Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 9
MIPS Recursion

FIT 1008
Introduction to Computer Science

Calling:

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

Returning:

1. Set $v0 to return value
2. Deallocate local variables
3. Restore $fp
4. Restore $ra
5. Return with jr instruction
6. Deallocate arguments
7. Restore temporary registers

Function calling convention

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**

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**

Recursive algorithms

- Solve a Large problem by solving subproblems
  - ➡ of the same kind as the original.
  - ➡ simpler to solve

Each **subproblem** is solved with the **same algorithm** …

… **until** subproblems are so "simple" that they can be solved without further reductions (**base case**)

```
def factorial(x):
    if n == 0:
        return 1
    else:
        return x*factorial(x-1)
```

Recursive procedure/method

Must have the following **components**:

1. At least one base case
2. At least one recursive call whose result is combined
3. Convergence to base case (must be “simpler”)

In **factorial**:

1. if n==0:
2. factorial(n-1)
3. (n-1)

```
def factorial(n):
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

Power

$$x^N = ?$$

$$2^{10} = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$$

$$2^5 \quad x \quad 2^5$$

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2$$

$$2^2 \quad x \quad 2^2$$

Divide and Conquer Approach

If N is even:

$$X^N = X^{\frac{N}{2}} \times X^{\frac{N}{2}}$$

If N is odd:

$$X^N = X^{\frac{N-1}{2}} \times X^{\frac{N-1}{2}} \times X$$

even

$$X^N = X^{\frac{N}{2}} \times X^{\frac{N}{2}}$$

odd

$$X^N = X^{\frac{N-1}{2}} \times X^{\frac{N-1}{2}} \times X$$

Must have the following **components**:

1. At least one base case
2. At least one recursive call whose result is combined
3. Convergence to base case (must be "simpler")

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
return value
```

fourth call	x = 2, n = 0 value= 1
third call	x = 2, n = 1 value= 1
second call	x = 2, n = 2 value= 1
first call	x = 2, n = 5 value= 1

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
return value
```

fourth call returns 1

third call	x = 2, n = 1 value= 1
second call	x = 2, n = 2 value= 1
first call	x = 2, n = 5 value= 1

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
return value
```

fourth call returns 1

third call returns 1*1*2=2

second call	x = 2, n = 2 value= 1
first call	x = 2, n = 5 value= 1

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
    else:
        value = value*value*x
return value
```

fourth call returns 1

third call returns 1*1*2=2

second call returns 2*2=4

first call { x = 2, n = 5 ; value= 1

