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

Lecture 8
Function in MIPS

FIT 1008
Introduction to Computer Science

Objectives for this lecture

- Function calling.
- To be able to access function arguments in MIPS
- To understand the steps for **function return**
- To be able to implement function return in MIPS

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

power.py

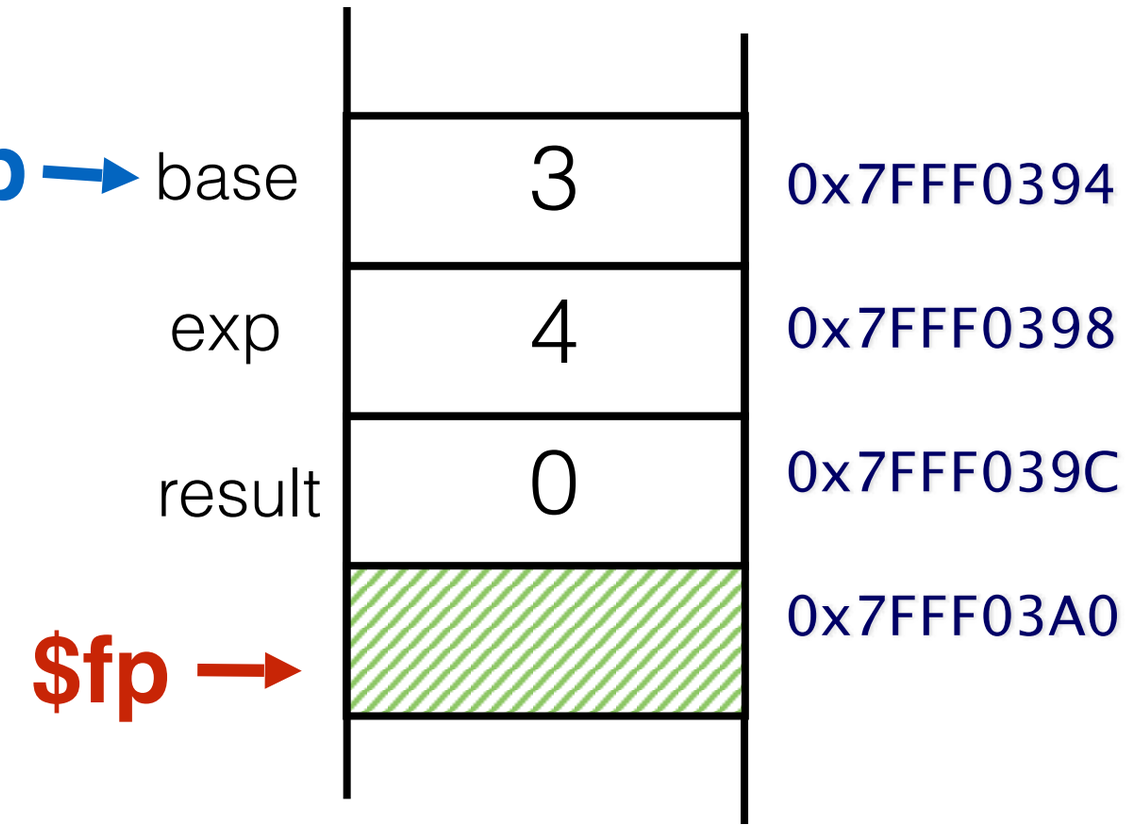
```
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result

main()
```

two *results*, but they are different…. local variables

power.py

```
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result


main()
```

Assume user has entered **3** for **base** and **4** for **exp**



power.py

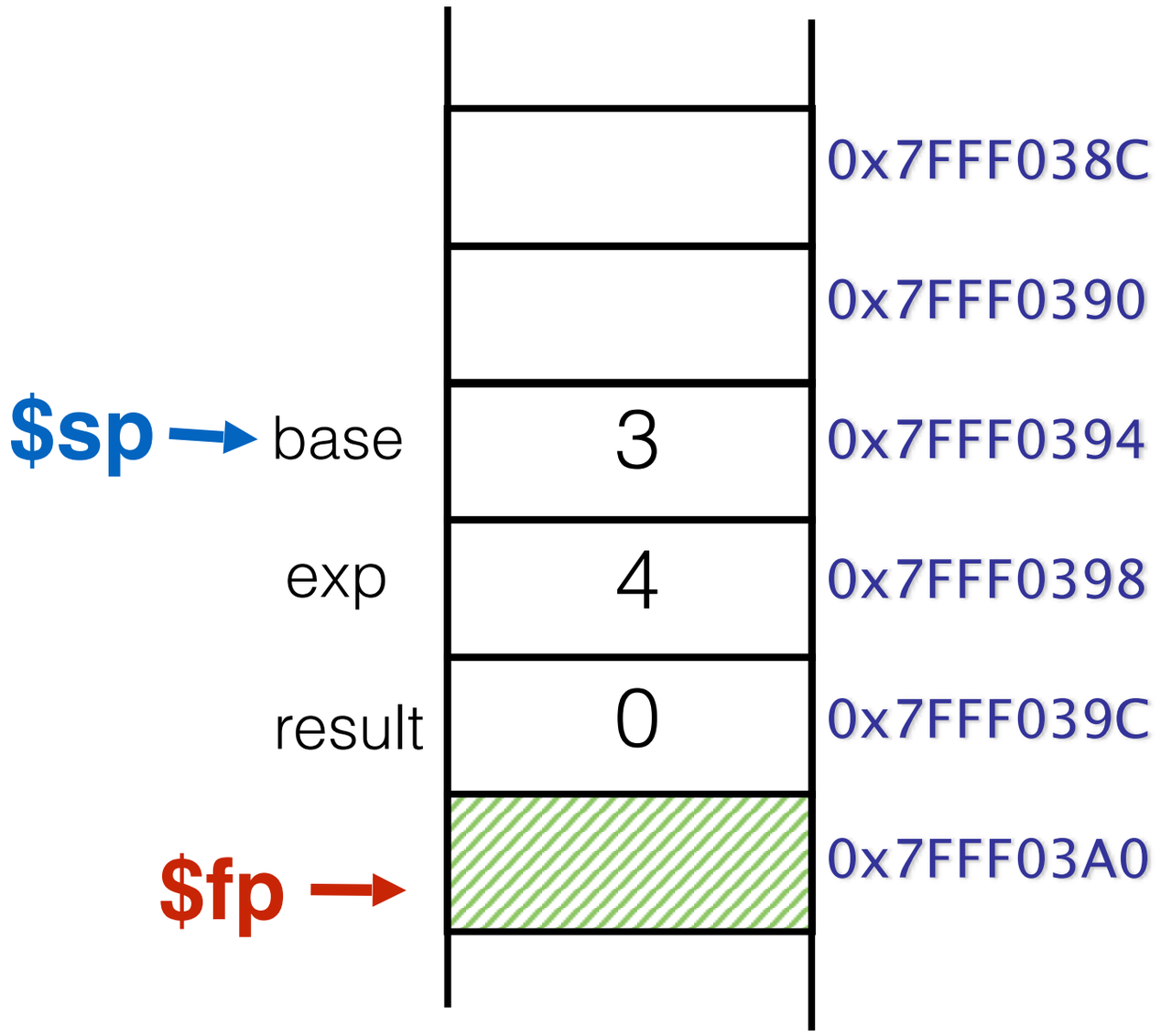
```python
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result

main()
```

```
        .text
main:   # 3 * 4 = 12 bytes local
        addi $fp, $sp, 0
        addi $sp, $sp, -12

            # Initialize locals
        sw $0, -12($fp)
            sw $0, -8($fp)
        sw $0, -4($fp)

            addi $v0, $0, 5
        syscall
        sw $v0, -12($fp)  # base

        addi $v0, $0, 5
        syscall
        sw $v0, -8($fp)   # exp
```

