Frame pointer

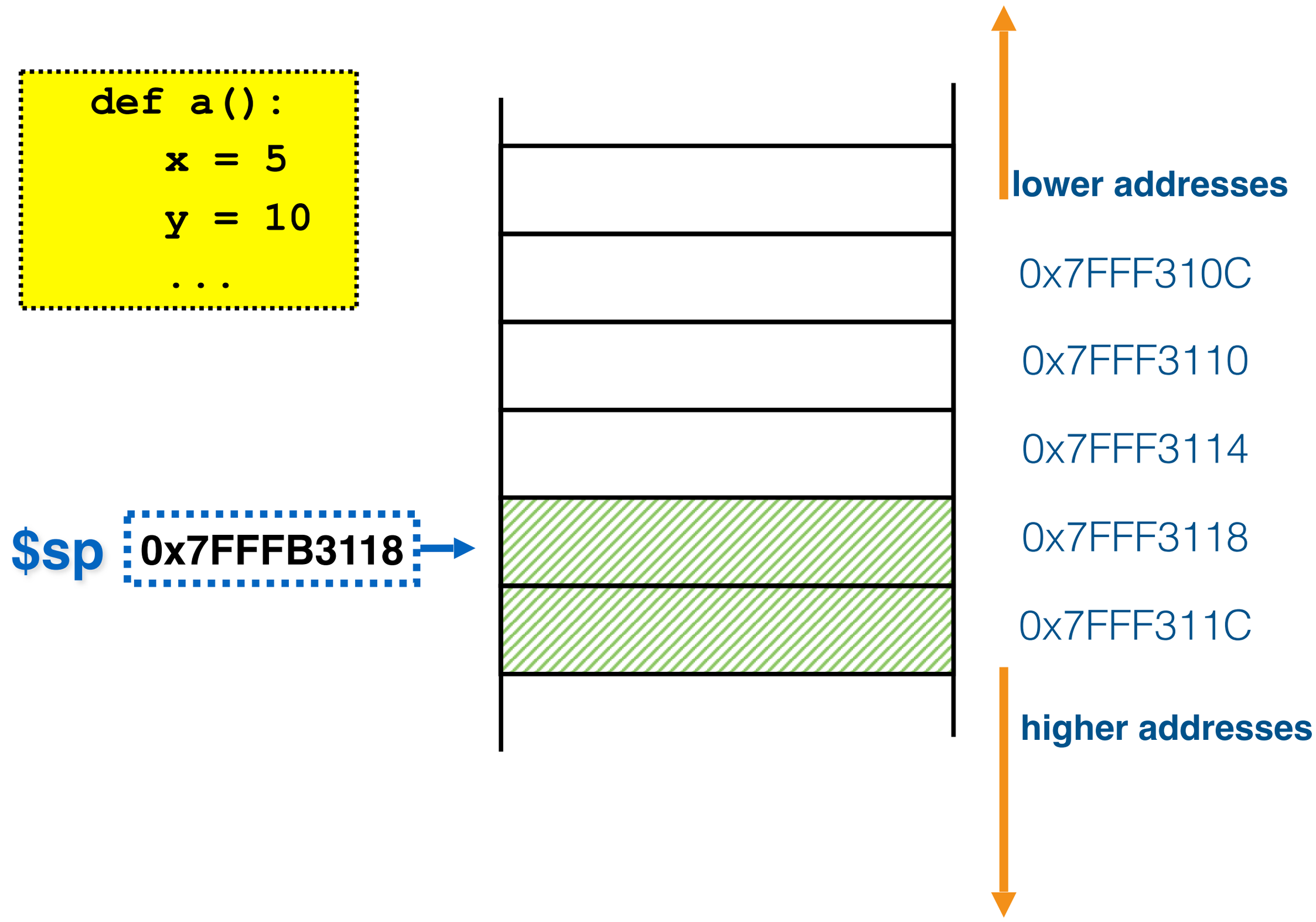
- Can access local variables relative to stack pointer (**$sp**).
- However, this may be problematic when passing arguments to functions:
  - Stack pointer moves to accommodate other function info
  - Relative locations of local variables change

Frame pointer

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

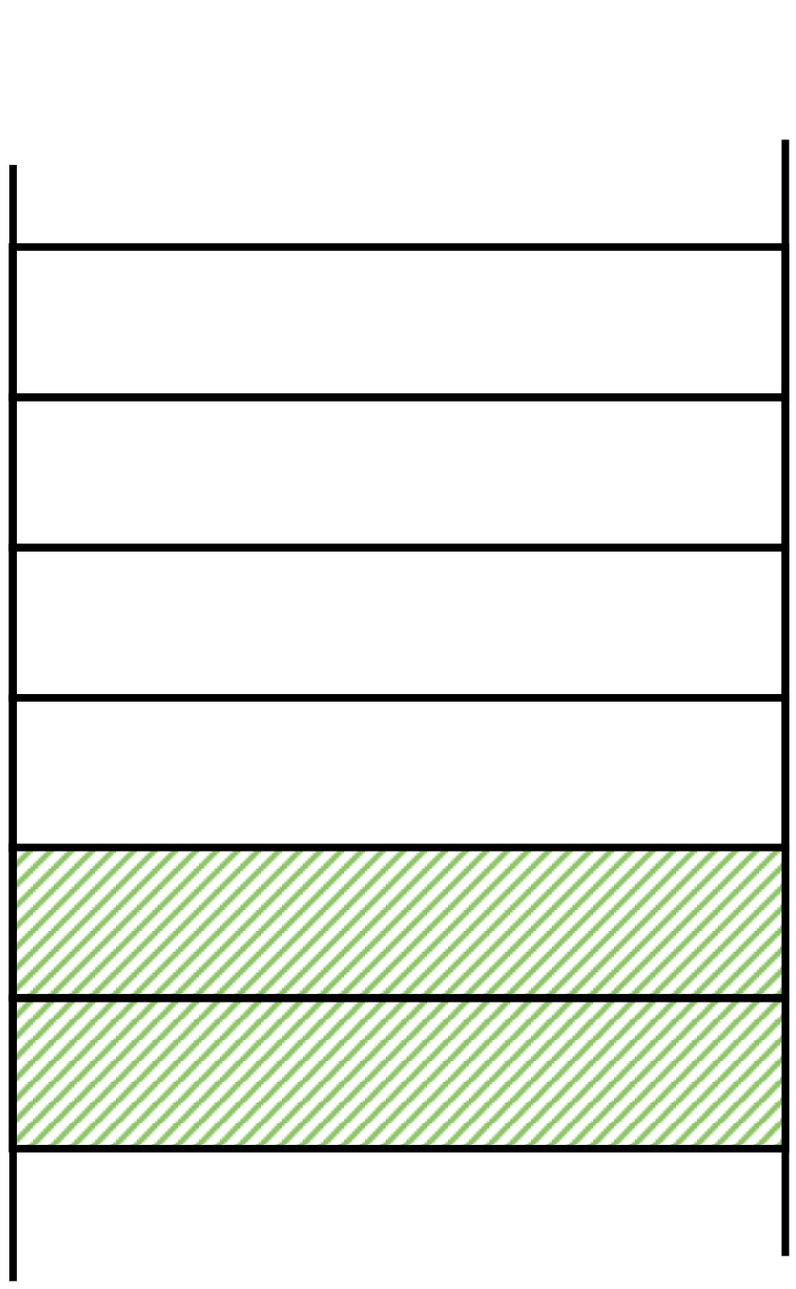
Example

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



Example

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



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

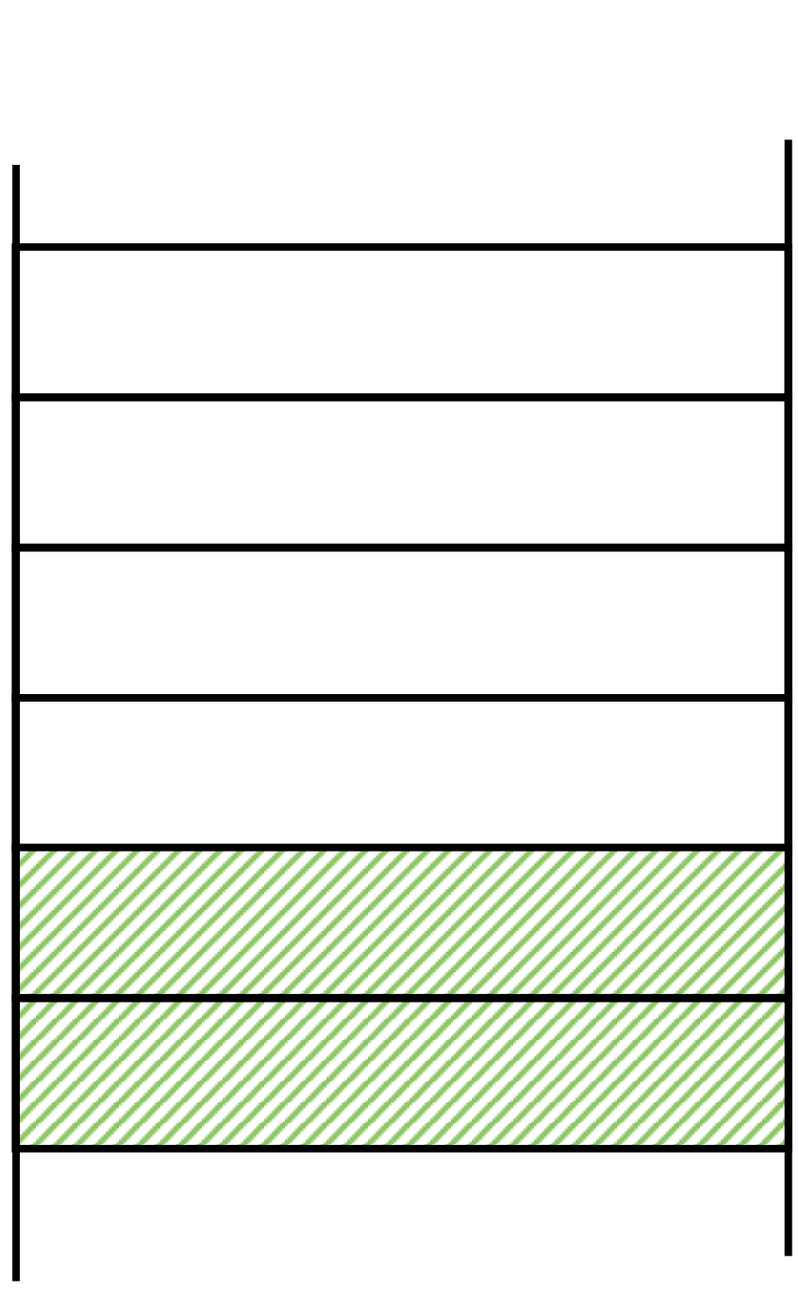
higher addresses

$sp **0x7FFFB3118**

Before allocating,
copy **$sp** to **$fp**

Example

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



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

higher addresses

$sp 0x7FFFB3118

$fp 0x7FFFB3118

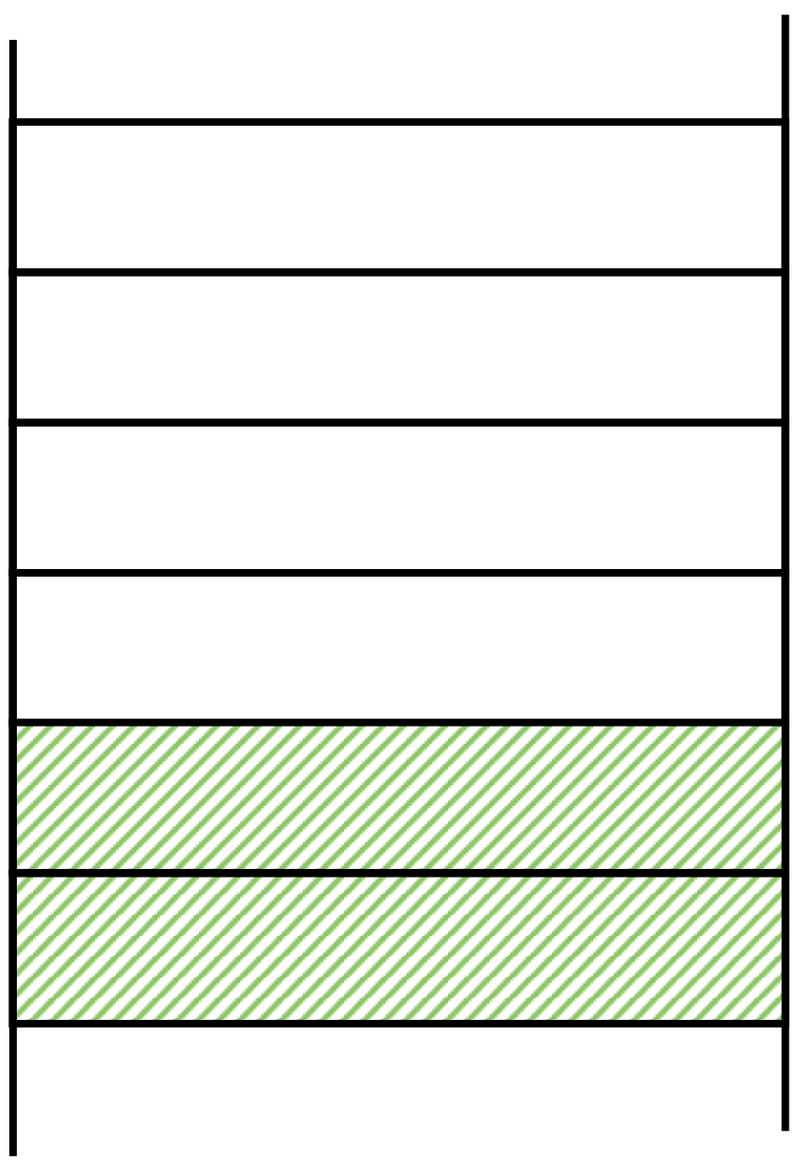
Before allocating, copy **$sp** to **$fp**

