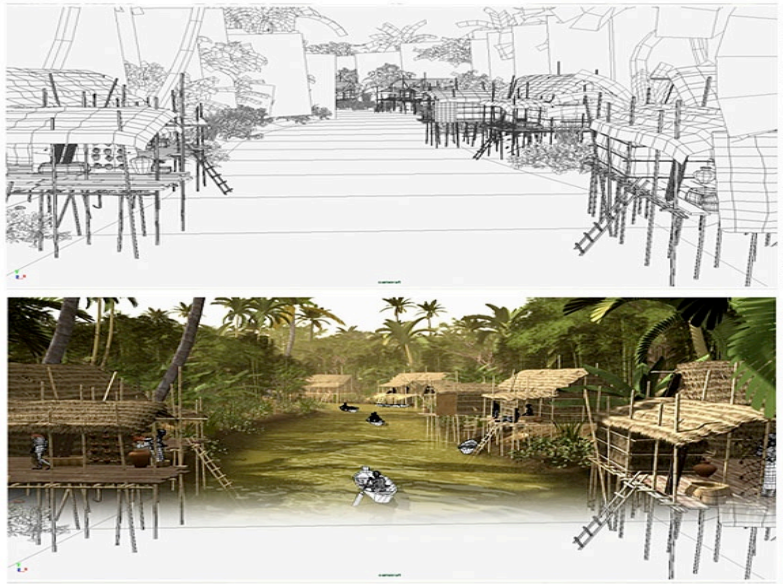
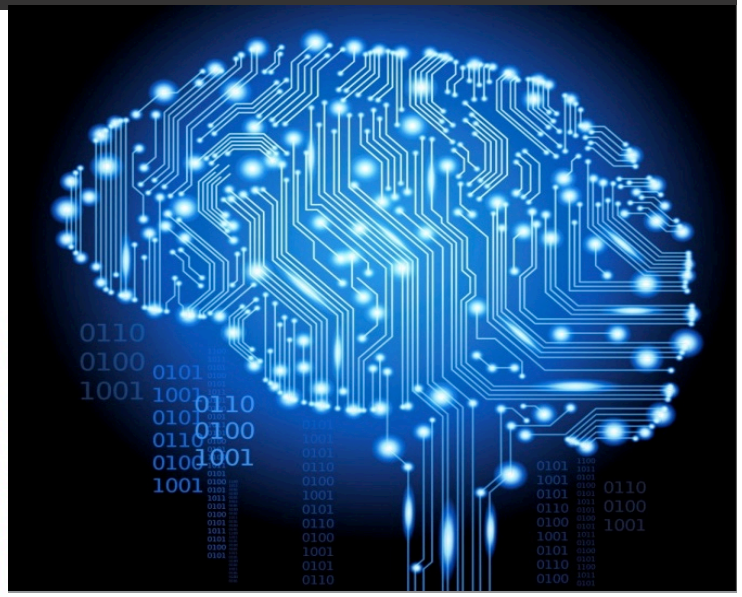
MONASH University

Information Technology

FIT1008/2085
MIPS – Function Return

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





Where are we up to?

- Discussed MIPS architecture and main design decisions
- Gone through the instruction set that we will use in this unit
- Practiced translation of decisions
- Discussed how to create and access arrays of integers
- Discussed how variables are stored and accessed
- Properties of functions and how they affect their implementation in MIPS
  - Call and return (`jal` and `jr`)
  - Cascade calls (save `$ra` in stack)
  - Access local variables (save `$fp` in stack)
  - Pass parameters (use stack)
- Calling convention (who does what, why and how it is done in MIPS)
- Structure of stack (stack frames)

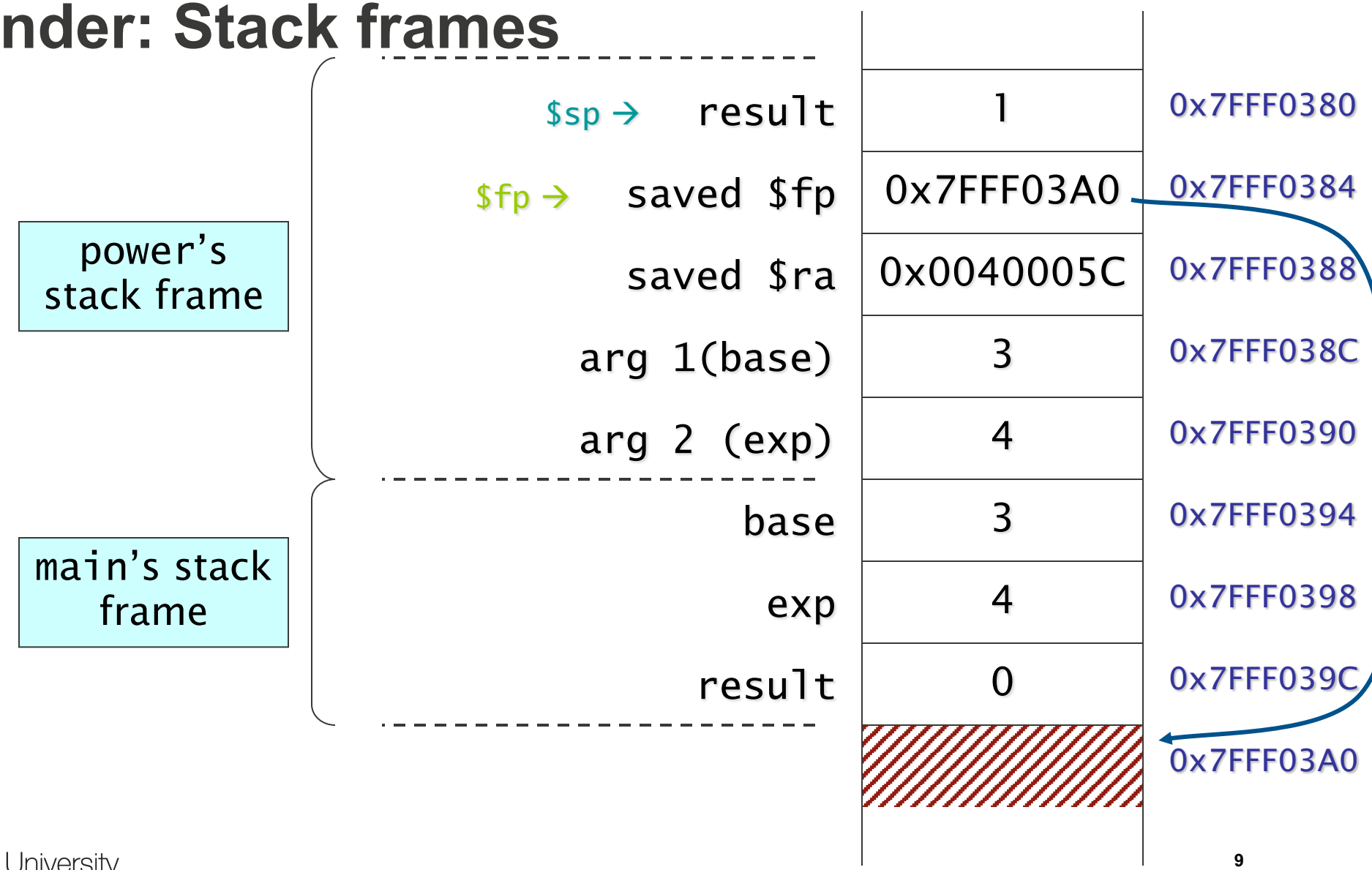
Learning objectives for this lecture

- To understand the steps for function return
- To be able to implement function return in MIPS
- To understand how recursion is implemented in MIPS

MONASH University

Function Return (callee point of view)

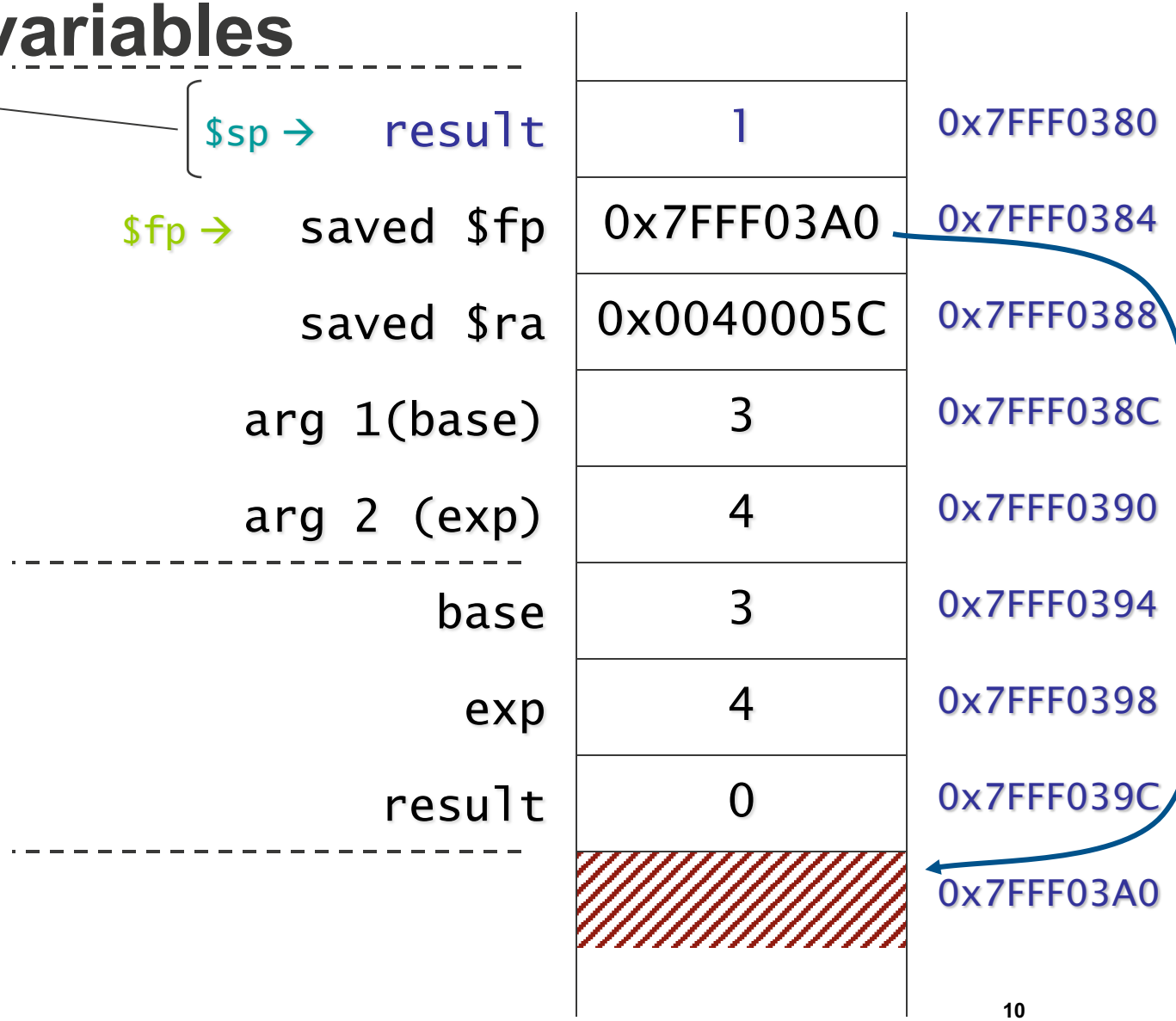
Reminder: Stack frames



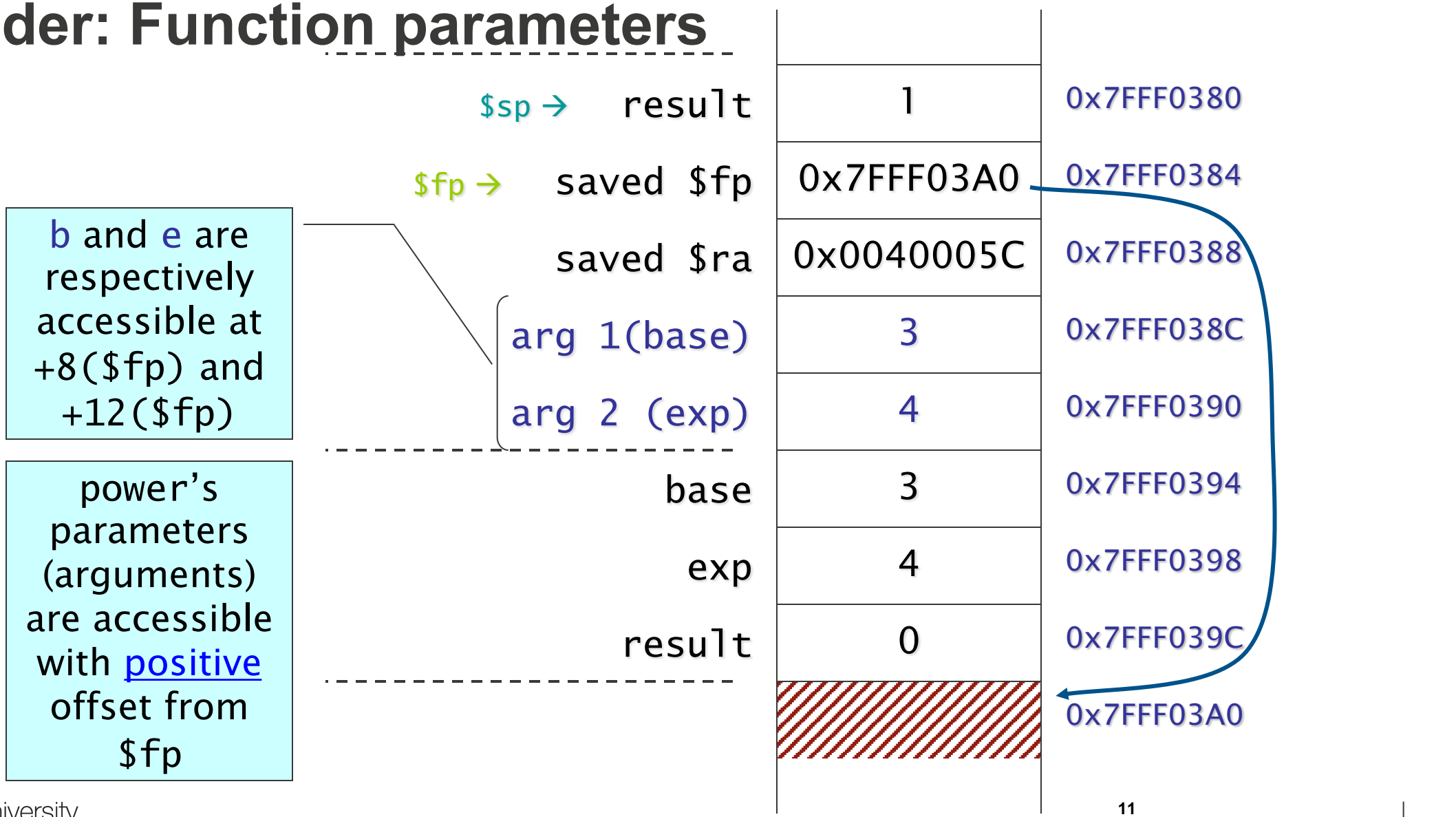
Reminder: Local variables

result is at -4($fp)

power's local variables are accessed as main's were, with negative offsets from $fp



Reminder: Function parameters



Example: callee

```
result is at -4($fp)
base is at 8($fp)
exp is at 12($fp)
```

```
""" Illustrating also function return. """

def power(base:int, exp:int) -> int:
    """ Computes base to the power of exp. """
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result
```

```
loop: # Stop if not (exp > 0)
      lw $t0, 12($fp)  # load exp
      slt $t0, $0, $t0 # is 0 < exp?
      beq $t0, 0, end  # if not go to end

      # result *= base
      lw $t0, -4($fp) # load result
      lw $t1, 8($fp)  # load base
      mult $t0, $t1   # base*result
      mflo $t0
      sw $t0, -4($fp) # result=base*result

      # exp -= 1
      lw $t0, 12($fp)  # load exp
      addi $t0, $t0, -1
      sw $t0, 12($fp)  # exp = exp-1

      # Repeat loop.
      j loop

end:  # Now ready to return.
# Continued in next lecture ...
```

Last lecture we left it at this point

Function return

- When returning from a function, the stack must be restored to its initial state
- This is achieved by undoing the steps made during calling of function, in reverse order

Function return convention

On return from function, caller:

4. Clears function arguments by popping allocated space
5. Restores saved temporary registers by popping their values off stack
6. Uses the return value found in $v0, if necessary

None of this needs to be memorised! It is all in the MIPS reference sheet (Moodle Week 1)

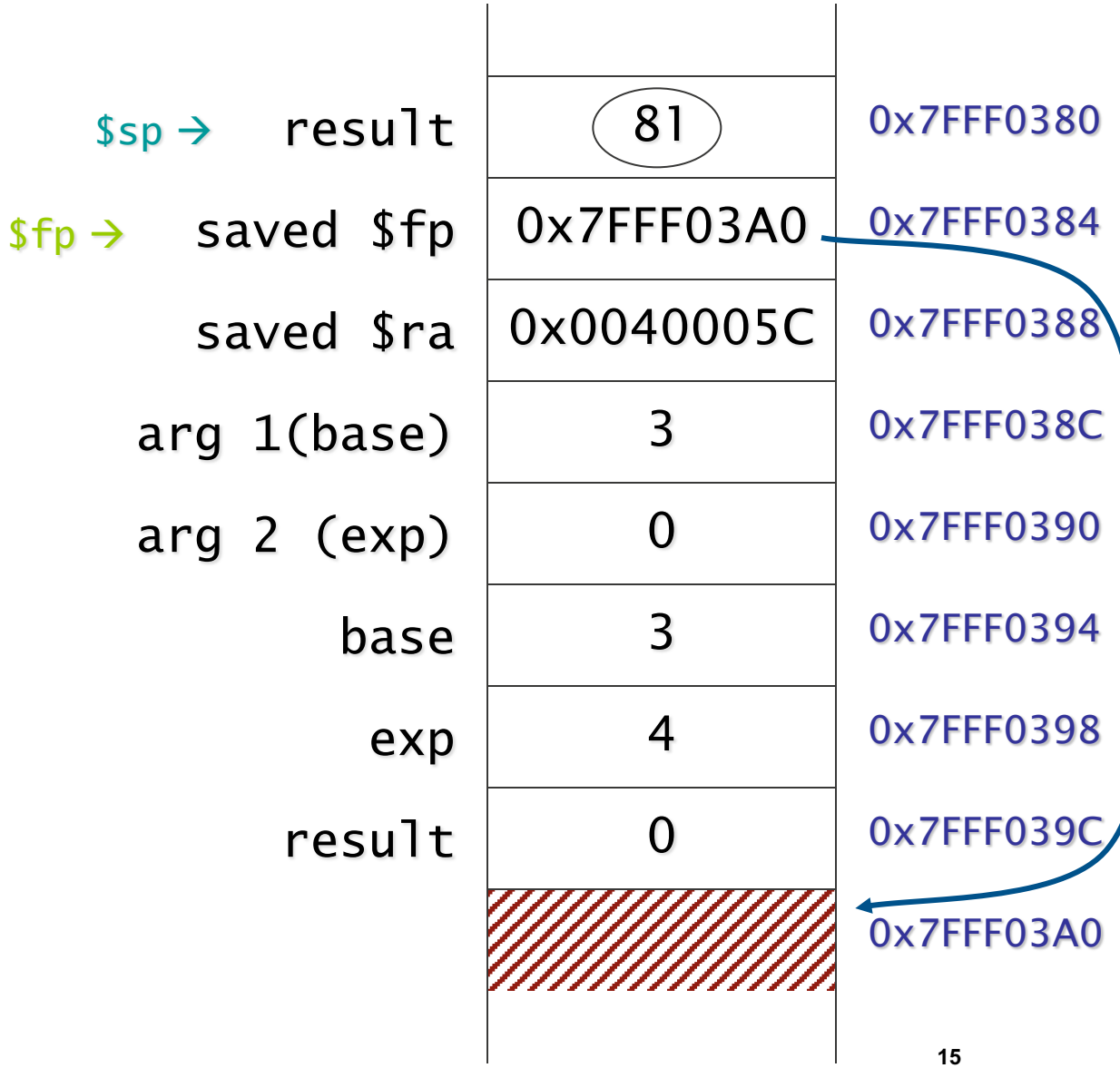
On function exit, callee:

5. Chooses return value by setting register `$v0`, if necessary
6. Deallocates local variables by popping allocated space
7. Restores `$fp` by popping its saved value off stack
8. Restores `$ra` by popping its saved value off stack
9. Returns with `jr $ra`

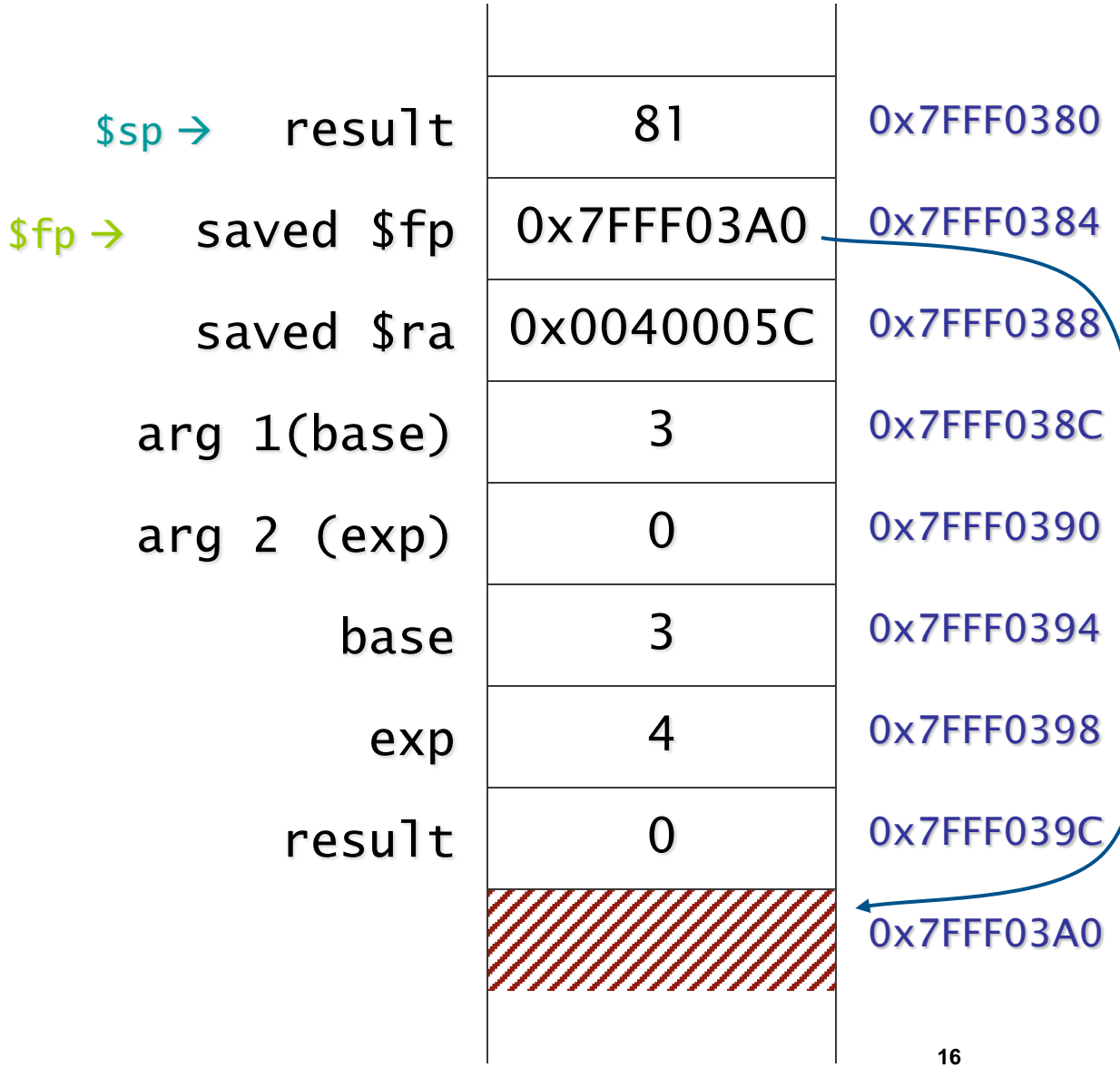
Example: callee

$v0 = 81

callee step 5: put return value in register $v0



Example: callee

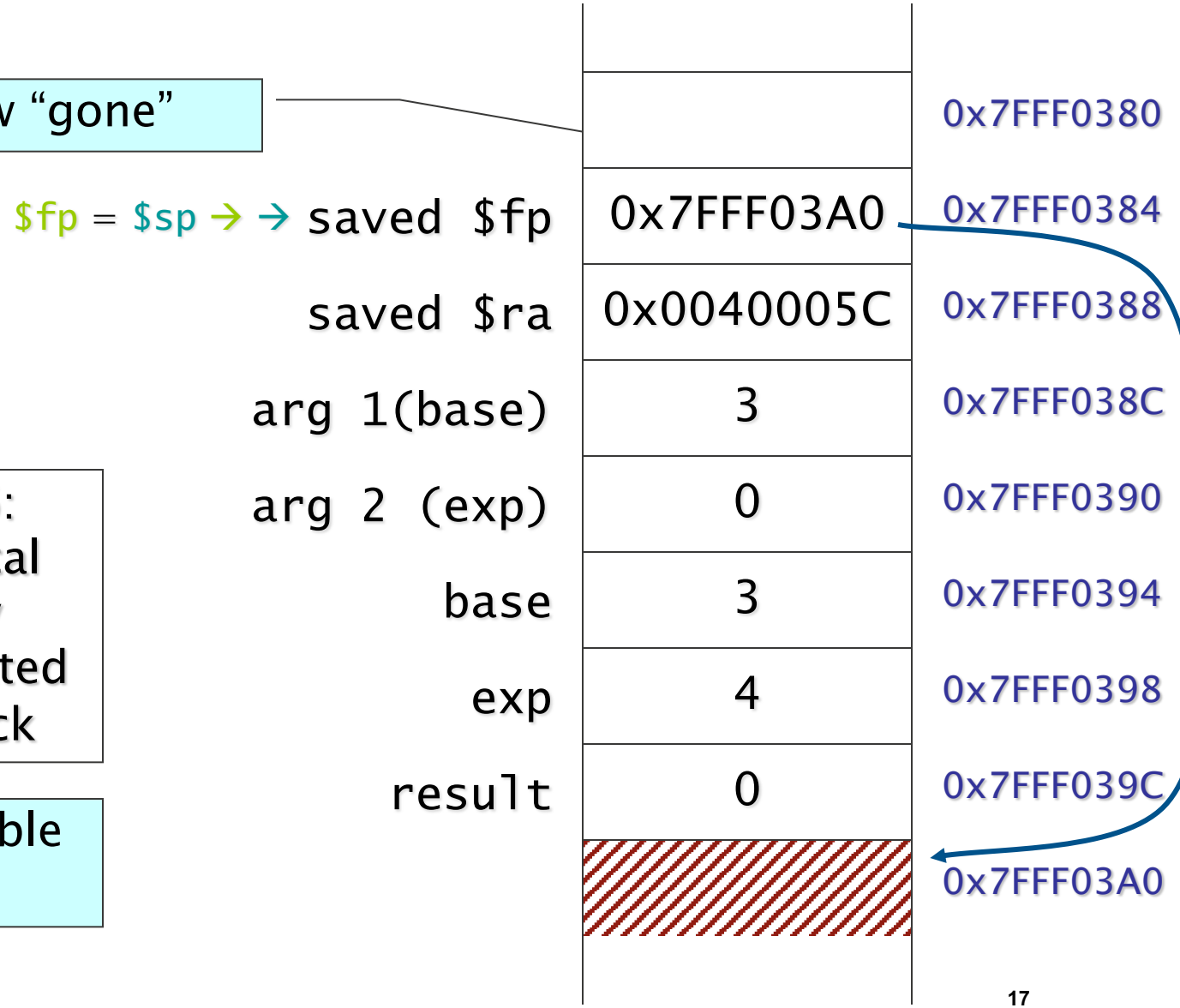


callee step 6: deallocate local variables by popping allocated space off stack

one local variable to delete

Example: callee

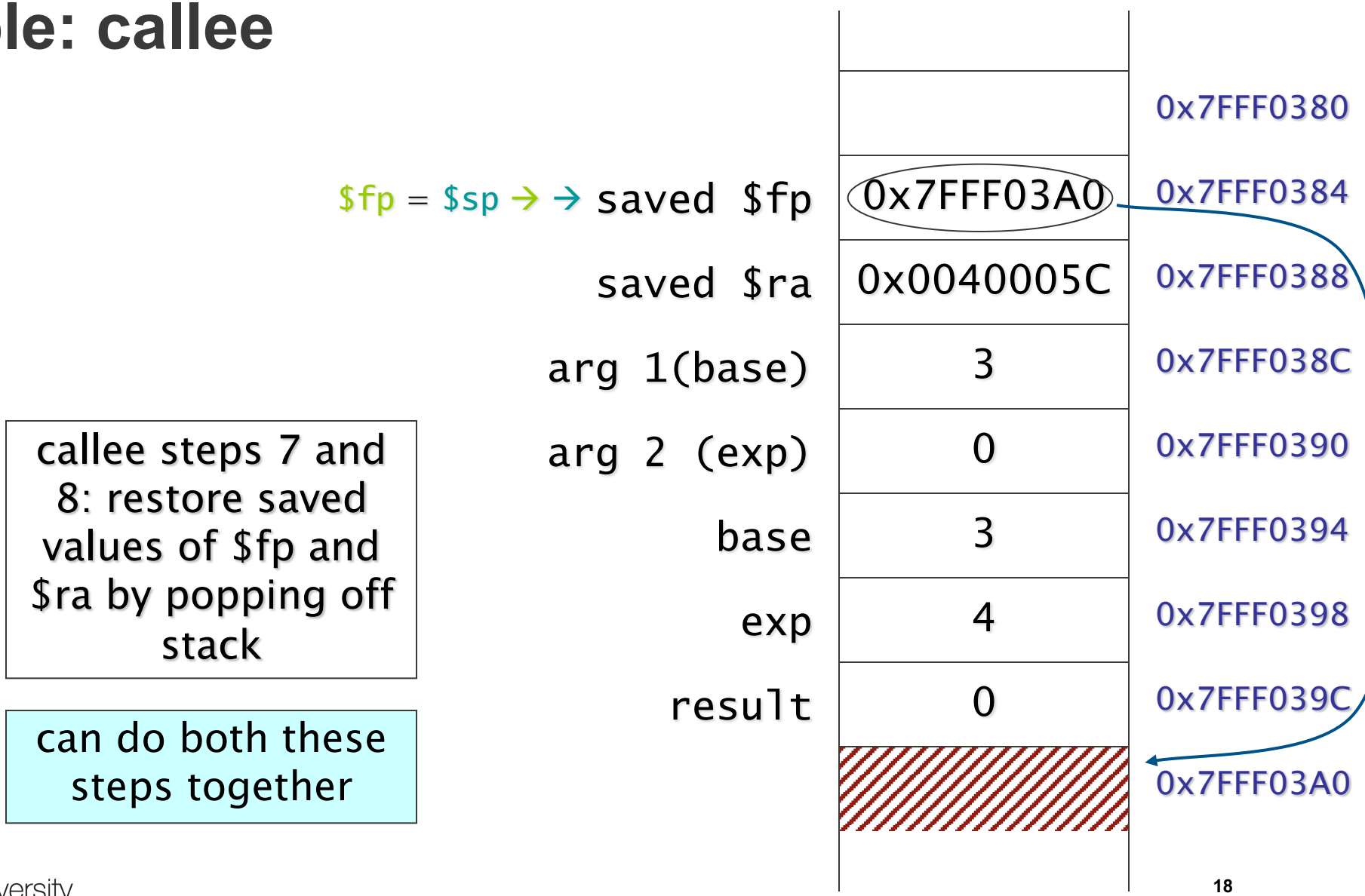
result is now “gone”



