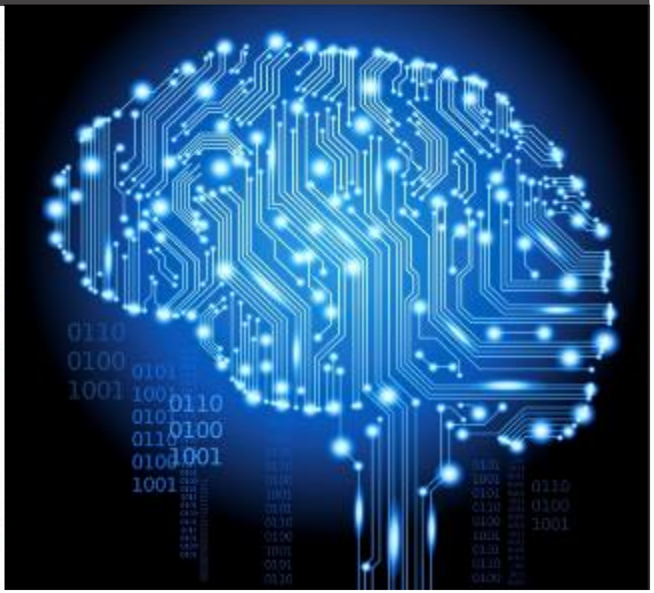
MONASH University

Information Technology

FIT1008/FIT2085 Lecture 7

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia

Working with Memory



Where are we at:

- **We have seen the basics of:**
  - MIPS architecture
  - MIPS Instruction set (the subset we will use)
  - Storing and accessing global variables
  - Compiling basic arithmetic, selection and loops into assembler
  - Creating and accessing arrays of integers

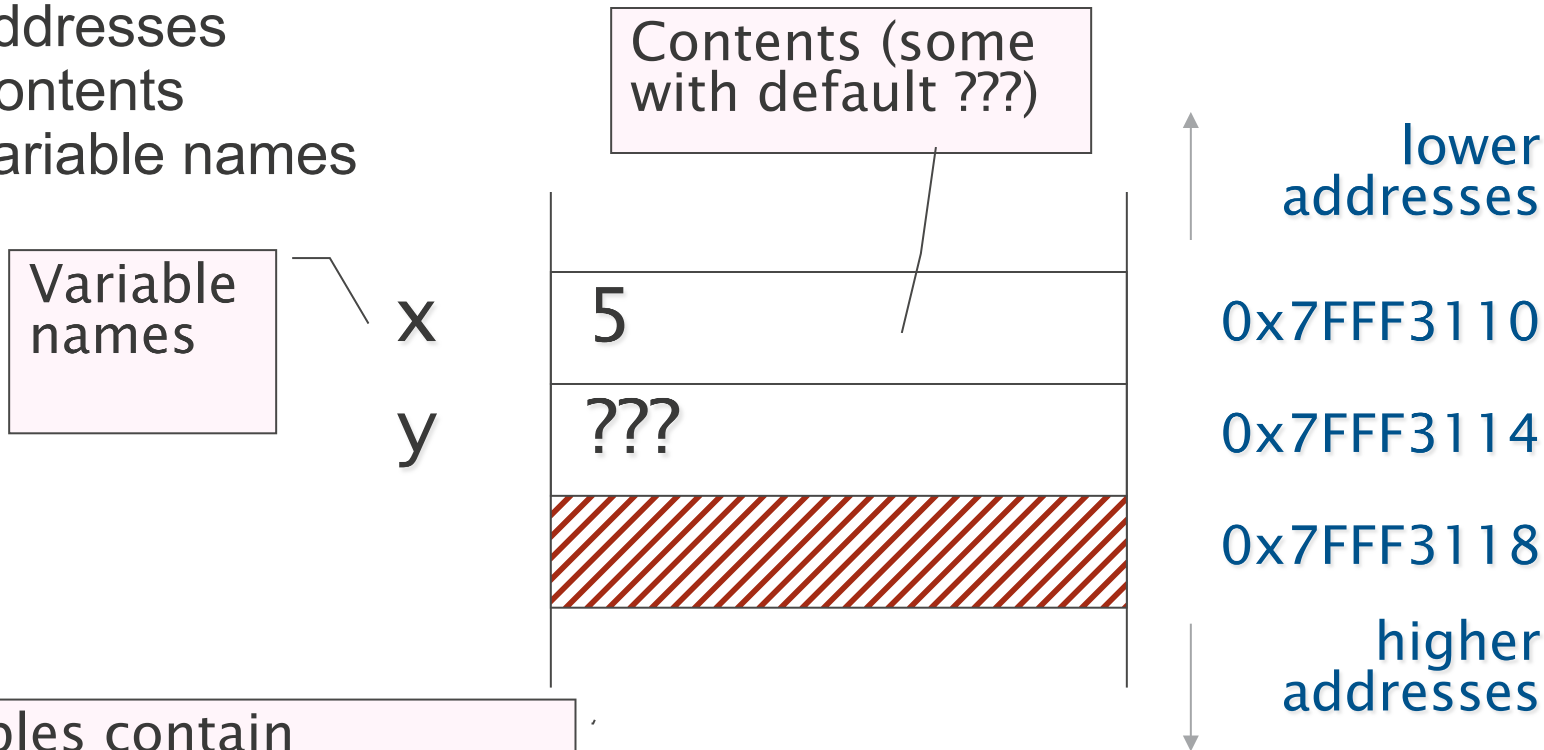
Learning objectives for this lecture:

- **To understand how to compile local variables in MIPS and why**
- **To achieve this we will discuss:**
  – 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

Memory diagrams

We are assuming numbers appear directly at the memory location (not true in Python, but true in C or Java) and occupy 4 bytes

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



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

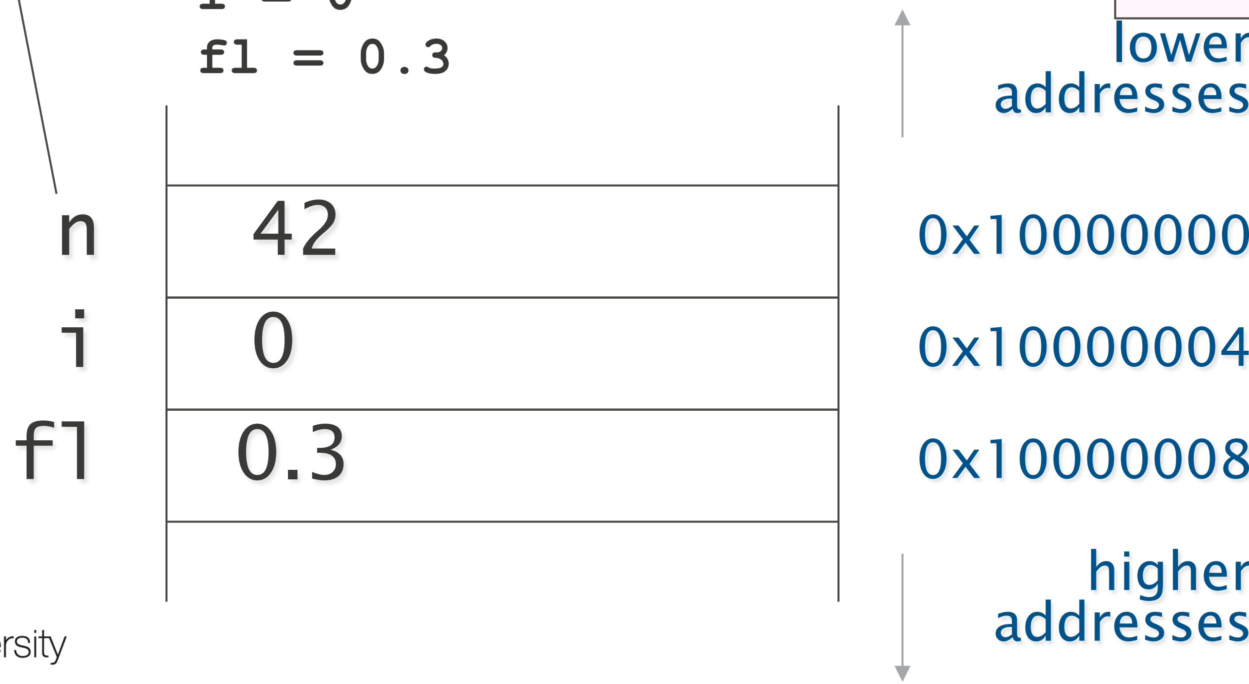
Memory diagrams: global variables

- **Not crucial for global variables (stored in data segment)**
  - Every variable has a label to identify it
  - This label is used to access its contents (`lw/sw`)

```
// global variables
n = 42
i = 0
fl = 0.3
```

start of data segment

global variable names … uniquely map to an address



Memory diagrams: local variables

- **Why do local variables not have a label?**
  - That is, why not store local variables in the data segment?
- **Think about the properties of data segment**
  - Accessible from all parts of the program
  - All labels must be different – they are unique
  - Each location can hold only one discrete value
- **Think about the properties of local variables**
  - Accessible only within a method/function
  - May have several vars with same name (different scopes)
    - A global and a local, or even several locals within a function (the latter is not possible in Python or JS; it is in C, Java…)
  - May have more than one version of the same function's variables (due to recursion)
- **So: data segment not suited for local variables**
- **But then, where will we store them?**