Caller

Step 1: Save temporary registers by pushing their values on stack

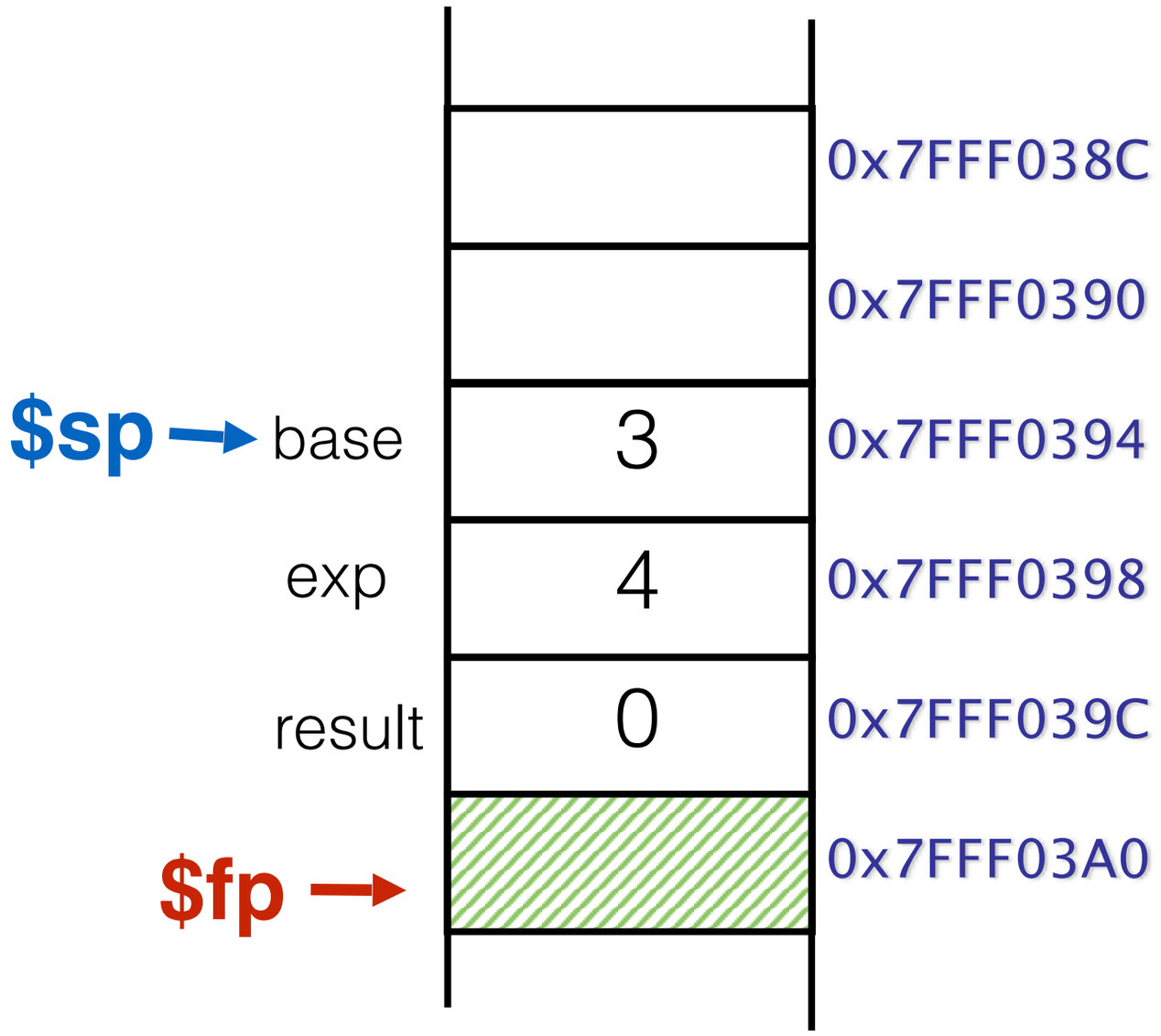
(not needed in this program)