```
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
    else:
        value = value*value*x
return value
```

fourth call returns 1

third call returns 1*1*2=2

second call returns 2*2=4

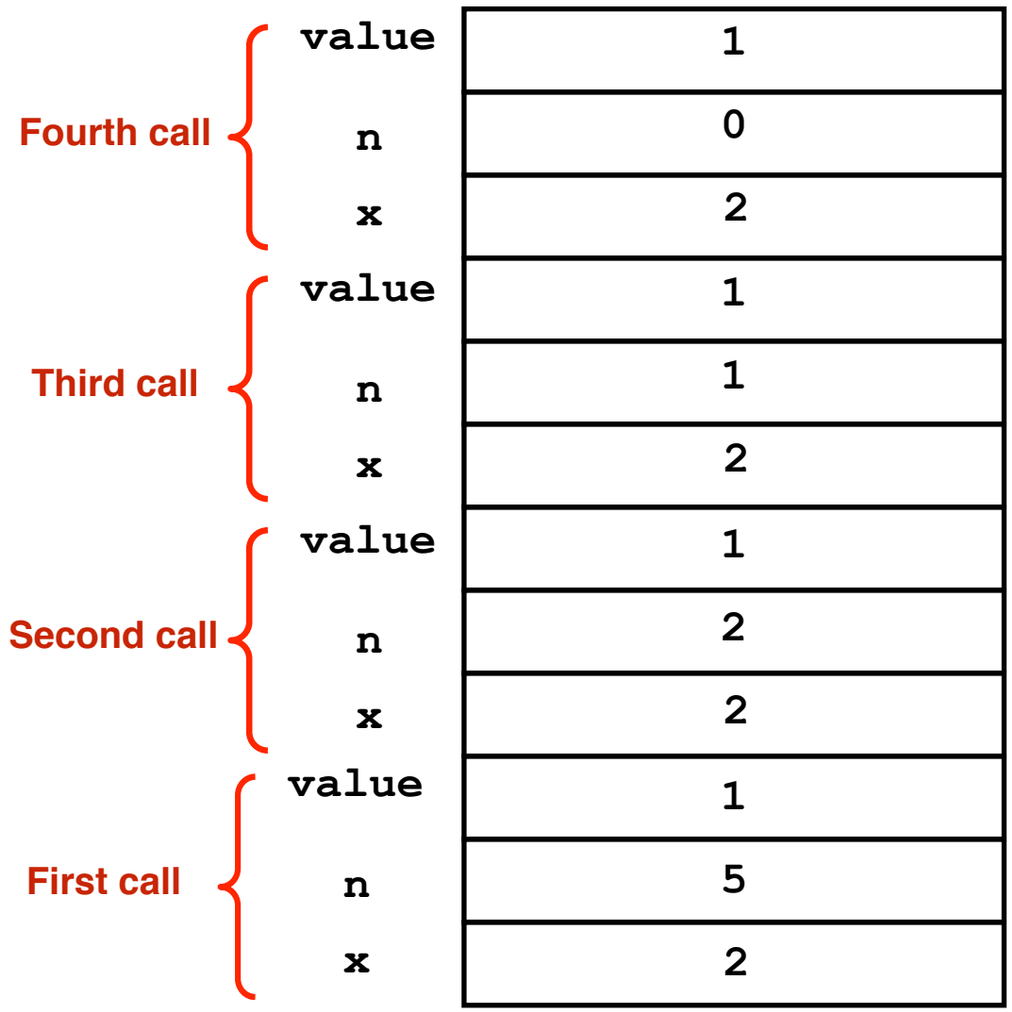
first call returns 4*4*2=32

Recursion: the Runtime **Stack**

- The system implements recursion by using the runtime stack.
- Each recursive call reserves a portion of the stack to store parameters and local variables **(push)**.