callee step 6: deallocate local variables by popping allocated space off stack

one local variable to delete

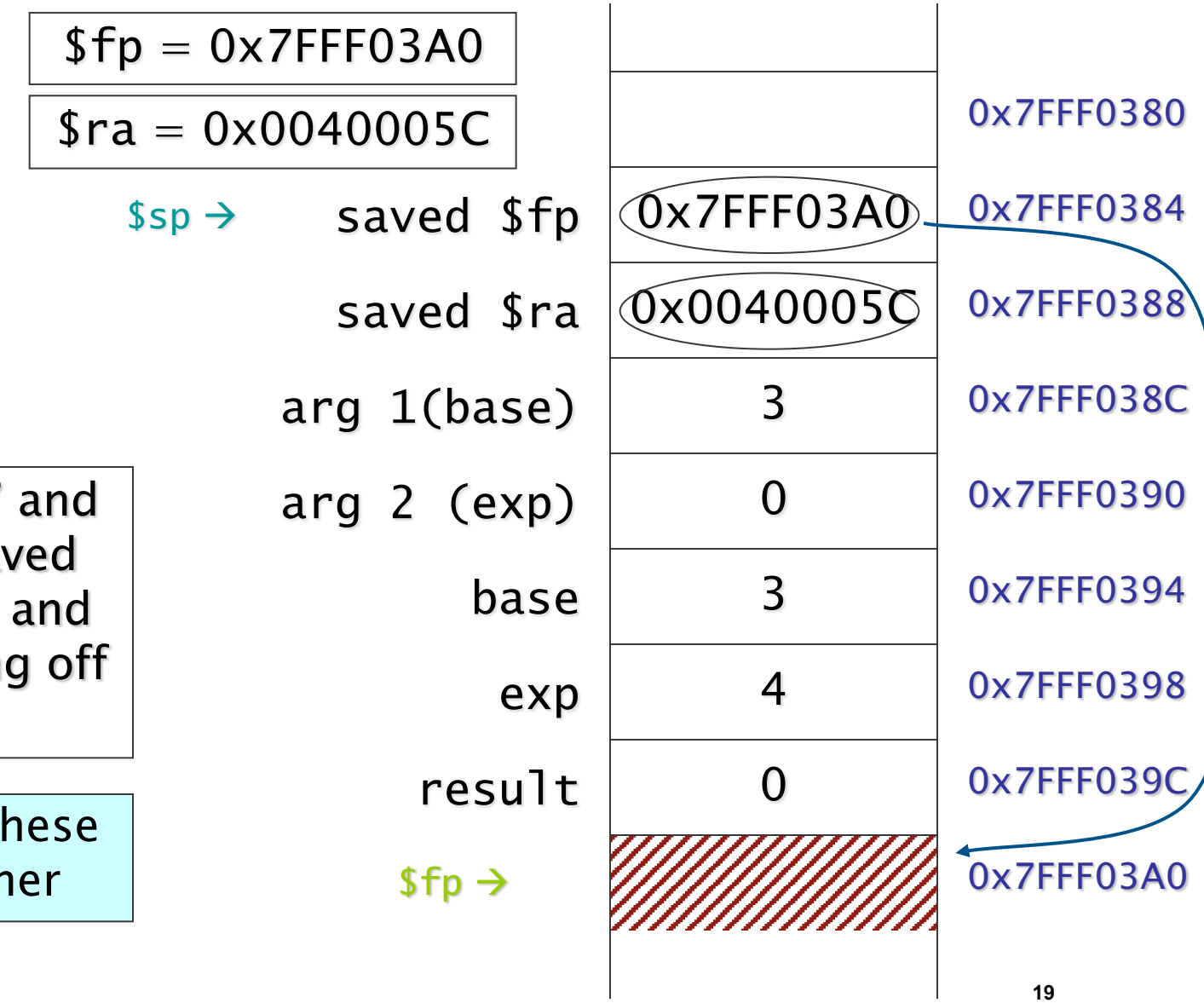
Example: callee



callee steps 7 and 8: restore saved values of $fp and $ra by popping off stack

can do both these steps together

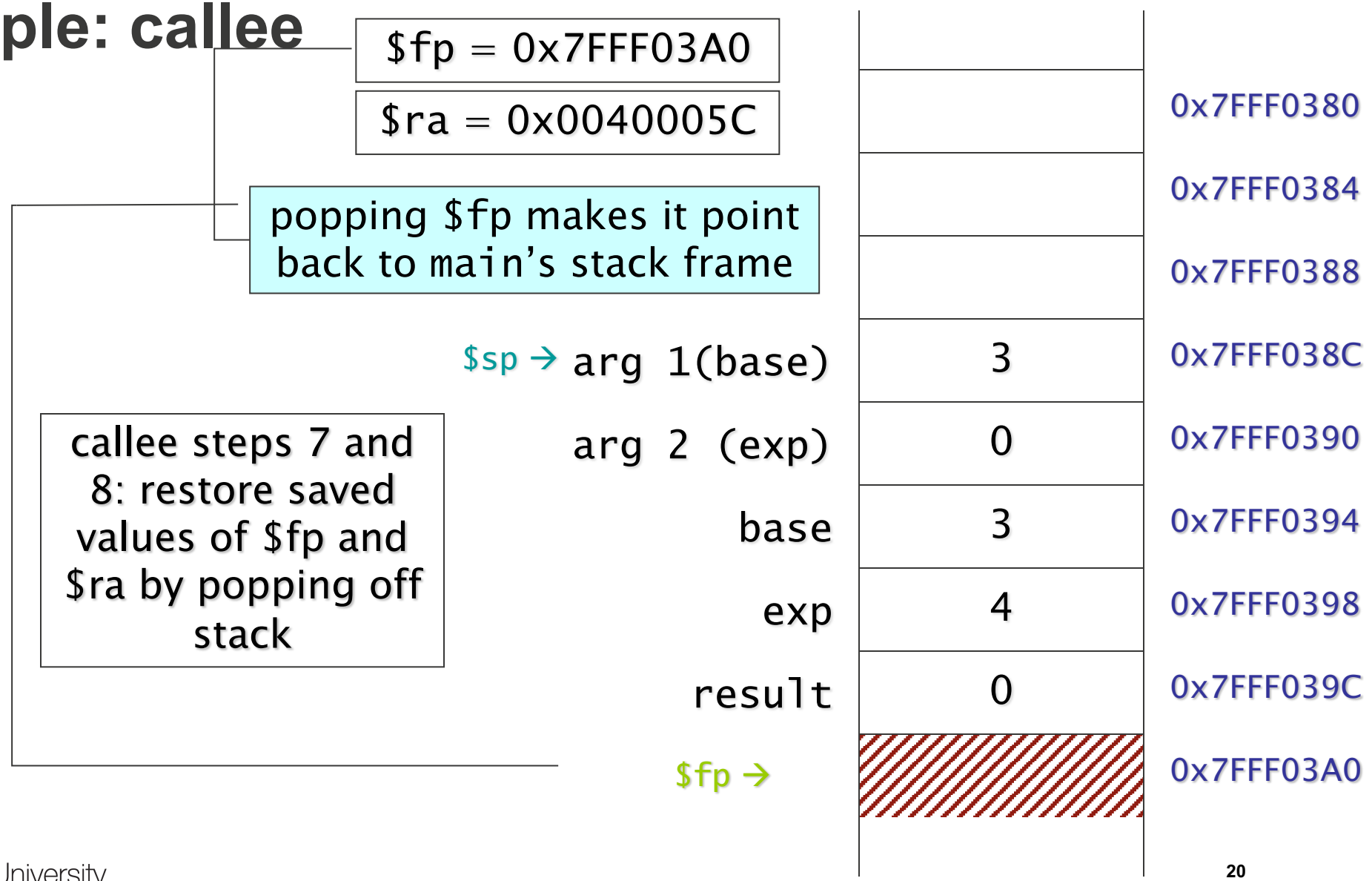
Example: callee



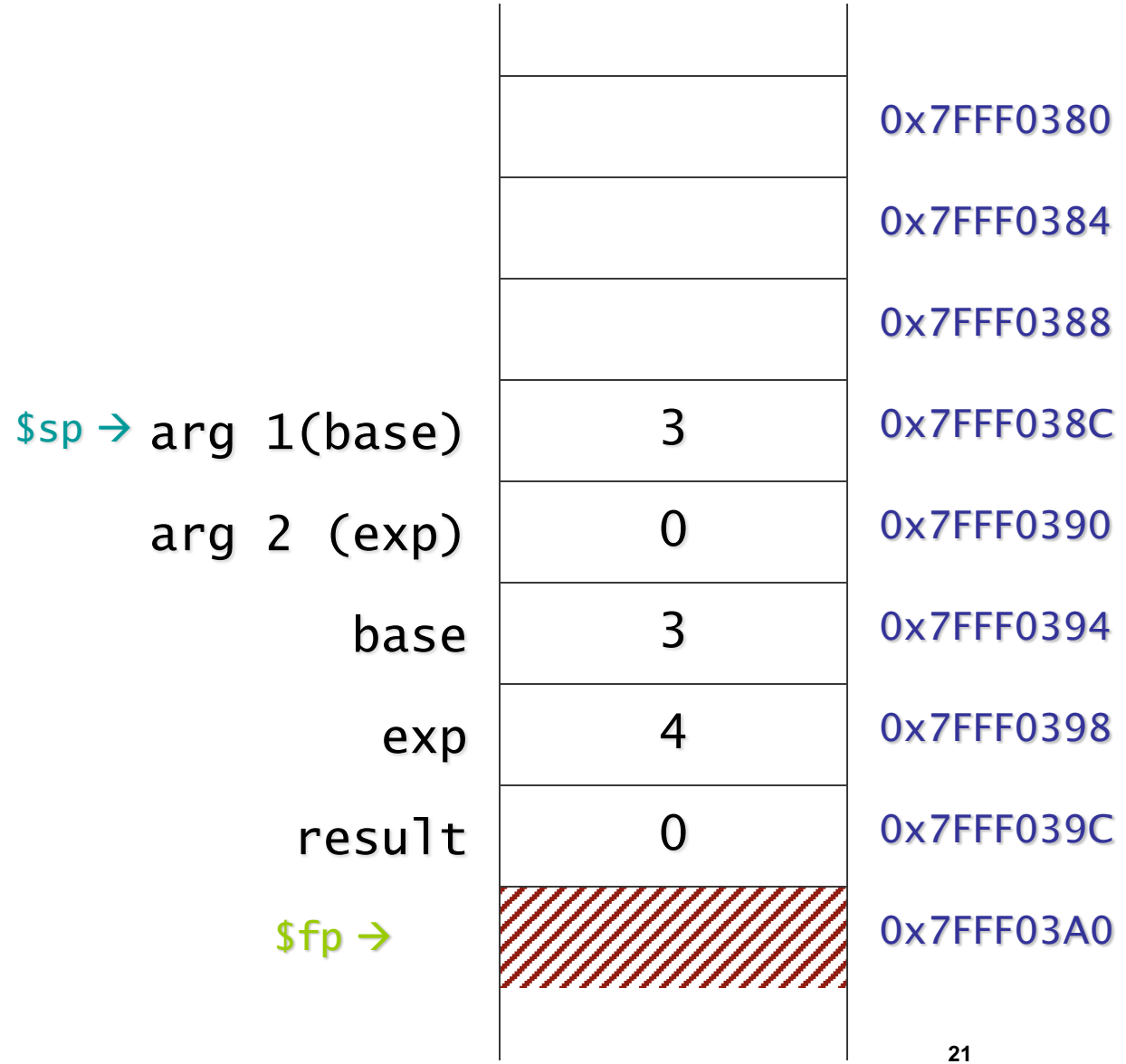
callee steps 7 and 8: restore saved values of $fp and $ra by popping off stack

can do both these steps together

Example: callee



Example: callee



callee step 9: return by executing `jr $ra`

nothing visible on stack

Example: callee

```
result is at -4($fp)
base is at 8($fp)
exp is at 12($fp)
```

```
""" Illustrating also function return. """

def power(base:int, exp:int) -> int:
    """ Computes base to the power of exp. """
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result
```

```
# ... Continued

    # Return result in $v0
    lw $v0, -4($fp) # $v0=result

    # Remove local var.
    addi $sp, $sp, 4

    # Restore $fp and $ra
    lw $fp, 0($sp) #restore $fp
    lw $ra, 4($sp) #restore $ra
    addi $sp, $sp, 8 #deallocate

    # Return to caller.
    jr $ra
```

MONASH University

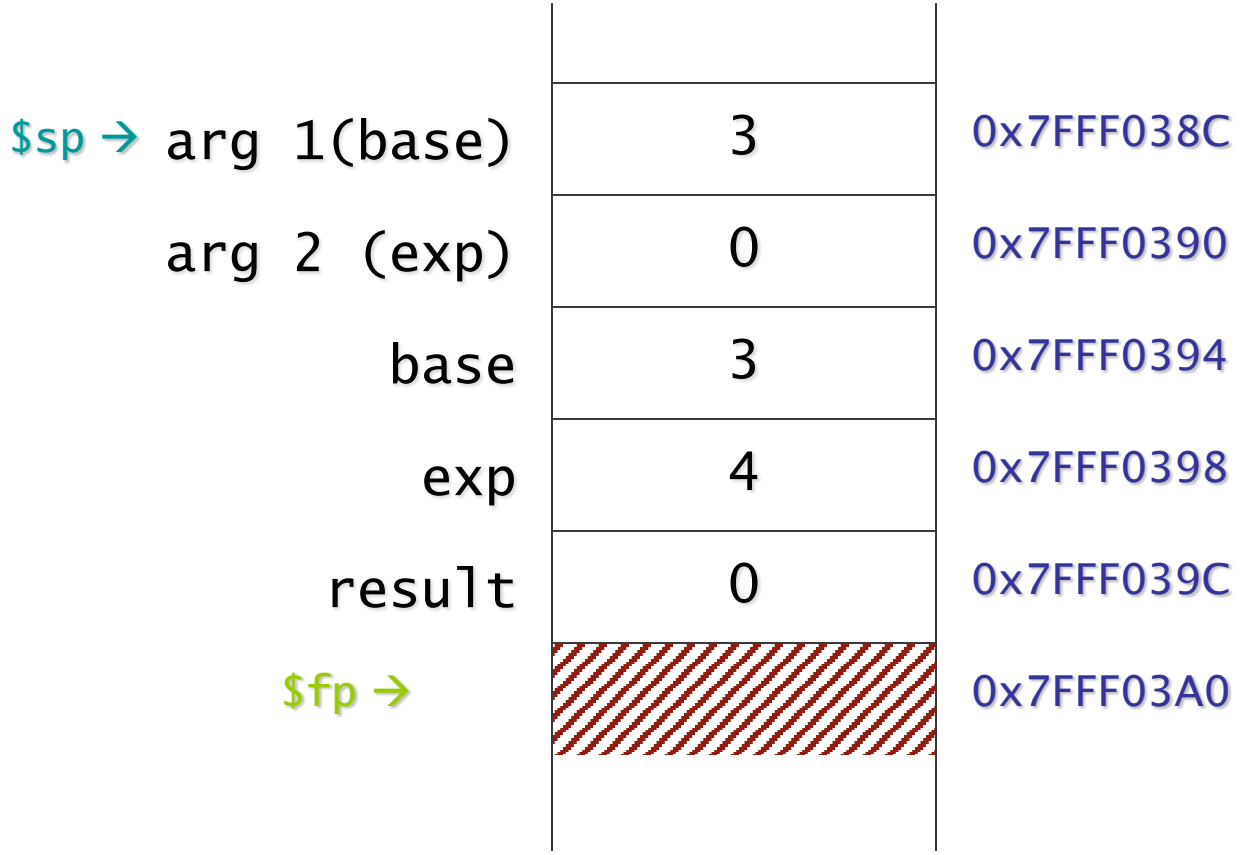
Function Return (caller point of view)

Example: caller

```
""" Illustrating function call. """

def power(base, exp):
    ...
    return result

def main():
    base, exp, result = 0, 0, 0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

now back in caller, about to assign return value of function to `main`'s local variable `result`

Example: caller



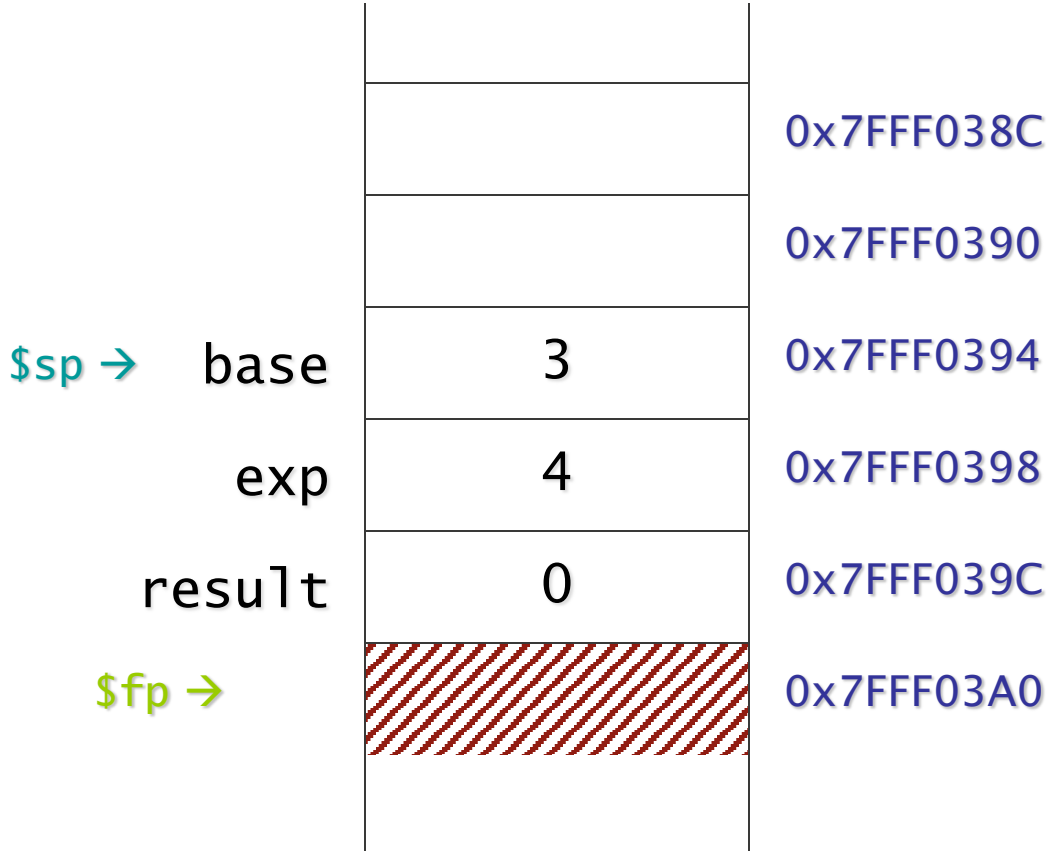
caller step 4: clear function arguments by popping them off stack

base and exp are no longer needed, deallocate them

Example: caller

caller step 4: clear function arguments by popping them off stack

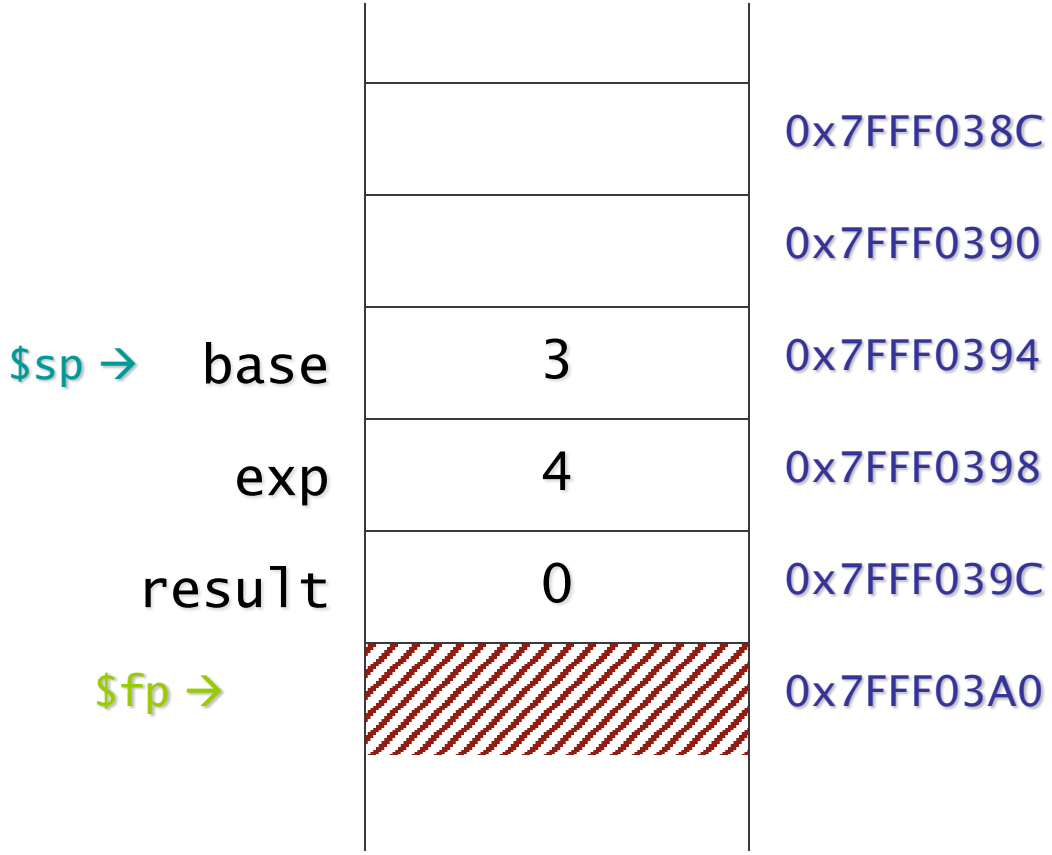
base and exp are no longer needed, deallocate them



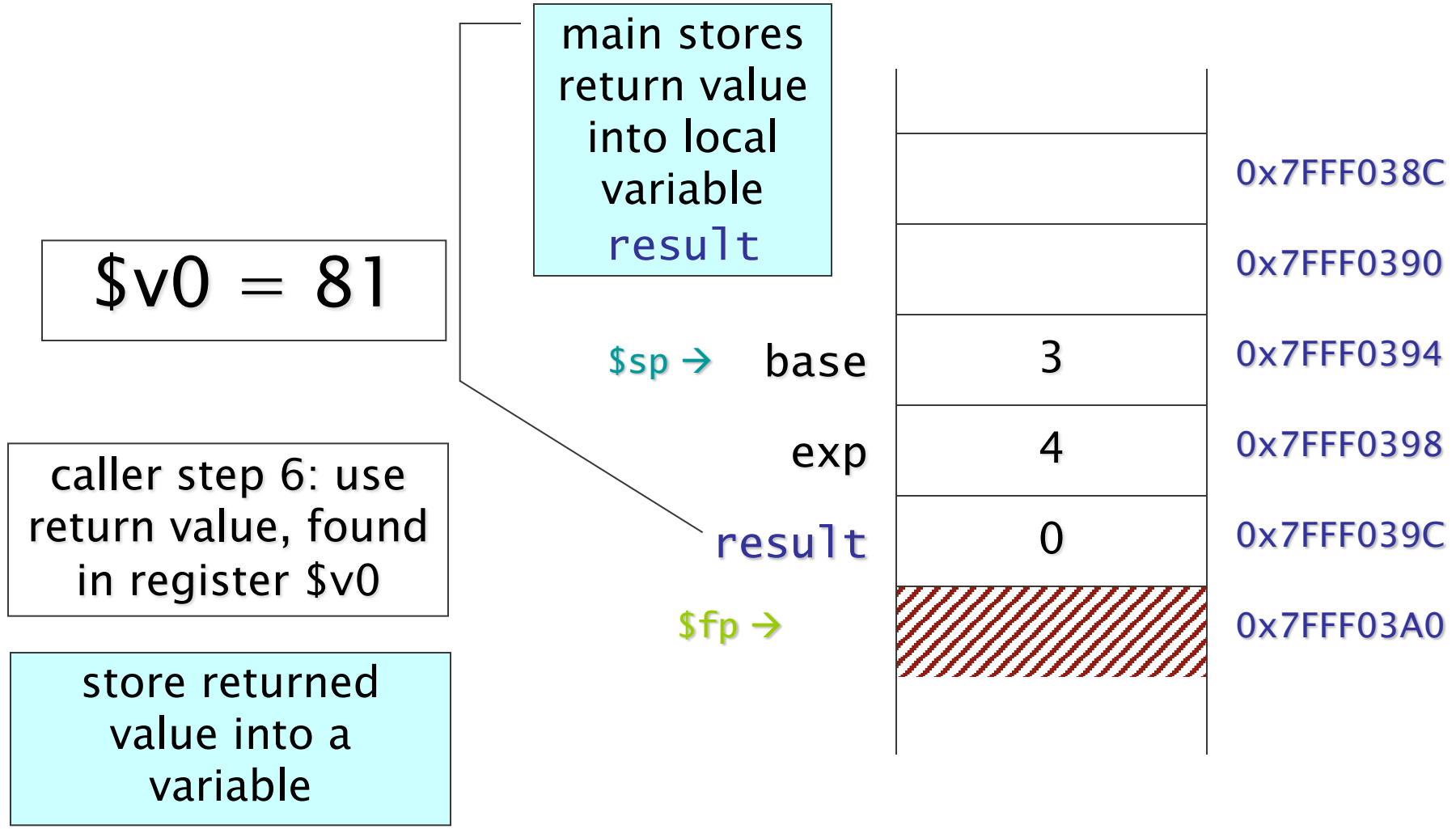
Example: caller

caller step 5: restore saved temporary registers by popping their values off stack

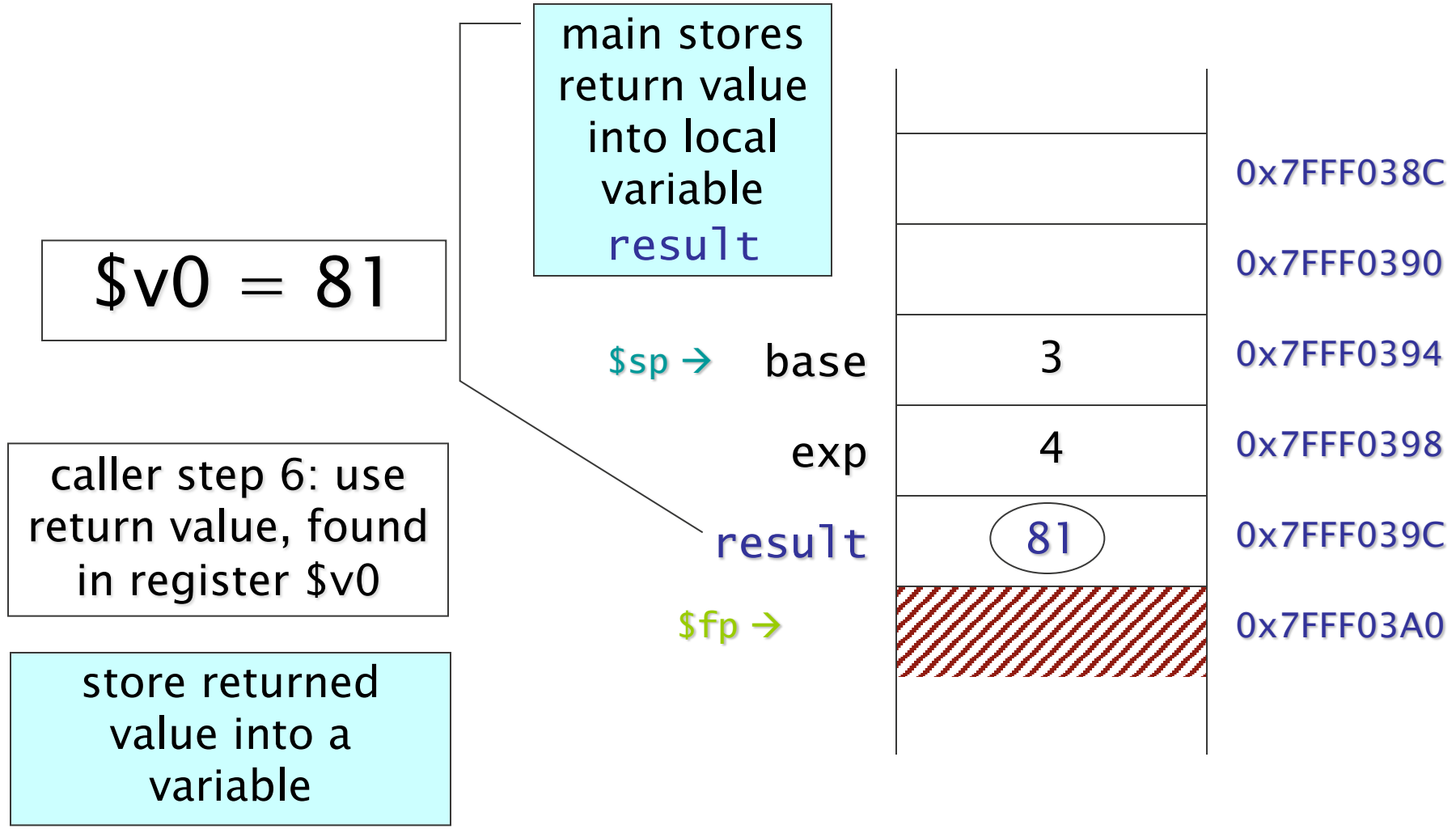
we didn't save any, so nothing to do here



Example: caller



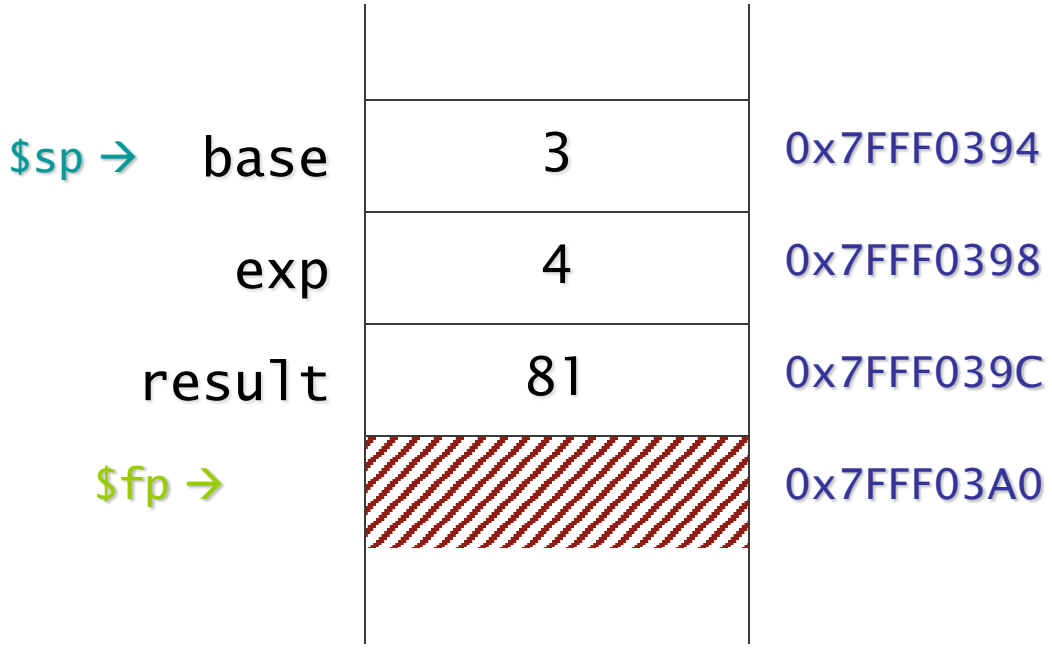
Example: caller



Example: caller

Done!

after the end of the function call the stack has been returned to its original state



Example: caller

