

Lecture 7

Memory in MIPS

FIT 1008
Introduction to Computer Science



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Objectives

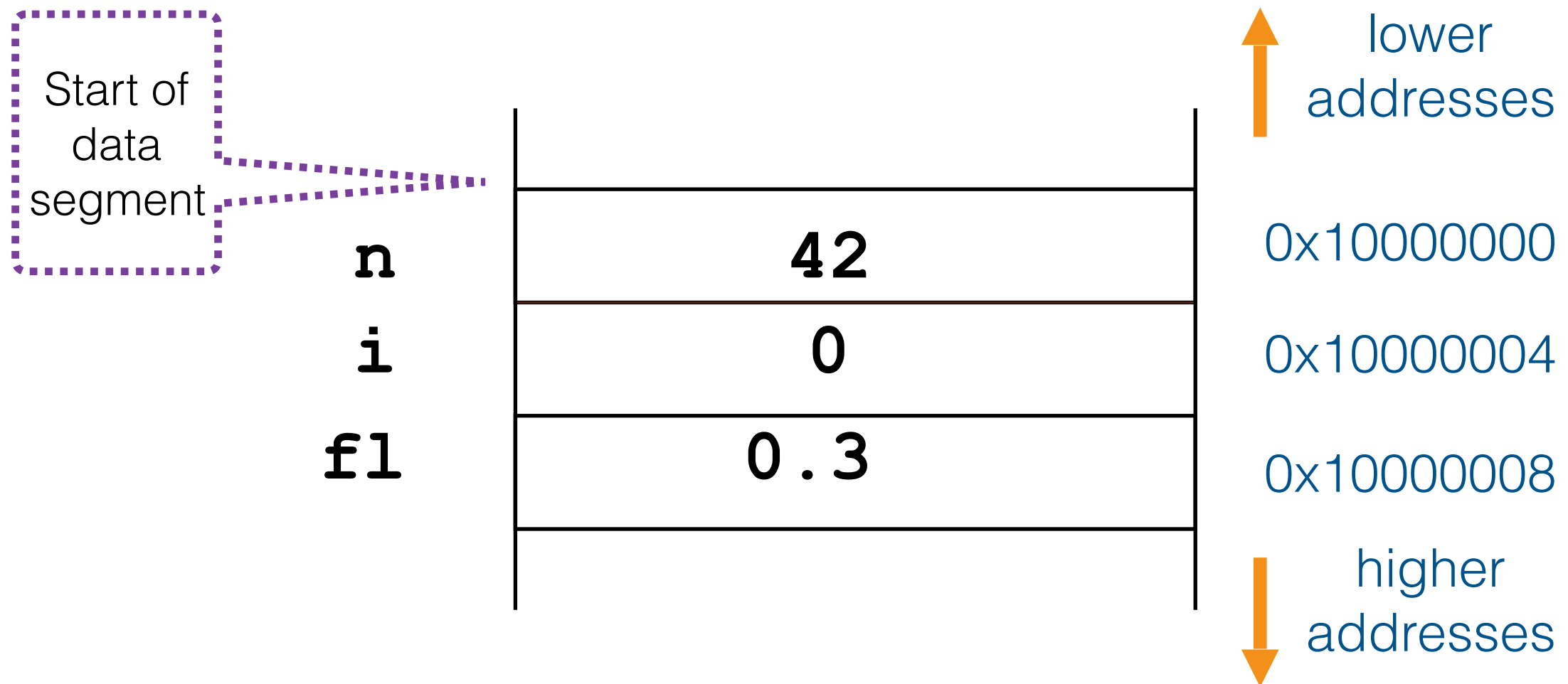
- The need for **memory diagrams** and how to draw them
- How the system stack works and the role played by **\$sp**
- How (and why) **local variables** are stored on the stack and how to access them
- How to use **addressing modes** to access variables

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.

