Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 6
Memory in MIPS

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Objectives

- The need for **memory diagrams** and how to draw them
- How the system stack works and the role played by **$sp** and **$fp**
- How (and why) **local variables** are stored on the stack and how to access them
- How to use **addressing modes** to access variables

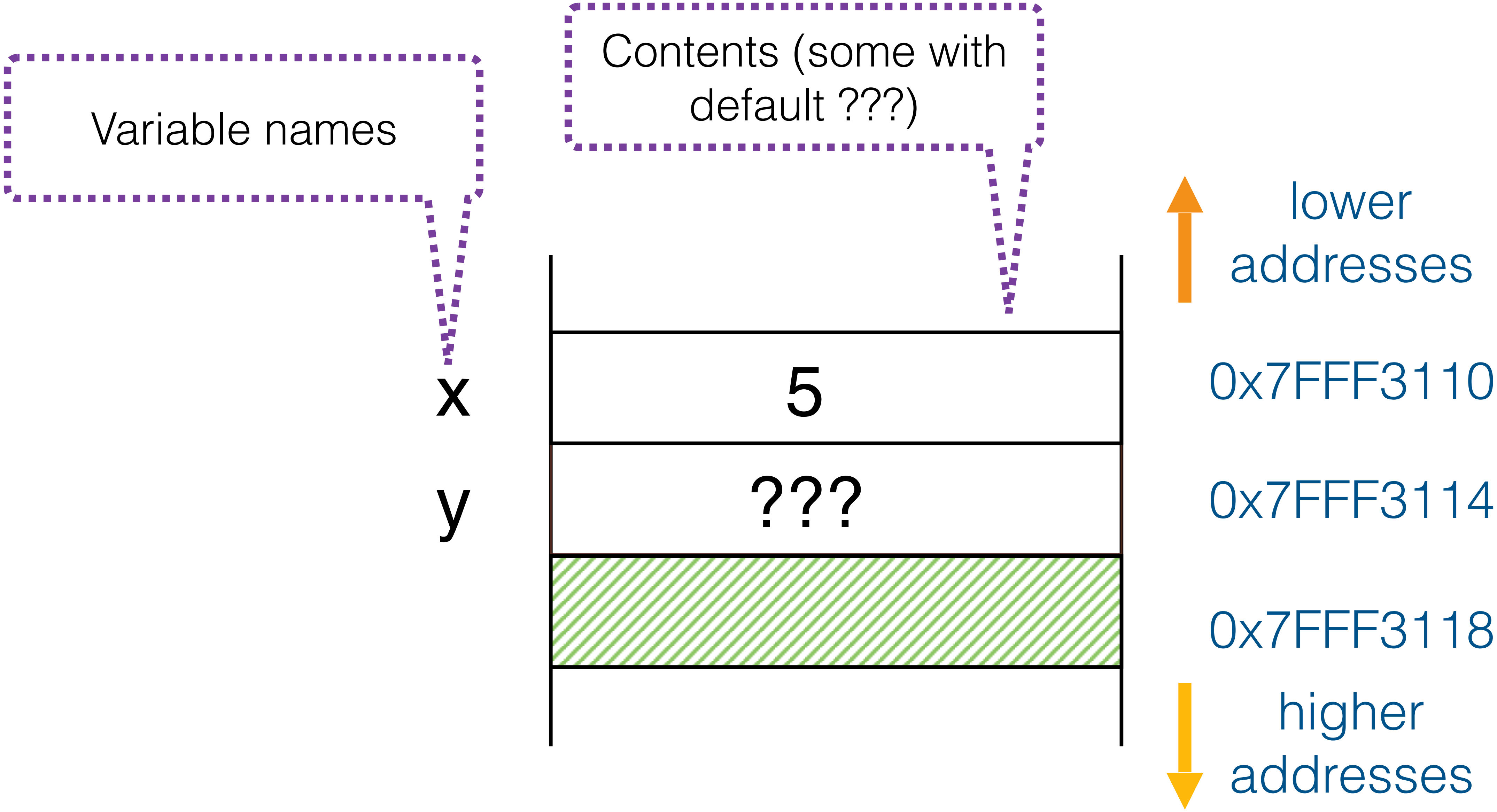
What we have seen

- How to define and use global variables
- How to allocate memory on the Heap.
- How to use memory on the Heap.

Memory diagrams

- Useful **for humans** to know how to access variables
- Show memory allocated to variables:
  - ➡ Addresses
  - ➡ Contents
  - ➡ Variable names

Recall: we assume numbers appear directly at the memory location (not true in Python, but true in C or Java) and occupy 4 bytes.



Variable names

Contents (some with default ???)

lower addresses

x 5 0x7FFF3110

y ??? 0x7FFF3114

0x7FFF3118

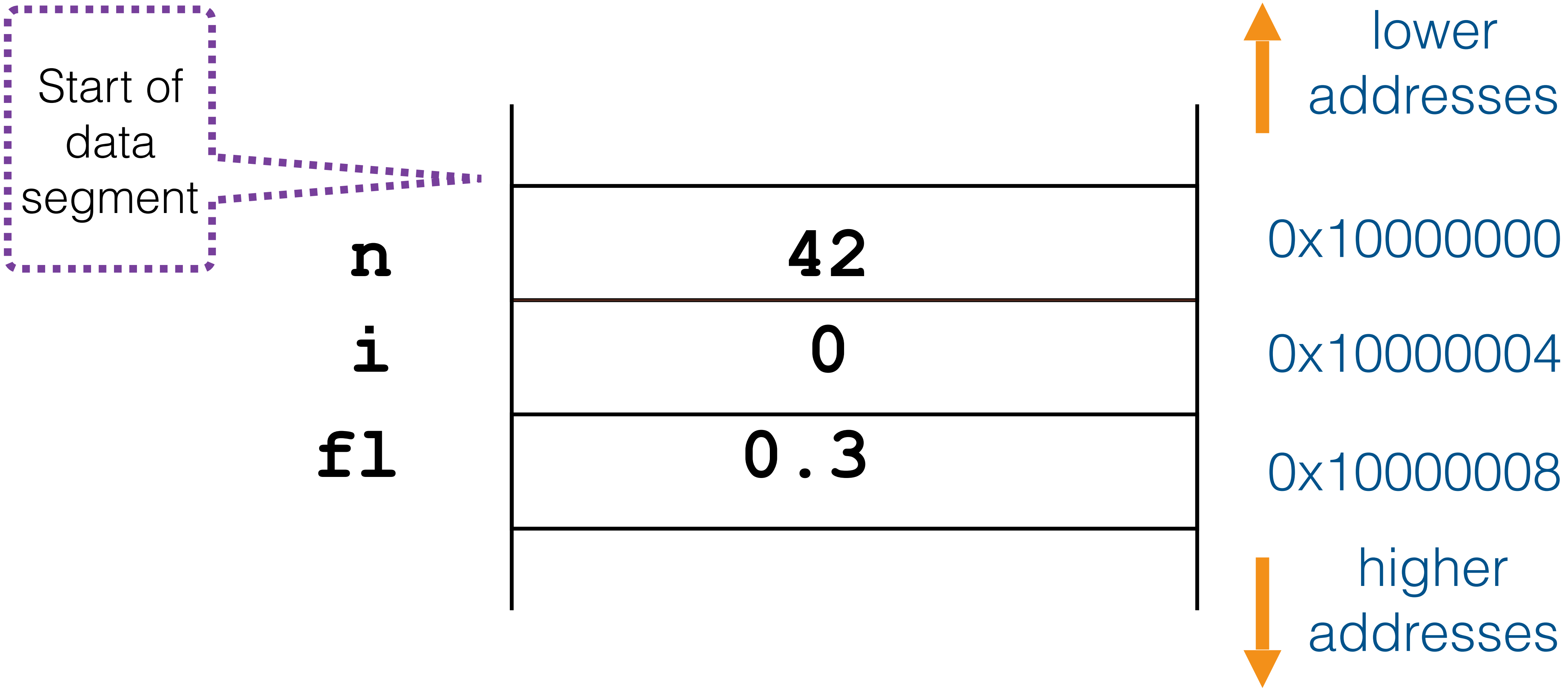
higher addresses

When variables contain addresses of other variables, helpful to draw arrow (pointer)

Global variables

- Memory map **not** crucial for **global variables** (stored in **data segment**)

- Global variables: every variable has a **label** to identify it



Start of data segment

lower addresses

n	**42**	0x10000000
i	**0**	0x10000004
fl	**0.3**	0x10000008

higher addresses

Local variables

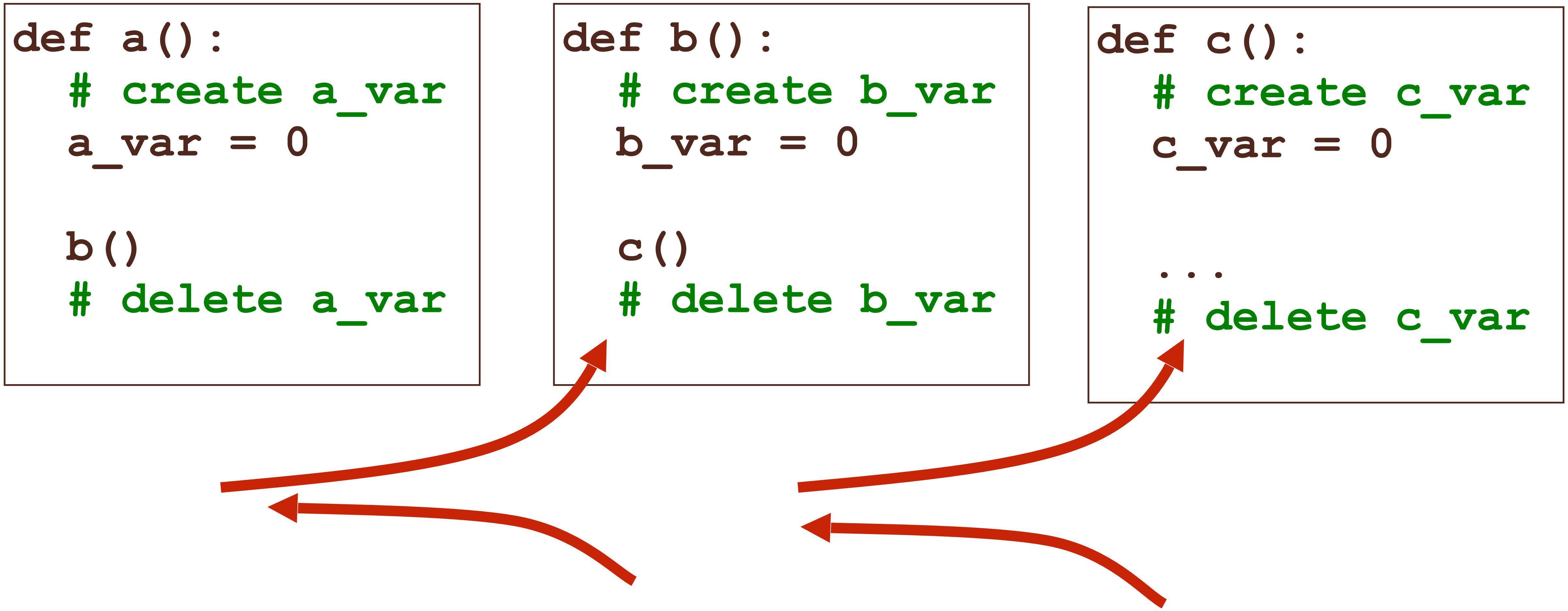
- Why not store local variables in the data segment? Local variables do not have labels.