Actually, within a "block", which might be a loop, if-then-else, etc

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
```

It is a stack!

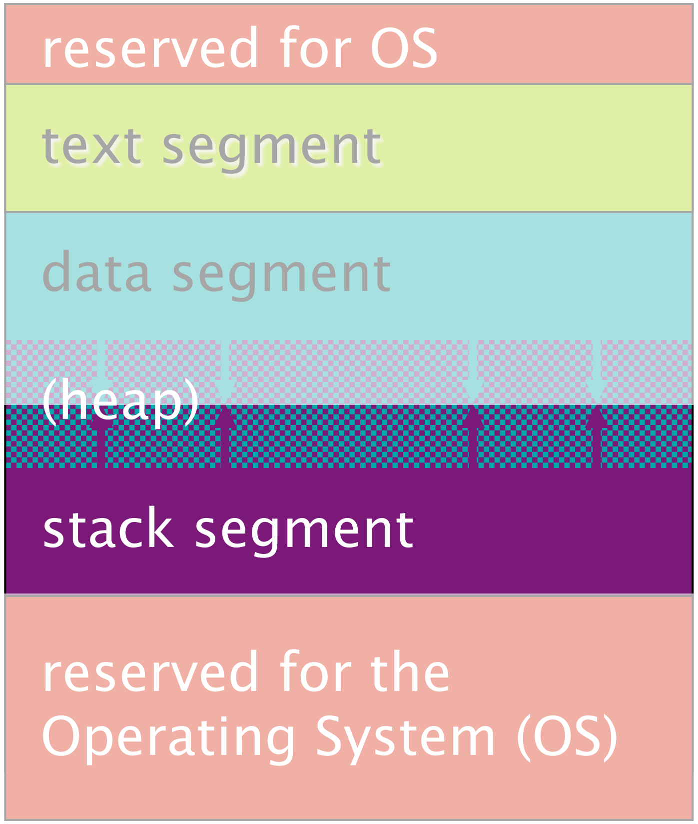
- **A data type that follows LIFO: Last In First Out**
- **Adding an element: push**
  - The element is added at the top of the stack
- **Deleting an element: pop**
  - The element is popped from the top of the stack
- **An element can only be accessed if it is at the top of the stack**

Properties of local variables (cont)

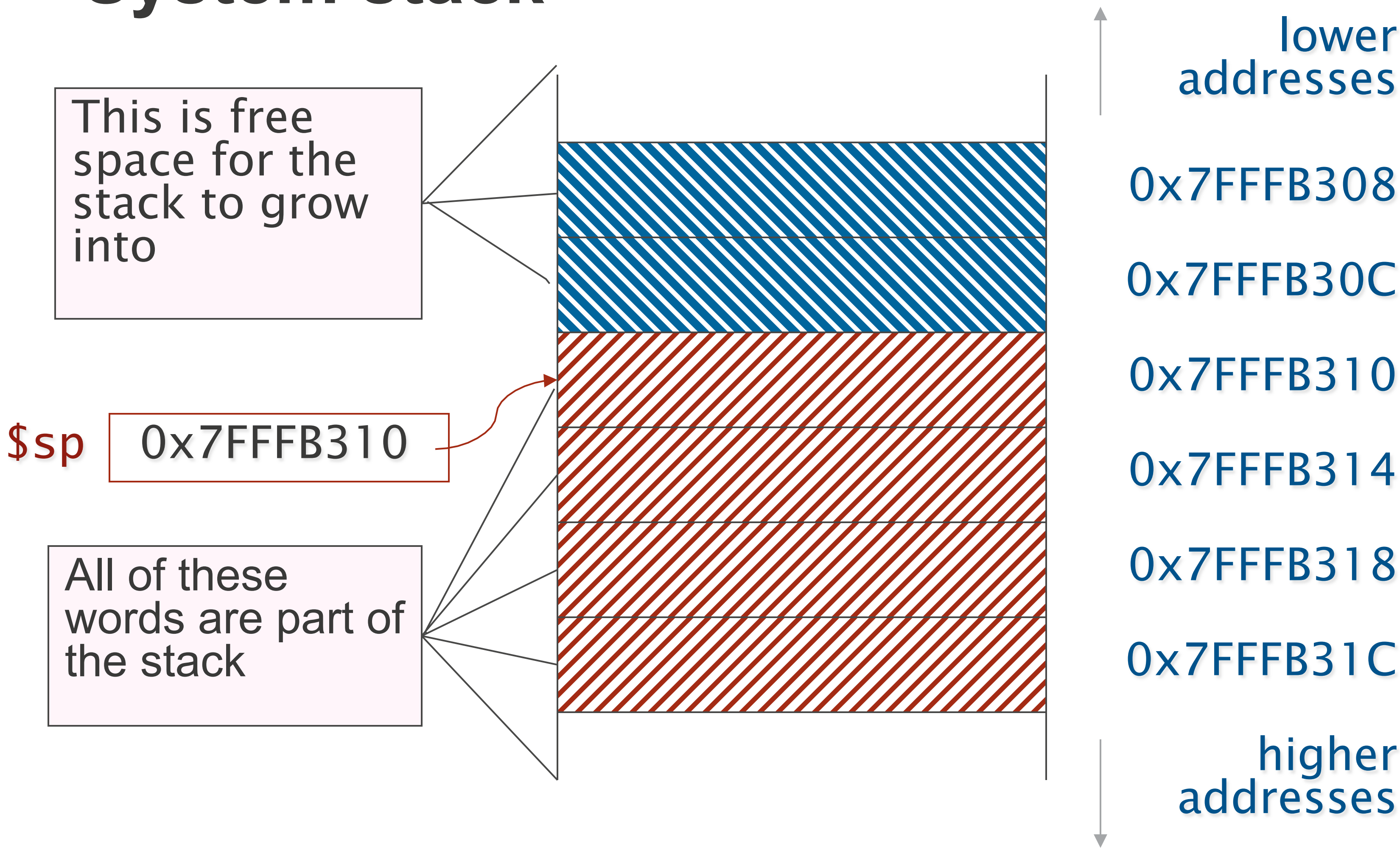
- **Allocation/deallocation of local variables obeys LIFO**
  - The last allocated is the first to be deallocated
- **A stack data structure is ideal for storing them**
  - Allocate a variable by pushing it on the stack