Global variables

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

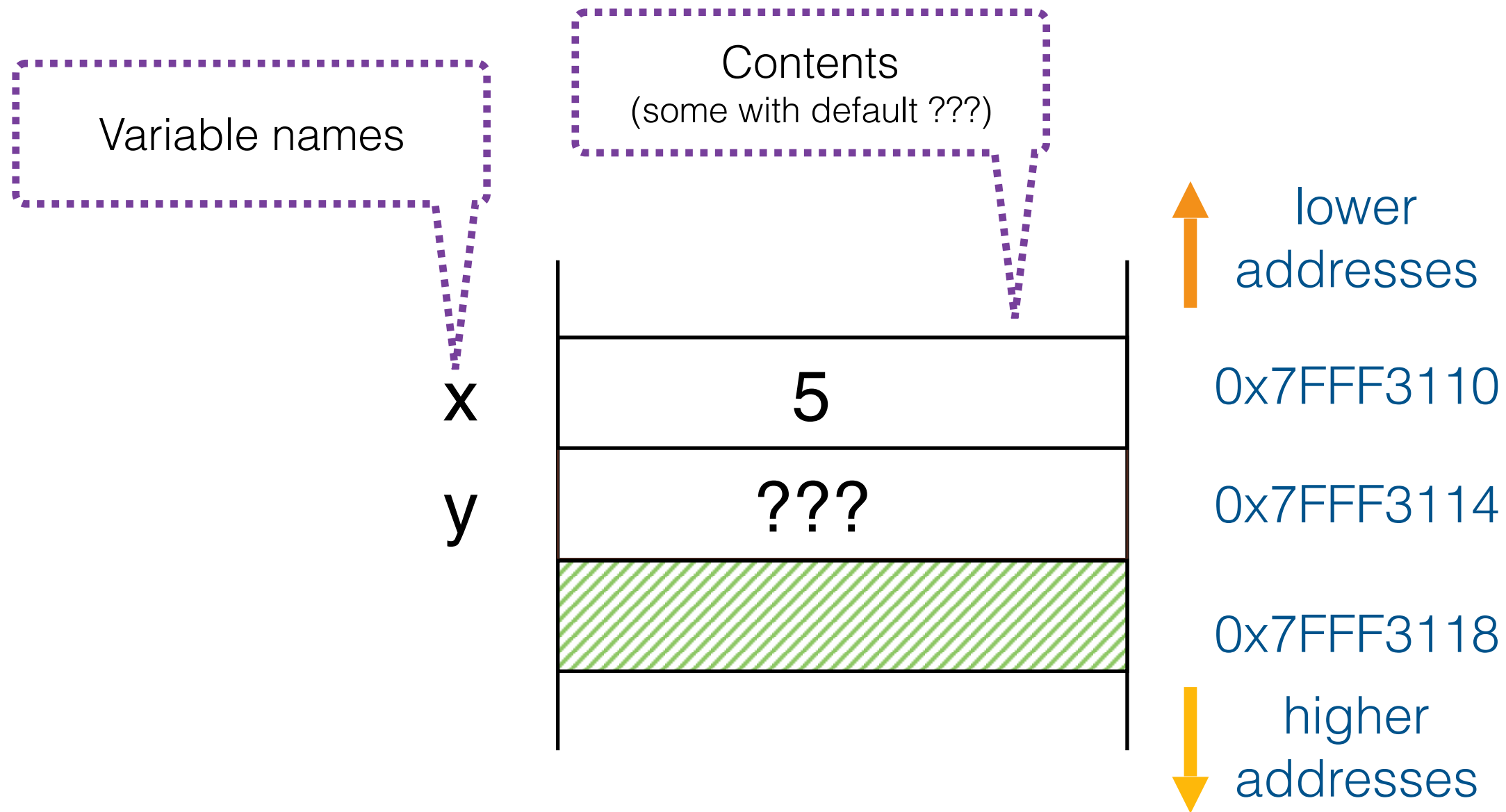


- Local variables, on the other hand, will not have a label so we will need a **memory map**

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.



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

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)