Example

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

$sp 0x7FFFB3118

$fp 0x7FFFB3118



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

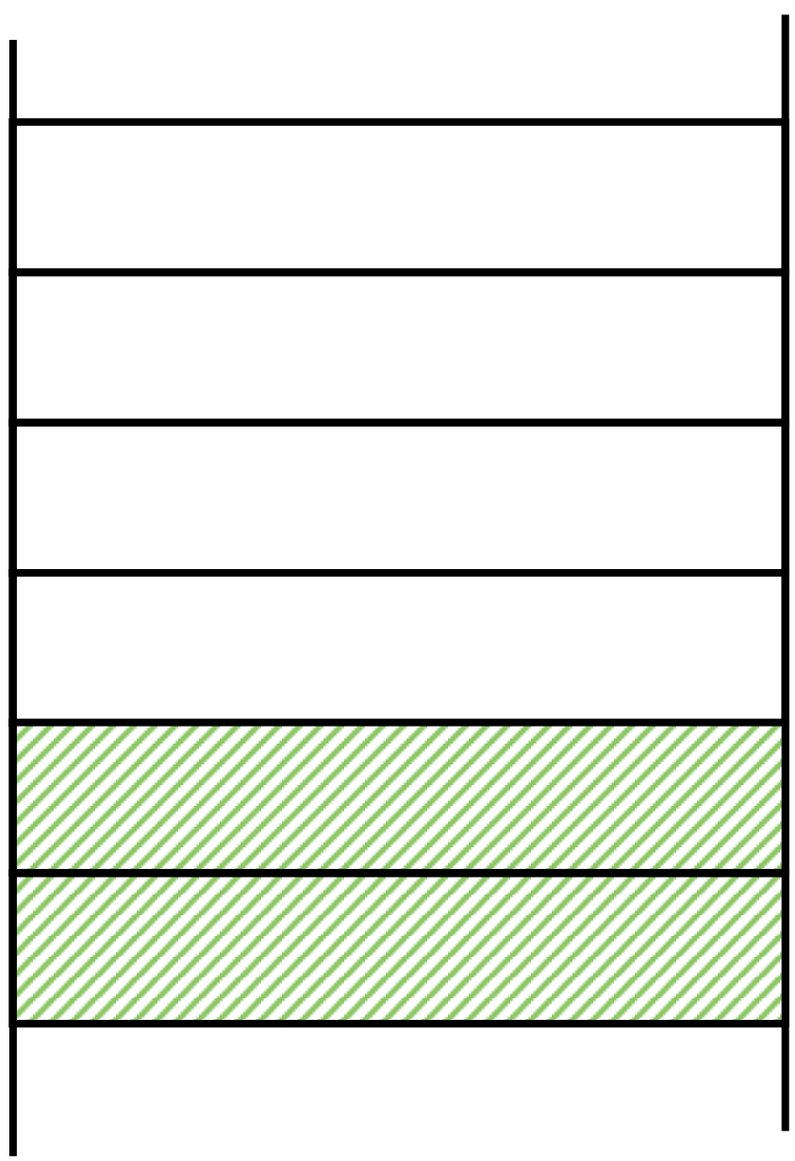
0x7FFF3118

0x7FFF311C

higher addresses

Example

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



lower addresses

0x7FFF310C

0x7FFF3110

0x7FFF3114

0x7FFF3118

0x7FFF311C

higher addresses

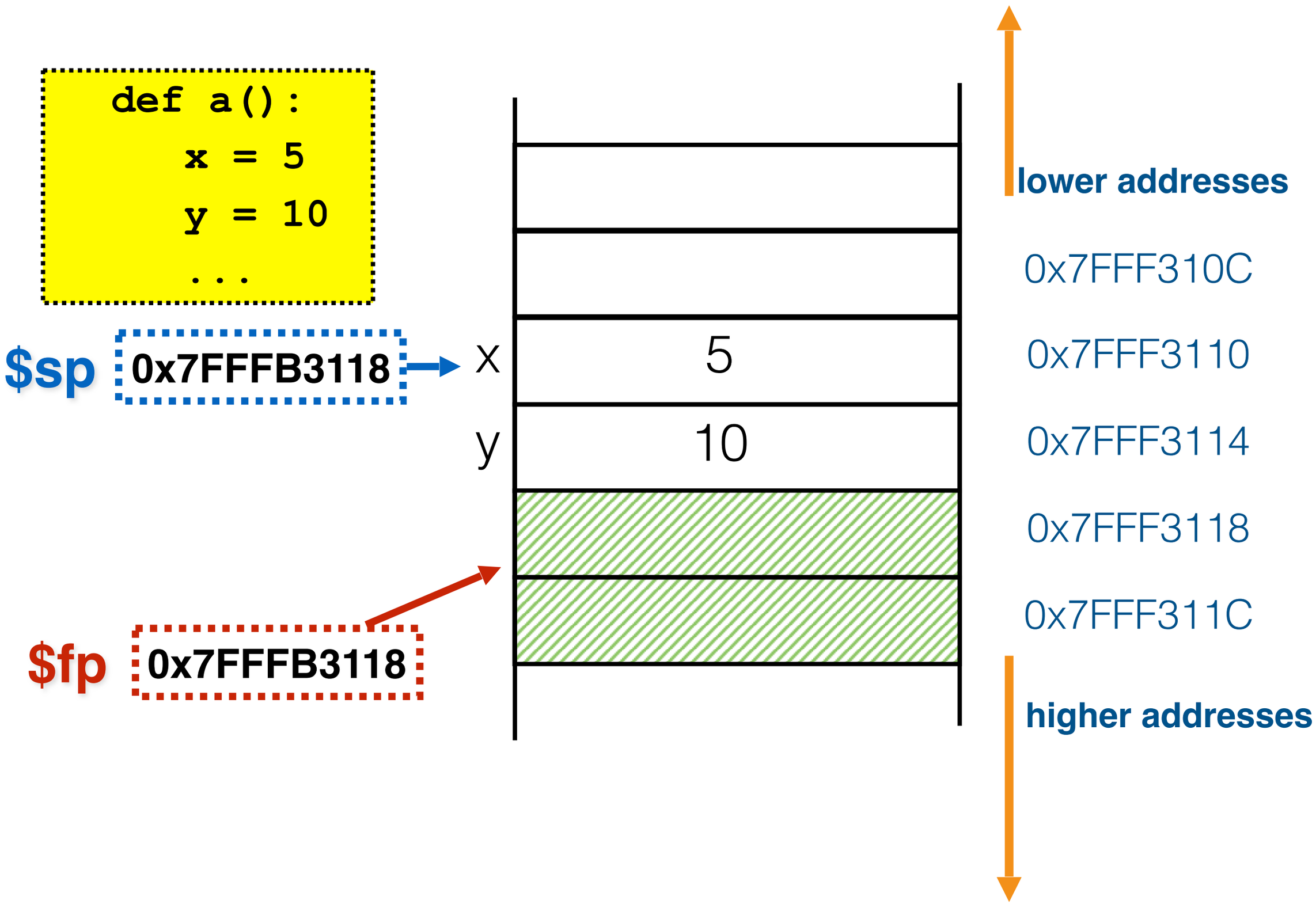
$sp **0x7FFFB3118**

$fp **0x7FFFB3118**

Allocate by subtracting from **$sp** as before

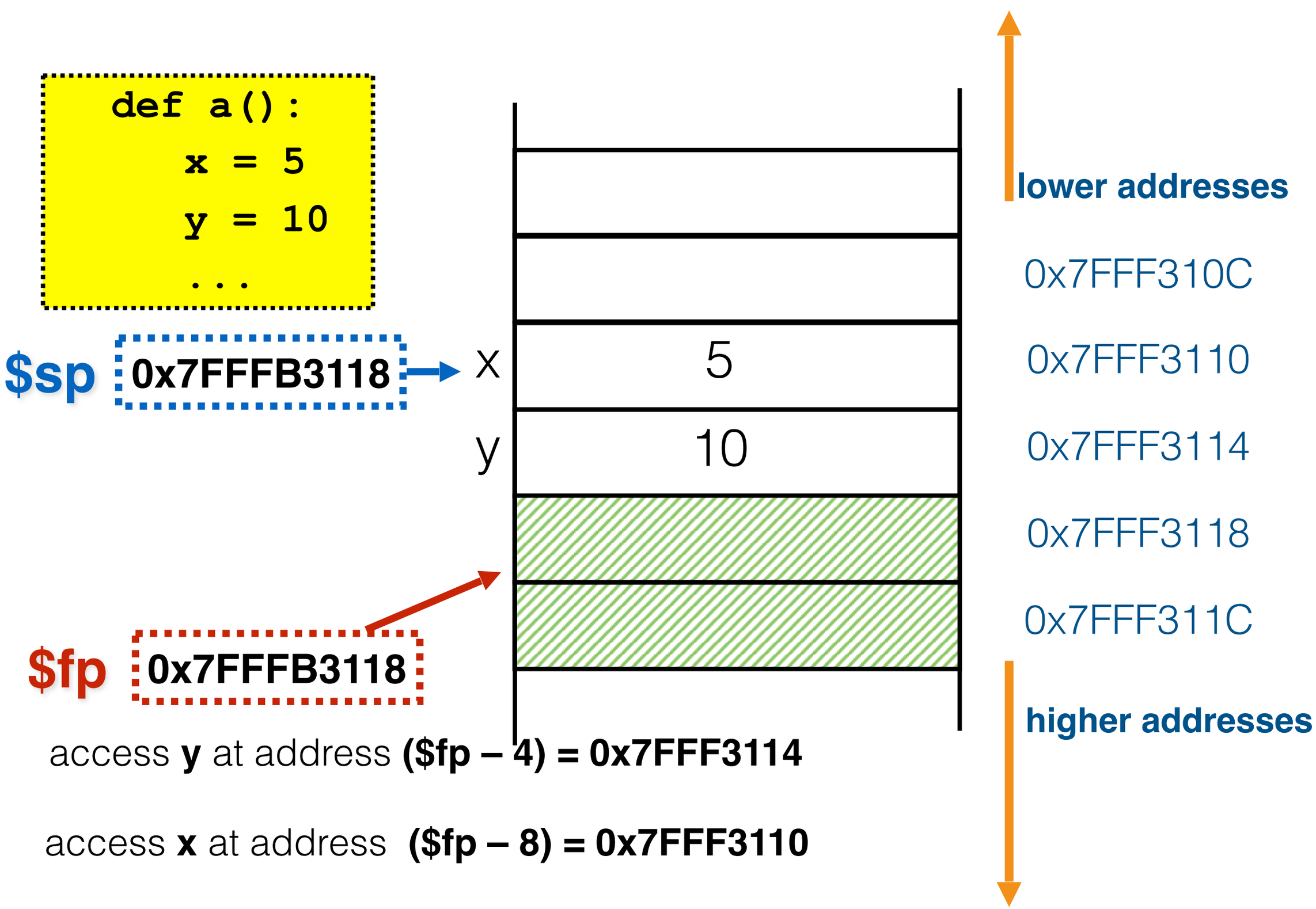
Example

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



Example

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



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

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

- Local variables are referred to without names in MIPS.
- Therefore, remembering their address is vital: **diagrams help**

```
// 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
        # memory for 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 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
        # memory for 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 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
        # memory for 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 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
        # memory for 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 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
```

```
// 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
```

Faithful translation: regs for **g** and **a** are not re-used, they are reloaded

```
// 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
```

Faithful translation: regs for **g** and **a** are not re-used, they are reloaded

```
// 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
```

Faithful translation: regs for **g** and **a** are not re-used, they are reloaded

```
// 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
```

Faithful translation: regs for **g** and **a** are not re-used, they are reloaded

```
// 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
```

```
.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
        # memory for local variable
        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 ...
```

```
 ... 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
```

```
// 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)
```

- System stack:
  - ➡ Pushing and popping
  - ➡ **$sp** and **$fp**

- Local variables:
  - ➡ Stored on stack
  - ➡ Accessed with negative offset from **$fp**

- Addressing: register + constant

Reminder: why using functions?

- As **encapsulation** of a sequence of instructions:
  - ➡ Can be called repeatedly (**reuse**)
  - ➡ Can call other functions
  - ➡ Are self-contained
  - ➡ Can have their own private (local) variables/data

- As **abstractions**:
  - ➡ Can be generalised by taking parameters.
  - ➡ Can inform through return values.

- As **hiders** of information: make sure caller cannot access/ modify internal data

Function calling: return where?

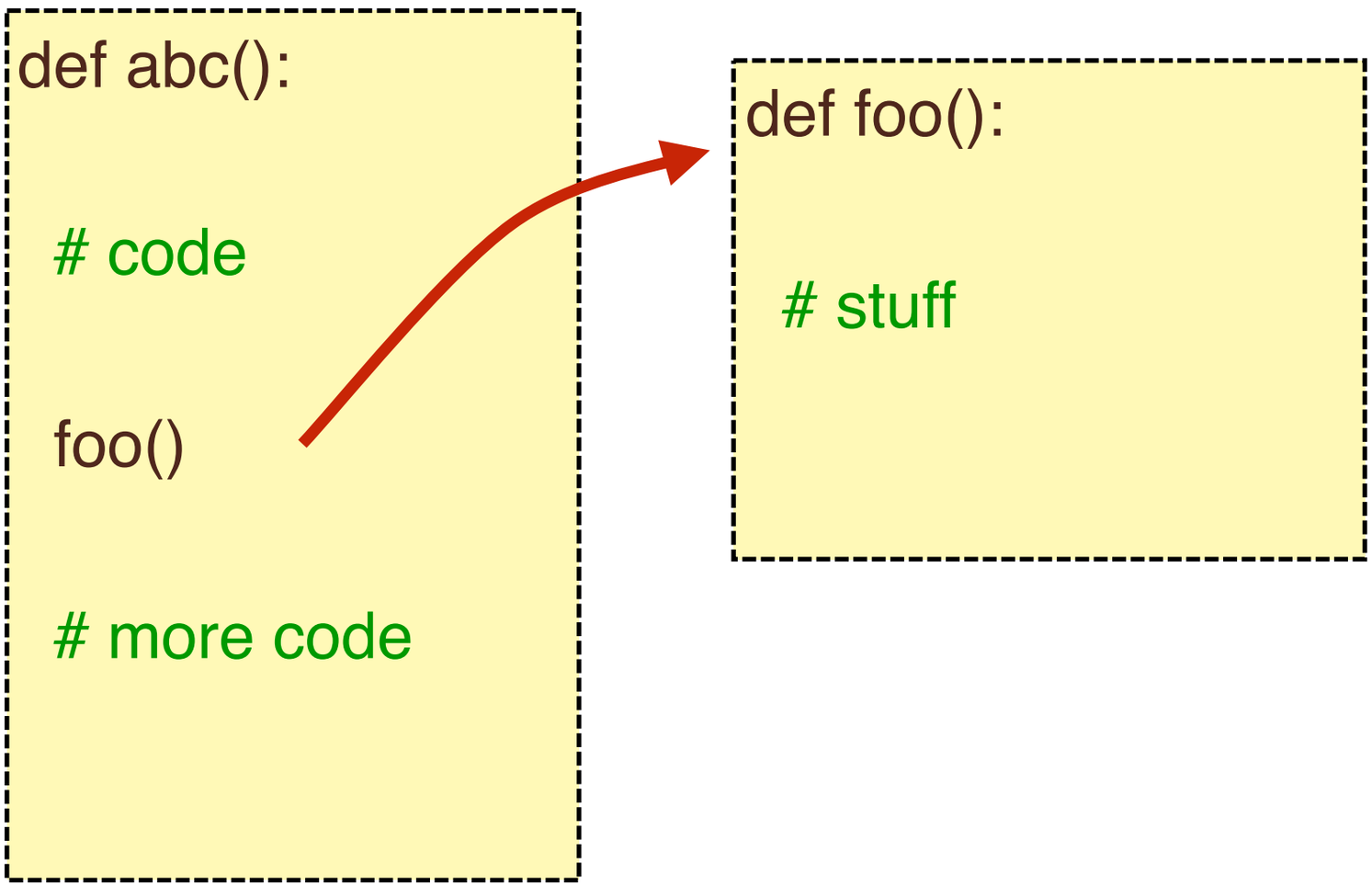
```
def foo():

  # stuff
```

Function calling: return where?