- Control is then given to the function to modify its variables according to its definition.
- Each time a call finishes: area is removed after transferring the return value to its place **(pop)**.

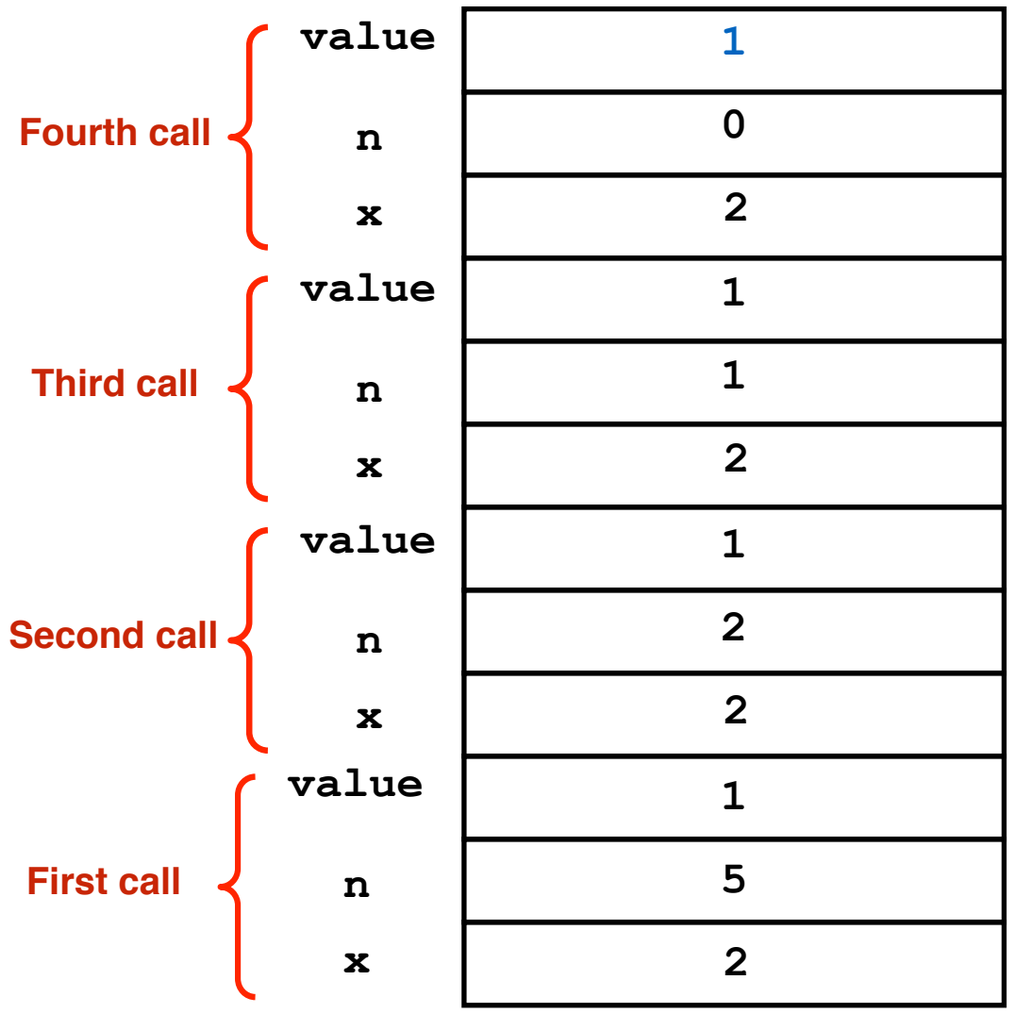
Simplified Runtime Stack

fourth call	x = 2, n = 0 value= 1
third call	x = 2, n = 1 value= 1
second call	x = 2, n = 2 value= 1
