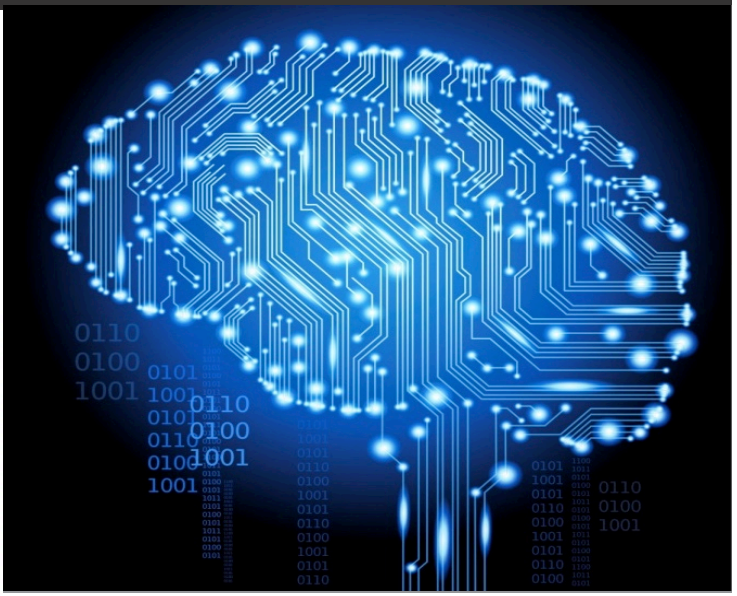
MONASH University

Information Technology

FIT1008/2085
MIPS – Local Variables

Prepared by:
Maria Garcia de la Banda
Revised by D. Albrecht, J. Garcia



Where are we up to?

- Know the MIPS R2000 architecture and can name the main parts
- Understand the fetch-decode-execute cycle
- Able to use assembler directives
- Can program in assembly using the MIPS instruction set we will use
- Know the basics of the instruction formats
- Able to translate simple programs with if-then-elses and loops
- Able to create and access arrays of integers in memory
- Know how to translate from high-level to MIPS faithfully

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

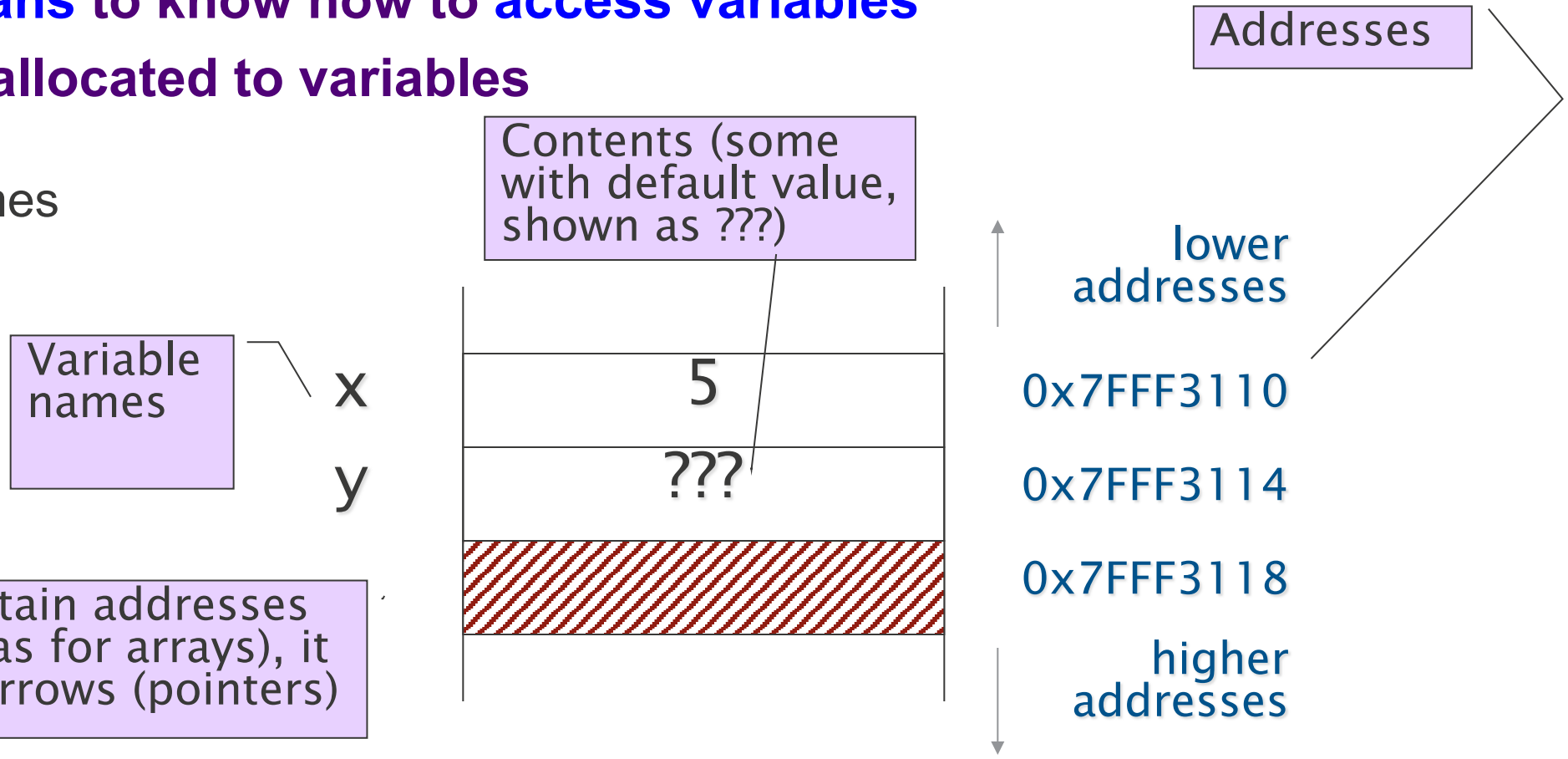
MONASH
University

Need for Memory Diagrams

Memory diagrams

Remember: 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
  - Variable names
  - Contents

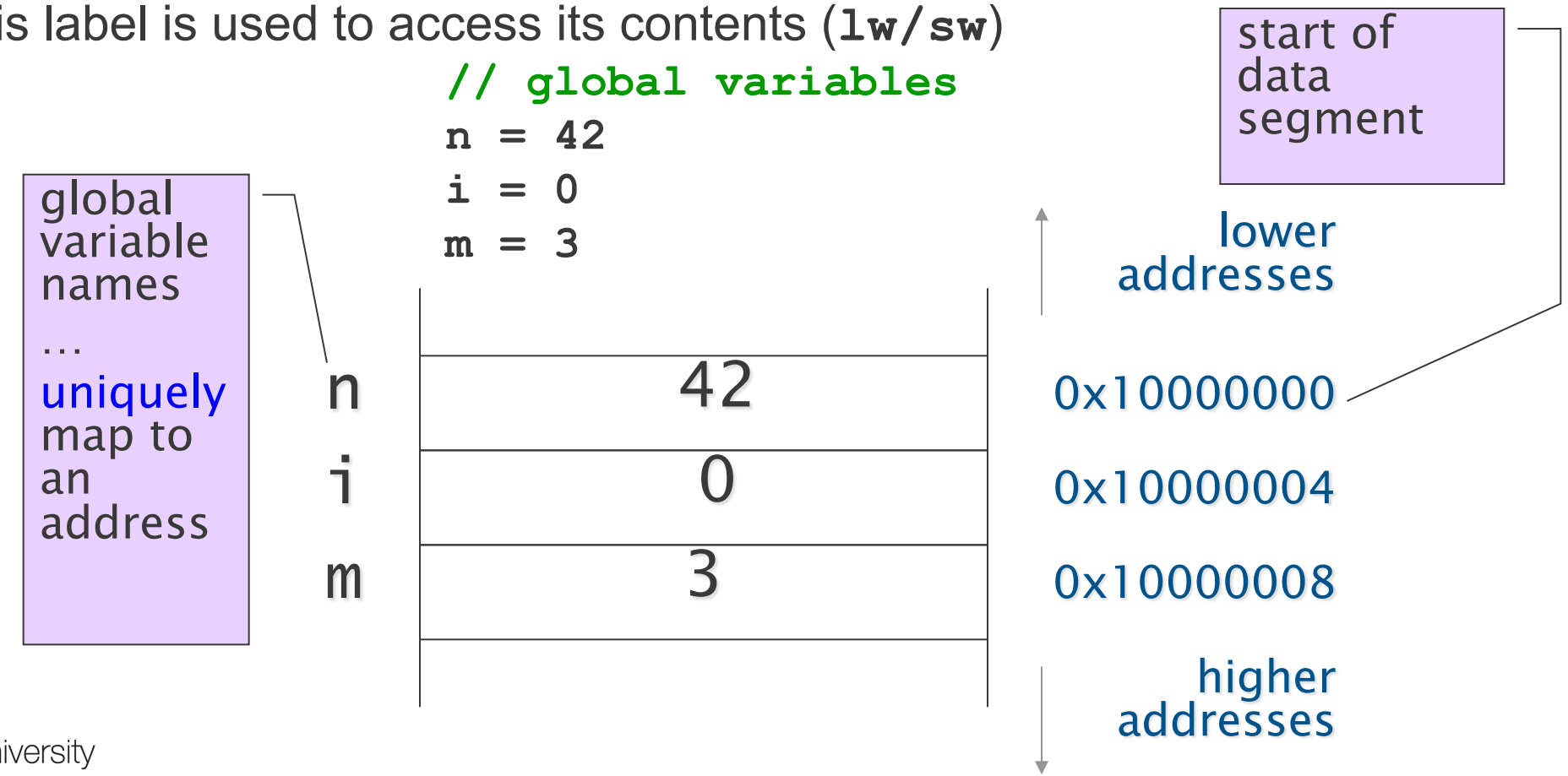


When variables contain addresses of other variables (as for arrays), it is helpful to draw arrows (pointers)