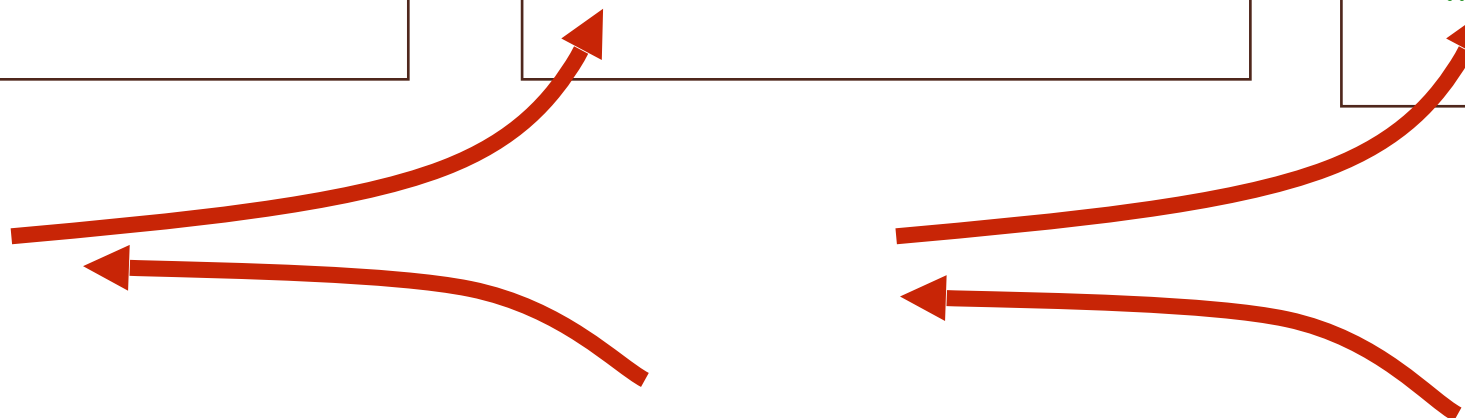
Properties of local variables

- 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: **pushing**

This is the new word we want to push onto the stack

\$sp **0x7FFFB310**

**subtract 4 from
stack pointer**



lower addresses

0x7FFFB308

0x7FFFB30C

0x7FFFB310

0x7FFFB314

0x7FFFB318

0x7FFFB31C

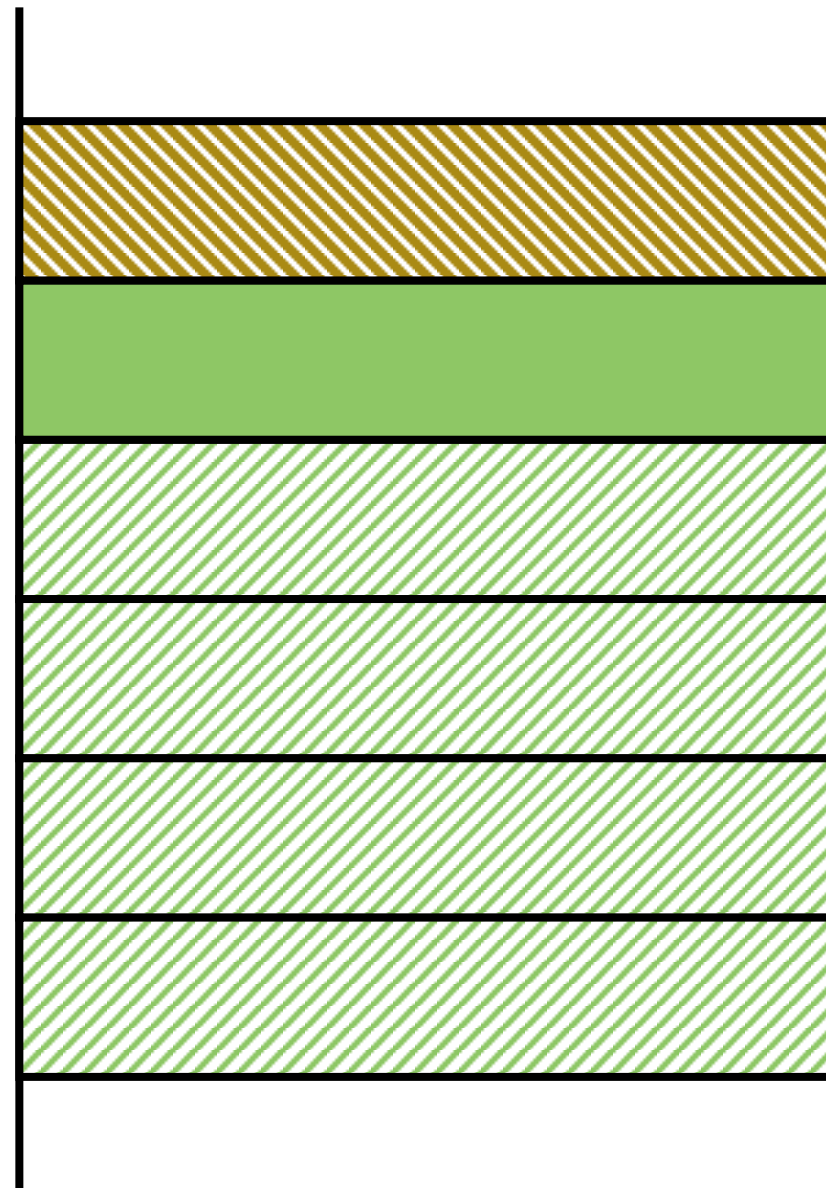