first call	x = 2, n = 5 value= 1



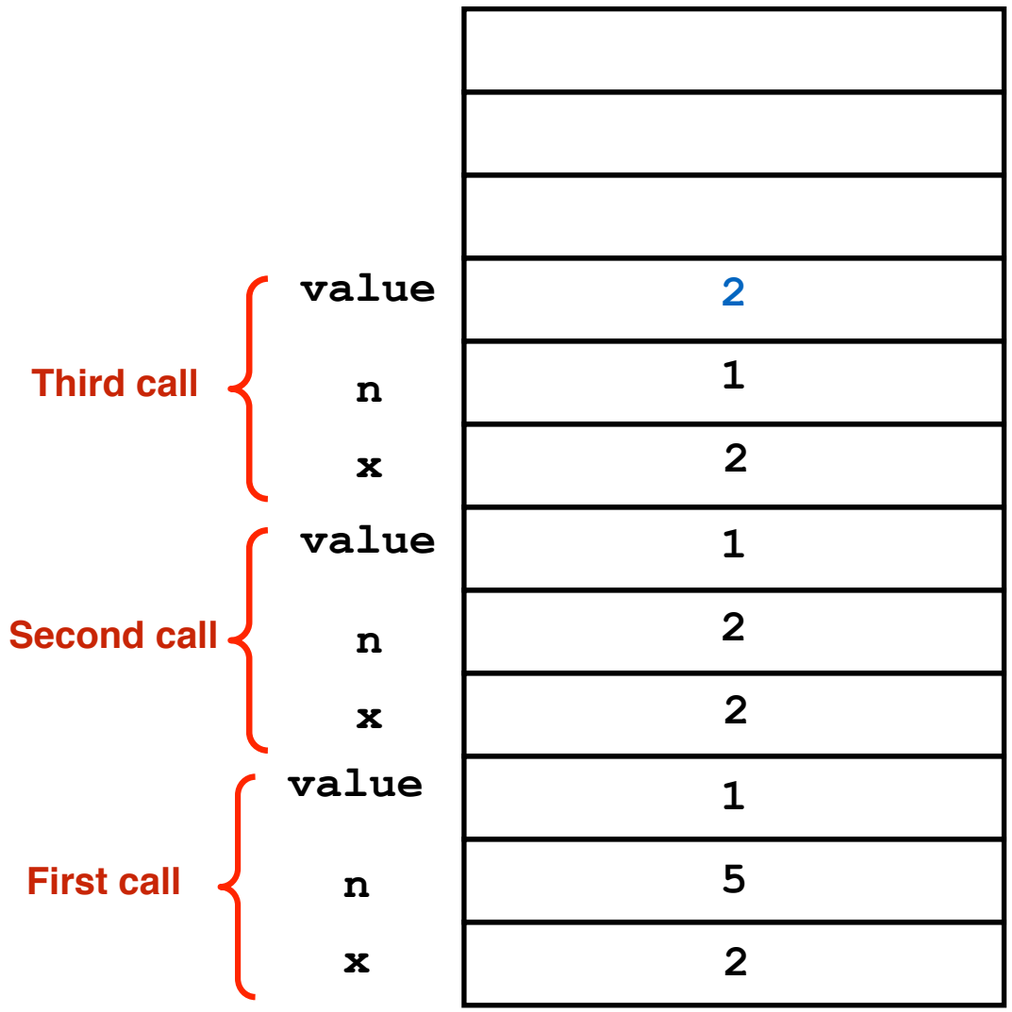
Simplified Runtime Stack

fourth call	x = 2, n = 0 value= 1	4th call returns 1
third call	x = 2, n = 1 value= 1	
second call	x = 2, n = 2 value= 1	
first call	x = 2, n = 5 value= 1	



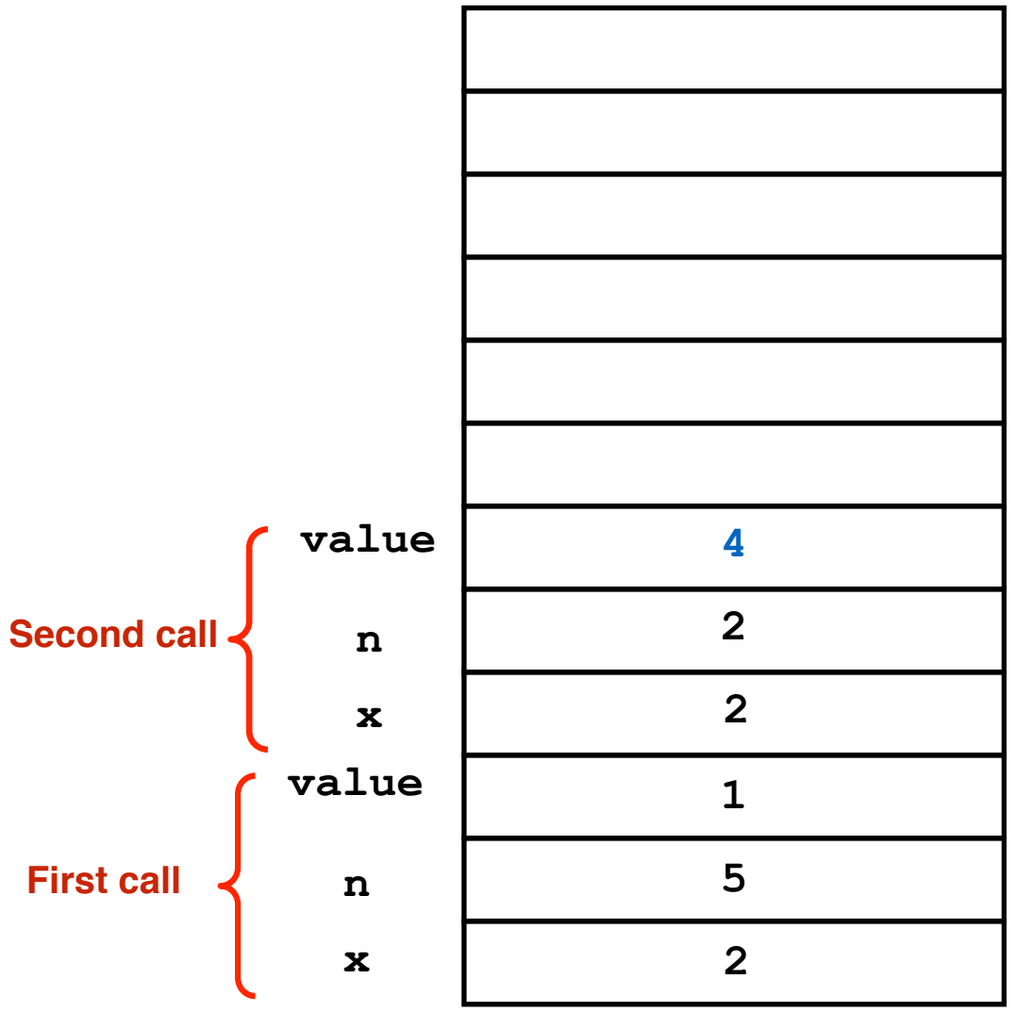
Simplified Runtime Stack

fourth call	x = 2, n = 0 value= 1	4th call returns 1
third call	x = 2, n = 1 value= 1	3rd call returns 1*1*2=2