Memory diagrams: global variables

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

```
// global variables
n = 42
i = 0
m = 3
```



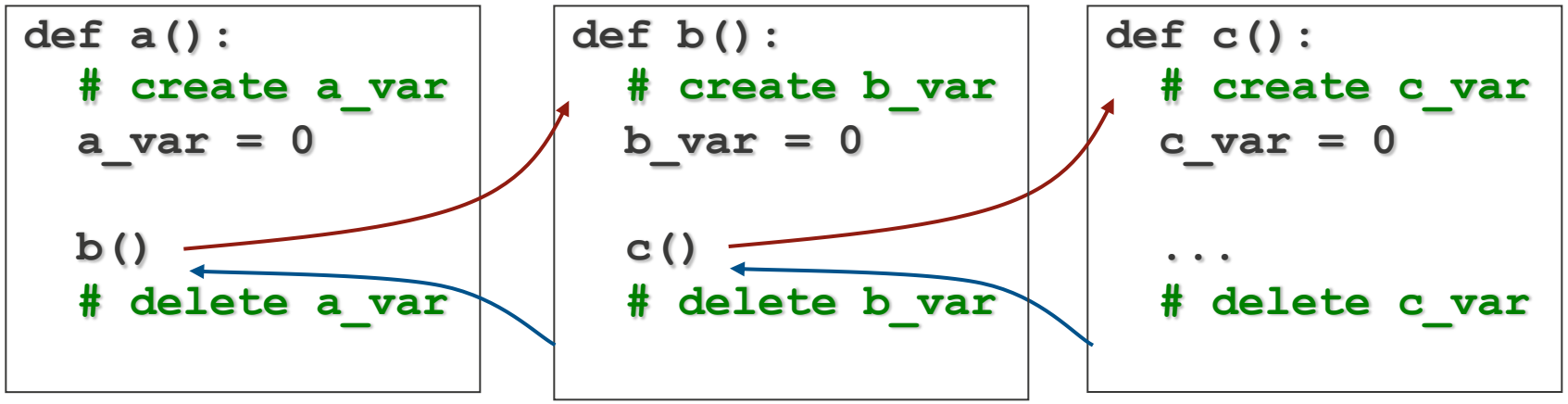
Memory diagrams: local variables

- **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



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)

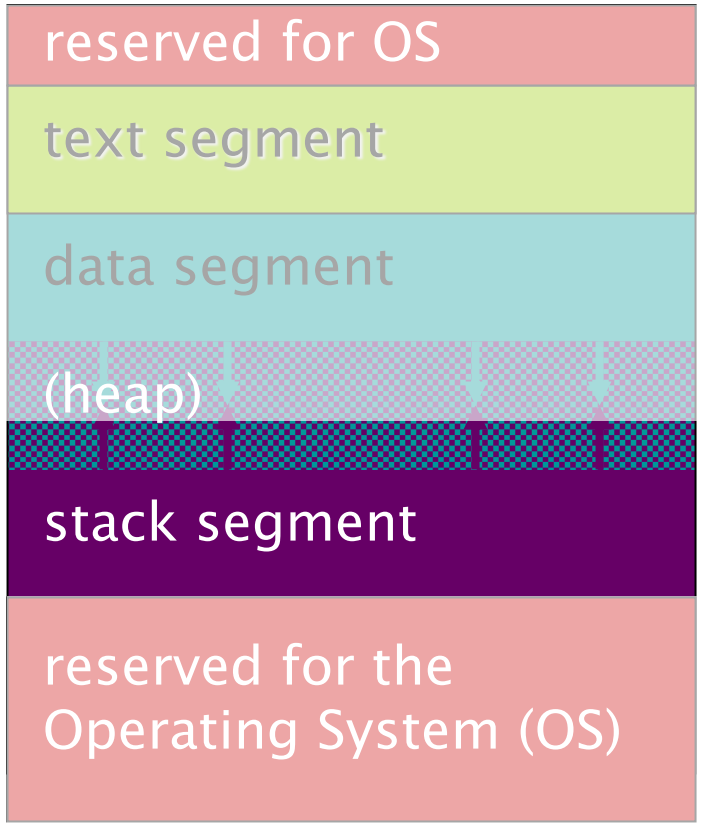
- **Allocation/deallocation of local variables obeys LIFO**
  - The last allocated is the first to be deallocated
- **A stack data structure is ideal for storing them**
  - Allocate a variable by pushing it on the stack
  - Deallocate a variable by popping it off the stack
- **Also helpful for storing 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