higher addresses

System stack: **popping**

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

\$sp **0x7FFFB30C**

**then add 4 to the
stack pointer**



lower addresses

0x7FFFB308

0x7FFFB30C

0x7FFFB310

0x7FFFB314

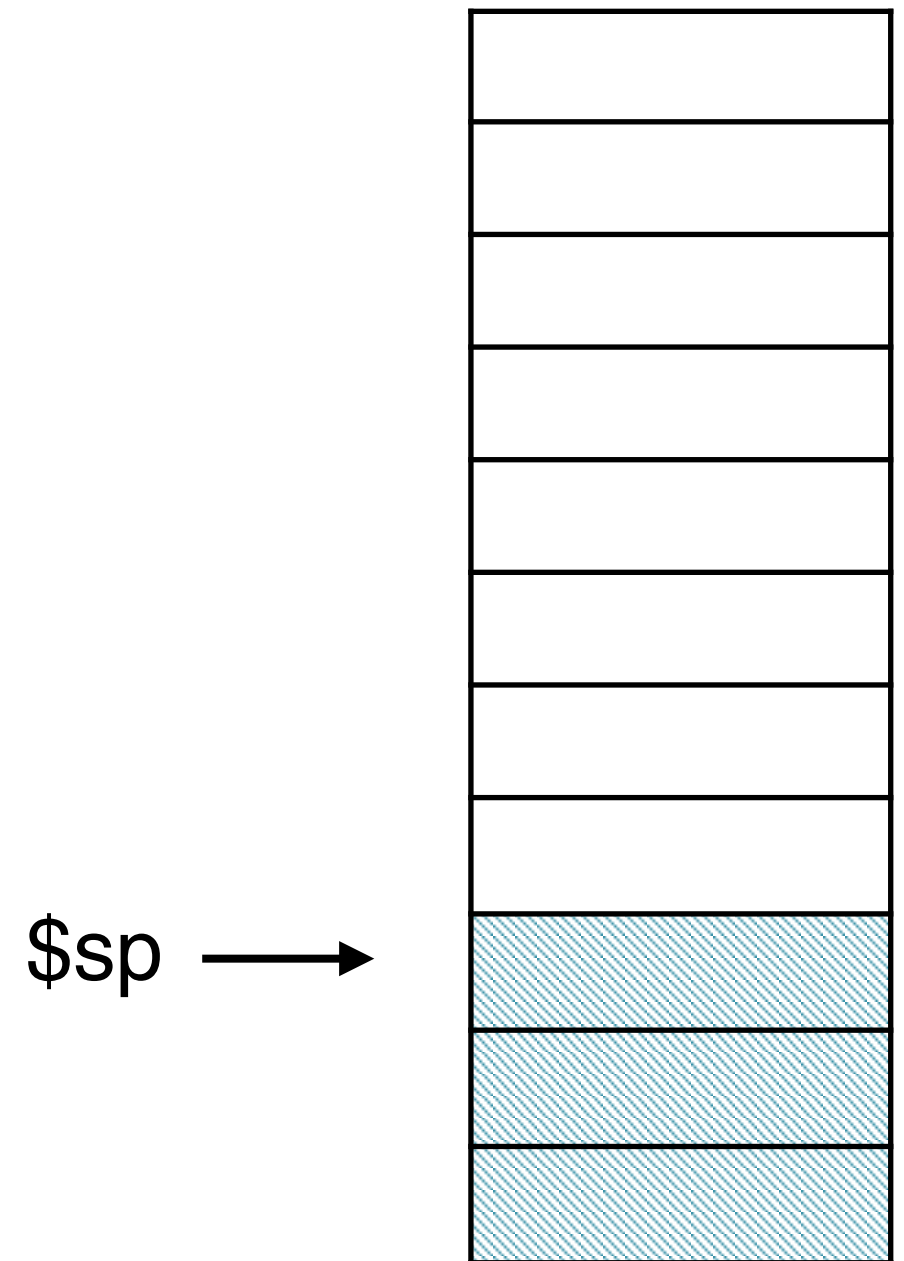
0x7FFFB318

0x7FFFB31C

higher addresses

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**)