Caller

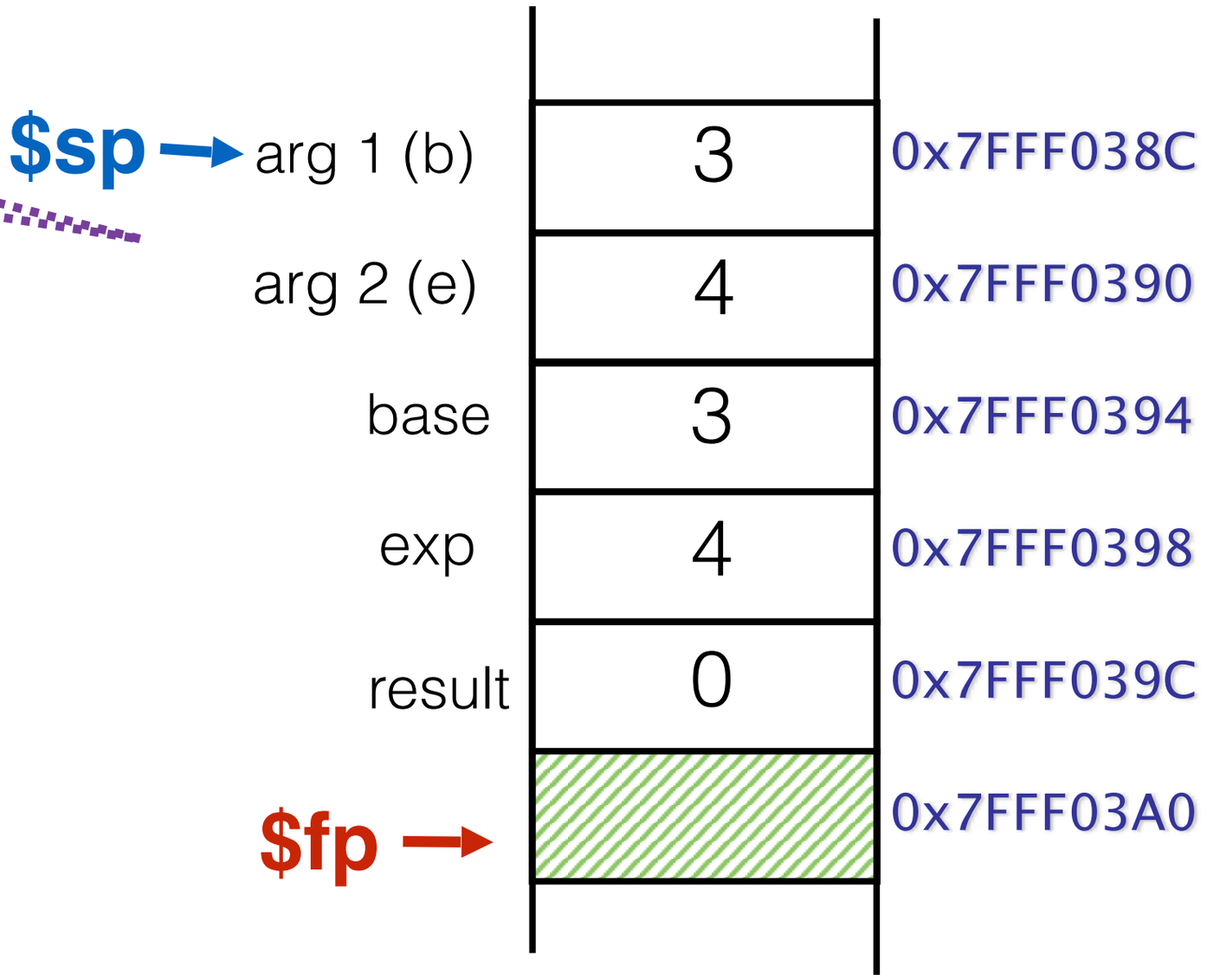
Step 2: Push function arguments onto the stack

```
def power(b, e):
```



Caller

Note the **offsets.**
b at 0($sp)
e at 4($sp)

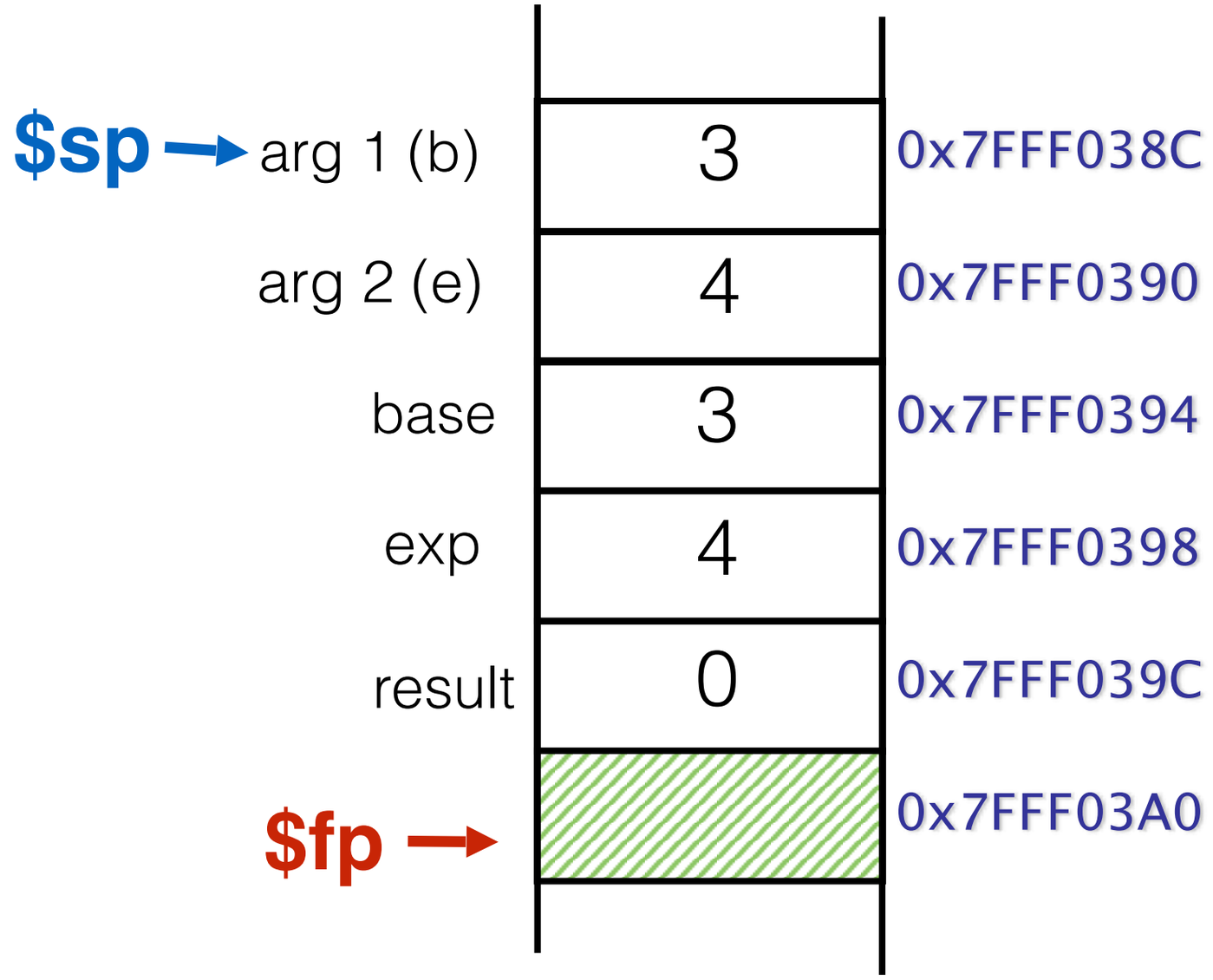


Step 2: Push function arguments onto the stack

```
def power(b, e):
```

Caller

```
.
# push 2 * 4 = 8 bytes
# of arguments
addi $sp, $sp, -8

# arg 1 = base
lw $t0, -12($fp) # base
sw $t0, 0($sp)   # arg 1

# arg 2 = exp
lw $t0, -8($fp)  # exp
sw $t0, 4($sp)   # arg 2
```