```
""" Illustrating function call. """

def power(base, exp):
    ...
    return result


def main():
    base, exp, result = 0, 0, 0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

```
# ... main function, continued
        # Last lecture, we
        # finished with this ...
        # jal power

        # Remove arguments, they
        # are no longer needed.
        # 2 * 4 = 8 bytes.
        addi $sp, $sp, 8

        # Store return value
        # result = power(base,exp)
        sw $v0, -4($fp)

        # Print result
        addi $v0, $0, 1
        lw $a0, -4($fp)
        syscall

        # remove locals, then exit
        addi $sp, $sp, 12
        addi $v0, $0, 10
        syscall
```

Remove locals is not necessary for main

base is at -12($fp)
exp is at -8($fp)
result is at -4($fp)

Function calling convention: a certain symmetry

In summary, caller:

1. saves temporary registers by pushing their values on stack
2. pushes arguments on stack

3. calls the function with `jal` instruction
– (function runs until it returns, then:)
4. clears function arguments by popping allocated space
5. restores saved temporary registers by popping their values off the stack
6. uses the return value found in $v0

Function calling convention: a certain symmetry

In summary, callee:

1. saves `$ra` by pushing its value on stack
2. saves `$fp` by pushing its value on stack
3. copies `$sp` to `$fp`
4. allocates local variables

– (body of function goes here, then:)

5. chooses return value by setting register `$v0`
6. deallocates local variables by popping allocated space
7. restores `$fp` by popping its saved value
8. restores `$ra` by popping its saved value
9. returns with `jr $ra`

None of this needs to be memorised! It is all in the MIPS reference sheet (Moodle Week 1)

Making functions visible from the outside

- **MIPS has the assembler directive:**

  `.globl label`

- **Makes `label` global, so it can be referenced (called) from other MIPS files**
- **Tasks 3 and 4 of interview prac1: you must declare all functions as `.globl`**
  - Including main! (so we can test it)
- **For this to work, the compiler has to "link" the assembled files into a single executable**
- **In MARS, you will need to do as the prac sheet says:**
  - Declare functions as global using `.globl` in the `.data` part
  - In Settings, tick the flag for "Assemble all files in directory"
  - Create one folder called Task3 and another Task4
  - Add to each folder the .asm file for the task
  - Add the test harness for each task
  - Assemble and run as usual

MONASH University

Recursion and the Function Call/Return Convention

Recursion

- **Recursive functions call themselves at some point**
  - Solving a simpler/smaller version of the same problem
- **Function calling convention works exactly the same for recursive functions**
  - Don't need to do anything special!!
- **Each invocation of the function has its own stack frame**
  - Local variables and parameters
    - With their current values
  - Return address
    - Where to return to

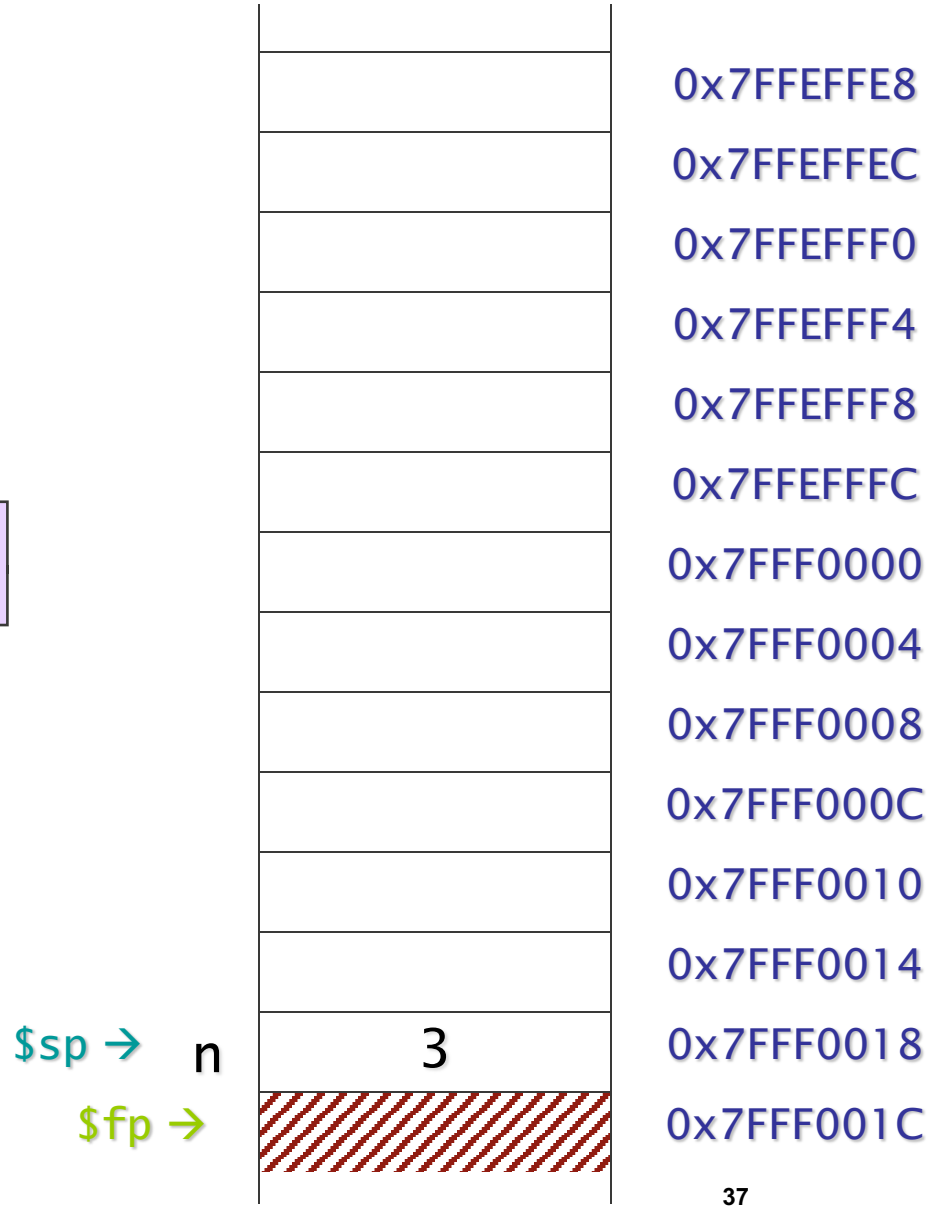
```
def factorial(p:int) -> int:
    """Computes the factorial of p."""
    result = 0
    if p <= 1 :  # Base case
        result = 1
    else:        # Recursive case
        result =
            factorial(p - 1) * p
    return result
```

Parameter: variable in the definition (**p** above)
Argument: the value **p** has when factorial is called
Often treated as synonyms (we will do this too)

Recursion example

```
""" Illustrating recursion. """

def factorial(p:int) -> int:
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

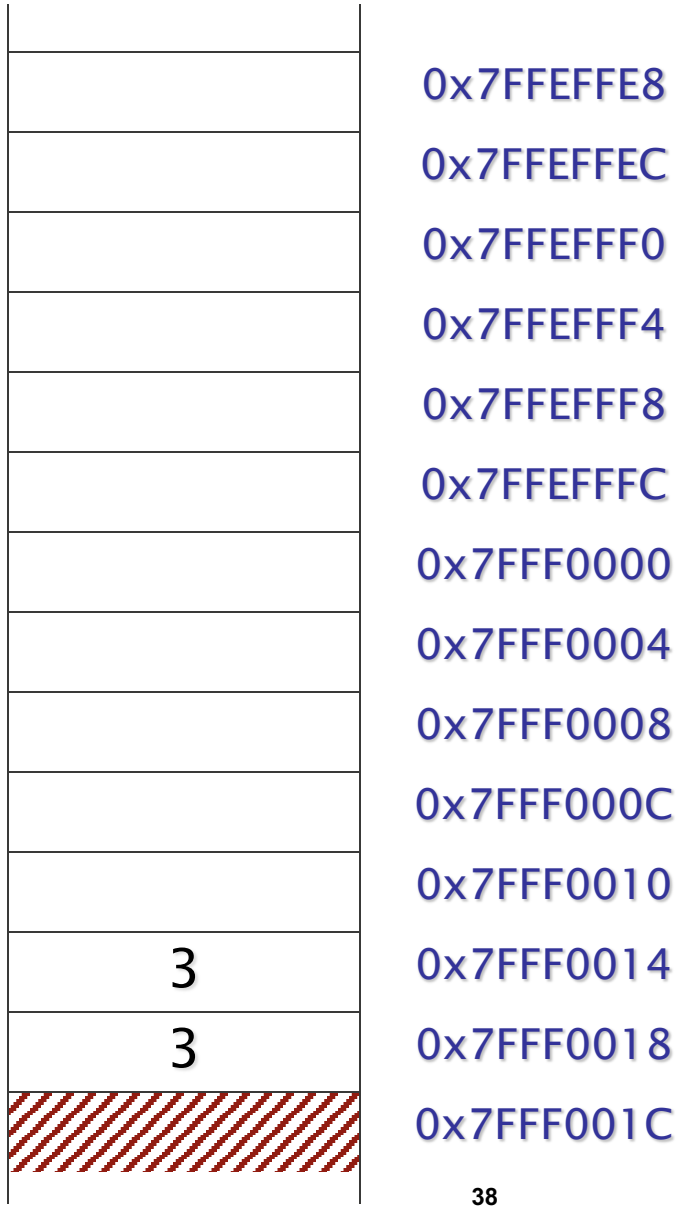
Assume you read 3 into n



Recursion example

```
""" Illustrating recursion. """

def factorial(p:int) -> int:
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

Assume you read 3 into n



From the point of view of main, the argument is at 0($sp)

Recursion example: caller

```
""" Illustrating recursion. """

def factorial(p:int) -> int:
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

n is at -4($fp)

```
    .text
main: # copy $sp to $fp, space locals
addi $fp, $sp, 0  # copy
addi $sp, $sp, -4 # one local
# n = 0
sw   $0, -4($fp)

# read(n)
addi $v0, $0, 5
syscall
sw $v0, -4($fp)

# Call and print factorial
# 1 * 4 = 4 bytes arg.
addi $sp, $sp, -4
lw $t0, -4($fp)  # load n
sw $t0, 0($sp)   # arg 1 = n
jal factorial    # call
# Clear argument.
addi $sp, $sp, 4
# Print factorial(n)
addi $a0, $v0, 0 # factorial(n)
addi $v0, $0, 1
syscall

# remove locals, then exit
addi $sp, $sp, 4
addi $v0, $0, 10
syscall
```

Recursion example: callee

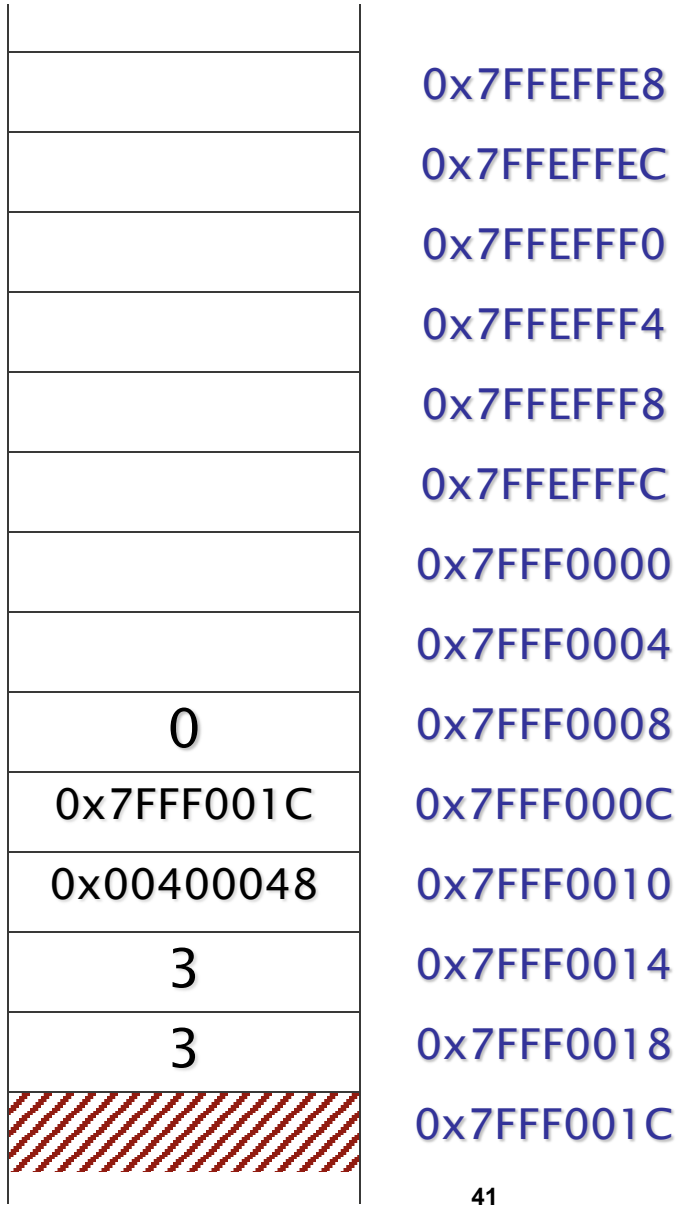
```
def factorial(p:int) -> int:
    """Recursively computes p! """
    result = 0
     if p <= 1 :
          # Base case
         result = 1
     else:
         # Recursive case
         result = factorial(p - 1) * p
     return result
```

Recursion example: callee

```
def factorial(p:int) -> int:
    """Recursively computes p! """
    result = 0
     if p <= 1 :
          # Base case
          result = 1
     else:
          # Recursive case
          result = factorial(p - 1) * p
     return result
```

result is at -4($fp)

p is at 8($fp)



Recursion example: callee