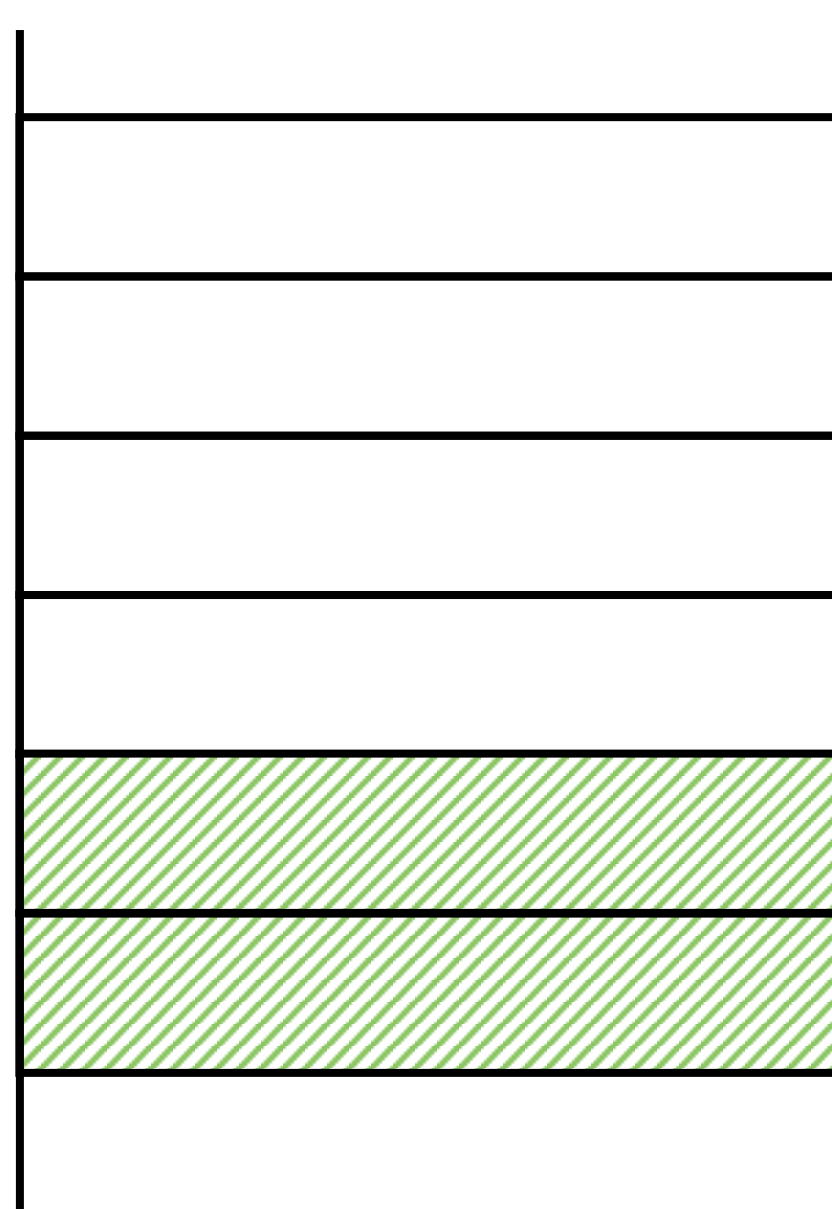
Example

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

\$sp

0x7FFFB3118

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



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

higher addresses

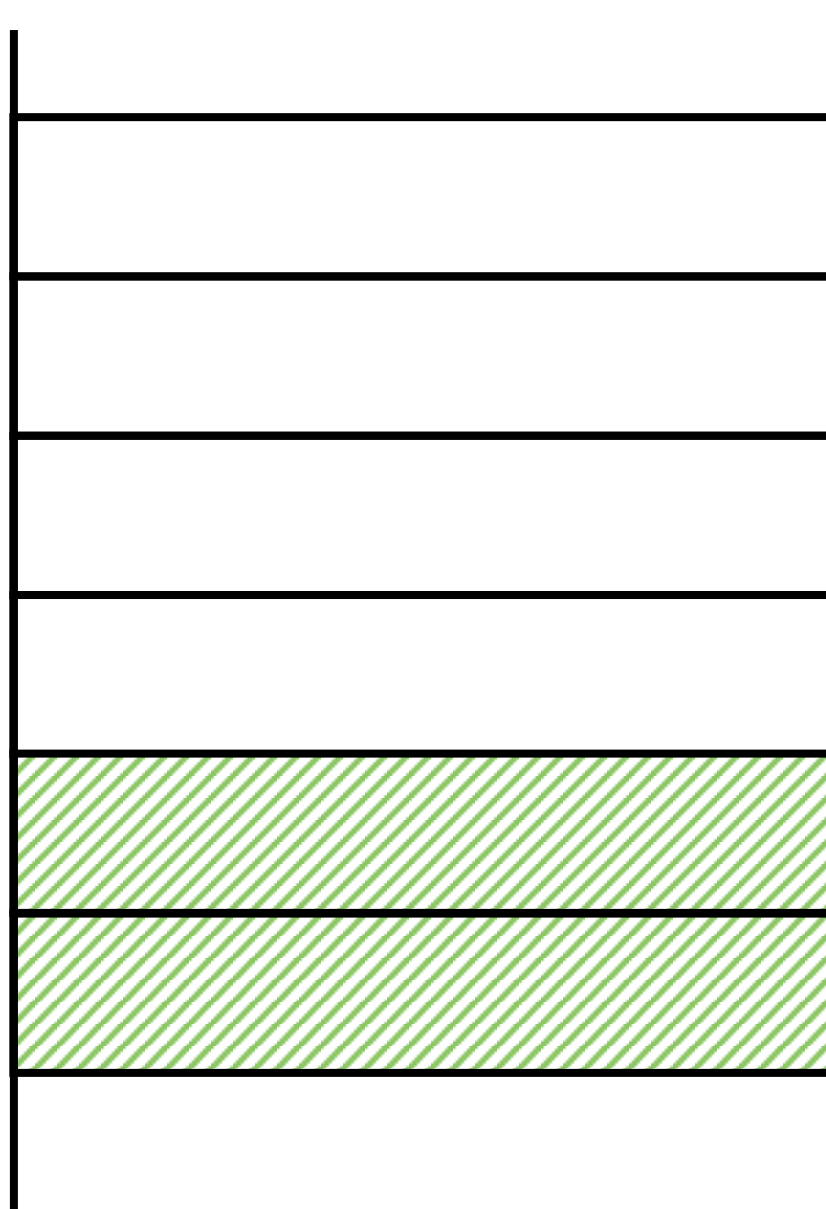
Example

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

\$sp

0x7FFFB3118

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



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

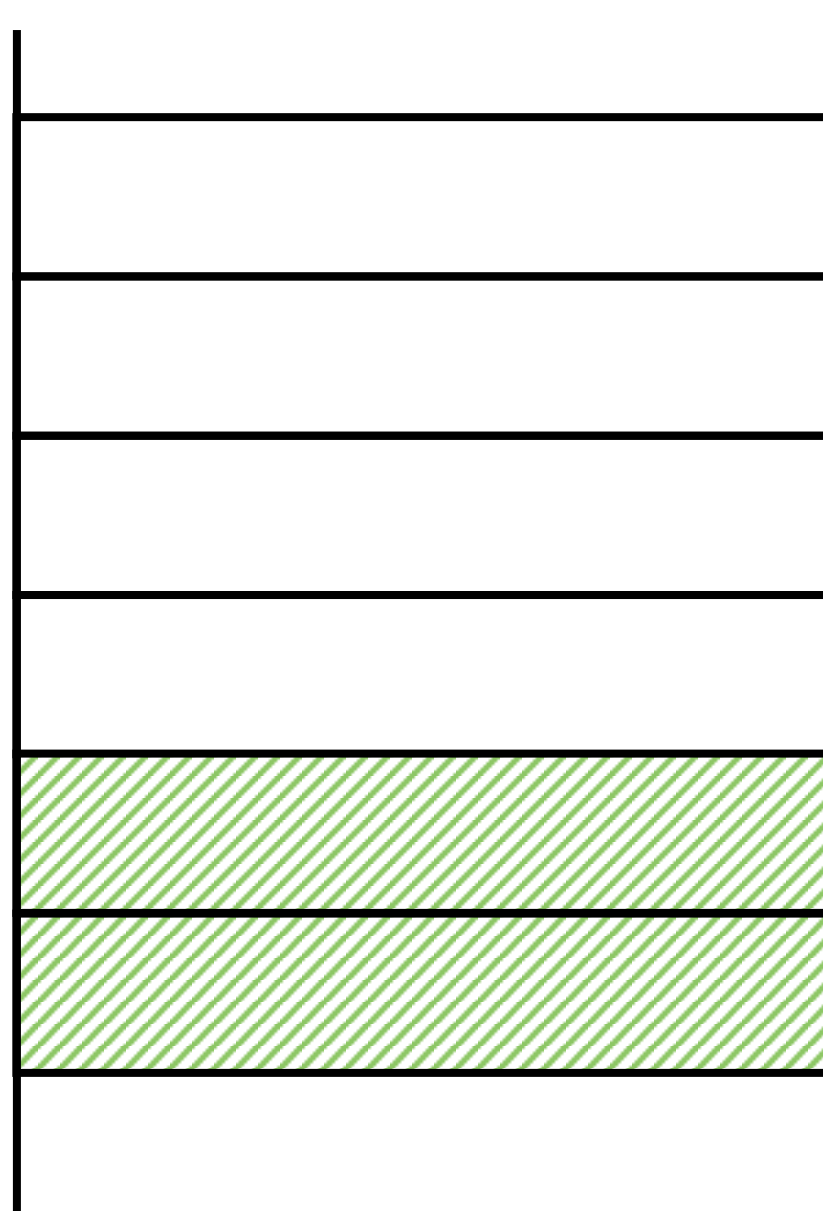
0x7FFF311C

higher addresses

Example

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

