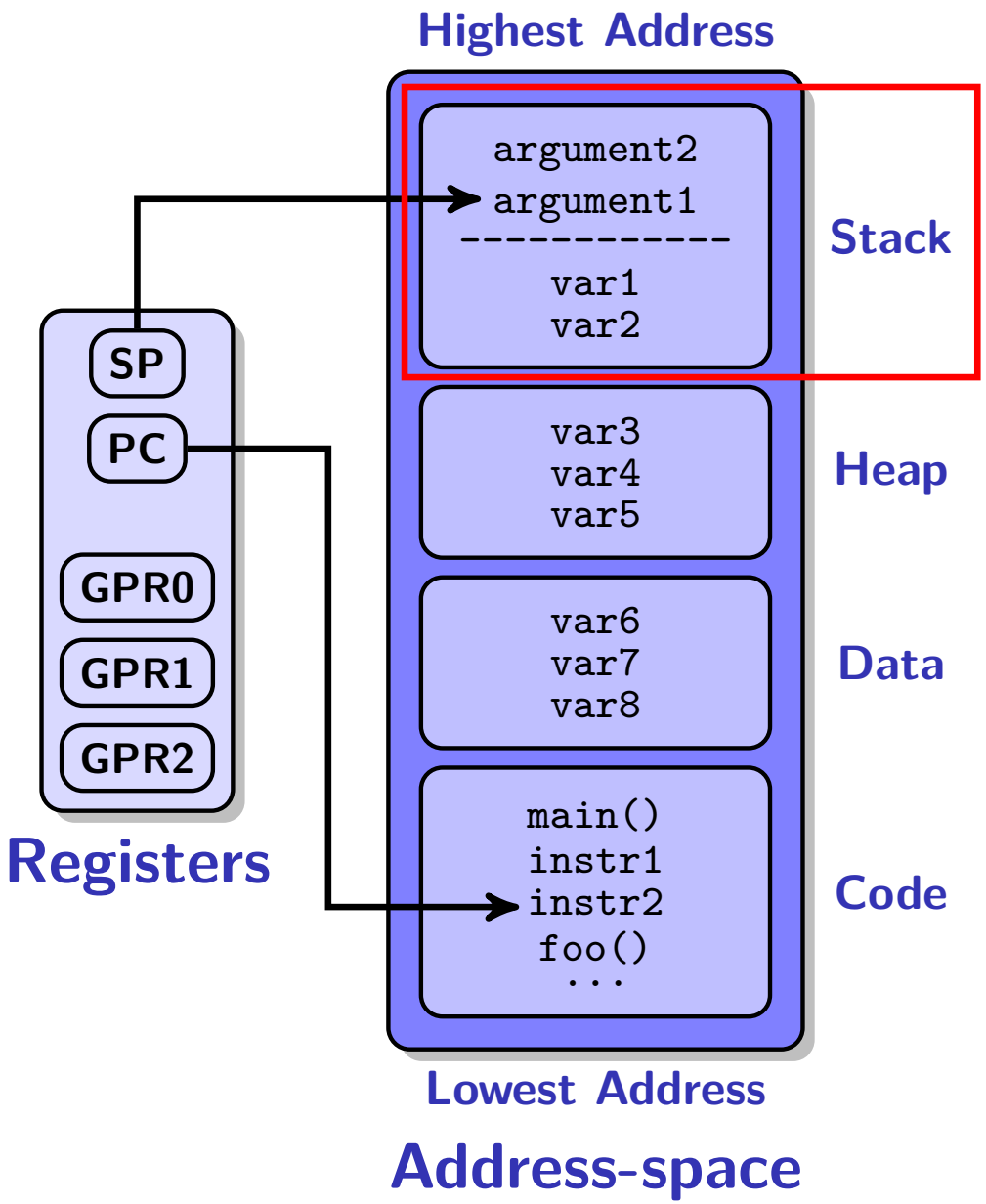
Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

Program Overview



Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

université de BORDEAUX

Managing Stack Data

Mnemonic	Operand	Operation
`push`	src	Push the content of 'src' on the stack
`pop`	dst	Pop the content from the stack to 'dst'

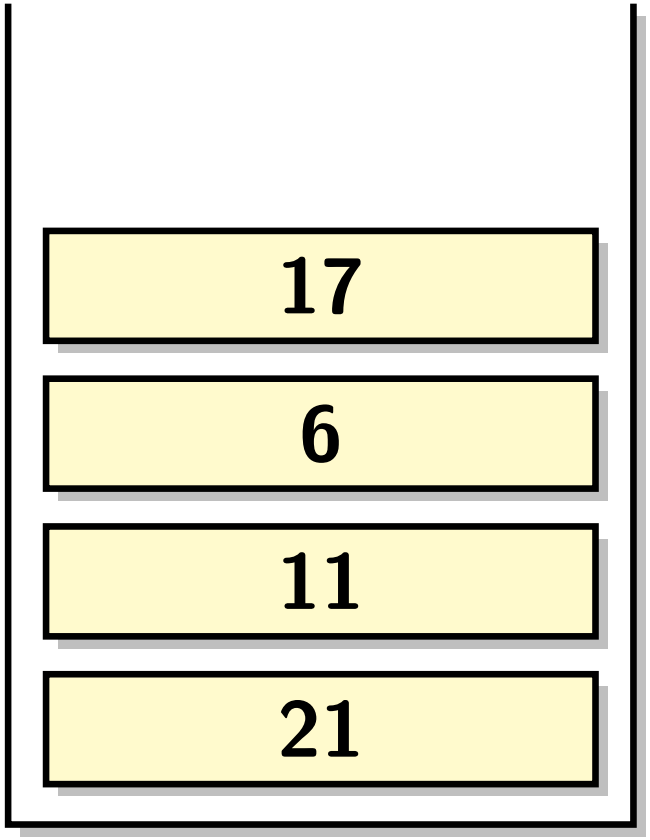
Managing Stack Frames

Mnemonic	Operation
`enter`	Create a new stack-frame
`leave`	Restore the previous stack-frame

Managing Call Stack

Mnemonic	Operand	Operation
`call`	addr	Save `eip` and jump to a function at 'addr'
`ret`	–	Restore saved `eip` and return from a function

Stack (Basic Principle)

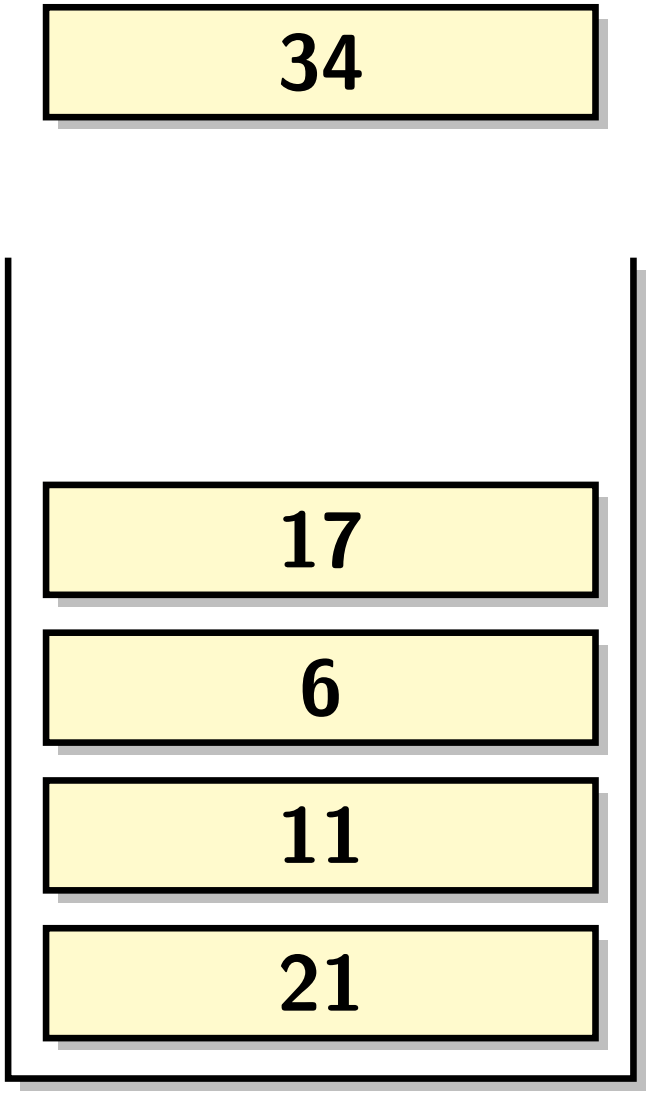


Last In First Out (LIFO)

Only two operations:

- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (Basic Principle)

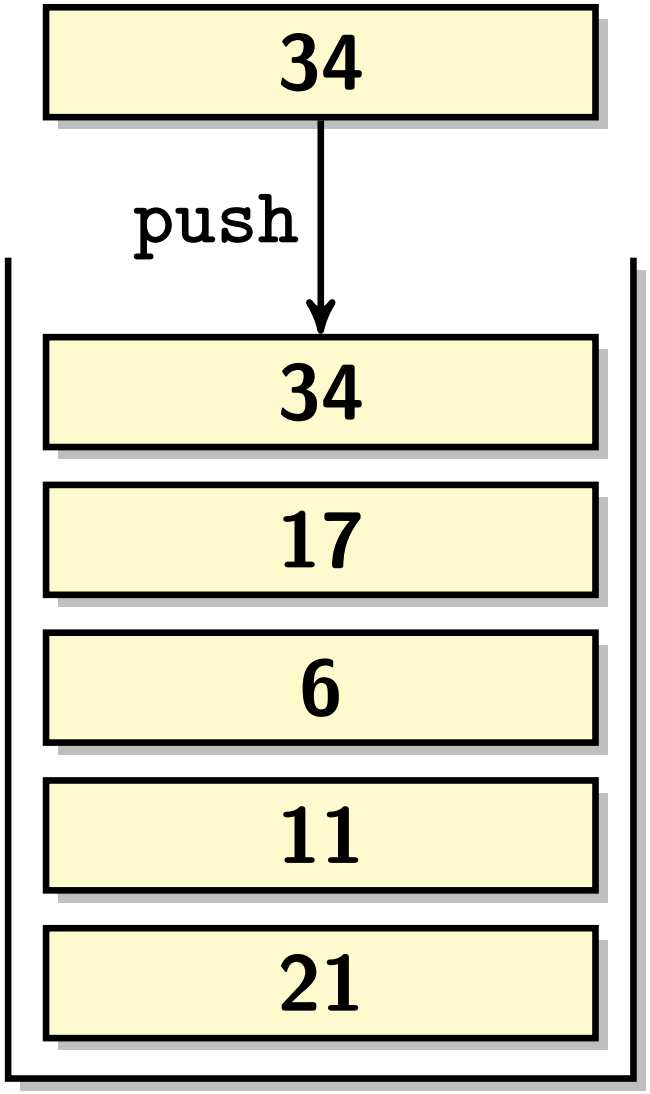


Last In First Out (LIFO)

Only two operations:

- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (Basic Principle)

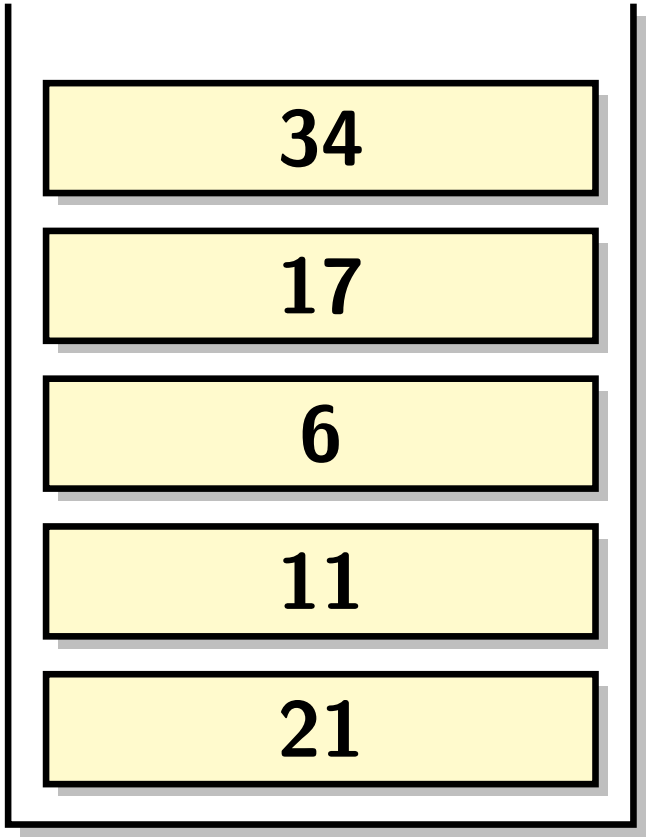


Last In First Out (LIFO)

Only two operations:

- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (Basic Principle)

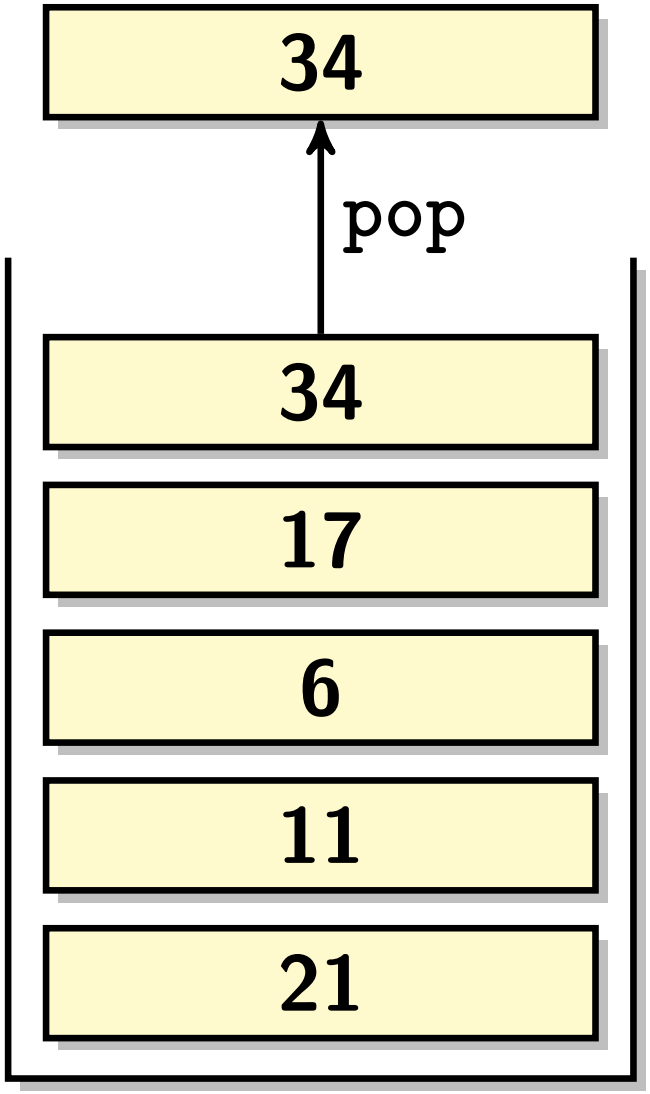


Last In First Out (LIFO)

Only two operations:

- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (Basic Principle)

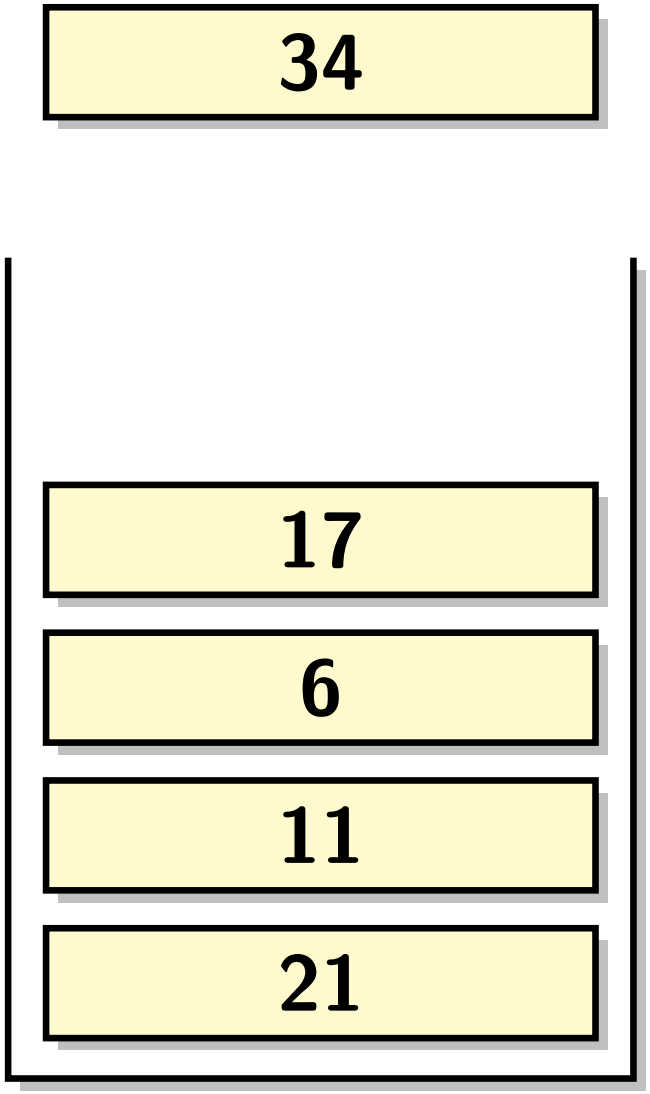


Last In First Out (LIFO)

Only two operations:

- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (Basic Principle)

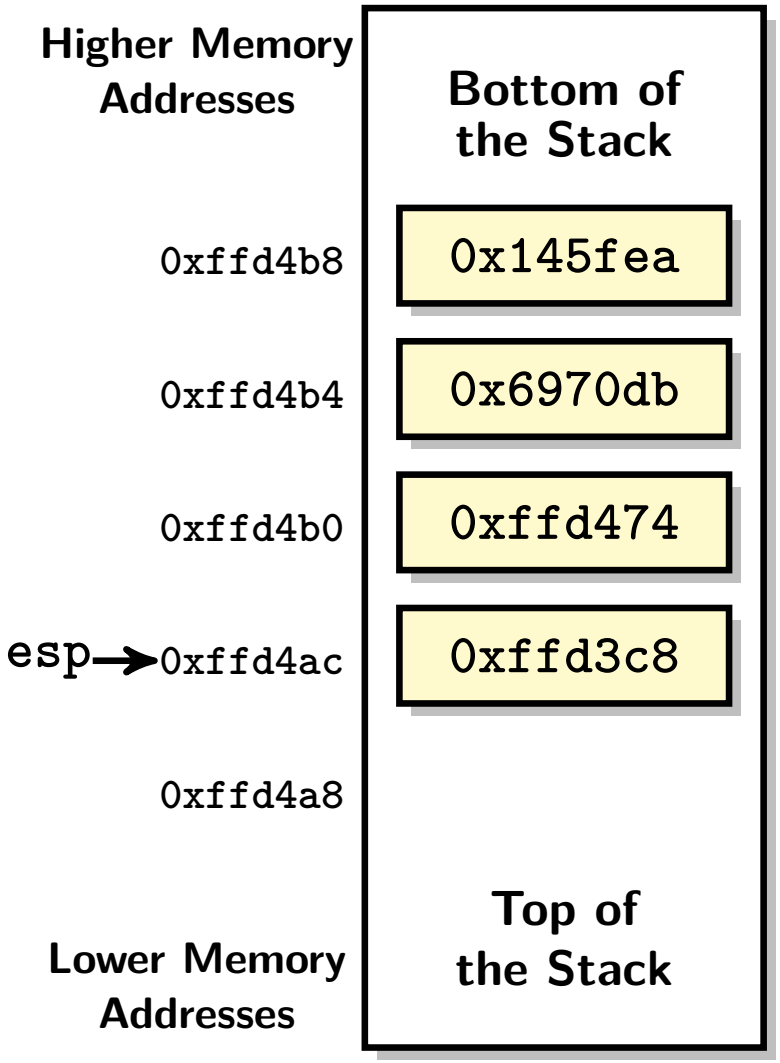


Last In First Out (LIFO)

Only two operations:

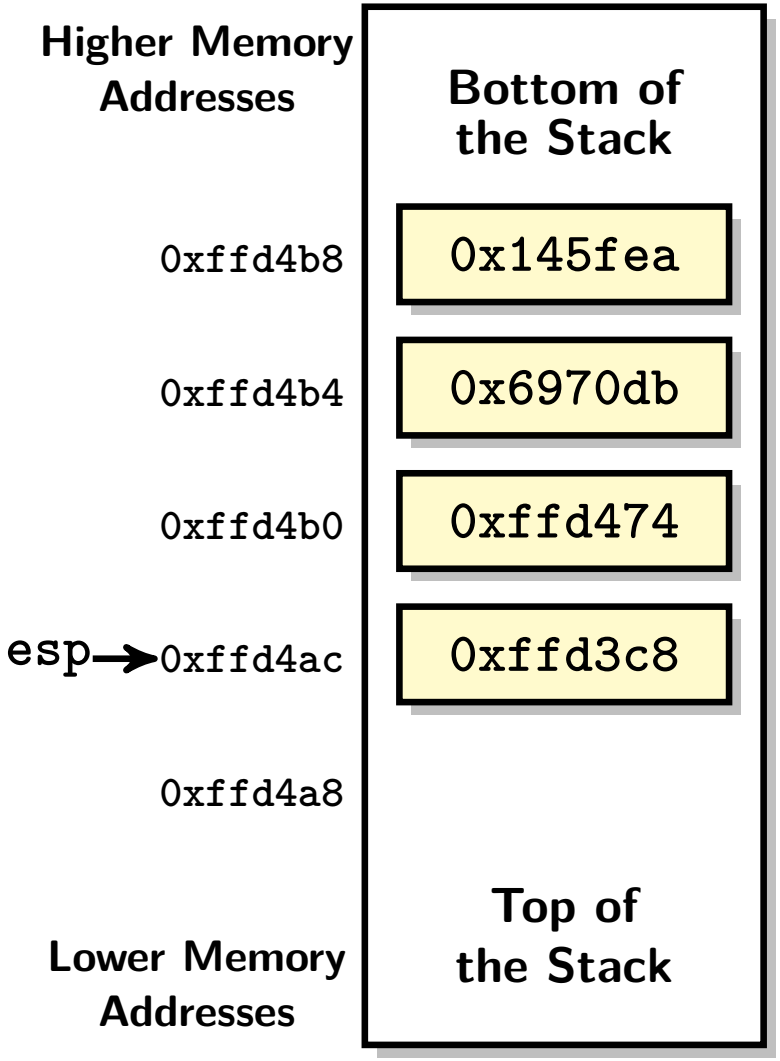
- **push**
  Push an item on the stack.
- **pop**
  Pop an item from the stack.

Stack (In Memory)

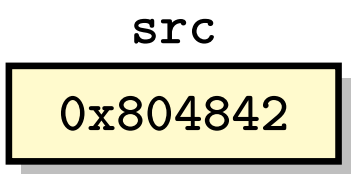


- Memory area is managed as a stack.
- It grows toward lower addresses.
- Register `esp` (stack-pointer) contains:
  - Address of the stack's top element.
  - Lowest address of the memory area.