second call	x = 2, n = 2 value= 1	
first call	x = 2, n = 5 value= 1	

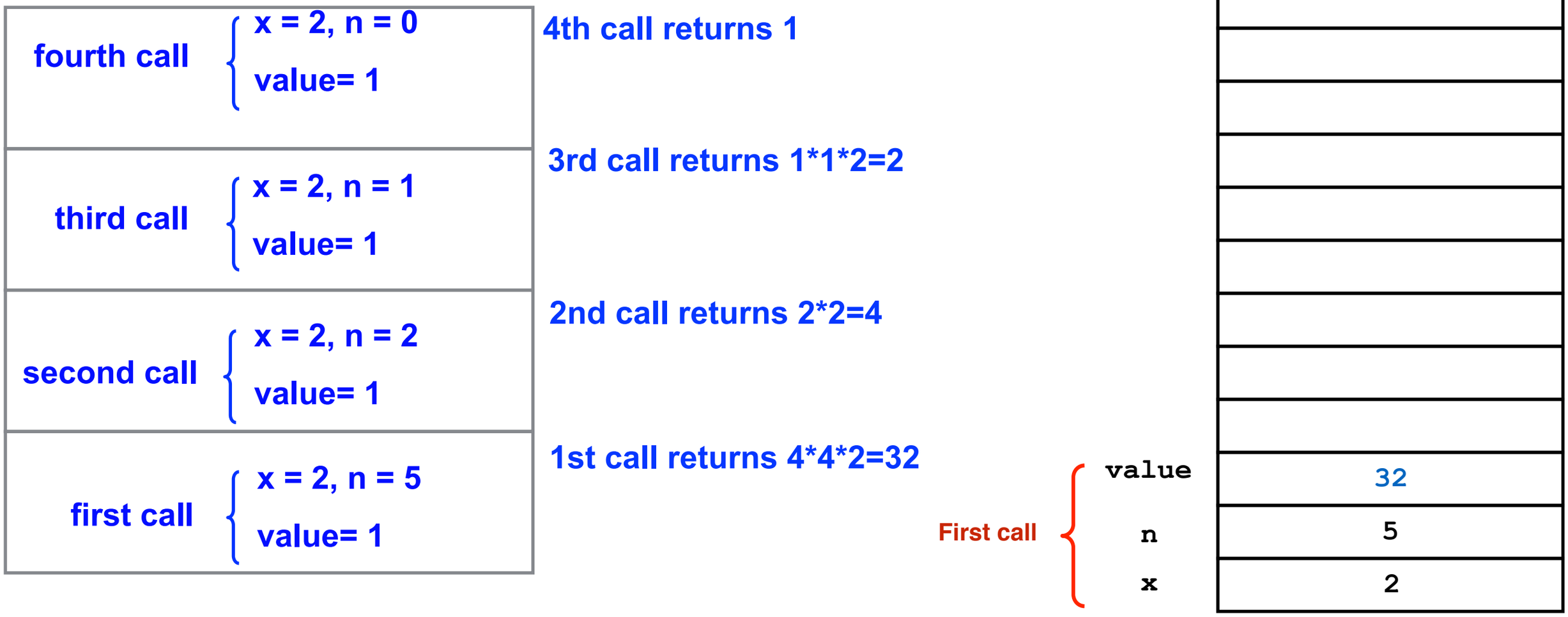


Simplified Runtime Stack

fourth call	x = 2, n = 0 value= 1	4th call returns 1
third call	x = 2, n = 1 value= 1	3rd call returns 1*1*2=2
second call	x = 2, n = 2 value= 1	2nd call returns 2*2=4
first call	x = 2, n = 5 value= 1	



Simplified Runtime Stack

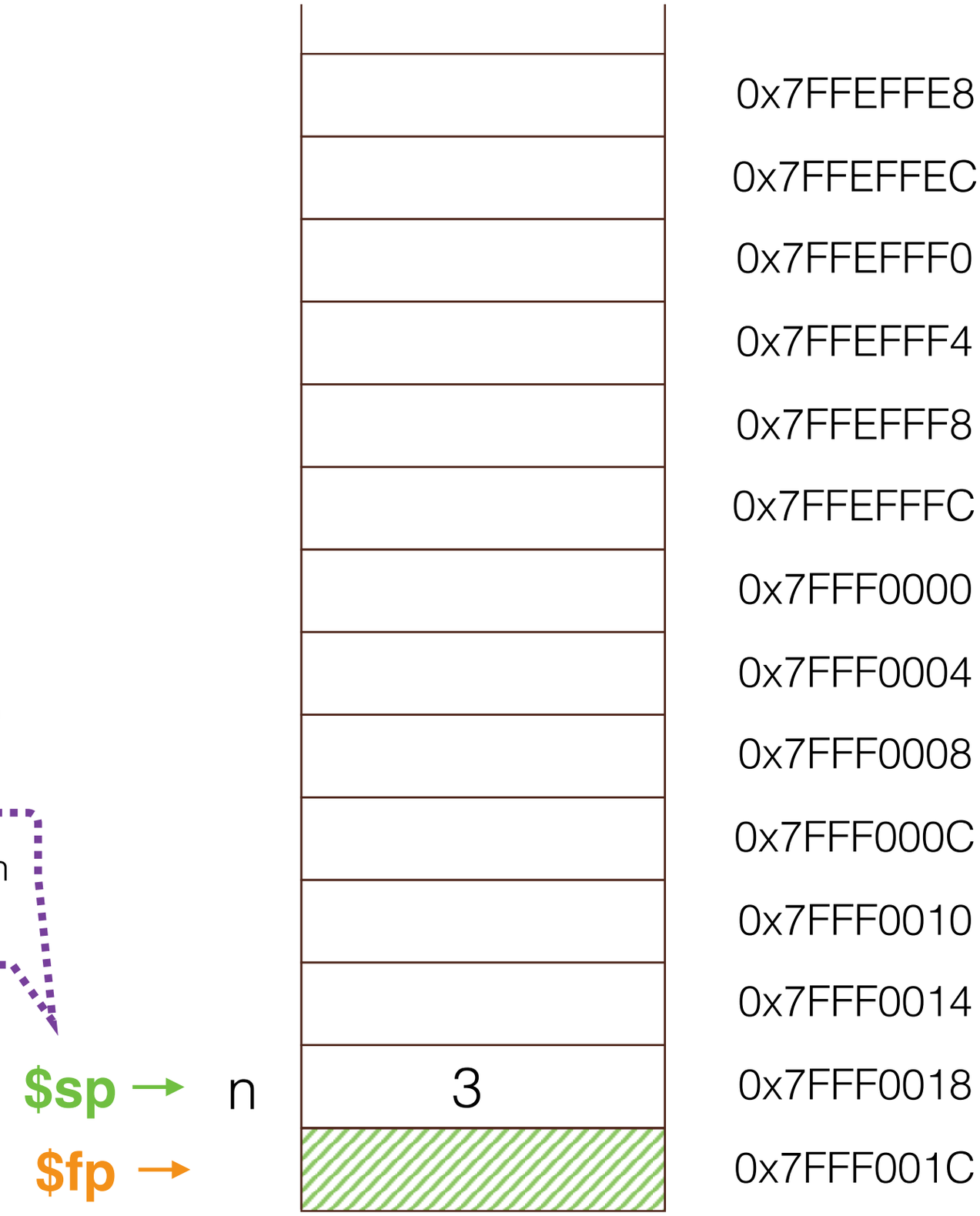


MIPS