\$sp **0x7FFFB3110** →



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

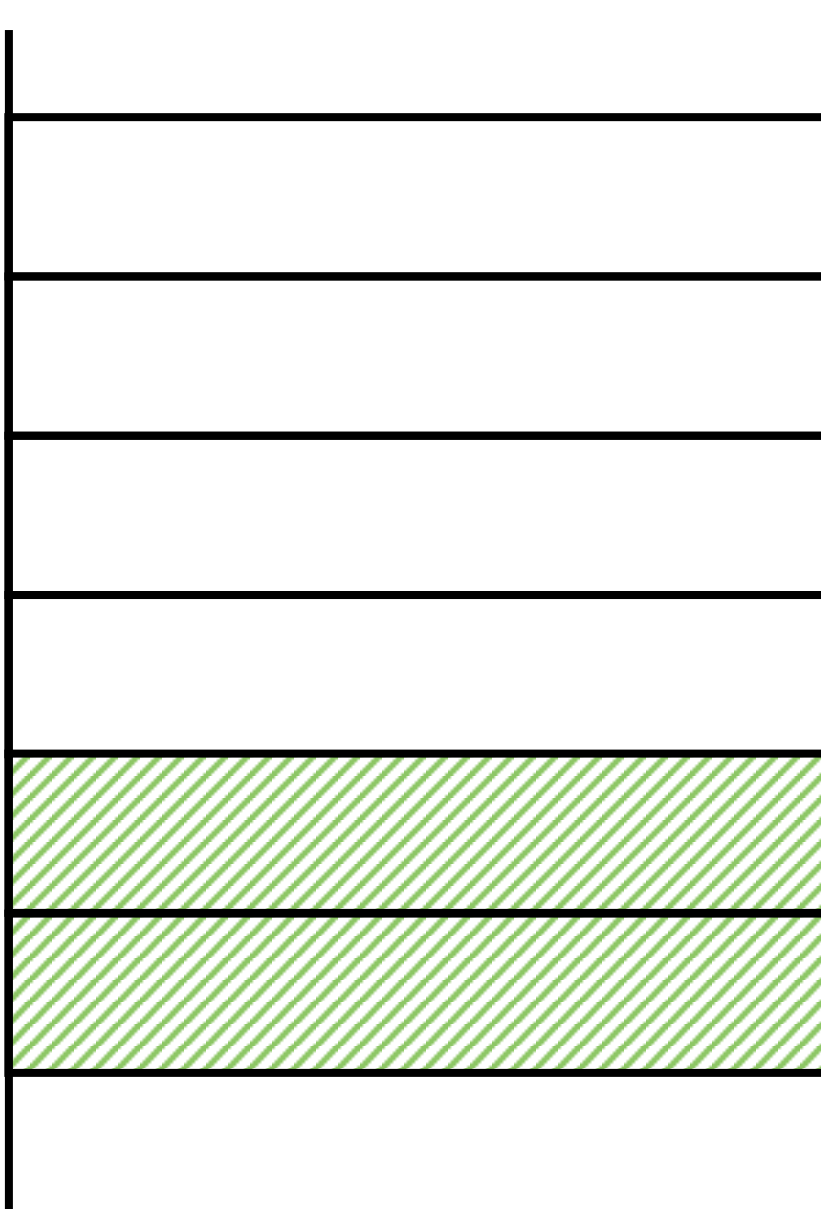
higher addresses

Example

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

\$sp **0x7FFFB3110** →

x
y



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C



higher addresses

Assign variables

Example

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

\$sp

0x7FFFB3110



x

5

y

10

Store initial values

lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

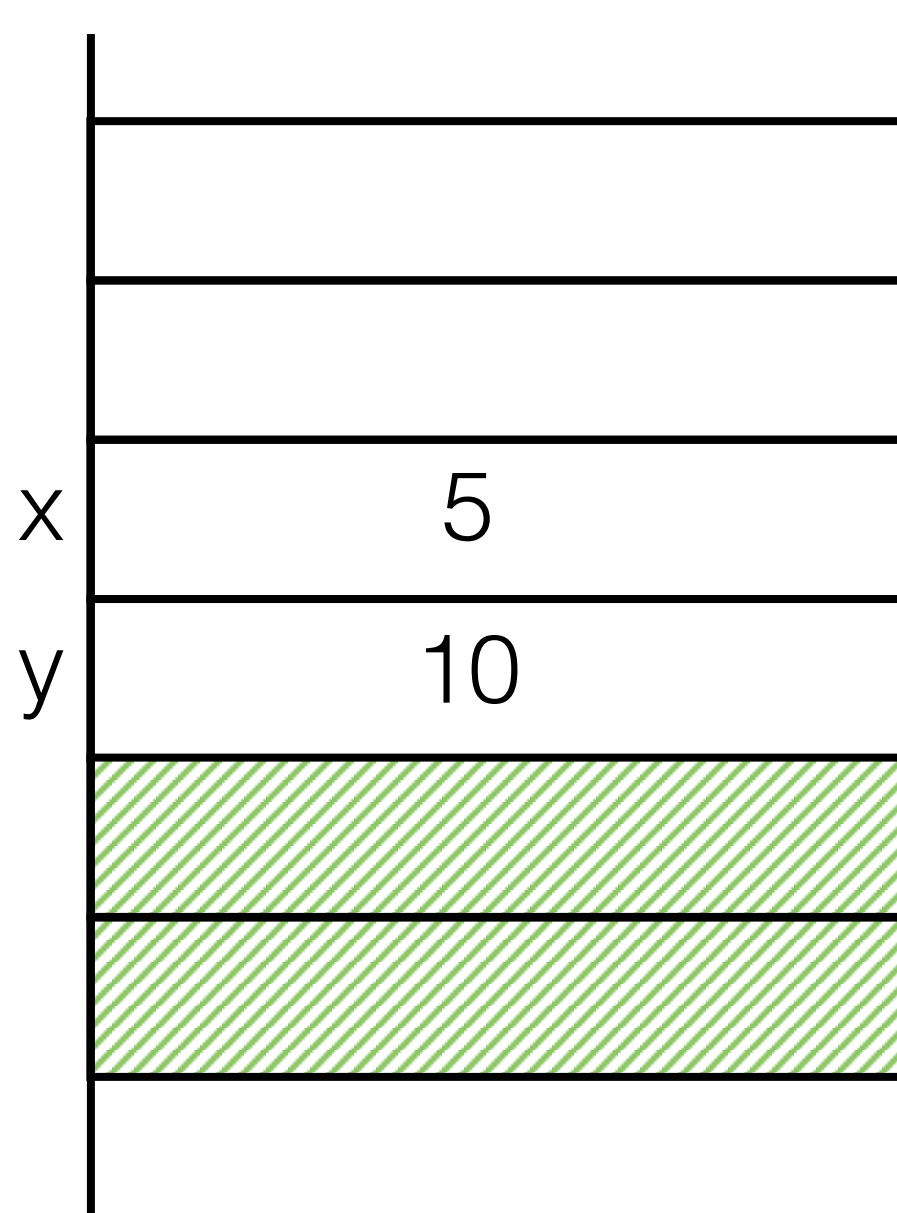
0x7FFF311C

higher addresses

Example