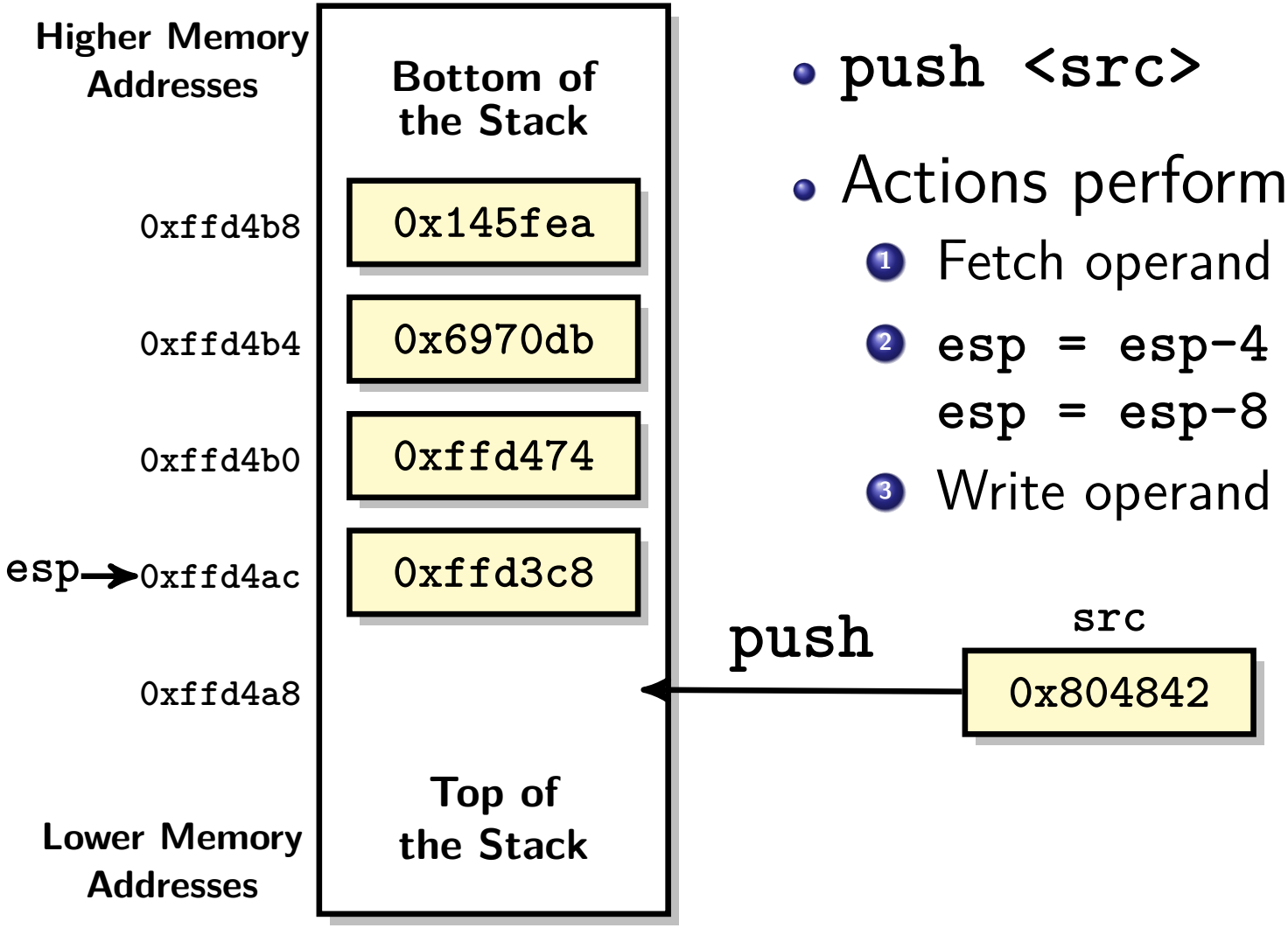
push



- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

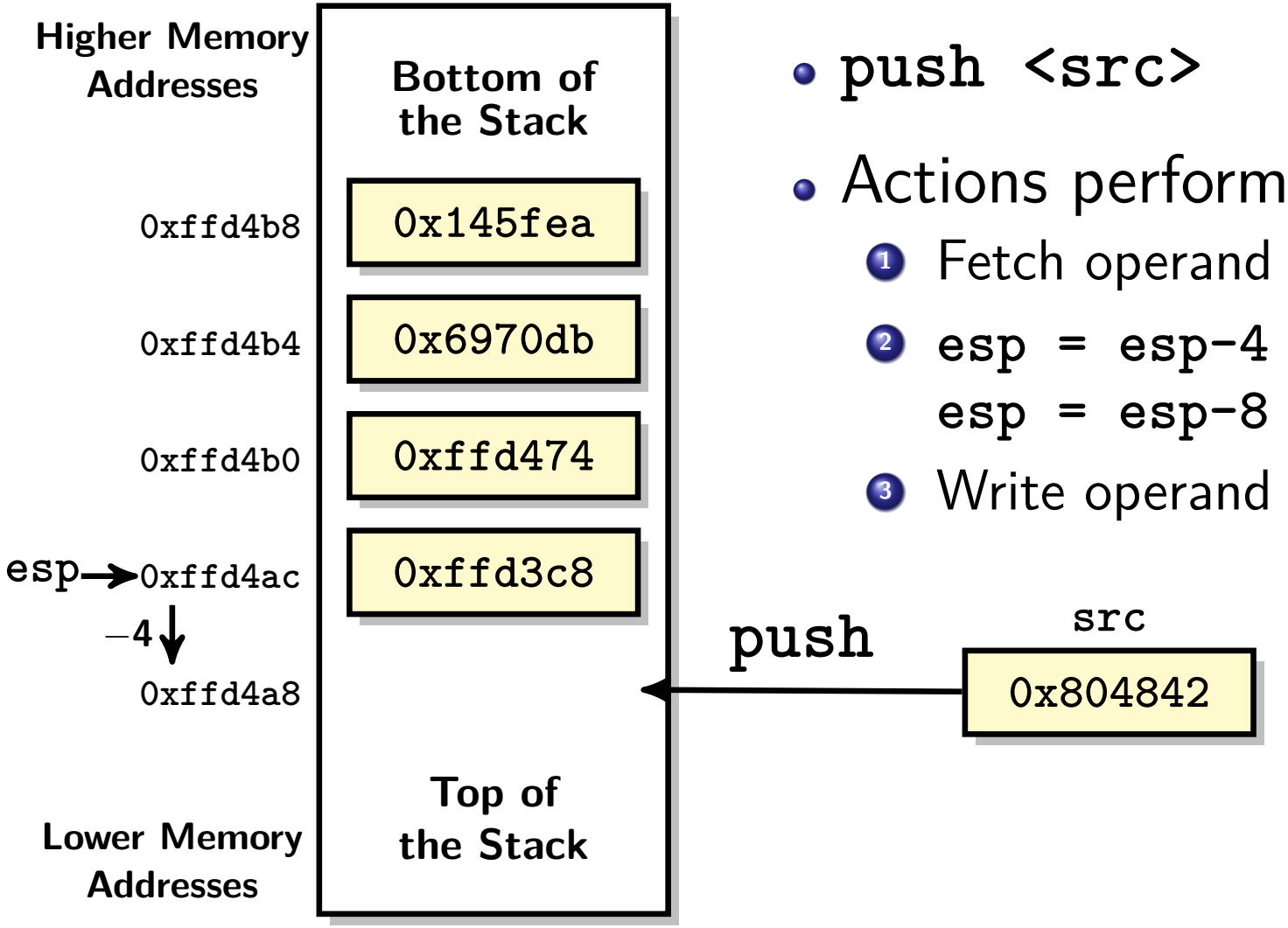


push



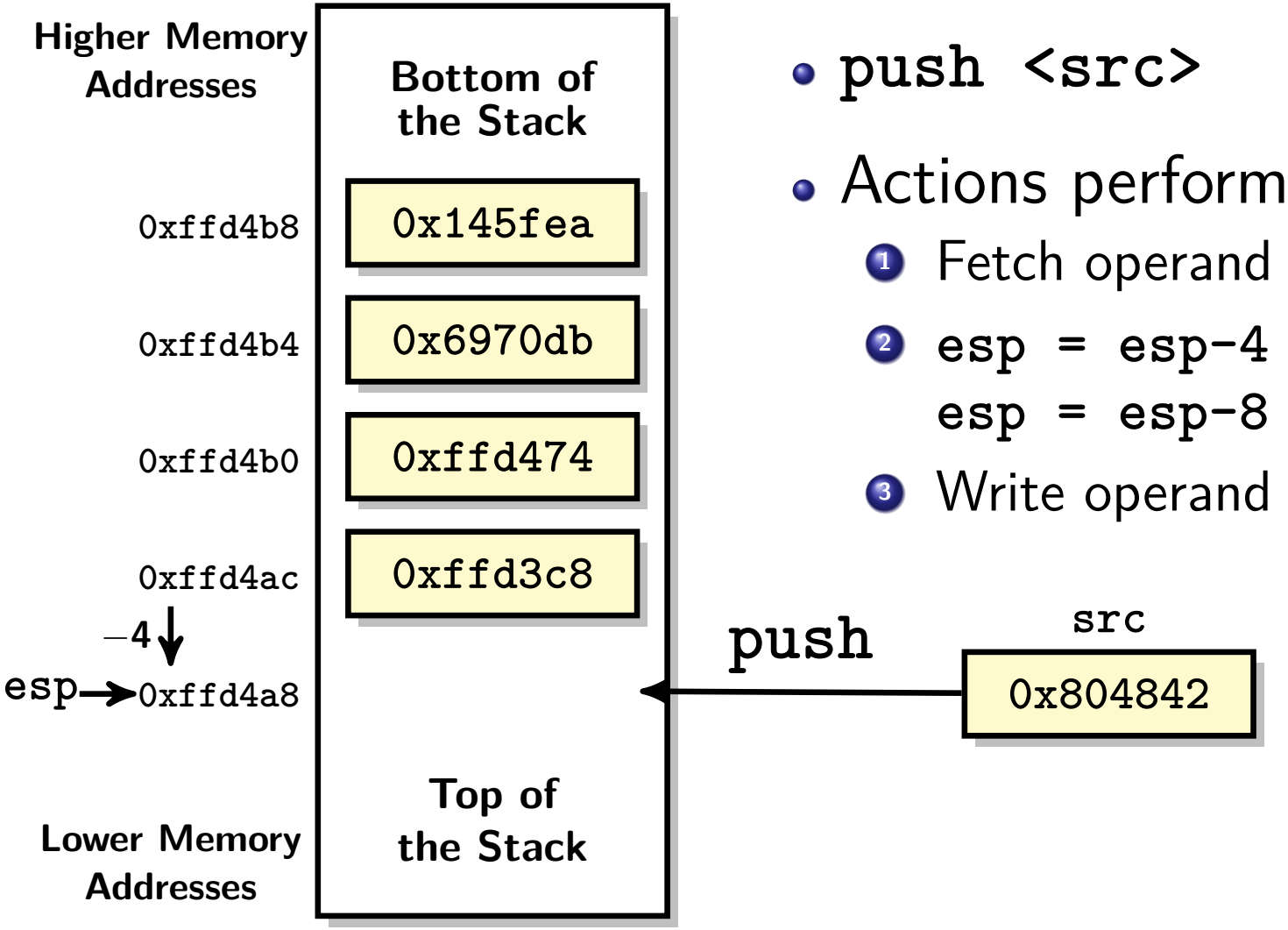
- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

push



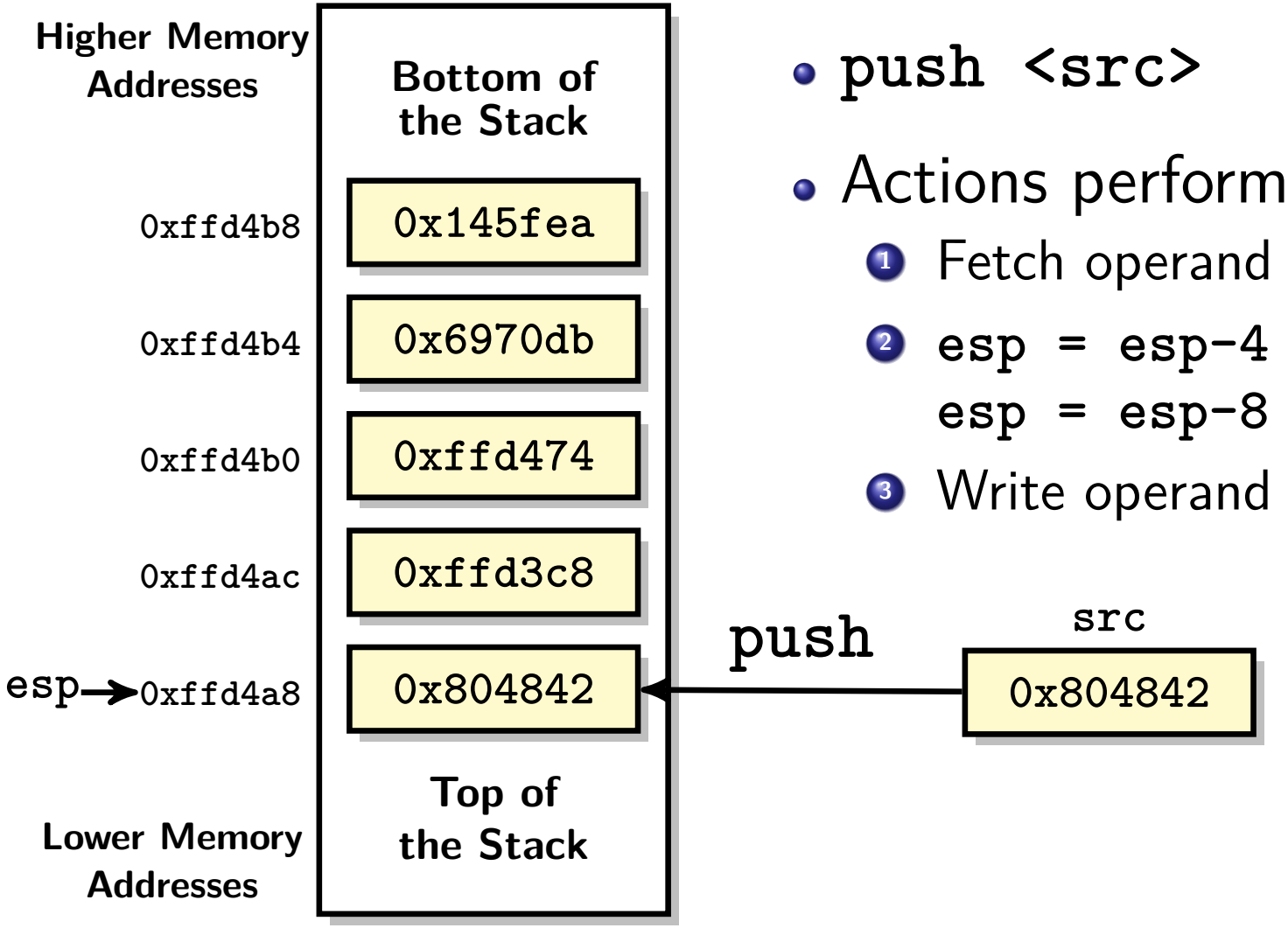
- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

push



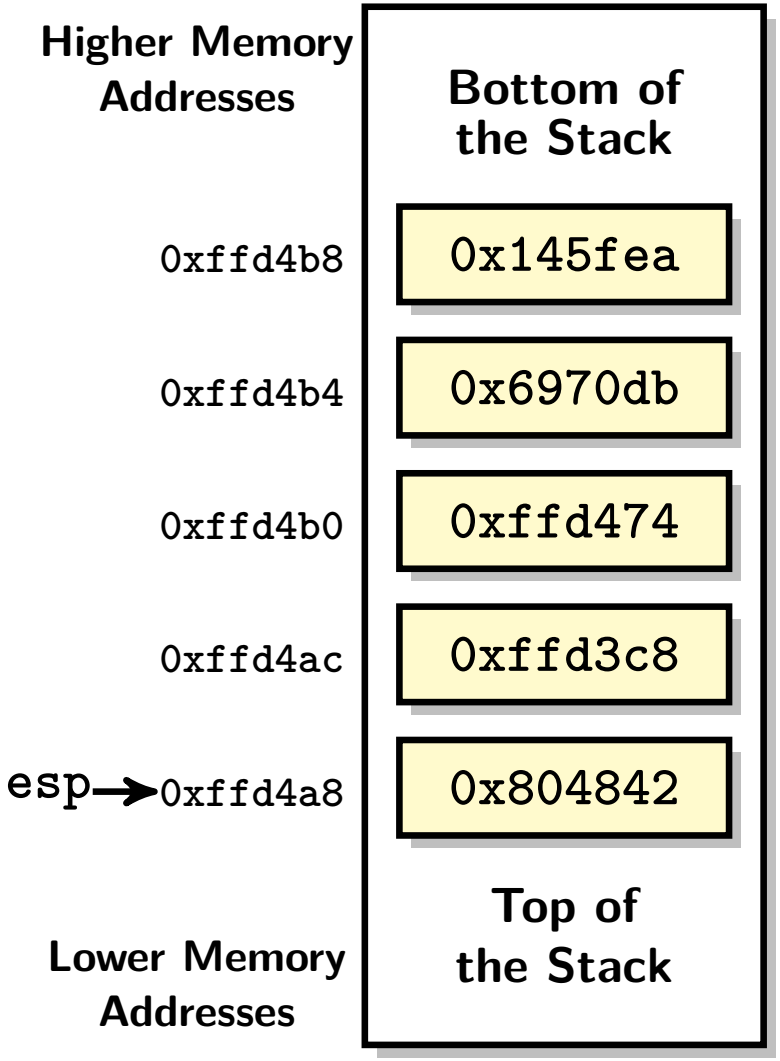
- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

push



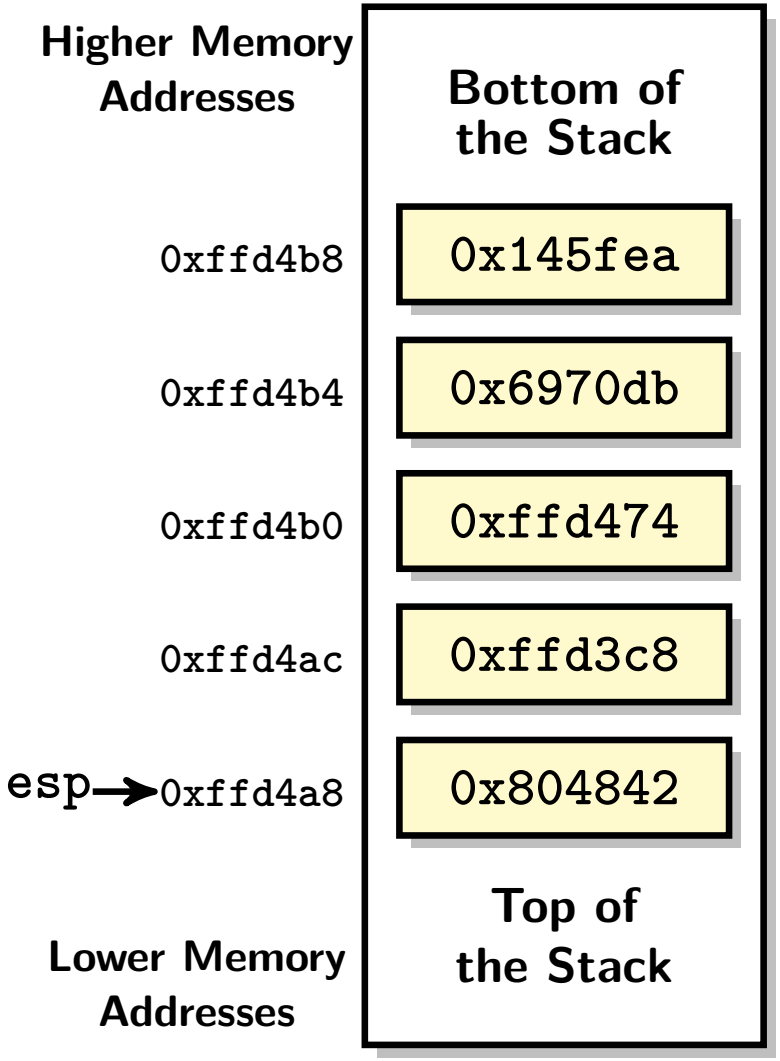
- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

push



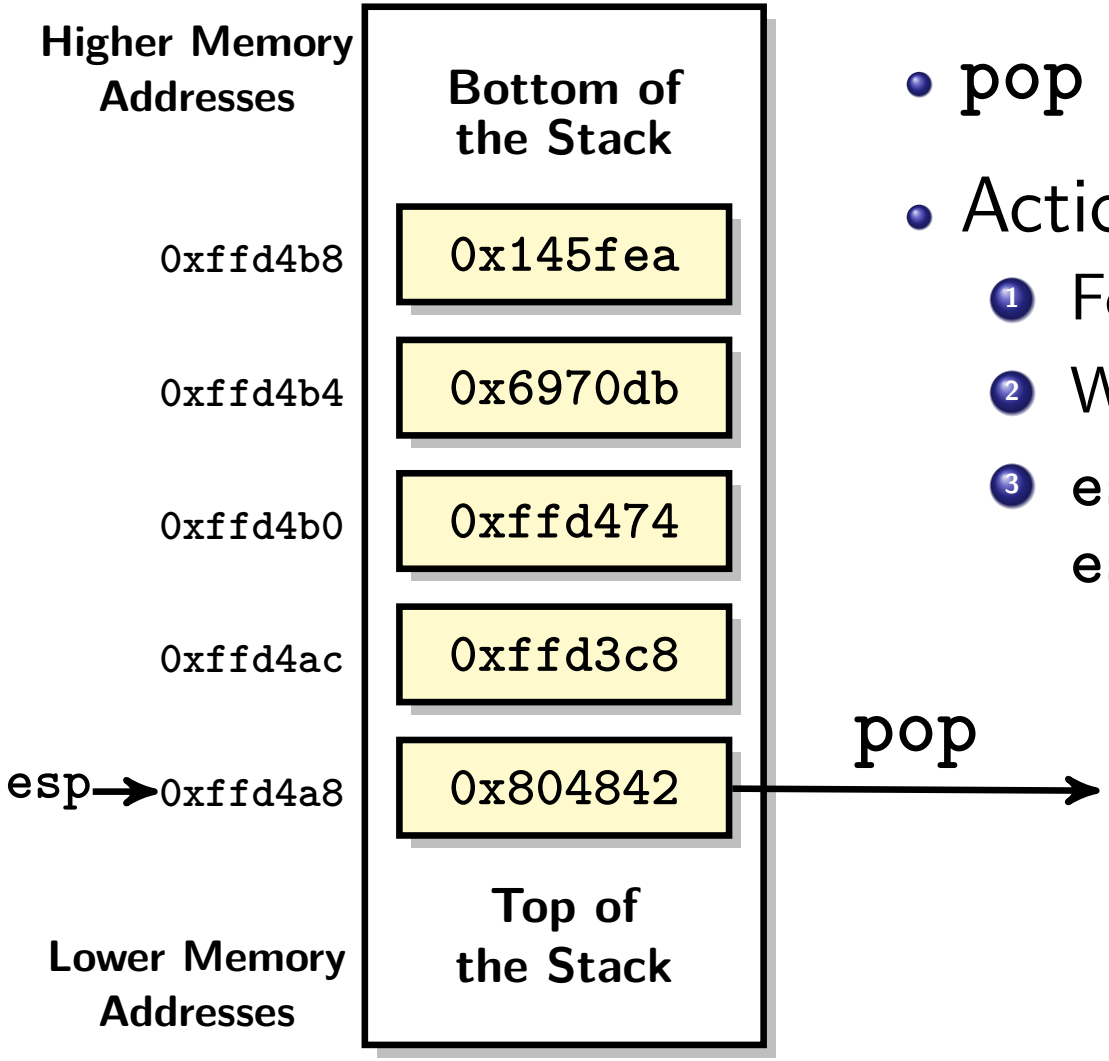
- `push <src>`
- Actions performed:
  1. Fetch operand from `src`;
  2. `esp = esp-4` (32 bits)
     `esp = esp-8` (64 bits);
  3. Write operand to `(esp)`.

pop



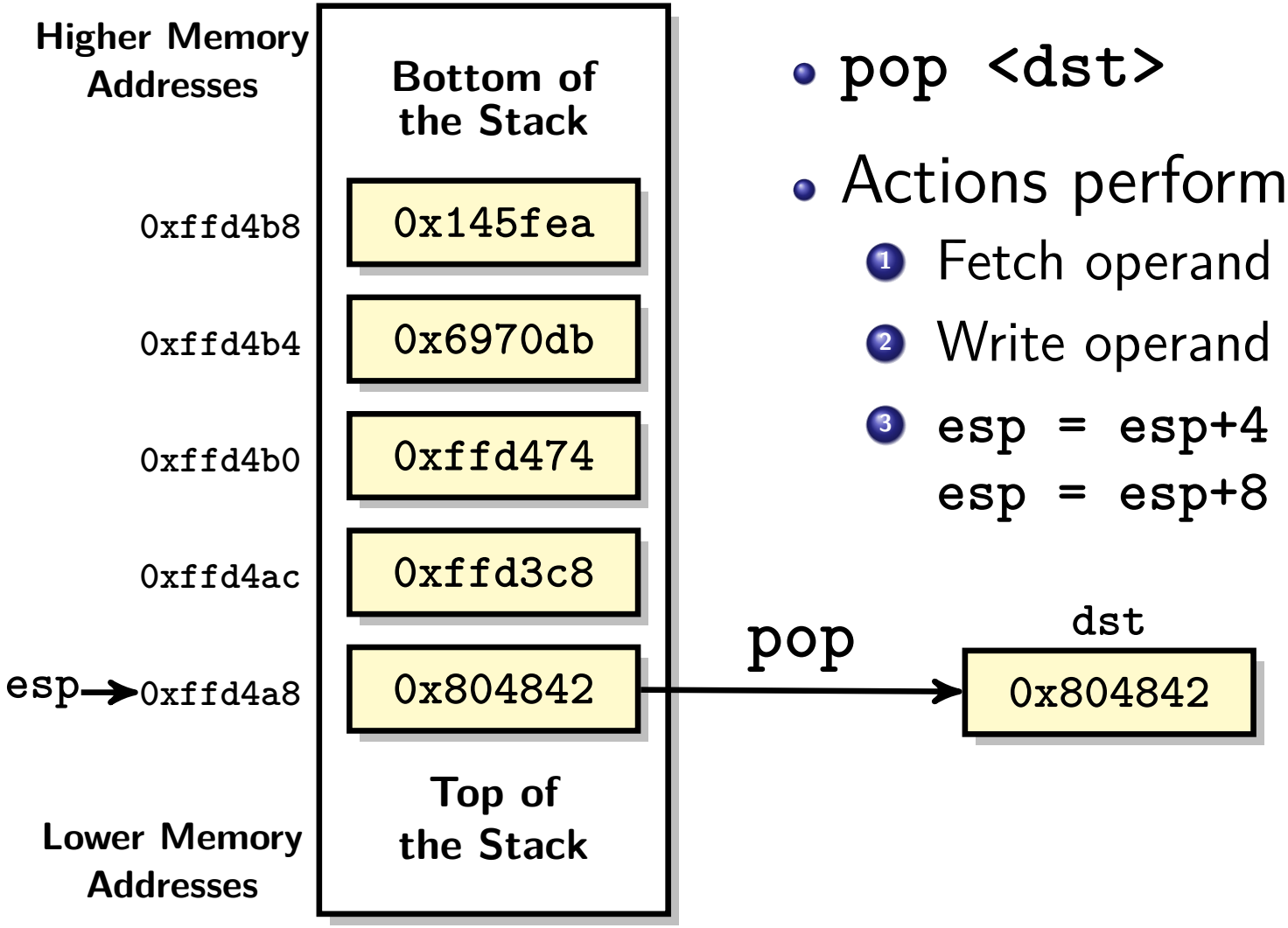
- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

pop



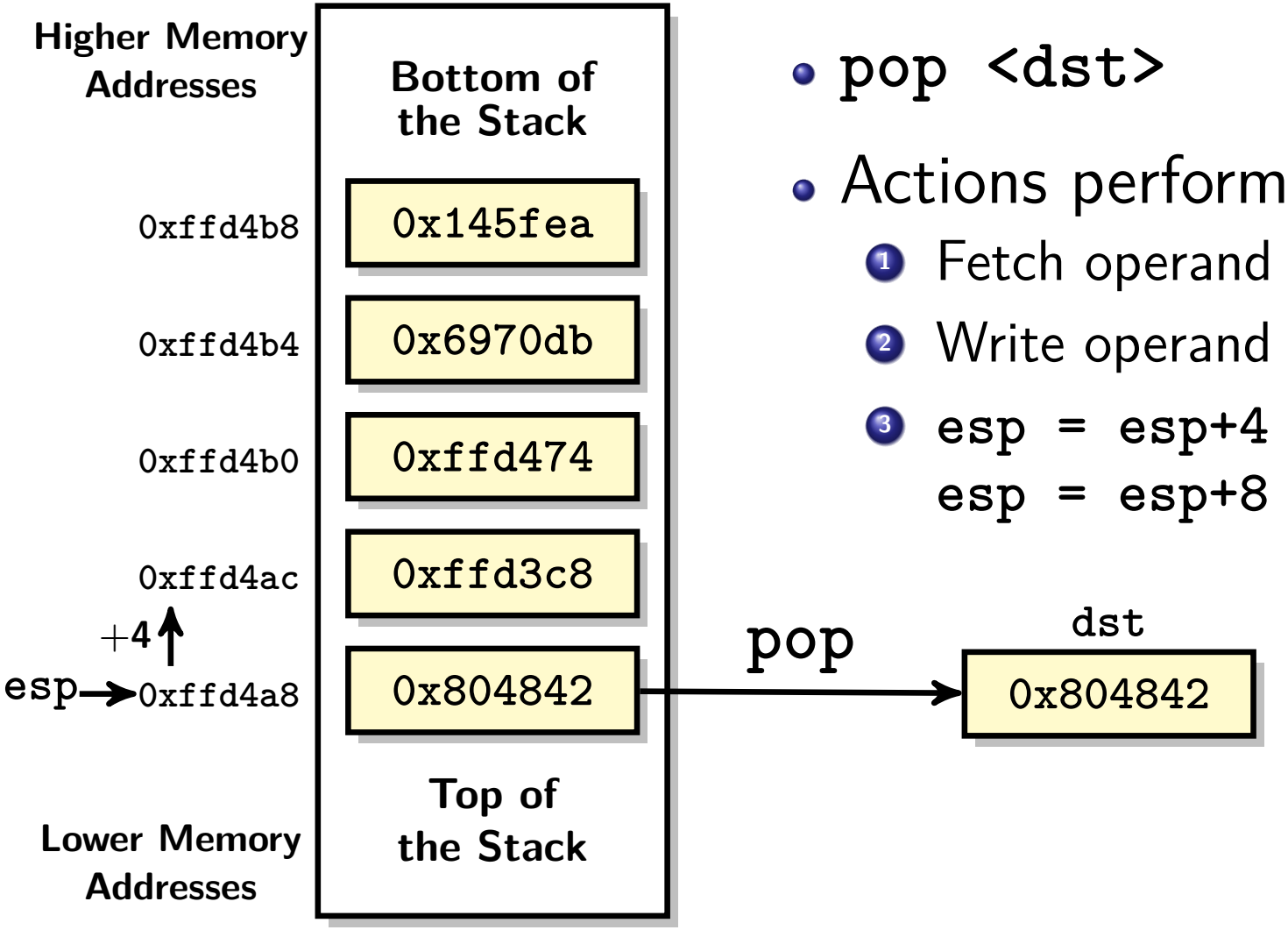
- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

pop



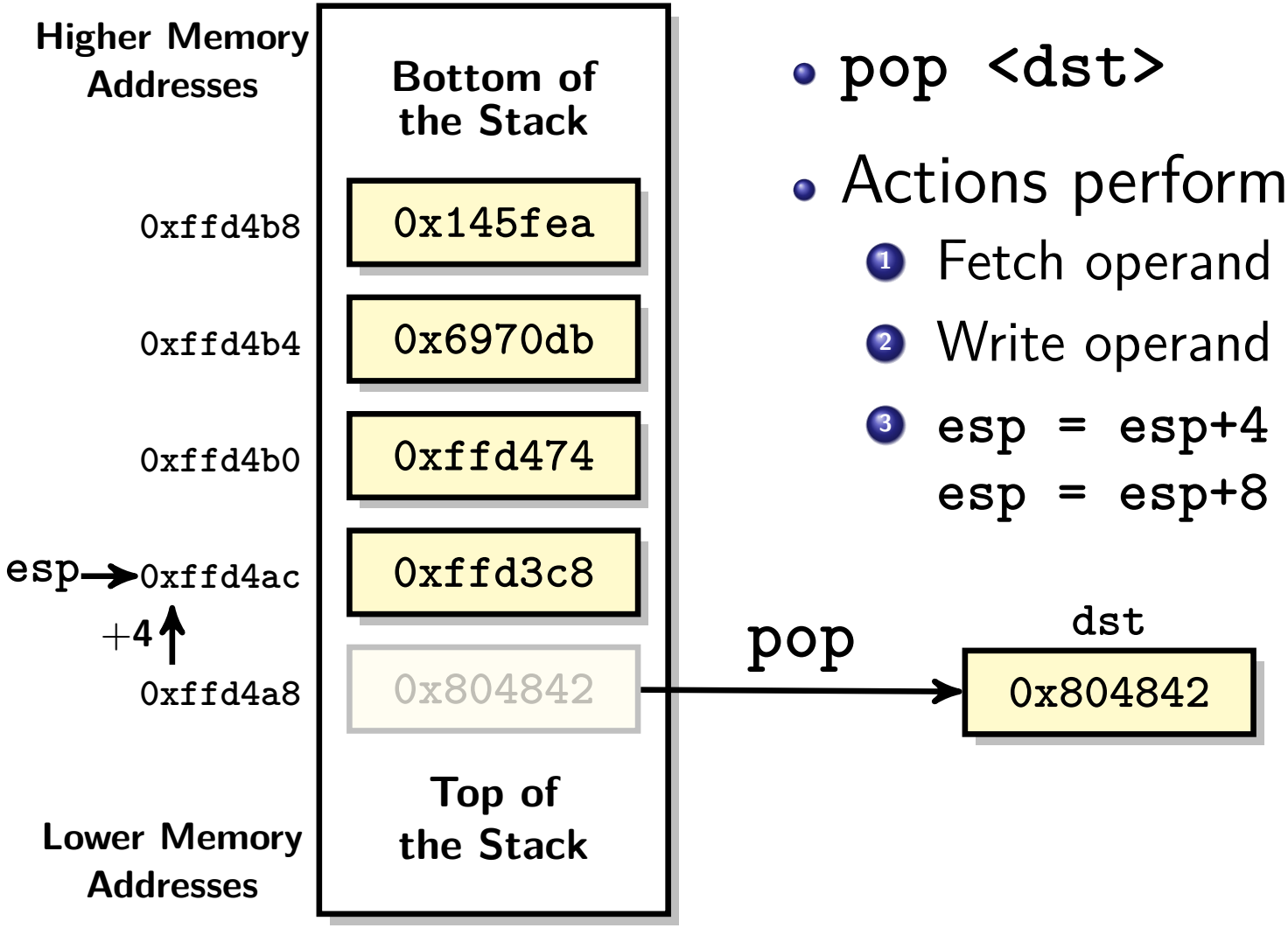
- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