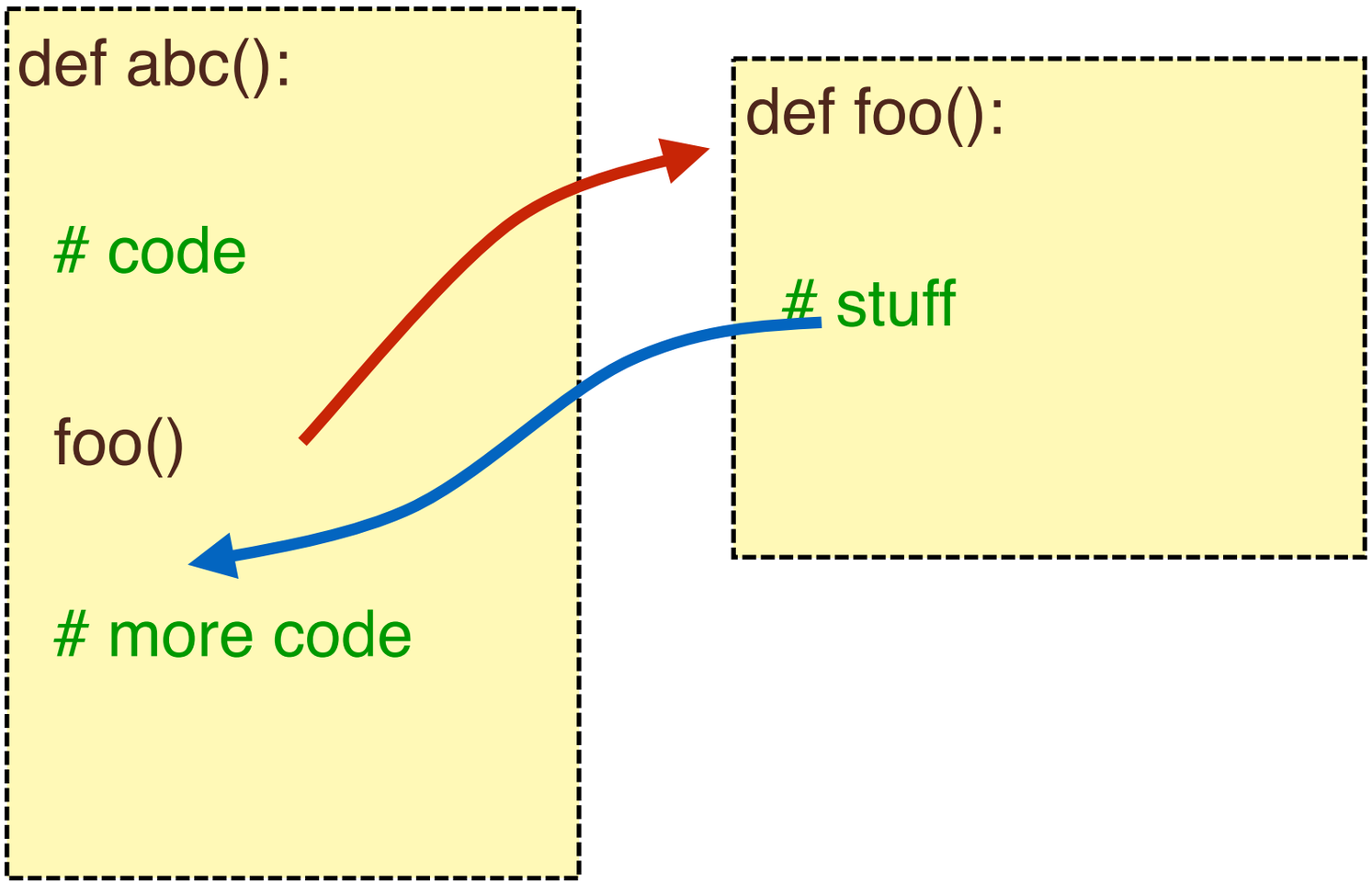
```
def abc():

  # code


  foo()

  # more code
```

```
def foo():

  # stuff
```

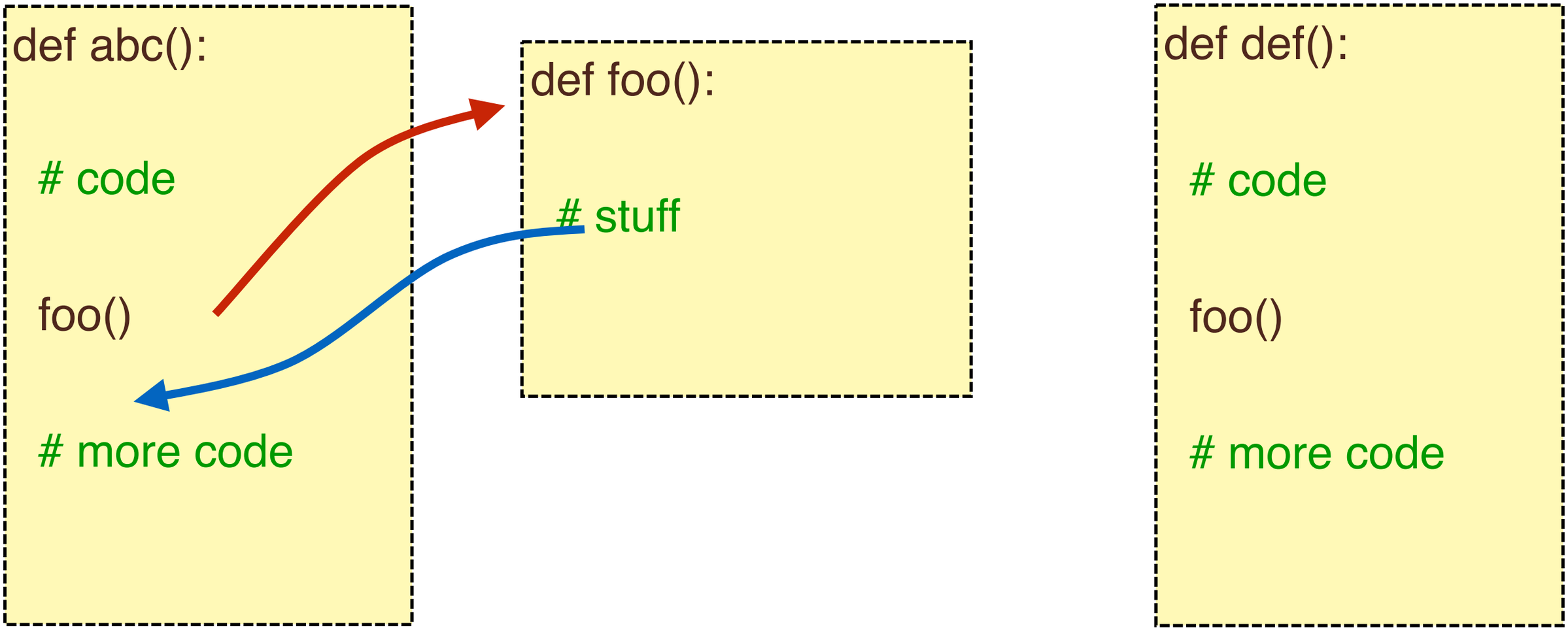
Function calling: return where?



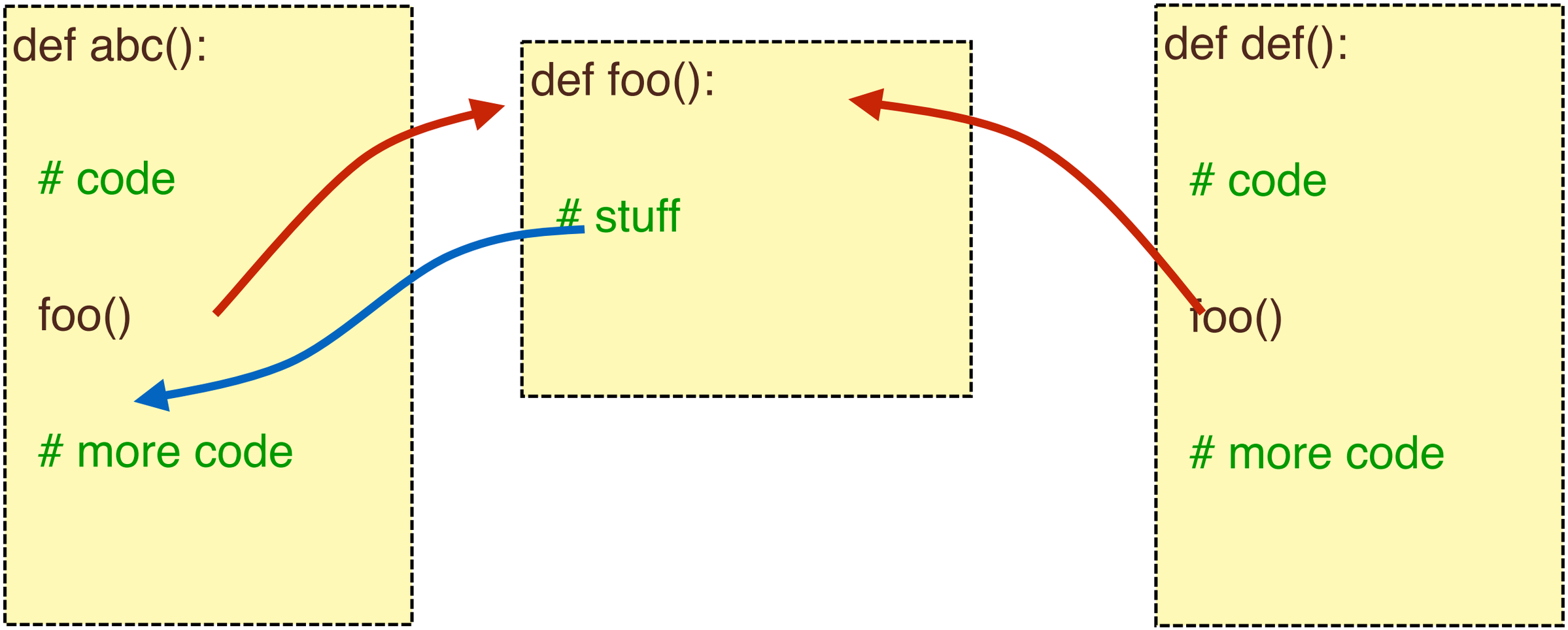
Function calling: return where?



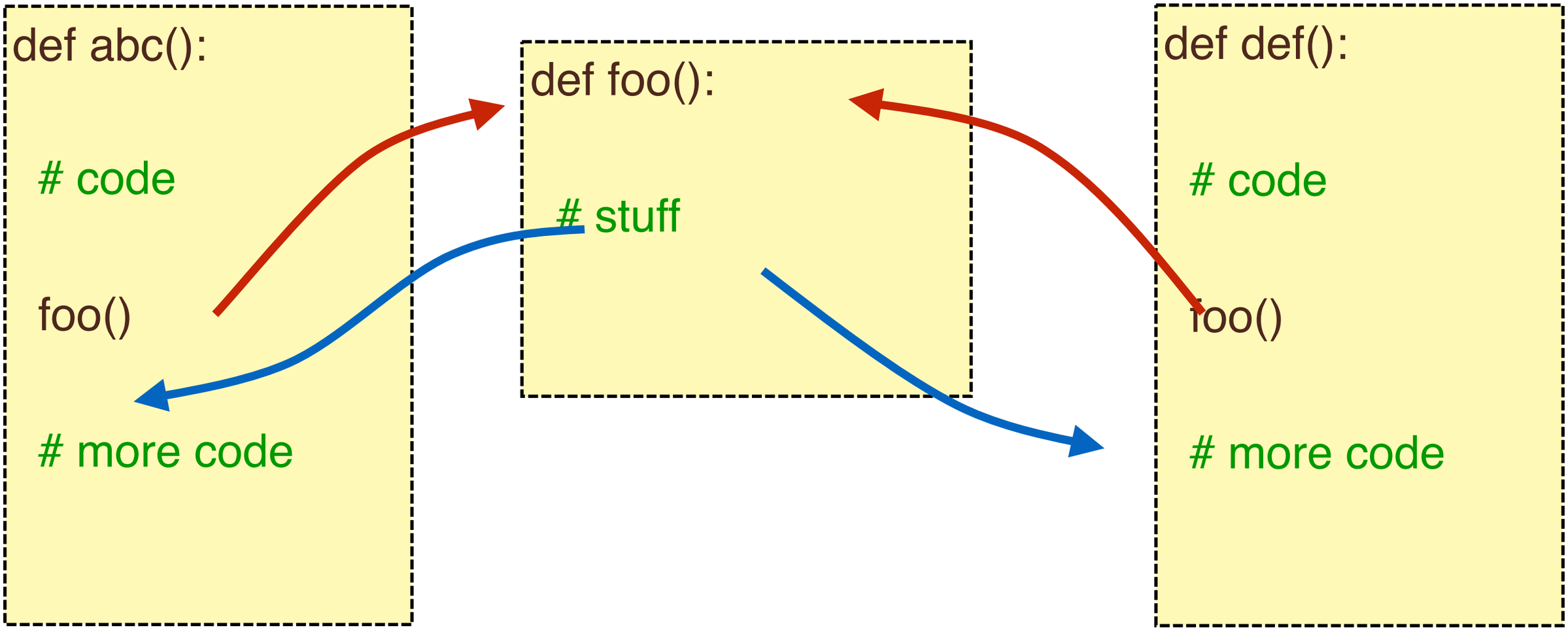
Function calling: return where?



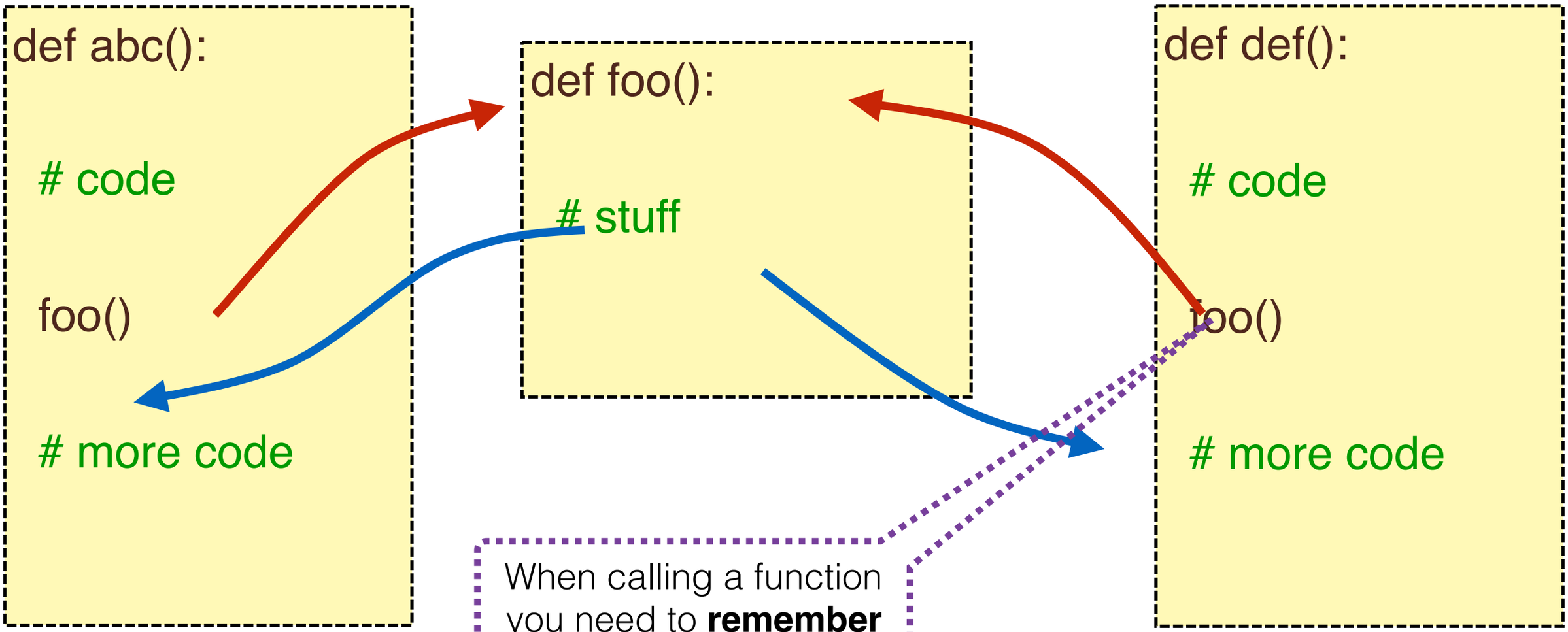
Function calling: return where?