- **Properties of Data segment**
  - ➡ Accessible from all parts of the program
  - ➡ Labels must be **unique**
  - ➡ Each location can hold only one discrete value

- **Properties of Local Variables**
  - ➡ Accessible only within a function.
  - ➡ Several variables with same name (different scopes) within the same function
  - ➡ May have more than one version of the same function's variables existing (due to recursion)

Properties of local variables

- Must be created/allocated at function entry
- Must be destroyed/deallocated at function exit
- Other functions may be called in between, with the same rules



```
def a():
  # create a_var
  a_var = 0

  b()
  # delete a_var
```

```
def b():
  # create b_var
  b_var = 0

  c()
  # delete b_var
```

```
def c():
  # create c_var
  c_var = 0

  ...
  # delete c_var
```

Properties of local variables

- Allocation/deallocation is LIFO:
  Last allocated, first deallocated

- A stack data structure is ideal for storing local variables
  - ➡ **Allocate** = **push**
  - ➡ **Deallocate** = **pop**

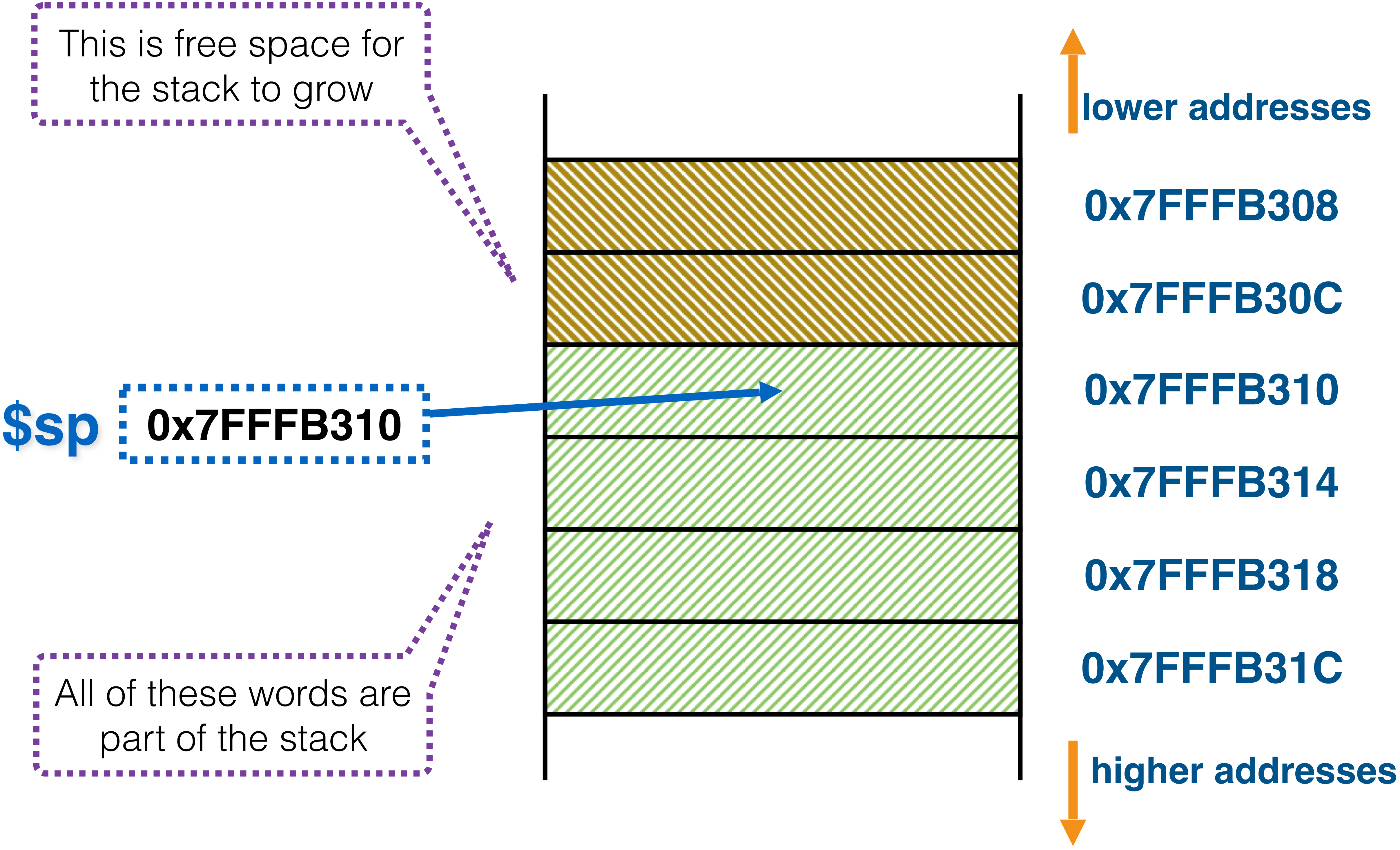
- Most computers provide a memory stack for programs to use (initialised by OS): **system stack** or **runtime stack** or **process stack**

- The instruction set provides operations for pushing/popping off the system stack.

System Segment

- Register **$sp** (stack pointer) indicates the top of stack
  - ➡ Contains the address of the word of memory at the top of stack (i.e., with lowest address)
  - ➡ Its value changes during the execution of a function
- How do we push and pop variables?