pop



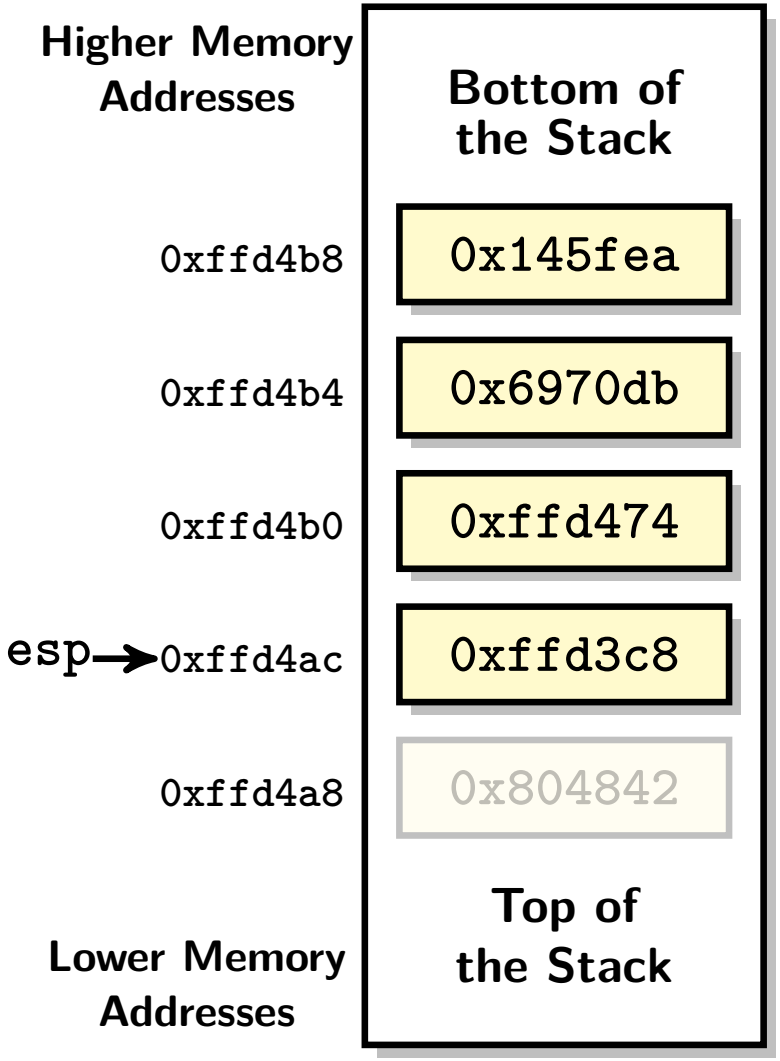
- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

pop

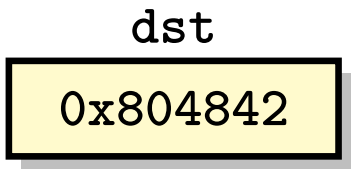


- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

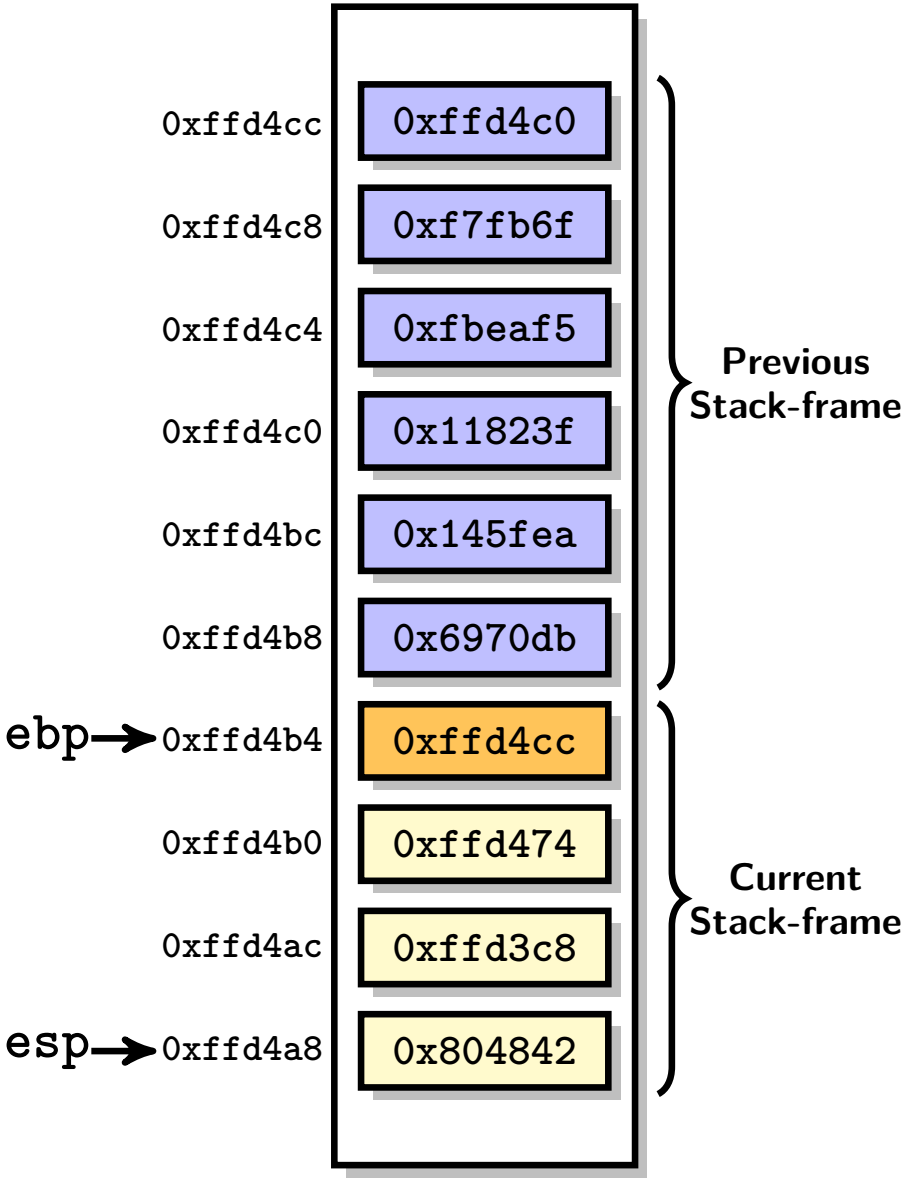
pop



- `pop <dst>`
- Actions performed:
  1. Fetch operand from `esp`;
  2. Write operand to `dst`.
  3. `esp = esp+4` (32 bits)
     `esp = esp+8` (64 bits);

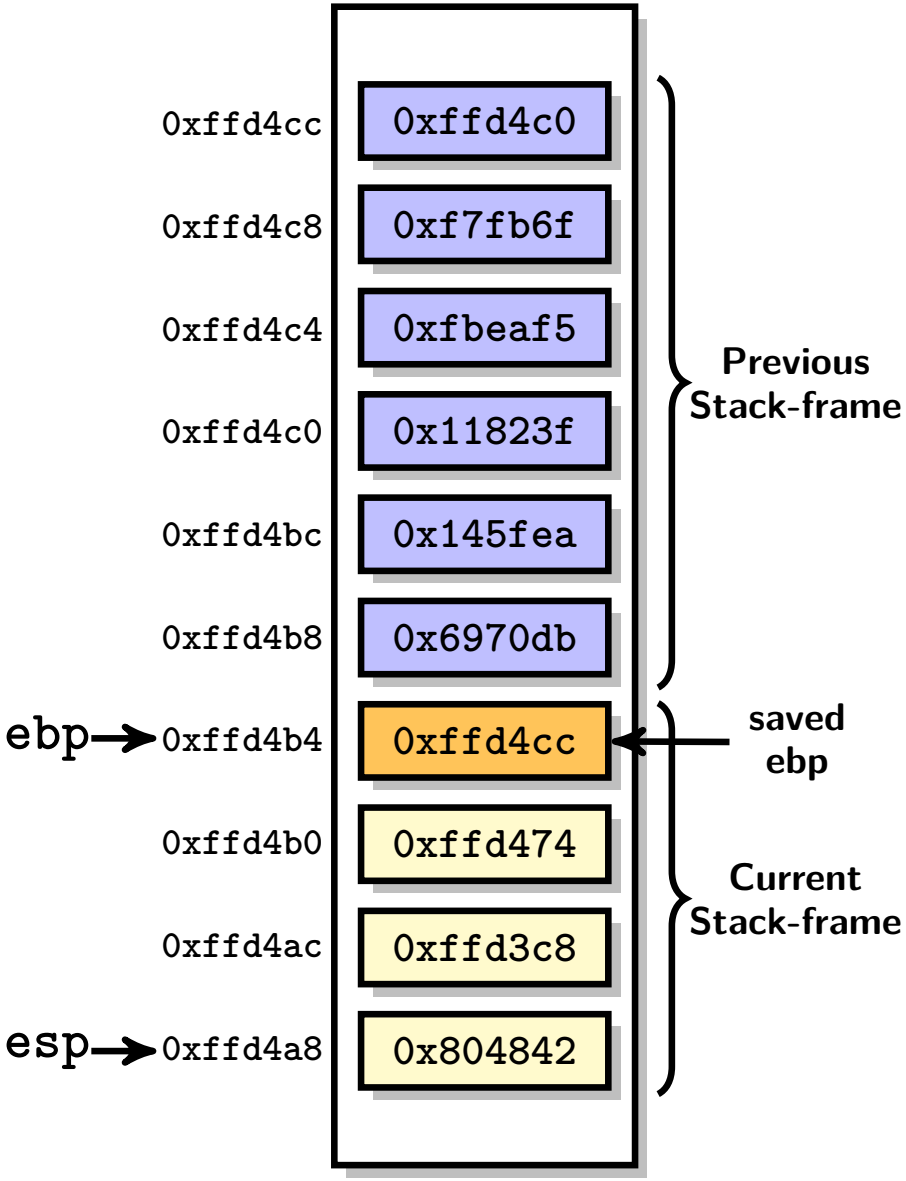


Stack Frame



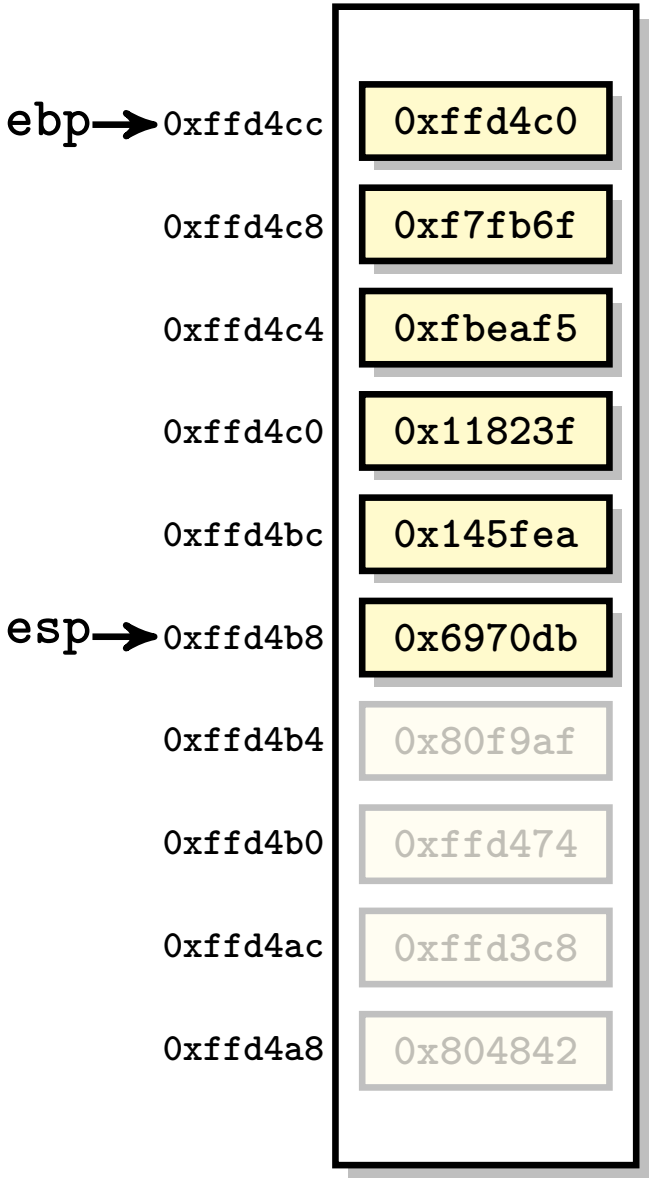
- A stack-frame is represented by the couple: **(`esp`, `ebp`)**
- Register `esp` (stack-pointer) contains:
  - Address of the stack's top element.
  - Lowest address of the stack-frame.
- Register `ebp` (base-pointer) contains:
  - Address of the stack's bottom element.
  - Highest address of the stack-frame.
- Stack's bottom element is always the **saved** `ebp` from the previous stack-frame.
- Created on '`enter`' and discarded on '`leave`'.

Stack Frame



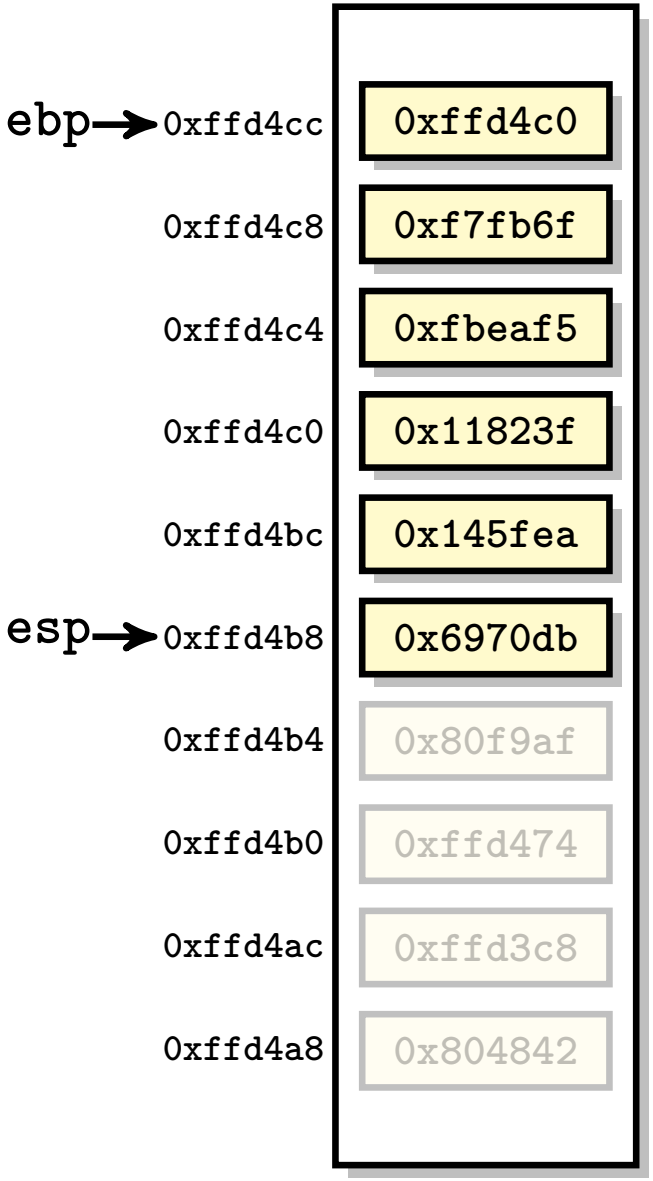
- A stack-frame is represented by the couple: **(`esp`, `ebp`)**
- Register `esp` (stack-pointer) contains:
  - Address of the stack's top element.
  - Lowest address of the stack-frame.
- Register `ebp` (base-pointer) contains:
  - Address of the stack's bottom element.
  - Highest address of the stack-frame.
- Stack's bottom element is always the **saved** `ebp` from the previous stack-frame.
- Created on '`enter`' and discarded on '`leave`'.

enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

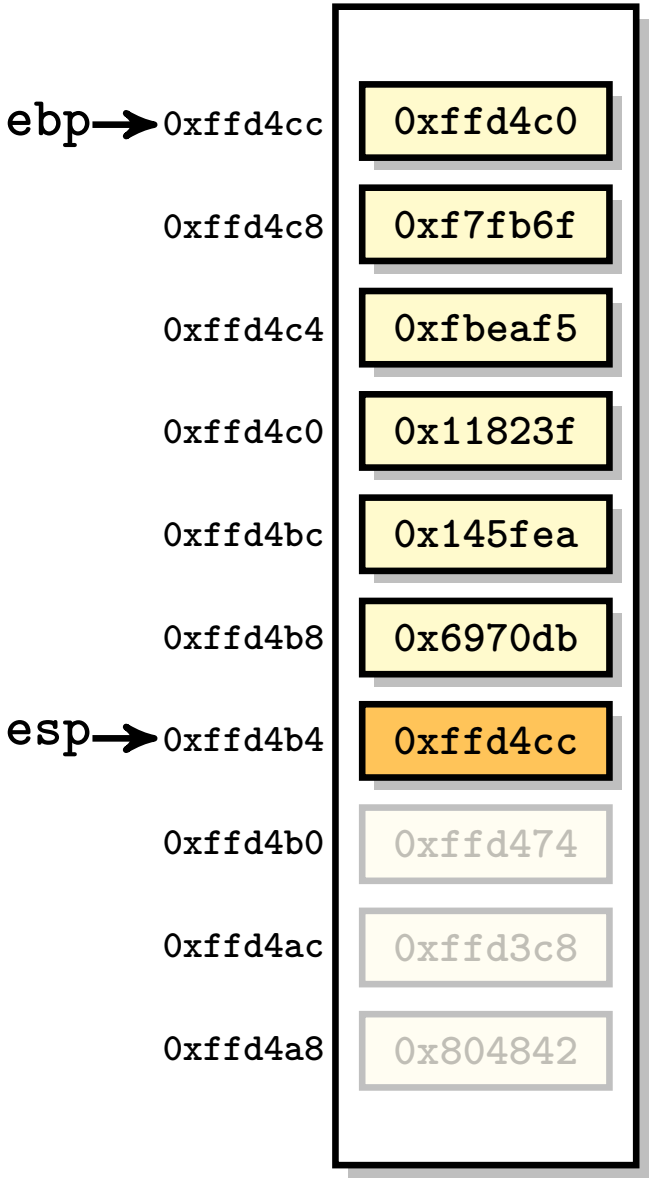
enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

Saving **ebp**.

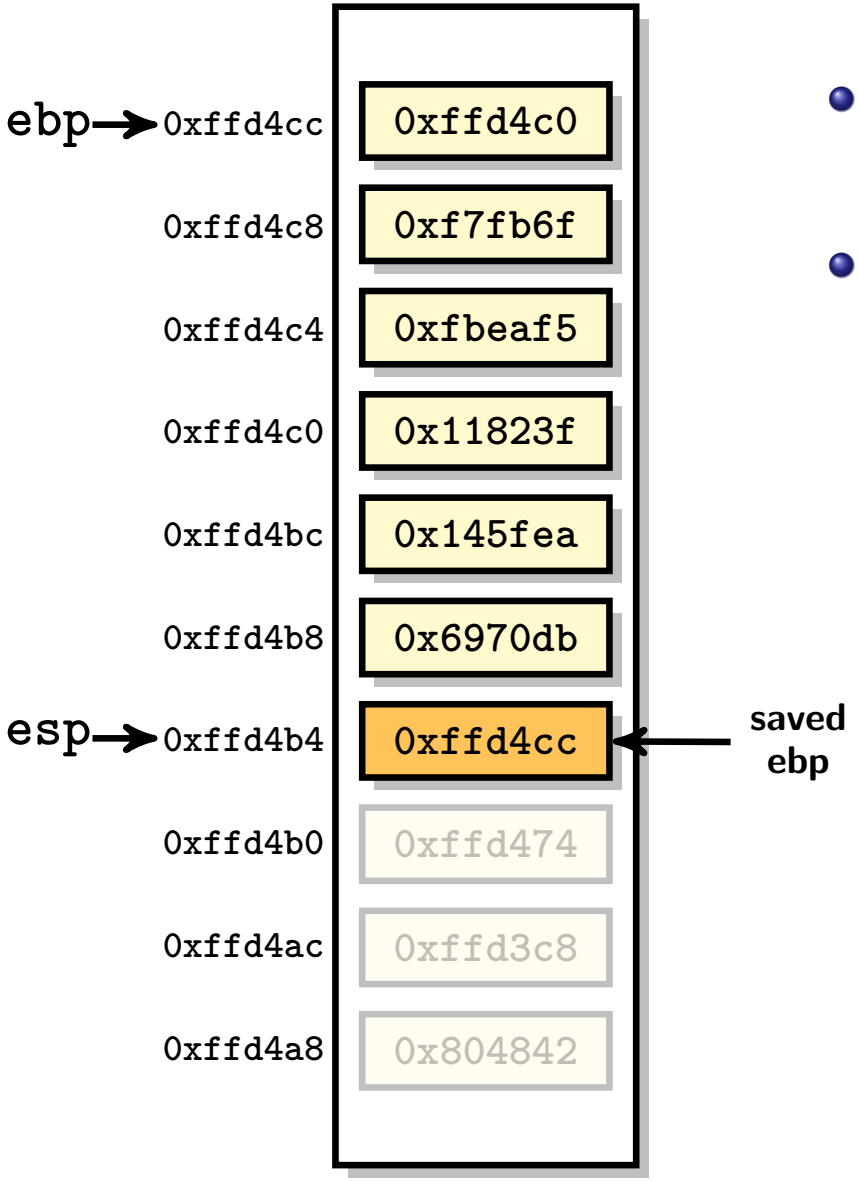
enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

Saving **ebp**.

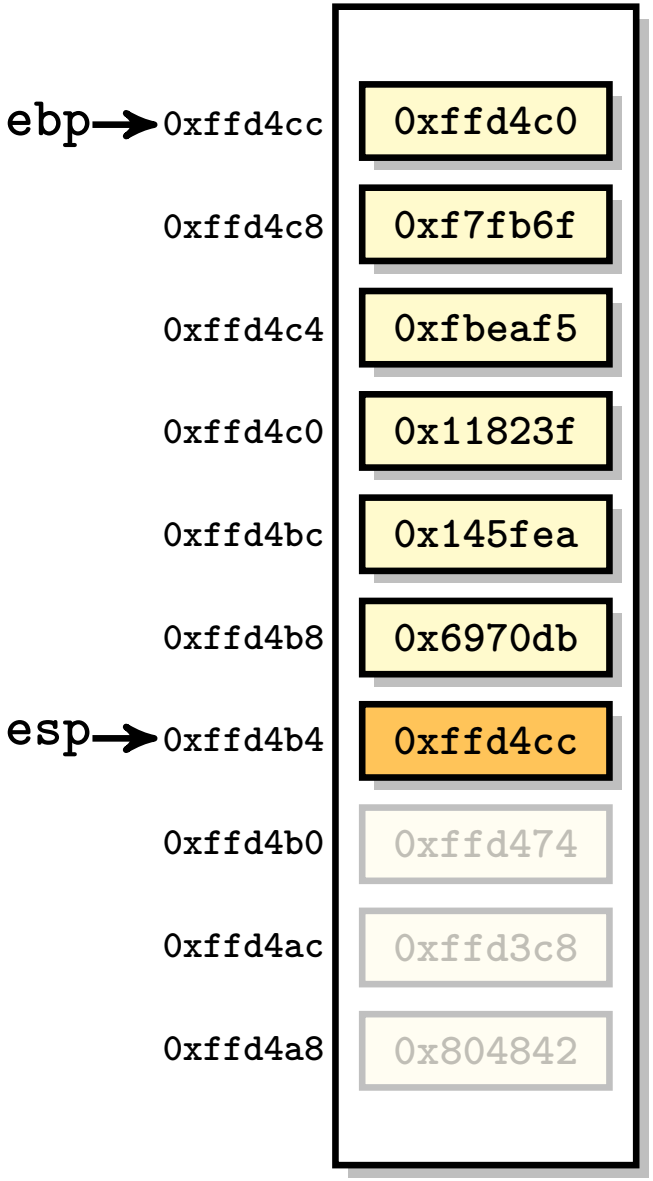
enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

Saving ebp.

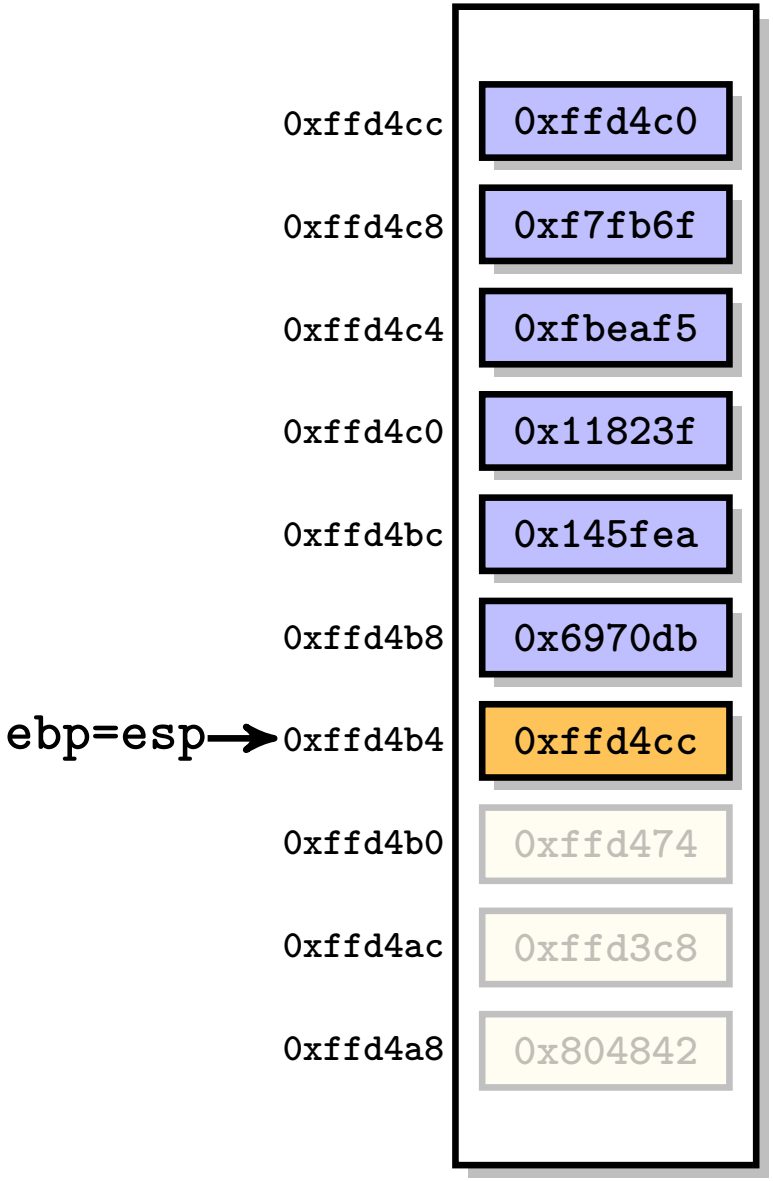
enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

Starting a new stack-frame.