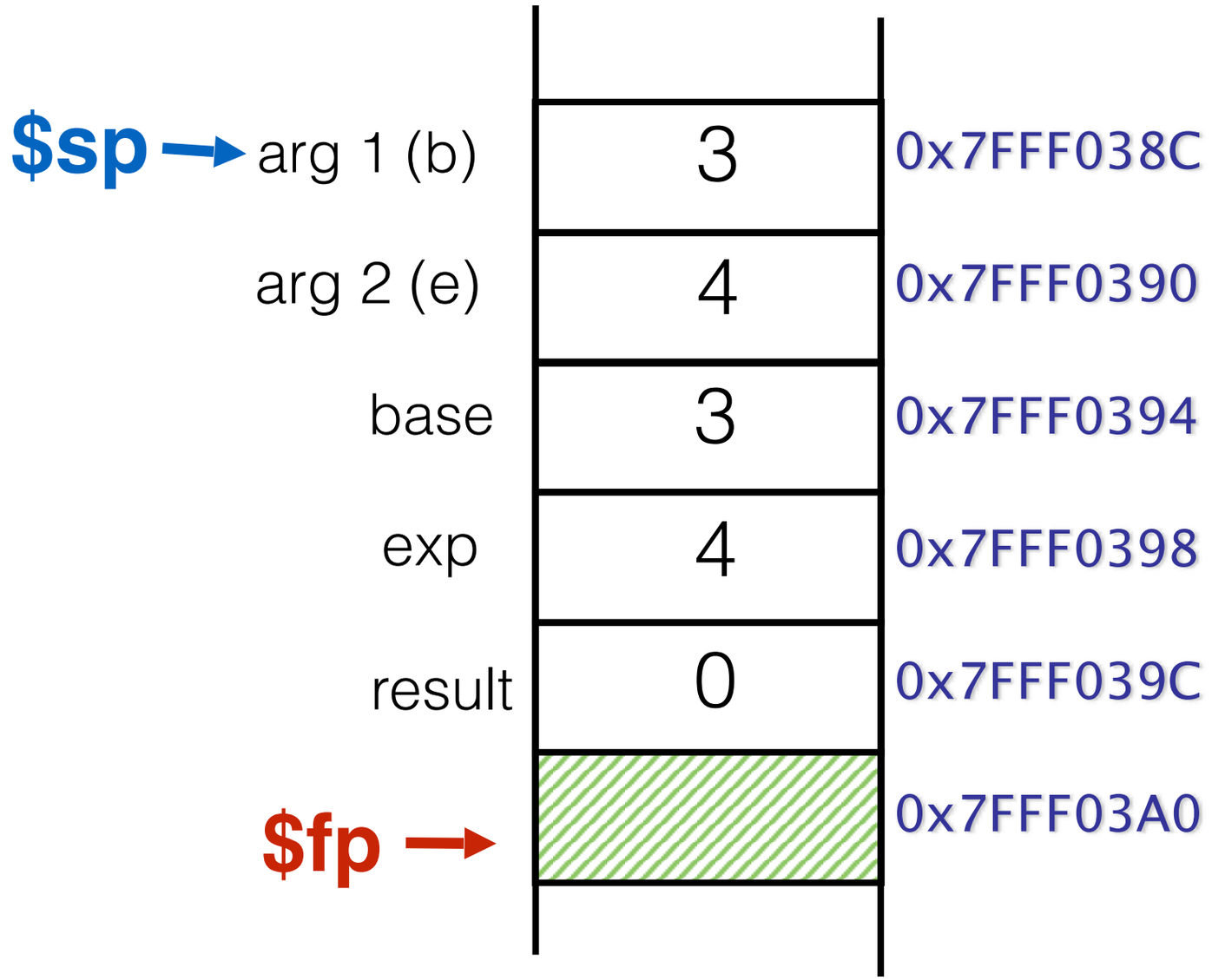


Caller

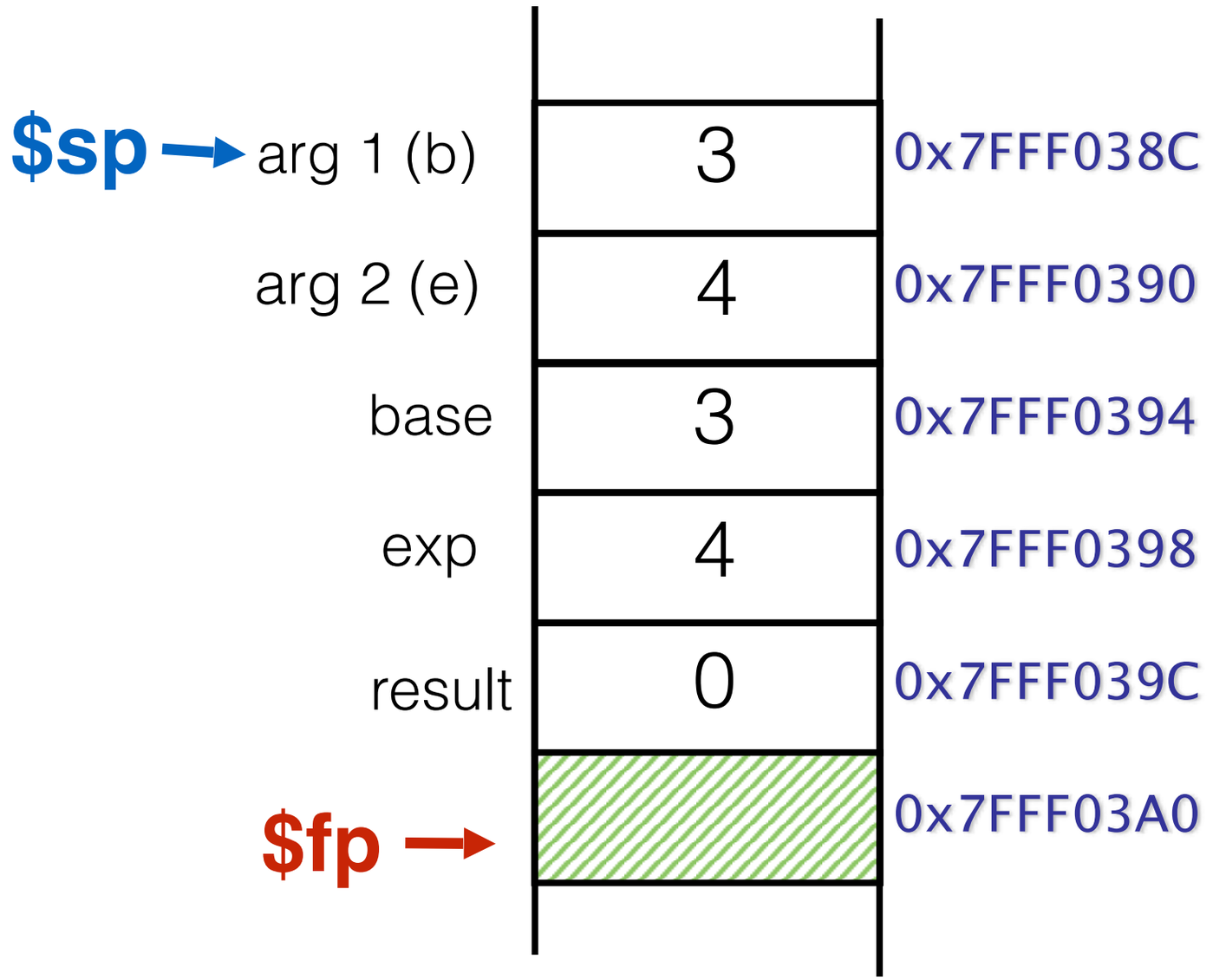
jal power

Step 3: Call function with **jal**

(no visible effect on stack)