```
def main():
    n = 0
    n = int(input())
    print(factorial(n))


 def factorial(p):
result = 0
if p > 1:
    result = p * factorial(p - 1)
else:
    result = 1
return result
```

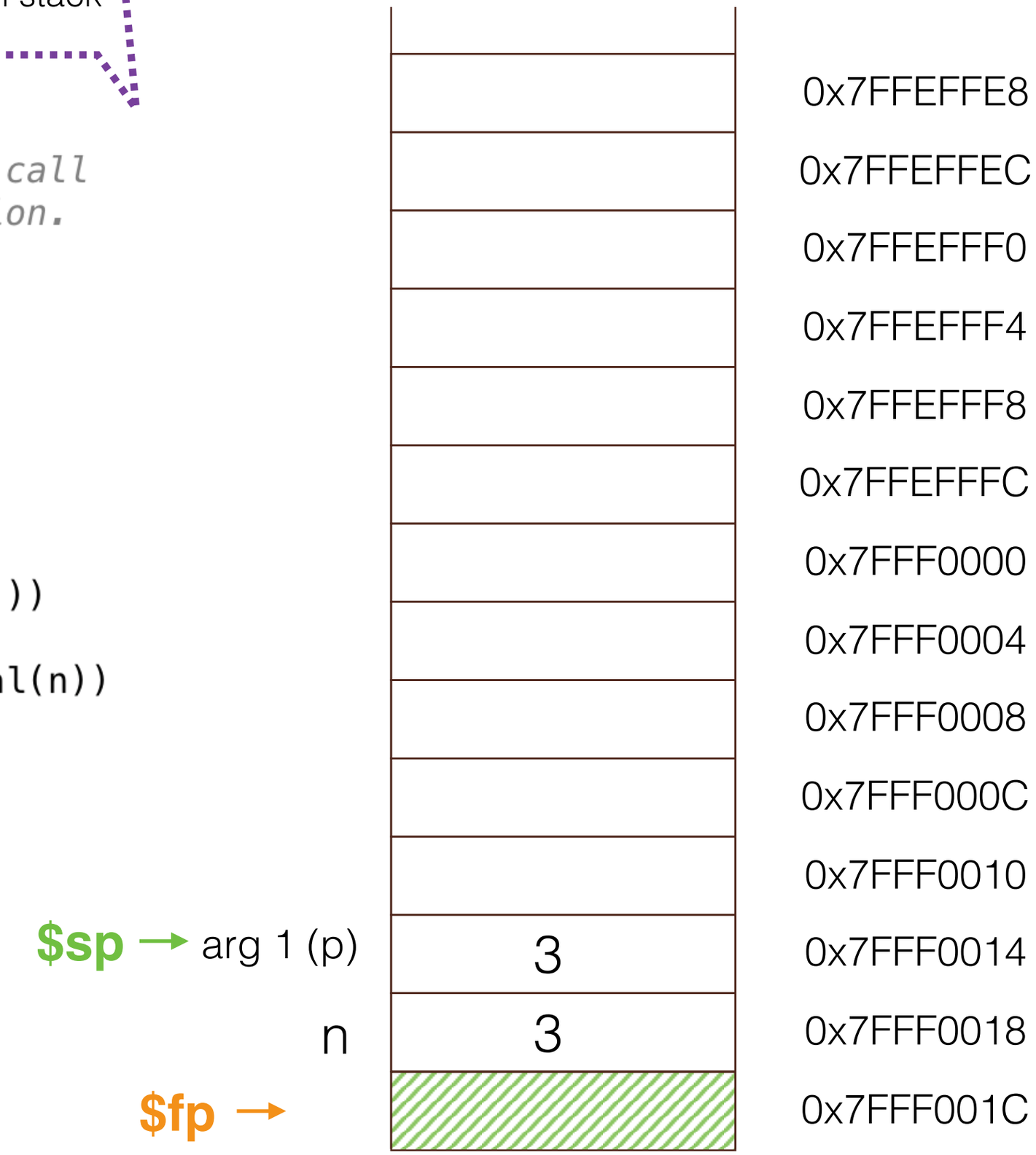
```python
# Main program to call
# factorial function.