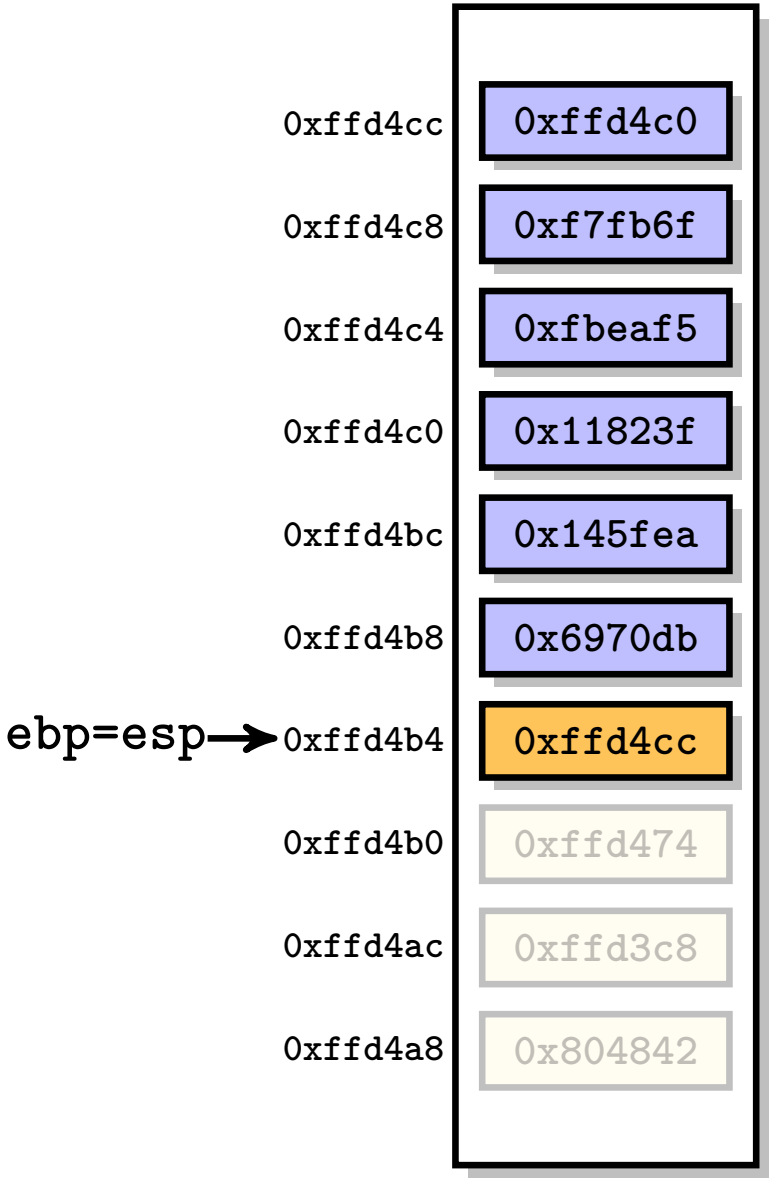
enter



- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

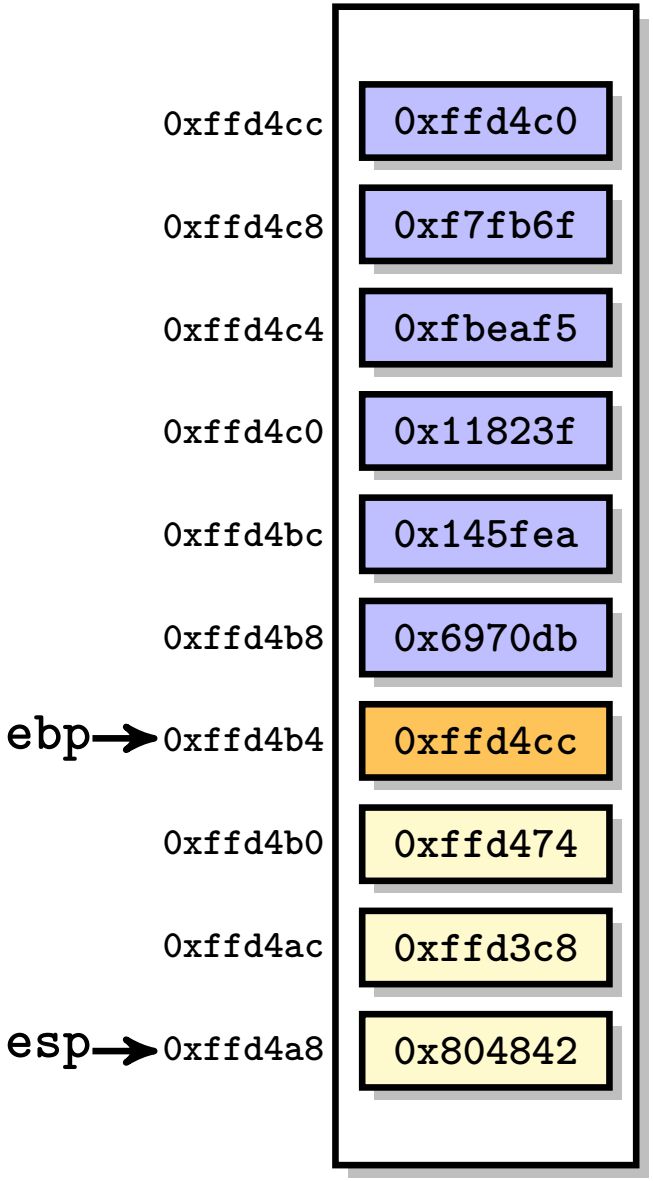
Starting a new stack-frame.

enter



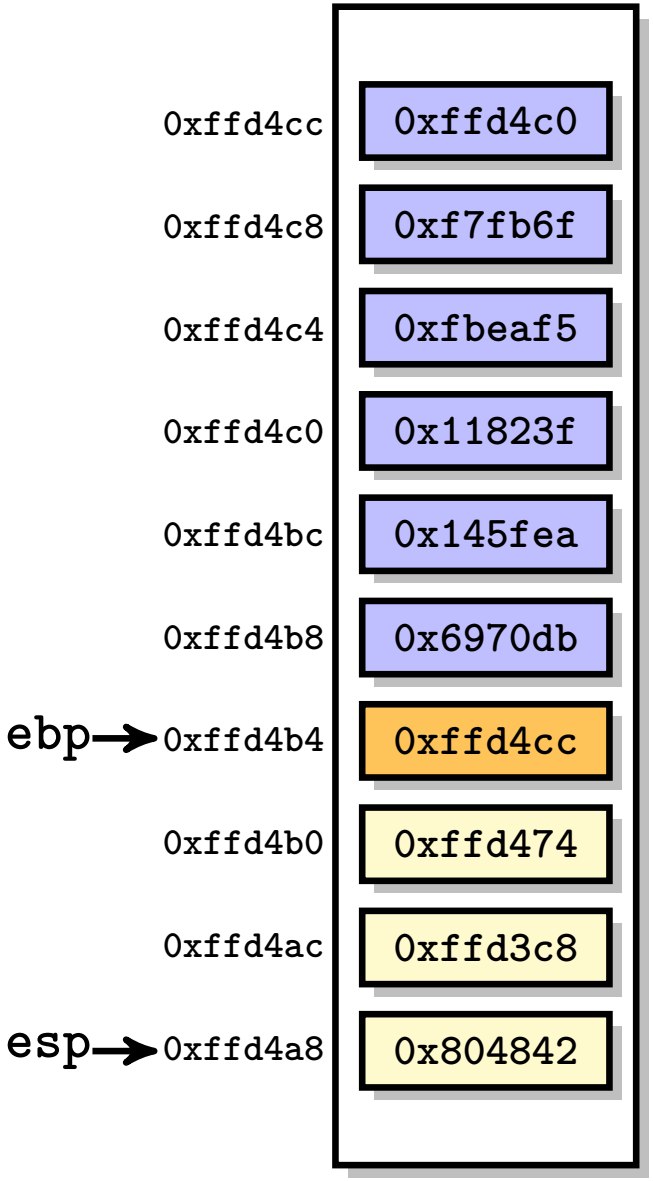
- **enter**
  (save previous stack-frame and create a fresh one)
- Actions performed:
  1. `push %ebp`
  2. `mov %esp, %ebp`

leave



- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

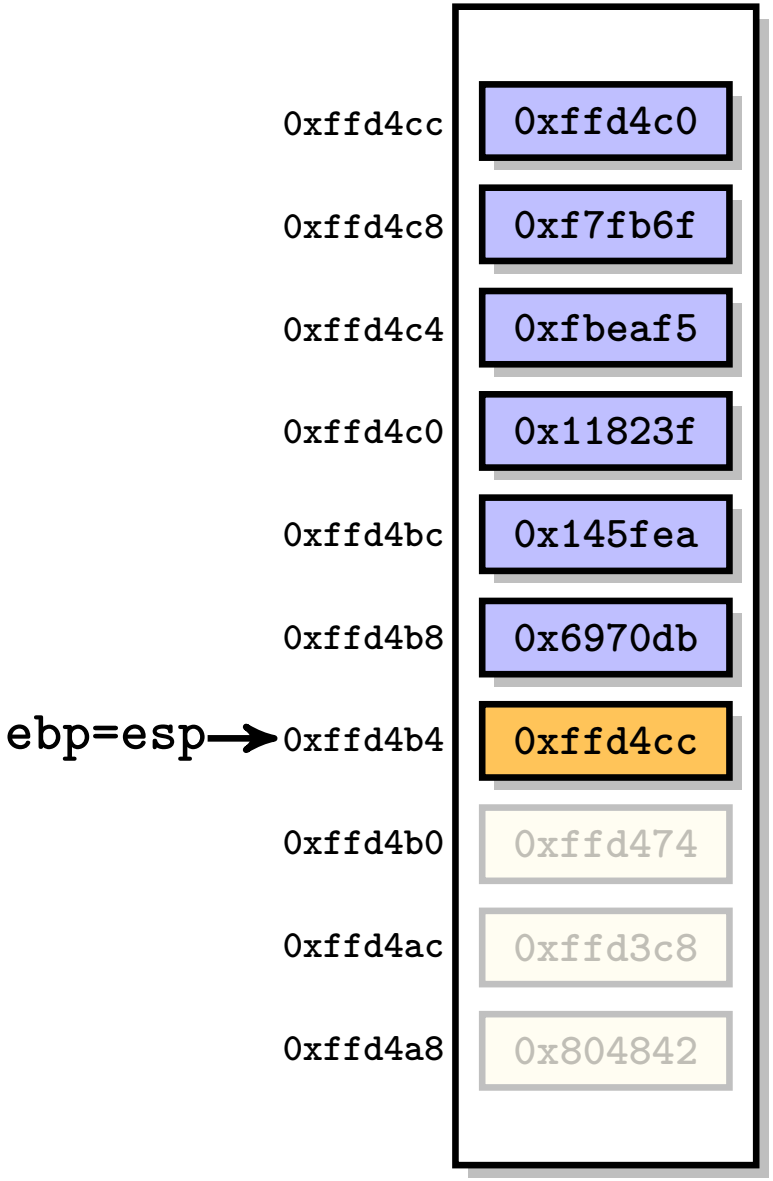
leave



- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

Cleaning the stack-frame

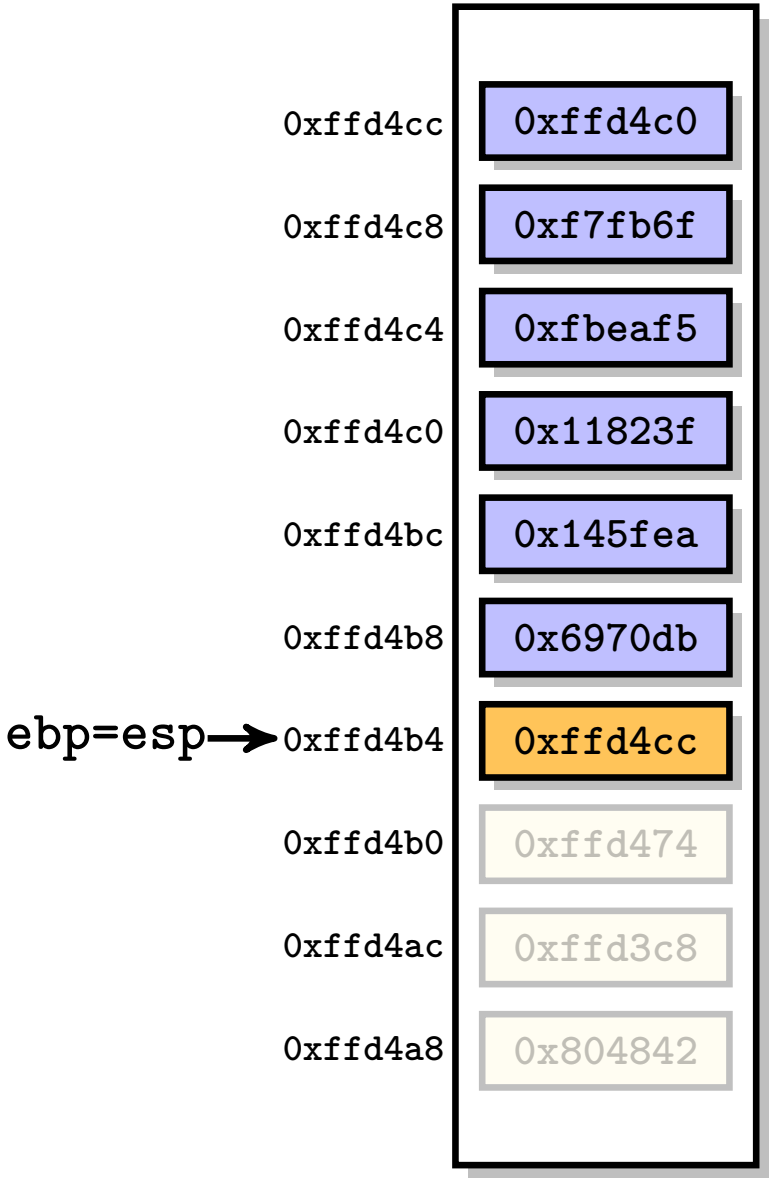
leave



- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

Cleaning the stack-frame

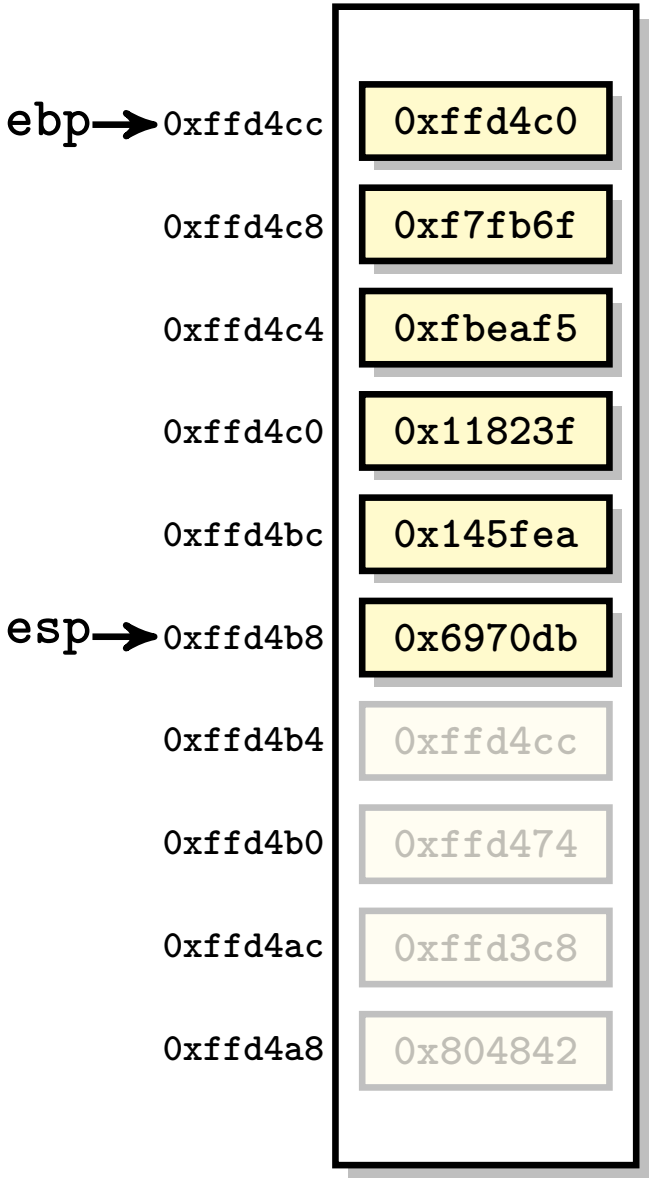
leave



- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

Restoring **ebp** register

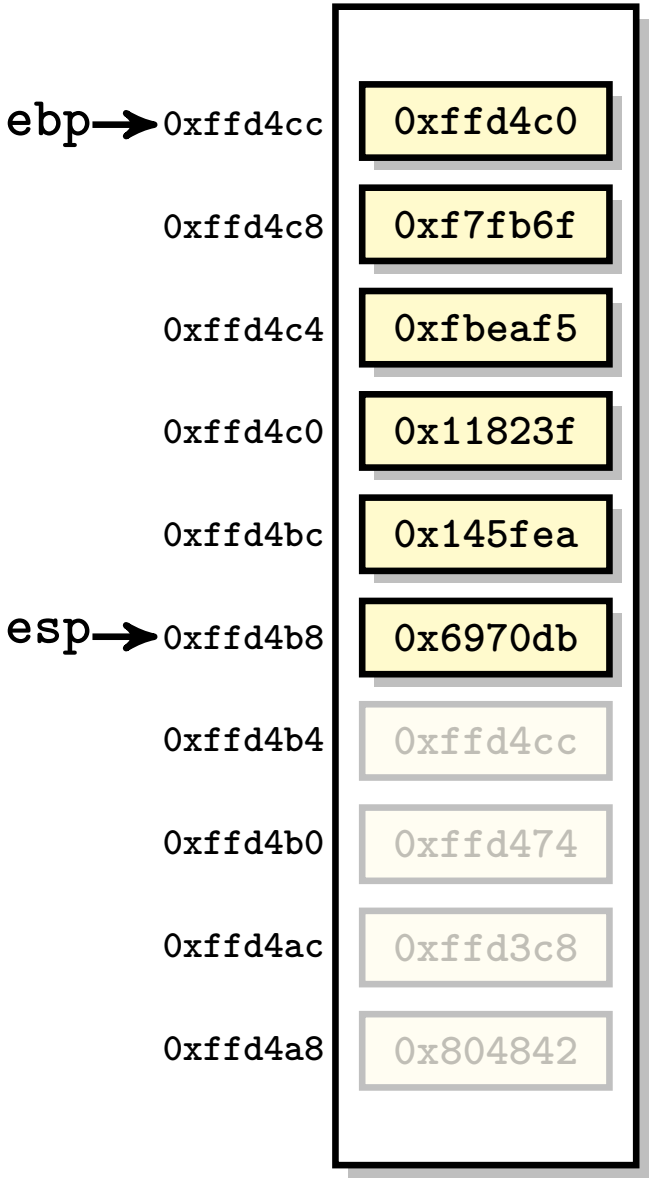
leave



- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

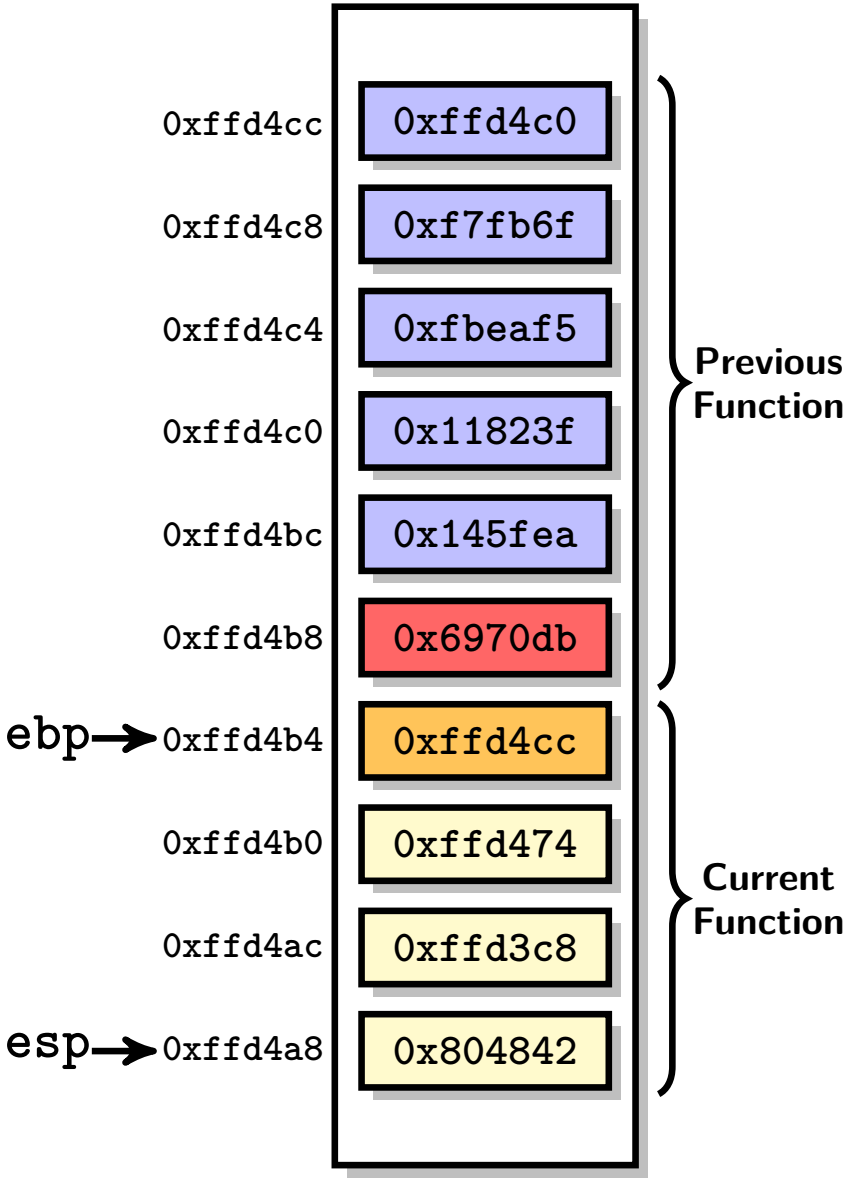
Restoring **ebp** register

leave



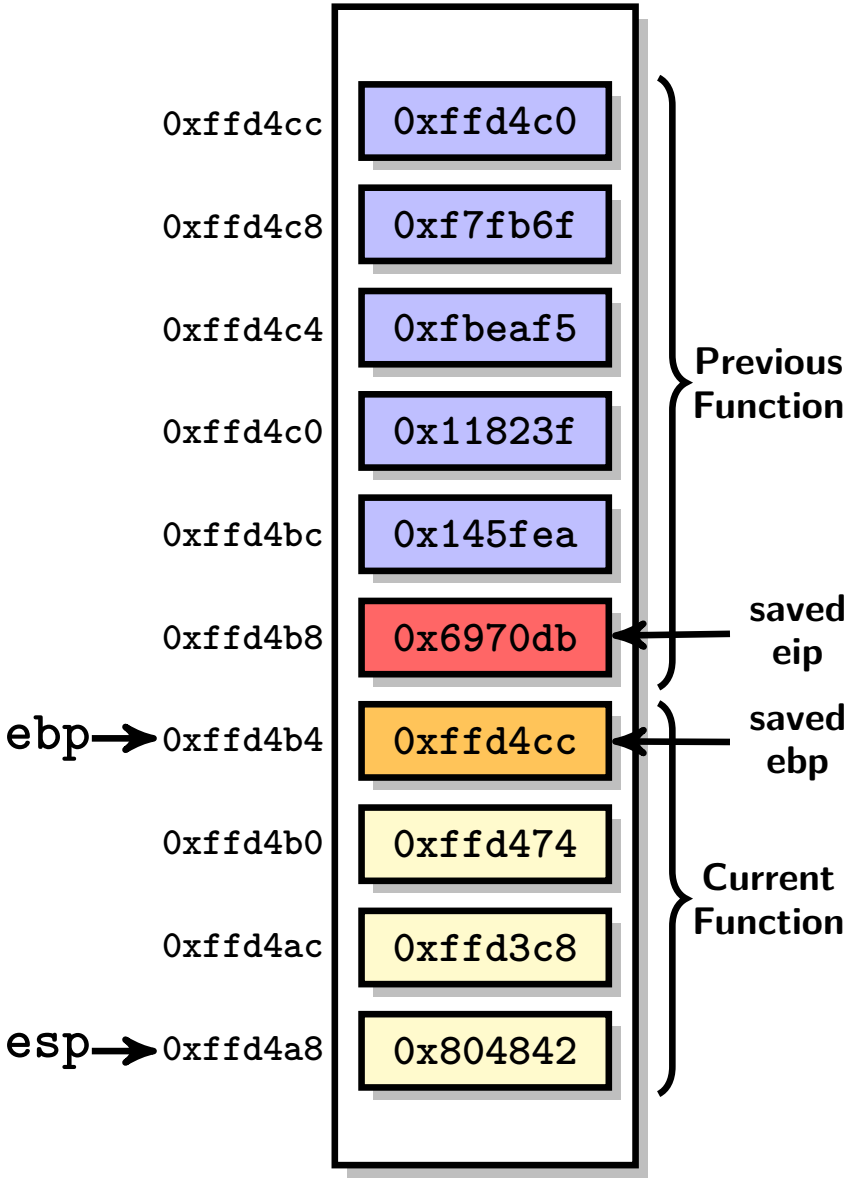
- `leave`
  (exit current stack-frame and restore previous one)
- Actions performed:
  1. `mov %ebp, %esp`
  2. `pop %ebp`

Call Stack



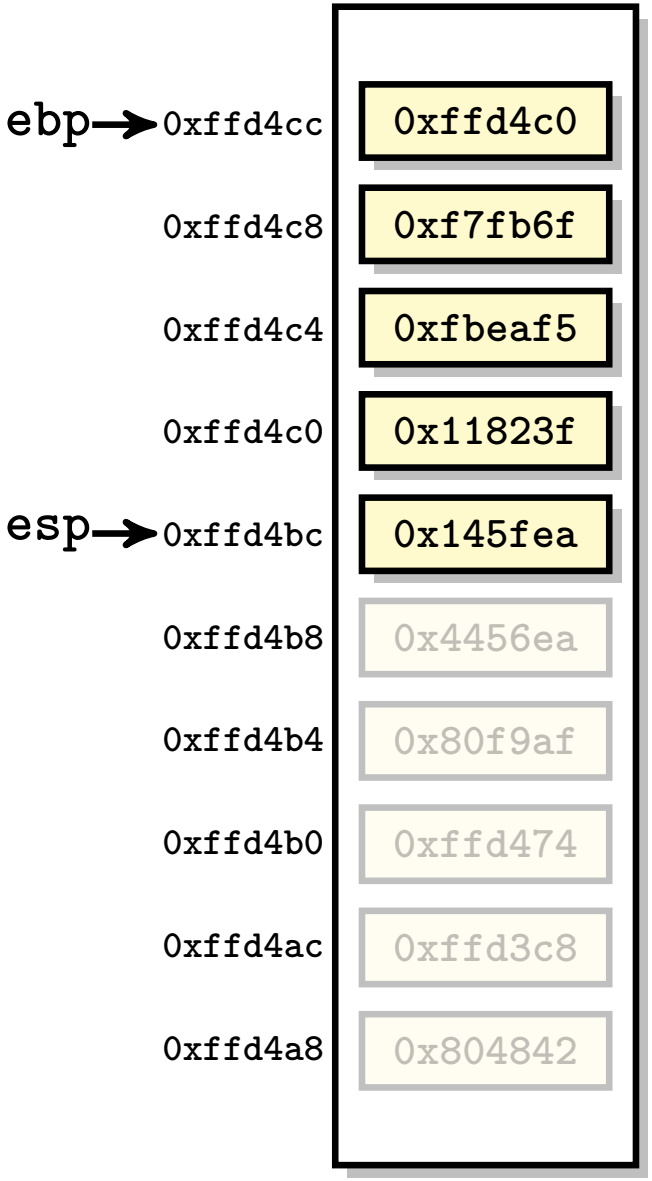
- When calling a function, one need to save the context of the current function (next instruction to execute).
- The register **eip** (instruction-pointer) of the current function is pushed on the stack before leaving to the next function.
- Stack-frame top element is always the **saved eip** before leaving to another function.
- **eip** is saved on '**call**' and restored on '**ret**'.