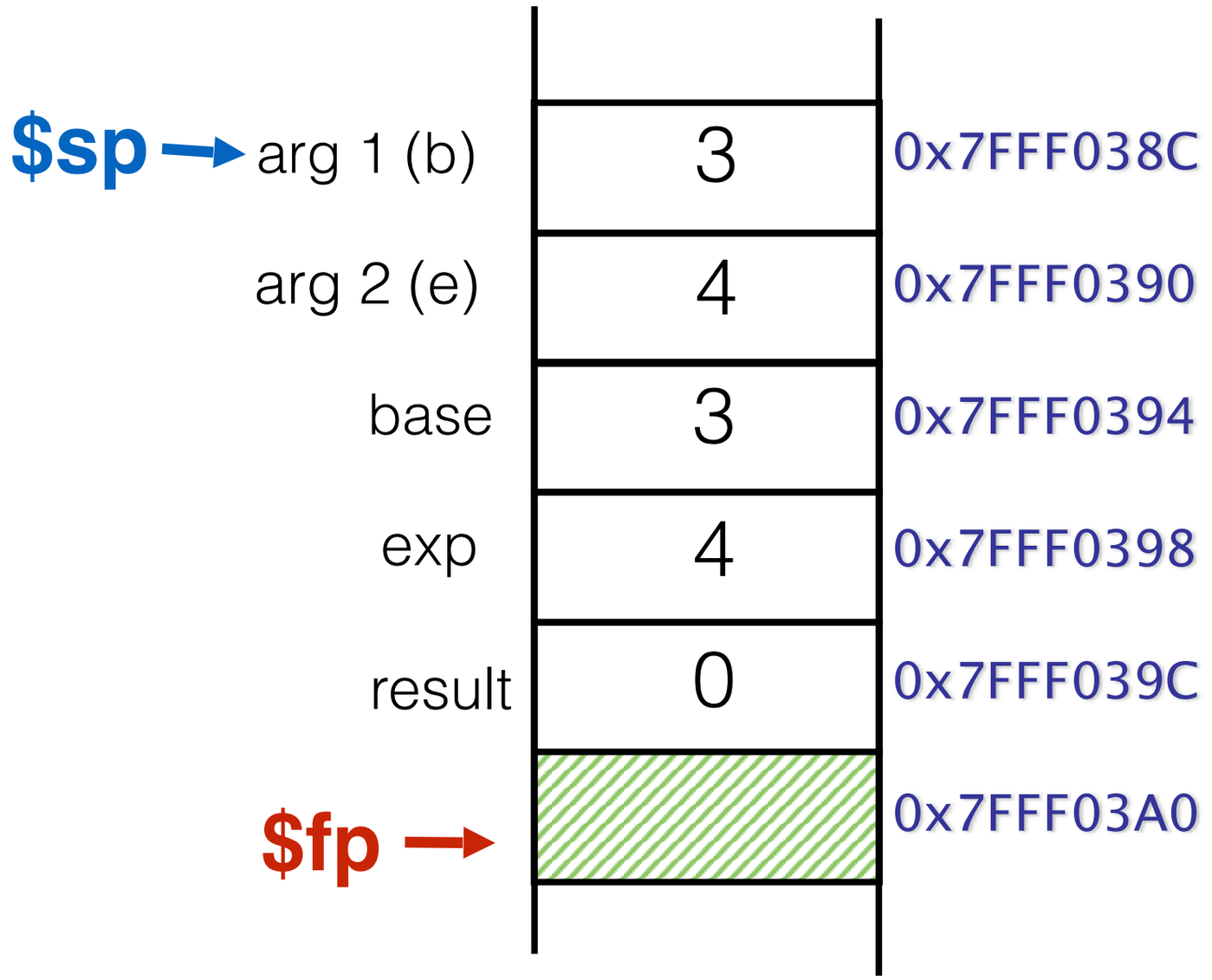
```python
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result

main()
```



Callee

Step 4 and 5:
Save **$ra** and **$fp**

(can do both steps at once)