def factorial(p):
    ...
    return result

def main():
    n = 0

    n = int(input())

    print(factorial(n))
```

Frame for main function with local variable n



pass an argument to factorial,
so we push argument on stack

```
# Main program to call
# factorial function.

def factorial(p):
    ...
    return result

def main():
    n = 0

    n = int(input())

    print(factorial(n))
```



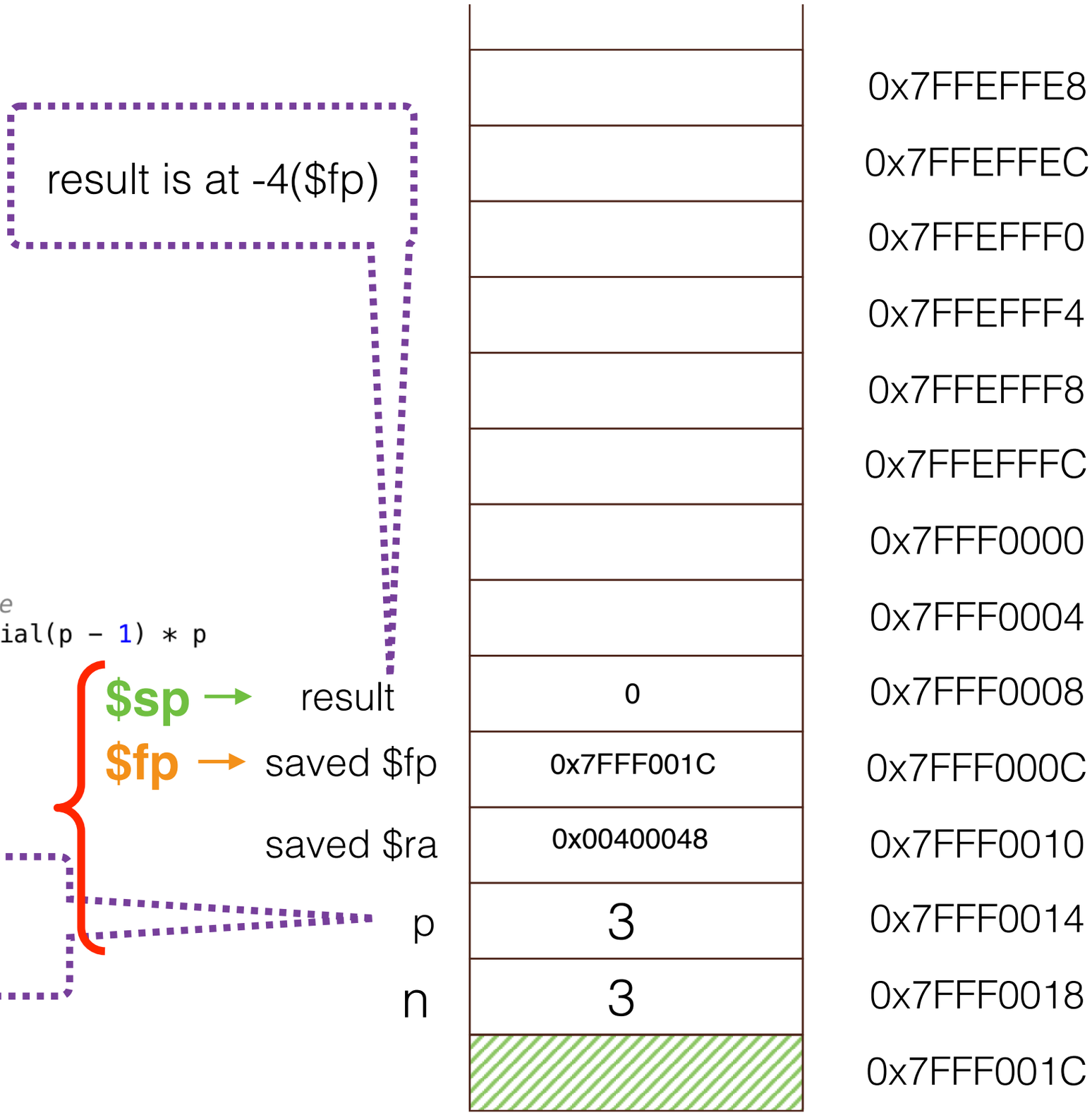
```python
def main():
    n = 0
    n = int(input())
    print(factorial(n))
```

```
main:
        #copy fp into sp
        addi $fp, $sp, 0

        #allocate space for 1 local variable
        addi $sp, $sp, -4 #n

        # read integer
        li $v0, 5
        syscall
        sw $v0, -4($fp) #n = input()

        # prepare caller to call factorial function

        #CALLER PREP: 1. save temp registers -- none

        #CALLER PREP:2. pass arguments on stack
        addi $sp, $sp, -4 # space for one argument
        lw $t0, -4($fp)
        sw $t0, 0($sp) #copy argument

        #CALLER PREP: 3. call function using jal
        jal factorial

        #CALLER CLEAN: 1 clears arguments off stack
        addi $sp, $sp, 4 # 1 argument