Call Stack



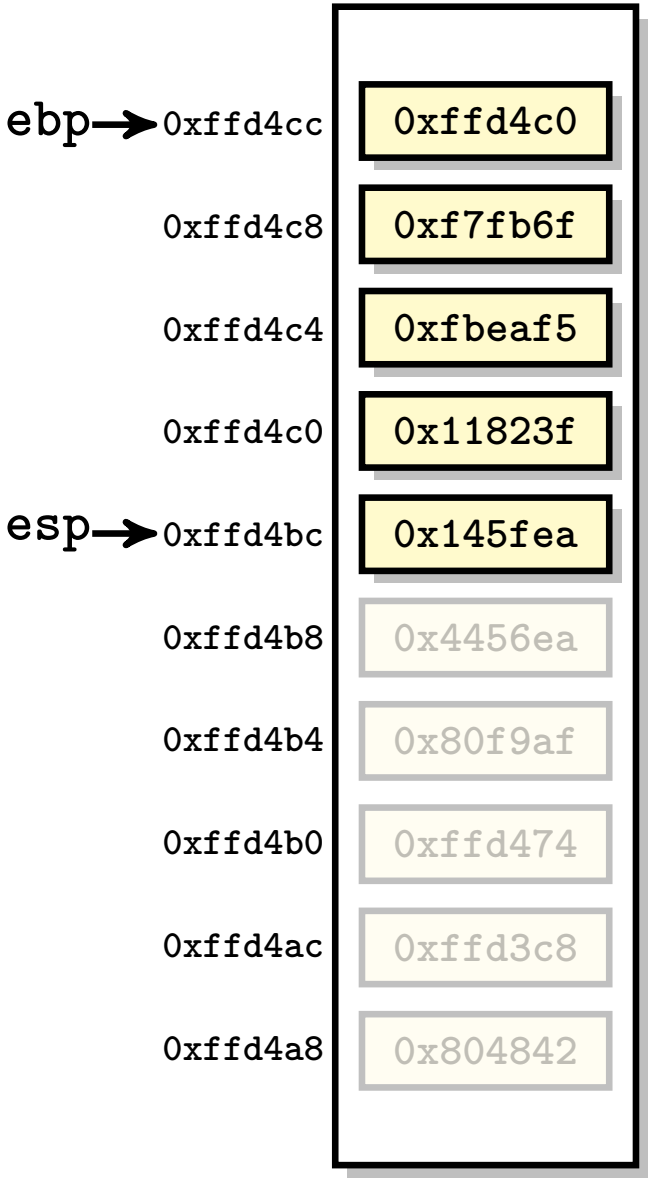
- When calling a function, one need to save the context of the current function (next instruction to execute).
- The register **eip** (instruction-pointer) of the current function is pushed on the stack before leaving to the next function.
- Stack-frame top element is always the **saved eip** before leaving to another function.
- **eip** is saved on '**call**' and restored on '**ret**'.

call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

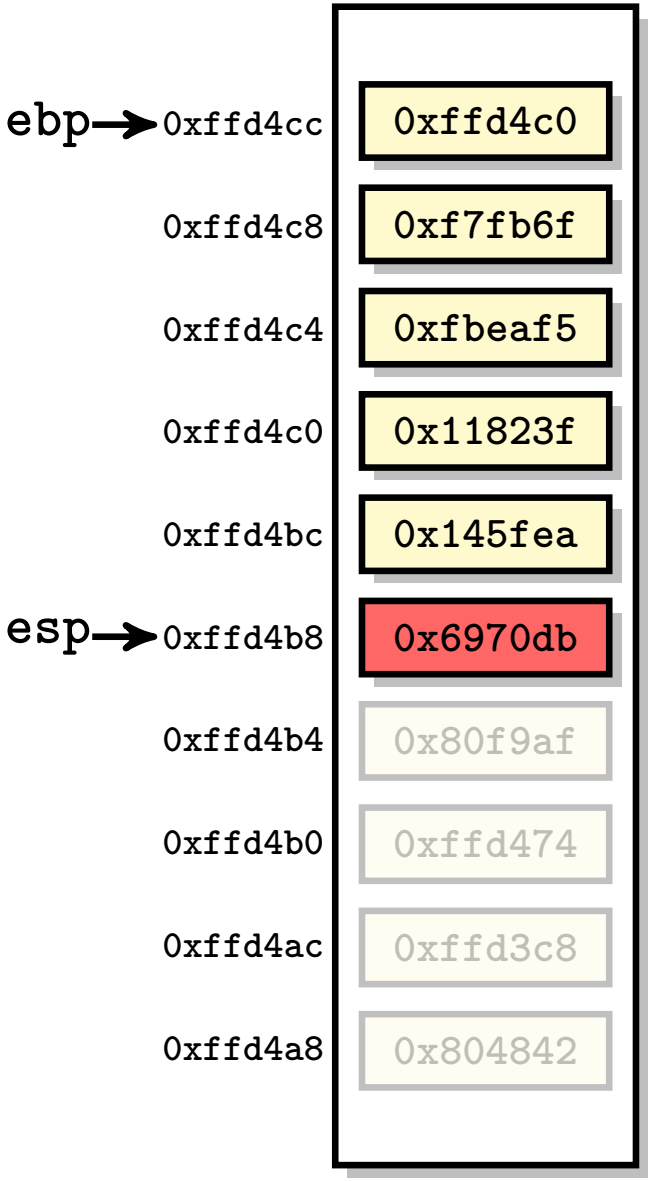
call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Saving `eip`.

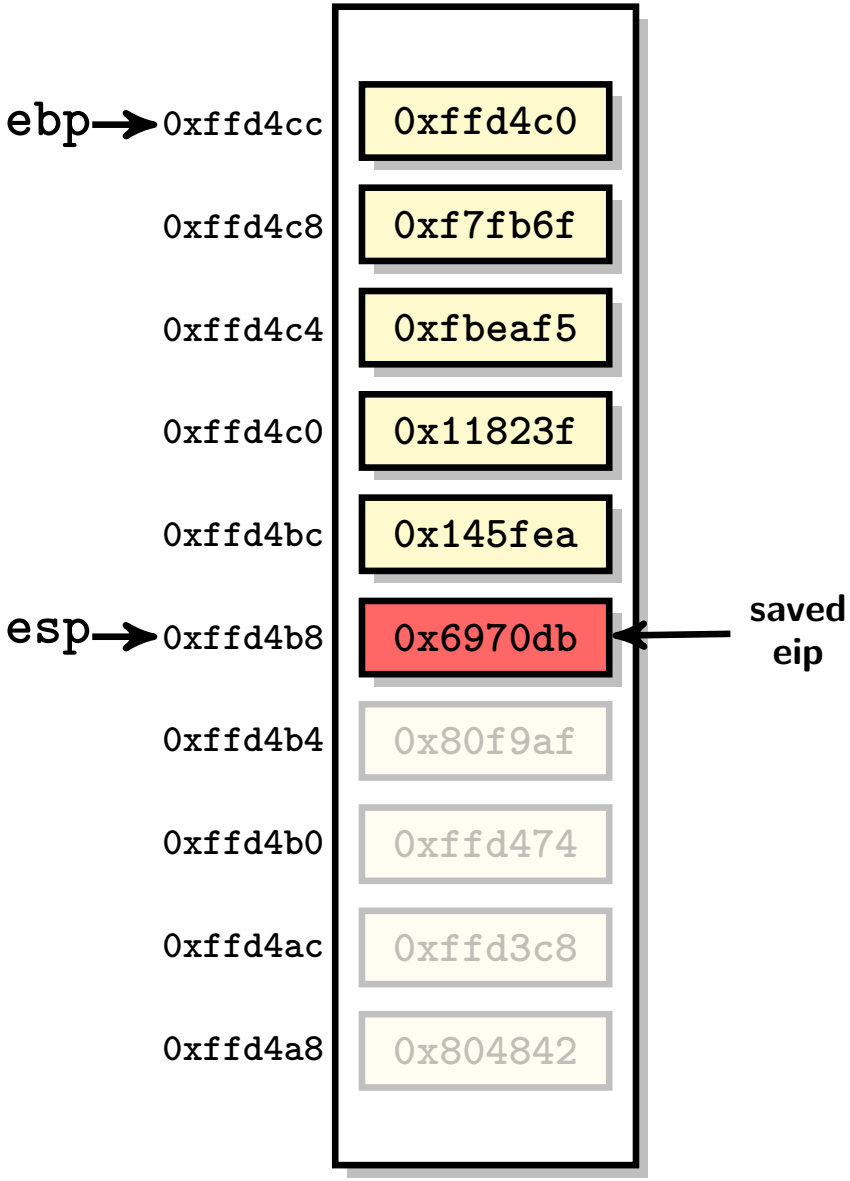
call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Saving `eip`.

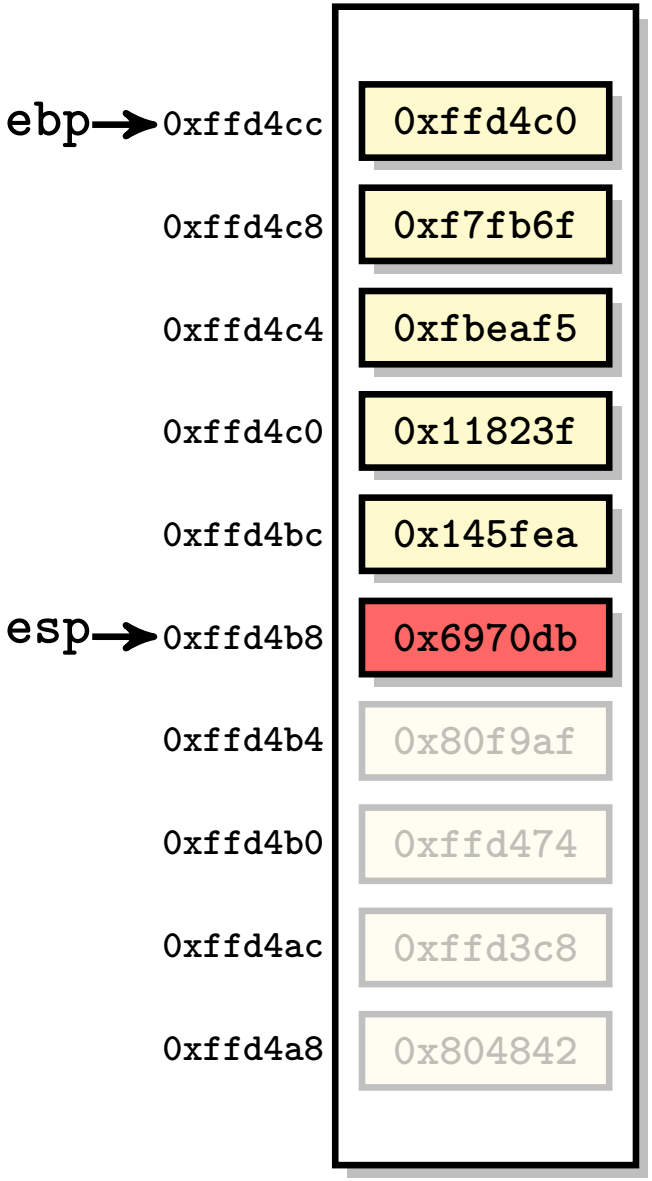
call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Saving `eip`.

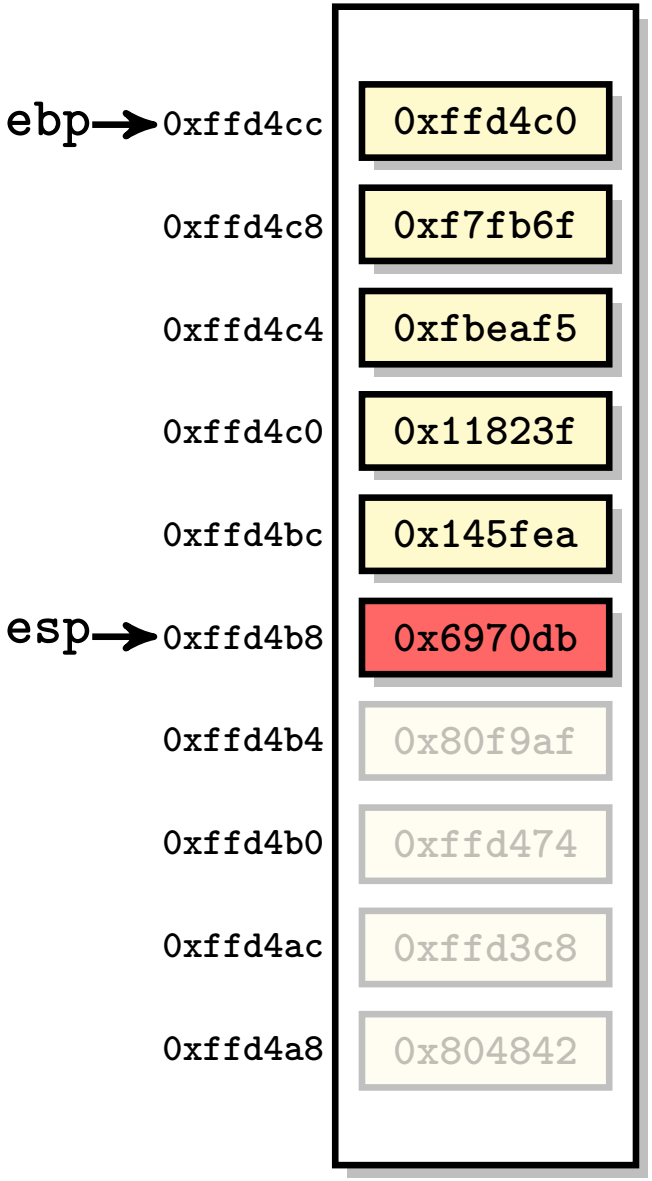
call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Saving `eip`.

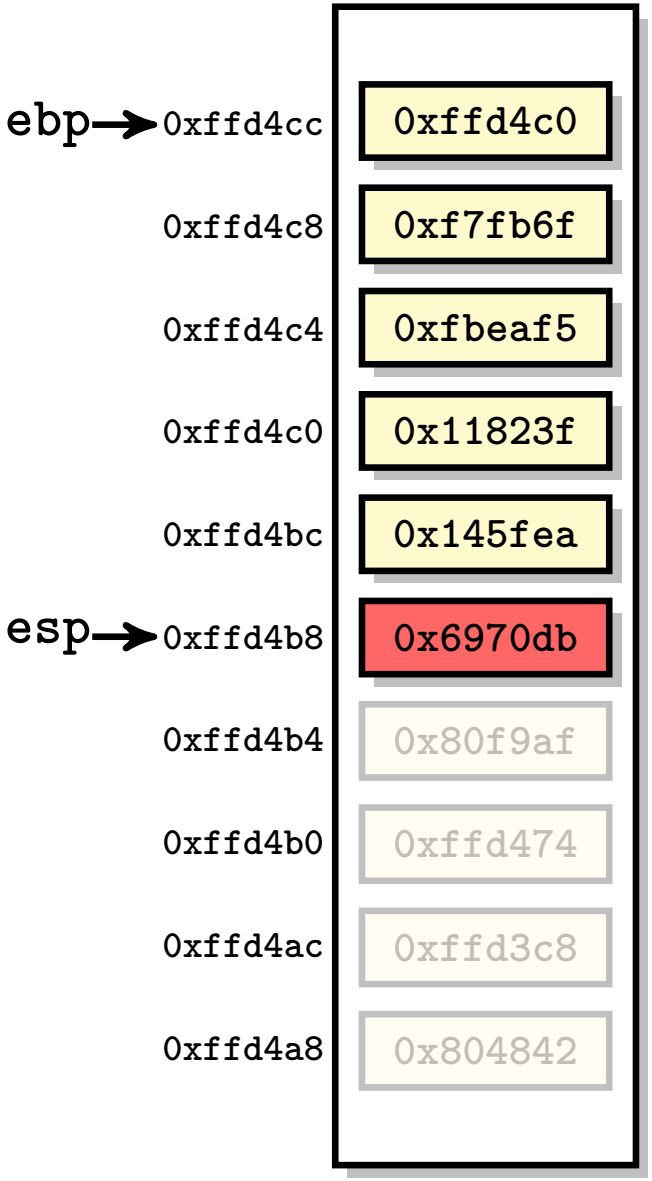
call



- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Setting `eip` to new address.

call

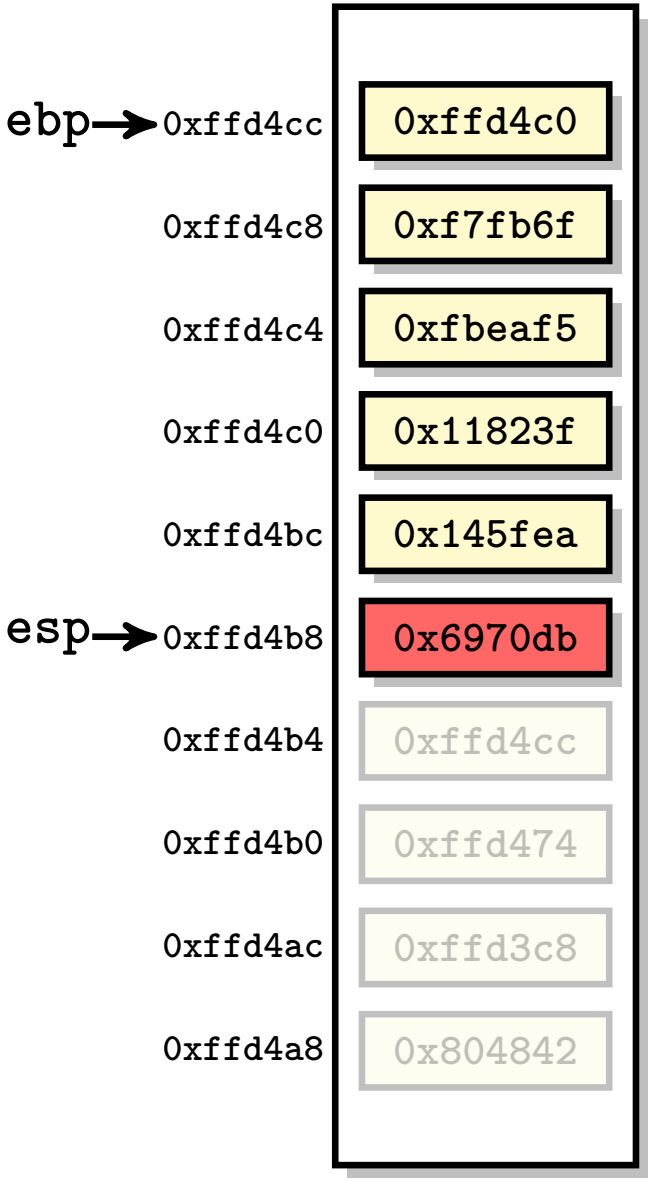


- `call <addr>`
  (save current `eip` and continue execution at `addr`)
- Actions performed:
  1. `push %eip`
  2. `mov addr, %eip`

Warning

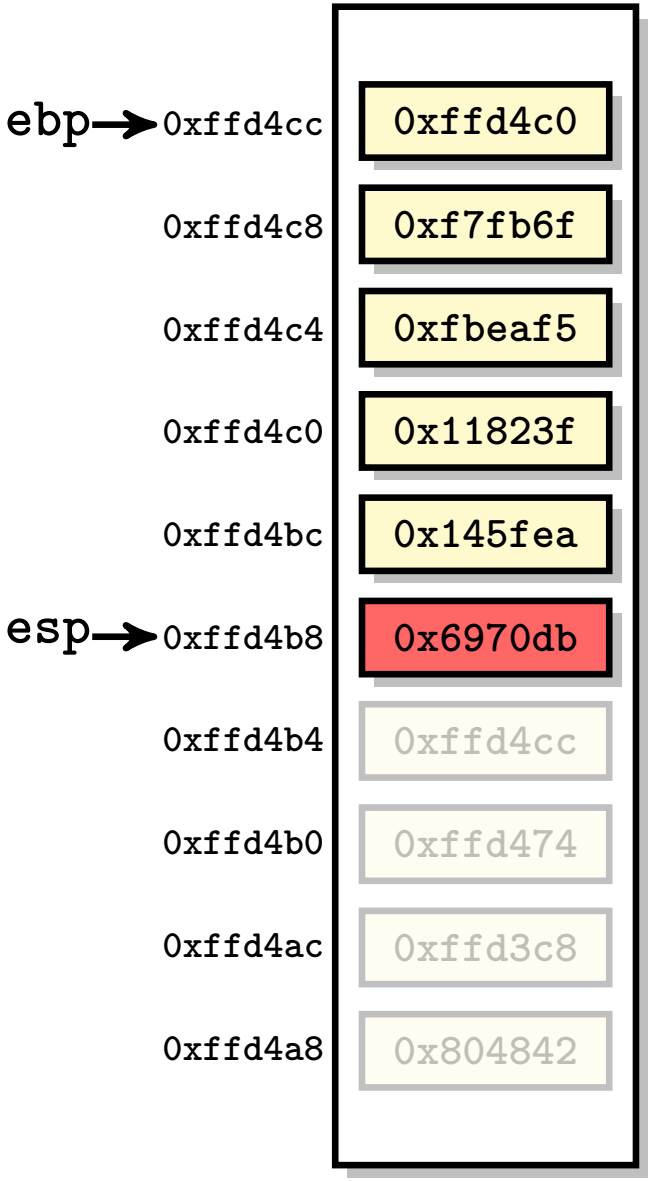
In x86-32 `eip` cannot be addressed as an operand.
So, these actions cannot really be executed.
Note that this is not anymore the case in x86-64.

ret



- **ret** (restore previous execution)
- Actions performed:
  1. pop %eip

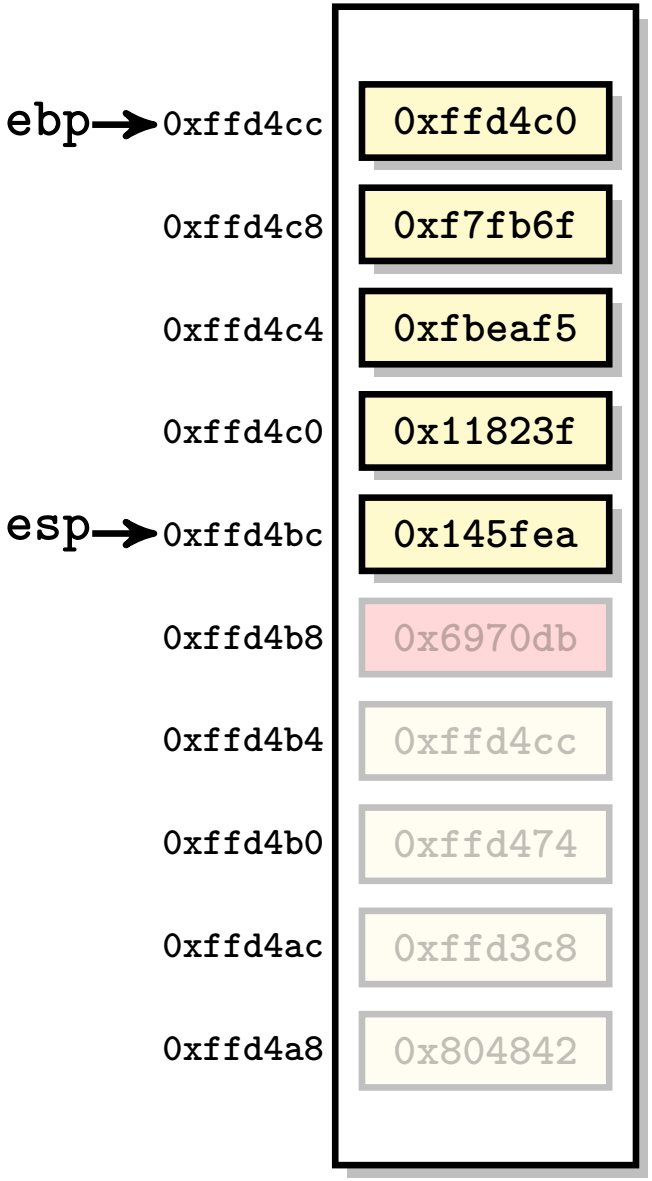
ret



- **ret** (restore previous execution)
- Actions performed:
  1. `pop %eip`

Restoring **eip** register

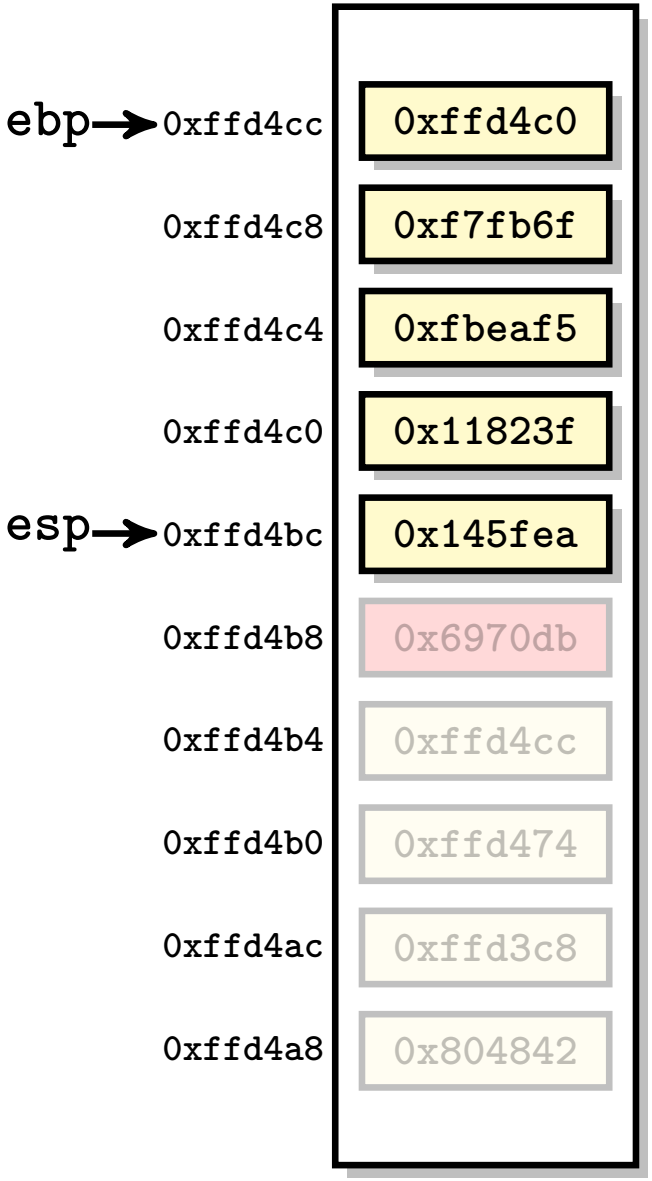
ret



- **ret** (restore previous execution)
- Actions performed:
  1. `pop %eip`

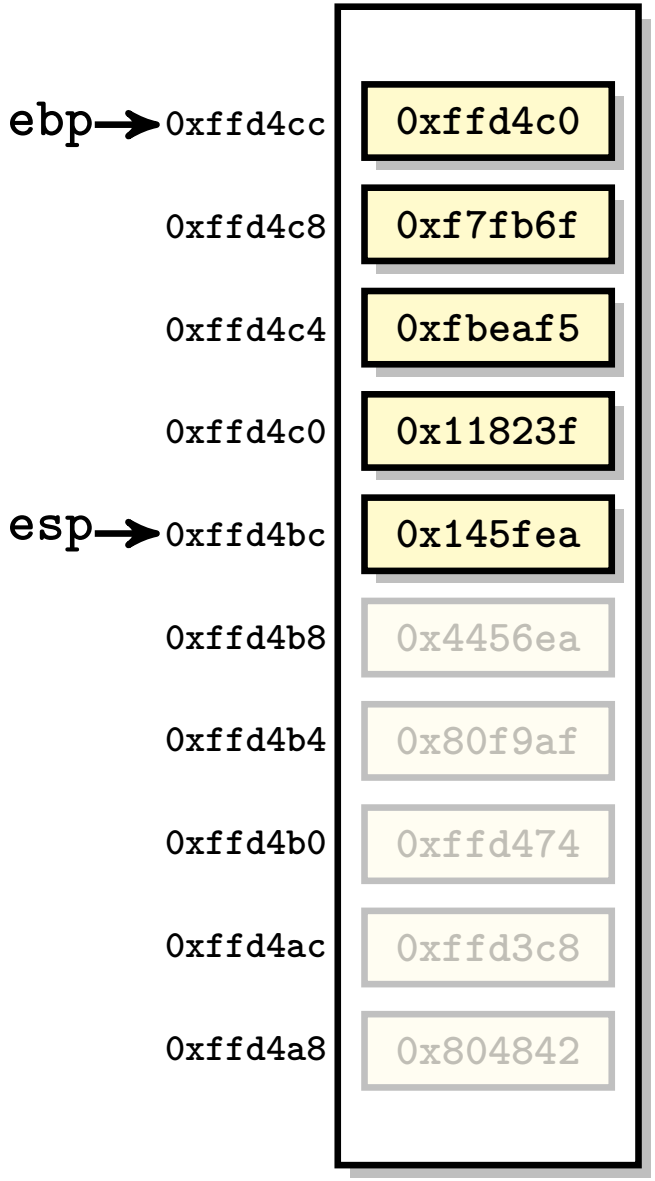
Restoring **eip** register

ret



- **ret** (restore previous execution)
- Actions performed:
  1. `pop %eip`

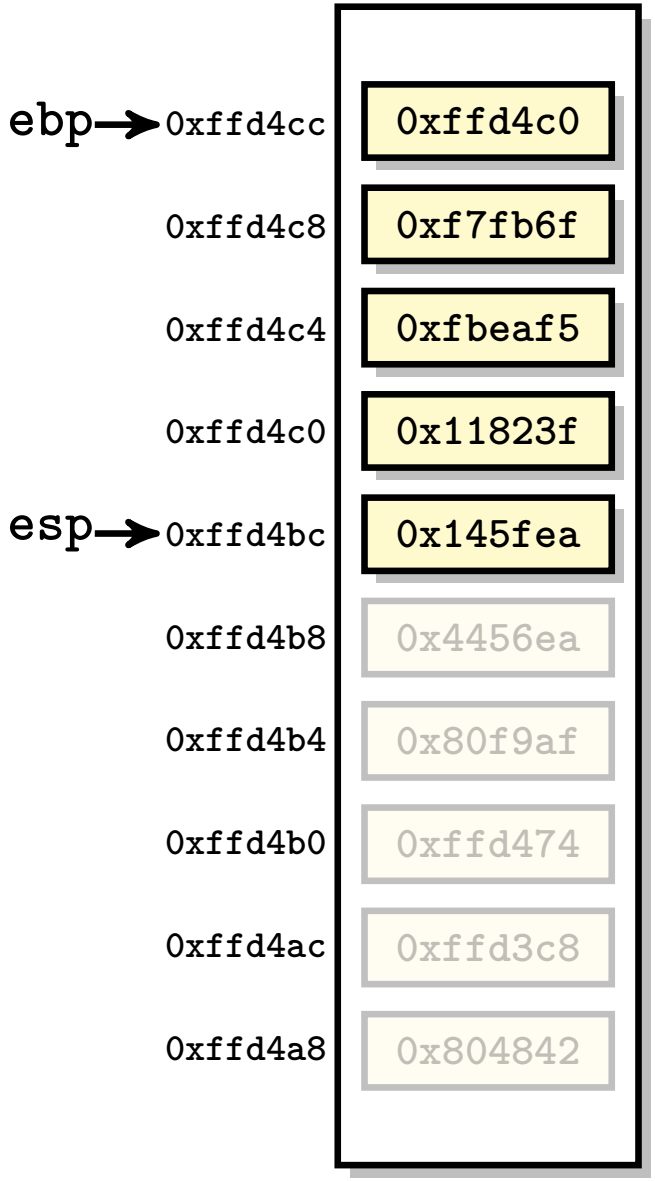
A Full Example (Entering a function)



Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

A Full Example (Entering a function)

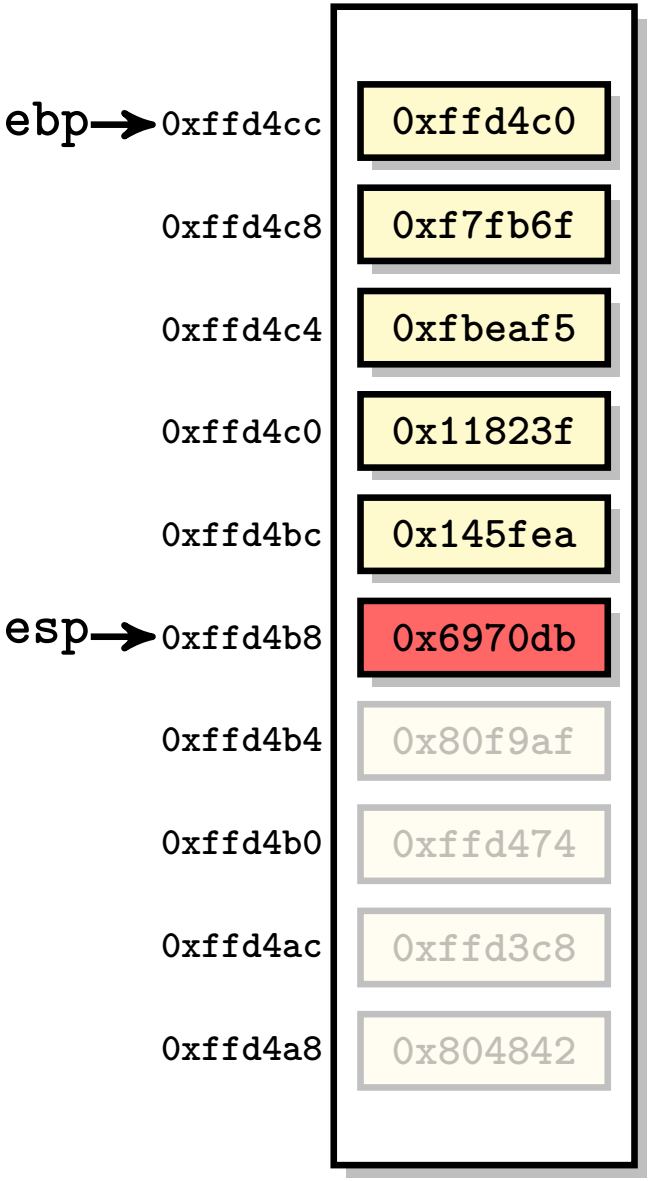


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving `eip` and setting it.

A Full Example (Entering a function)

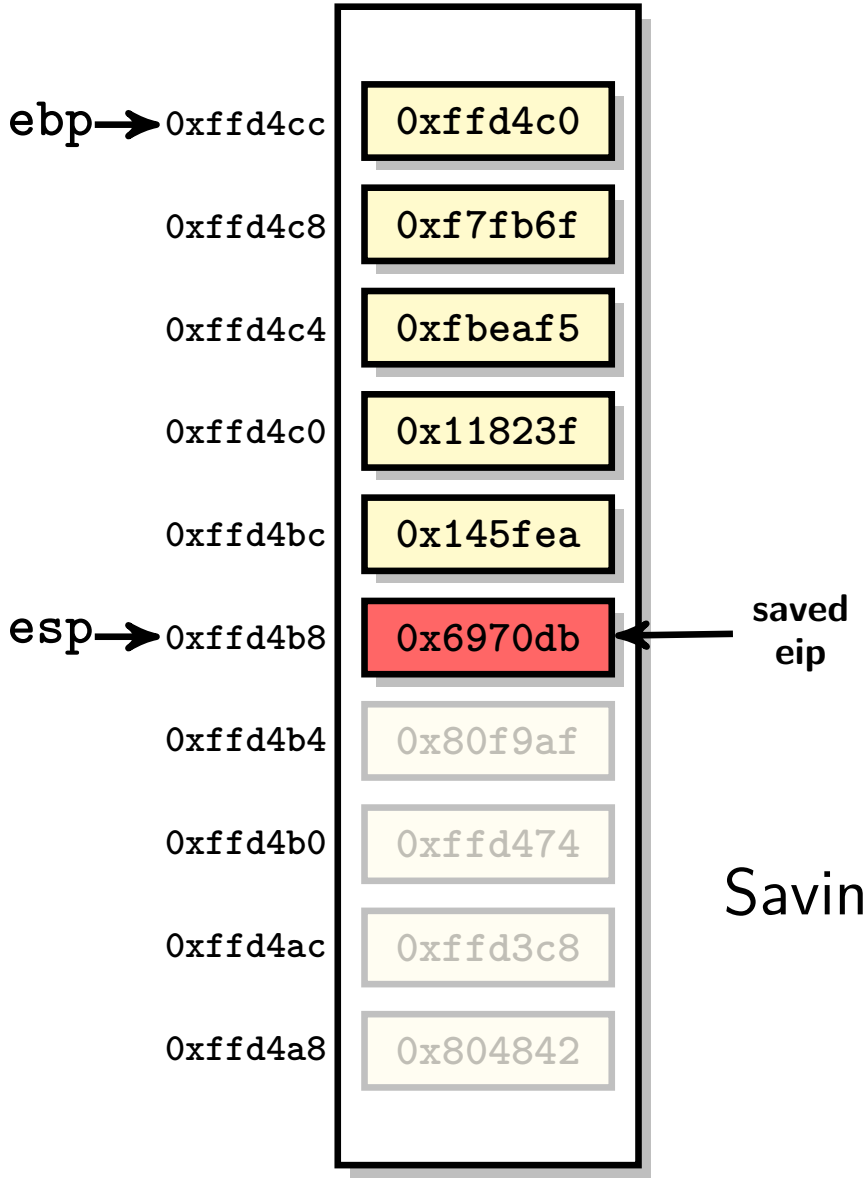


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving `eip` and setting it.

A Full Example (Entering a function)

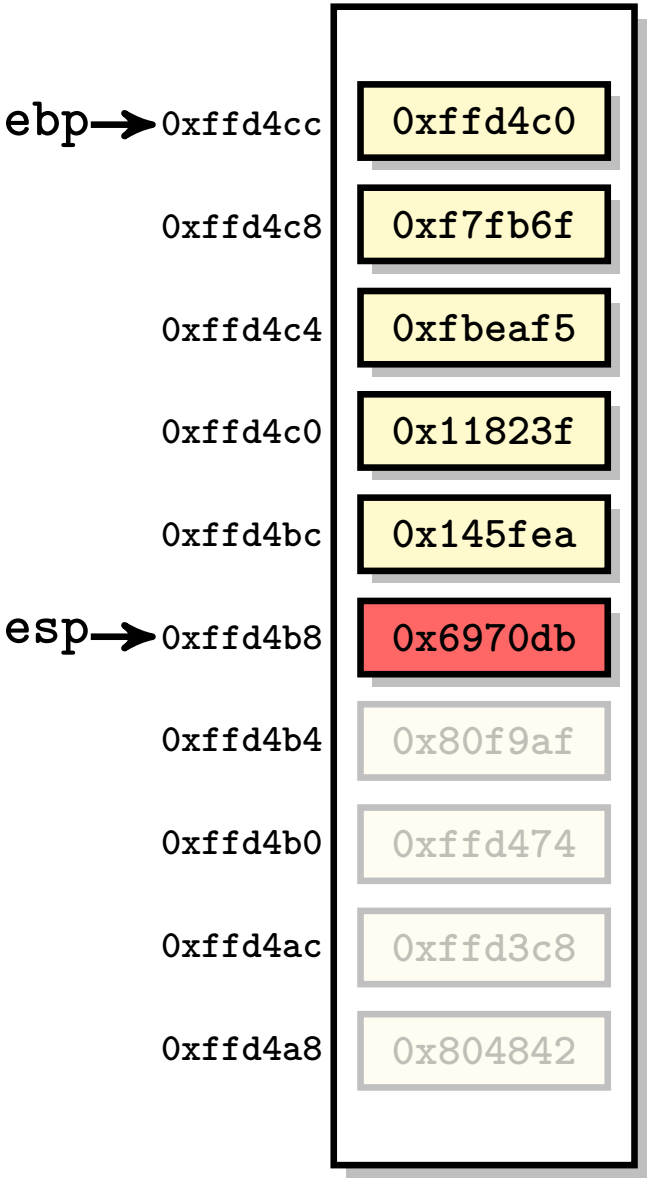


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving `eip` and setting it.

A Full Example (Entering a function)

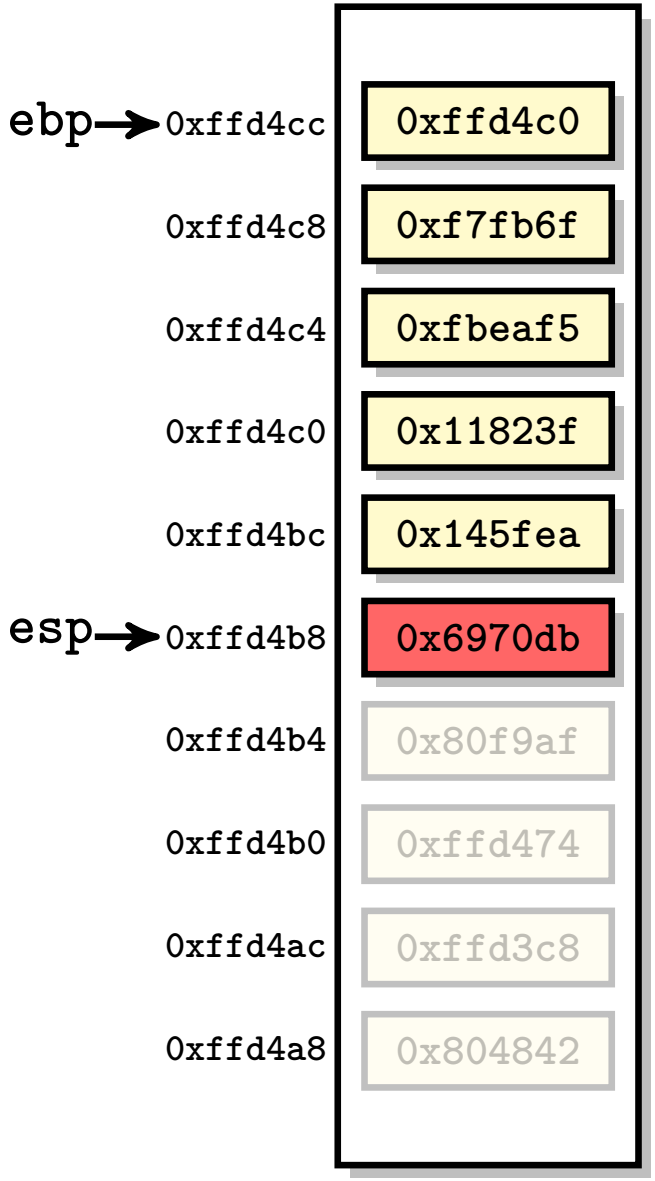


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving `eip` and setting it.

A Full Example (Entering a function)

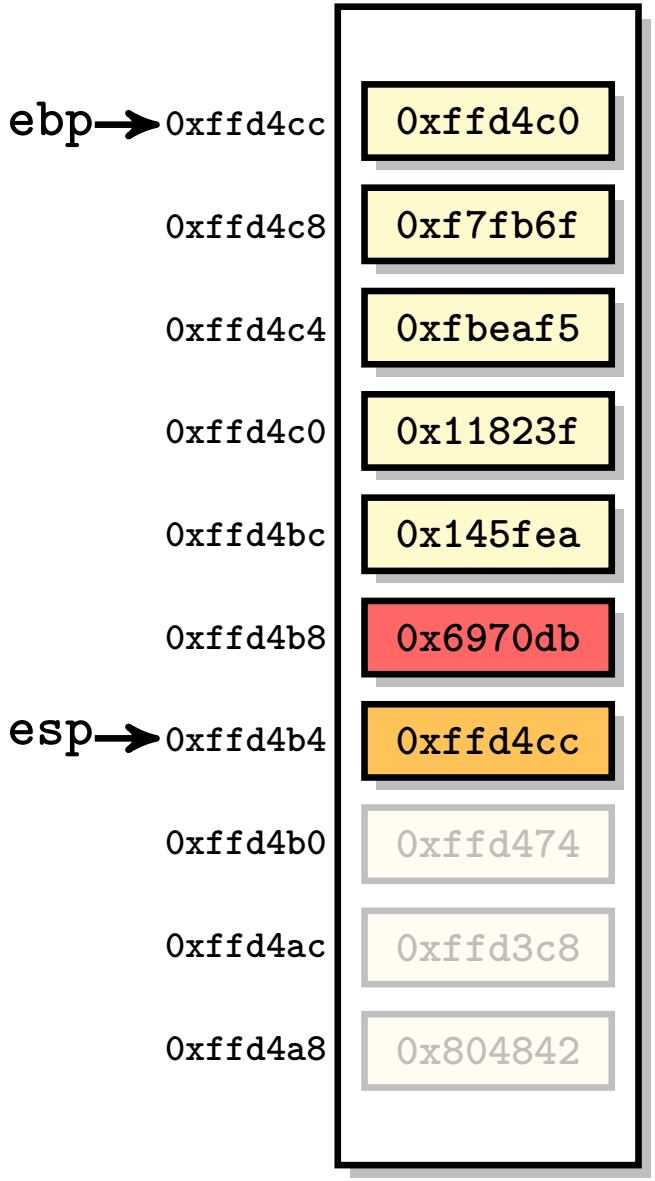


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving ebp.

A Full Example (Entering a function)

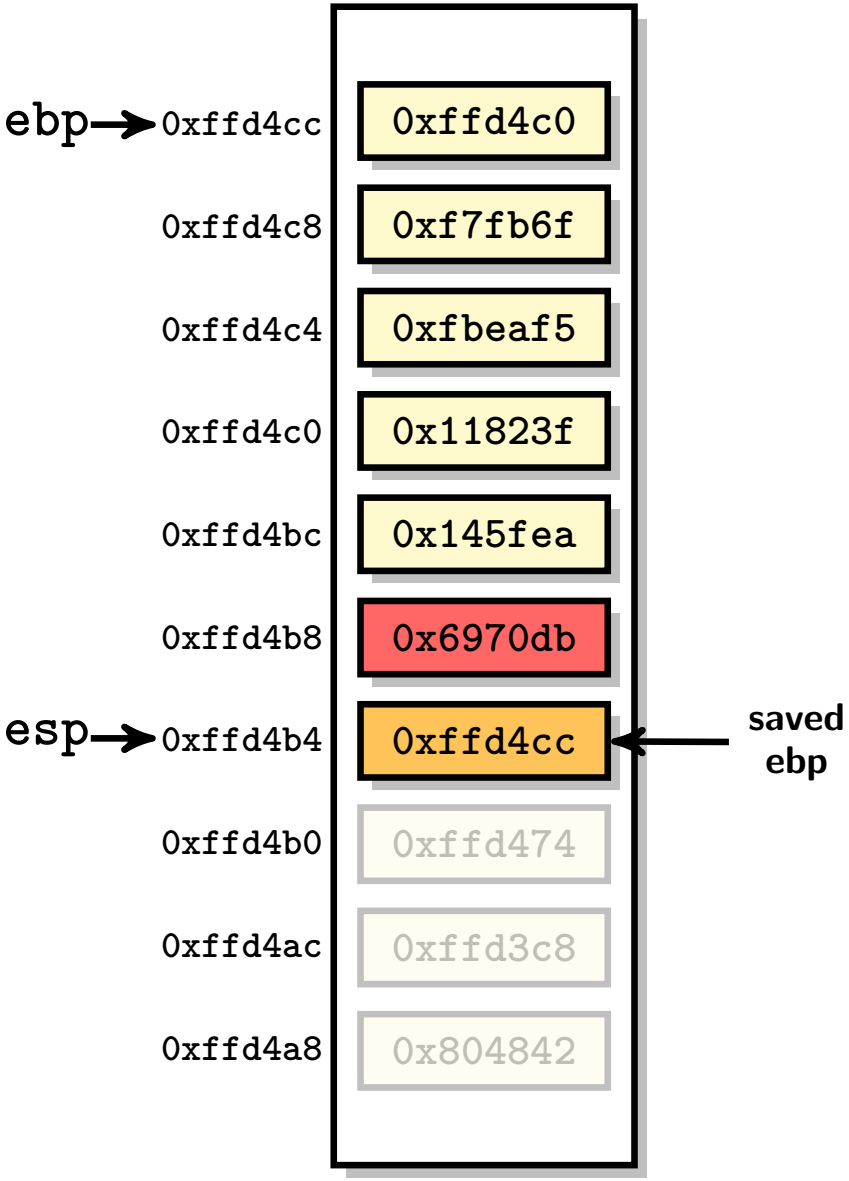


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving ebp.

A Full Example (Entering a function)

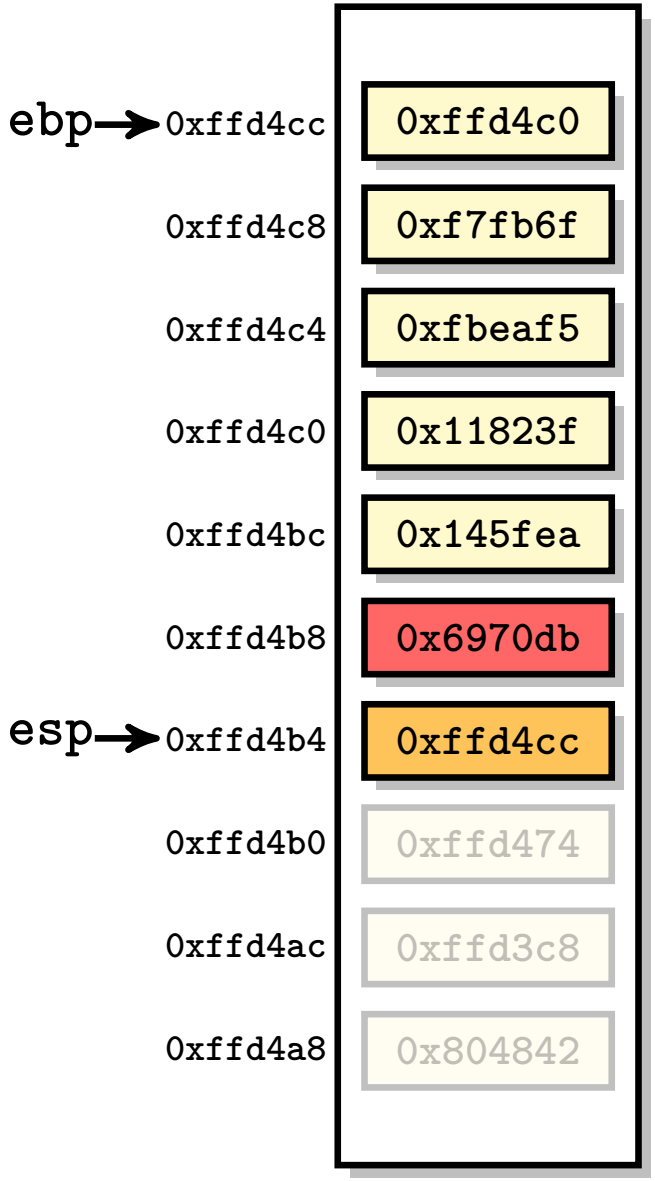


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving ebp.

A Full Example (Entering a function)

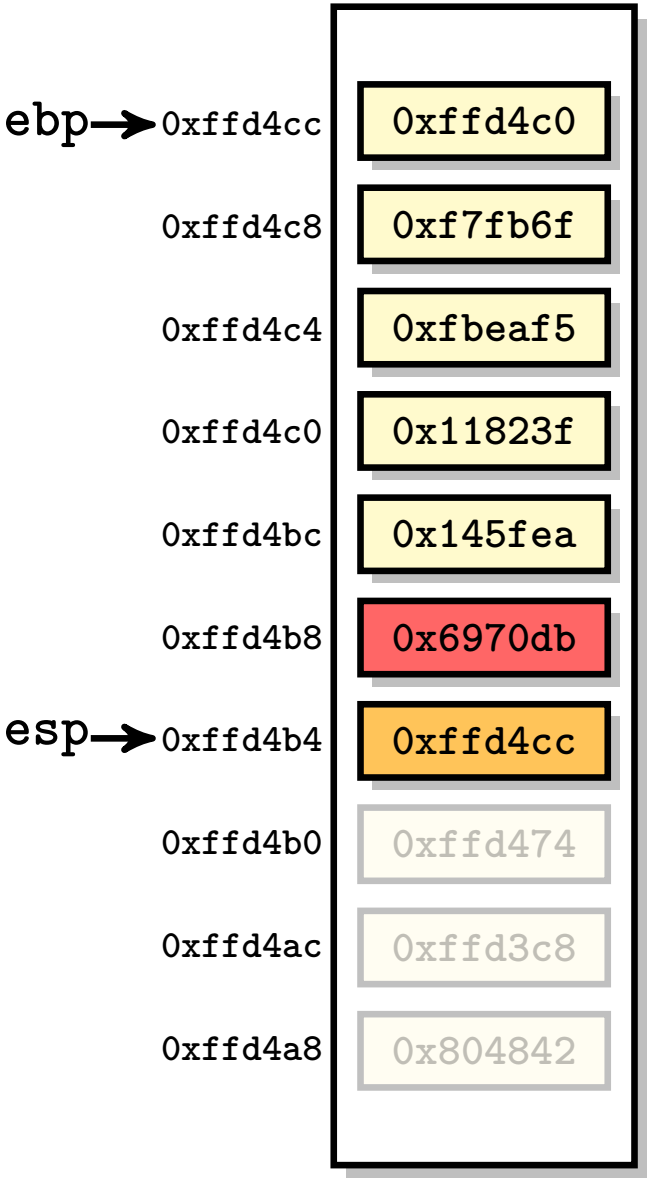


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Saving ebp.

A Full Example (Entering a function)

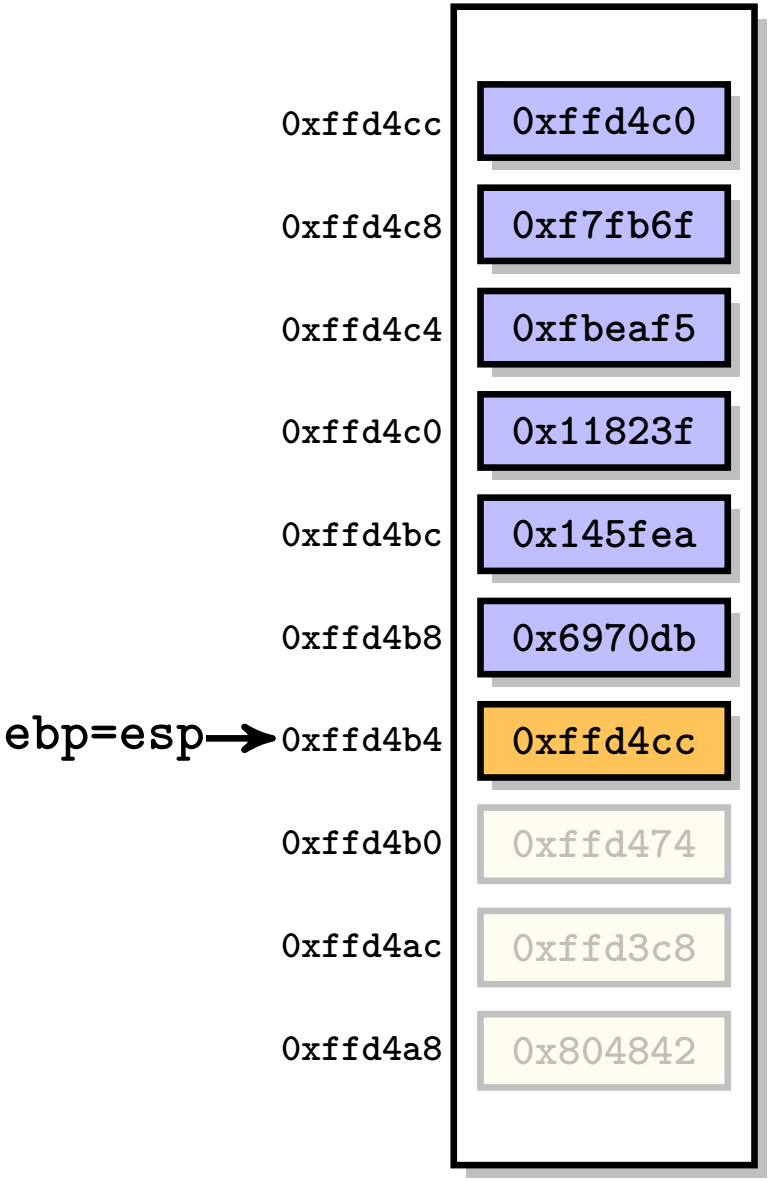


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Starting a new stack-frame.

A Full Example (Entering a function)

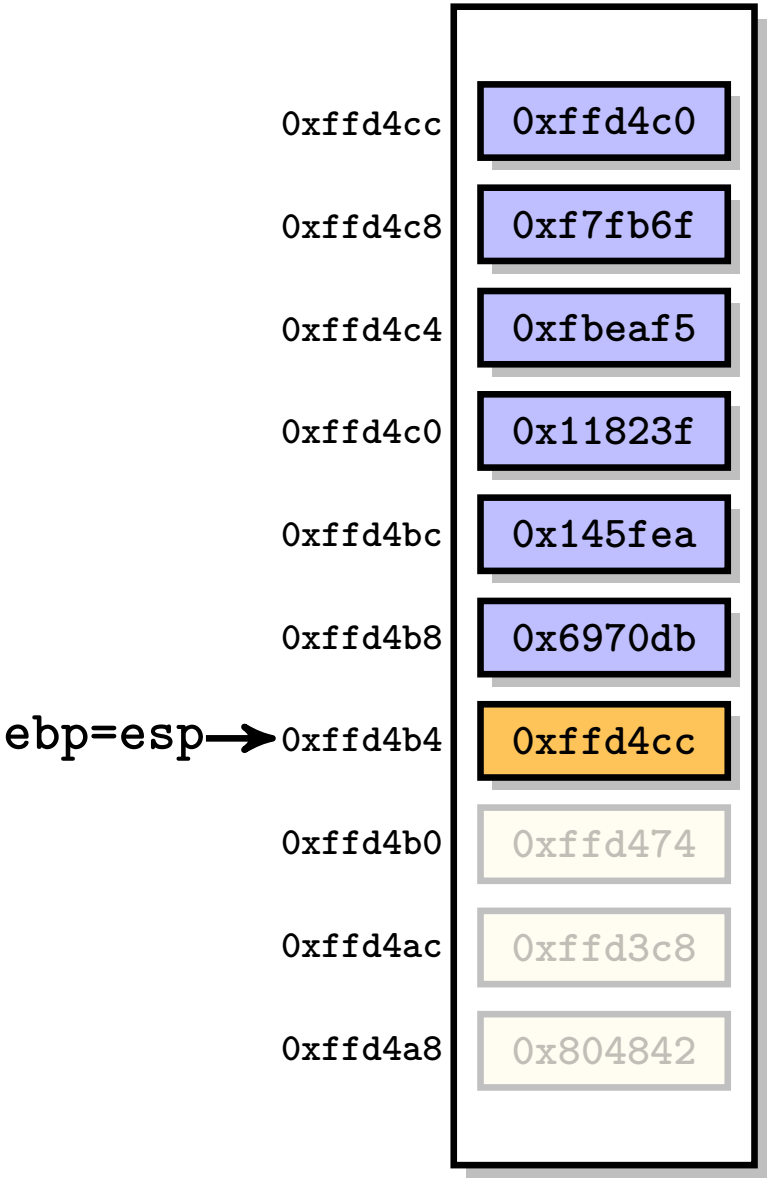


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Starting a new stack-frame.