MONASH University

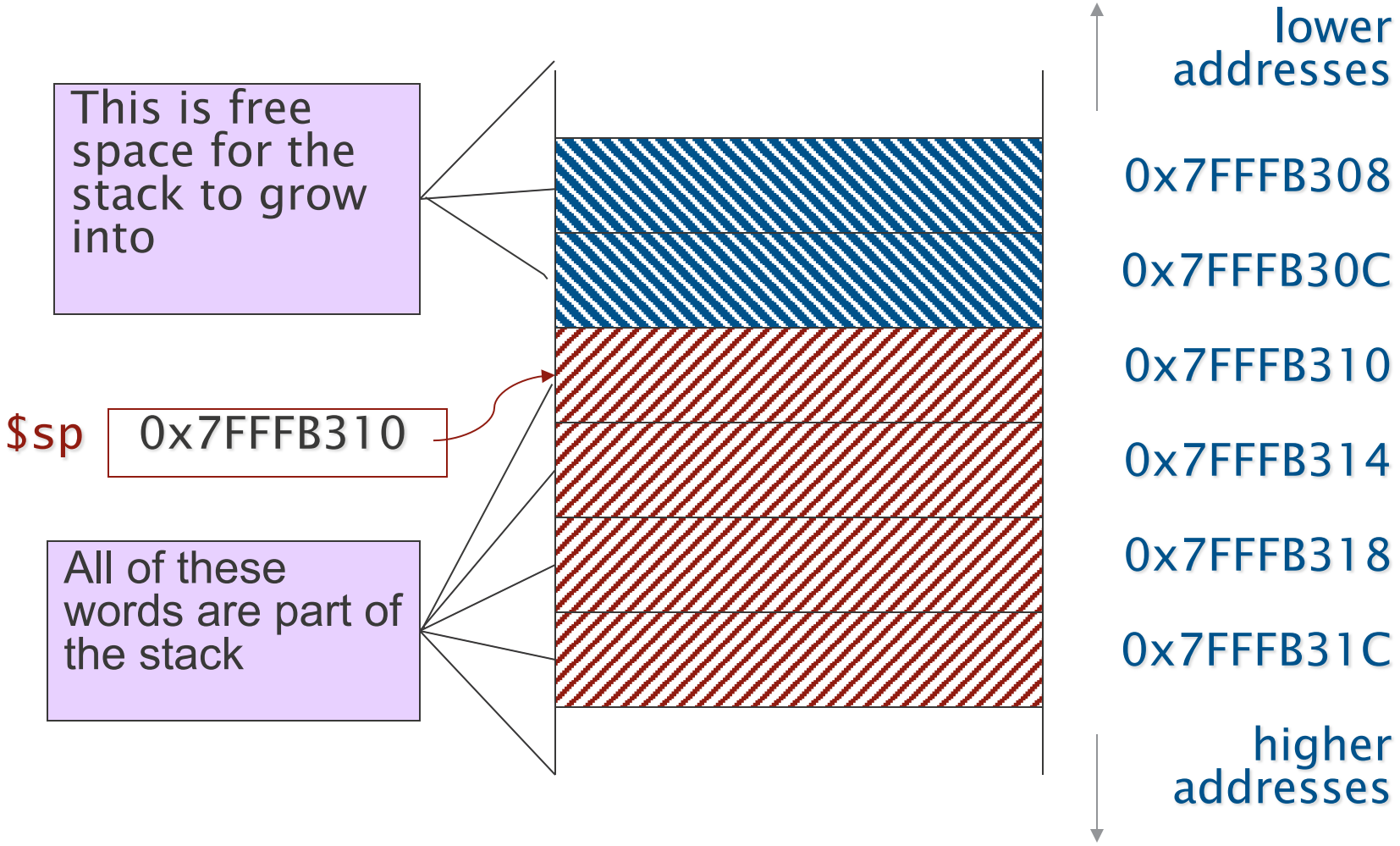
The System Stack and the Stack Pointer

System stack in MIPS

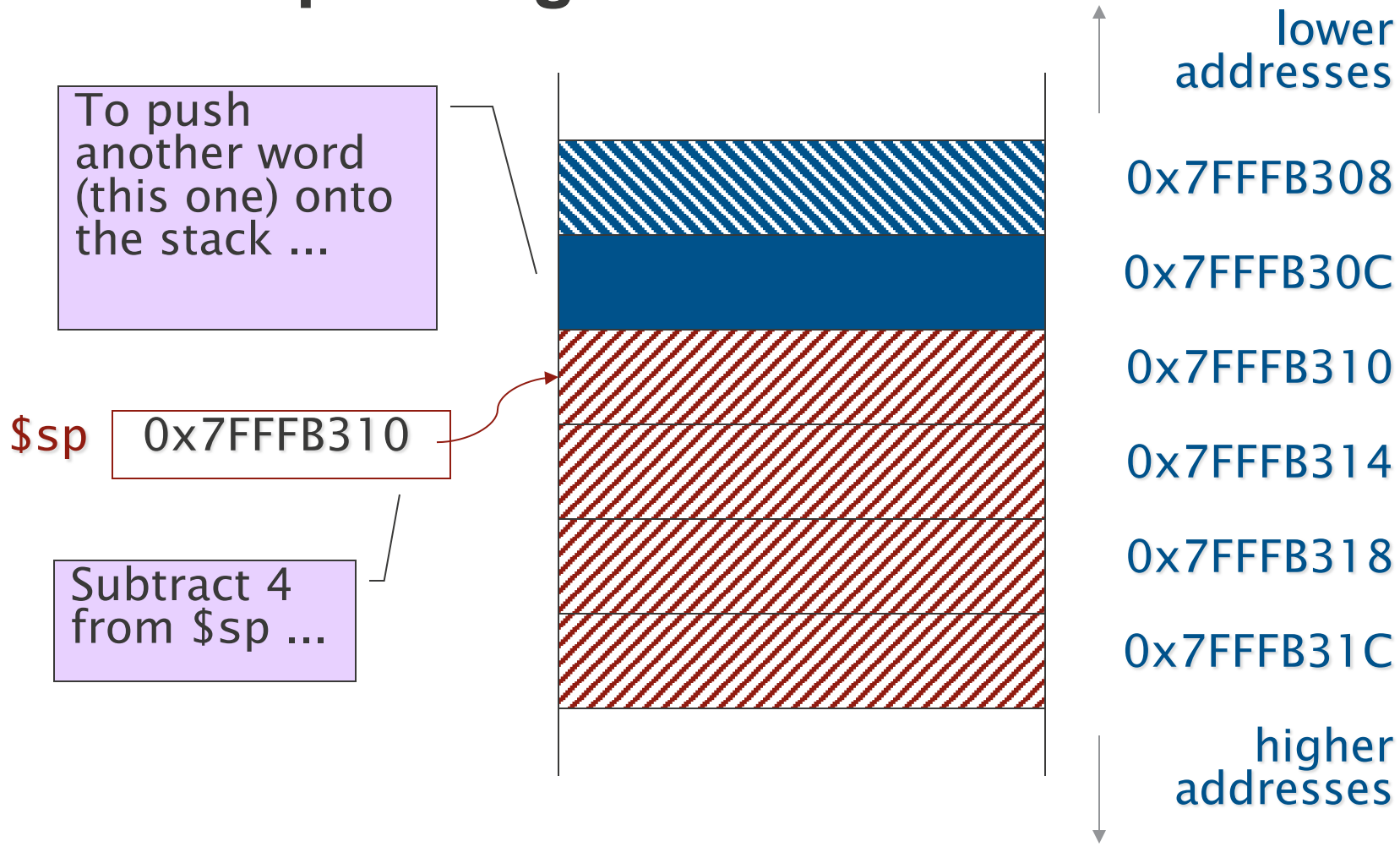
- **Has its own segment of memory**
  - Stack segment: to address 0x7FFFFFFF (0x80000000 is OS)
- **Register `$sp` (stack pointer) indicates the top of stack**
  - Contains the address of the word of memory at the top of stack (i.e., with lowest address)
  - Its value changes during the execution of a function
- **How do we push and pop variables?**