        #CALLER CLEAN: 2. restore temp reg
        # none

        #CALLER CLEAN: 3. use return value in $v0
        addi $a0, $v0, 0
        li $v0, 1
        syscall # print $v0

        addi $sp, $sp, 4 # deallocate local variables

        li $v0, 10
        syscall # exit
```

```python
def factorial(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)



saves $ra and $sp on stack

```
def factorial(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
    sw $ra, 4($sp)
    sw $fp, 0($sp)
    addi $fp, $sp, 0

    # 1 * 4 = 4 bytes local
    addi $sp, $sp, -4
    sw   $0, -4($fp) # result = 0

    # if p <= 1 ...
    lw $t0, 8($fp)  # p
    addi $t1, $0, 1
    slt $t0, $t1, $t0
    bne $t0, $0, rec

    # result = 1
    addi $t0, $0, 1
    sw $t0, -4($fp)  # result
    j end
```

continues…

```
rec:    # Recursive call.

        # 1 * 4 = 4 bytes arg.
        addi $sp, $sp, -4

        # argument 1 = p-1
        lw $t0, 8($fp)    # p
        addi $t0, $t0, -1 # p-1
        sw $t0, 0($sp)    # arg 1
        jal factorial

        # Clean up argument.
        addi $sp, $sp, 4

        # Multiply by p.
        lw $t0, 8($fp)  # p
        mult $v0, $t0
        mflo $t0

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

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

        # Destroy local variable
        addi $sp, $sp, 4


        # Function exit.
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addi $sp, $sp, 8
        jr $ra
```

```python
def factorial(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
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        addi $fp, $sp, 0



        # 1 * 4 = 4 bytes local
        addi $sp, $sp, -4
        sw   $0, -4($fp) # result = 0


        # if p <= 1 ...
        lw $t0, 8($fp)  # p
        addi $t1, $0, 1
        slt $t0, $t1, $t0
        bne $t0, $0, rec

        # result = 1
        addi $t0, $0, 1
        sw $t0, -4($fp)  # result
        j end
```

```
rec:    # Recursive call.

    # 1 * 4 = 4 bytes arg.
    addi $sp, $sp, -4

    # argument 1 = p-1
    lw $t0, 8($fp)    # p
    addi $t0, $t0, -1 # p-1
    sw $t0, 0($sp)    # arg 1
    jal factorial

    # Clean up argument.
    addi $sp, $sp, 4
```

```
    # Multiply by p.
    lw $t0, 8($fp)  # p
    mult $v0, $t0
    mflo $t0
```

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

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

```
    # Destroy local variable
    addi $sp, $sp, 4


    # Function exit.
    lw $fp, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    jr $ra
```

continues…

```python
def factorial(p):
    result = 0
    if p > 1:
        result = p * factorial(p - 1)
    else:
        result = 1
return result
```

```
factorial:
        #CALLEE PREP: 1. save $ra and $fp on stack
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $fp, 0($sp)

        #CALLEE PREP: 2. copy $sp into $fp
        addi $fp, $sp, 0

        #CALLEE PREP: 3 ALLOCATE LOCAL VARIABLES
        addi $sp, $sp, -4
        sw $0, -4($fp) # result = 0

        #BUSINESSS
        # if p > 1 go to recursive call
        lw $t0, 8($fp) #t0 = p
        addi $t1, $0, 1 #t1 =1
        bgt $t0, $t1, rec

        #else
        addi $t1, $0, 1
        sw $t1, -4($fp) #result = 1
        j end

        rec:
                #CALLER AGAIN

                #CALLER PREP: 1. save temp registers -- none

                #CALLER PREP:2. pass arguments on stack
                addi $sp, $sp, -4 # space for one argument
                lw $t0, 8($fp) #t0 = p
                addi $t0, $t0, -1 #t0 = p-1
                sw $t0, 0($sp) #copy argument

                #CALLER PREP: 3. call function using jal
                jal factorial

                #CALLER CLEAN: 1 clears arguments off stack
                addi $sp, $sp, 4 # 1 argument

                #CALLER CLEAN: 3. use return value in $v0
                #multiply by p
                lw $t0, 8($fp) #t0 = p
                mult $t0, $v0 #lo = p*factorial(p-1)
                mflo $t0
                sw $t0, -4($fp) #result = lo

        end:

                #CALLEE CLEAN: 1. $v0 to return value
                #return result
                lw $v0, -4($fp)

                #CALLE CLEAN: 2. deallocate local variables
                addi $sp, $sp, 4 # 1 local

                #CALLEE CLEAN: 3 restore saved $ra
                lw $fp, ($sp)
                lw $ra, 4($sp)
                addi $sp, $sp, 8

                #CALLEE CLEAN: 4 return to caller
                jr $ra
```

Summary

- Recursion in MIPS
- Recursion is memory-intensive
- Function calling/returning convention applies to implement recursion.