```
def factorial(p:int) -> int:
    """Recursively computes p! """
    result = 0
    if p <= 1 :
        # Base case
        result = 1
    else:
        # Recursive case
        result = factorial(p - 1) * p
    return result
```

```
factorial: # Function entry
        addi $sp, $sp, -8 # alloc space
        sw $ra, 4($sp)    # save $ra
        sw $fp, 0($sp)    # save $fp
        addi $fp, $sp, 0  # copy $sp $fp
        addi $sp, $sp, -4 # space locals

        # result = 0
        sw   $0, -4($fp)

        # if p <= 1 ...
        lw $t0, 8($fp)    # load p
        addi $t1, $0, 1   #
        slt $t0, $t1, $t0 # is 1<p?
        bne $t0, $0, else # if yes, go

        # result = 1
        addi $t0, $0, 1
        sw $t0, -4($fp)
        j endif #jump over else
```

p is at 8($fp)
result is at -4($fp)

Recursion example: callee

```
def factorial(p:int) -> int:
    """Recursively computes p! """
    result = 0
     if p <= 1 :
          # Base case
          result = 1
     else:
          # Recursive case
          result = factorial(p - 1) * p
    return result
```

p is at 8($fp)
result is at -4($fp)

```
# ... Continued
else:    # Recursive call.
         # 1 * 4 = 4 bytes arg
         addi $sp, $sp, -4 #alloc space
         # argument 1 = p-1
         lw $t0, 8($fp)    # load p
         addi $t0, $t0, -1 # p-1
         sw $t0, 0($sp)    # arg 1=p-1
         # call factorial
         jal factorial
         # Deallocate argument
         addi $sp, $sp, 4

         # Multiply factorial(p-1) by p
         lw $t0, 8($fp) # load p
         mult $v0, $t0
         mflo $t0

         # Store it in result
         sw $t0, -4($fp)
```

Recursion example: callee

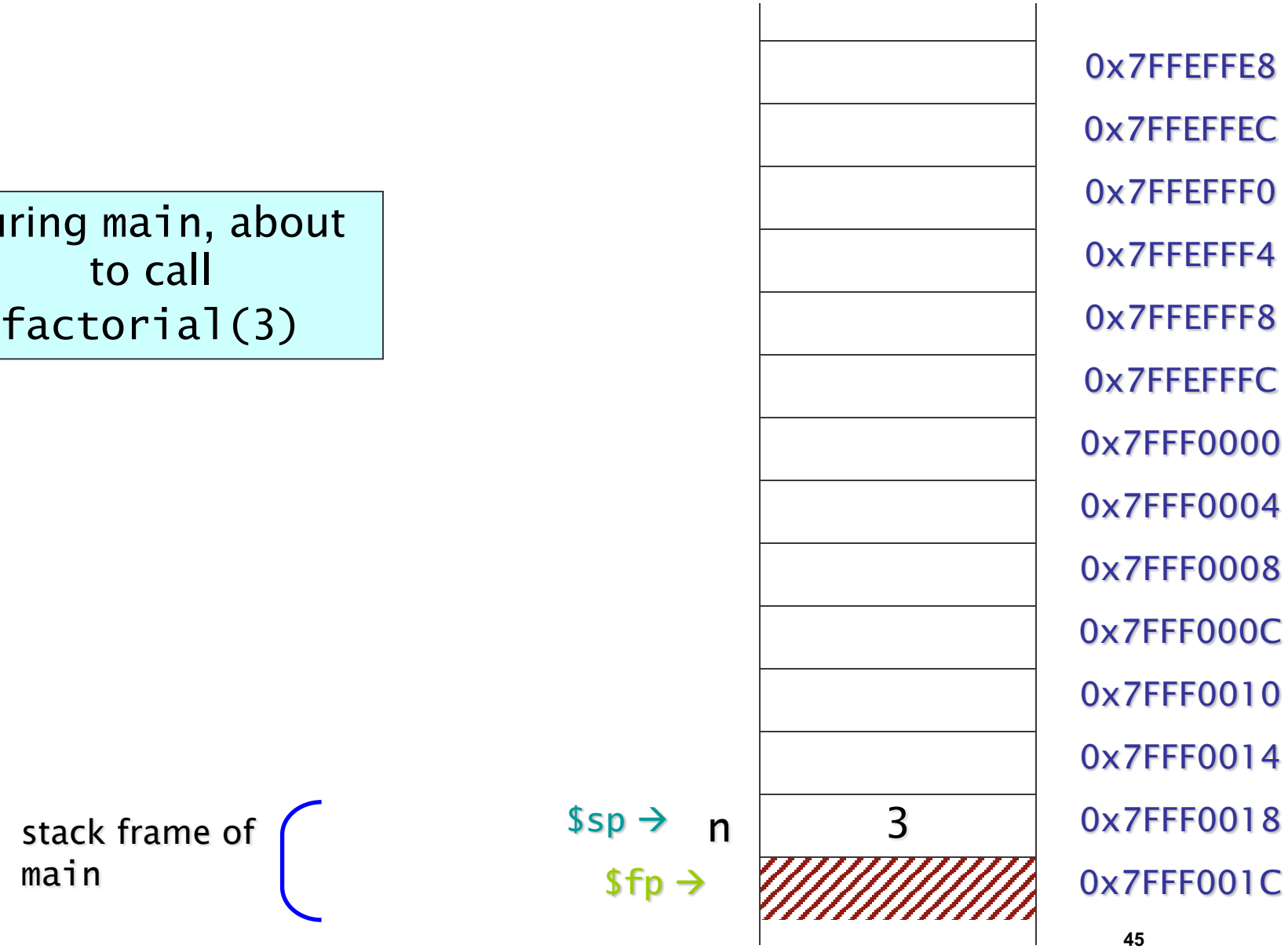
```
def factorial(p:int) -> int:
    """Recursively computes p! """
    result = 0
     if p <= 1 :
          # Base case
          result = 1
     else:
          # Recursive case
          result = factorial(p - 1) * p
     return result
```

```
# ... Continued again

endif:  # return result
        lw $v0, -4($fp)