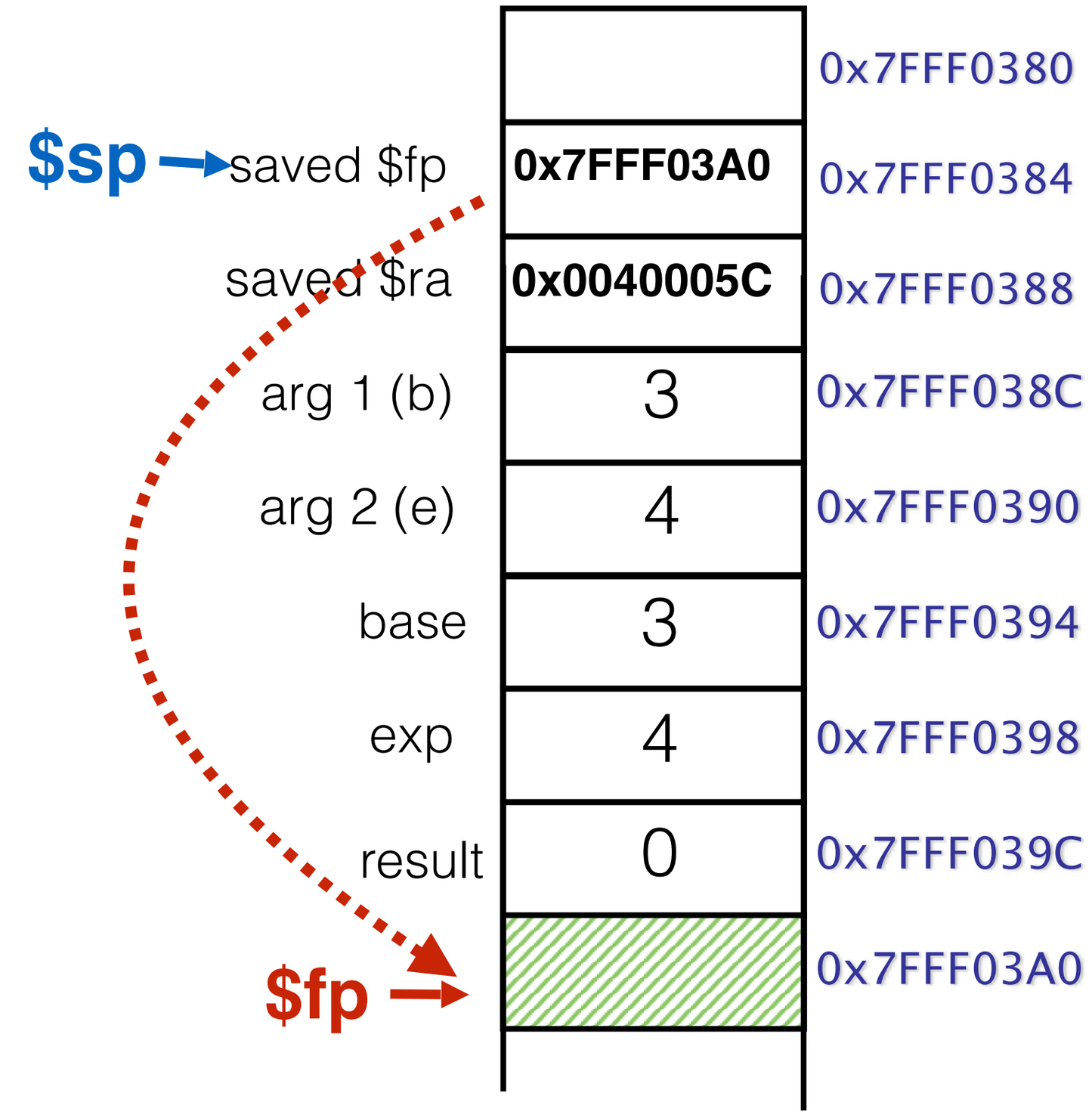


Callee

Saved **$sp** contains address of other location in stack

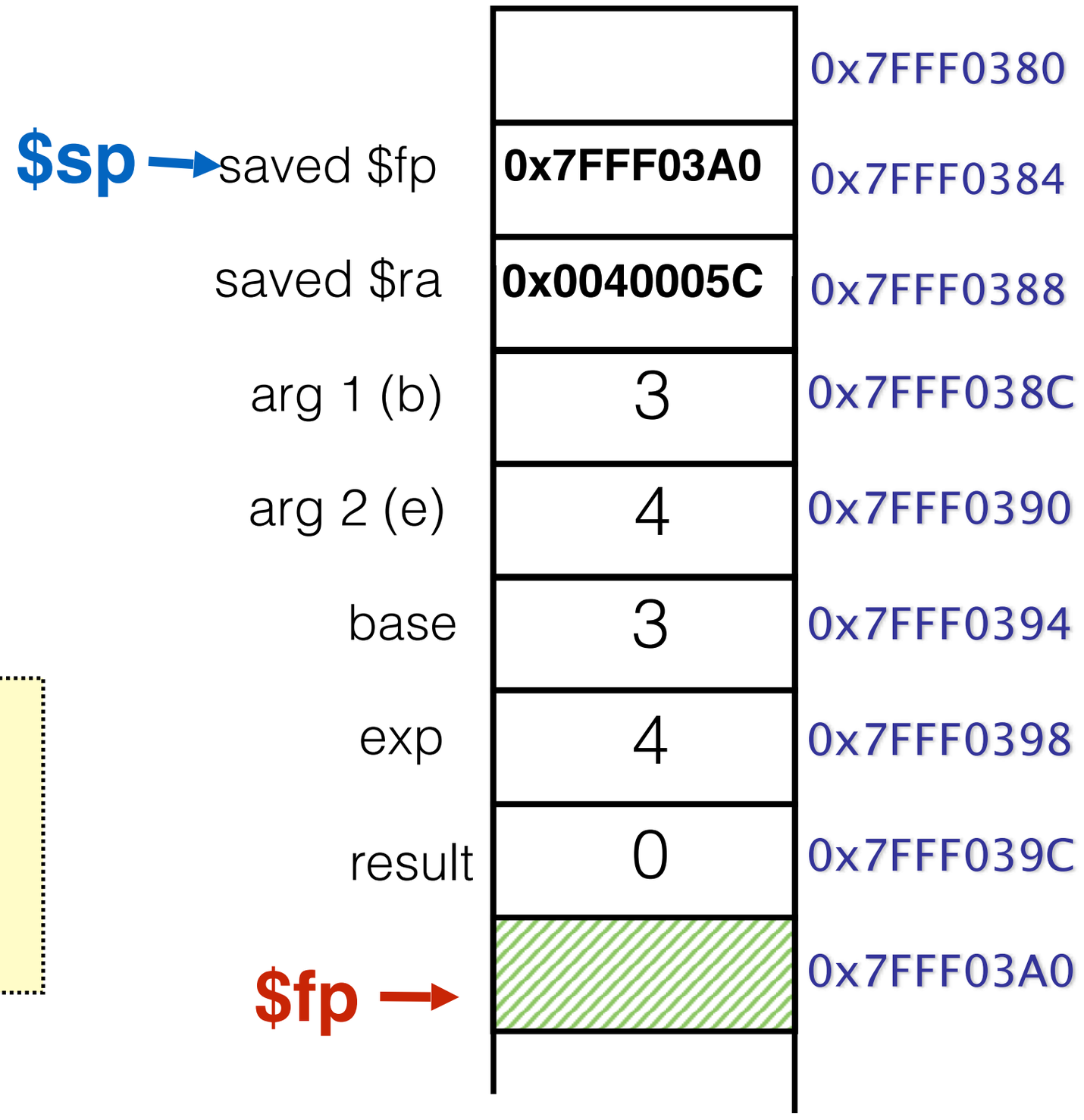
Step 4 and 5:
Save **$ra** and **$fp**

(can do both steps at once)