System stack



This is free space for the stack to grow

lower addresses

0x7FFFB308

0x7FFFB30C

$sp 0x7FFFB310

0x7FFFB310

0x7FFFB314

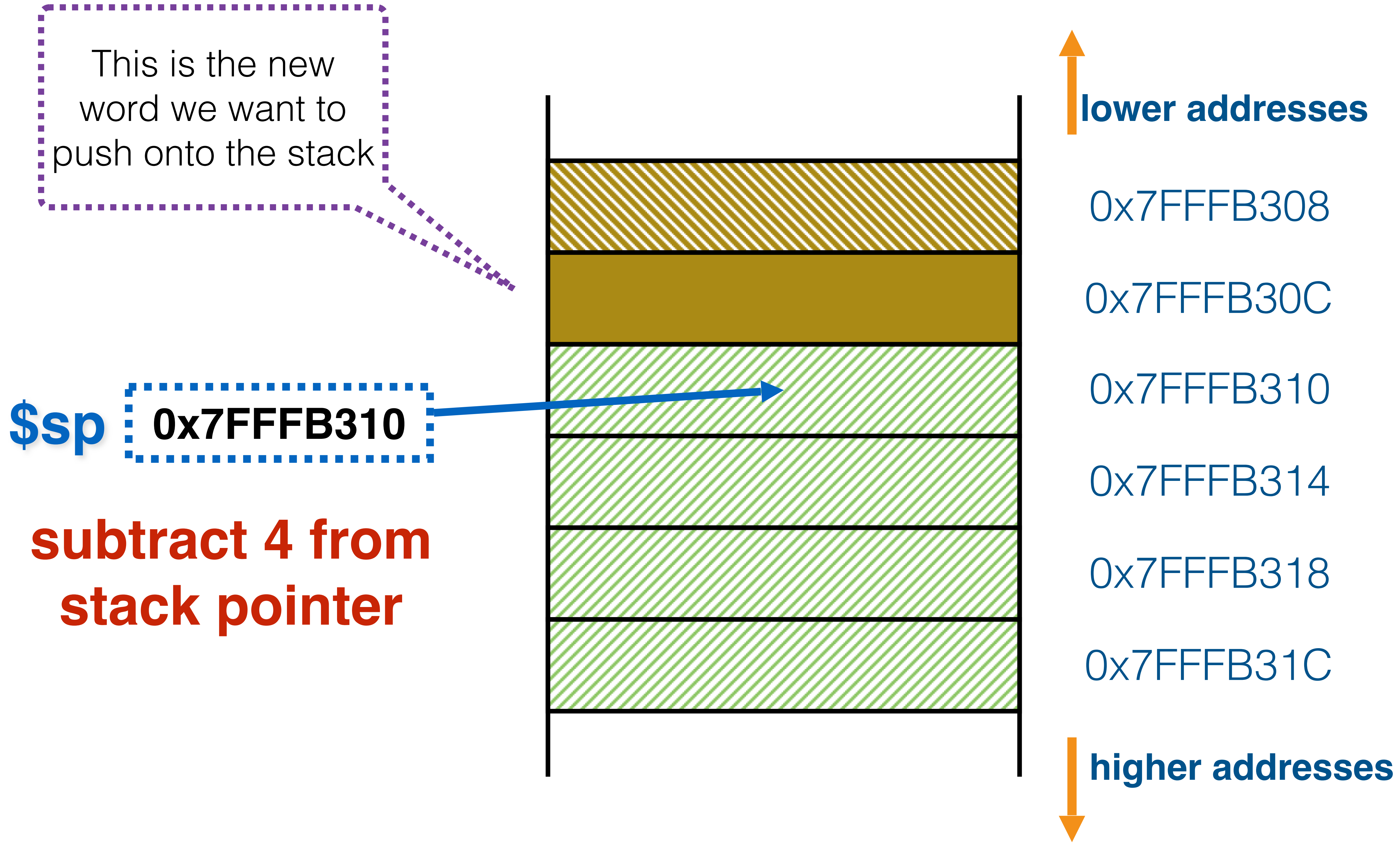
0x7FFFB318

0x7FFFB31C

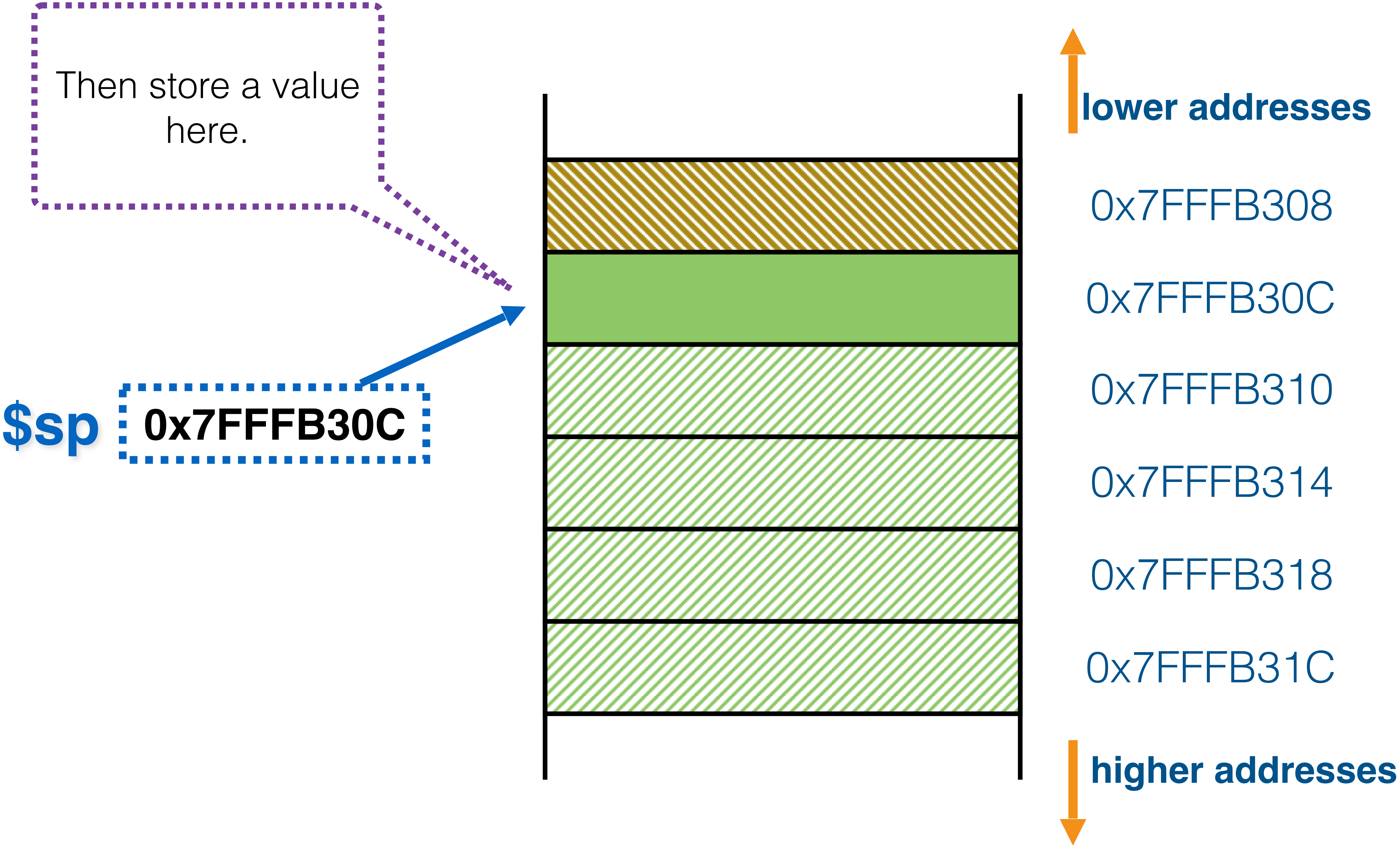
All of these words are part of the stack

higher addresses

System stack: **pushing**



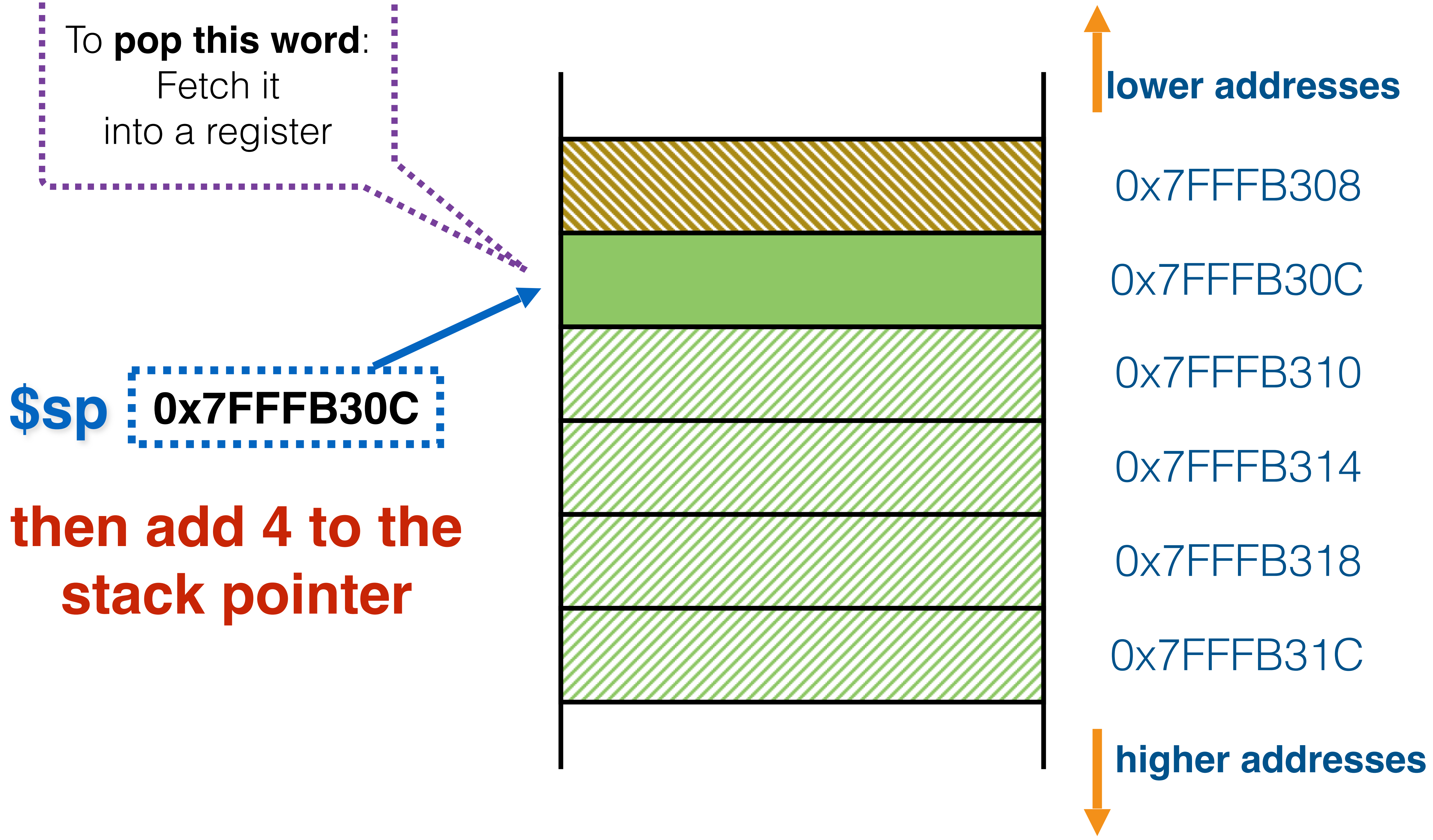
System stack: **pushing**



Then store a value here.