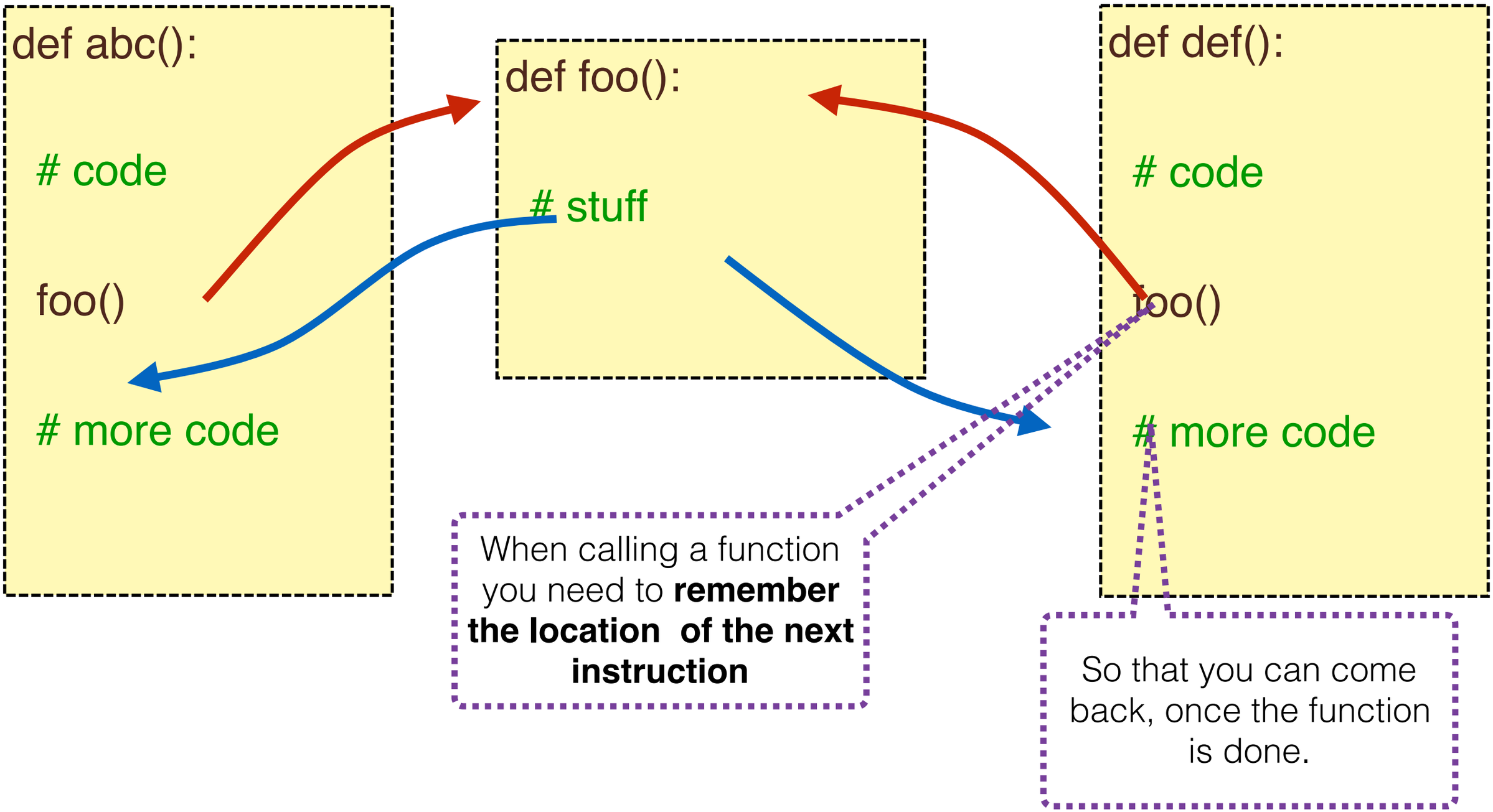
Function calling: return where?



Function calling: return where?



Function calling: return where?



Jump Instructions

- jump (go) to label, e.g.,

```
j foo          # set PC = foo
               # so, go to foo
```

- jump to label and link (remember origin), e.g.,

```
jal foo        # $ra = PC+4; PC = foo, so same
               # but setting a return address
```

- jump to address contained in register, e.g.,

```
jr $t0         # set PC=$t0, so go to the
               # address contained in $t0
```

- jump to register and link (remember origin), e.g.,

```
jalr $t0       # $ra = PC+4; PC = $t0, same
               # but setting a return address
```

sqr.py

```
def sqr(n):
    return n*n

print(sqr(int(input())))
```

Simple convention

```python
def sqr(n):
    return n*n

print(sqr(int(input())))
```

```
        .text

        addi    $v0, $0, 5      # read integer
        syscall

        add     $a0, $0, $v0    # $a0 = $v0
        jal     sqr             # $v0 = sqr($a0)

        add     $a0, $0, $v0    # $a0 = $v0
        addi    $v0, $0, 1      # print $a0
        syscall

        addi    $v0, $0, 10     # exit
        syscall

sqr:    mult    $a0, $a0        # LO = $a0*$a0
        mflo    $v0             # $v0 = LO
        jr      $ra             # return $v0
```

```python
def sqr(n):
    return n*n

print(sqr(int(input())))
```

Recall:
jal stores
PC + 4 in $ra

```
        .text

        addi    $v0, $0, 5      # read integer
        syscall

        add     $a0, $0, $v0    # $a0 = $v0
        jal     sqr             # $v0 = sqr($a0)

        add     $a0, $0, $v0    # $a0 = $v0
        addi    $v0, $0, 1      # print $a0
        syscall

        addi    $v0, $0, 10     # exit
        syscall

sqr:    mult    $a0, $a0        # LO = $a0*$a0
        mflo    $v0             # $v0 = LO
        jr      $ra             # return $v0
```

```python
def sqr(n):
    return n*n

print(sqr(int(input())))
```

Recall:
jal stores
PC + 4 in $ra

```
.text

        addi    $v0, $0, 5      # read integer
        syscall

        add     $a0, $0, $v0    # $a0 = $v0
        jal     sqr             # $v0 = sqr($a0)

        add     $a0, $0, $v0    # $a0 = $v0
        addi    $v0, $0, 1      # print $a0
        syscall

        addi    $v0, $0, 10     # exit
        syscall

sqr:    mult    $a0, $a0        # LO = $a0*$a0
        mflo    $v0             # $v0 = LO
        jr      $ra             # return $v0
```

Function calling in MIPS

To **write** a function

- ➡ Put **label** at the **start** of the function
- ➡ Write body of the function
- ➡ **End** function with jr $ra

To **call** a function

- ➡ Write jal label
- ➡ When the function returns, program will continue from the next instruction

Passing data

- Some functions take **parameters**. We need a way of passing parameters from caller to function.

- Some functions **return values**. We need a way of getting the return value safely back to caller.

- Reserve some **registers** for these tasks
  - We can use the **“syscall”** data passing method.
  - Pass function **parameters** in **$a0, $a1, $a2, $a3.**
  - **Return** values in **$v0, $v1**

sqr.py

```
def sqr(n):
    return n*n

print(sqr(int(input())))
```

sqr.py

```
def sqr(n):
    return n*n

print(sqr(int(input())))
```

Only one argument

No other local variables

No function calls

Single value returned

Limitations

This simple function-calling convention works, but has limits

- Function must not call other functions
- Function call is limited to four arguments (**$a0**-**$a3**)
- Function must only write to “safe” registers **$v0**-**$v1, $a0**-**$a3, $t0**-**$t9**
- Function must not use local variables, only arguments
- Function can only return two values **$v0** and **$v1**

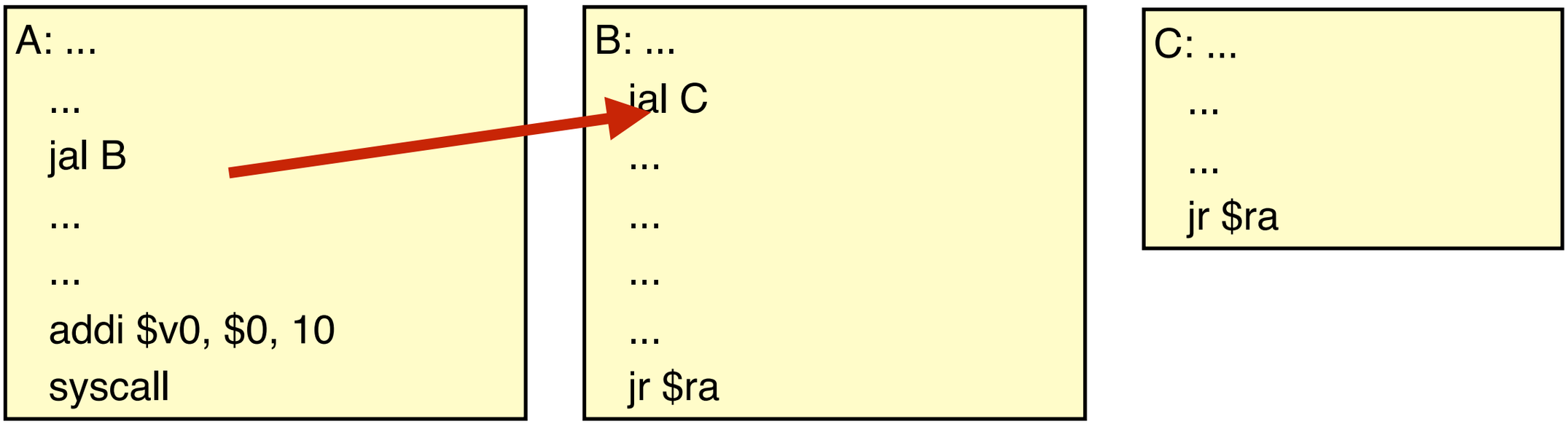
Function can call other functions

```
A: ...
   ...
   jal B
   ...
   ...
   addi $v0, $0, 10
   syscall
```

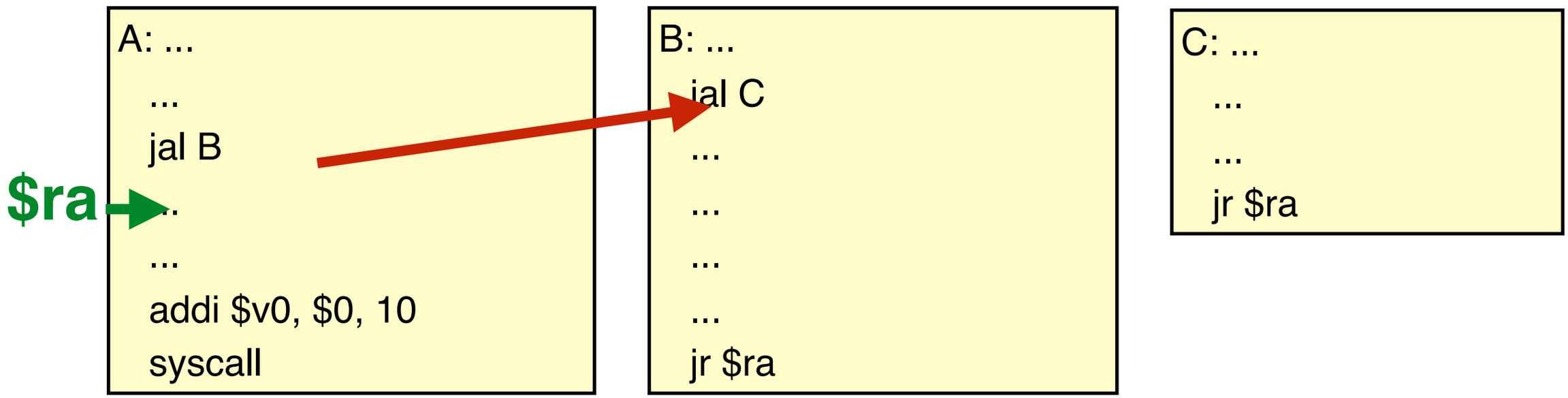
```
B: ...
   jal C
   ...
   ...
   ...
   ...
   jr $ra
```

```
C: ...
   ...
   ...
   jr $ra
```