  - Deallocate a variable by popping it off the stack
- **Also helpful for storing other function related info**
- **Thus, most computers provide a memory stack for programs to use:**
  - Called system stack or runtime stack or process stack
  - Initialized by operating system
  - User programs push/pop the system stack as needed
  - The instruction set provides operations for doing this

System stack in MIPS

- **Has its own segment of memory**
  - Stack segment: to address 0x7FFFFFFF (0x80000000 is OS)
- **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

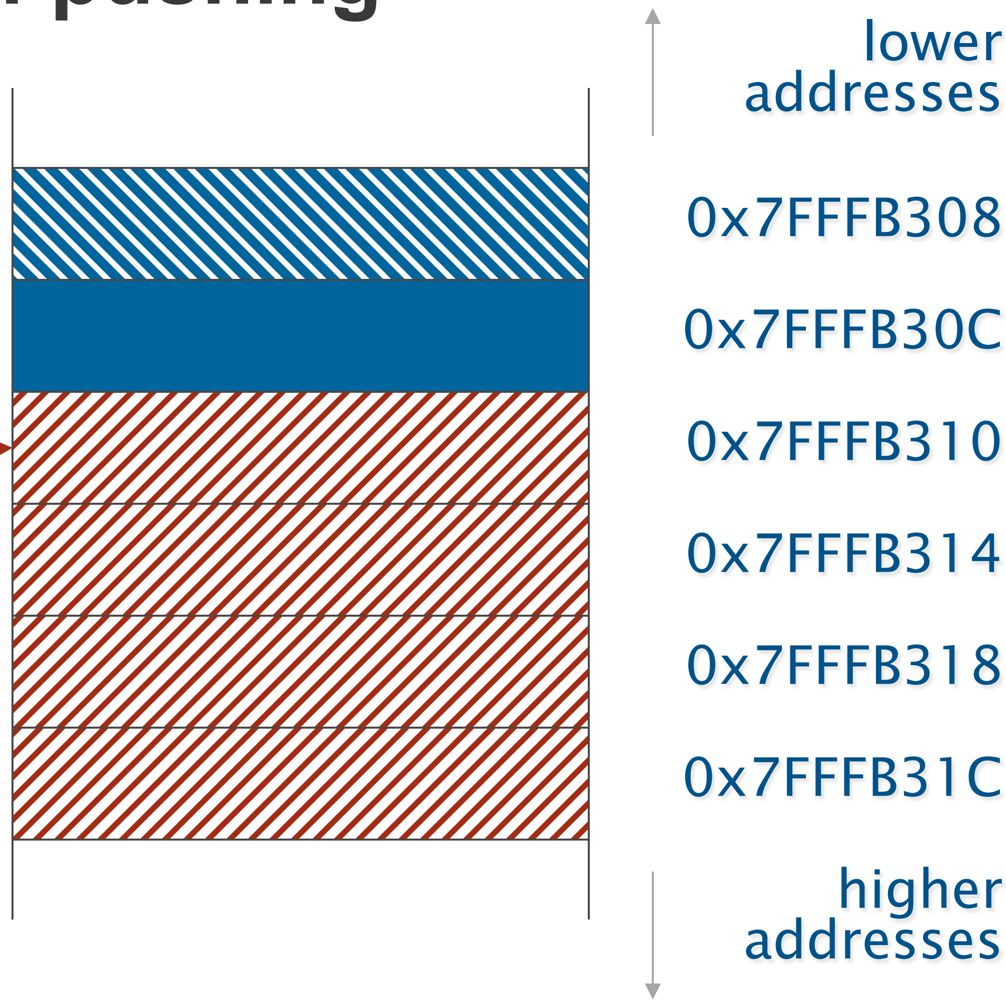


System stack: pushing

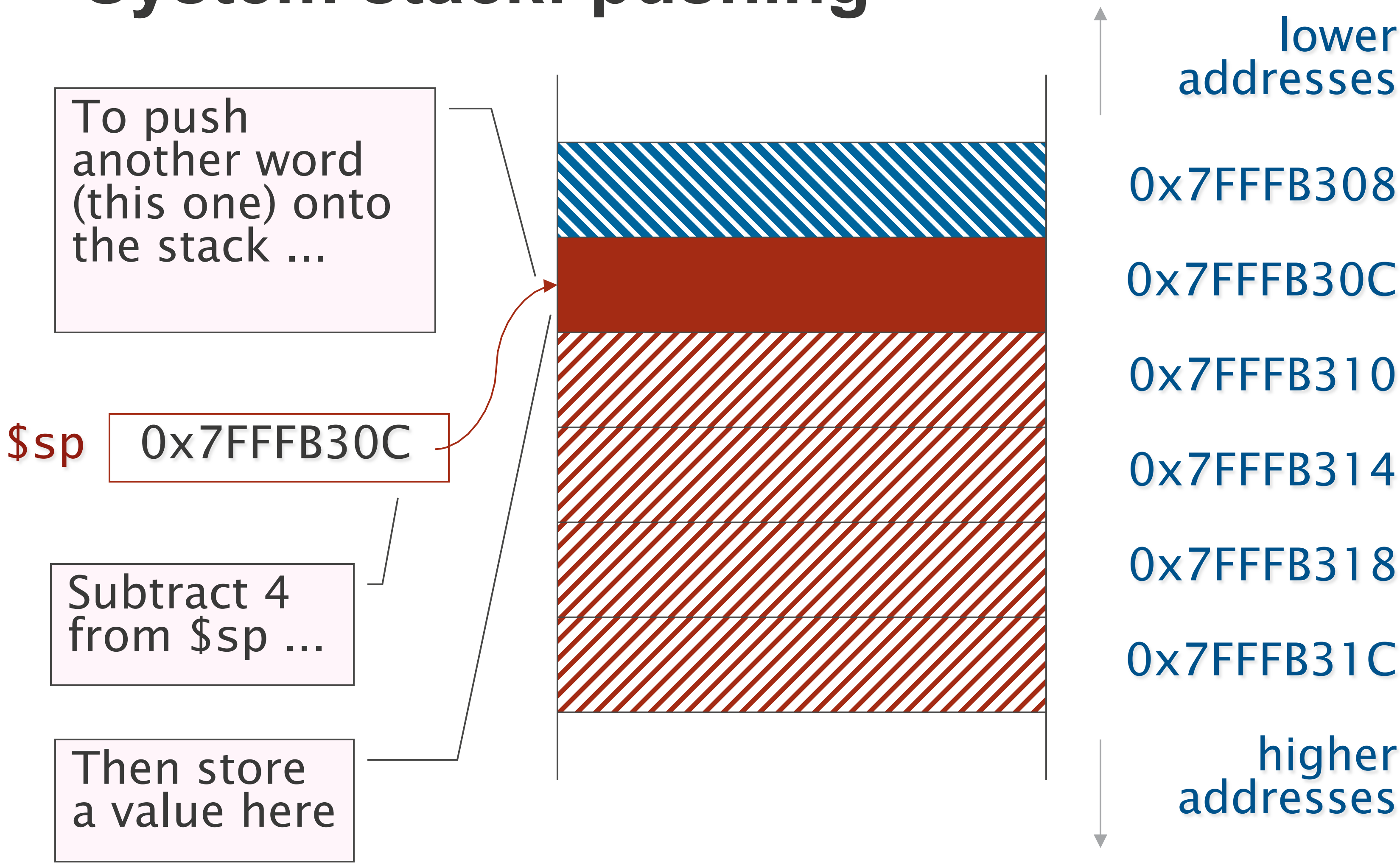
To push another word (this one) onto the stack ...

$sp 0x7FFFB310

Subtract 4 from $sp ...



System stack: pushing



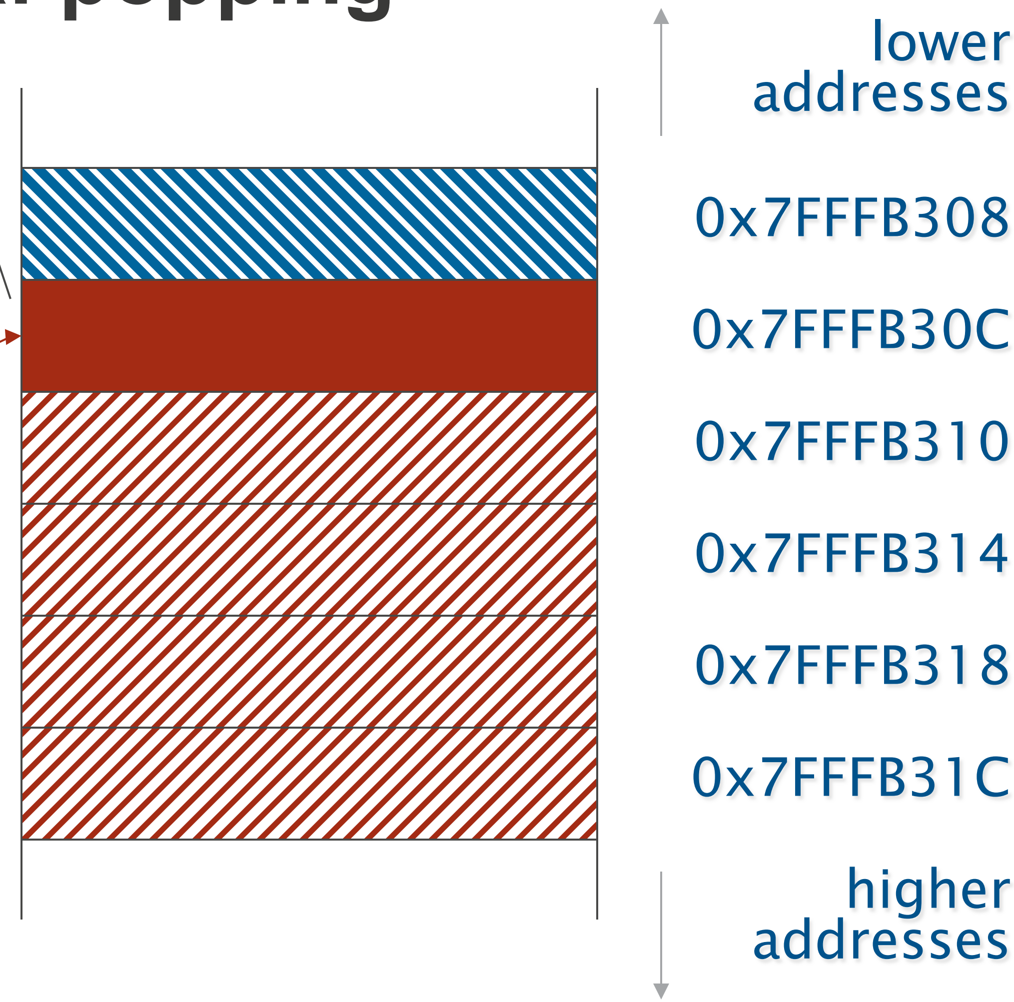