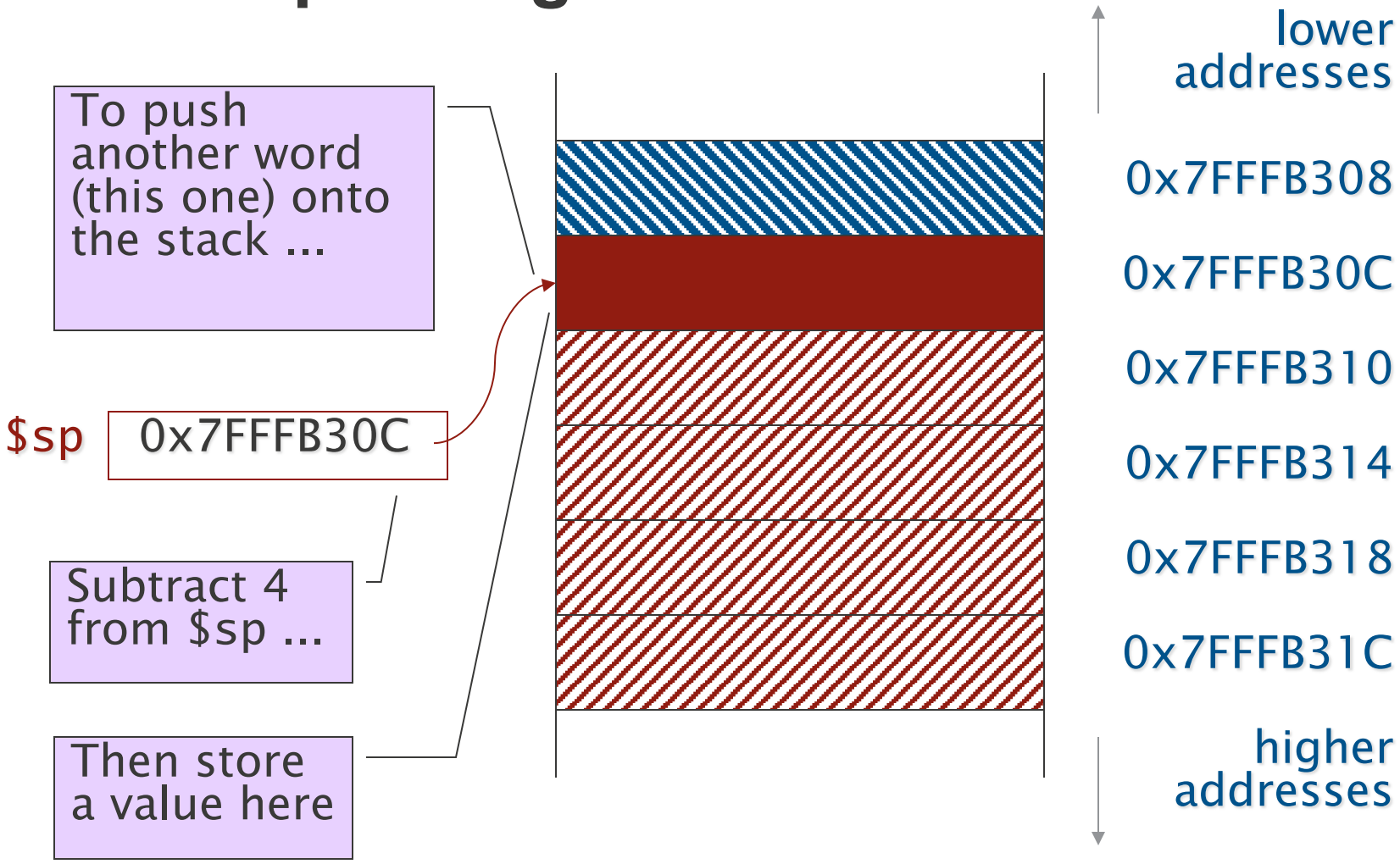
System stack



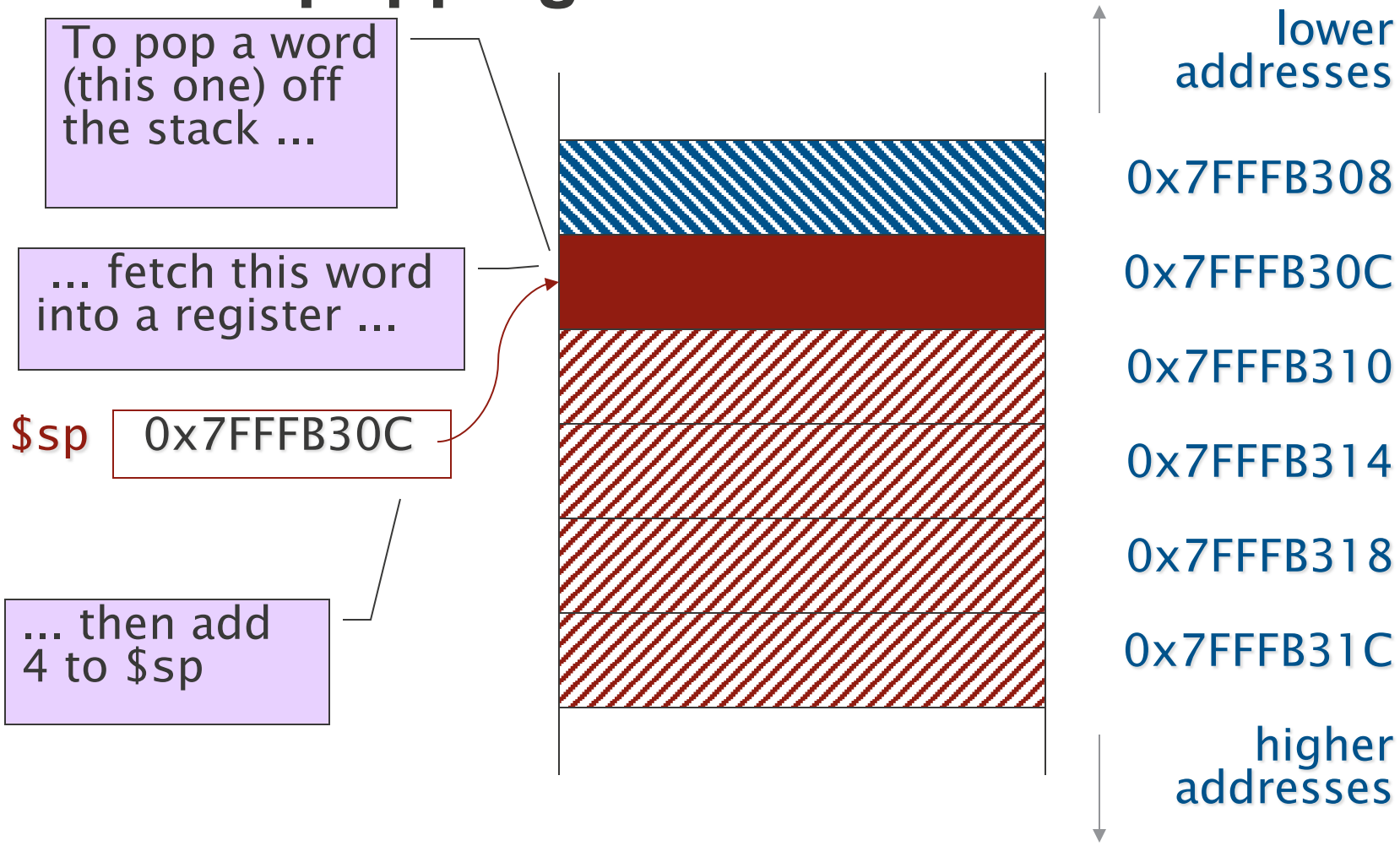
System stack: pushing



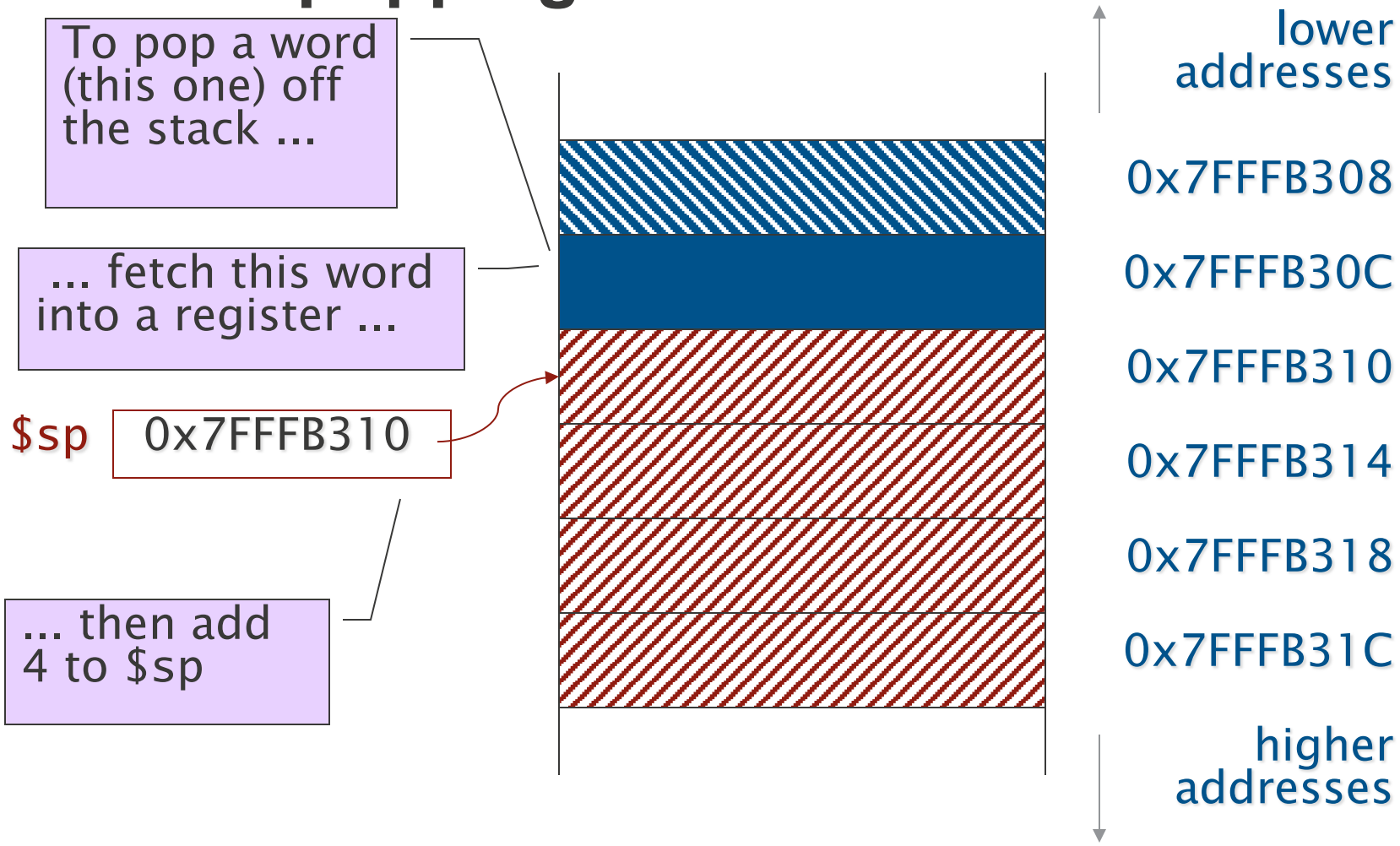
System stack: pushing



System stack: popping

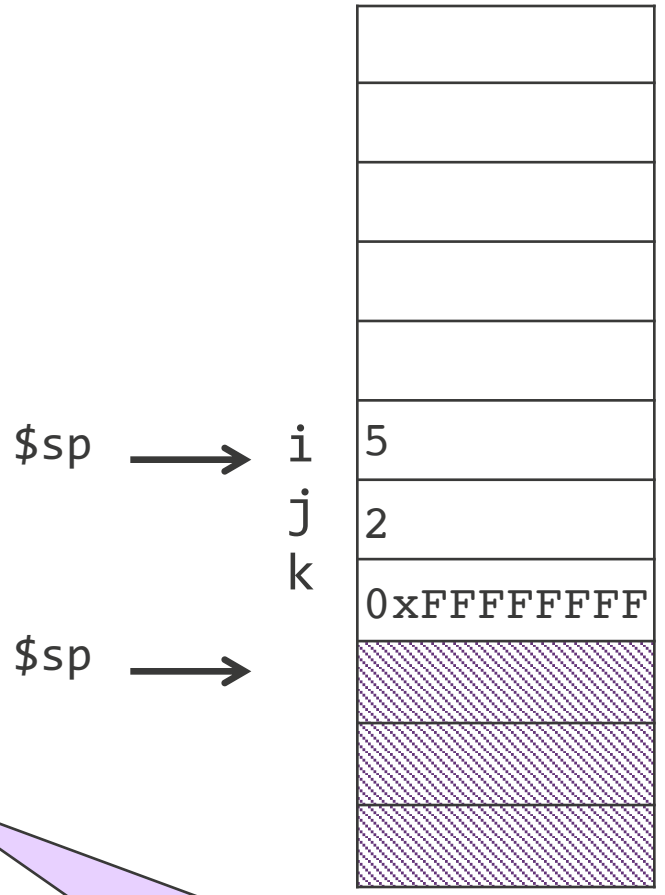


System stack: popping



How does the system stack work?

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



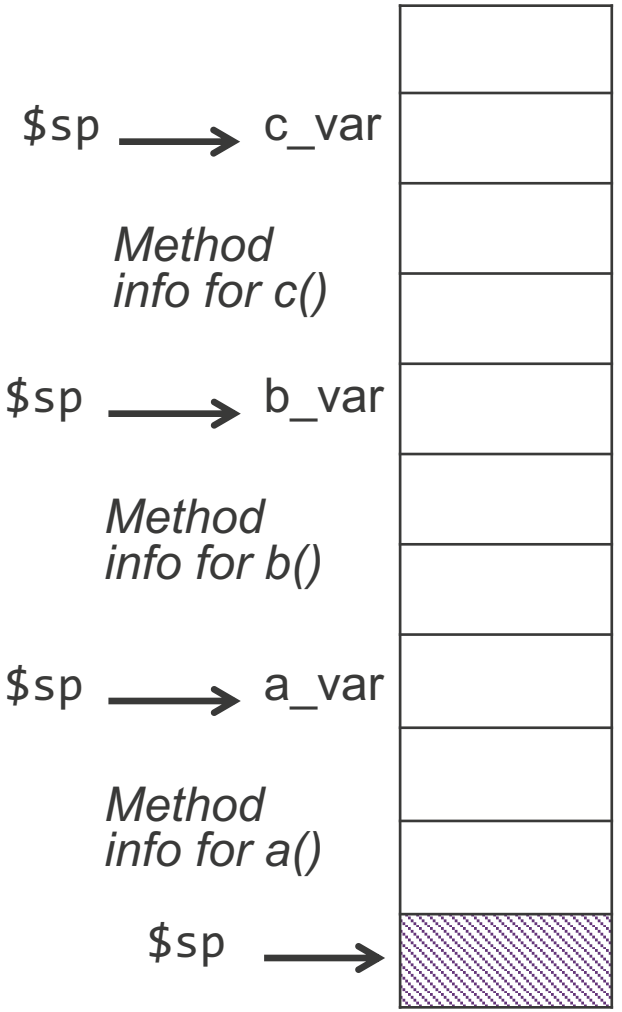
Example:

```
def a():

  a_var = 0

  b()
```

```
def b():

  b_var = 0

  c()
```