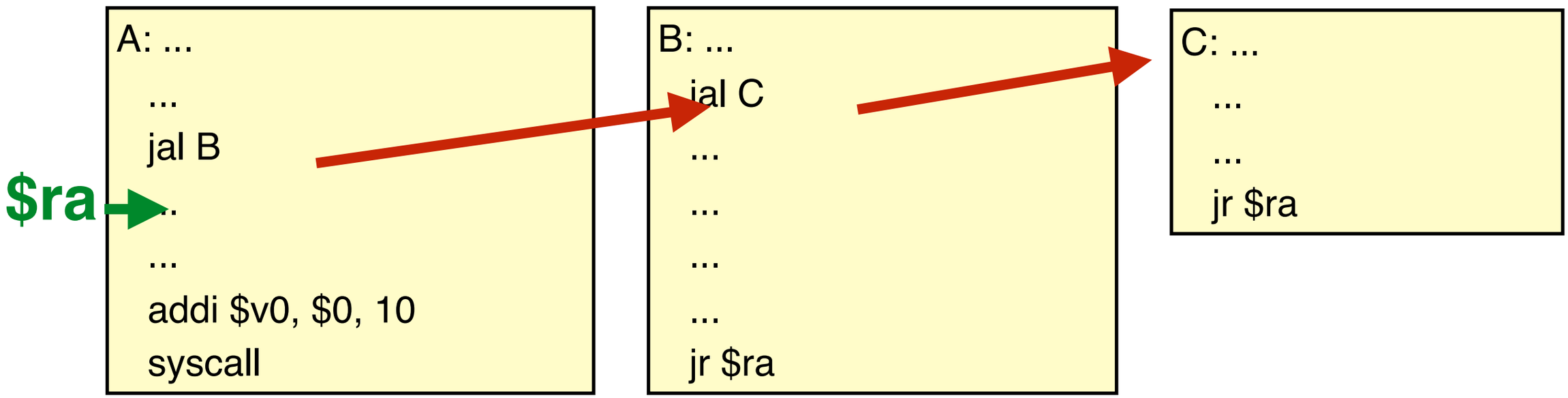
Function can call other functions



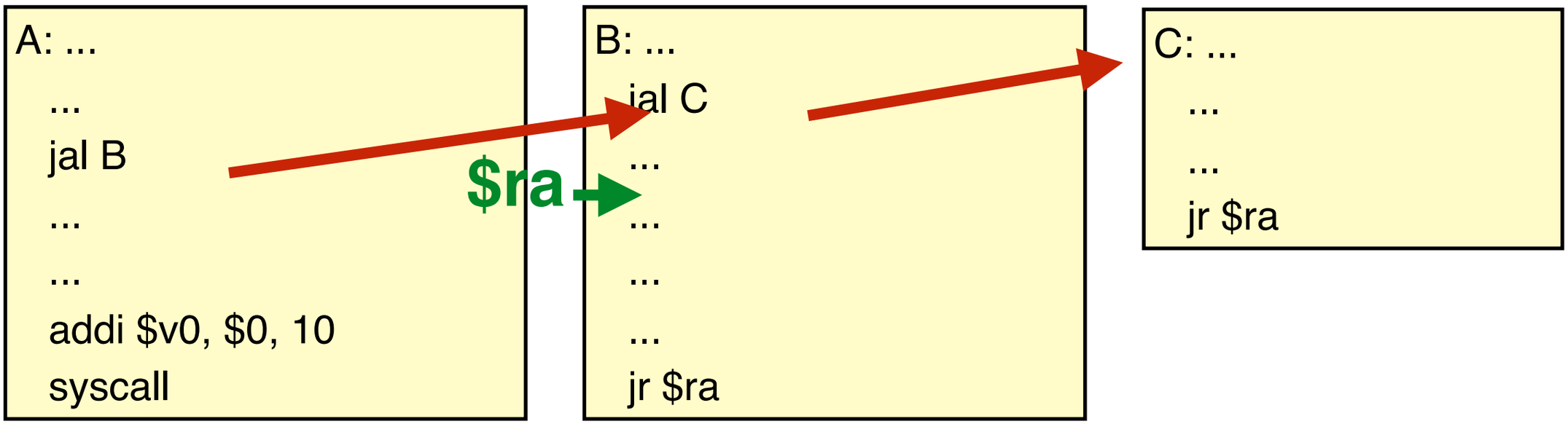
Function can call other functions



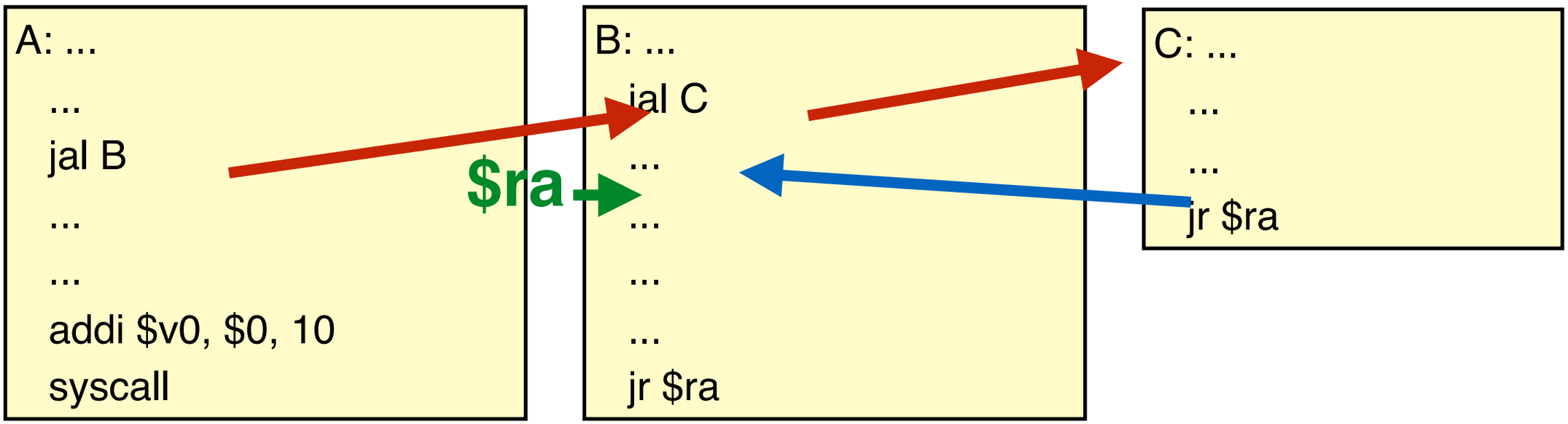
Function can call other functions



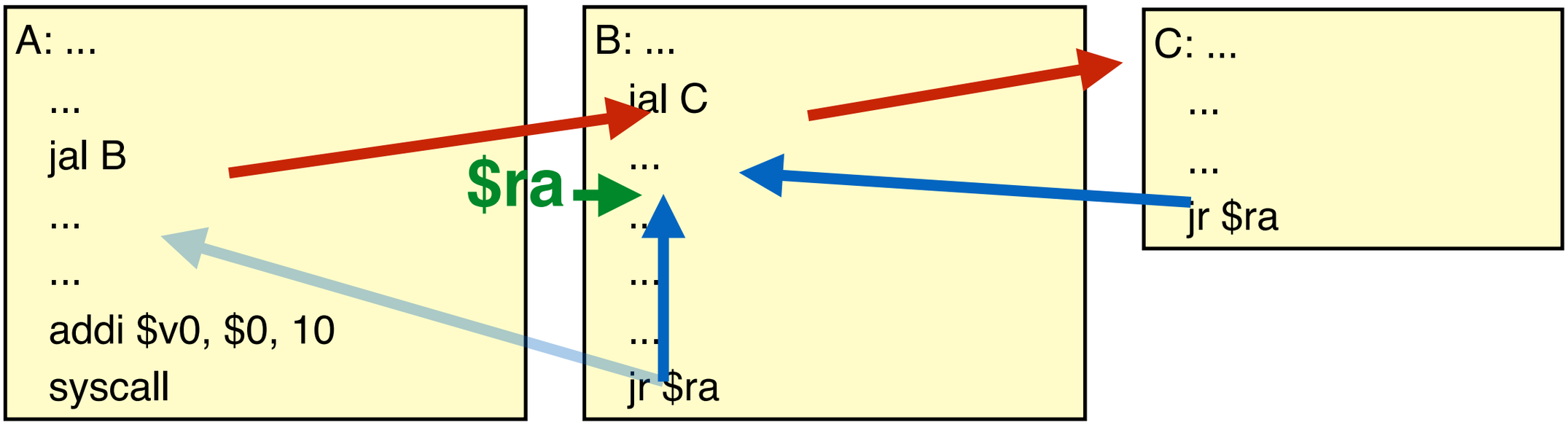
Function can call other functions



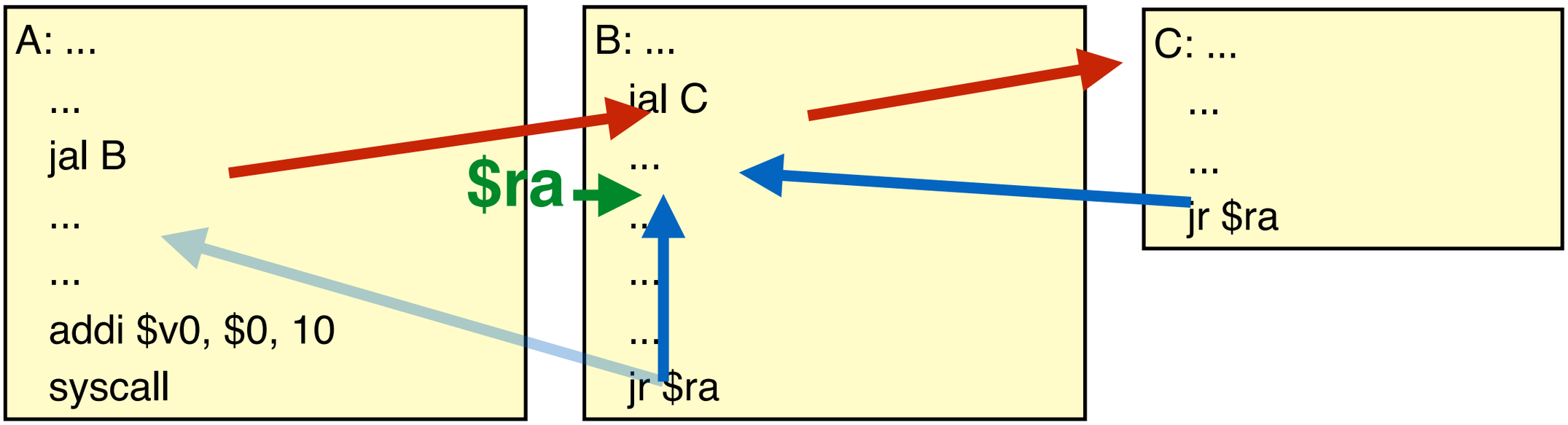
Function can call other functions



Function can call other functions



Function can call other functions



Original **$ra** is lost!

Original **$ra** is lost!

```
A: ...
   ...
   jal B
   ...
   ...
   addi $v0, $0, 10
   syscall
```

```
B: #push $ra
   jal C
   ...
   ...
   ...
   #pop into $ra
   jr $ra
```

```
C: #push $ra
   ...
   #pop into $ra
   jr $ra
```

Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.

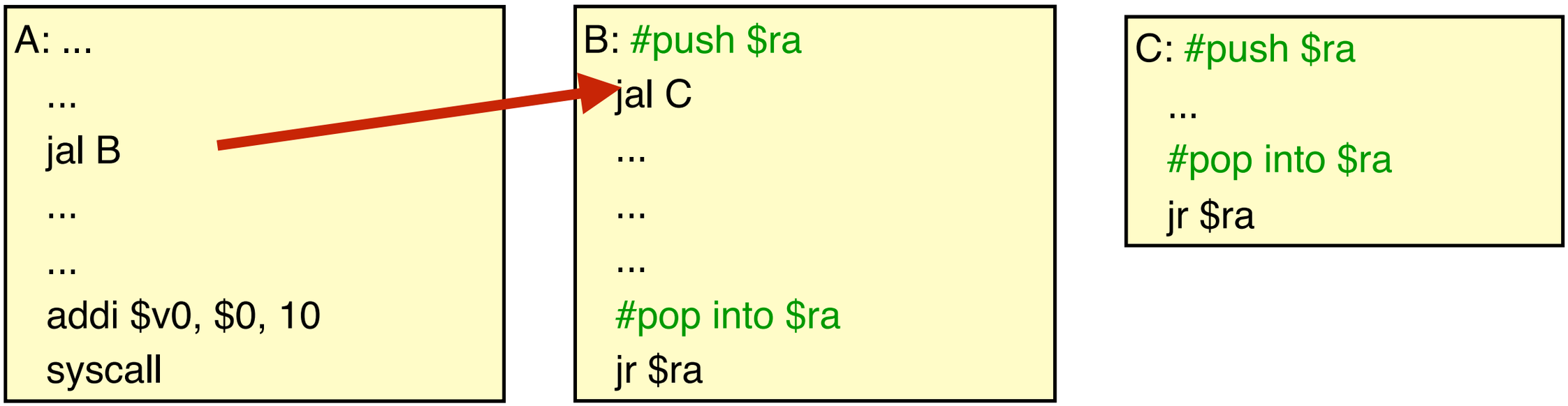
```
A: ...
   ...
   jal B
   ...
   ...
   addi $v0, $0, 10
   syscall
```

```
B: #push $ra
   jal C
   ...
   ...
   ...
   #pop into $ra
   jr $ra
```

```
C: #push $ra
   ...
   #pop into $ra
   jr $ra
```

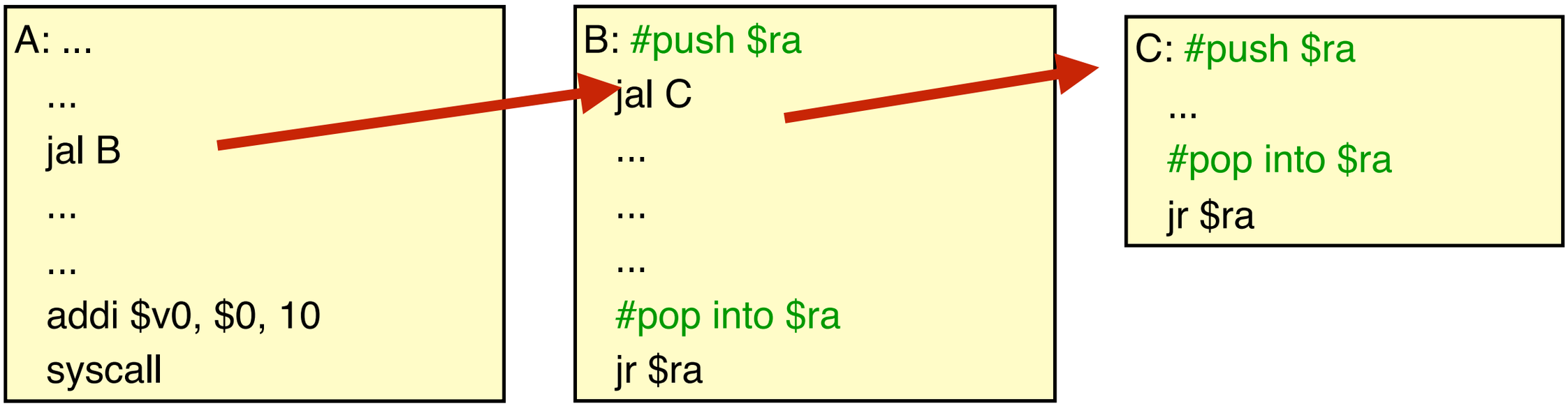
Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.



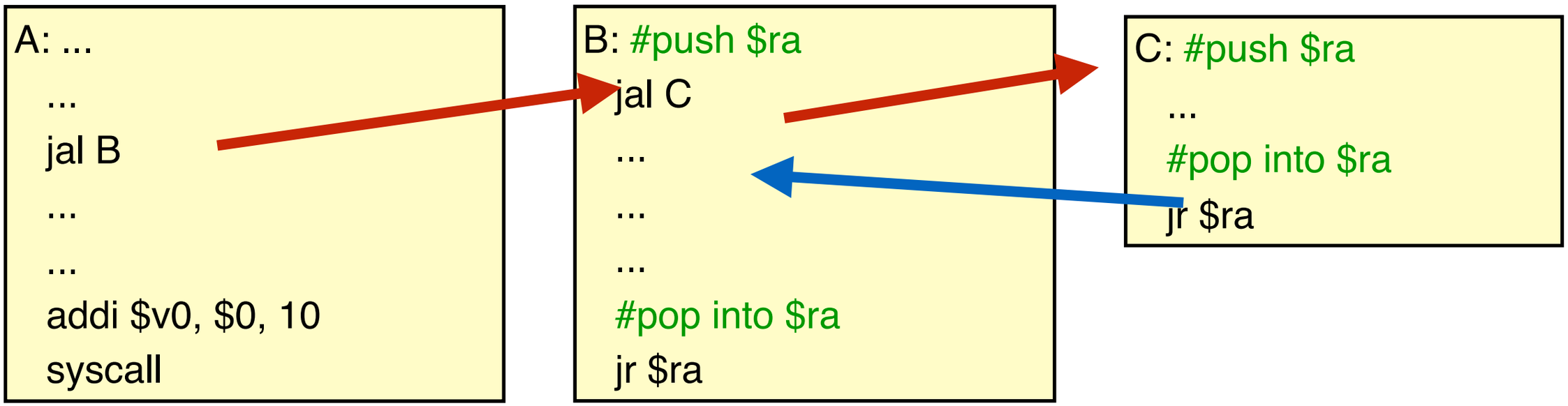
Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.