System stack: popping

To pop a word (this one) off the stack ...

... fetch this word into a register ...

$sp 0x7FFFB30C

... then add 4 to $sp



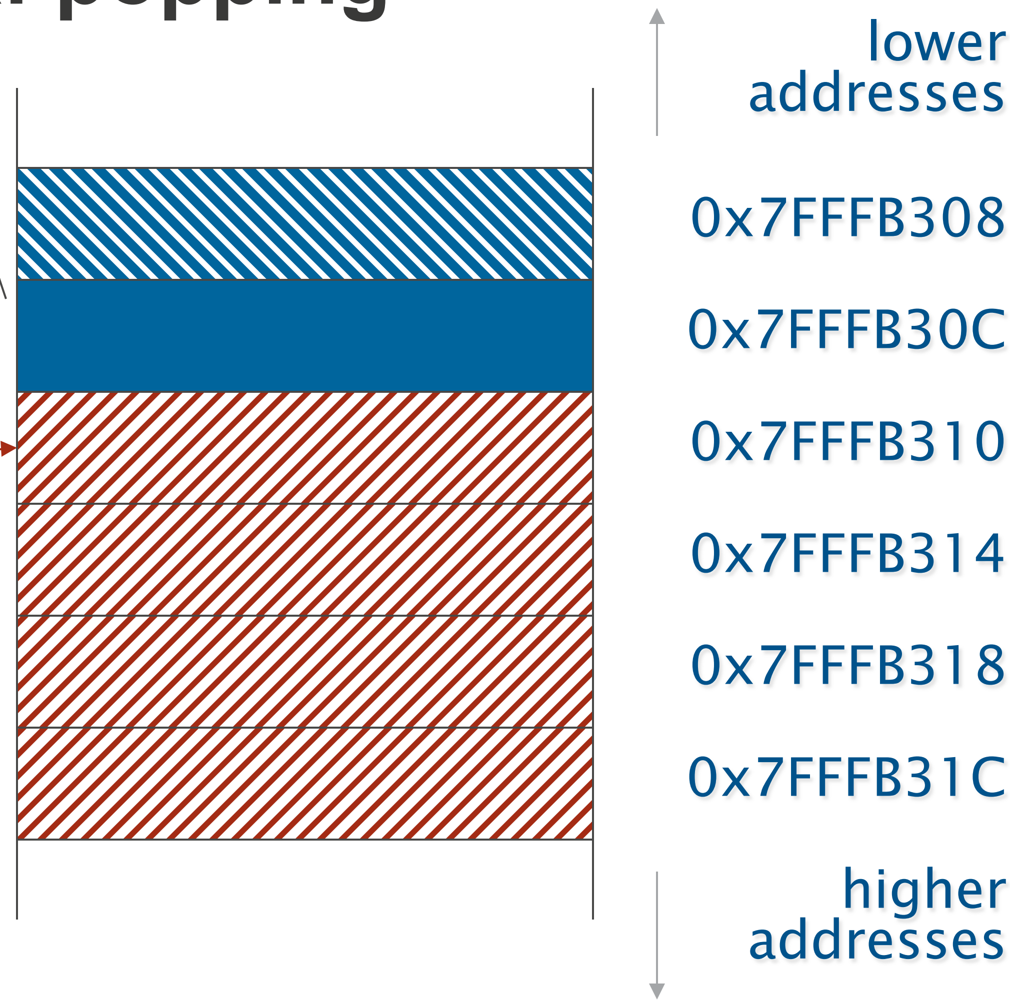
System stack: popping

To pop a word (this one) off the stack ...

... fetch this word into a register ...

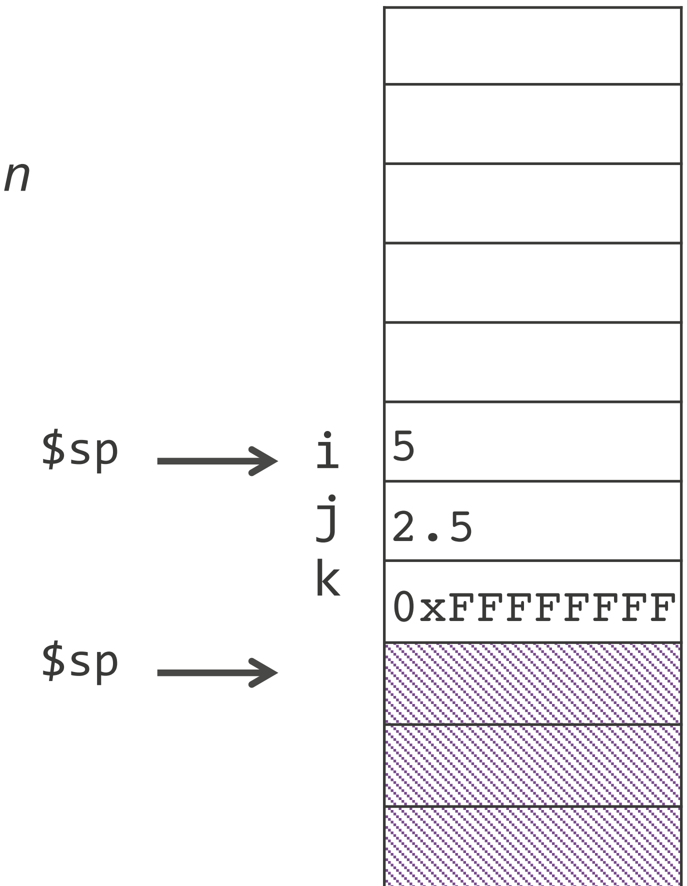
$sp 0x7FFFB310

... then add 4 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 `$sp`)
  - Initialize space by storing values in newly allocated space
- **During function**
  - Use variables using lw/sw
- **At the end of the function**
  - Deallocate variables by popping allocated space from stack (add *n* bytes to `$sp`)



Not necessary on exit from **`main`** since program is ending

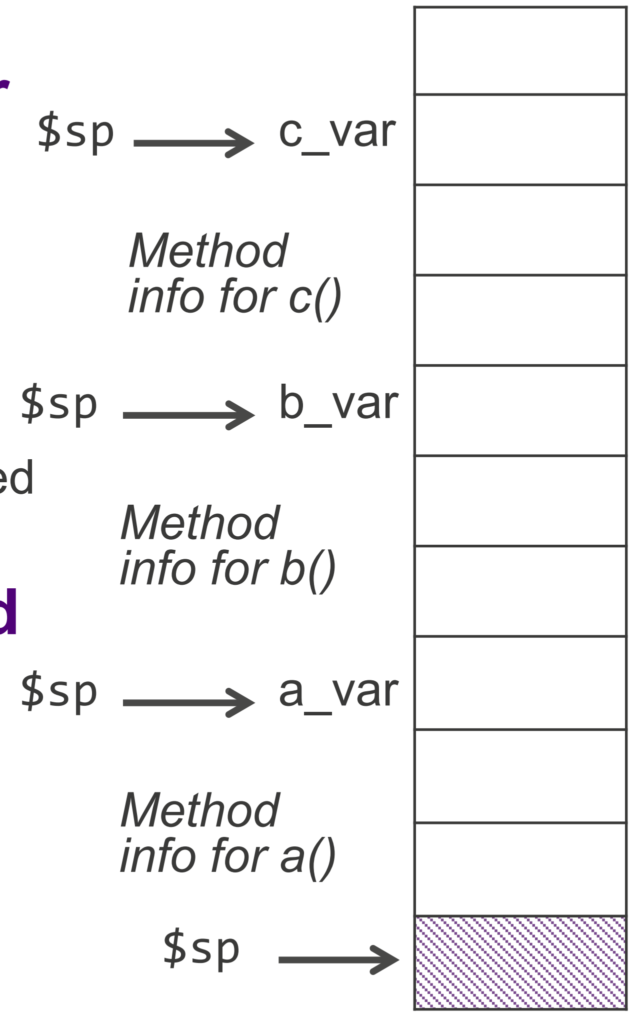
Example:

```
def a():
    a_var = 0
    b()
```

```
def b():
    b_var = 0
    c()
```

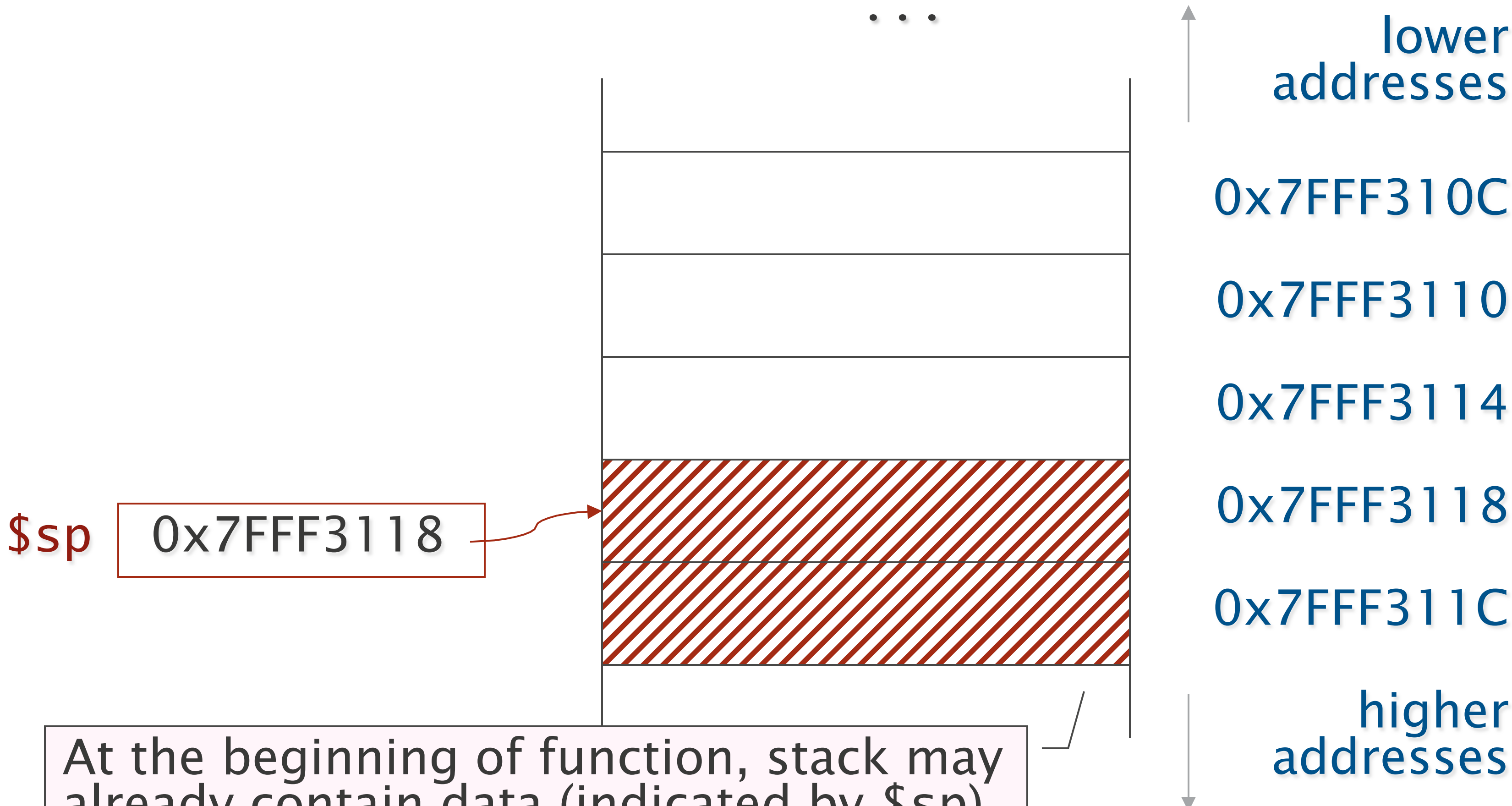
```
def c():
    c_var = 0
    ...
```

- **Method a() creates a_var**
- **a() calls b()**
  - b() creates b_var
  - b() calls c()
    - c() creates c_var
    - c() exits; c_var is deleted
  - b() exits; b_var is deleted
- **a() exits; a_var is deleted**



Example

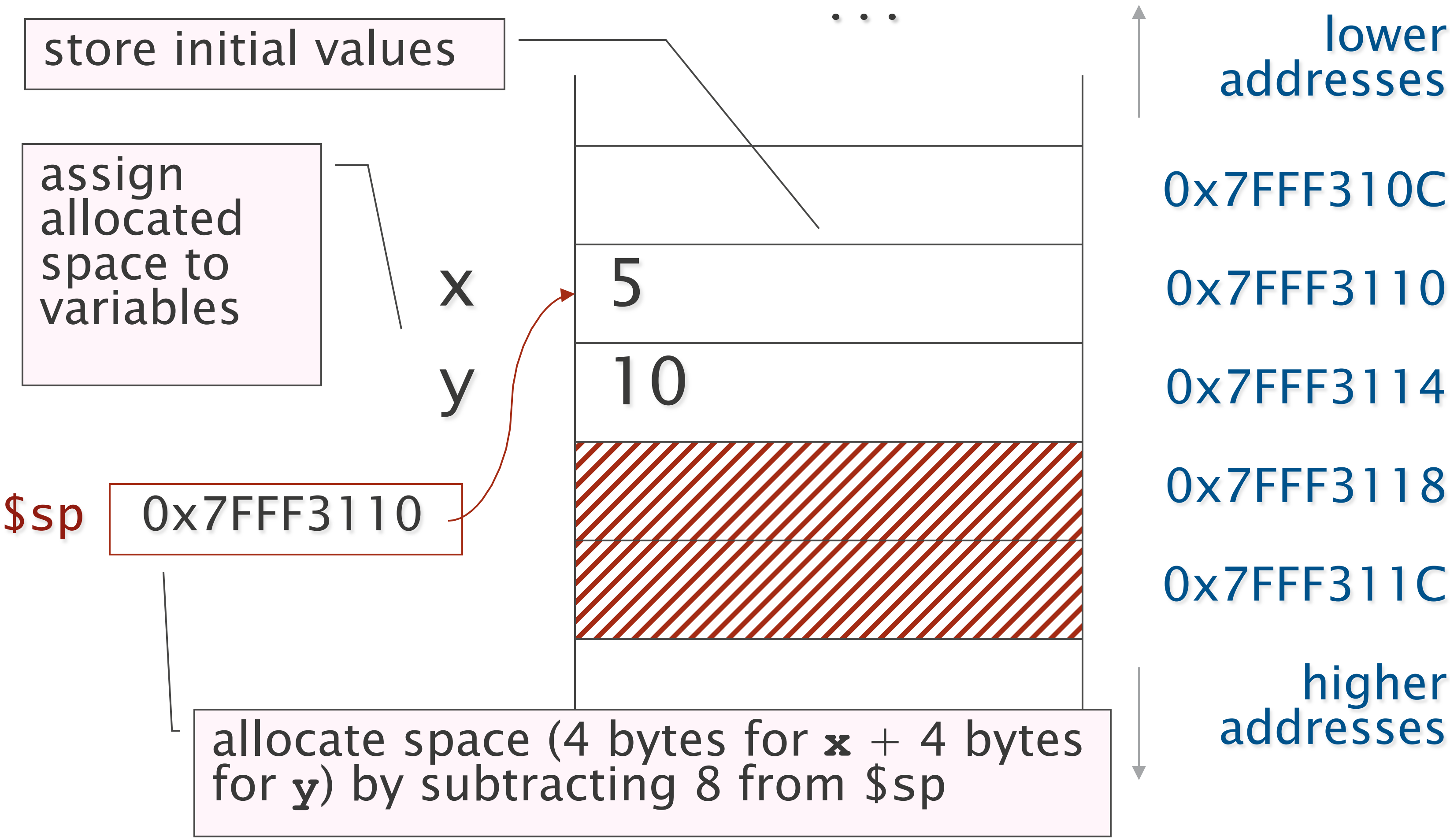
```
def main():
    x = 5
    y = 10
    ...
```



At the beginning of function, stack may already contain data (indicated by $sp)

Example

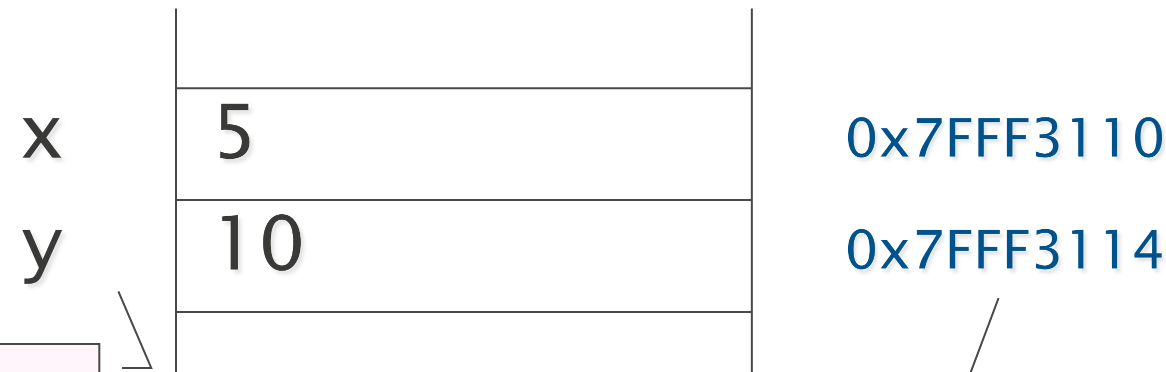
```
def main():
    x = 5
    y = 10
    ...
```



MONASH University

Memory diagram: local variables

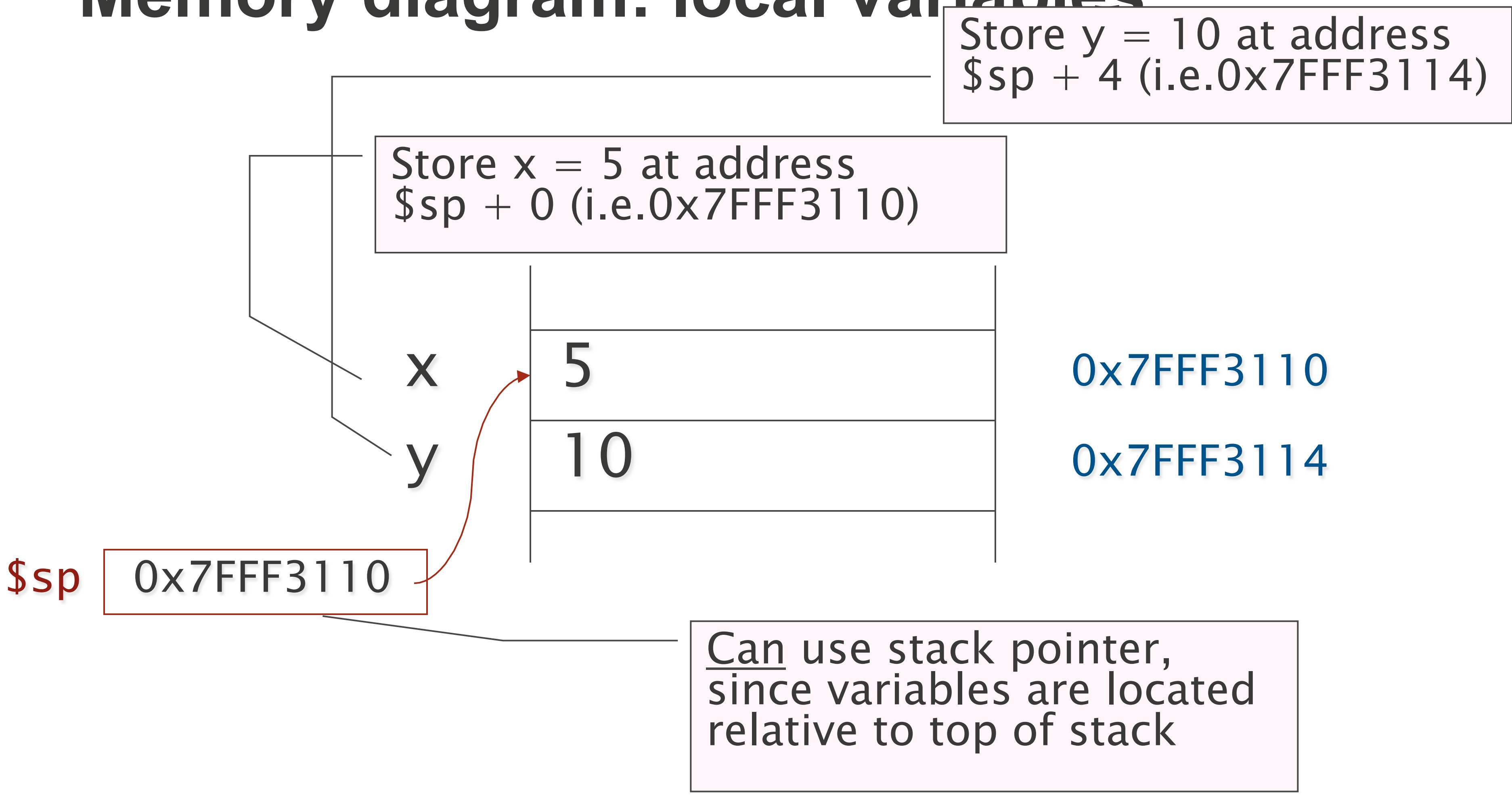
- **How do we use them?**
  - To refer to local variables. But how?