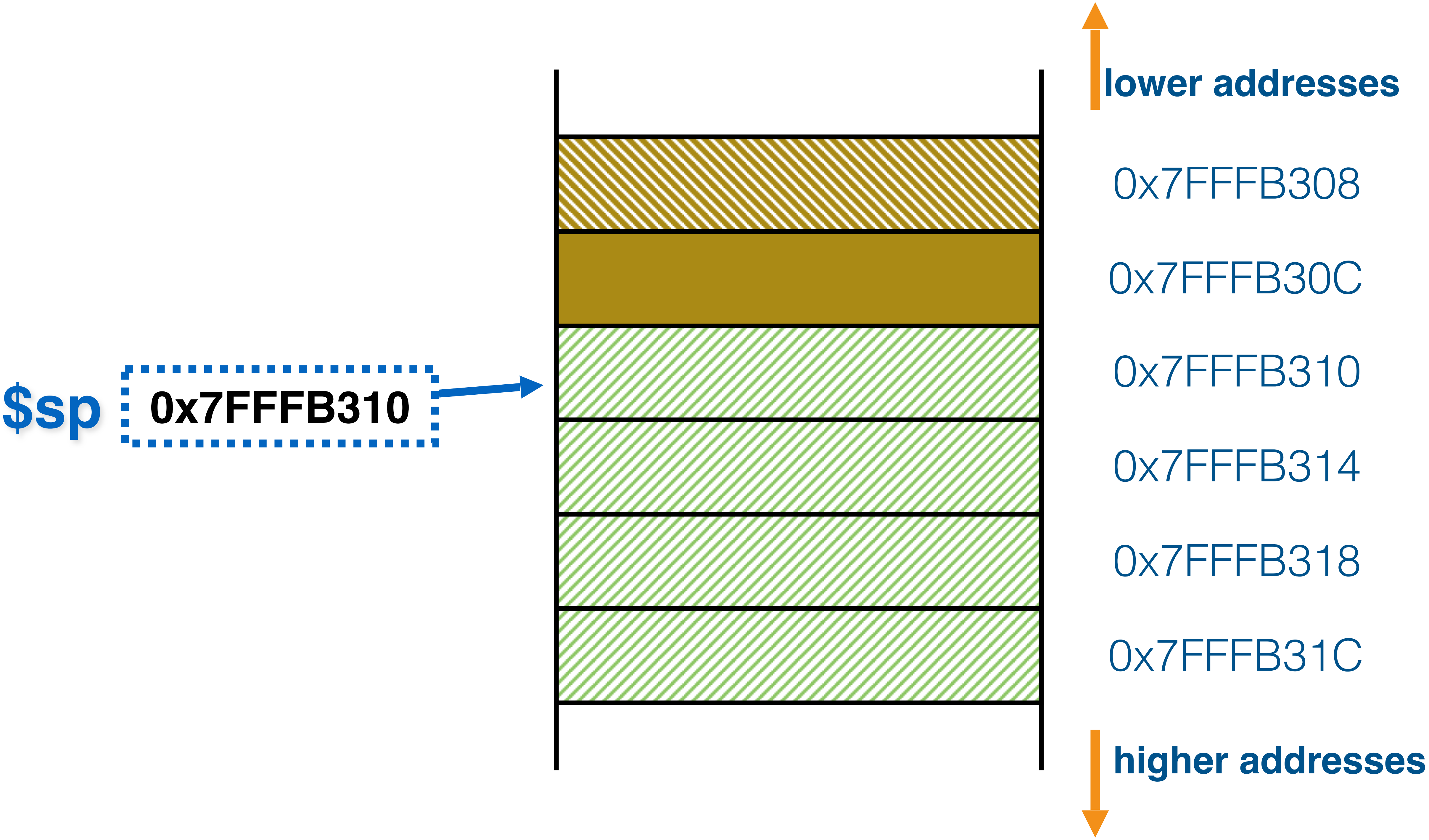
System stack: **popping**

To **pop this word**:
Fetch it
into a register



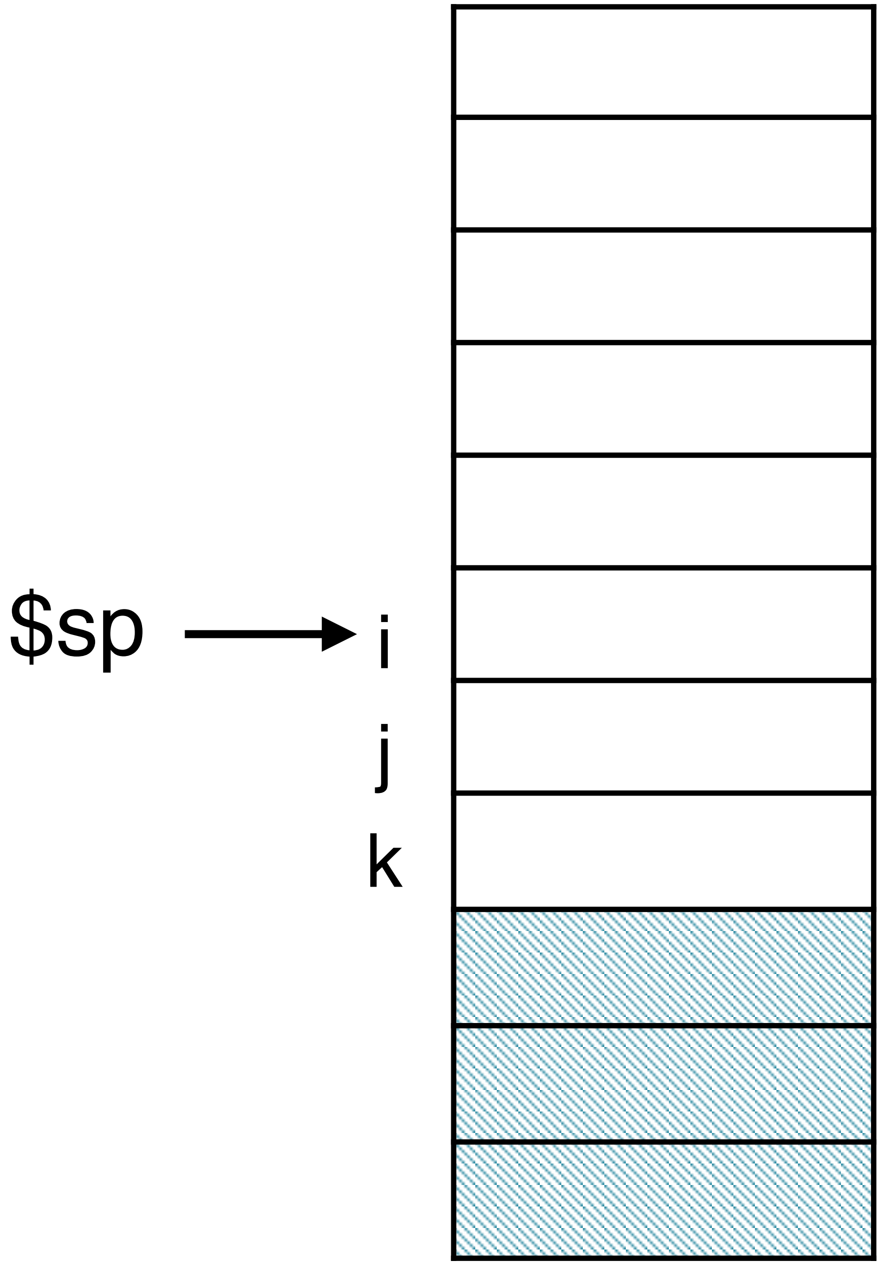
then add 4 to the stack pointer

System stack: **popping**



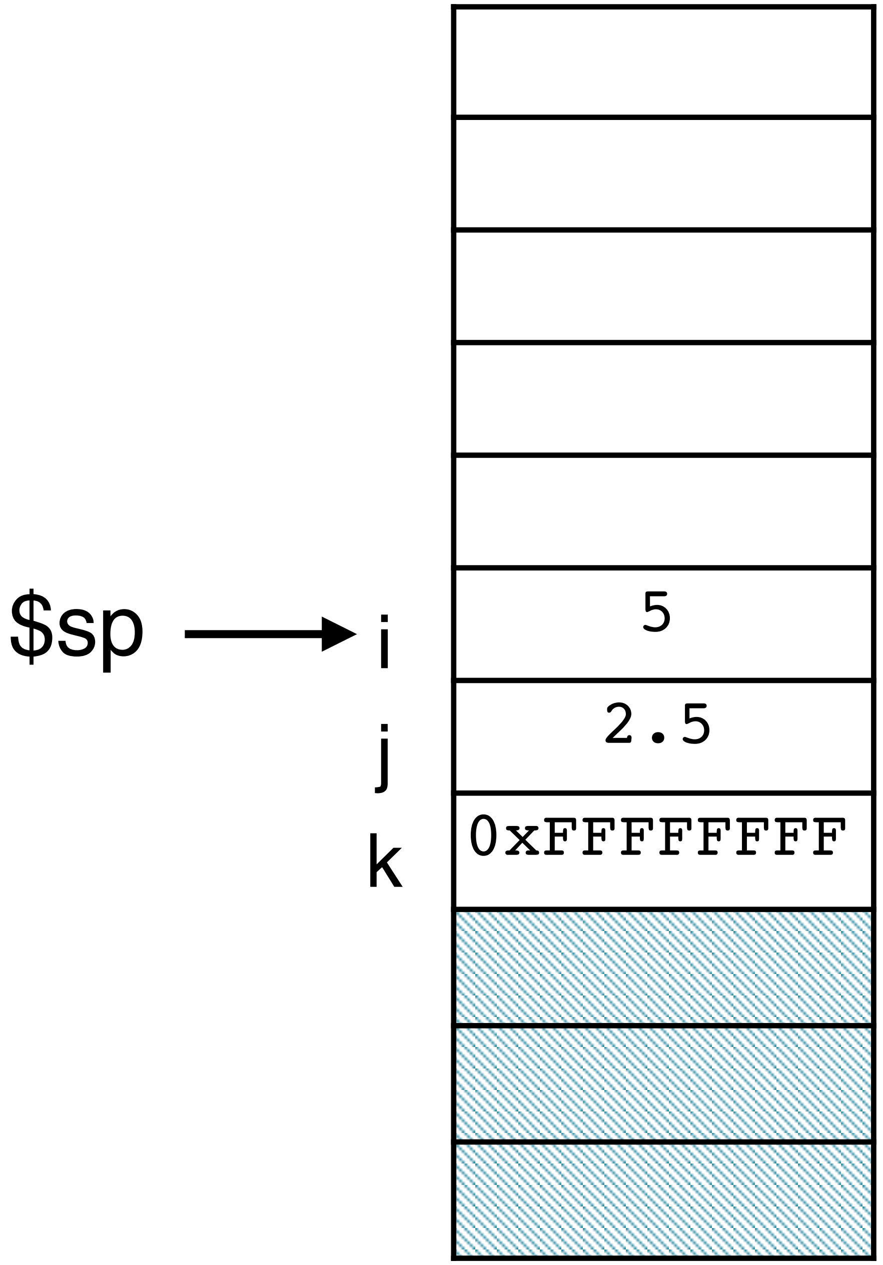
How does the system stack work?

- At the beginning of a function
  - ➡ **Allocate** variables by **pushing** necessary space onto stack (subtract n bytes from **$sp**)
  - ➡ Initialise space by storing values in newly allocated space



How does the system stack work?

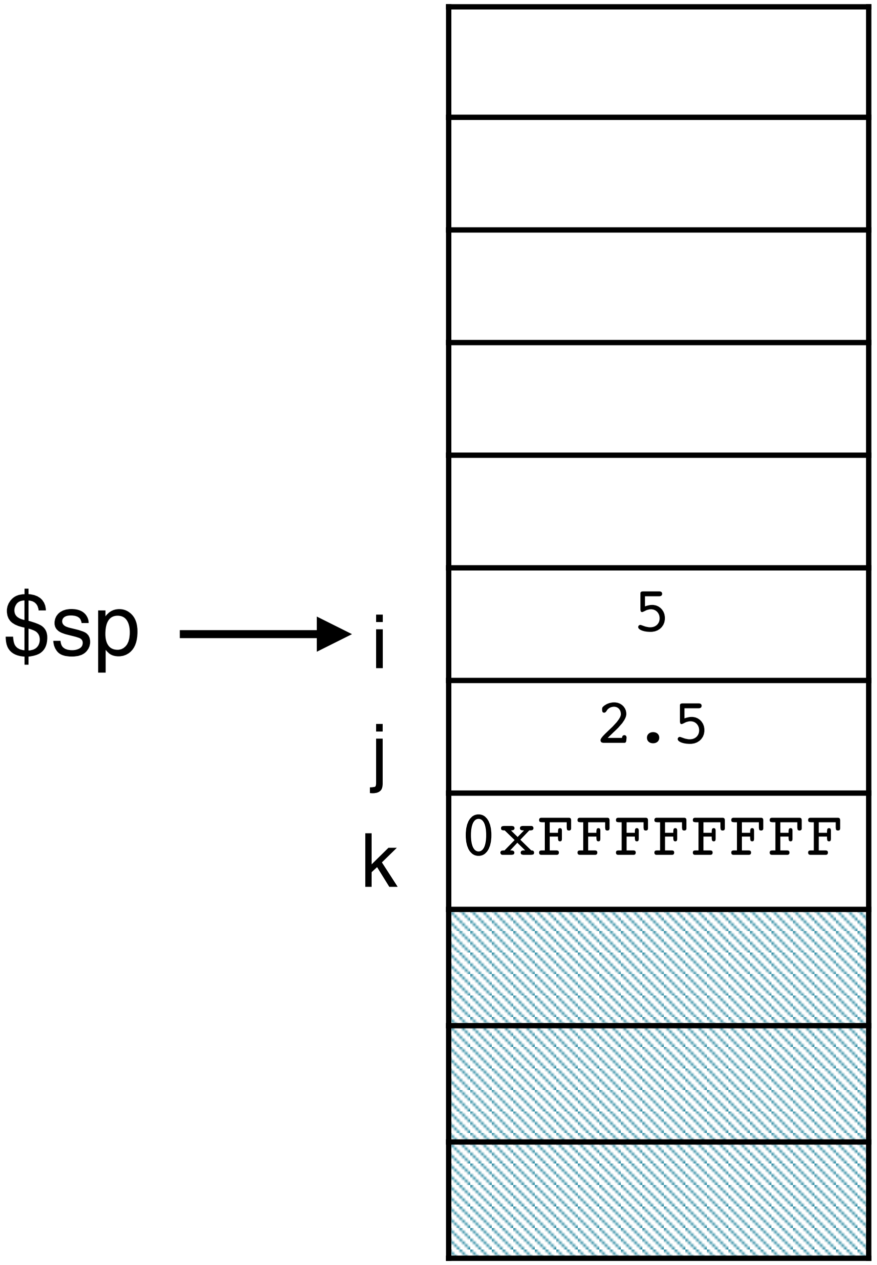
- At the beginning of a function
  - ➡ **Allocate** variables by **pushing** necessary space onto stack (subtract n bytes from **$sp**)
  - ➡ Initialise space by storing values in newly allocated space



How does the system stack work?