Callee

```
power:  # Save $ra and $fp
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
```

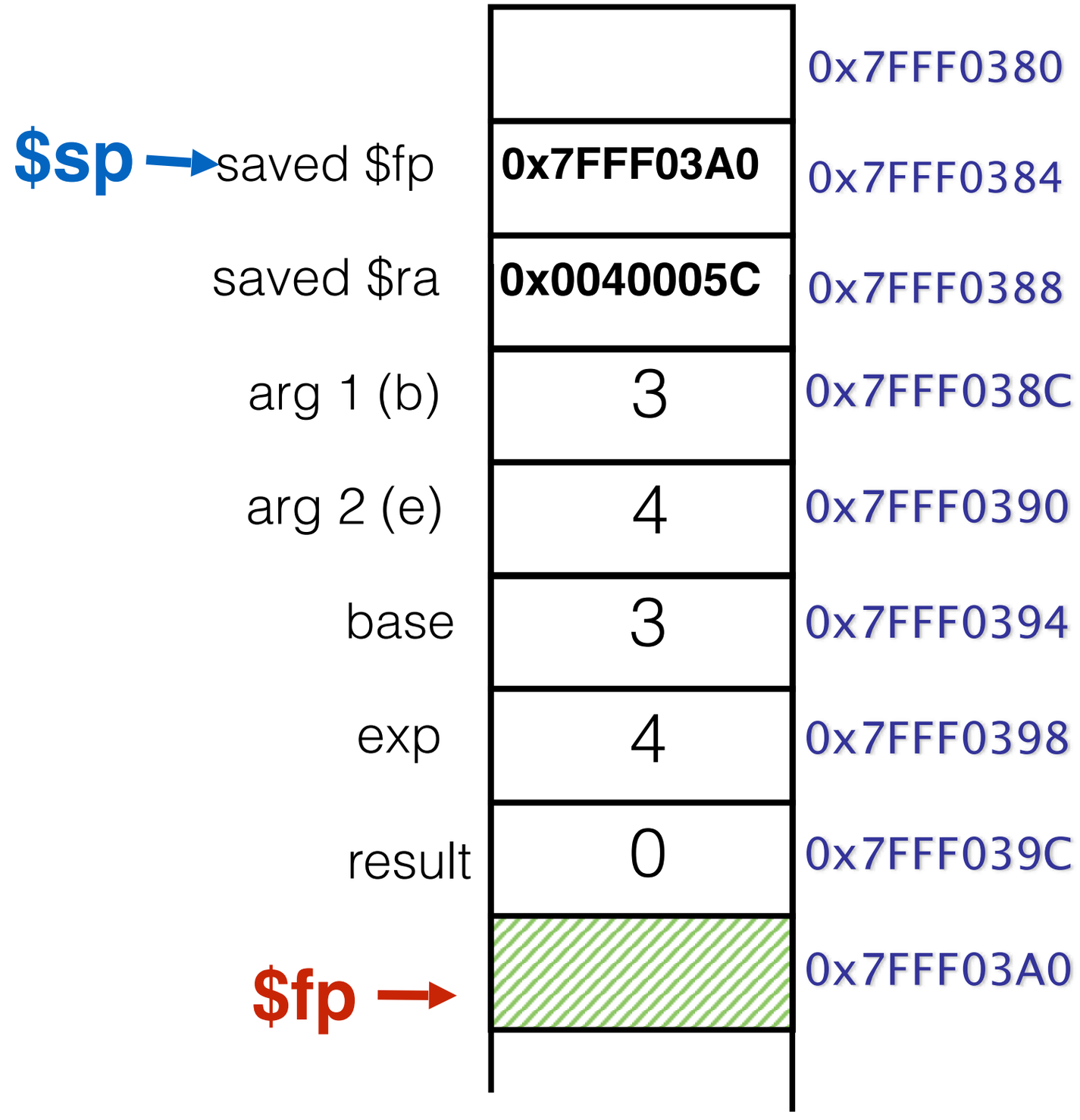


Callee

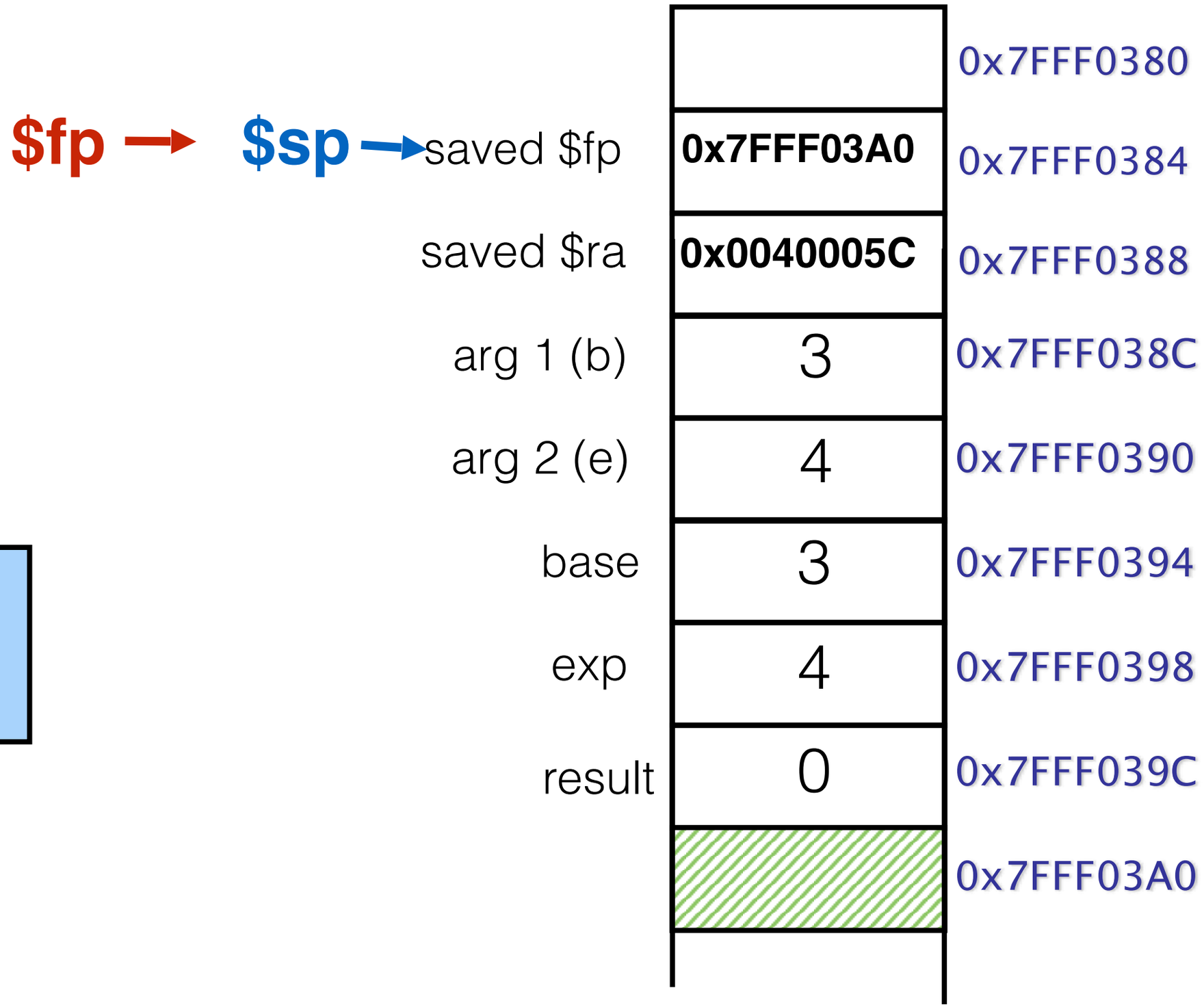
```
# Copy $sp to $fp
addi $fp, $sp, 0
```

Step 6:
Save **$sp** into **$fp**

now main's local variables are inaccessible