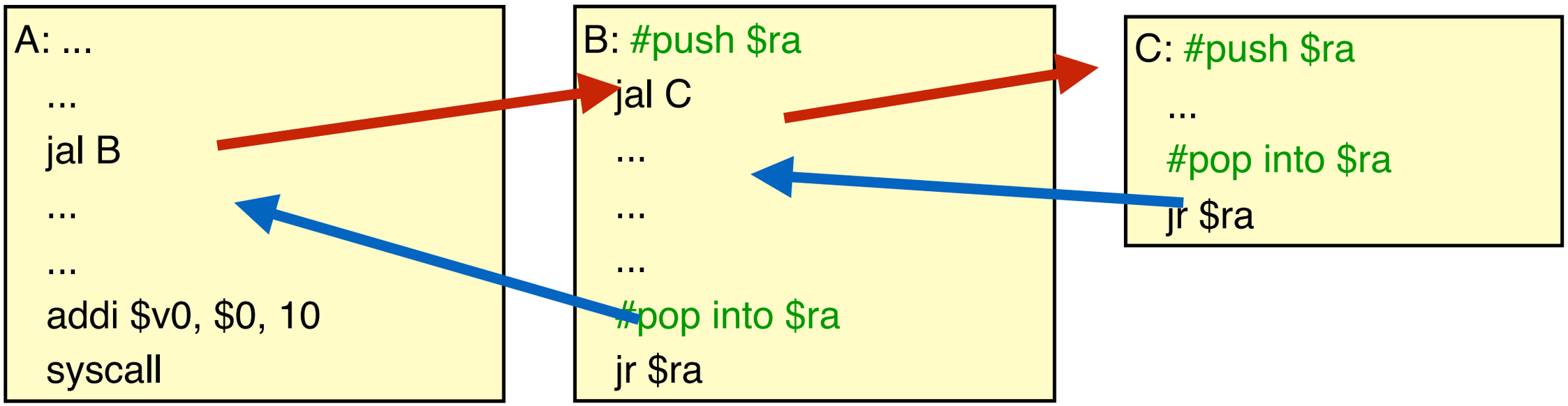
Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.



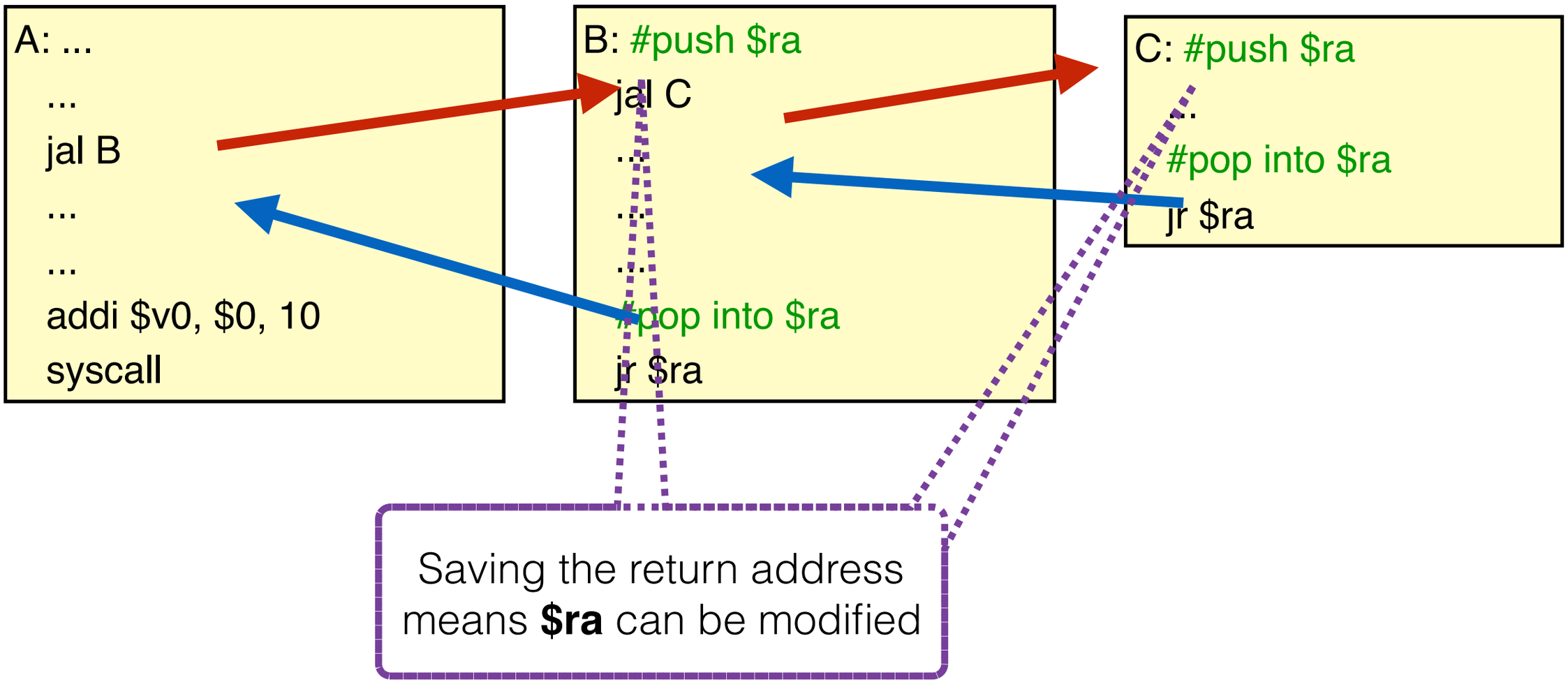
Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.



Original **$ra** is lost!

Solution: Save and restore $ra register on the stack upon function entry/exit.



Too many arguments

```
addi $a0, $0, 1
lw $a1, x
addi $a2, $0, 0
lw $a3, -4($fp)
addi $??, $0, 2
jal five
...
```

```
five: # takes 5
      # parameters
      ...
      # examine
      # $a0, etc
      ...
      jr $ra
```

Too many arguments

```
addi $a0, $0, 1
lw $a1, x
addi $a2, $0, 0
lw $a3, -4($fp)
addi $??, $0, 2
jal five
...
```

No such register **$a4**

```
five: # takes 5
      # parameters
      ...
      # examine
      # $a0, etc
      ...
      jr $ra
```

Too many arguments

```
addi $a0, $0, 1
lw $a1, x
addi $a2, $0, 0
lw $a3, -4($fp)
addi $??, $0, 2
jal five
...
```

```
five: # takes 5
      # parameters
      ...
      # examine
      # $a0, etc
      ...
      jr $ra
```

No such register **$a4**

Not enough registers to use as function arguments

Not enough registers to use as function arguments

```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```

Not enough registers to use as function arguments

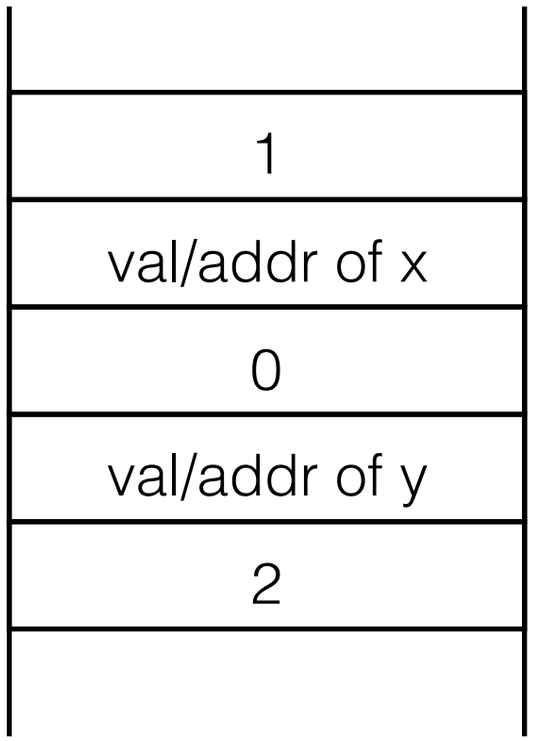
Solution: Save arguments on the stack

```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```

Not enough registers to use as function arguments

Solution: Save arguments on the stack

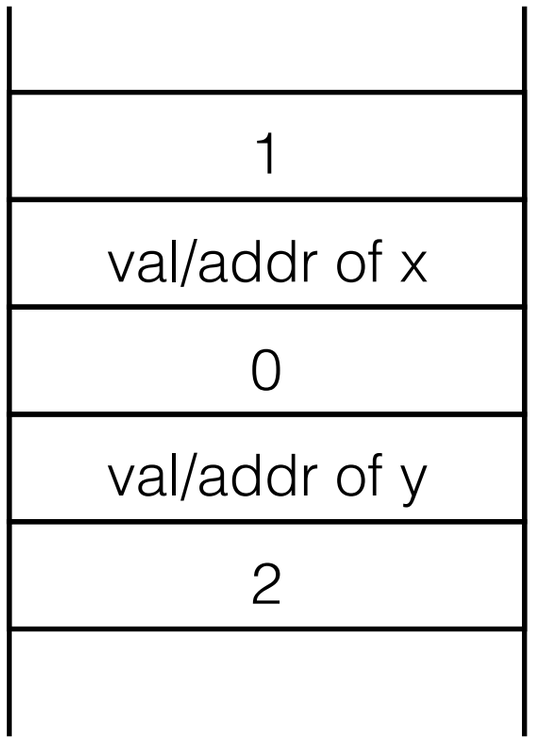
```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```



Not enough registers to use as function arguments

Solution: Save arguments on the stack

```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```

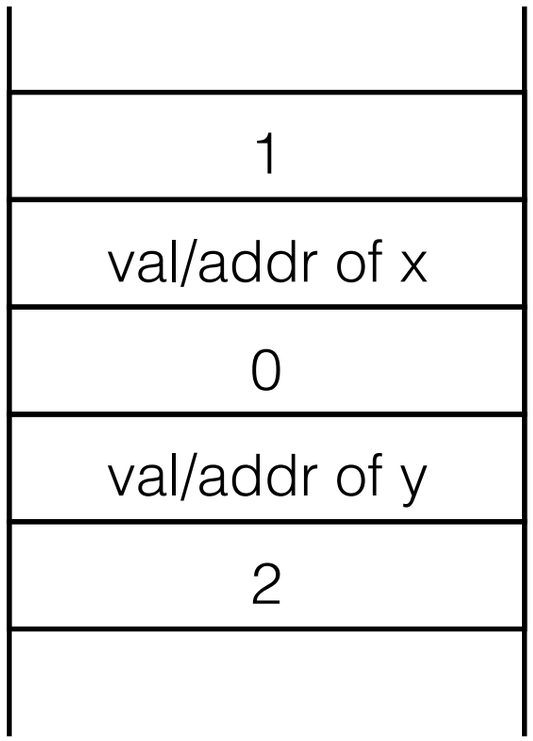



```
five: # takes 5
      # parameters
      ...
      # examine
      # stack
      ...
      jr $ra
```

Not enough registers to use as function arguments

Solution: Save arguments on the stack

```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```




```
five: # takes 5
      # parameters
      ...
      # examine
      # stack
      ...
      jr $ra
```

FIT1008:

For simplicity we will use the stack to pass **all** arguments

Saving registers

```
...
lw $t0, a
...
...
jal func
...
...
# $t0 has been
# changed!
add $t0, $t0, $v0
...
```

```
func: ...
      # trashes
      # $t0
      lw $t0, x
      ...
      jr $ra
```

Function may use registers which hold important values.

Saving registers