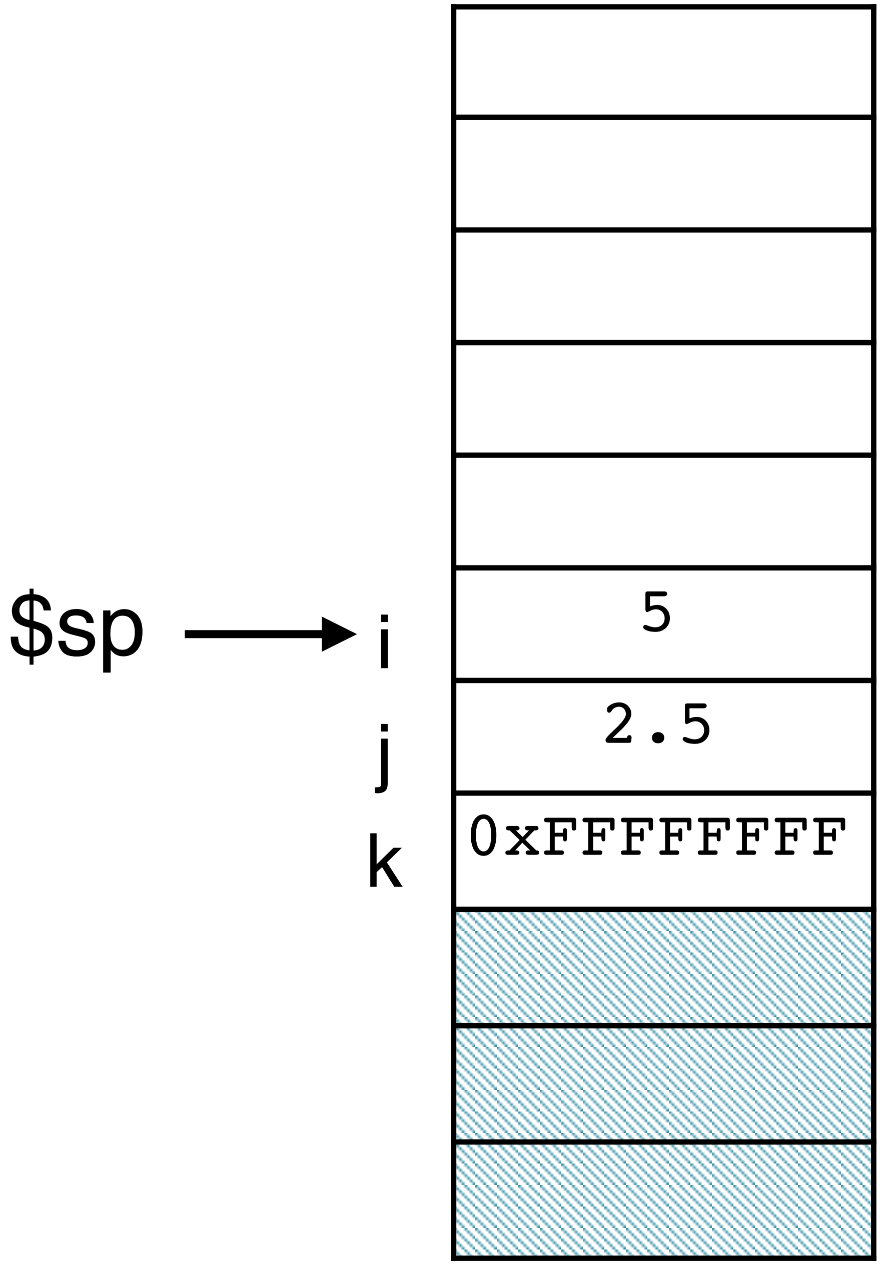
- At the beginning of a function
  - ➡ **Allocate** variables by **pushing** necessary space onto stack (subtract n bytes from **$sp**)
  - ➡ Initialise space by storing values in newly allocated space

- During function: use variables using **lw/sw**



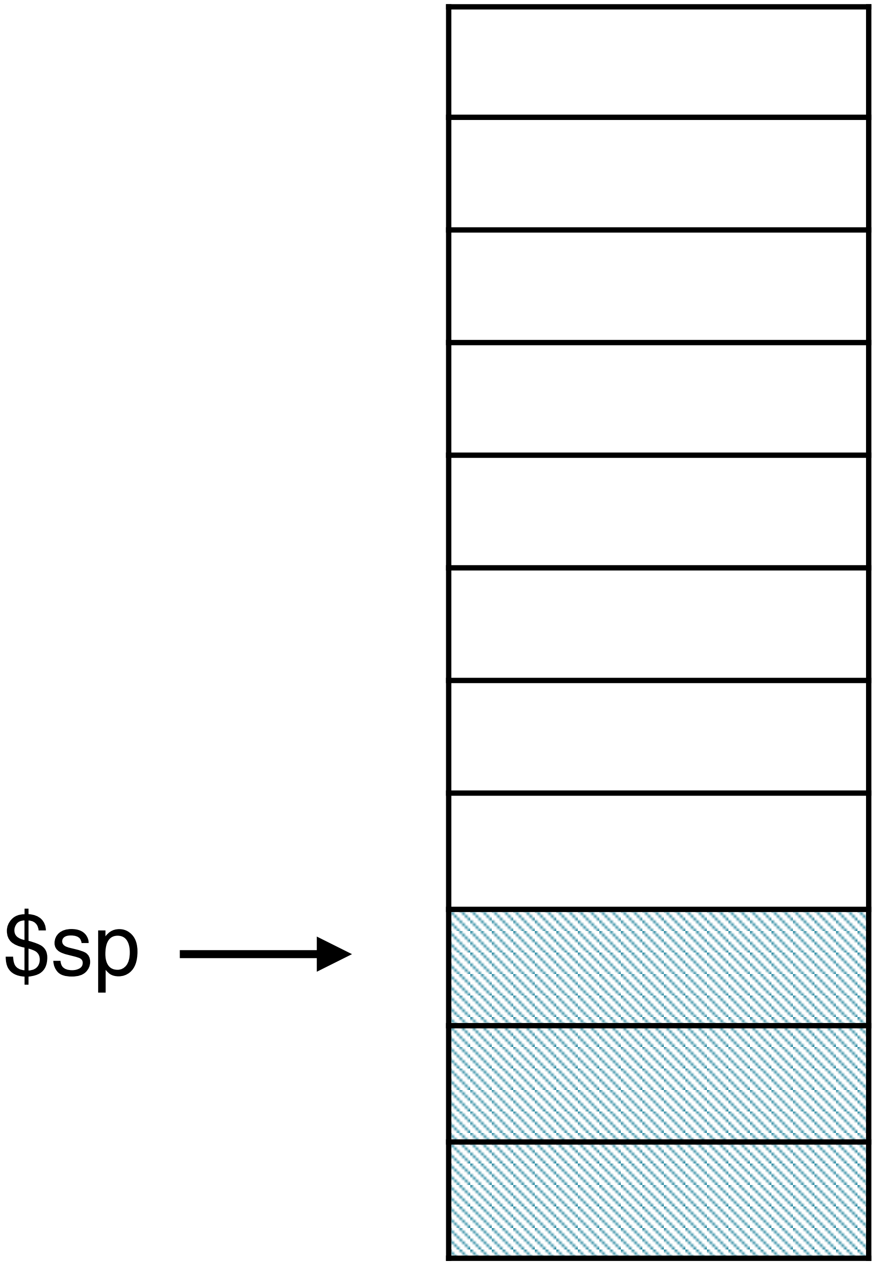
How does the system stack work?

- At the beginning of a function
  - ➡ **Allocate** variables by **pushing** necessary space onto stack (subtract n bytes from **$sp**)
  - ➡ Initialise space by storing values in newly allocated space
- During function: use variables using **lw/sw**
- At the end of the function: **Deallocate** variables by **popping** allocated space from stack (add n bytes to **$sp**)



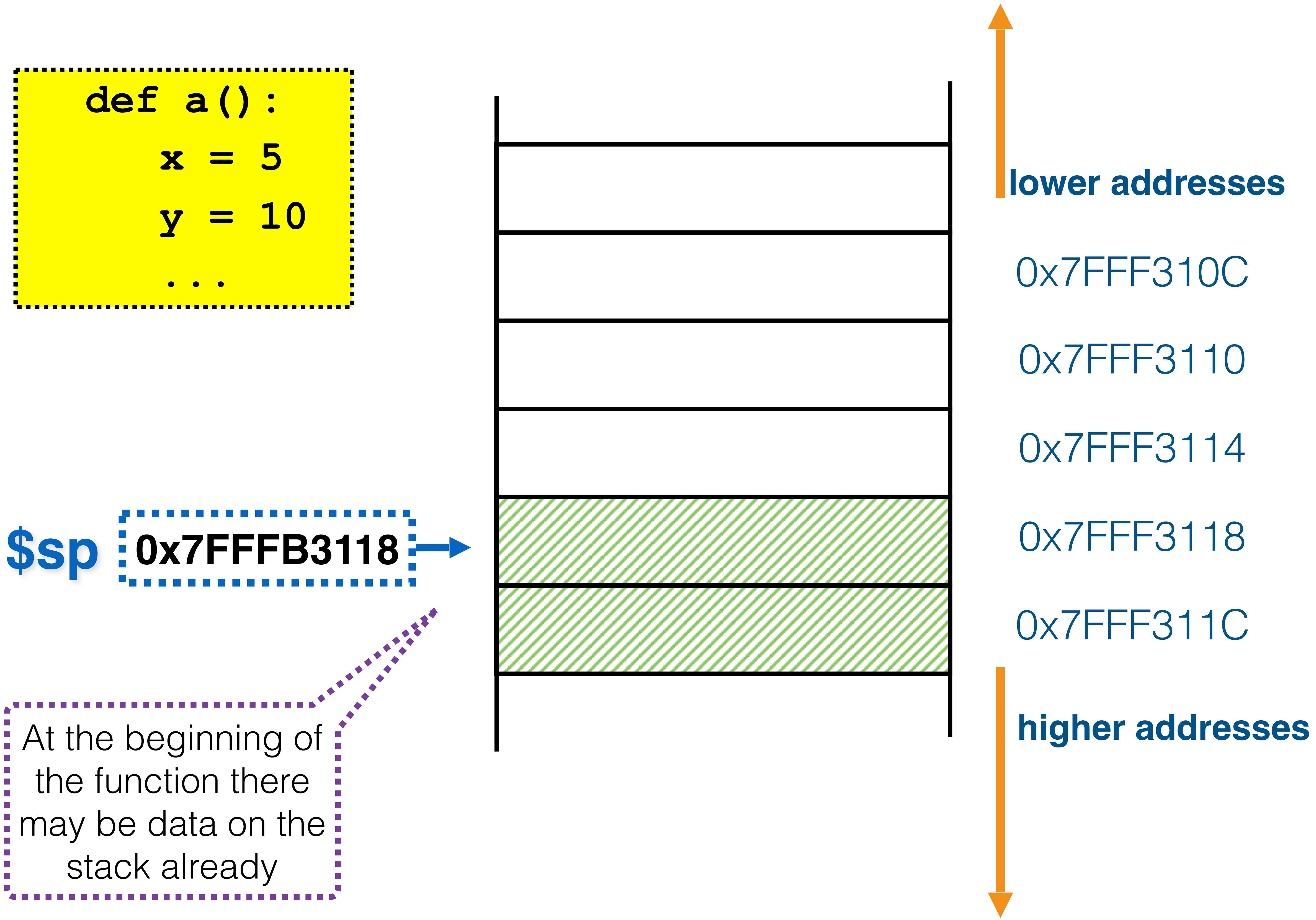
How does the system stack work?