Can't refer to location via label because local variable labels are not static and unique (only make sense at compile-time not run-time).

Can't refer to location via address because stack may not be same depth every time

Memory diagram: local variables



Reminder: addressing modes

This syntax means "use the address computed by adding `const` to the current contents of `$reg`" (i.e., `$reg + const`)

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

`const` may be any label or signed number or expression known at compile time, including 0

`$reg` may be any general-purpose register, including $0

MONASH University

Examples of addressing modes

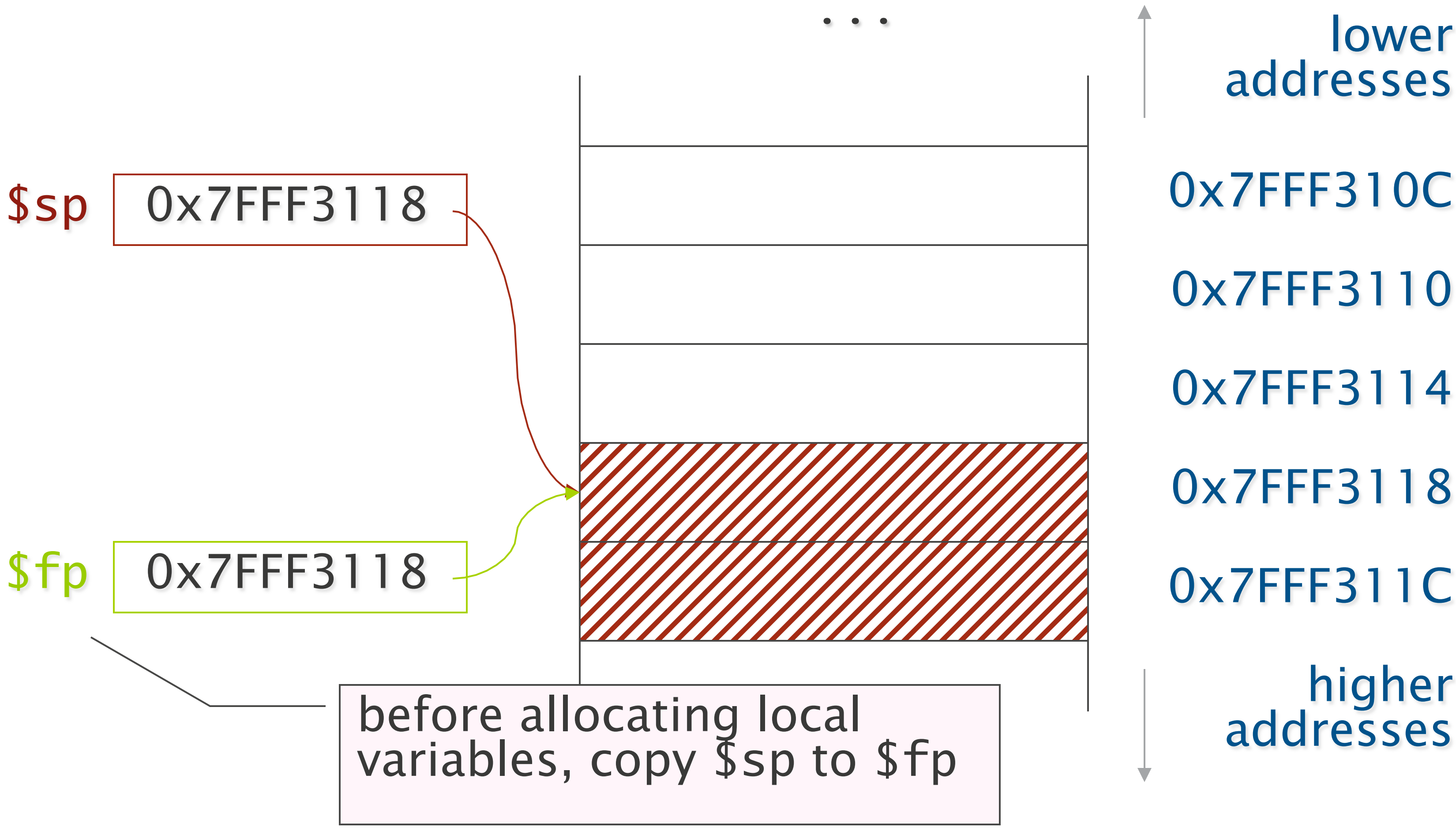
`sw $t0, 4($sp)`	address is ($sp + 4)
`sw $t0, -4($fp)`	address is ($fp - 4)
`lw $a0, 0($sp)` `lw $a0, ($sp)`	address is ($sp + 0)
`lw $a0, var($zero)` `lw $a0, var`	address is ($zero + address of var)

Frame pointer