```
...
lw $t0, a
...
...
jal func
...
...
# $t0 has been
# changed!
add $t0, $t0, $v0
...
```

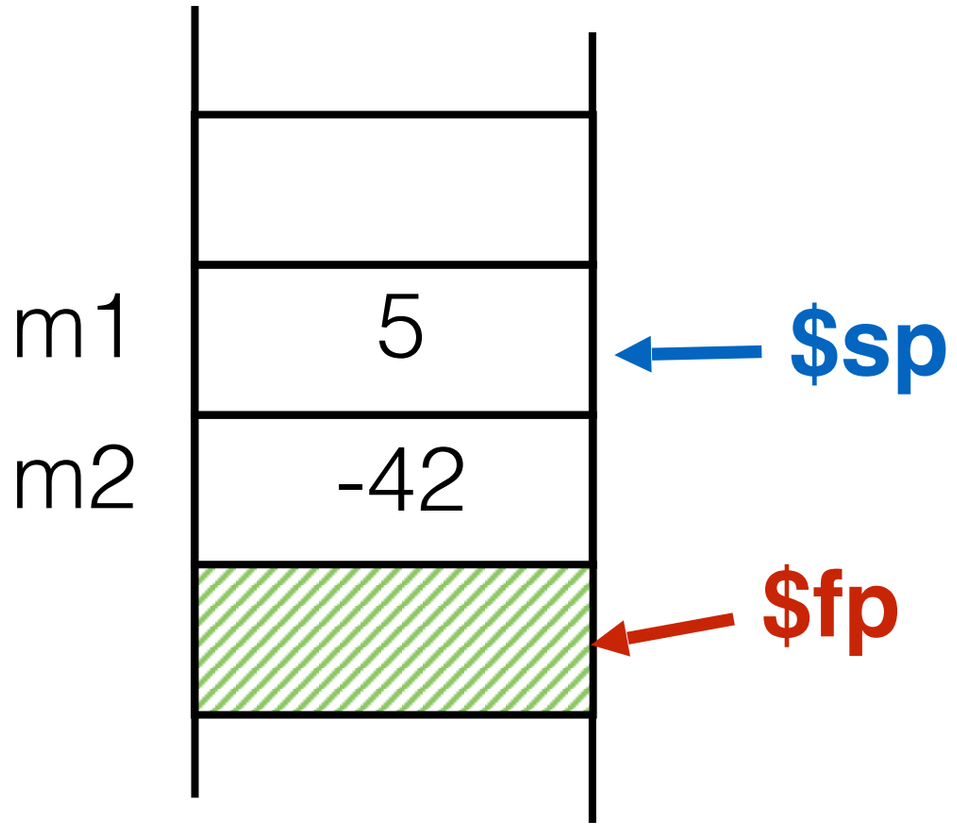
```
func: ...
      # trashes
      # $t0
      lw $t0, x
      ...
      jr $ra
```

Function may use registers which hold important values.

Solution: save/restore registers on stack.

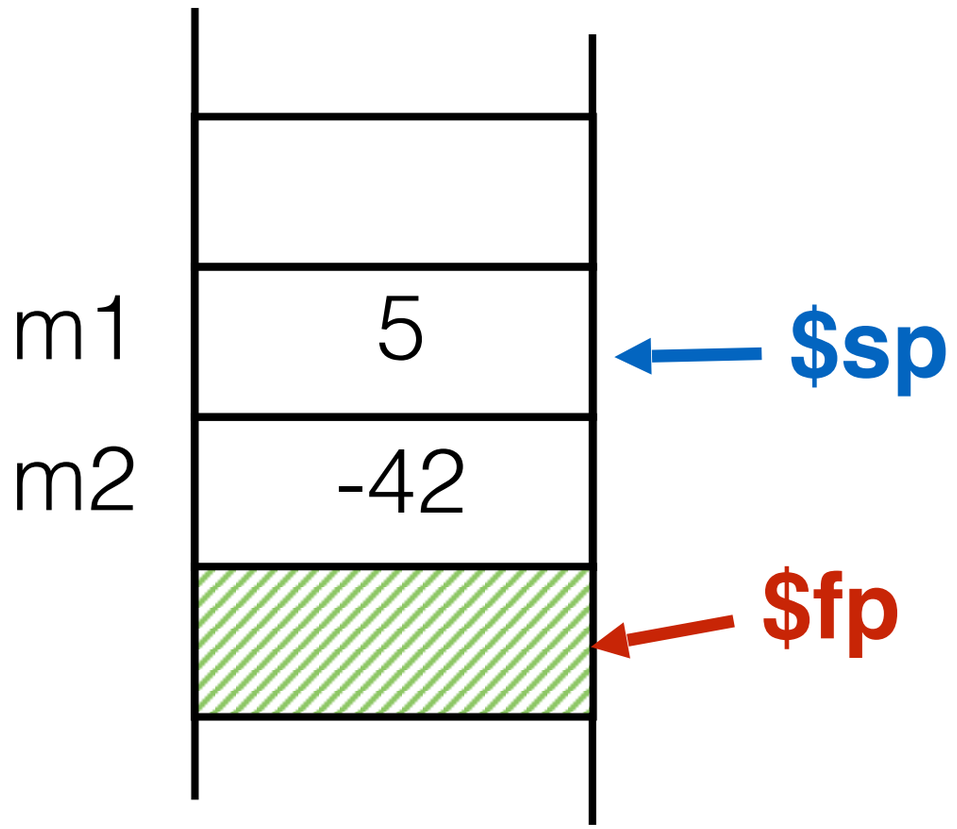
Local variables needed

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```



Allocate local variables

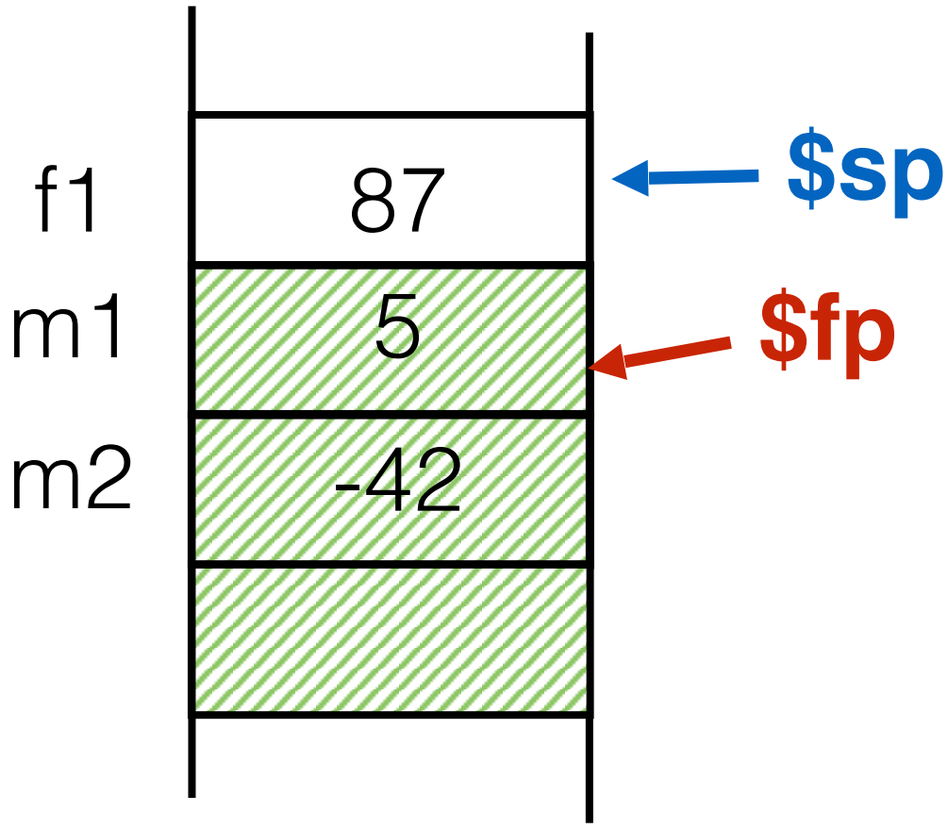
```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```




```
def func():

  f1 = 87
  ...
```

Allocate local variables

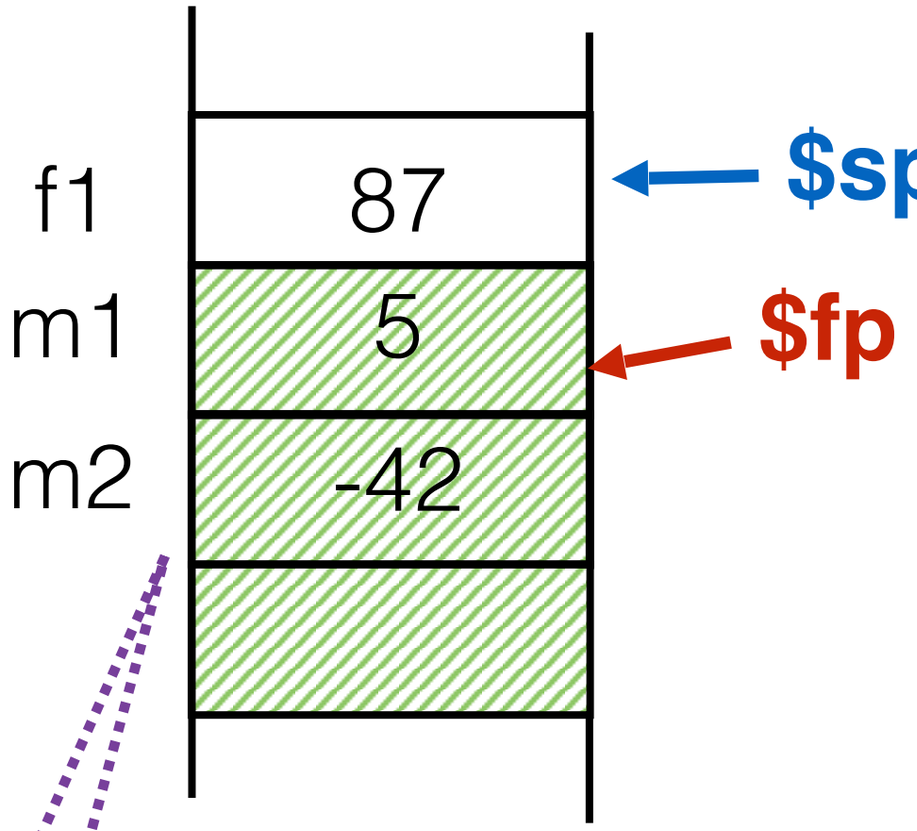
```
def main():

  m1 = 5
  m2 = -42

   ...
  func()
```




```
def func():

  f1 = 87

  ...
```

Allocate local variables

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```

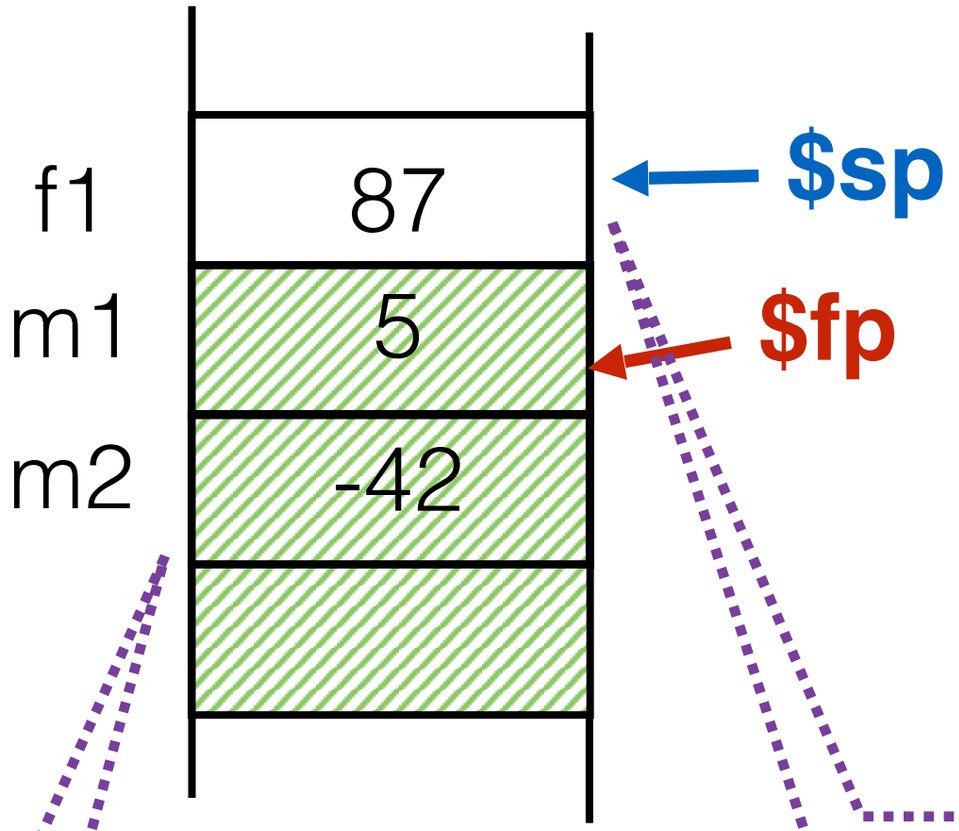



```
def func():

  f1 = 87
  ...
```

Inside **func(), m1** and **m2** are inaccessible

Allocate local variables

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```



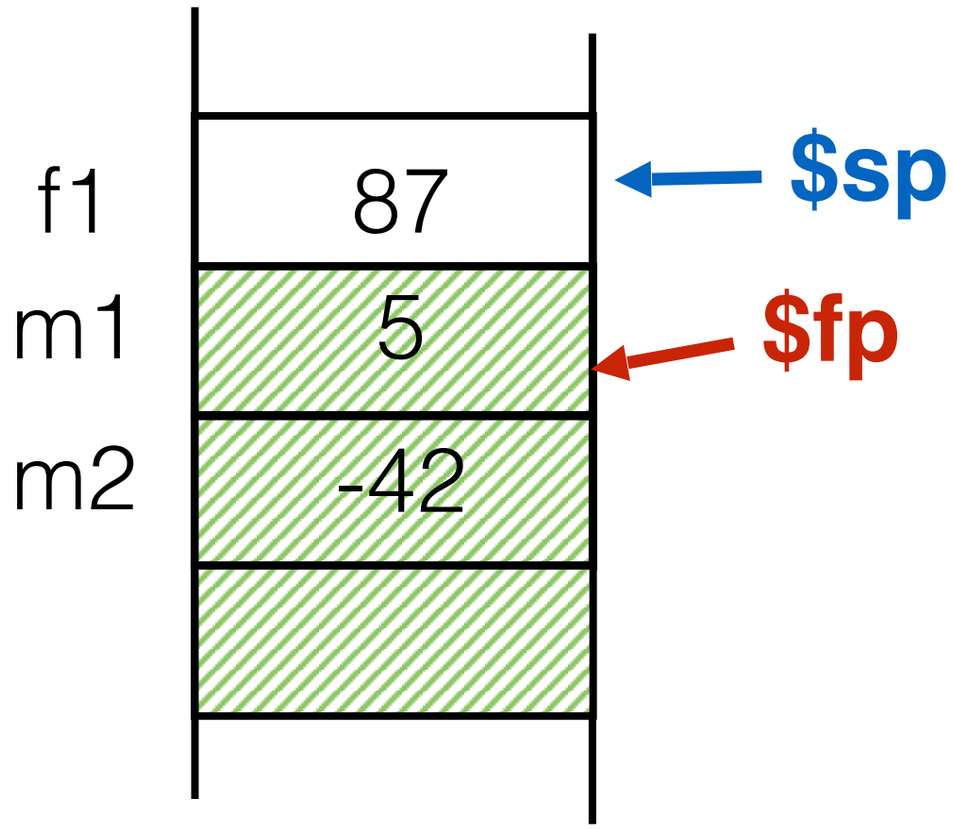

```
def func():

  f1 = 87
  ...
```

Inside **func(), m1** and **m2** are inaccessible

Inside **func(), f1** is at **-4(fp)**

Restoring stack state

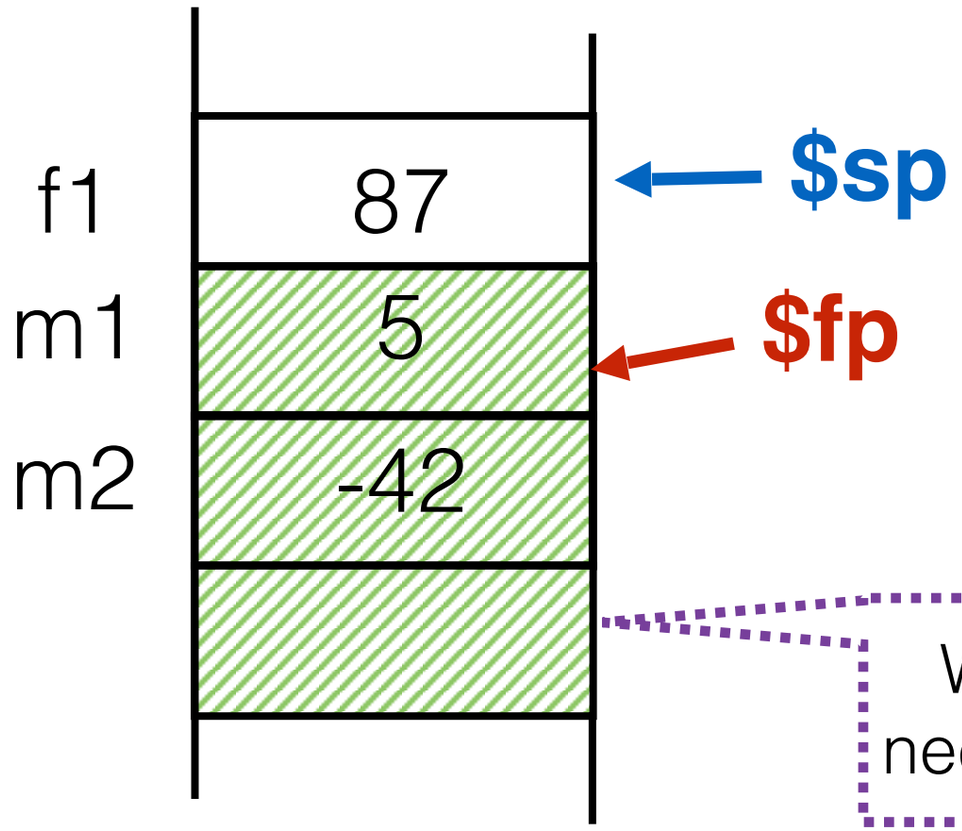
```
def main():

  m1 = 5
  m2 = -42
  ...
  func()
```




```
def func():

  f1 = 87
  ...
```