A Full Example (Entering a function)

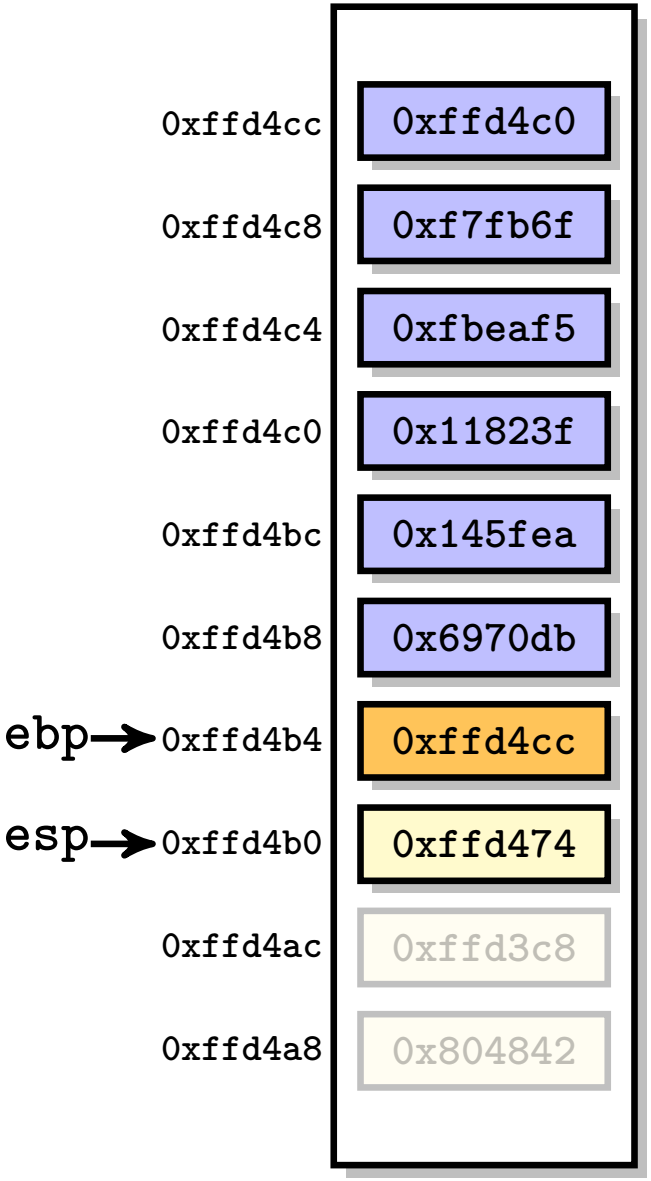


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Aligning data for efficiency.

A Full Example (Entering a function)

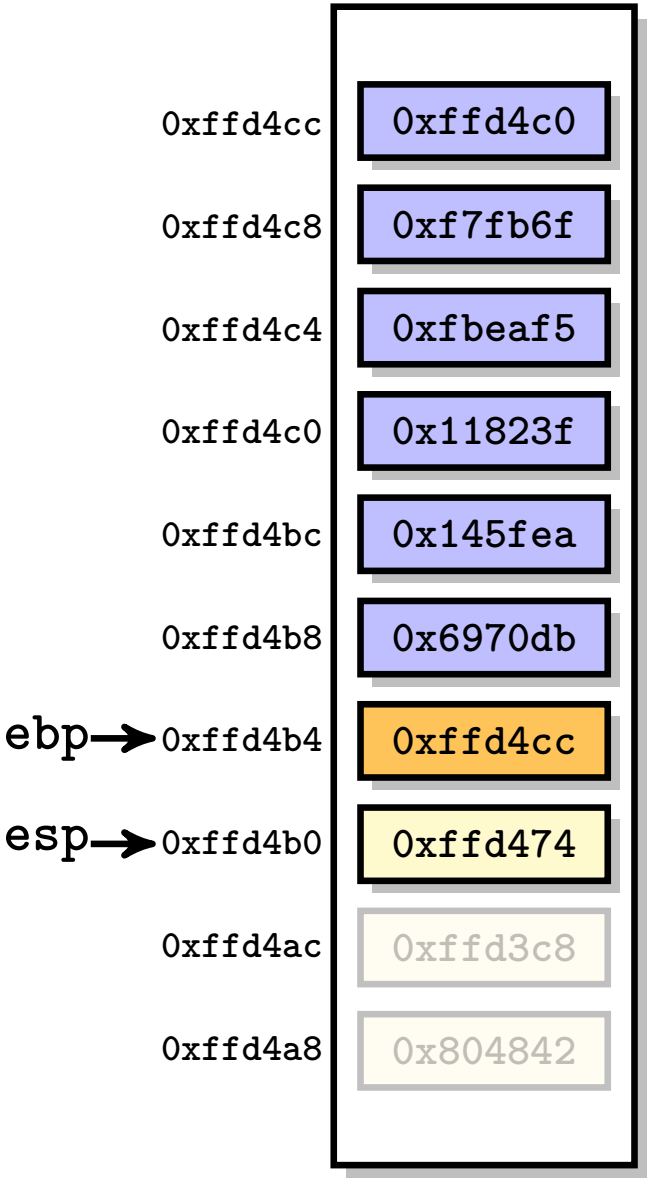


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Aligning data for efficiency.

A Full Example (Entering a function)

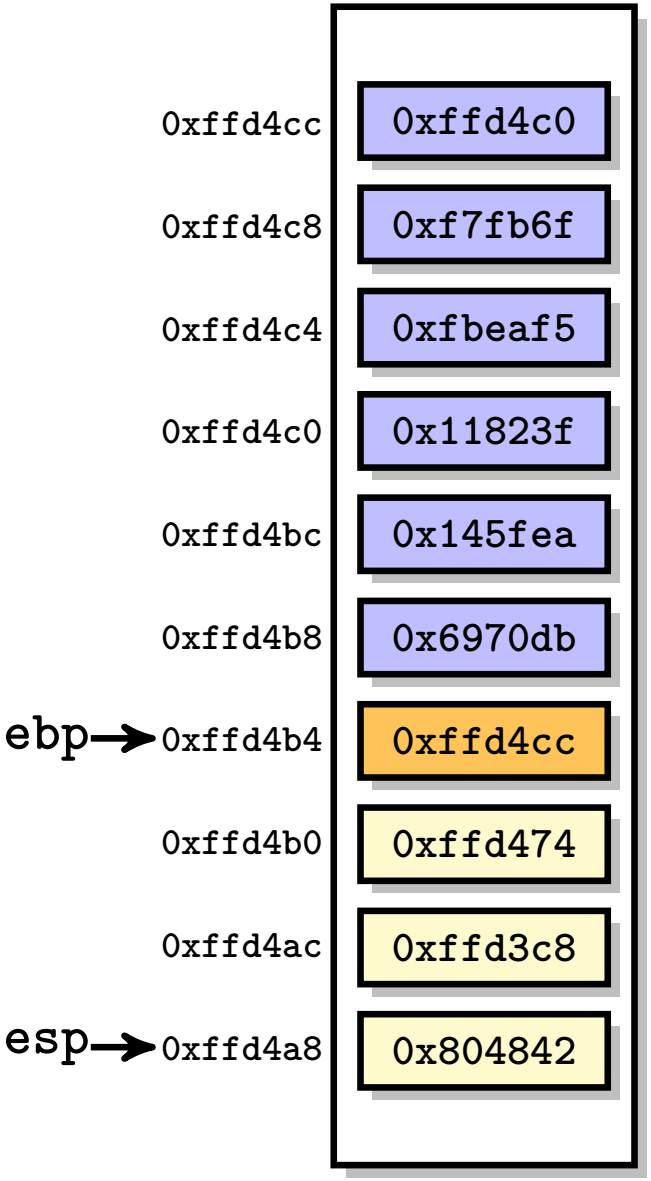


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Memory allocation for local variables.

A Full Example (Entering a function)

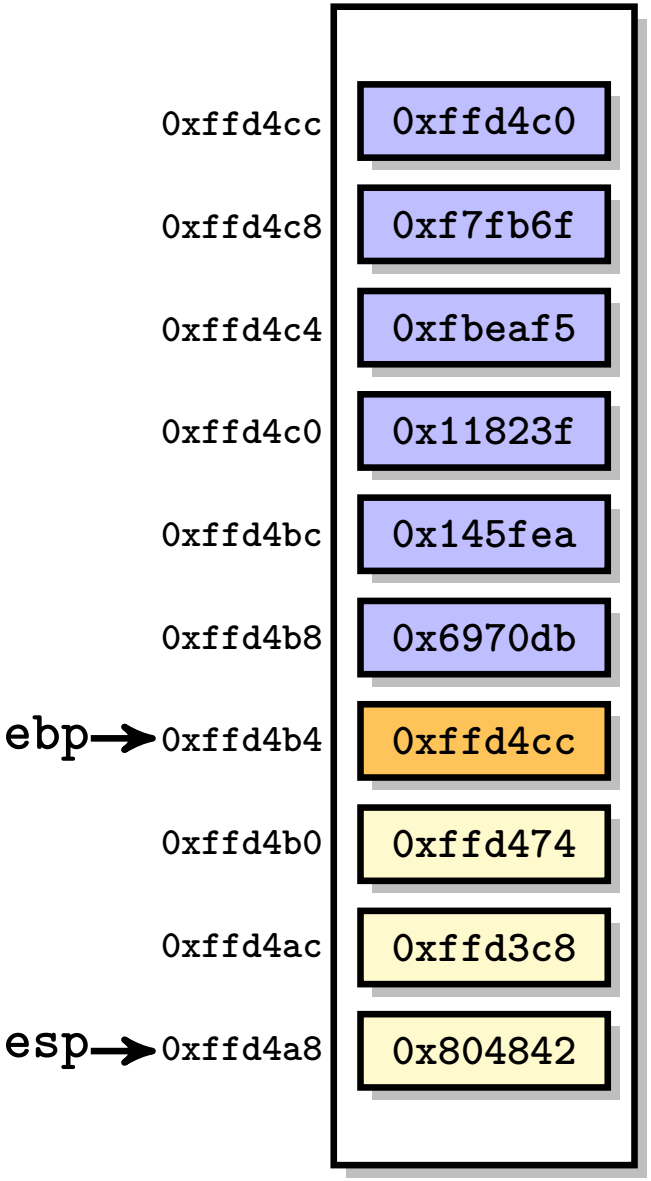


Actions performed:

1. `call addr`
2. `push %ebp`
3. `mov %esp, %ebp`
4. `and $0xfffff0, %esp`
5. `sub $0x8, %esp`

Memory allocation for local variables.

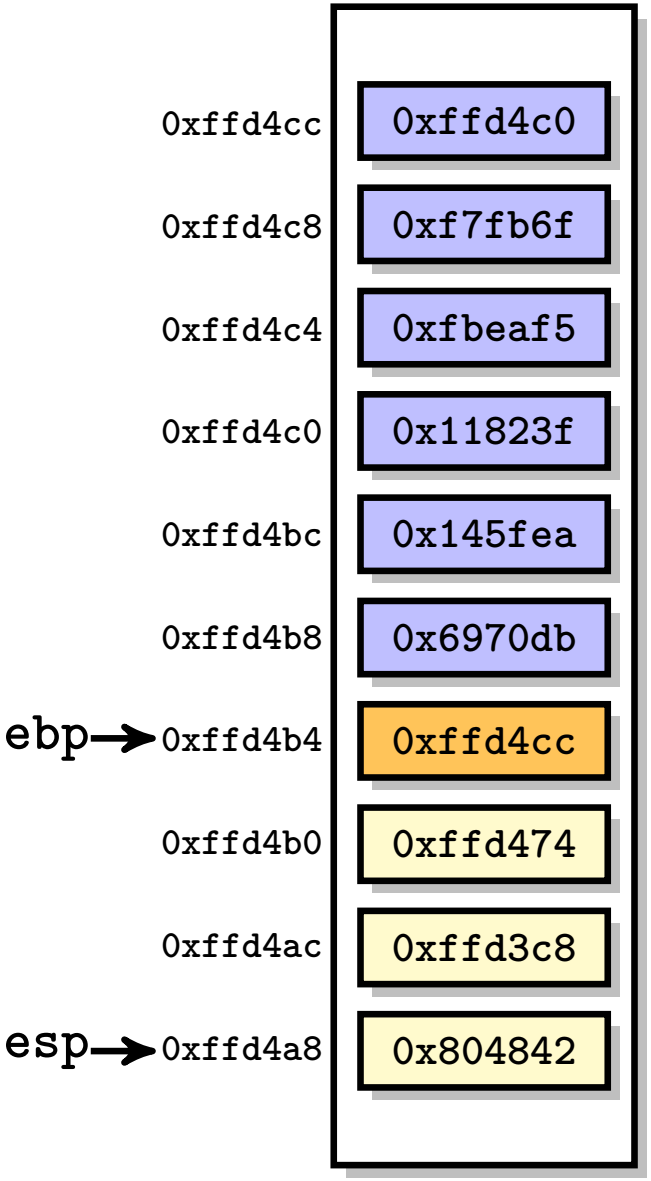
A Full Example (Exiting a function)



Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

A Full Example (Exiting a function)

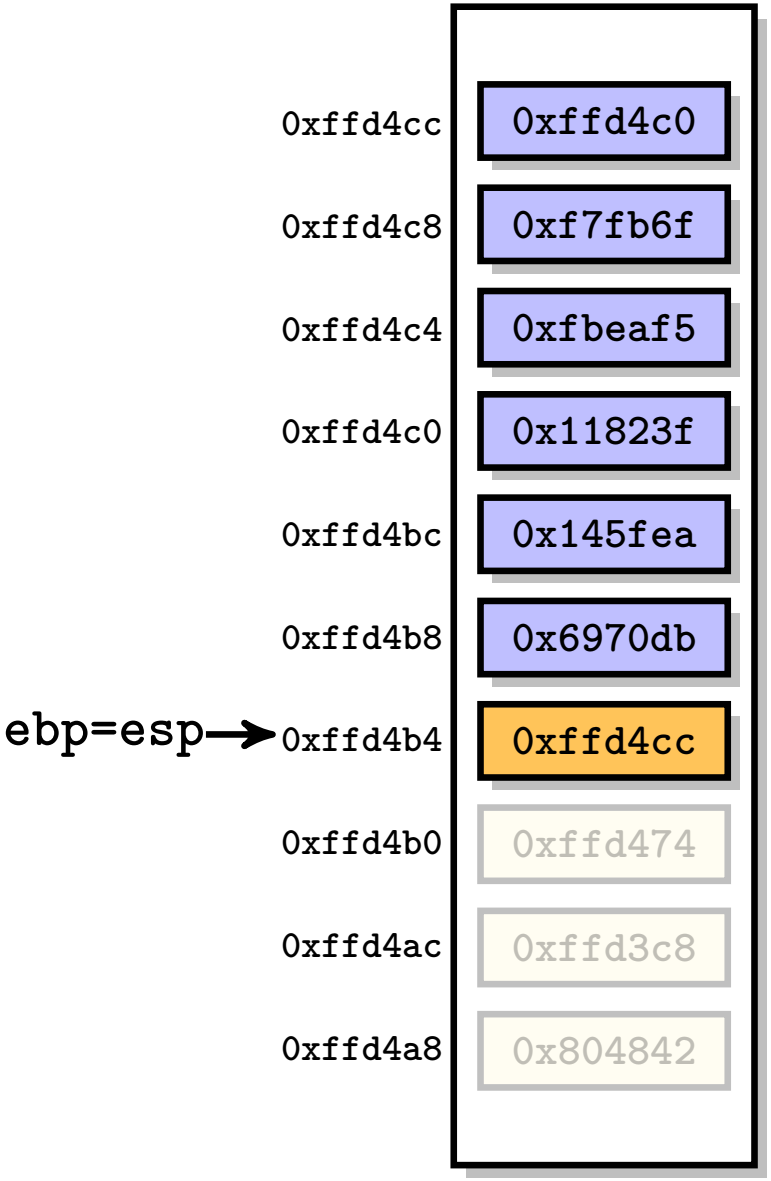


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Cleaning the stack-frame

A Full Example (Exiting a function)

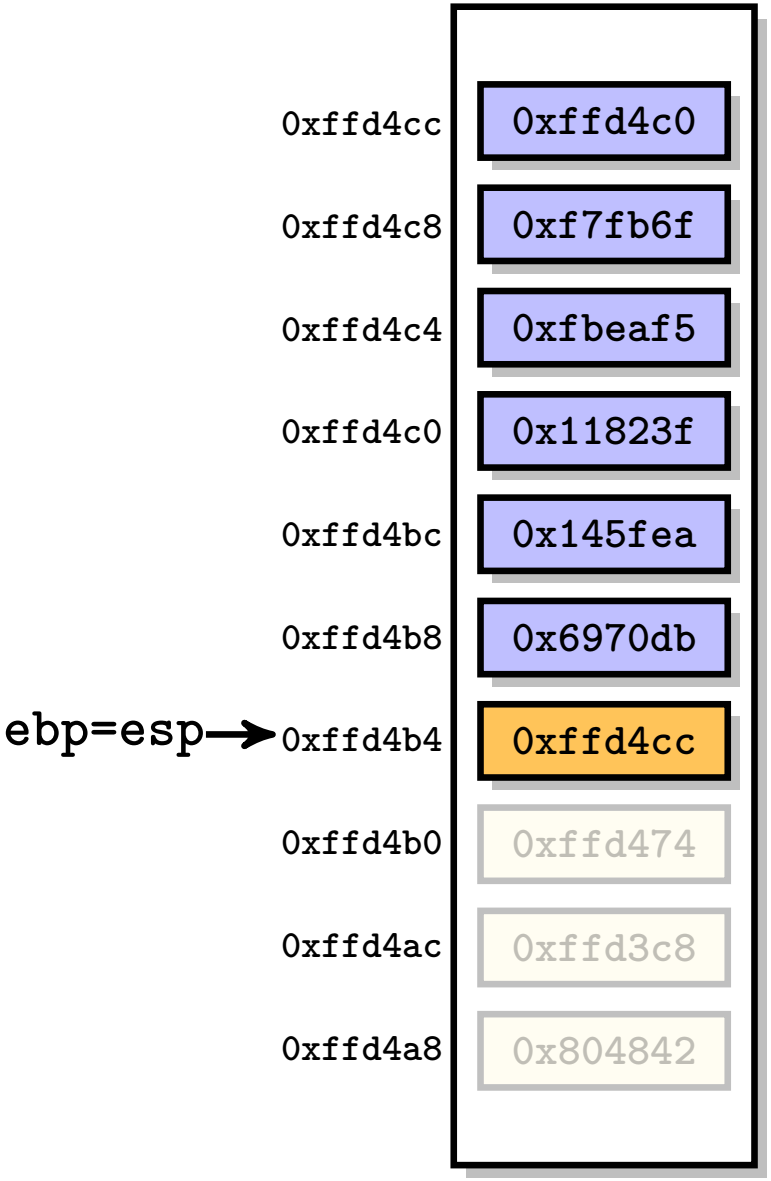


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Cleaning the stack-frame

A Full Example (Exiting a function)

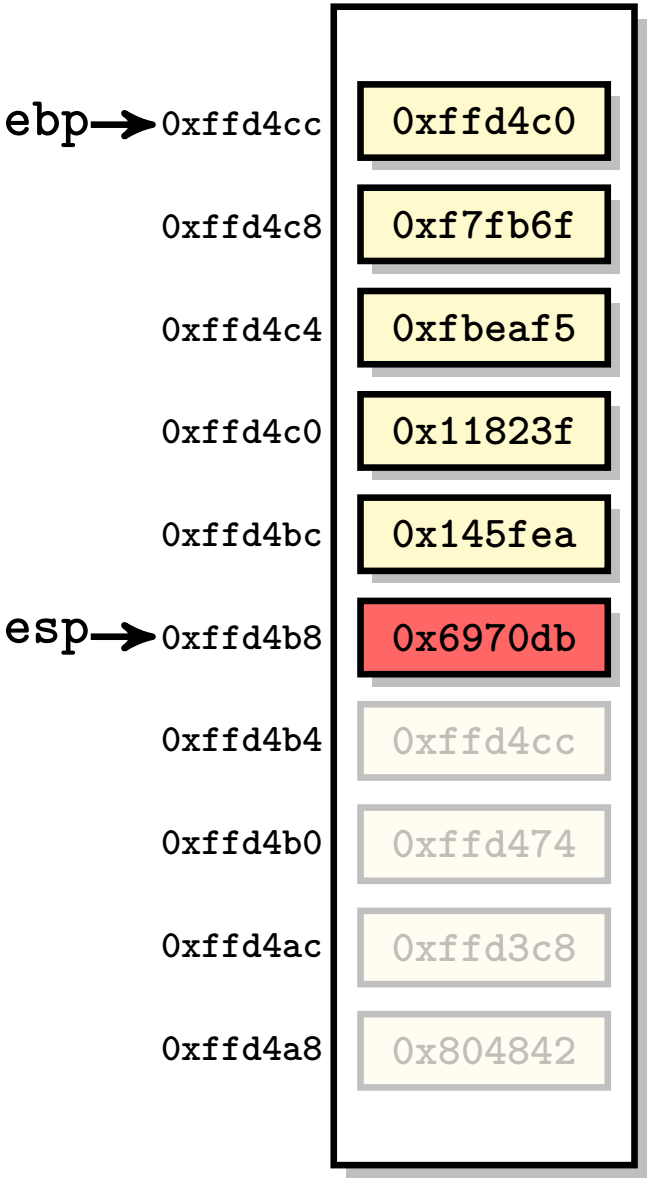


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Restoring **ebp** register

A Full Example (Exiting a function)

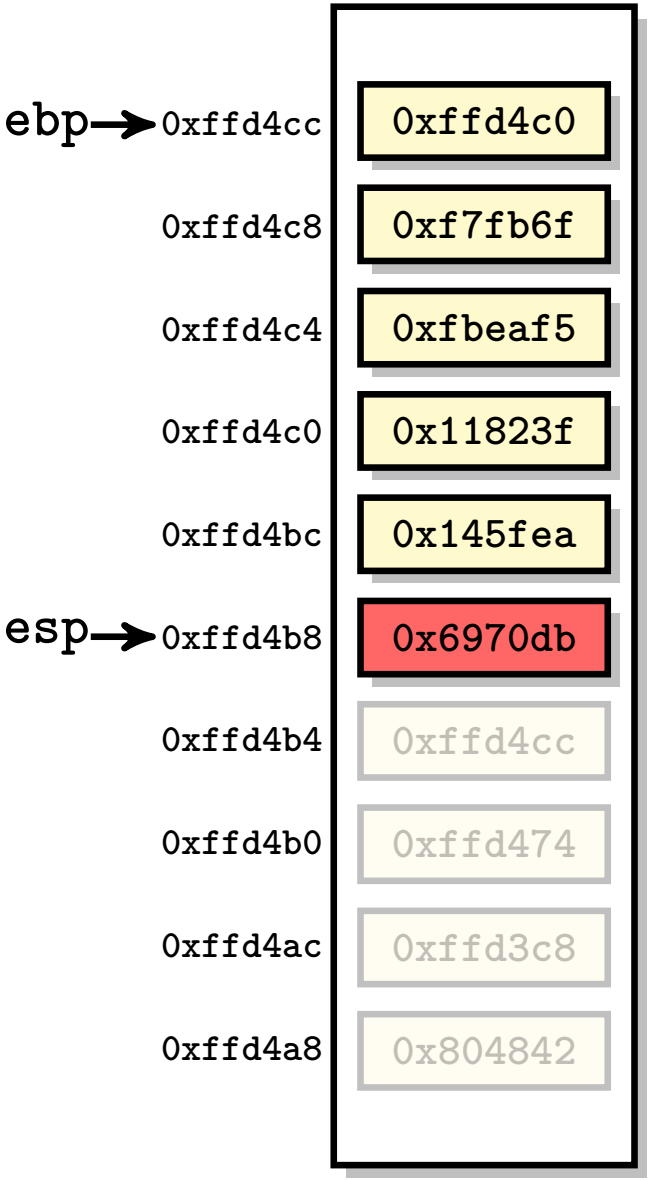


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Restoring **ebp** register

A Full Example (Exiting a function)

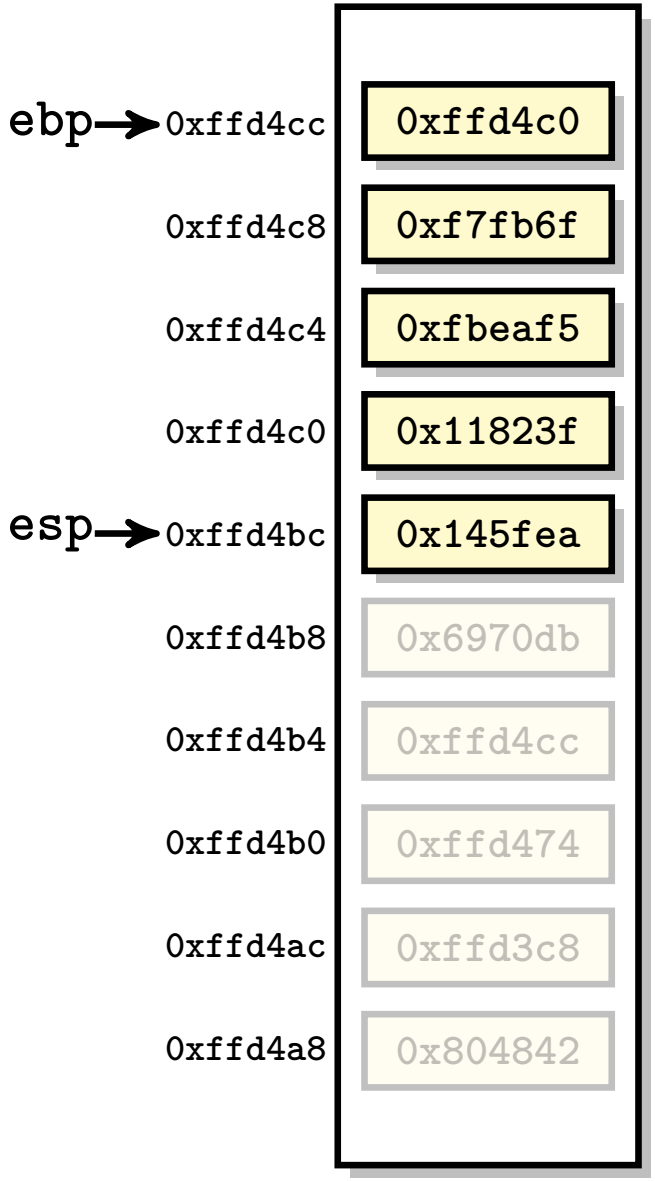


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Restoring `eip` register

A Full Example (Exiting a function)

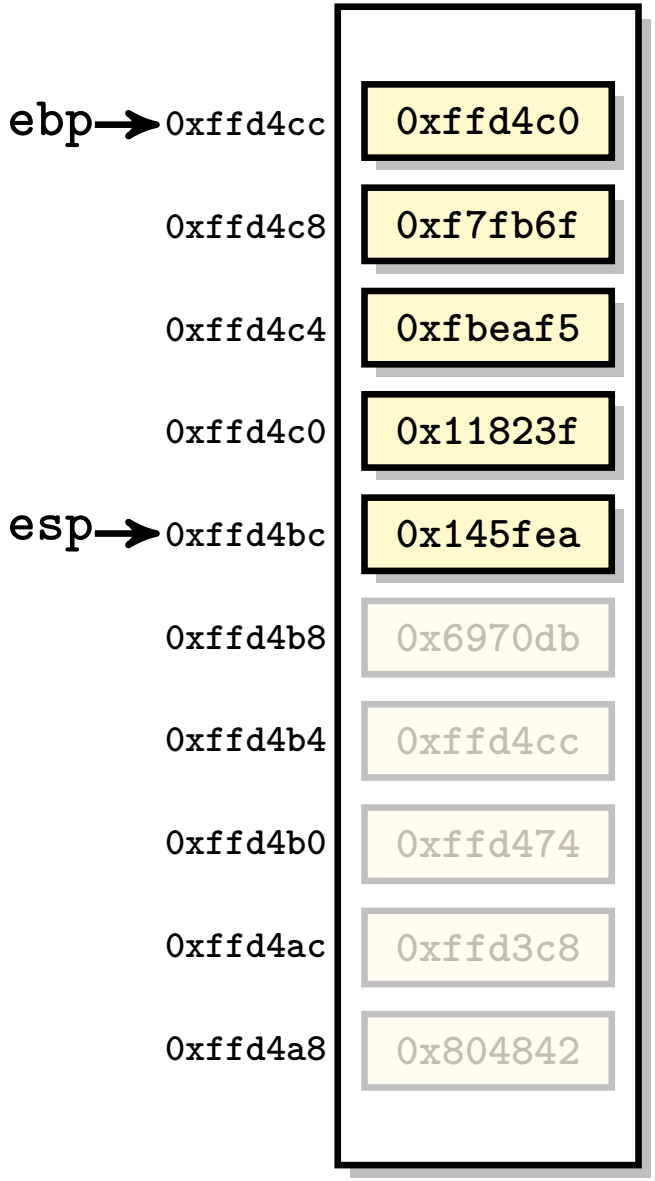


Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Restoring `eip` register

A Full Example (Exiting a function)



Actions performed:

1. `mov %ebp, %esp`
2. `pop %ebp`
3. `ret`

Code Examples (1/2)

```
.glob main

main:
movl  $20, %eax
pushl %eax            # Push in the stack
popl  %ebx            # Pop from the stack
movl  $15, -4(%ebp) # Push in the stack
movl  4(%ebp), %ebx # Pop from the stack
ret
```

Code Examples (2/2)

université de BORDEAUX

```
.glob main

main:
# Prelude
pushl %ebp        # Save base pointer
movl %esp, %ebp # Set stack pointer at base pointer
subl $8, %esp    # Allocate memory space for two words

# Data manipulations
pushl $10 # Push 10 in the stack
pushl $15 # Push 15 in the stack
popl %eax # Pop 15 from the stack
popl %ebx # Pop 10 from the stack

# Epilog
movl %ebp, %esp # Restore previous stack-pointer
popl %ebp        # Restore the old base pointer
ret              # Restore previous execution flow
```

Overview

Application Binary Interface

université de BORDEAUX

- An **ABI** defines a **system interface for compiled application programs**. It is composed of two parts:
  - A generic **high-level description** of the system at an application level;
  - A processor-specific **low-level description** for each processor family.
- The **ABI** provides the conventions to implement various features for each specific processor:
  - **Functions calling conventions** (how to implement function calls);
  - **Return value** (how to pass the return value to the caller);
  - **Stack-frame** (how to manage properly the stack);
  - **Exceptions** (how to implement exceptions).
- **Unix systems** (Linux, BSD, MacOS) usually follow the **System V ABI** with two **x86 processor-specific supplements**:
  - **SystemV i386 ABI** supplement
  - **SystemV amd64 ABI** supplement
- **Microsoft Windows systems** follow the **Microsoft ABI** with two **x86 processor-specific specifications**:
  - **Microsoft x32 ABI** specification
  - **Microsoft x64 ABI** specification

Register Usages

université de BORDEAUX

Volatile/Non-volatile Registers

A register is said **volatile** if it can be overwritten by the callee with no harm for the caller.