- **Can access local variables relative to stack pointer (`$sp`), but …**
- **Can be problematic when passing arguments to functions**
  - Stack pointer moves to accommodate other function info
  - Relative locations of local variables change
- **Better to access local variables relative to saved copy of stack pointer**
  - Copy made before subtracting from `$sp` to allocate local variables
- **Saved copy stored in register `$fp` (frame pointer)**
  - Local variables accessed relative to `$fp`

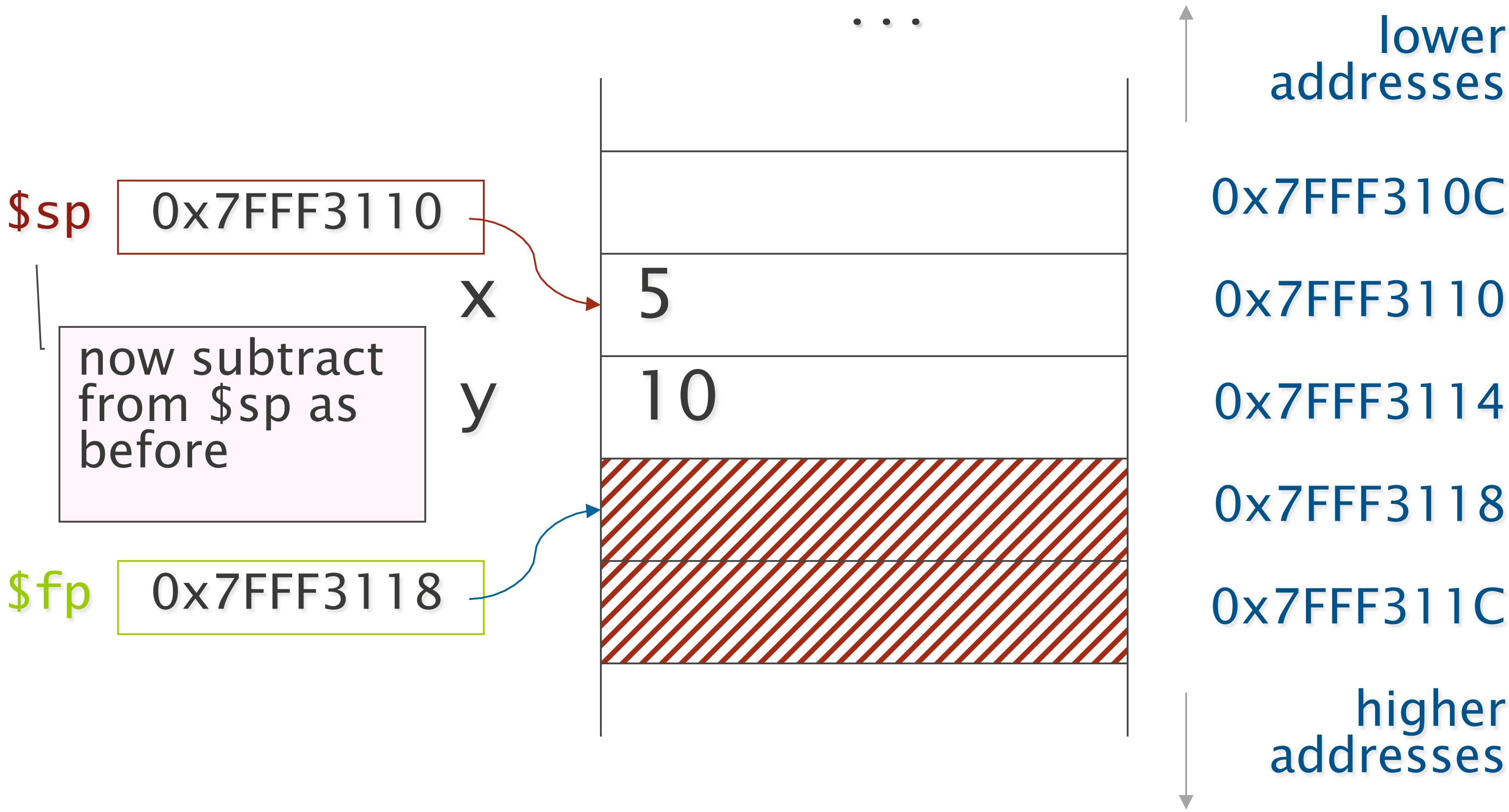
Local variables

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



Local variables

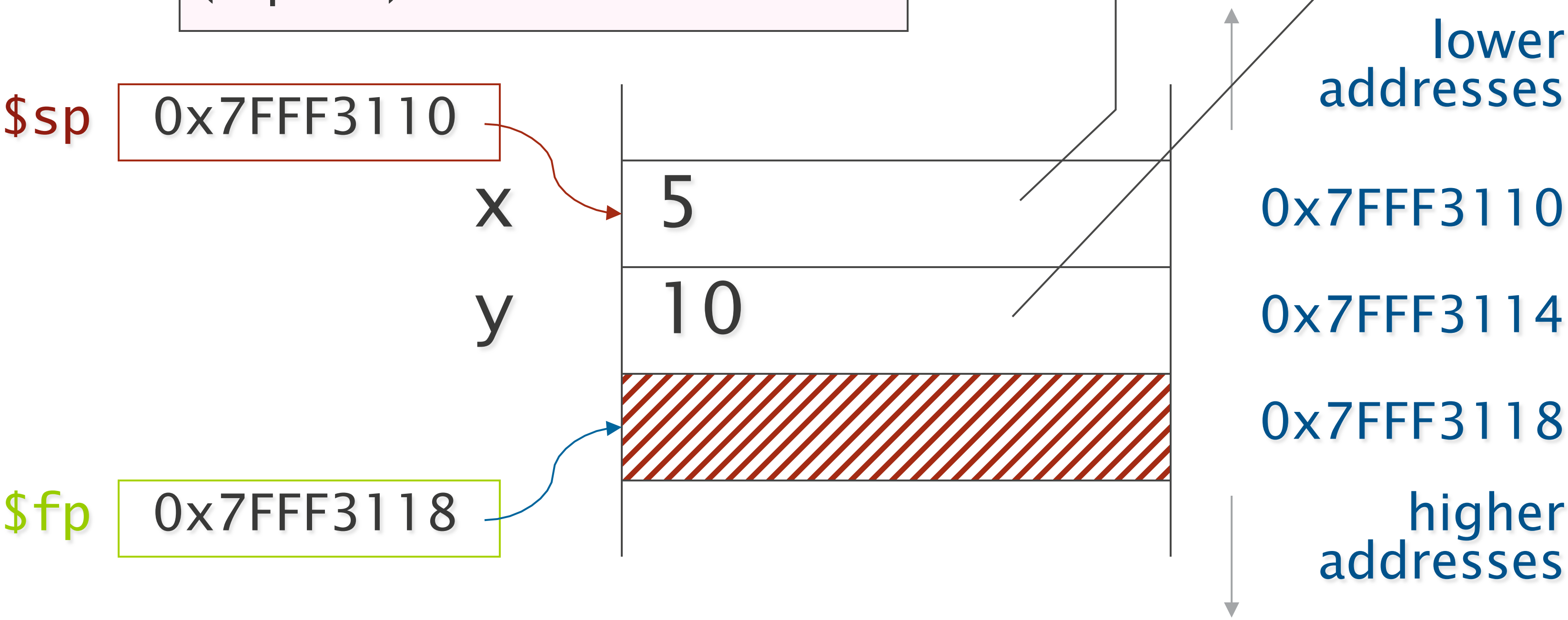
```
def main():
    x = 5
    y = 10
    ...
```



Local variables

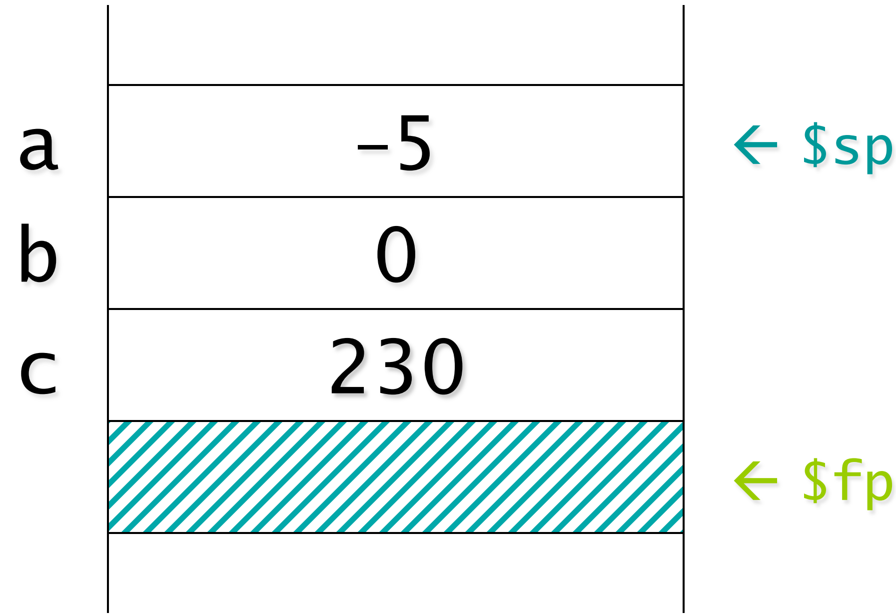
access y at address ($fp – 4) = 0x7FFF3114

access x at address ($fp – 8) = 0x7FFF3110