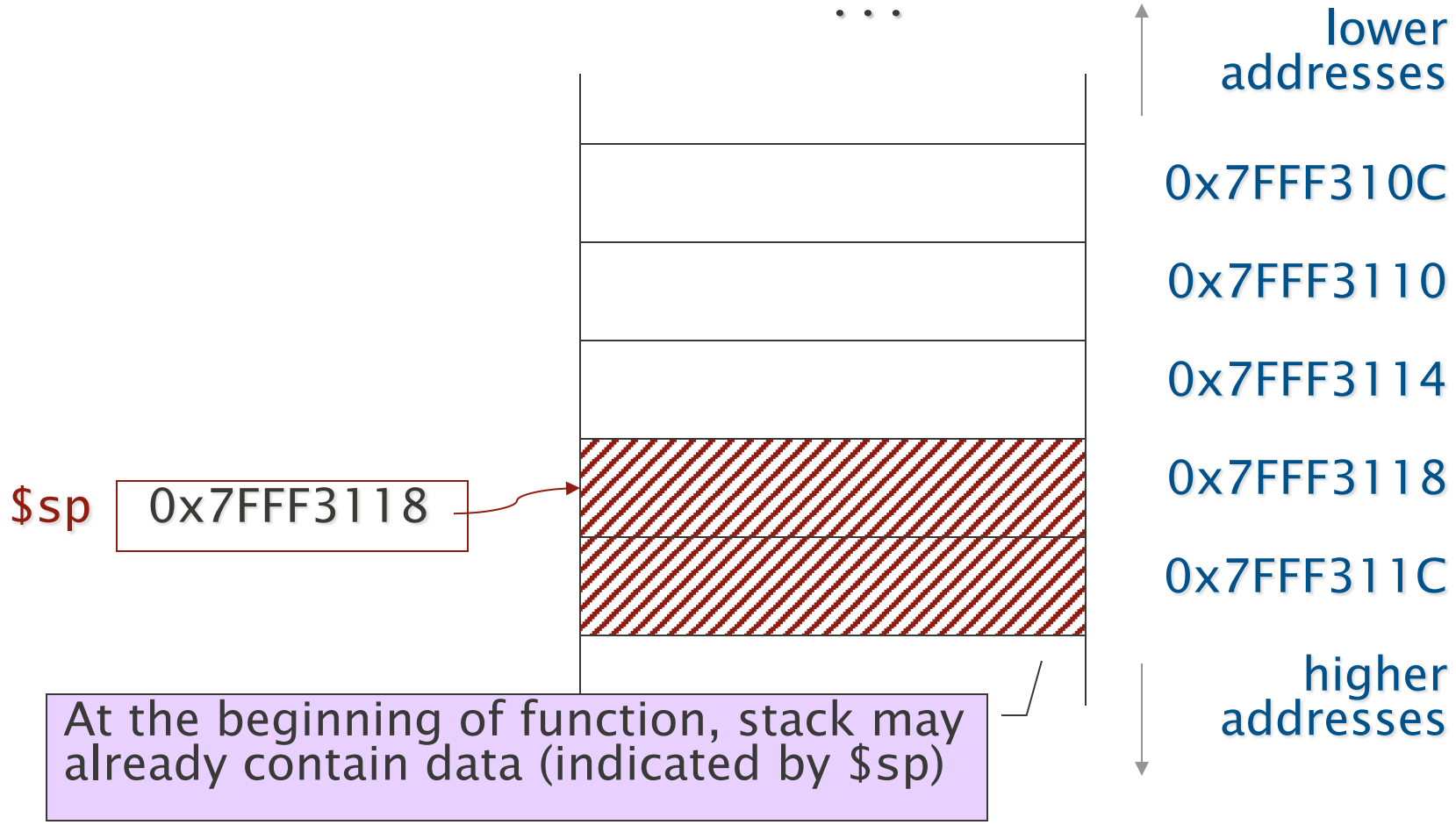
```
def c():

  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**



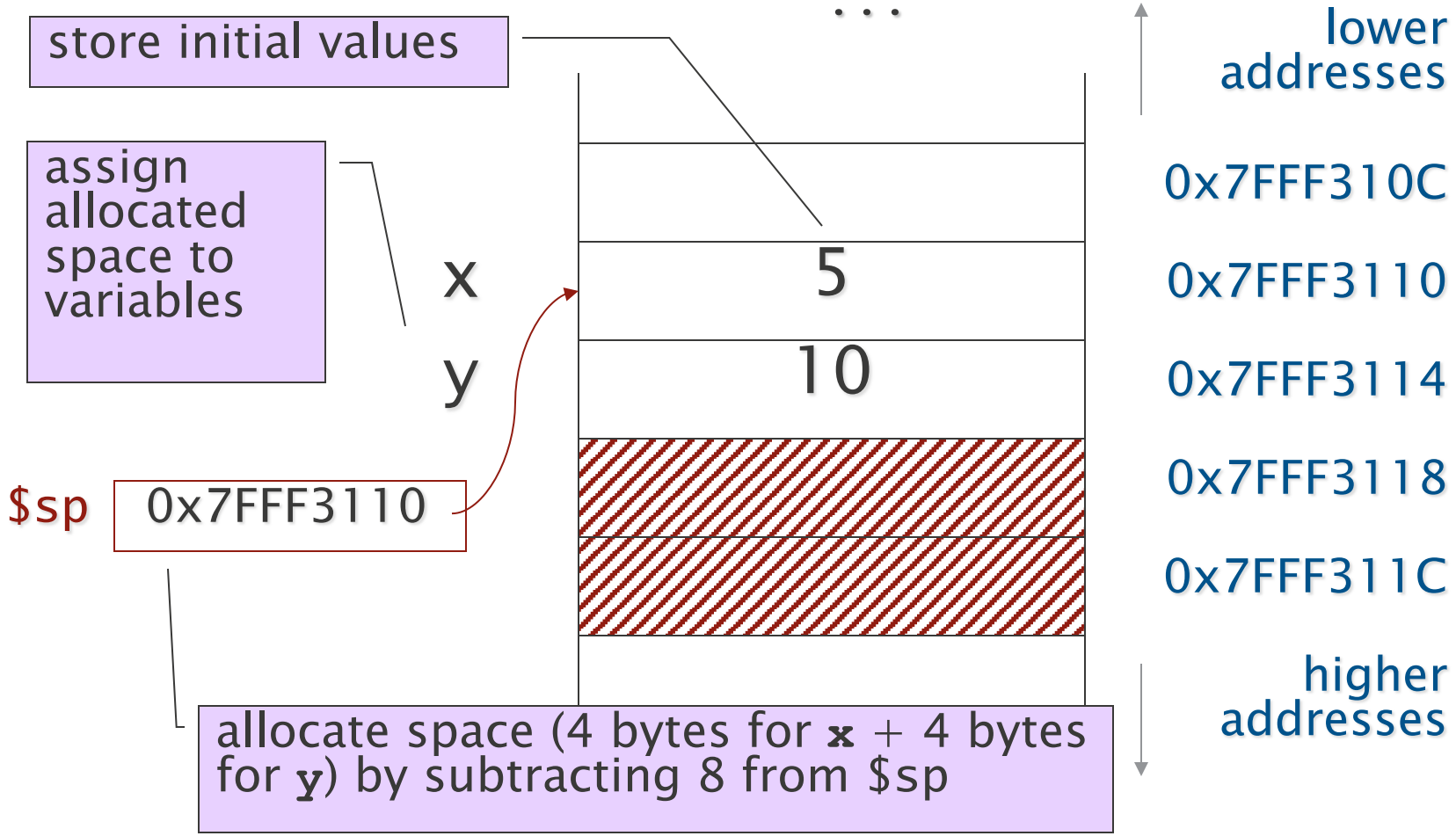
Example

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



Example

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

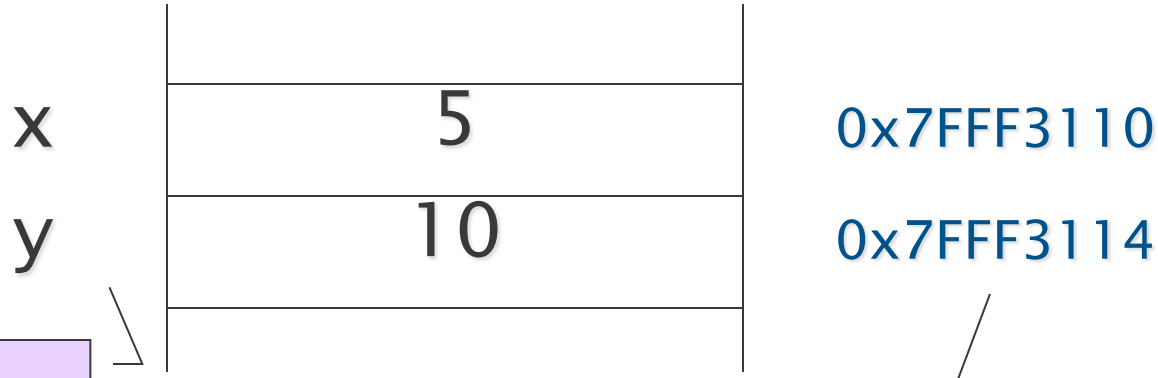


MONASH University