        # Function return
        # Deallocate local variable
        addi $sp, $sp, 4
        # Restore $fp and $ra
        lw $fp, 0($sp)   # restore $fp
        lw $ra, 4($sp)   # restore $ra
        addi $sp, $sp, 8 # dealloc
        # return
        jr $ra
```

p is at 8($fp)
result is at -4($fp)

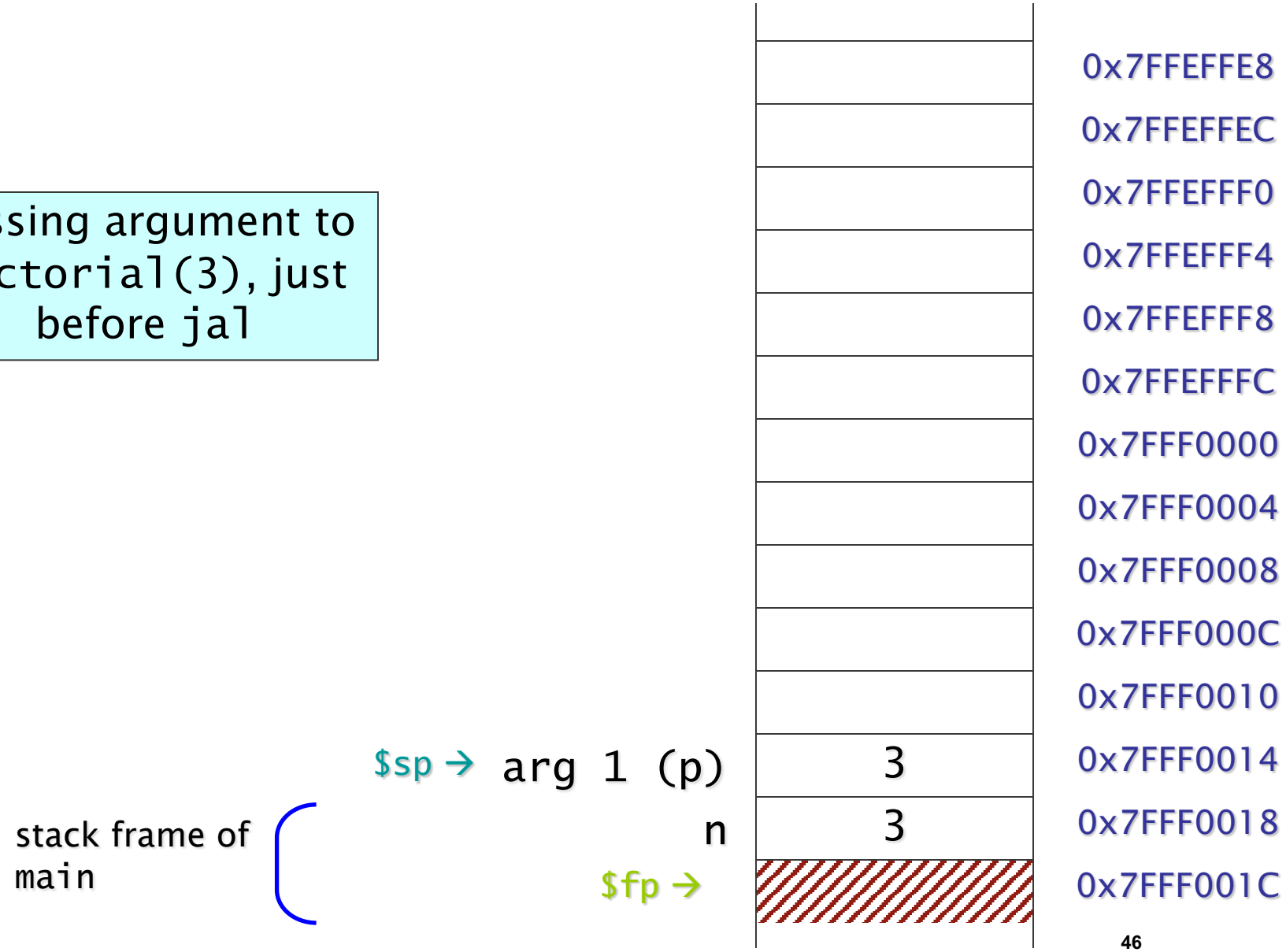
Recursion

during `main`, about to call `factorial(3)`



Recursion

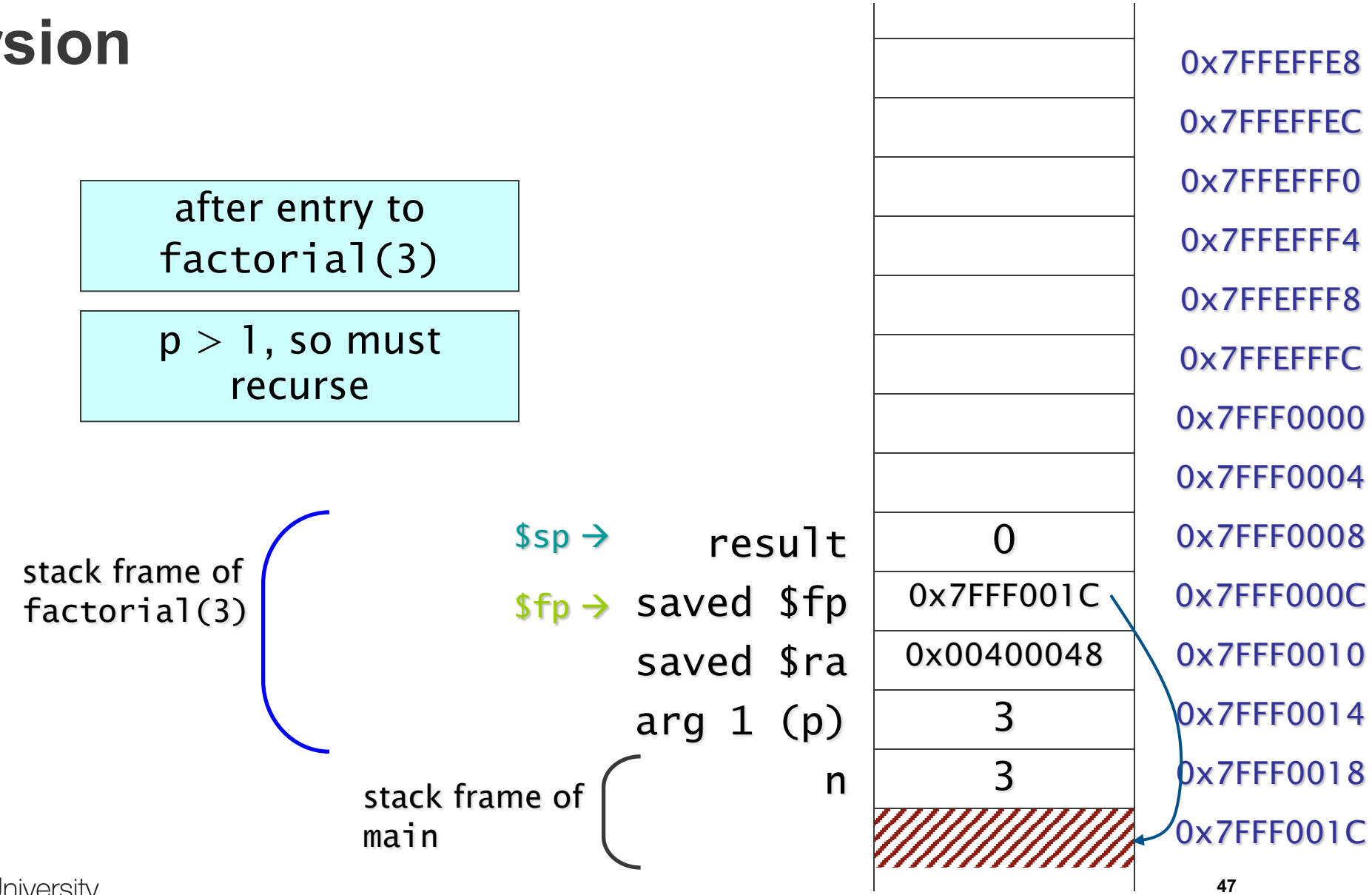
passing argument to `factorial(3)`, just before `jal`



Recursion

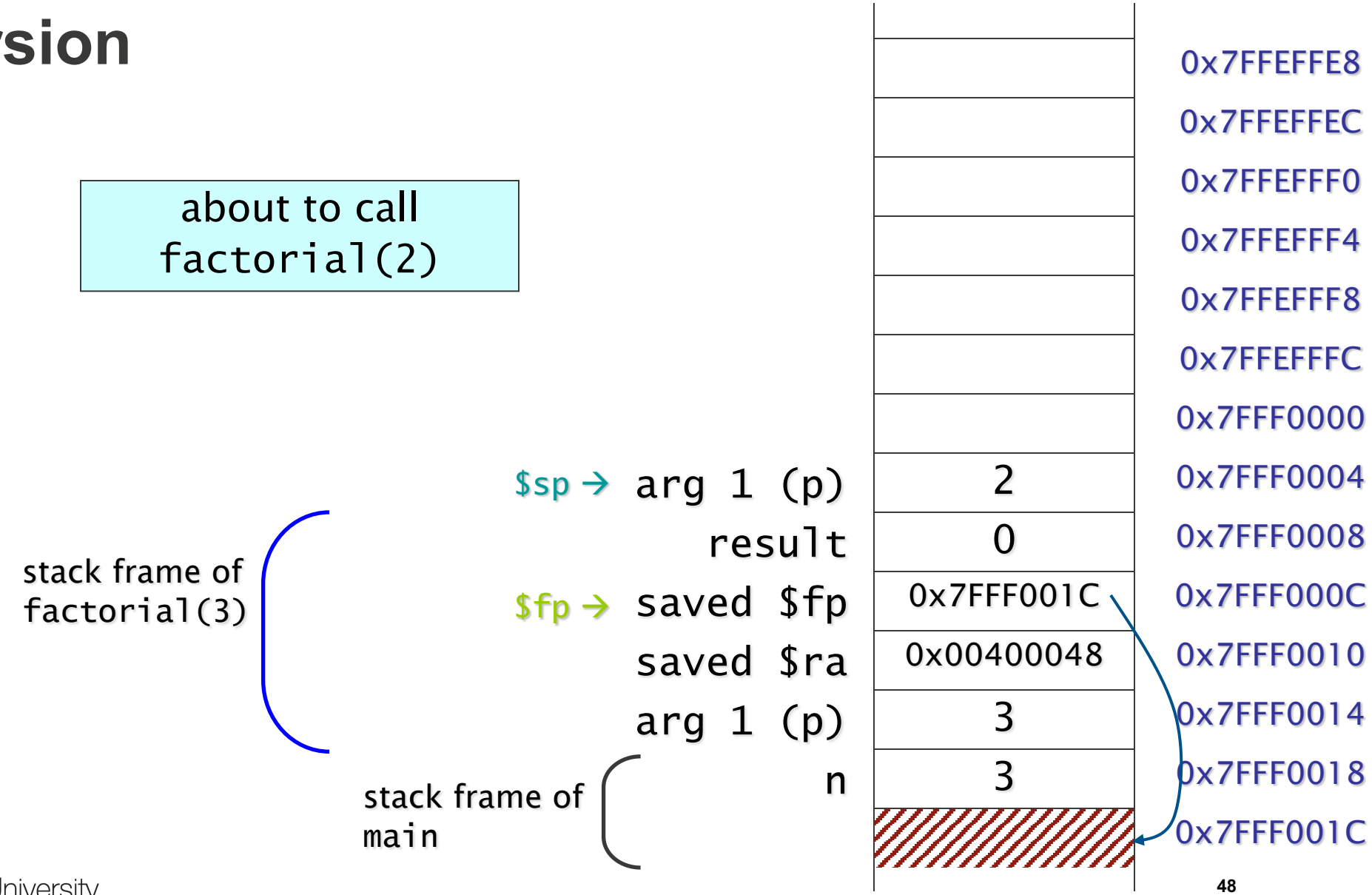
after entry to `factorial(3)`

`p > 1`, so must recurse



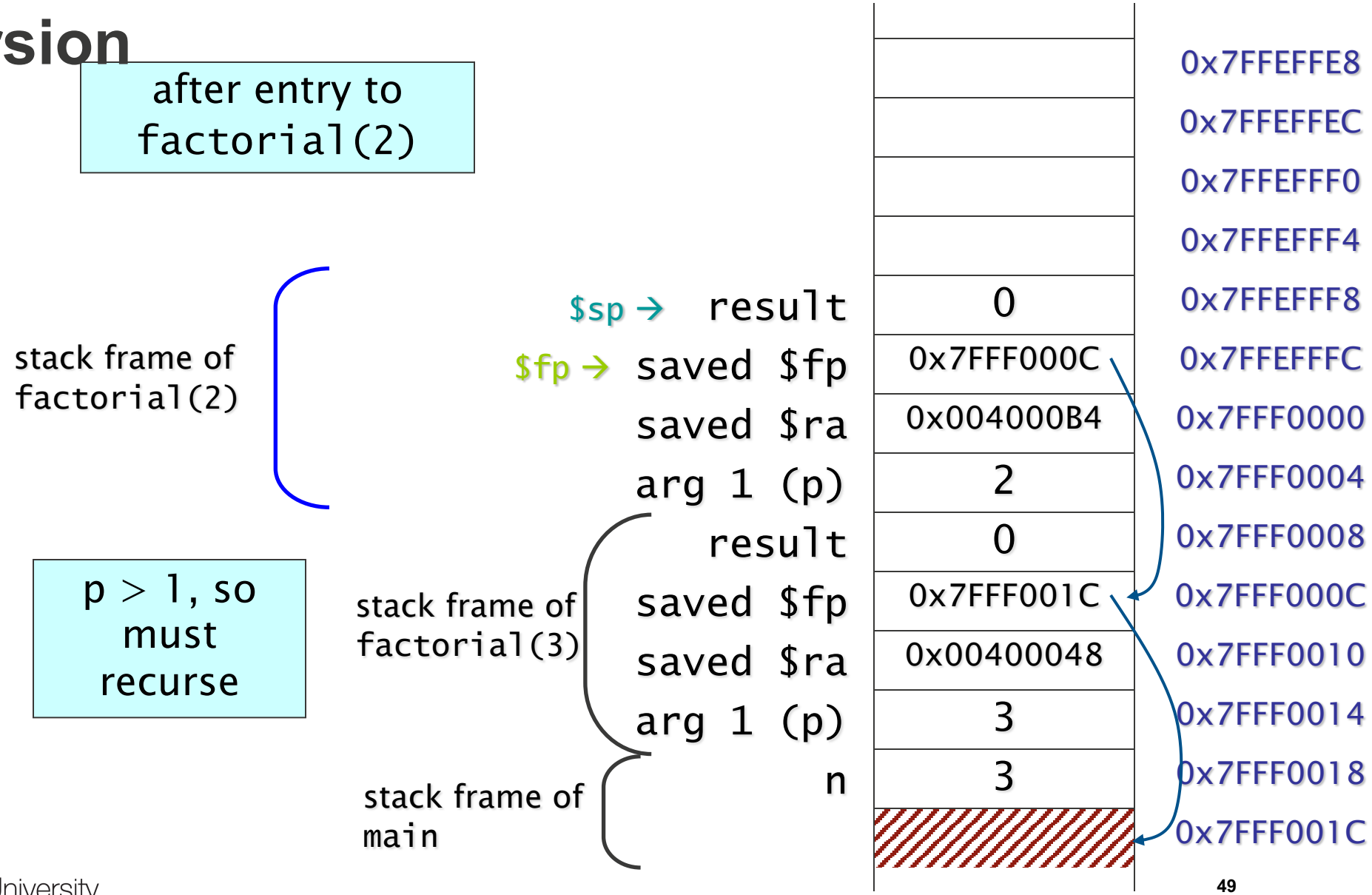
Recursion

about to call factorial(2)



Recursion

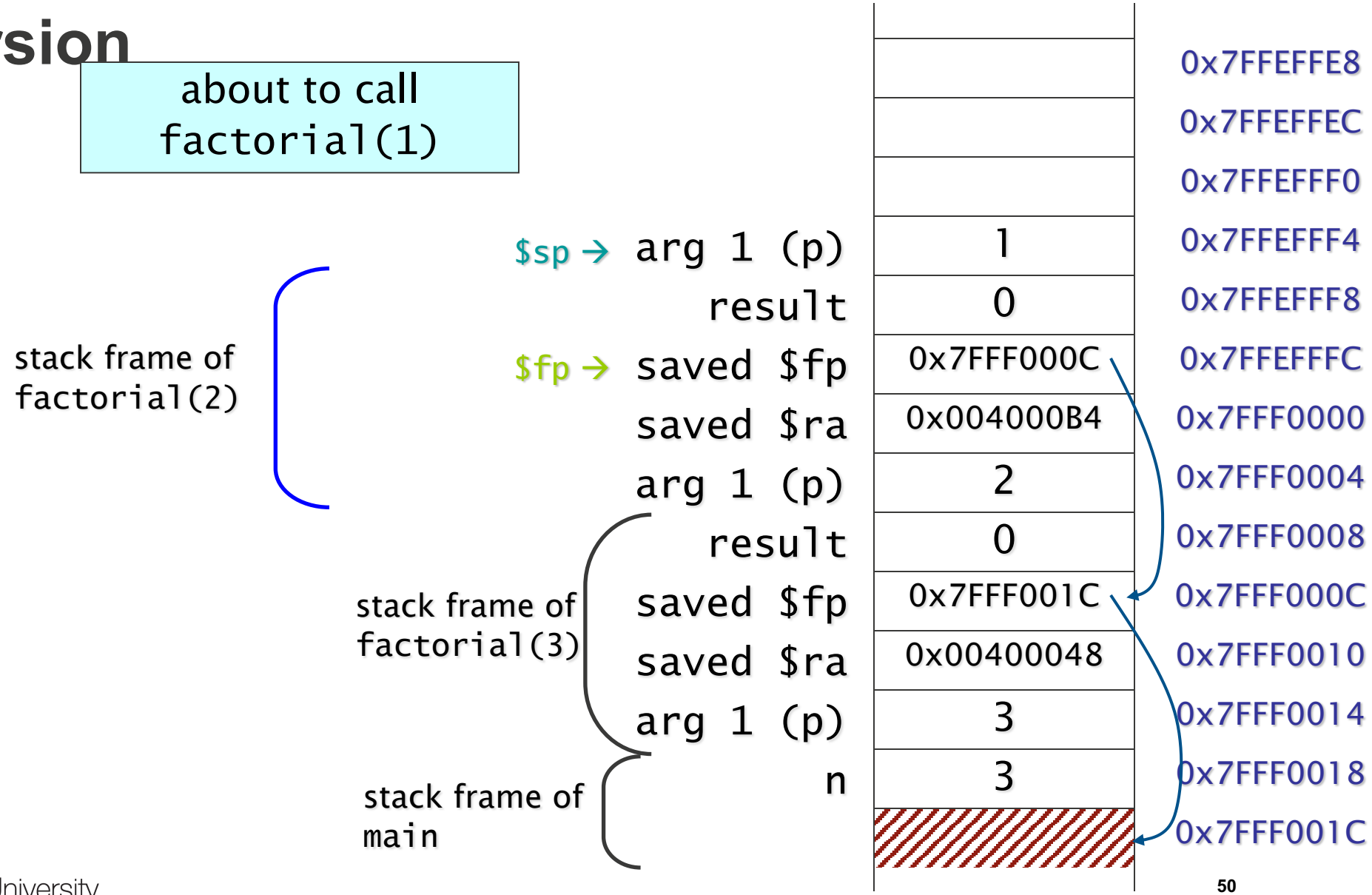
after entry to factorial(2)



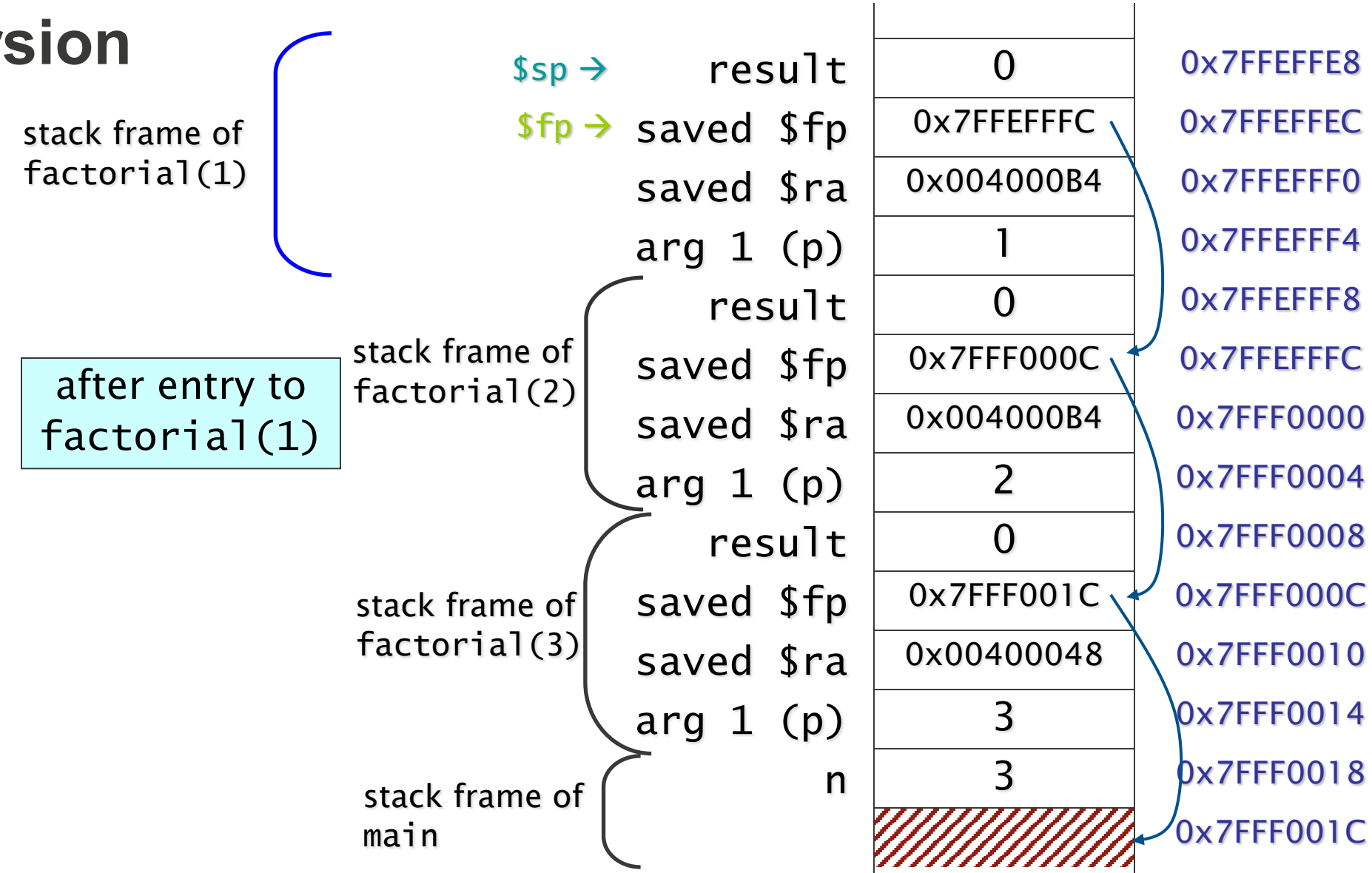
p > 1, so must recurse

Recursion

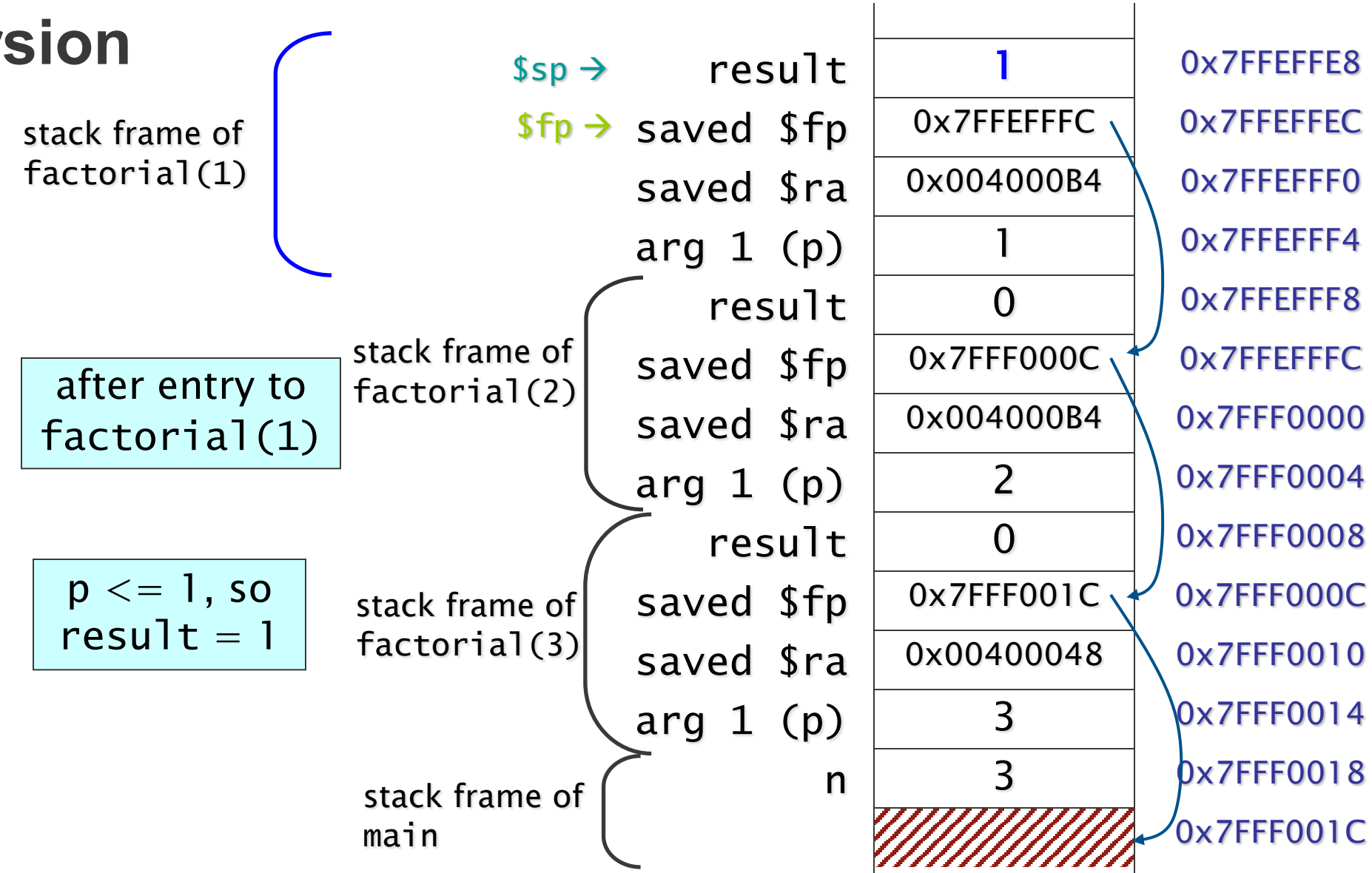
about to call factorial(1)



Recursion

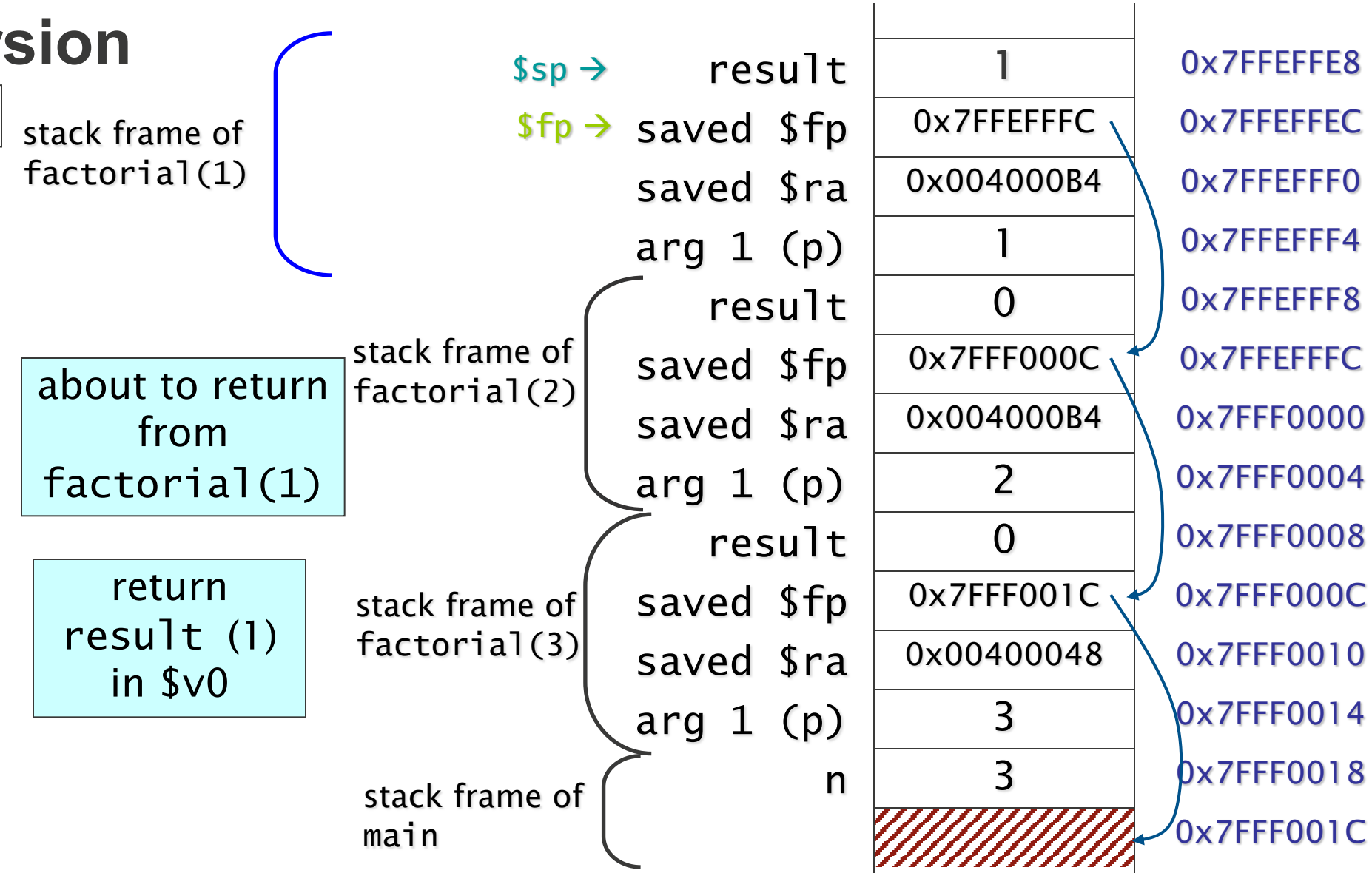

stack frame of factorial(1)
after entry to factorial(1)
stack frame of factorial(2)
stack frame of factorial(3)
stack frame of main
$sp →
$fp →
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