```
// A global variable
g = 123

def main():

   // Three local variables
   a = -5
   b = 0
   c = 230

   // Do some arithmetic
   b = g + a

   // Do some more arithmetic
   print(c - a)
```

g is a global variable and is stored in data segment, not on stack

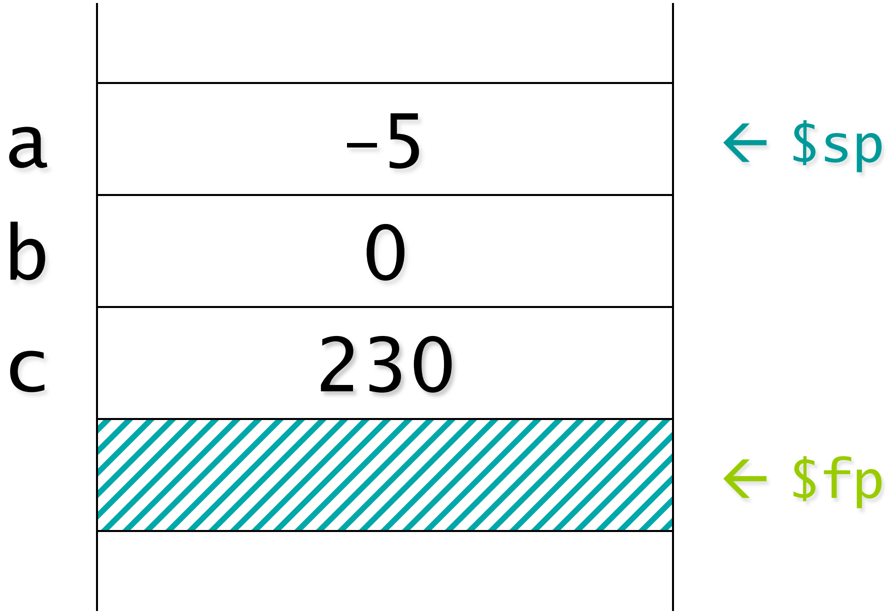


This memory diagram corresponds to this program point

a is at -12($fp)

b is at -8($fp)

c is at -4($fp)




```
        .data
# g is global, allocate
# in data segment
g:      .word 123

        .text
main:   # Copy $sp into $fp.
        addi $fp, $sp, 0

        # Allocate 12 bytes of
        # local variables.
        addi $sp, $sp, -12

        # Initalize local
        # variables.

        addi $t0, $0, -5      # a
        sw $t0, -12($fp)

        sw $0, -8($fp)        # b

        addi $t0, $0, 230     # c
        sw $t0, -4($fp)

        # ... rest of program
        # follows next slide ...
```

When compiling to MIPS I want you to…

- **Draw memory diagrams for local variables**
  - Since they are referred to without names in MIPS
  - Therefore, remembering their address is vital
- **Be "faithful":**
  - Translate each line of code independently of the others (i.e., without reusing the value of registers computed in previous instructions)
  - More lines, but less mistakes…
- **Comment appropriately:**
  - Each block corresponding to a line of Python code
  - Often each line (not for `syscalls`, but yes for most other blocks)

a is at -12($fp)

b is at -8($fp)

c is at -4($fp)

```
// A global variable
g = 123

def main():

    // Three local variables

    a = -5
    b = 0
    c = 230

    // Do some arithmetic
    b = g + a

    // Do some more arithmetic
    print(c - a)
```

```
        .data
# g is global, allocate
# in data segment
g:      .word 123

        .text
main:   # Copy $sp into $fp.
        addi $fp, $sp, 0

        # Allocate 12 bytes of
        # local variables.
        addi $sp, $sp, -12

        # Initalize local
        # variables.
        addi $t0, $0, -5     # a
        sw $t0, -12($fp)

        sw  $0, -8($fp)      # b

        addi $t0, $0, 230    # c
        sw $t0, -4($fp)
        # ... rest of program
        # follows next slide ...
```

a is at -12($fp)

b is at -8($fp)

c is at -4($fp)

Faithful translation: registers for g and a are not reused, they are re-loaded

```
// A global variable
g = 123

def main():

  // Three local variables
  a = -5
  b = 0
  c = 230

  // Do some arithmetic
  b = g + a

  // Do some more arithmetic
  print(c - a)
```

```
# ... here is the rest
# of the MIPS code ...

# b = g + a.
lw $t0, g              # g
lw $t1, -12($fp)       # a
add $t0, $t0, $t1      # g+a
sw $t0, -8($fp)        # store in b

# print(c-a)
addi $v0, $0, 1        # Print int
lw $t0, -4($fp)        # c
lw $t1, -12($fp)       # a
sub $a0, $t0, $t1      # c-a
syscall                # Do print.

# Now exit.
addi $v0, $0, 10     # Exit.
syscall

# If this function was not main
# it would need to deallocate
# local variables with:
# addi $sp, $sp, 12
```

Recap: Global vs Local variables

- **Names of global variables appear in assembly code:**

```
lw $t0, g
```

- **Names of local variables do not**
- **Instead, they are accessed with negative offset from frame pointer:**

```
lw $t0, -4($fp)
```

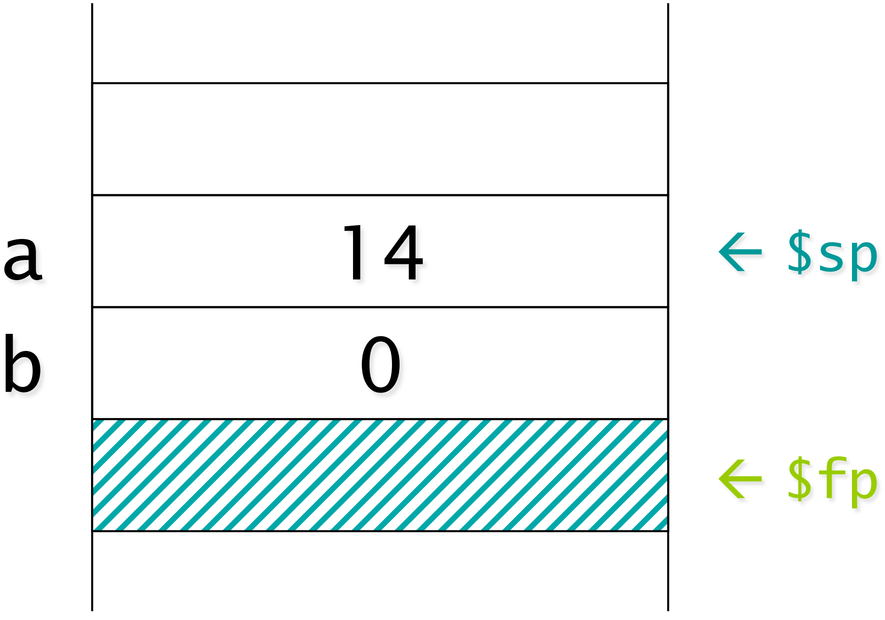
  - Offset will be positive for function parameters (later)

- **Thus, it is important to:**
  - Comment code
  - Draw stack memory diagram to know correct addresses

```
// A global variable
n = 4

def main():

   // Two local variables

   a = 14
   b = 0

   // Do some arithmetic
   b = (n * a) - 7

   // Do some more arithmetic
   b = b / 16

   // Do even more arithmetic
   print(b + n)
```

a is at -8($fp)

b is at -4($fp)



```
// A global variable
n = 4

def main():

  // Two local variables

  a = 14
  b = 0
  // Do some arithmetic
  b = (n * a) - 7

  // Do some more arithmetic
  b = b / 16

  // Do even more arithmetic
  print(b + n)
```

```
        .data
# allocate global n in data segment
n:      .word 4
        .text
main:# Copy $sp into $fp.
     addi $fp, $sp, 0

     # Allocate local variables
     addi $sp, $sp, -8

     # Initalize local variables
     addi $t0, $0, 14     # a
     sw $t0, -8($fp)
     sw $0, -4($fp)       # b

     # b = (n*a)-7.
     lw $t0, n            # n
     lw $t1, -8($fp)      # a
     mult $t0, $t1        # n*a
     mflo $t0
     addi $t0, $t0, -7    # (n*a)-7
     sw $t0, -4($fp)      # b=(n*a)-7
     # ... rest of program
     # follows next slide ...
```

a is at -8($fp)

b is at -4($fp)

```
// A global variable
n = 4

def main():

   // Two local variables
   a = 14
   b = 0

   // Do some arithmetic
   b = (n * a) - 7

   // Do some more arithmetic
   b = b / 16

   // Do even more arithmetic
   print(b + n)
```

a is at -8($fp)

b is at -4($fp)

```
# ... here is the rest
# of the MIPS code ...

# b = b/16
lw $t0, -4($fp)      #b
sra $t0, $t0, 4      #b/16
sw $t0, -4($fp)      #b = b/16

# printInt(b+n)
addi $v0, $0, 1       # Print int
lw $t0, -4($fp)       # b
lw $t1, n             # n
add $a0, $t0, $t1     # b+n
syscall               # Do print.

# Now exit.
addi $v0, $0, 10     # Exit.
syscall

# If this function was not main
# it would need to deallocate
# local variables with:
# addi $sp, $sp, 8
```

Summary

- **Memory diagrams**
- **System stack**
  - Pushing and popping
  - `$sp` and `$fp`
- **Local variables**
  - Stored on stack
  - Accessed with negative offset from `$fp`
- **Addressing modes**
  - Register + constant