- At the beginning of a function
  - ➡ **Allocate** variables by **pushing** necessary space onto stack (subtract n bytes from **$sp**)
  - ➡ Initialise space by storing values in newly allocated space
- During function: use variables using **lw/sw**
- At the end of the function: **Deallocate** variables by **popping** allocated space from stack (add n bytes to **$sp**)



Example

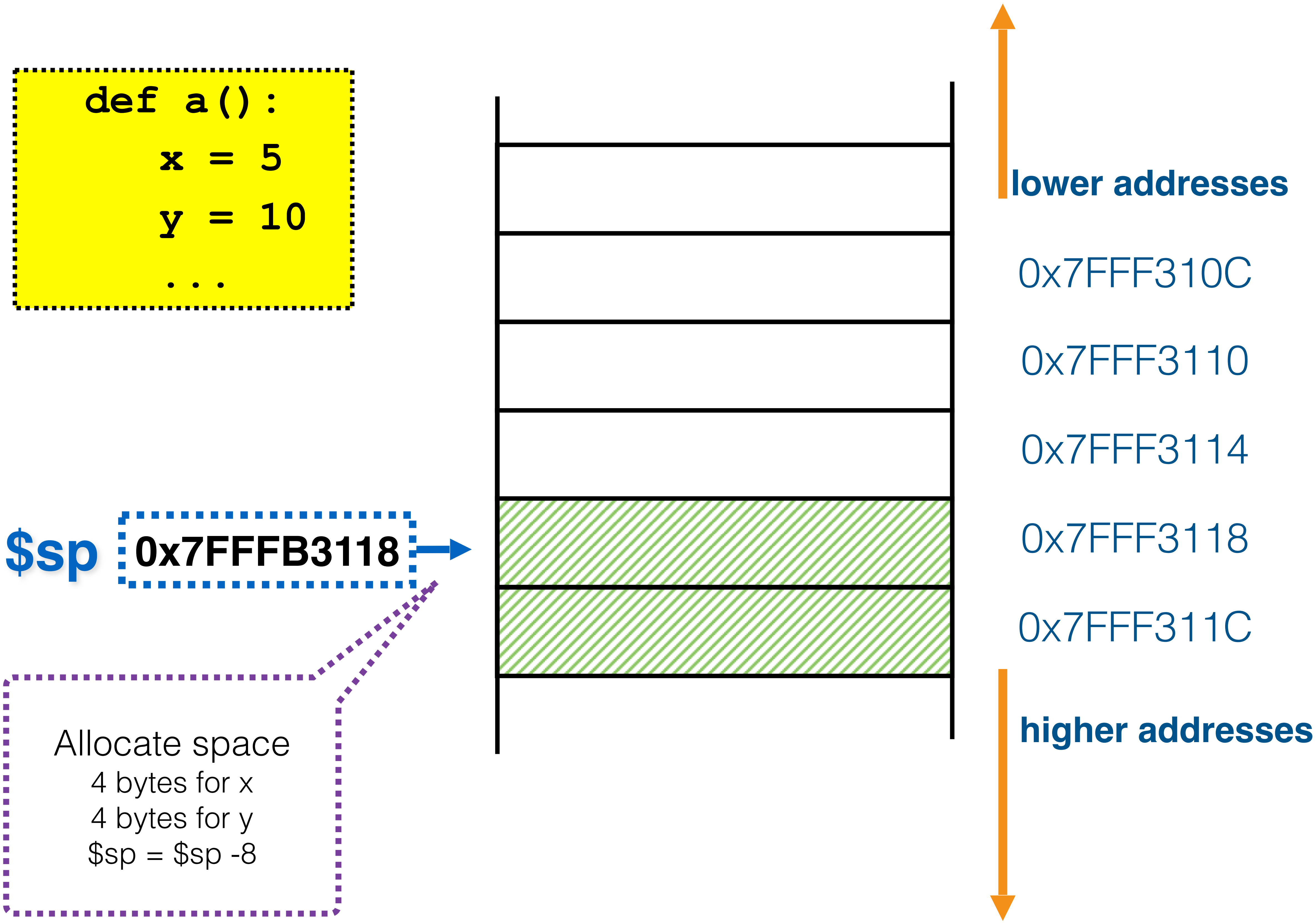
```
def a():
    x = 5
    y = 10
    ...
```



At the beginning of the function there may be data on the stack already

Example

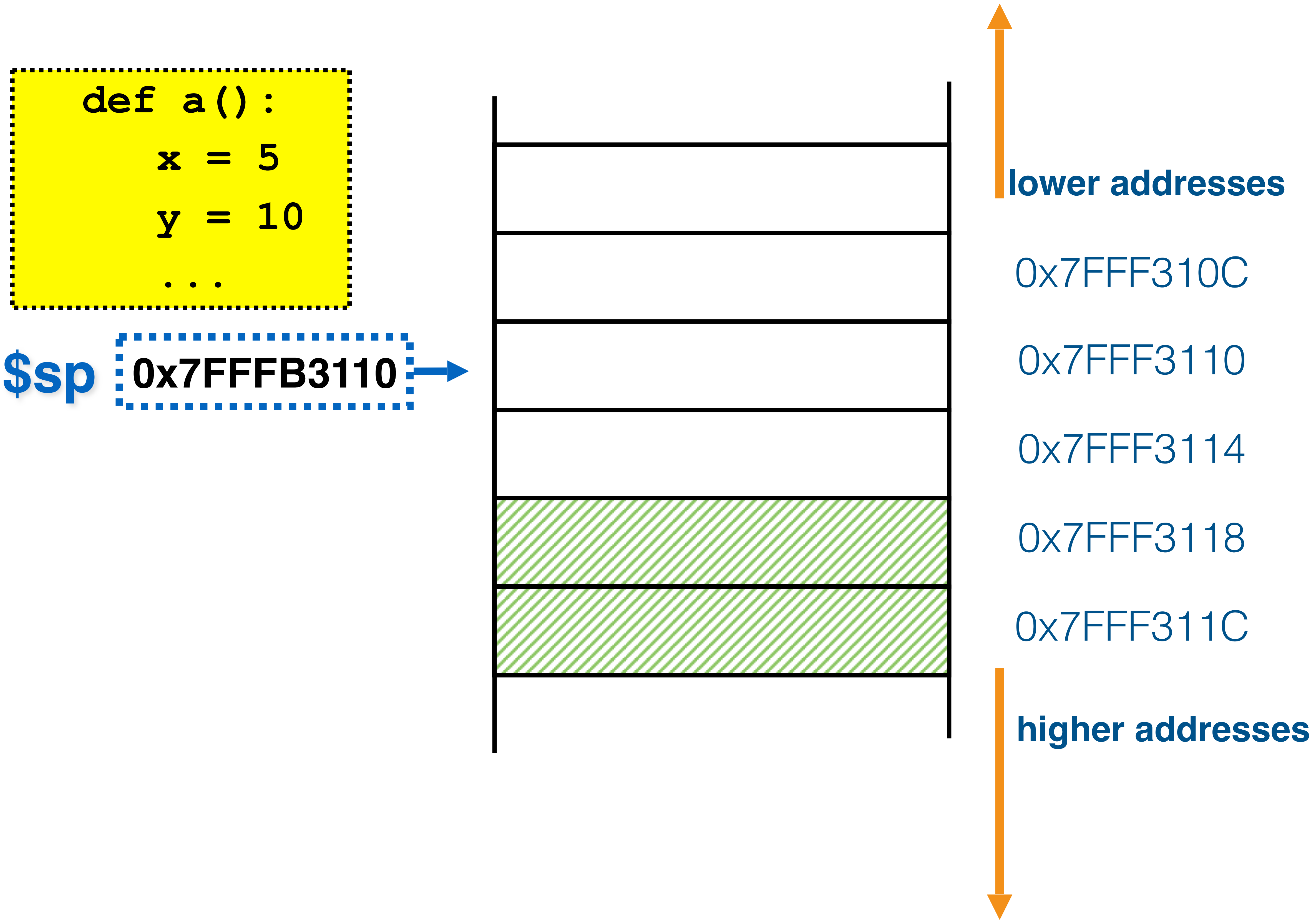
```
def a():
    x = 5
    y = 10
    ...
```



Allocate space
4 bytes for x
4 bytes for y
$sp = $sp -8

Example

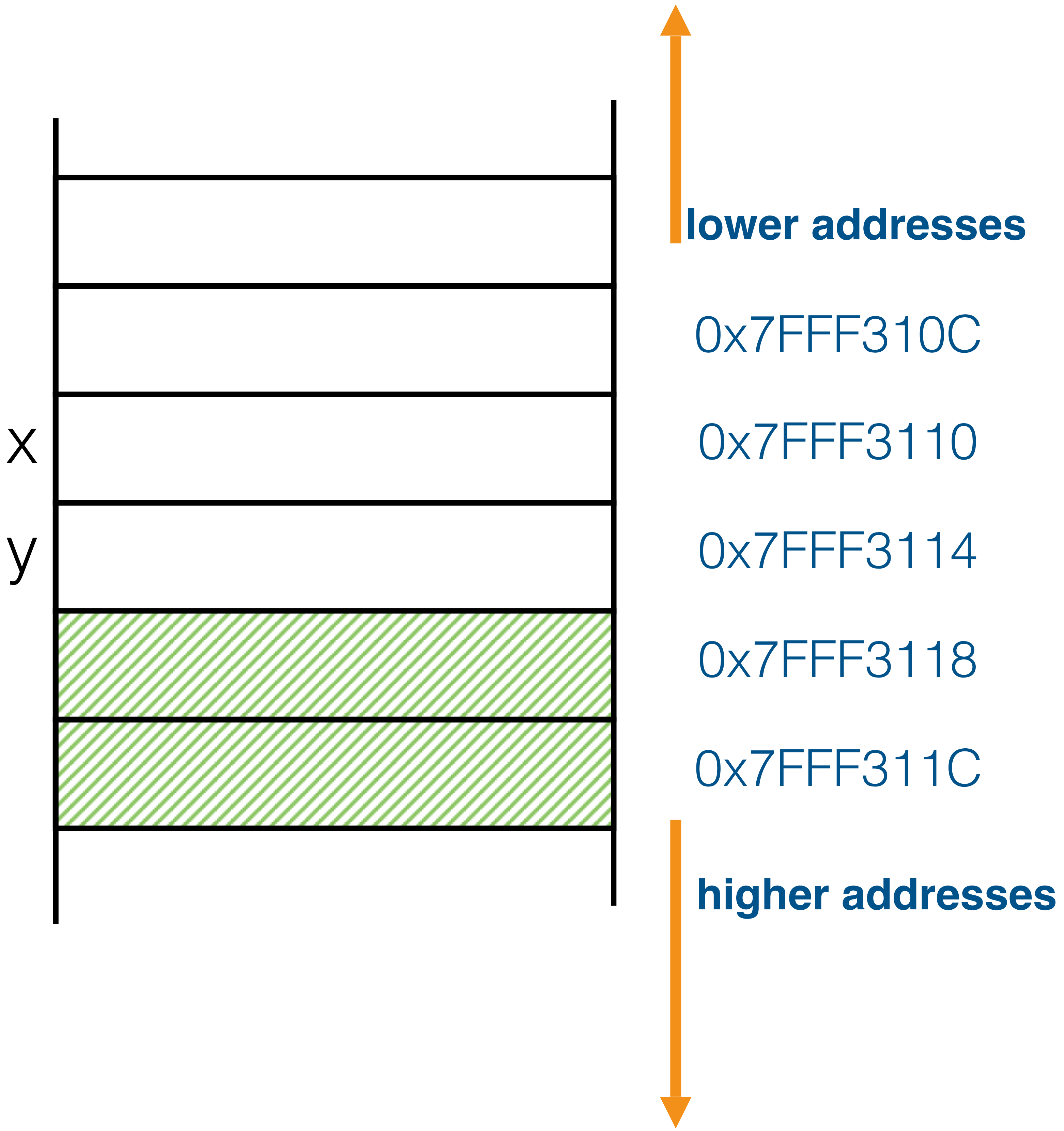
```
def a():
    x = 5
    y = 10
    ...
```