n
0
0x7FFEFFFC
0x004000B4
1
0
0x7FFF000C
0x004000B4
2
0
0x7FFF001C
0x00400048
3
3
0x7FFEFFE8
0x7FFEFFEC
0x7FFEFFF0
0x7FFEFFF4
0x7FFEFFF8
0x7FFEFFFC
0x7FFF0000
0x7FFF0004
0x7FFF0008
0x7FFF000C
0x7FFF0010
0x7FFF0014
0x7FFF0018
0x7FFF001C

Recursion


stack frame of factorial(1)
after entry to factorial(1)
p <= 1, so result = 1
stack frame of factorial(2)
stack frame of factorial(3)
stack frame of main
$sp →
$fp →
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
n
1
0x7FFEFFFC
0x004000B4
1
0
0x7FFF000C
0x004000B4
2
0
0x7FFF001C
0x00400048
3
3
0x7FFEFFE8
0x7FFEFFEC
0x7FFEFFF0
0x7FFEFFF4
0x7FFEFFF8
0x7FFEFFFC
0x7FFF0000
0x7FFF0004
0x7FFF0008
0x7FFF000C
0x7FFF0010
0x7FFF0014
0x7FFF0018
0x7FFF001C

Recursion

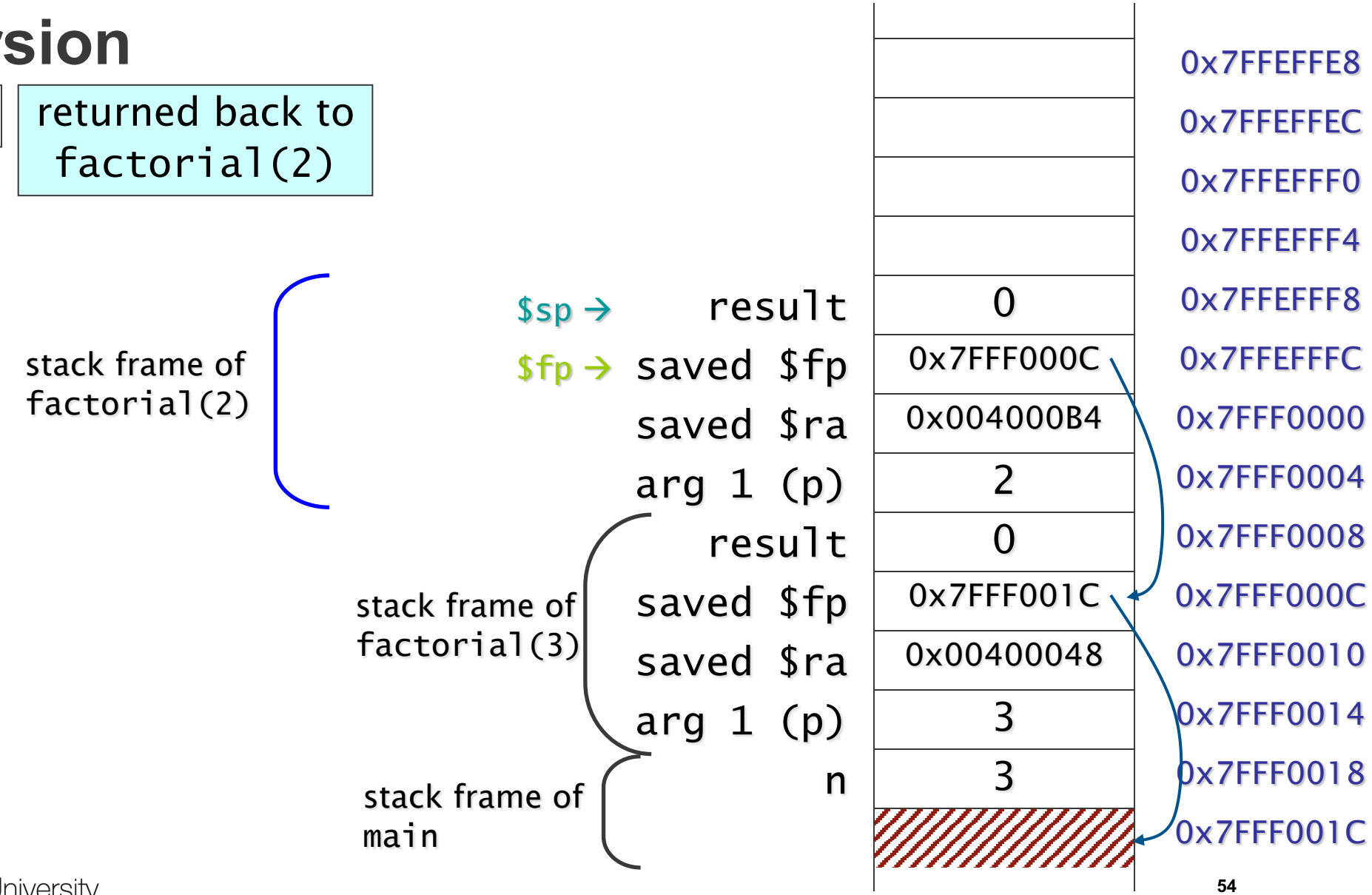
$v0 = 1



Recursion

$v0 = 1

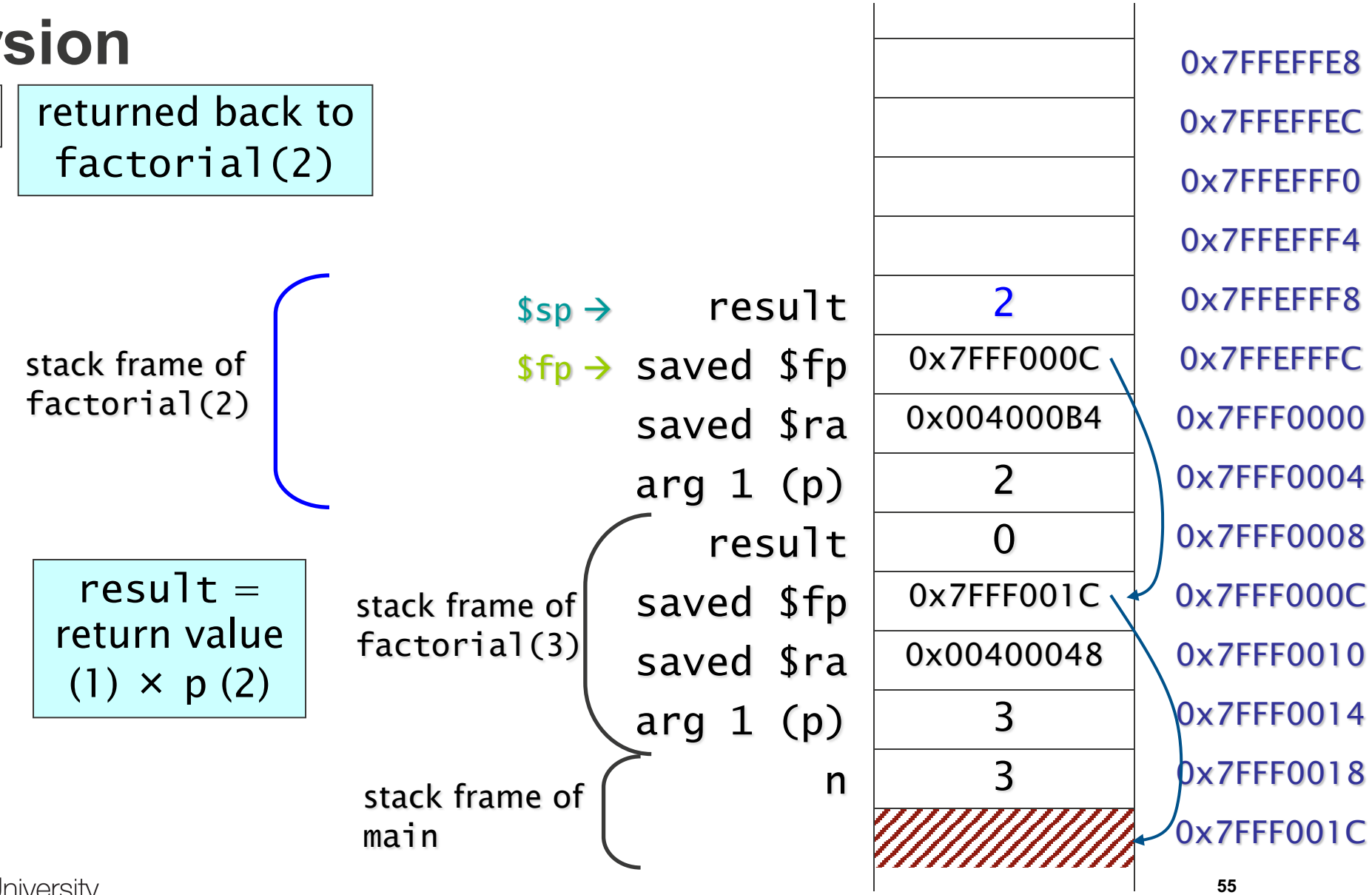
returned back to factorial(2)



Recursion

$v0 = 1

returned back to factorial(2)

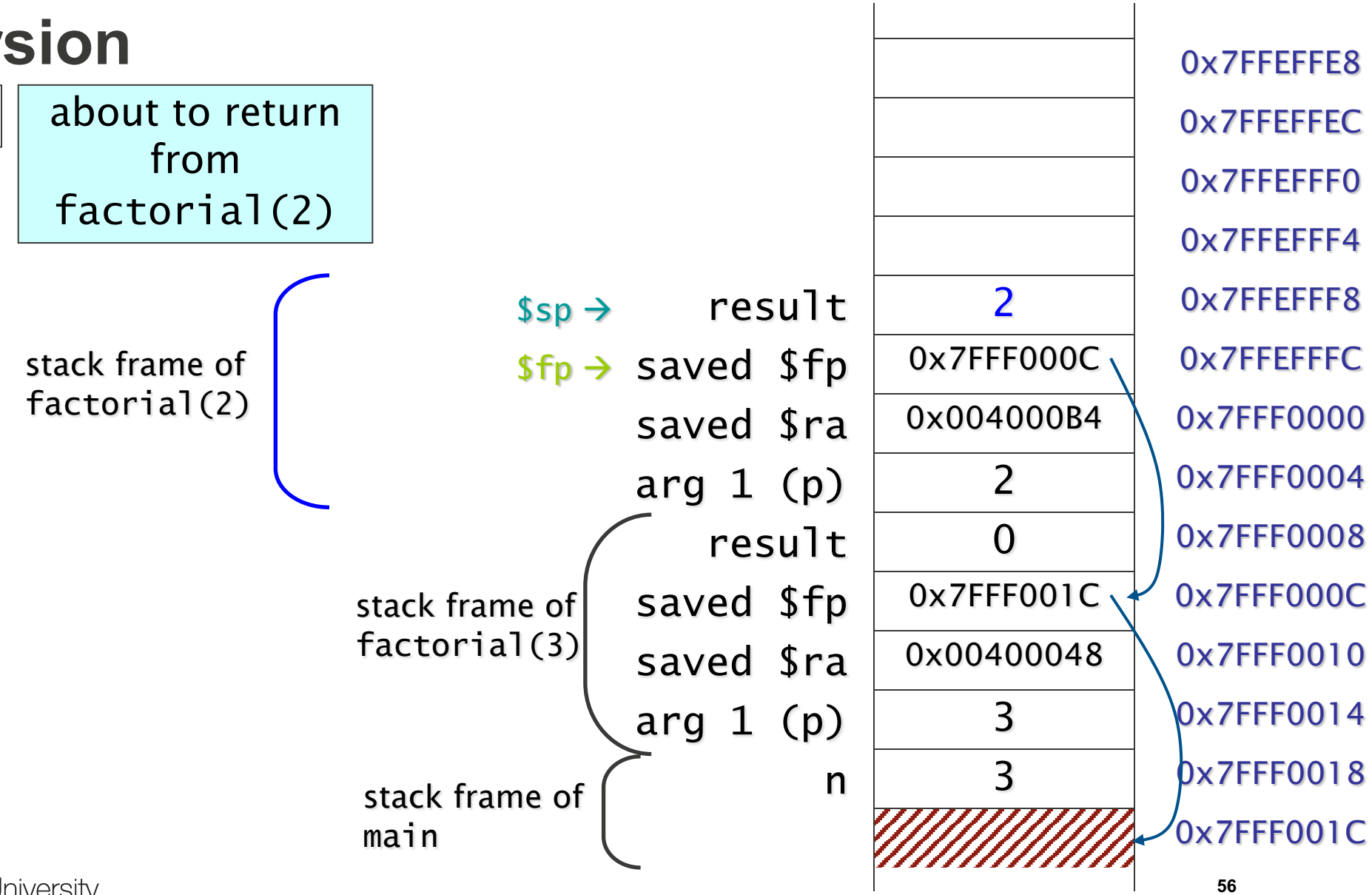


result = return value (1) × p (2)

Recursion

$v0 = 1

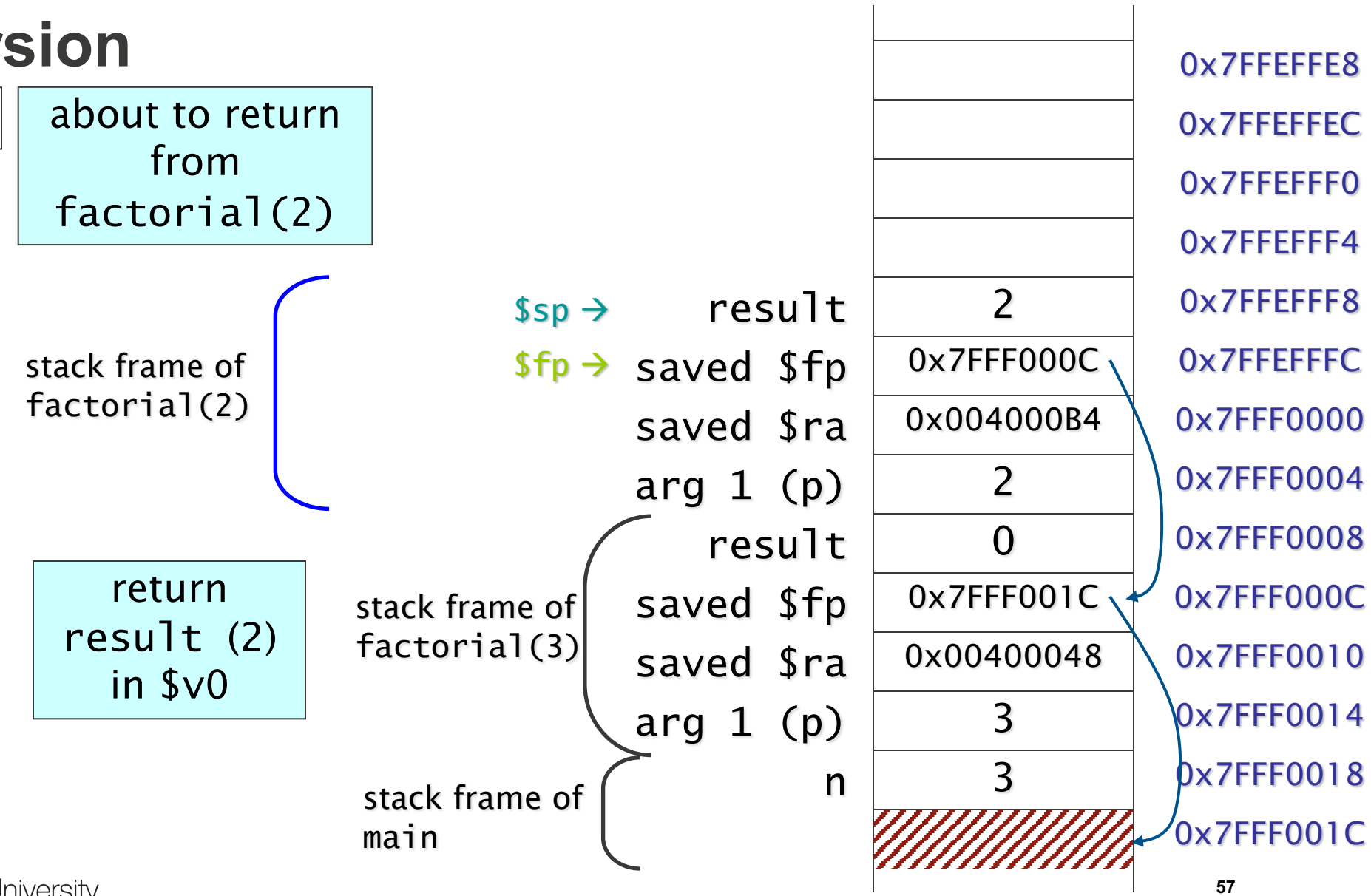
about to return from factorial(2)



Recursion

$v0 = 2

about to return from factorial(2)