On the contrary, a register is said **non-volatile** if the content of the register may be used by the caller. They must be **saved before use** and **restored before leaving** the callee.

Specific usage of pointers

- `eax`: Contains the **integer return code** if any;
- `st(0)`: Contains the **floating-point return code** if any;
- `esi`, `edi`: **Non-volatile registers** (callee must preserve these registers).
- `ecx`, `edx`: **Volatile registers** (callee can use freely these registers).
- `ebx`: Global offset table base register for position-independent code. For absolute code, it serves as a local register and has no specified role in the function calling sequence. **Non-volatile registers** (callee must preserve these registers).

Function Calls (A Few Vocabulary)

université de BORDEAUX

- **Function call**: `foo()` is a function calling the function `bar()`:
  - `foo()` is the **caller** function;
  - `bar()` is the **callee** function.

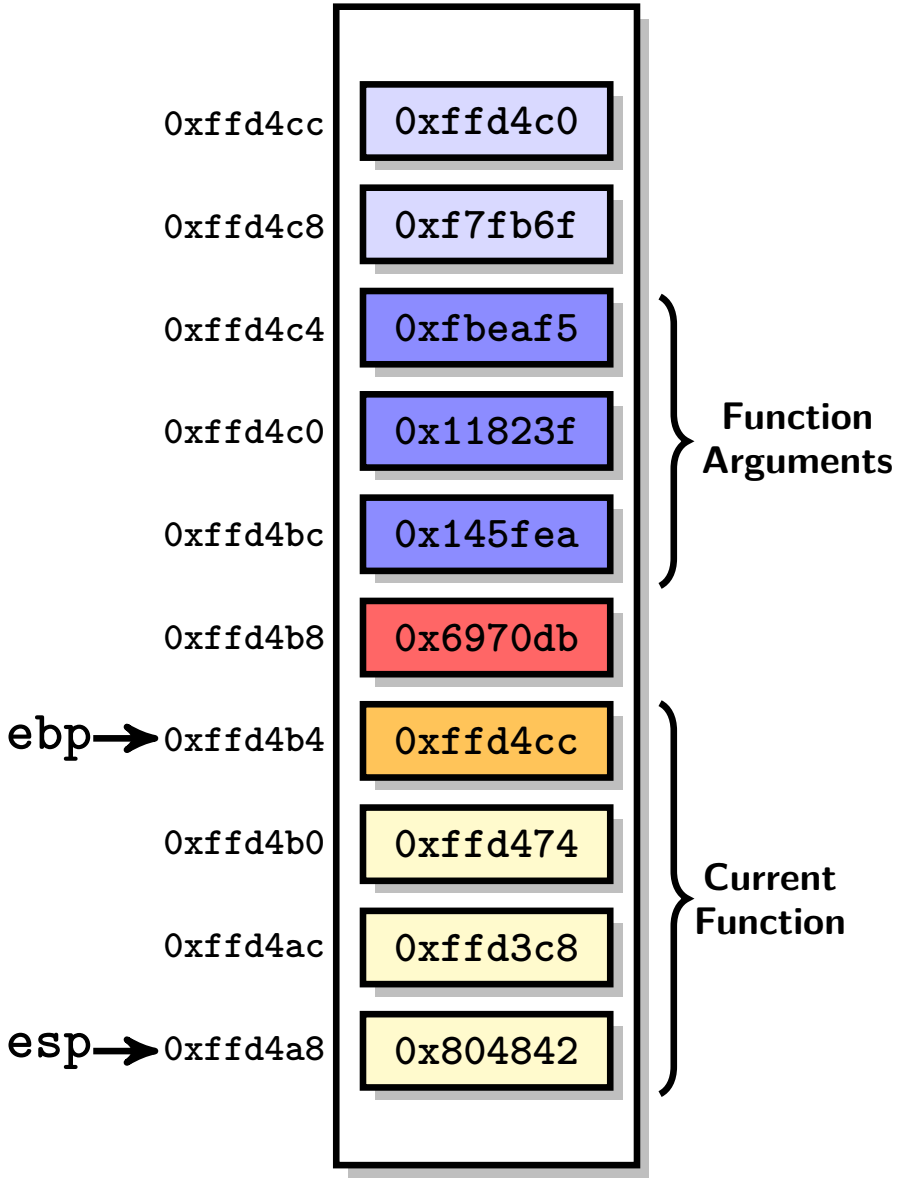
- **Local variable**: A variable whose scope is not getting outside of the function (also called **automatic variable**).

- **Parameters**: Data set by the caller function for the callee before start (also called **arguments**).

- **Return code**: Data set by the callee for the caller at the end of execution of the callee.

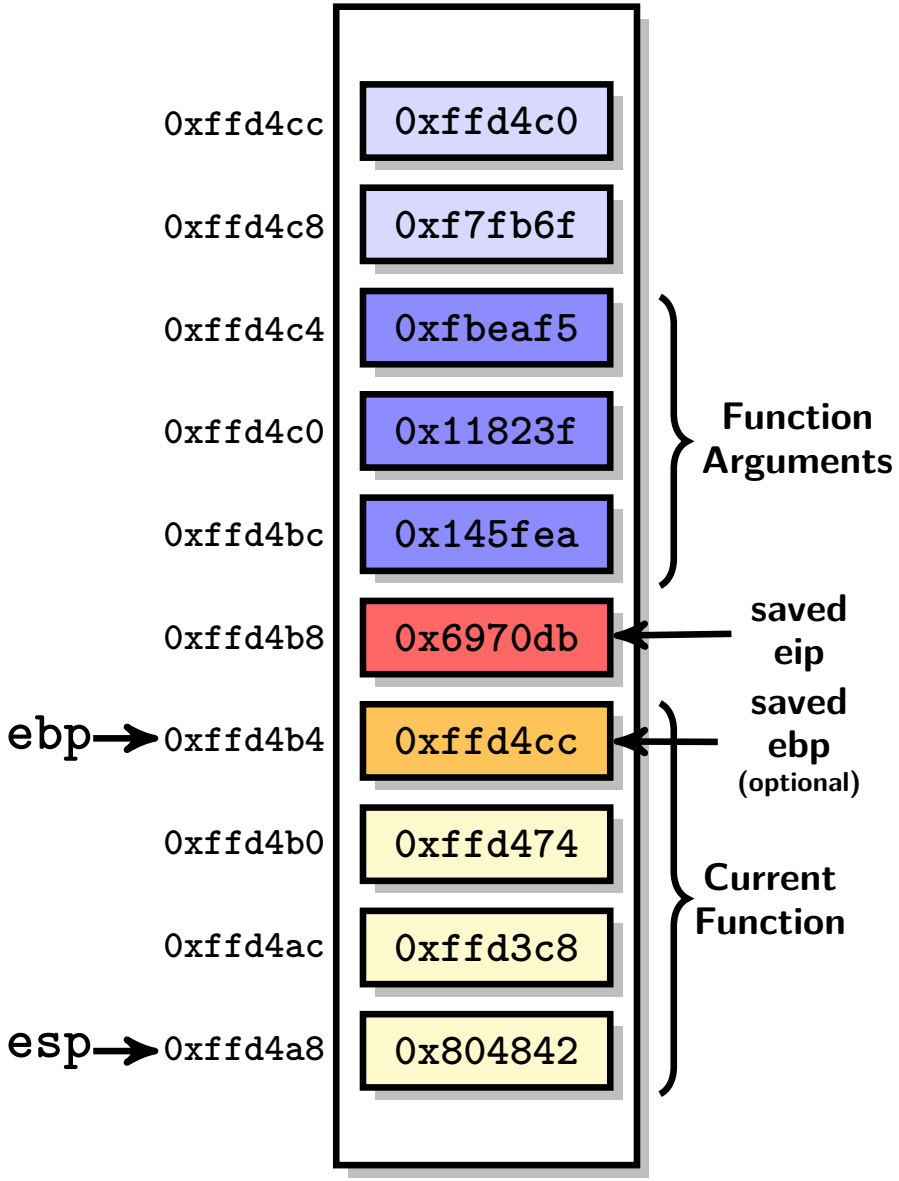
- **Call stack**: The chain of functions that have been currently called (*e.g.* `main()` $\rightarrow$ `foo()` $\rightarrow$ `bar()`).

Function Calling Conventions



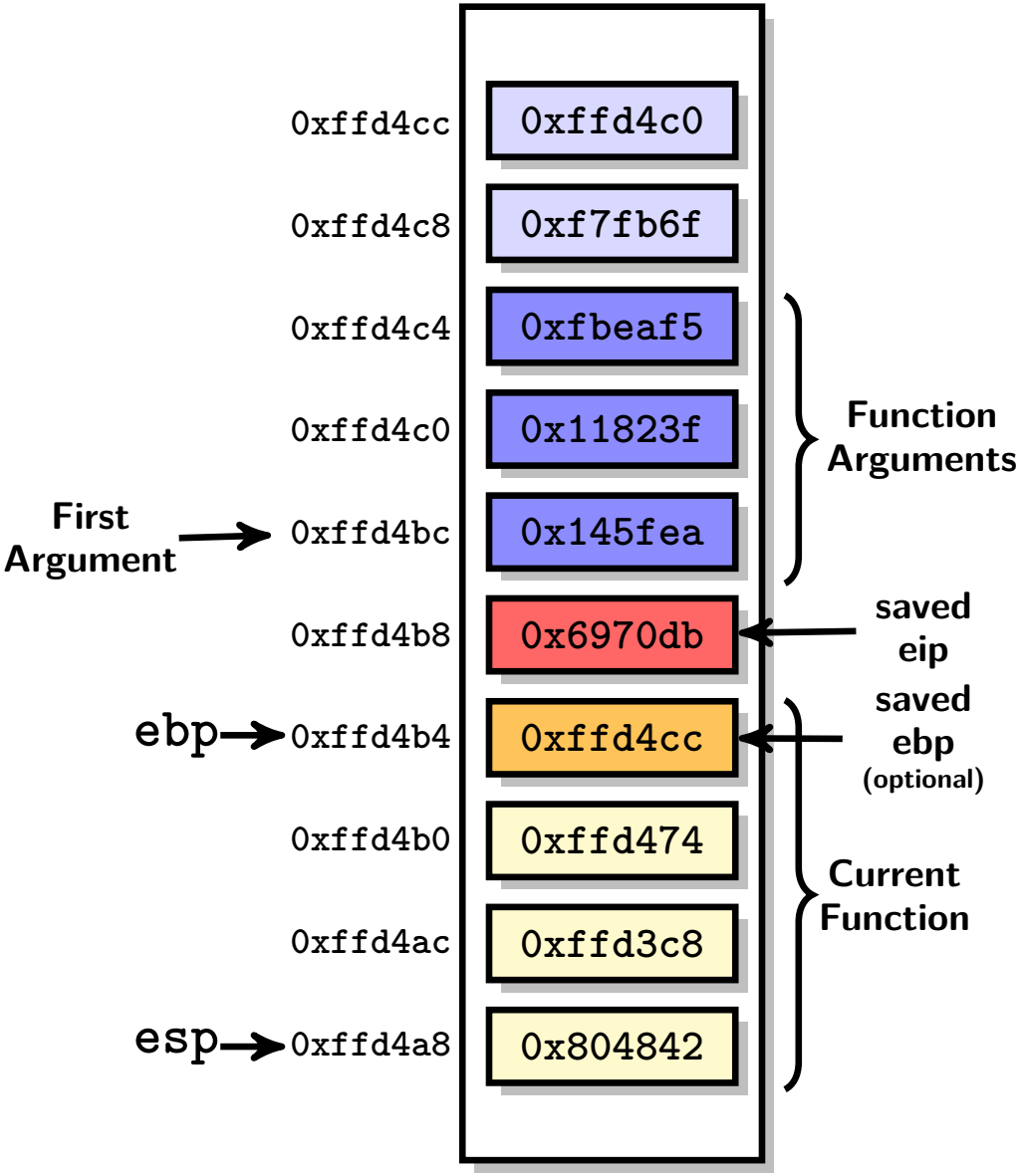
- There are several possible calling conventions for SystemV i386 ABI: `cdecl`, `stdcall`, `fastcall`, ... `cdecl` is mostly used and, anyway, **all arguments go through the stack**.
- Argument words are pushed onto the stack in reverse order;
- Arguments are referred through: `8(%ebp)` (first), `12(%ebp)` (second), ..., `4n+8(%ebp)` (n-*th*).
- Argument's size can be **more than a word**. To make it a multiple of a word, **tail padding** is used.
- In fact, the 'saved ebp' from the previous stack-frame is **optional** ('`-fomit-frame-pointer`' gcc option).

Function Calling Conventions



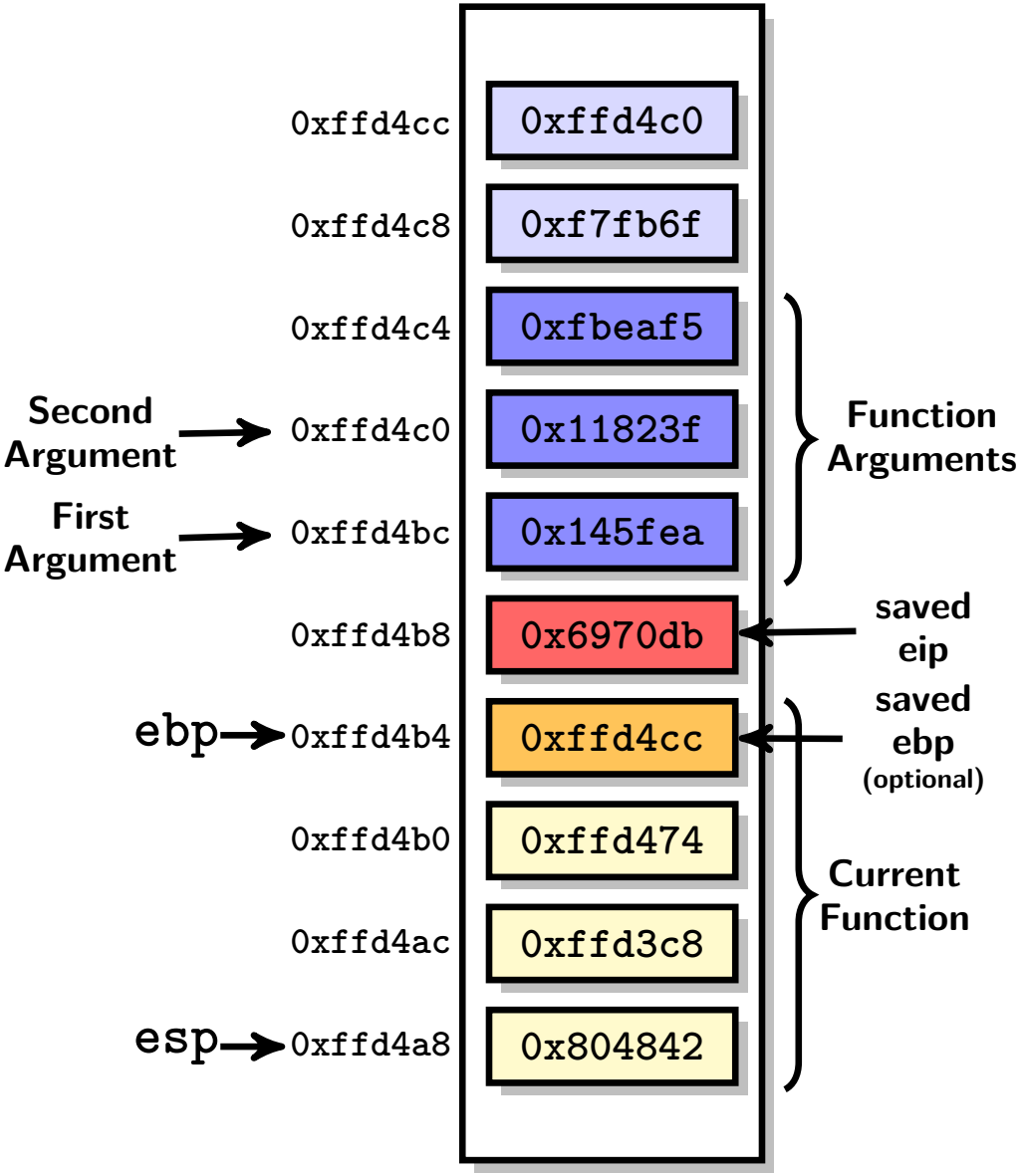
- There are several possible calling conventions for SystemV i386 ABI: `cdecl`, `stdcall`, `fastcall`, ... `cdecl` is mostly used and, anyway, **all arguments go through the stack**.
- Argument words are pushed onto the stack in reverse order;
- Arguments are referred through: `8(%ebp)` (first), `12(%ebp)` (second), ..., `4n+8(%ebp)` (n-*th*).
- Argument's size can be **more than a word**. To make it a multiple of a word, **tail padding** is used.
- In fact, the 'saved ebp' from the previous stack-frame is **optional** ('`-fomit-frame-pointer`' gcc option).

Function Calling Conventions



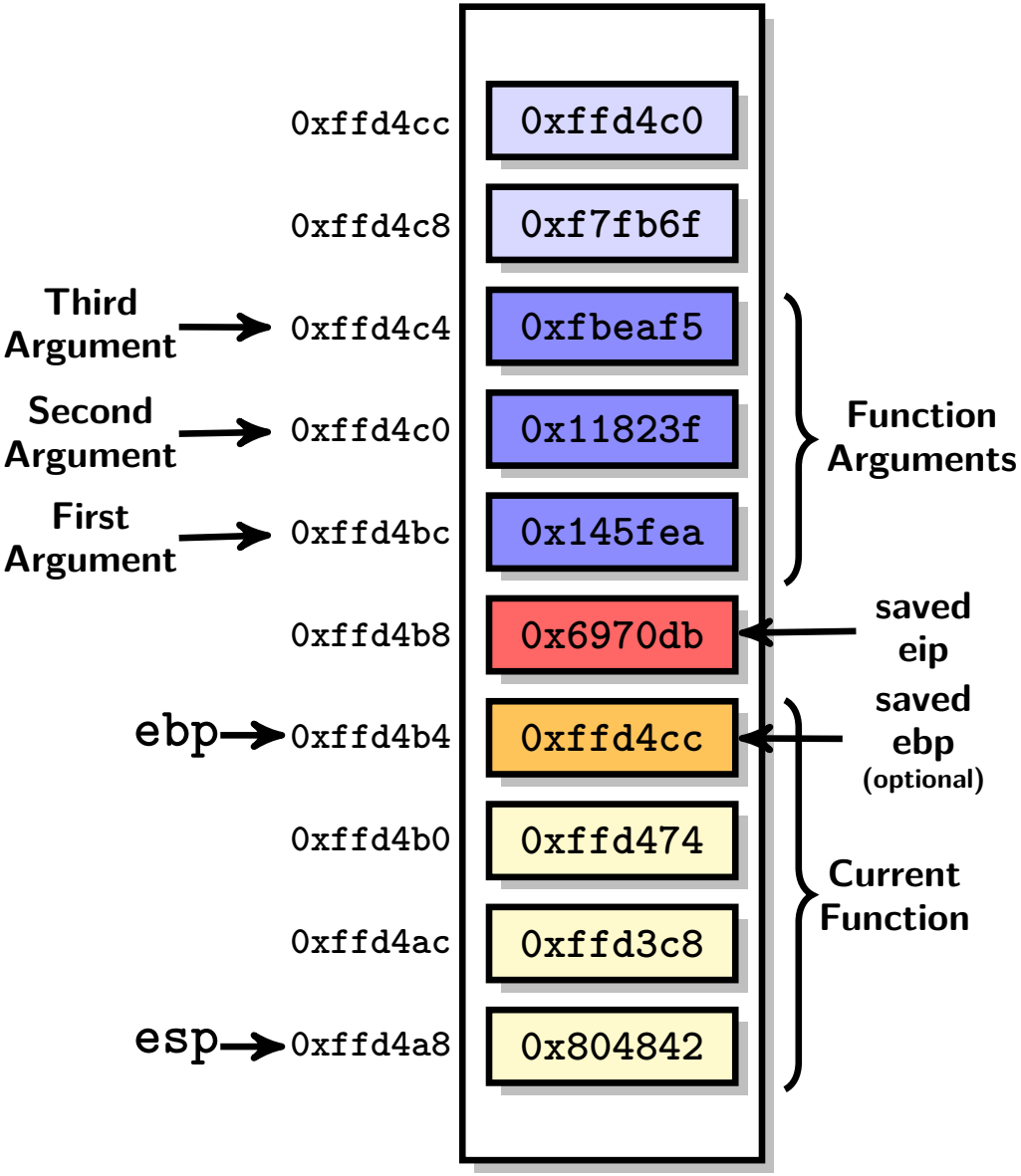
- There are several possible calling conventions for SystemV i386 ABI: `cdecl`, `stdcall`, `fastcall`, ... `cdecl` is mostly used and, anyway, **all arguments go through the stack**.
- Argument words are pushed onto the stack in reverse order;
- Arguments are referred through: `8(%ebp)` (first), `12(%ebp)` (second), ..., `4n+8(%ebp)` (n-*th*).
- Argument's size can be **more than a word**. To make it a multiple of a word, **tail padding** is used.
- In fact, the 'saved ebp' from the previous stack-frame is **optional** ('`-fomit-frame-pointer`' gcc option).

Function Calling Conventions



- There are several possible calling conventions for SystemV i386 ABI: `cdecl`, `stdcall`, `fastcall`, ... `cdecl` is mostly used and, anyway, **all arguments go through the stack**.
- Argument words are pushed onto the stack in reverse order;
- Arguments are referred through: `8(%ebp)` (first), `12(%ebp)` (second), ..., `4n+8(%ebp)` (n-*th*).
- Argument's size can be **more than a word**. To make it a multiple of a word, **tail padding** is used.
- In fact, the 'saved ebp' from the previous stack-frame is **optional** ('`-fomit-frame-pointer`' gcc option).

Function Calling Conventions



- There are several possible calling conventions for SystemV i386 ABI: `cdecl`, `stdcall`, `fastcall`, ... `cdecl` is mostly used and, anyway, **all arguments go through the stack**.
- Argument words are pushed onto the stack in reverse order;
- Arguments are referred through: `8(%ebp)` (first), `12(%ebp)` (second), ..., `4n+8(%ebp)` (n-*th*).
- Argument's size can be **more than a word**. To make it a multiple of a word, **tail padding** is used.
- In fact, the 'saved ebp' from the previous stack-frame is **optional** ('`-fomit-frame-pointer`' gcc option).

Function Calling Conventions (Examples)

université de BORDEAUX

Integral and Pointer Arguments

```
int foo(int a, int b, int* c, int &d)

%eax foo(8(%ebp), 12(%ebp), 16(%ebp), 20(%ebp))
```

Floating-point Arguments

```
float bar(float a, int b, float c)

%st(0) bar(8(%ebp), 16(%ebp), 20(%ebp))
```

Struct Arguments

```
int fuz(int a, struct b, struct c)

%eax fuz(8(%ebp), 12(%ebp), 20(%ebp))
```

Struct/Object Arguments

université de BORDEAUX

- `struct mystruct foo(int a, float b)`
- When a function return a structure, the caller is in charge to provide the memory space of the struct.
- First argument of such function is always the memory location of the struct memory space.
- The callee sets `%eax` to the value of the original address of the caller's area before it returns.
- The callee must remove this address from the stack before returning.

Function Calling Conventions & Registers

université de BORDEAUX

Two calling conventions for x86-64 (both inspired by `fastcall`):

- **Microsoft x64 calling convention** (Windows);
- **SystemV AMD64 calling convention** (Linux, BSD, MacOS).

Calling convention through registers

- **6 registers** for integer arguments: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`;
- **8 registers** for float/double arguments: `xmm0`–`xmm7`;
- First available register for the parameter type is used;
- No overlap, so you could have 14 parameters stored in registers;
- `struct` parameters are splitted between registers;
- **Everything else goes on the stack**;
- `rax` holds number of vector registers (`xmmX`).

(Non)-Volatile Registers

- **Volatile registers**: `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`–`r11`, `xmm0`–`xmm15`, `st0`–`st7`;
- **Non-volatile registers**: `rbx`, `rbp`, `rsp`, `r12`–`r15`.

Function Calling Conventions (Examples)

université de BORDEAUX

Integral and Pointer Arguments

```
int func1(int a, float b, int c)

rax func1(rdi, xmm0, rsi)
```

Floating-point Arguments (1)

```
float func2(float a, int b, float c)

xmm0 func2(xmm0, rdi, xmm1)
```

Floating-point Arguments (2)

```
float func3(float a, int b, int c)

xmm0 func3(xmm0, rdi, rsi)
```

Function Calling Conventions

```
typedef struct {
  int a, b;
  double d;
} structparm;

structparm s;

int e,f,g,h,i,j,k;
long double ld;
double m, n;
__m256 y;

extern void
func (int e, int f, structparm s, int g, int h,
      long double ld, double m, __m256 y, double n,
      int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);
```

%rdi:e	%xmm0:s.d	(%rbp):ld
%rsi:f	%xmm1:m	16(%rbp):j
%rdx:s.a,s.b	%ymm2:y	24(%rbp):k
%rcx:g	%xmm3:n	
%r8:h		
%r9:i		

Overview

References I

Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, September 2010.
Version 0.99.5.

Santa Cruz Operation, Inc.
System V Application Binary Interface: i386 Architecture Processor Supplement, fourth edition, March 1997.

Executable Files