Example

```
def a():
    x = 5
    y = 10
    ...
```

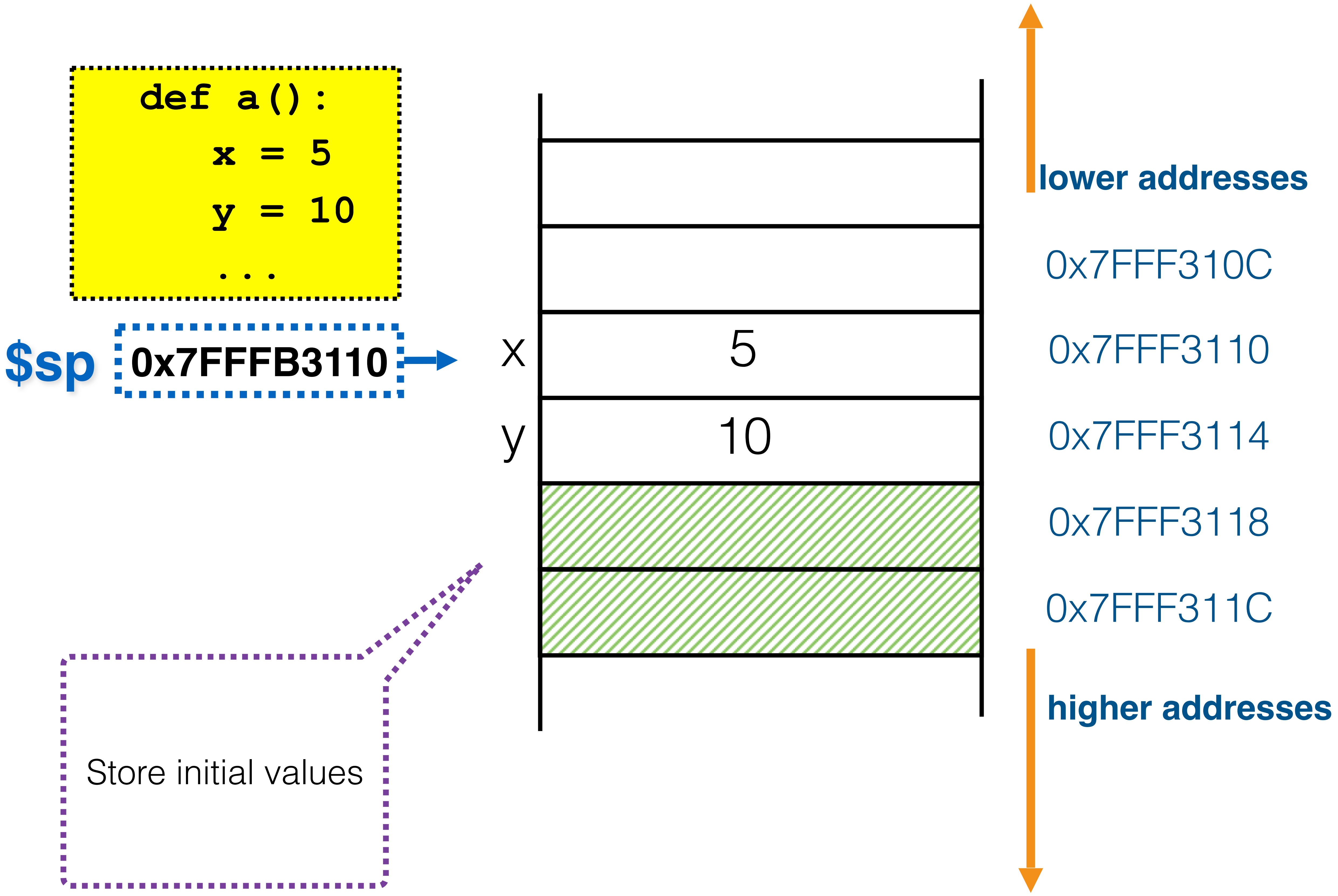
$sp 0x7FFFB3110



Assign variables

Example

```
def a():
    x = 5
    y = 10
    ...
```



$sp 0x7FFFB3110

lower addresses

		0x7FFF310C
x	5	0x7FFF3110
y	10	0x7FFF3114
	▨	0x7FFF3118
	▨	0x7FFF311C

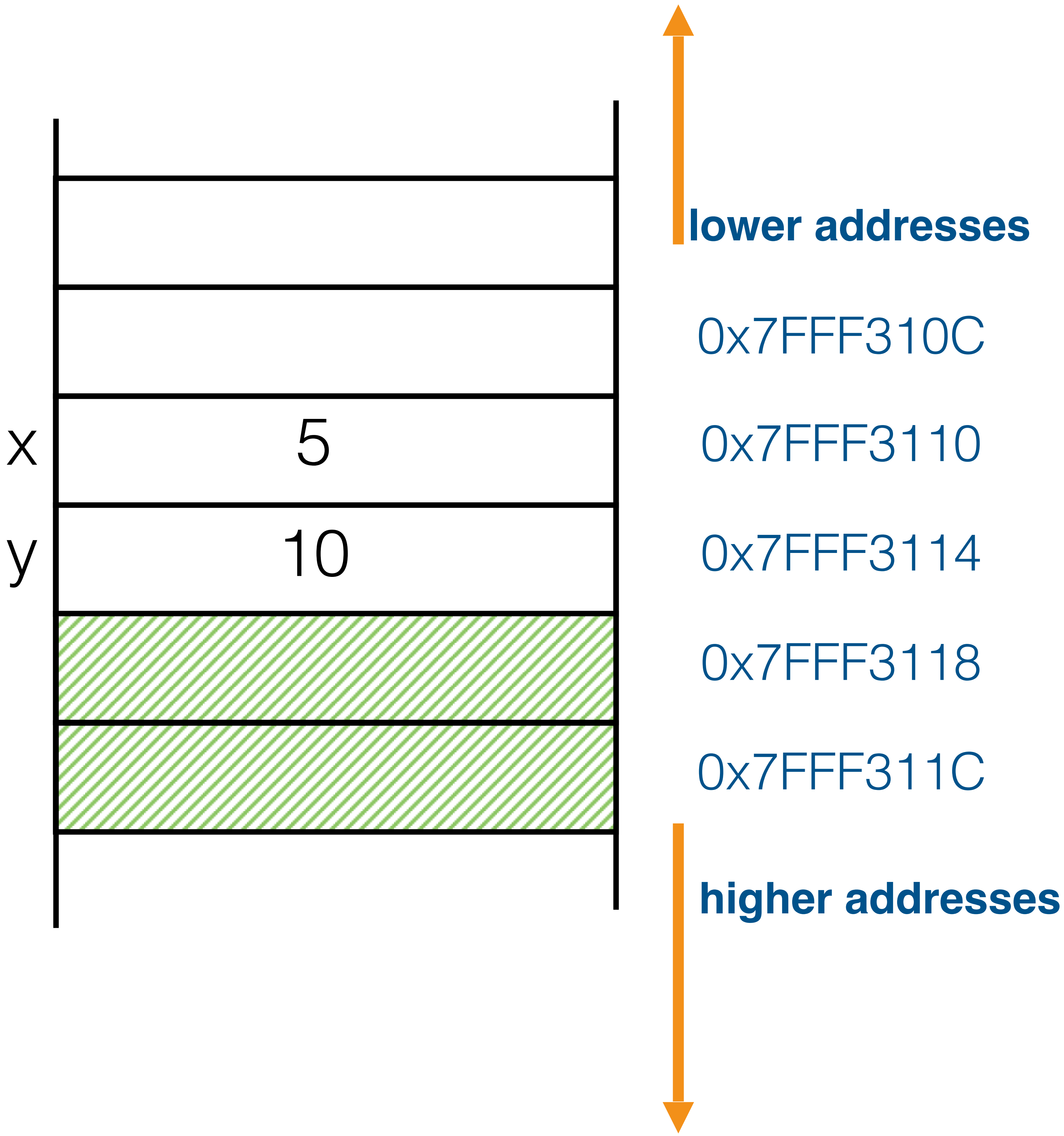
higher addresses

Store initial values

Example

```
def a():
    x = 5
    y = 10
    ...
```

$sp 0x7FFFB3110

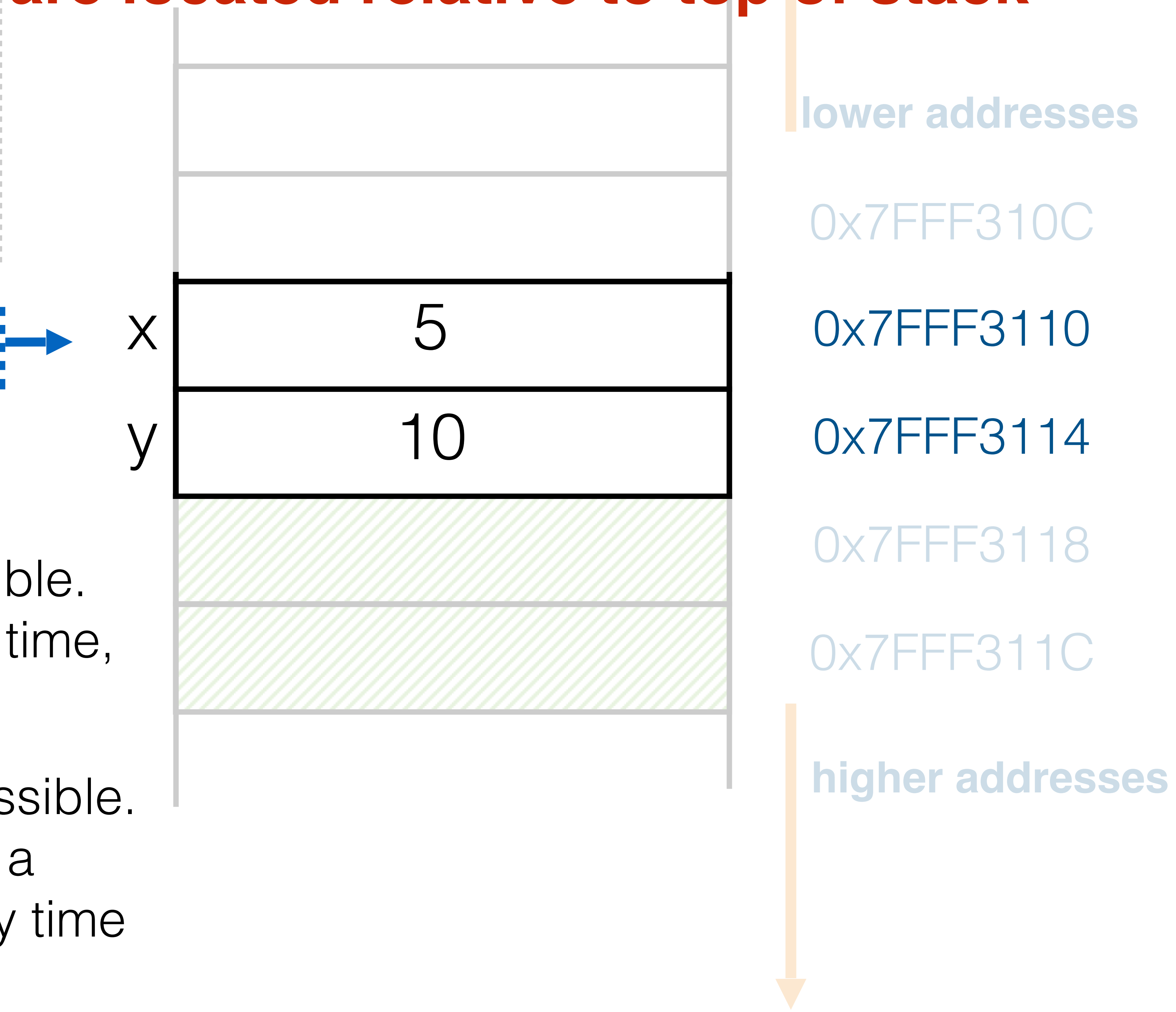


How do we use these values or refer to them?

I can use $sp since variables are located relative to top of stack

```
def a():
    x = 5
    y = 10
    ...
```

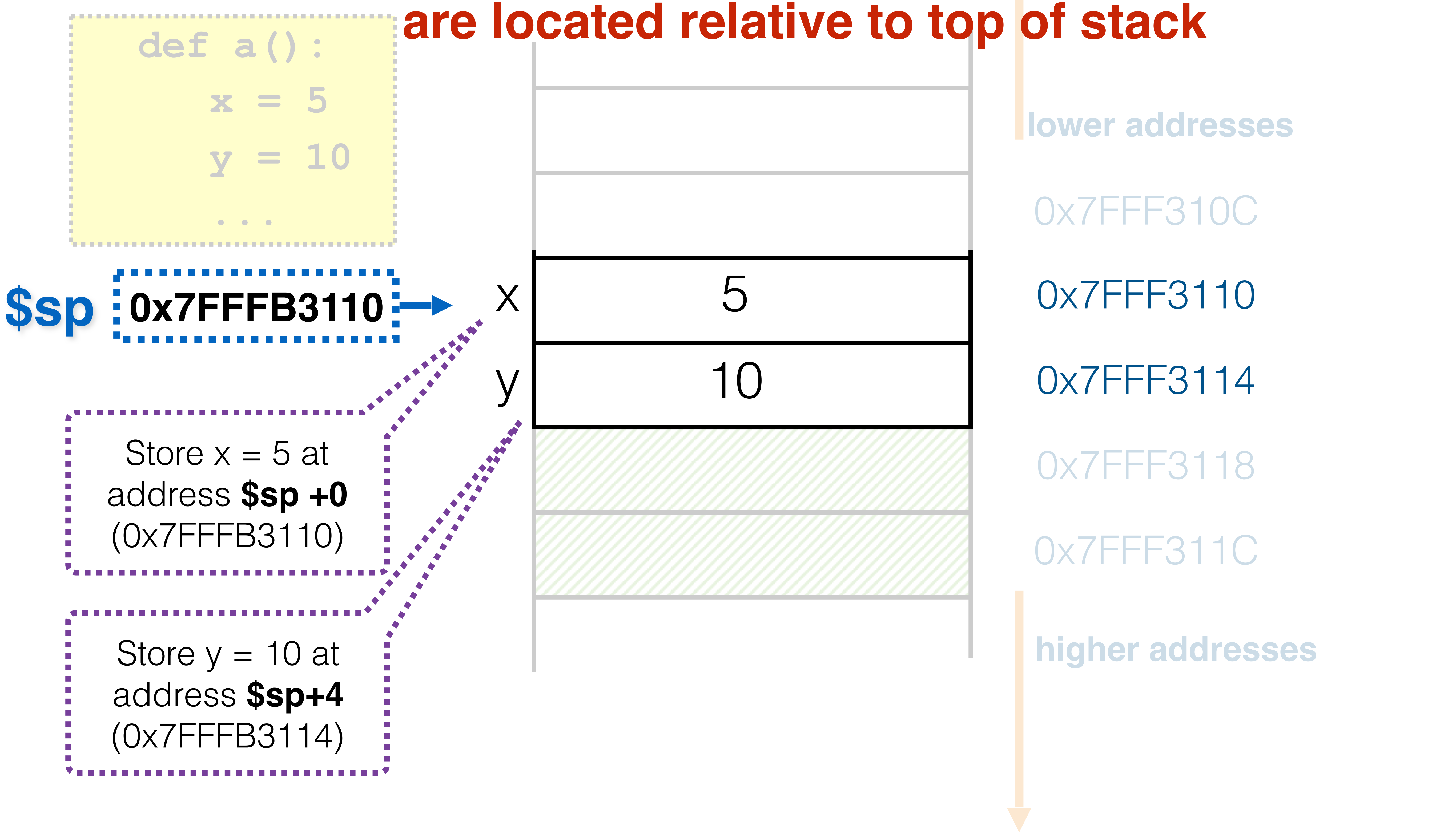
$sp **0x7FFFB3110**



Labels? Not possible. Labels are compile time, not run-time

Addresses? Not possible. stack may have a different depth every time