The Frame Pointer and Memory Diagrams

Memory diagram: local variables

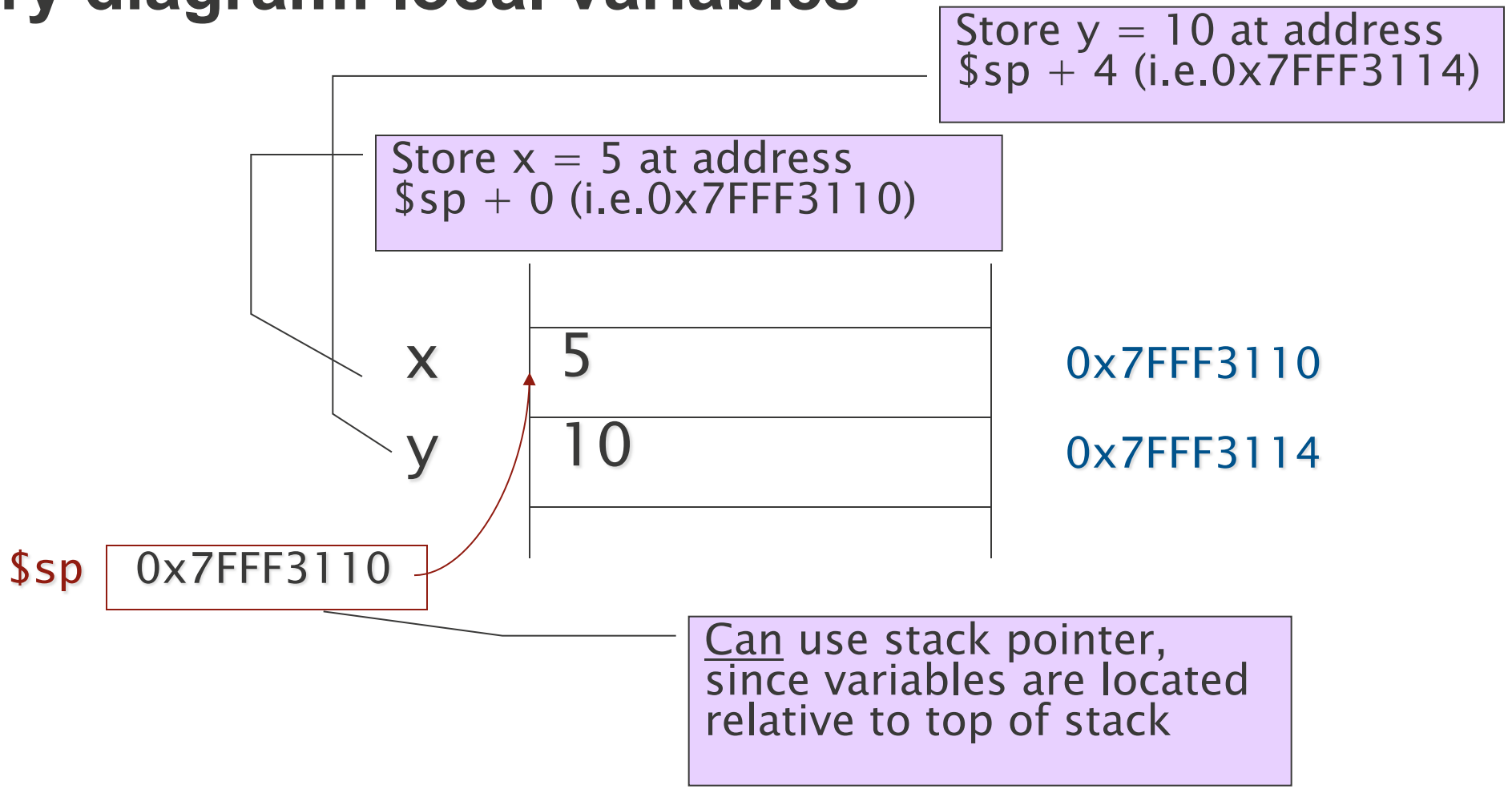
- **How do we use memory diagrams for local variables?**
  - We need to refer to local variables. But how?



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

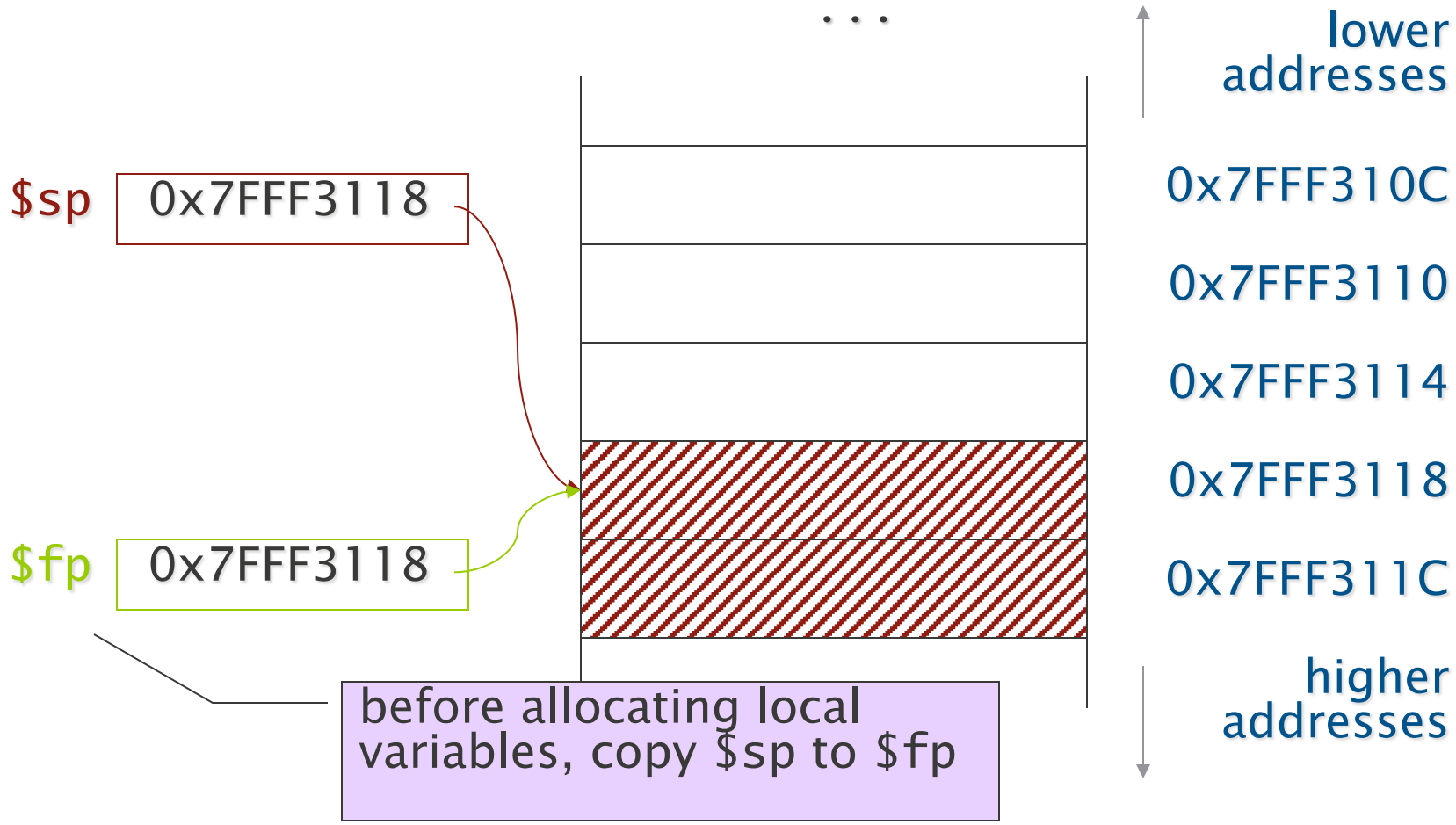


Frame pointer