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

\$sp **0x7FFFB3110** →



↑ **lower addresses**

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

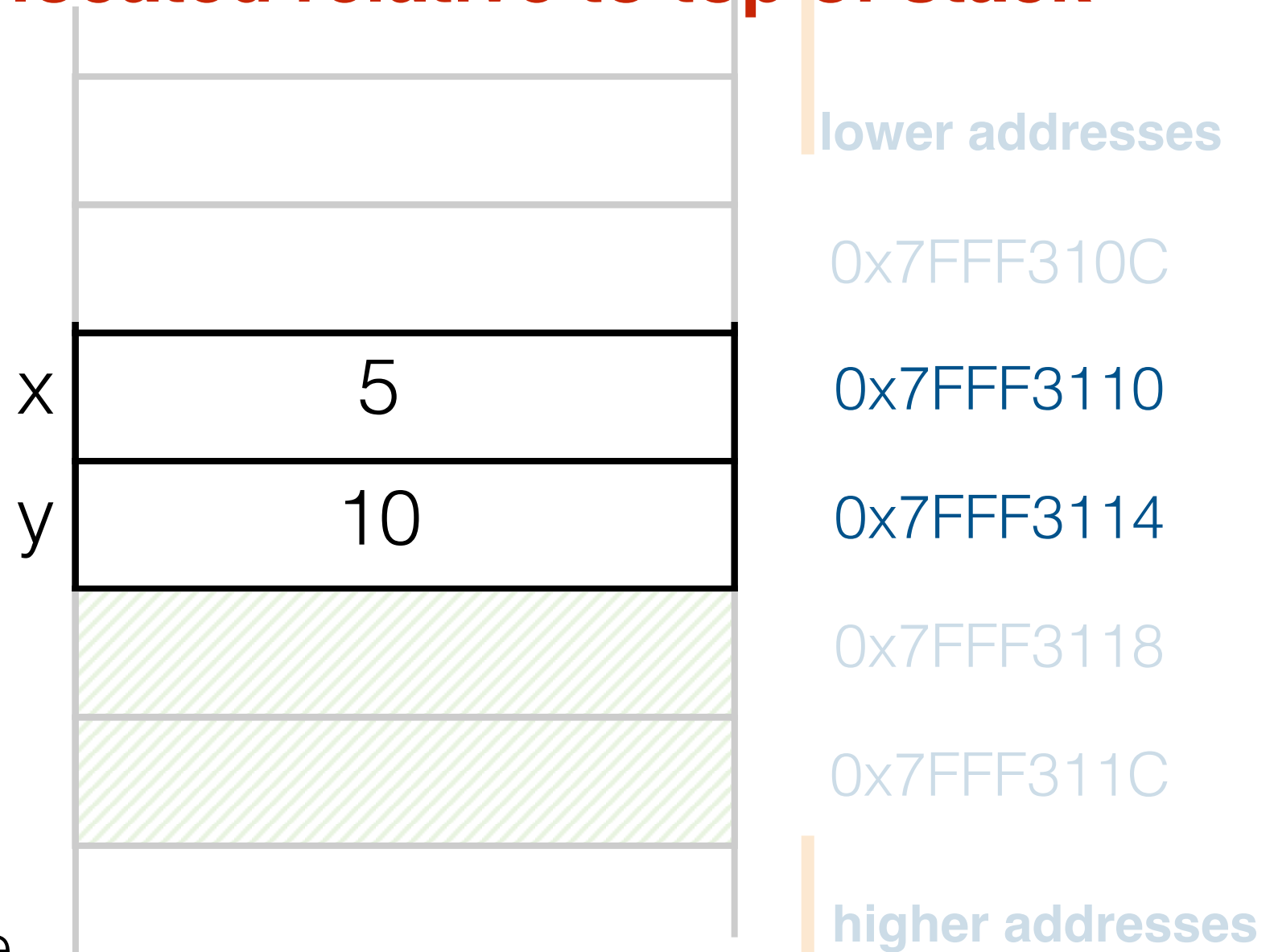
↓ **higher addresses**

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

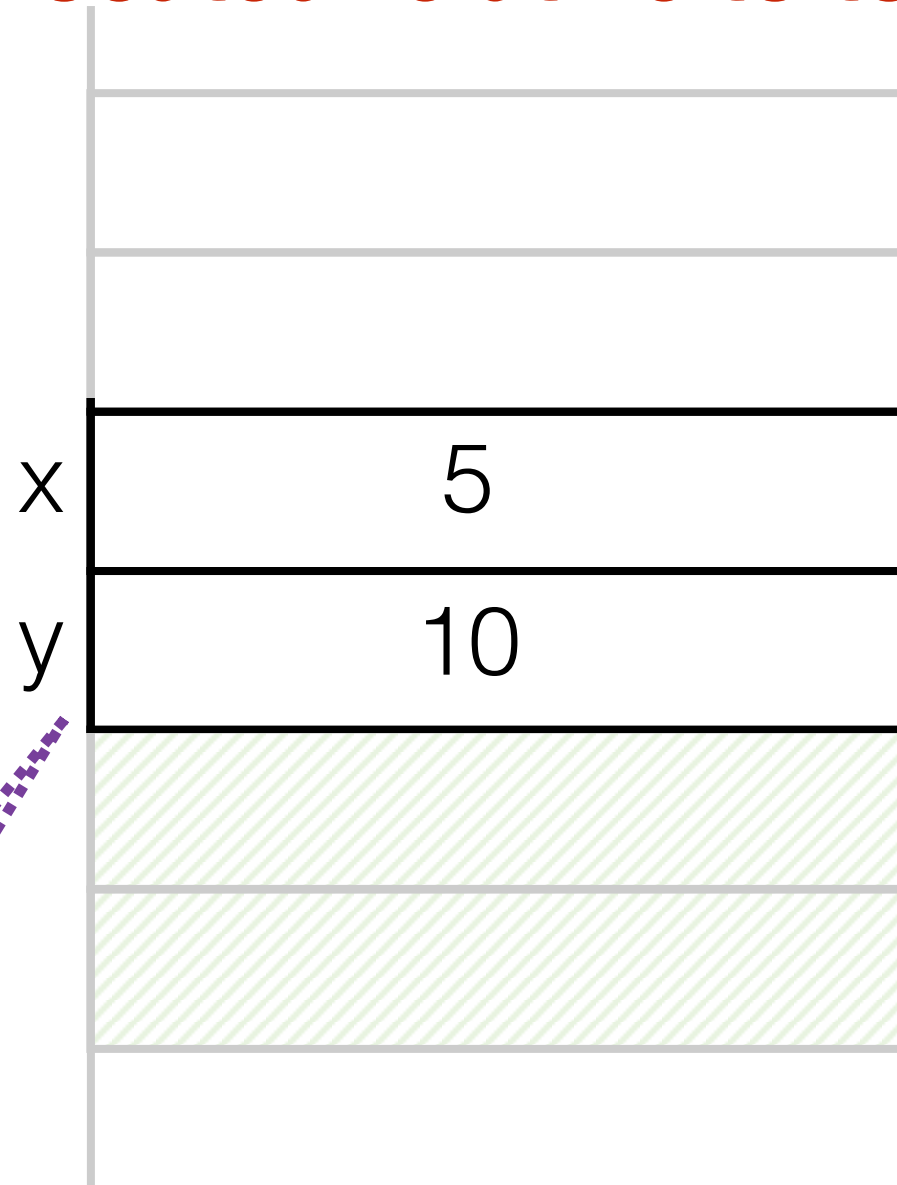
I can use `$sp` since variables are located relative to top of stack

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

`$sp` `0x7FFFB3110` →

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

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



lower addresses

`0x7FFF310C`

`0x7FFF3110`

`0x7FFF3114`

`0x7FFF3118`

`0x7FFF311C`

higher addresses

Examples of addressing modes

sw \$t0, 4(\$sp)

address is ($\$sp + 4$)

sw \$t0, -4(\$fp)

address is ($\$fp - 4$)

lw \$a0, 0(\$sp)

lw \$a0, (\$sp)



address is ($\$sp + 0$)

lw \$a0, var(\$zero)

lw \$a0, var



address is
($\$zero + \text{address of var}$)



- Can access local variables relative to stack pointer (**\$sp**).
- However, this may be problematic when passing arguments to **functions**:
 - Stack pointer moves to accommodate other function info
 - Relative locations of local variables change

Summary

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