I can use $sp since variables are located relative to top of stack



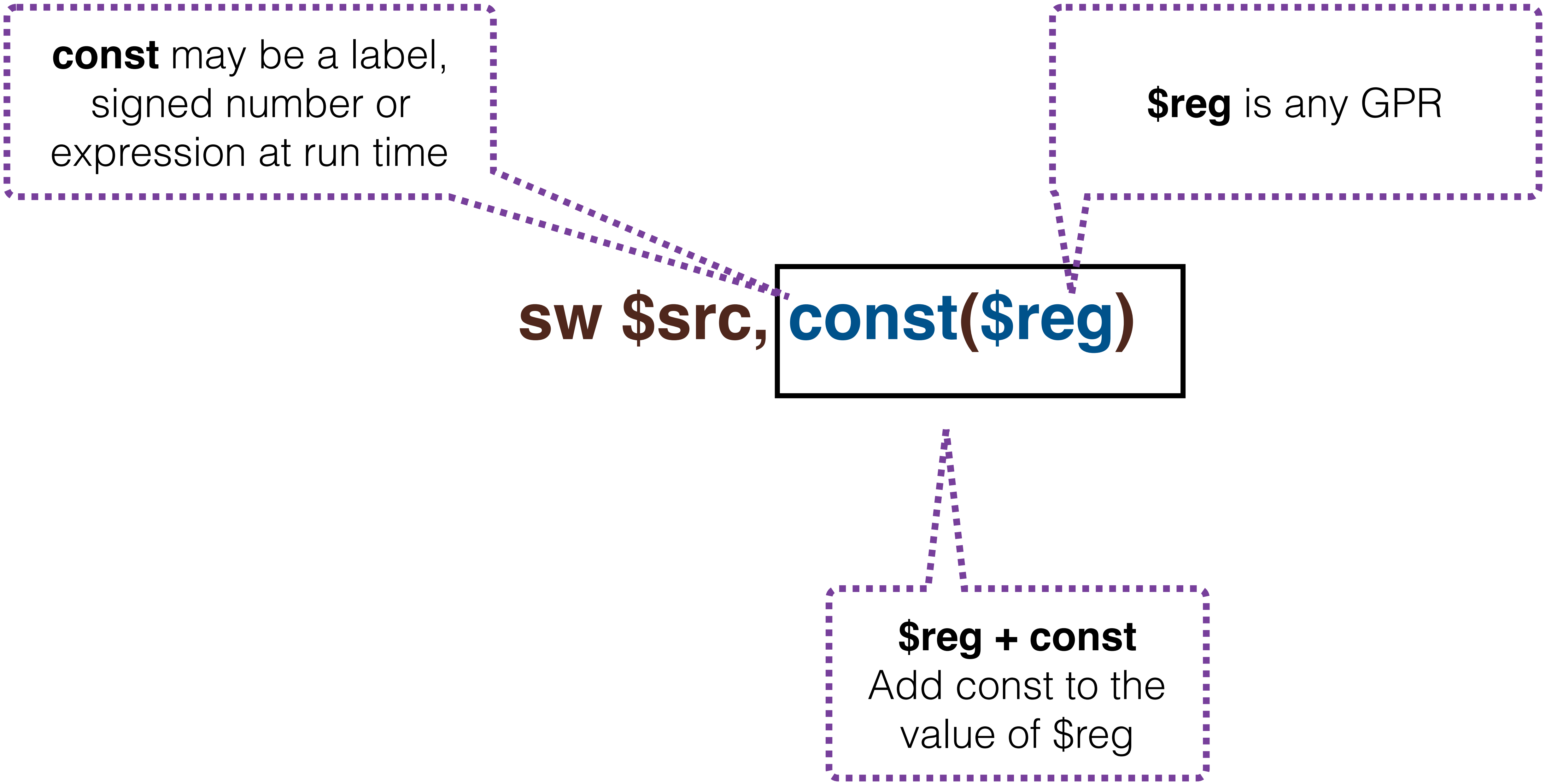
```
def a():
    x = 5
    y = 10
    ...
```

$sp 0x7FFFB3110

Store x = 5 at address **$sp +0** (0x7FFFB3110)

Store y = 10 at address **$sp+4** (0x7FFFB3114)

Reminder: addressing modes



const may be a label, signed number or expression at run time

$reg is any GPR

```
sw $src, const($reg)
```

$reg + const
Add const to the value of $reg

Examples of addressing modes

sw $t0, 4($sp)	address is ($sp + 4)
sw $t0, -4($fp)	address is ($fp – 4)
lw $a0, 0($sp)	address is ($sp + 0)
lw $a0, ($sp)	
lw $a0, var($zero)	address is ($zero + address of var)
lw $a0, var	

Summary

- Memory diagrams.
- System stack:
  - ➡ Pushing and popping
  - ➡ **$sp**
- Local variables:
  - ➡ Stored on stack
- Addressing: register + constant