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

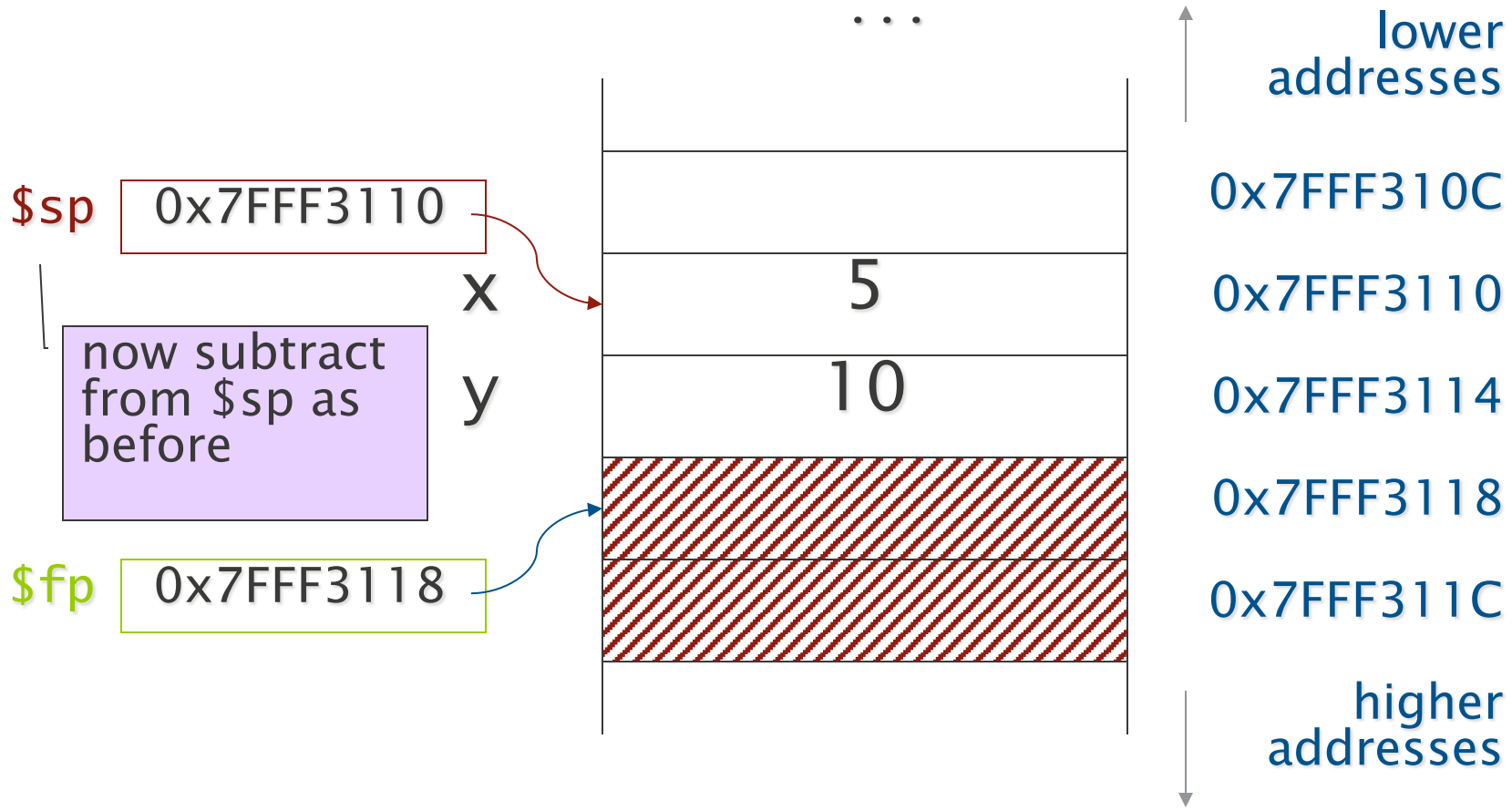
Local variables

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

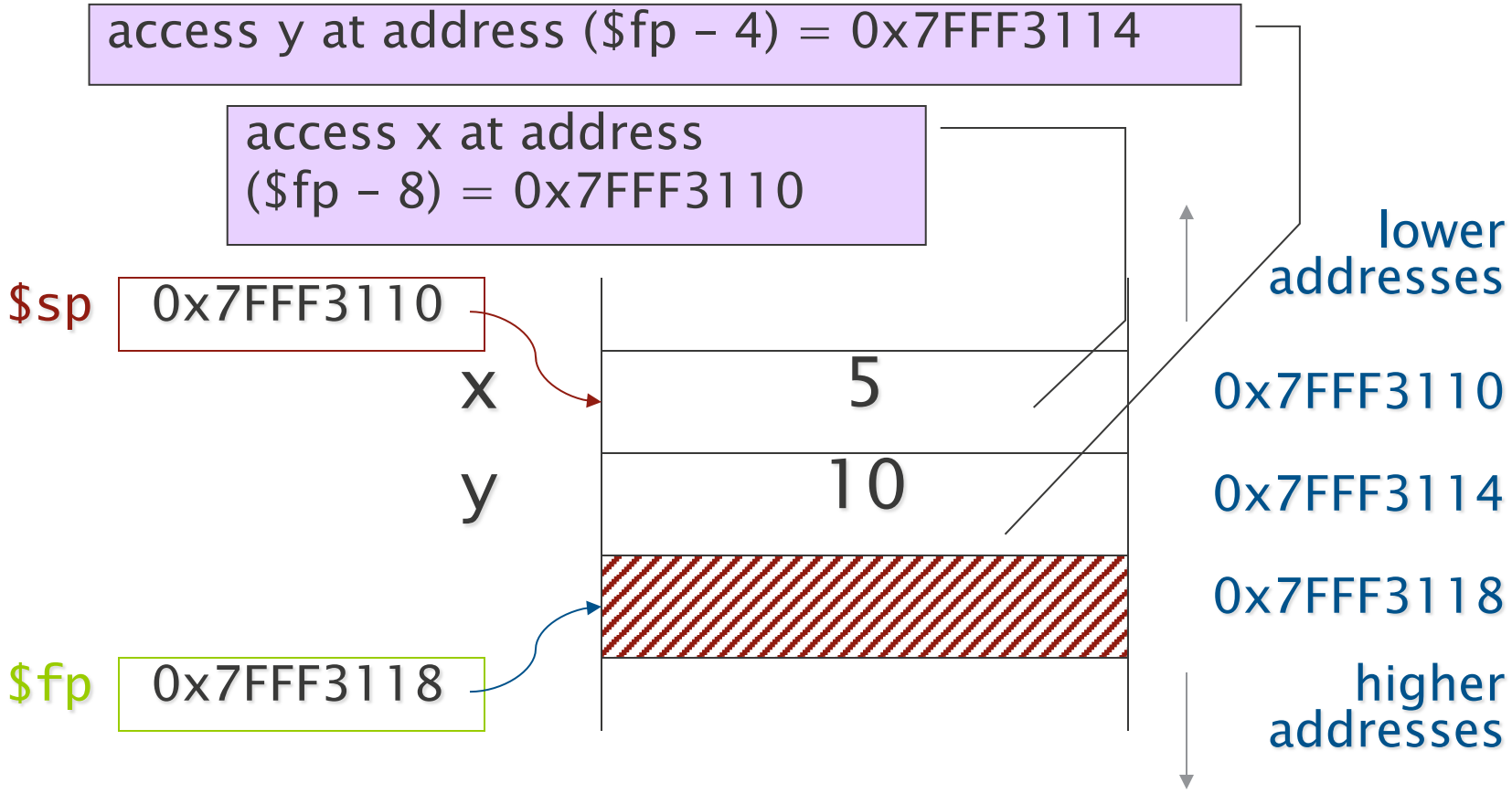


Local variables

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

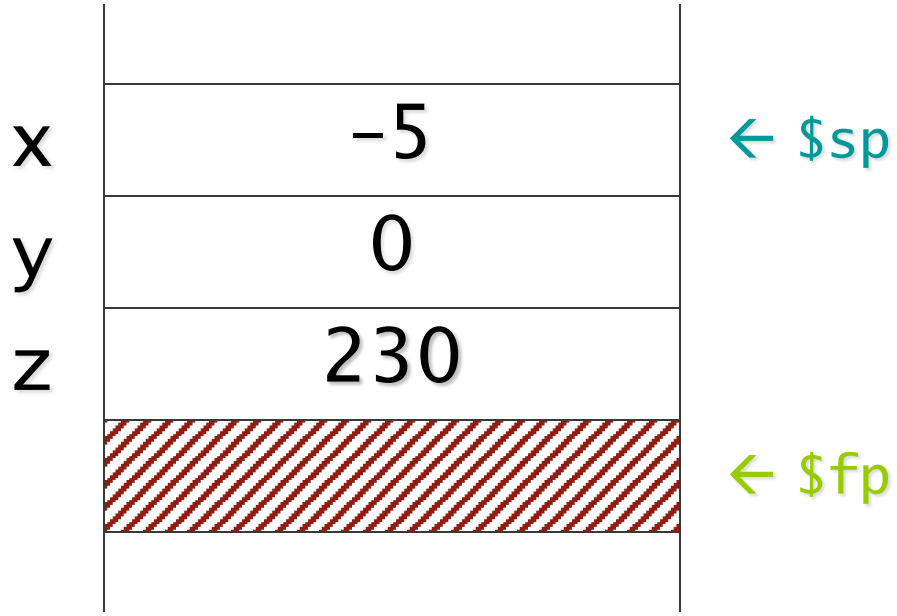


Local variables