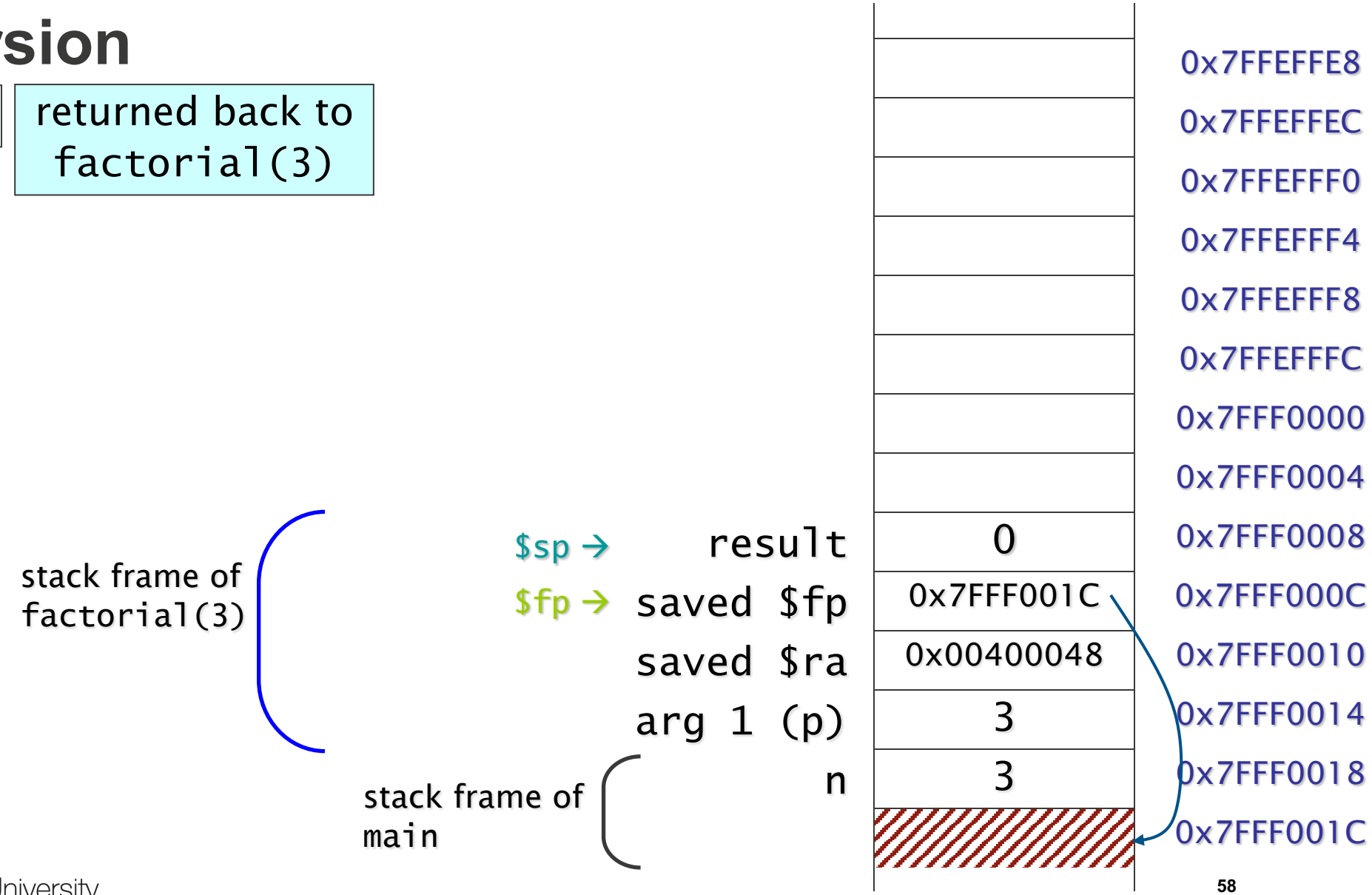


return result (2) in $v0

Recursion

$v0 = 2

returned back to factorial(3)

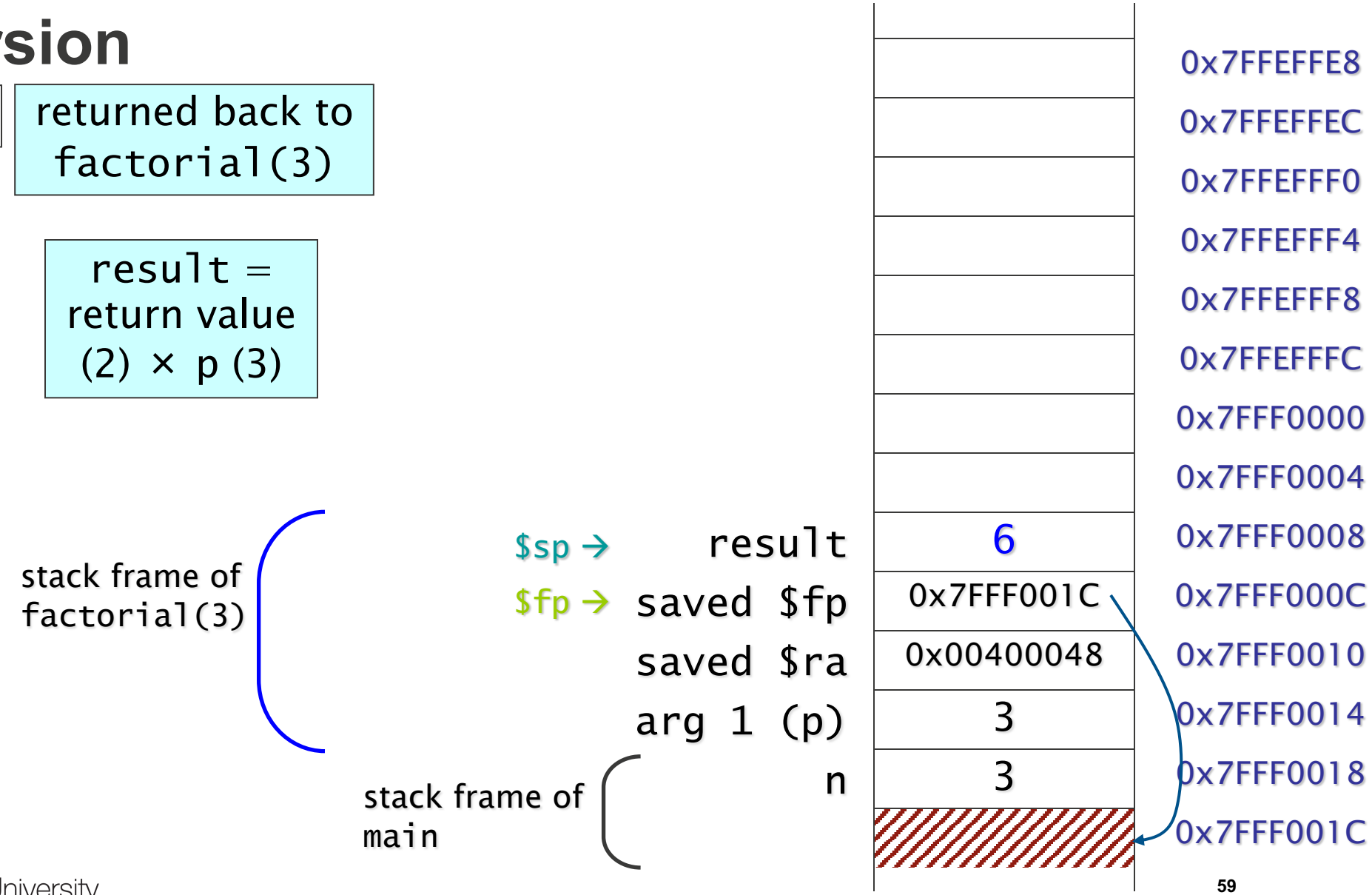


Recursion

$v0 = 2

returned back to factorial(3)

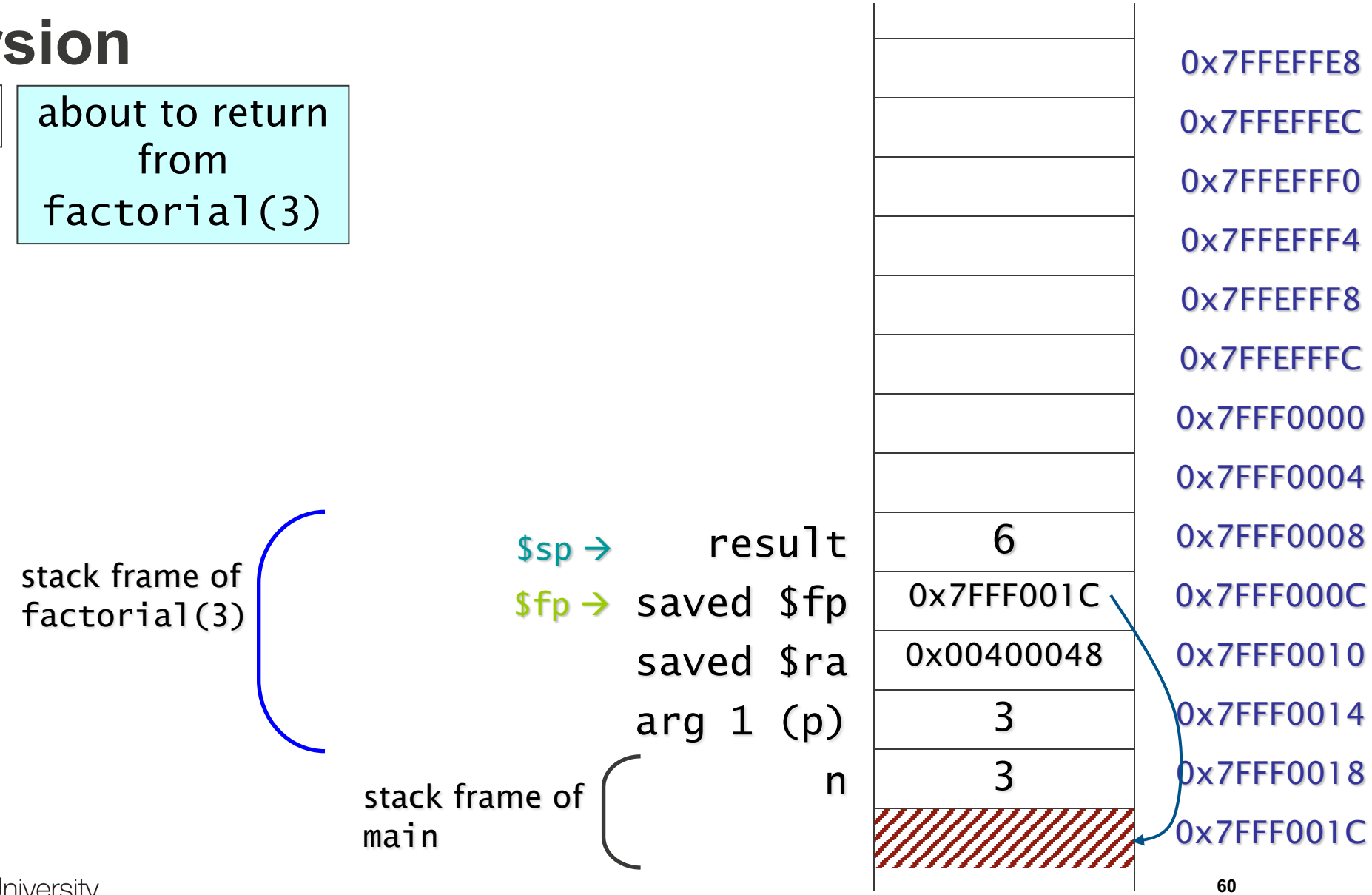
result = return value (2) × p (3)



Recursion

$v0 = 2

about to return from factorial(3)

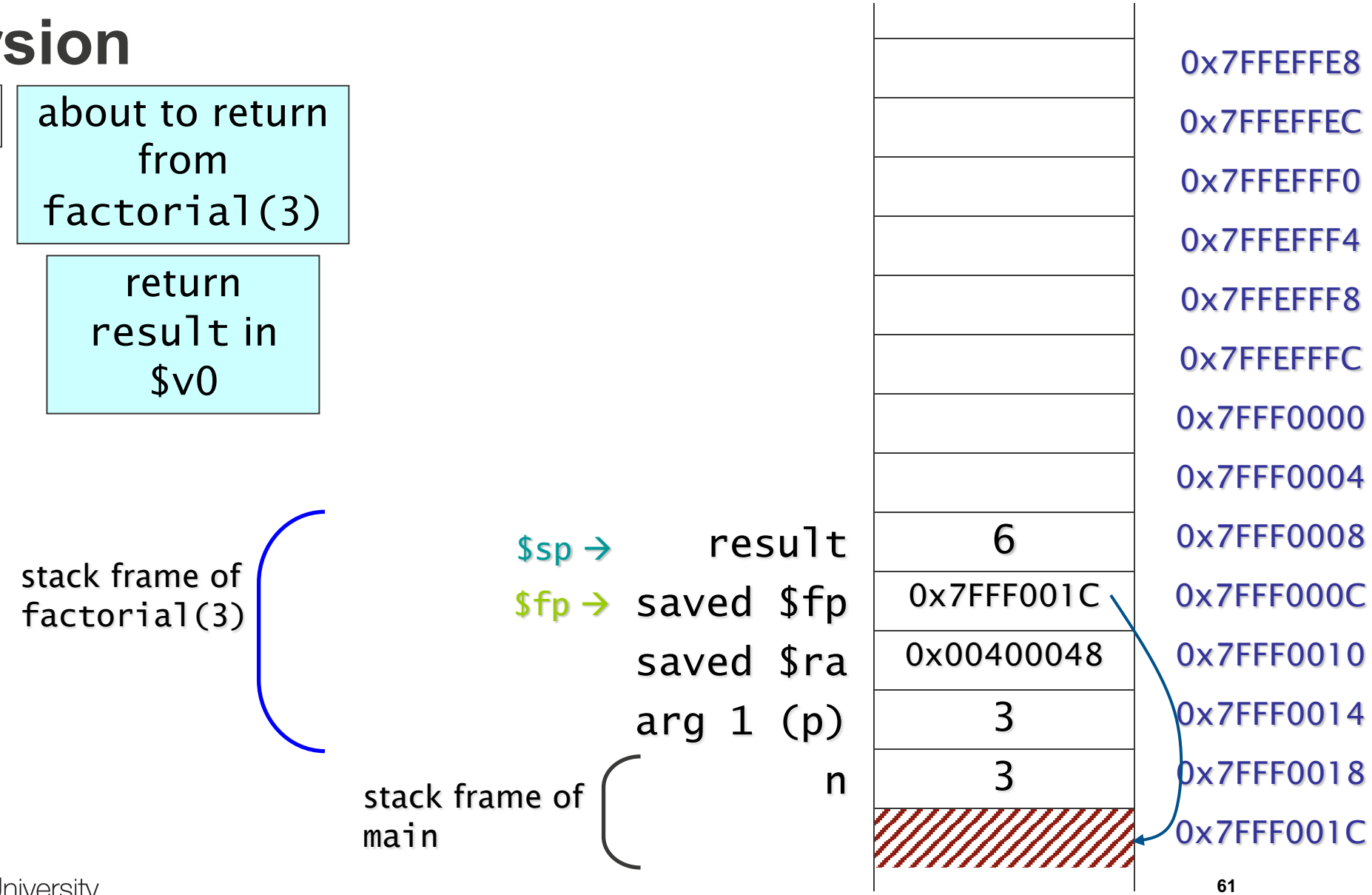


Recursion

$v0 = 6

about to return from factorial(3)

return result in $v0

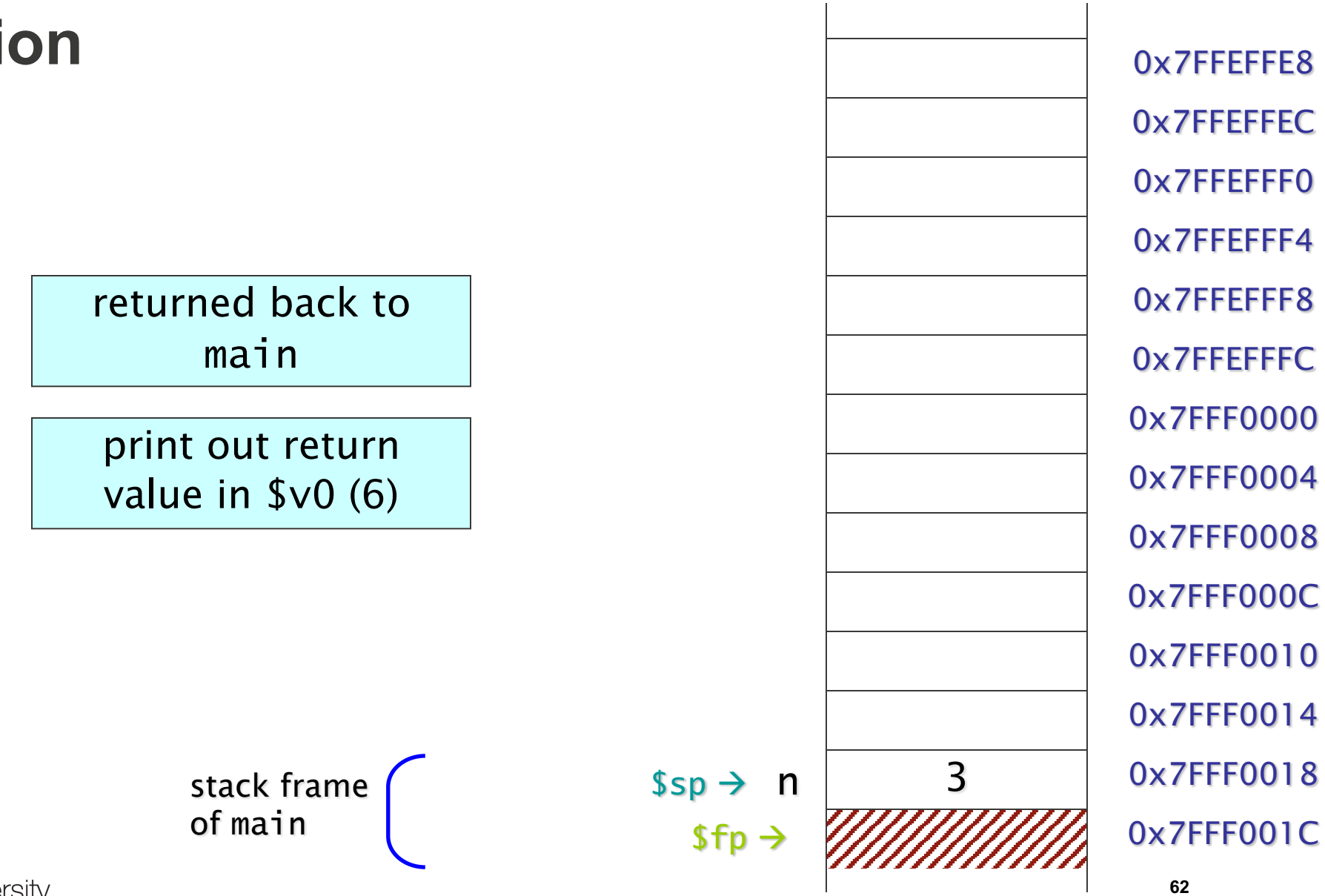


Recursion

$v0 = 6

returned back to `main`

print out return value in $v0 (6)



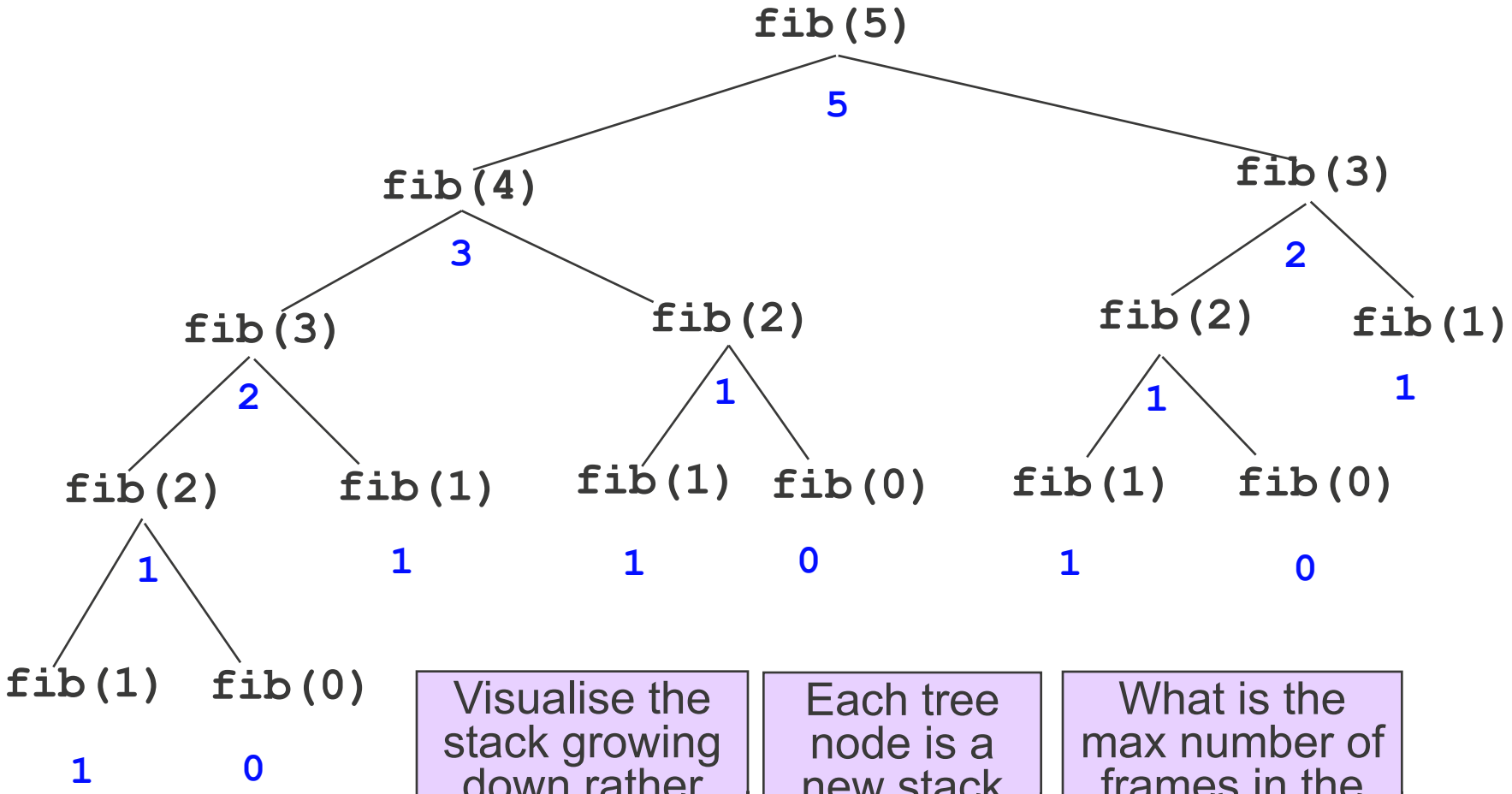
What about non-linear recursion?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return (fib(n-1) + fib(n-2))
```

What would the evolution of the system stack look like?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return (fib(n-1) + fib(n-2))
```

fibonacci's execution: call tree



Visualise the stack growing down rather than up

Each tree node is a new stack frame

What is the max number of frames in the stack?

Summary

- **Returning from functions**
  - Caller point of view
  - Calee point of view
- **Recursion**