Restoring stack state

```
def main():

  m1 = 5
  m2 = -42
  ...
func()
```

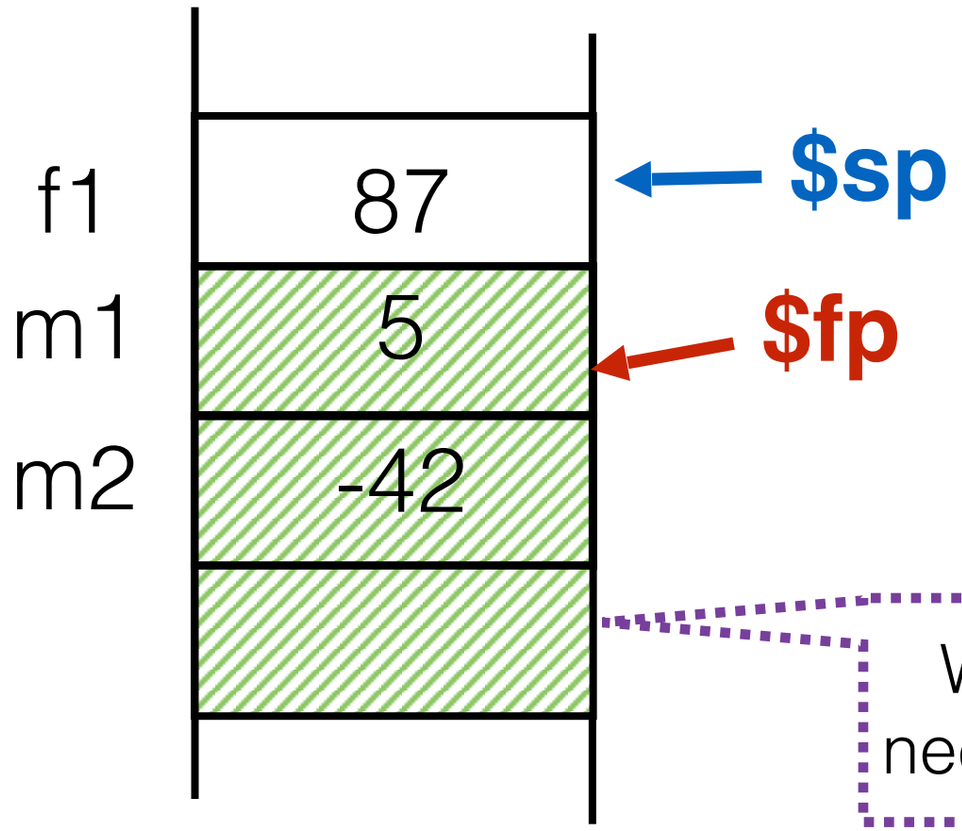



```
def func():

  f1 = 87
  ...
```

When **func()** ends **$fp** needs to move **back here**

Restoring stack state

```
def main():

  m1 = 5
  m2 = -42
  ...
func()
```



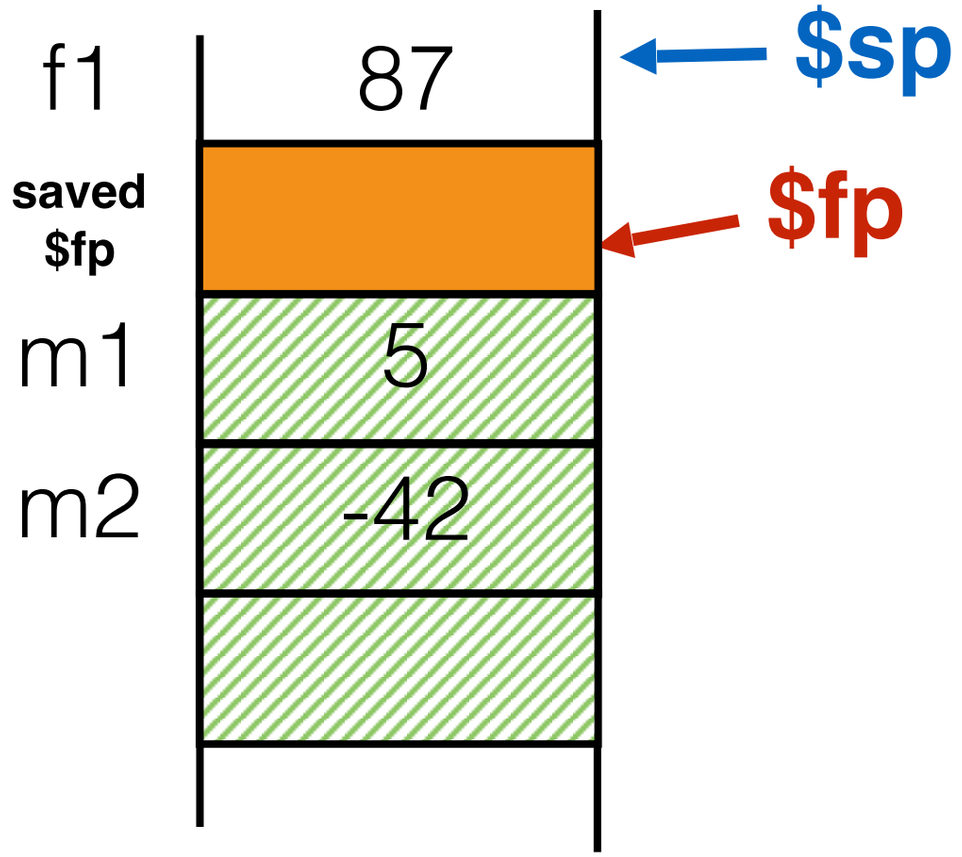

```
def func():

  f1 = 87
  ...
```

When **func()** ends **$fp** needs to move **back here**

Stack must be restored on function return

Stack must be restored on function return

```
def main():

  m1 = 5
  m2 = -42
  ...
  func()
```

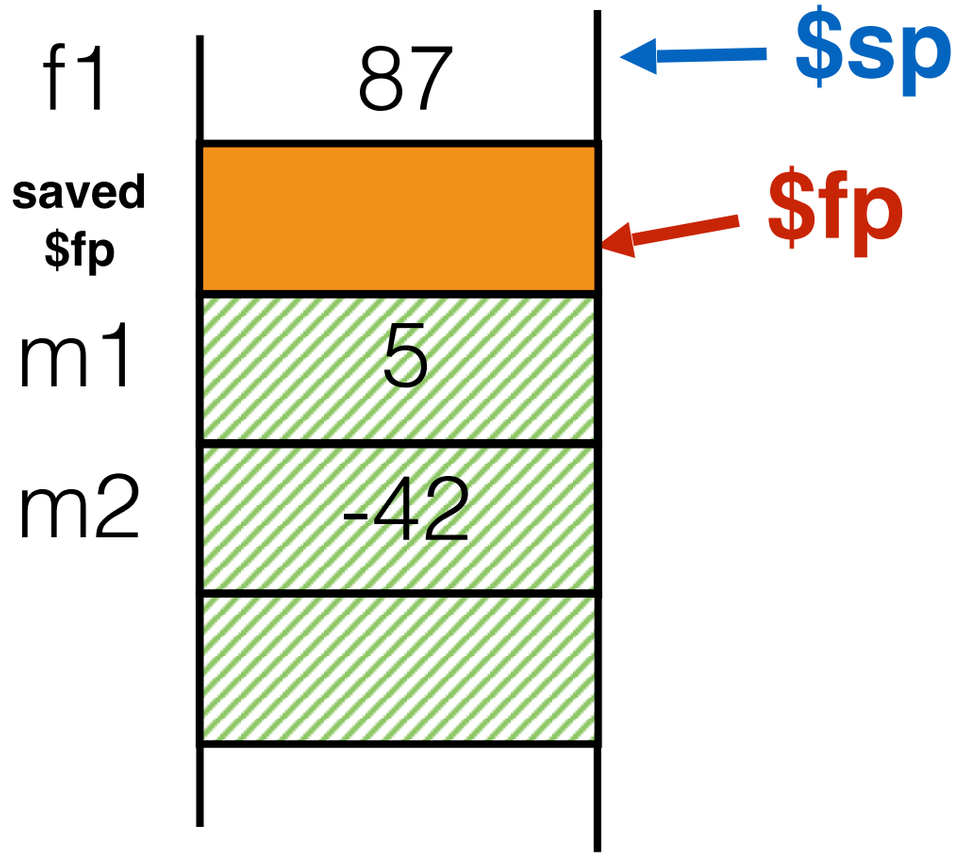



```
def func():
  # save $fp
  f1 = 87
  # restore $fp
  ...
```

Stack must be restored on function return

Solution: Save restore $fp on stack

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```

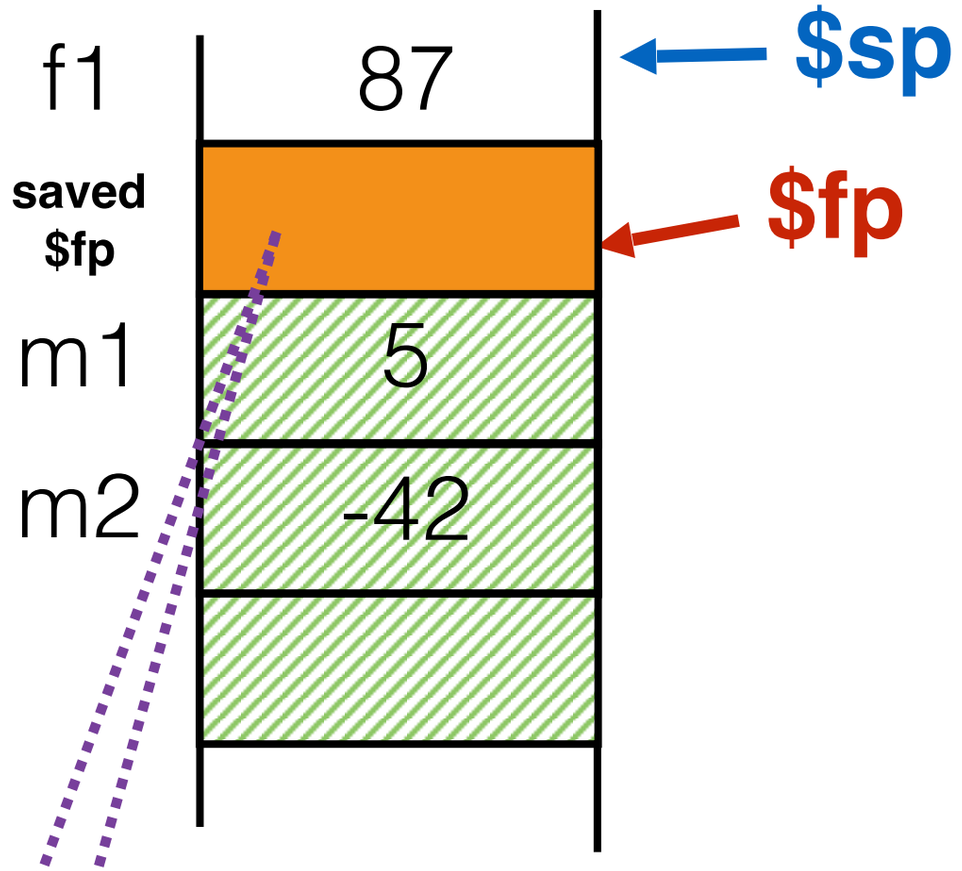



```
def func():
  # save $fp
  f1 = 87
  # restore $fp
  ...
```

Stack must be restored on function return

Solution: Save restore $fp on stack

```
def main():

  m1 = 5
  m2 = -42
  ...
  func()
```



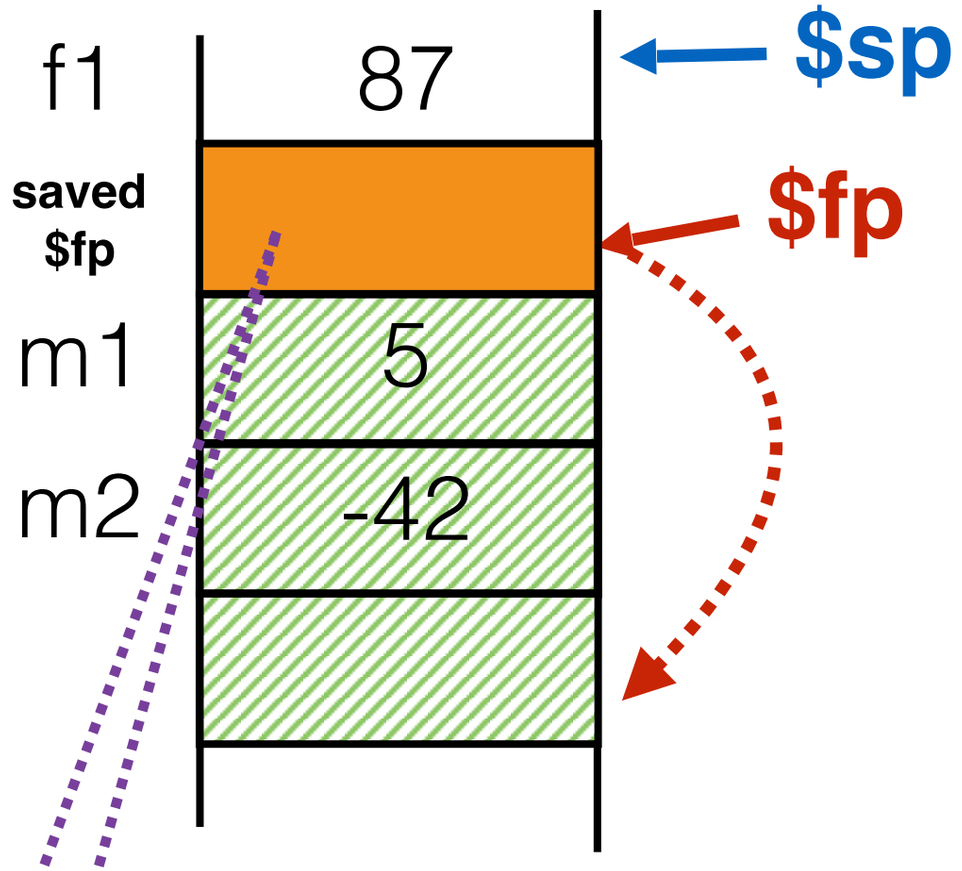

```
def func():
  # save $fp
  f1 = 87
  # restore $fp
  ...
```

By saving old **$fp** we can restore the stack state at the end of the function

Stack must be restored on function return

Solution: Save restore $fp on stack

```
def main():

  m1 = 5
  m2 = -42
  ...
  func()
```




```
def func():
  # save $fp
  f1 = 87
  # restore $fp
  ...
```

By saving old **$fp** we can restore the stack state at the end of the function

Convention

Function calling convention

Function calling convention

These **steps** must be performed **every time** a function starts:

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments
3. Call function with **jal** instruction

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments
3. Call function with **jal** instruction
4. Save $ra register

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments
3. Call function with **jal** instruction
4. Save $ra register
5. Save $fp register

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments
3. Call function with **jal** instruction
4. Save $ra register
5. Save $fp register
6. Update $fp

Function calling convention

These **steps** must be performed **every time** a function starts:

1. Save temporary registers
2. Save arguments
3. Call function with **jal** instruction
4. Save $ra register
5. Save $fp register
6. Update $fp
7. Allocate local variables