```
// A global variable
g = 123

def main():

  // Three local variables
  x = -5
  y = 0
  z = 230

  // Do some arithmetic
  y = g + x

  // Do some more arithmetic
  print(z - x)
```

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

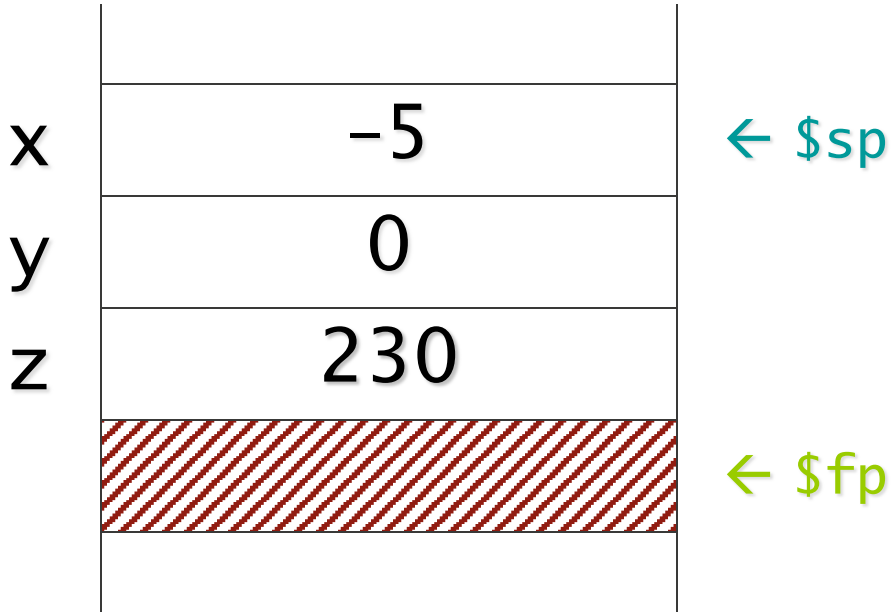


This memory diagram corresponds to this program point

x is at -12($fp)

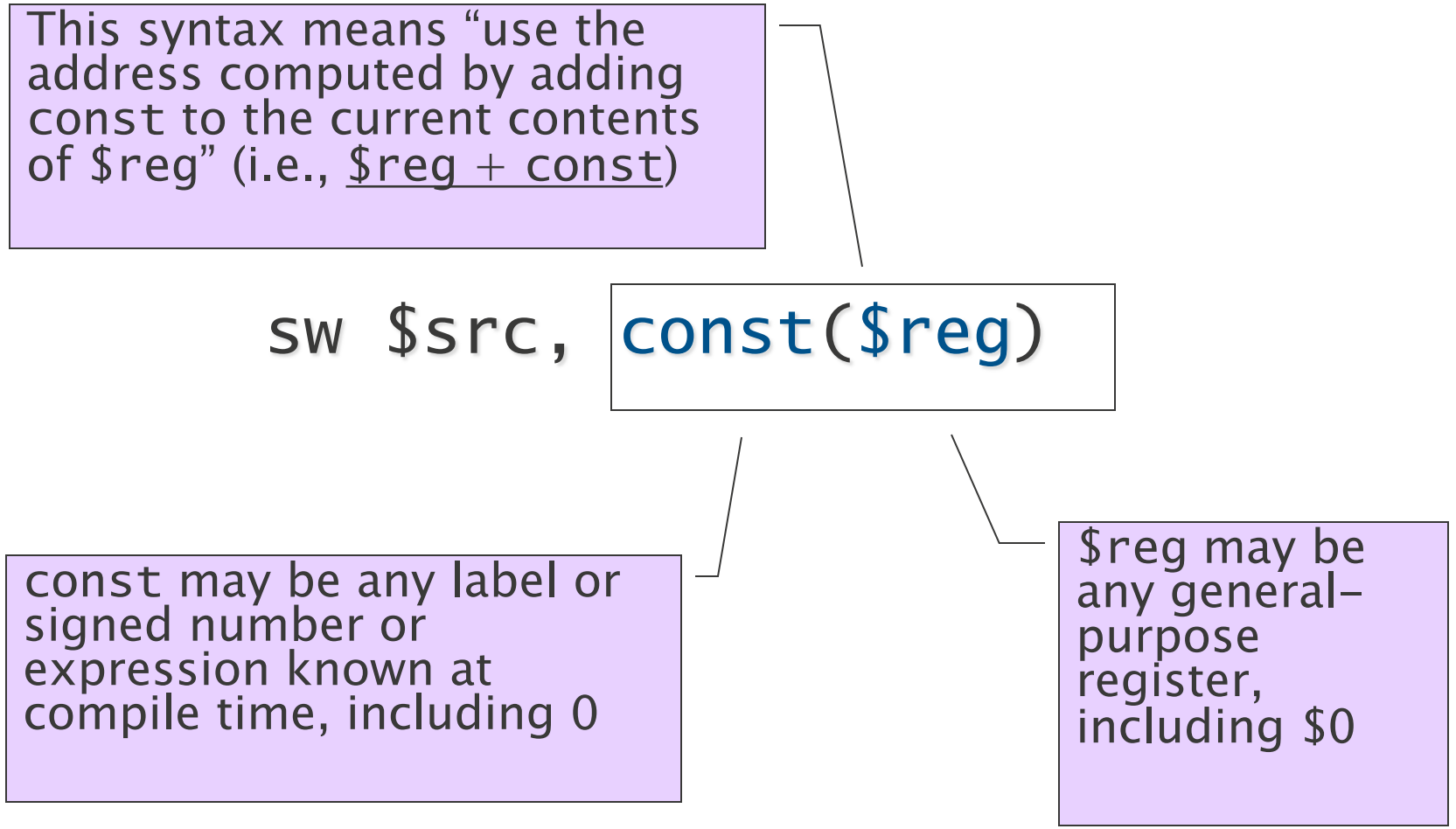
y is at -8($fp)

z 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
        sw $t0, -12($fp)   # x = -5

        sw $0, -8($fp)     # y = 0
        addi $t0, $0, 230

        sw $t0, -4($fp)    # z = 230
        # ... rest in next slide
...
```

Reminder: addressing modes (you used them for arrays)



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)

MONASH University

Putting it all together

x is at -12($fp)

y is at -8($fp)

z is at -4($fp)

```
// A global variable
g = 123

def main():
  // Three local variables
  x = -5
  y = 0
  z = 230
  // Do some arithmetic
  y = g + x

  // Do some more arithmetic
  print(z - x)
```

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

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

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

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

        sw  $0, -8($fp)    # y = 0

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

x is at -12($fp)

y is at -8($fp)

z 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

  x = -5
  y = 0
  z = 230

  // Do some arithmetic
  y = g + x

  // Do some more arithmetic
  print(z - x)
```

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

# y = g + x
lw $t0, g              # load g
lw $t1, -12($fp)       # load x
add $t0, $t0, $t1      # g+x
sw $t0, -8($fp)        # y = g+x

# print(z-x)
addi $v0, $0, 1
lw $t0, -4($fp)        # load z
lw $t1, -12($fp)       # load x
sub $a0, $t0, $t1      # z-x
syscall

# exit program
addi $v0, $0, 10
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

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
- **If you are asked to be "faithful" then:**
  - Translate each line of code independently of the others (i.e., without reusing the value of registers computed in previous instructions)
  - Keep loop/if-then-else conditions, order of lines, strings and prints as given
  - Encode globals as globals, and locals as locals
- **Comment appropriately:**
  - Each block should correspond to a line of Python code
  - Sub-blocks for complex Python lines are good
  - No need to comment each line for simple things, like easy maths or `syscalls`,
  - Comment lines for complex things like conditions, array access, etc

Summary

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

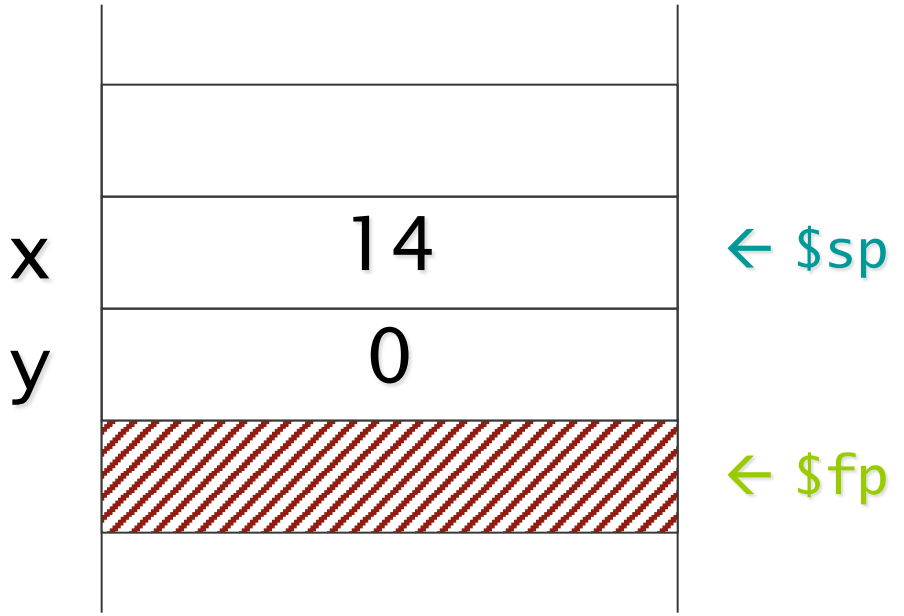
MONASH University

Another Example

```
// A global variable
n = 4

def main():

  // Two local variables

  x = 14
  y = 0

  // Do some arithmetic
  y = (n * x) - 7

  // Do some more arithmetic
  y = y // 16

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

x is at -8($fp)

y is at -4($fp)



```
// A global variable
n = 4

def main():

  // Two local variables

  x = 14
  y = 0

  // Do some arithmetic
  y = (n * x) - 7

  // Do some more arithmetic
  y = y // 16

  // Do even more arithmetic
  print(y + 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
sw $t0, -8($fp)     # x = 14
sw $0, -4($fp)      # y = 0

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

x is at -8($fp)

y is at -4($fp)

```
// A global variable
n = 4

def main():

  // Two local variables

  x = 14
  y = 0

  // Do some arithmetic
  y = (n * x) - 7

  // Do some more arithmetic
  y = y // 16

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

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

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

# print(y+n)
addi $v0, $0, 1
lw $t0, -4($fp)     # y
lw $t1, n
add $a0, $t0, $t1   # y+n
syscall

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


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

x is at -8($fp)

y is at -4($fp)