Callee



Step 7:
allocate local variables

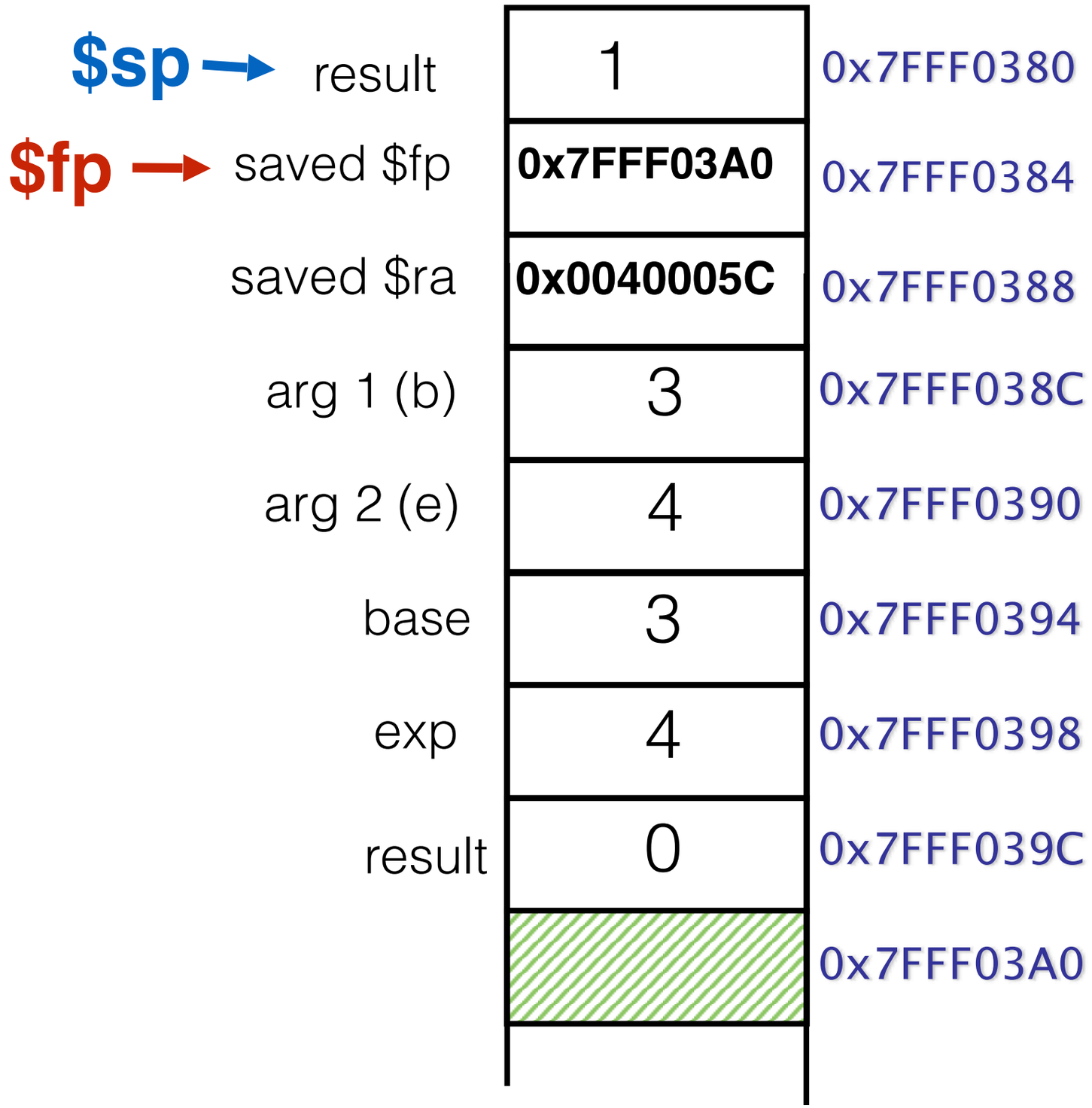
in this function,
one local variable **result**

Callee

```
# Allocate local variables
# 1 * 4 = 4 bytes.
addi $sp, $sp, -4

# Initialize locals.
addi $t0, $0, 1
sw $t0, -4($fp)  # result
```

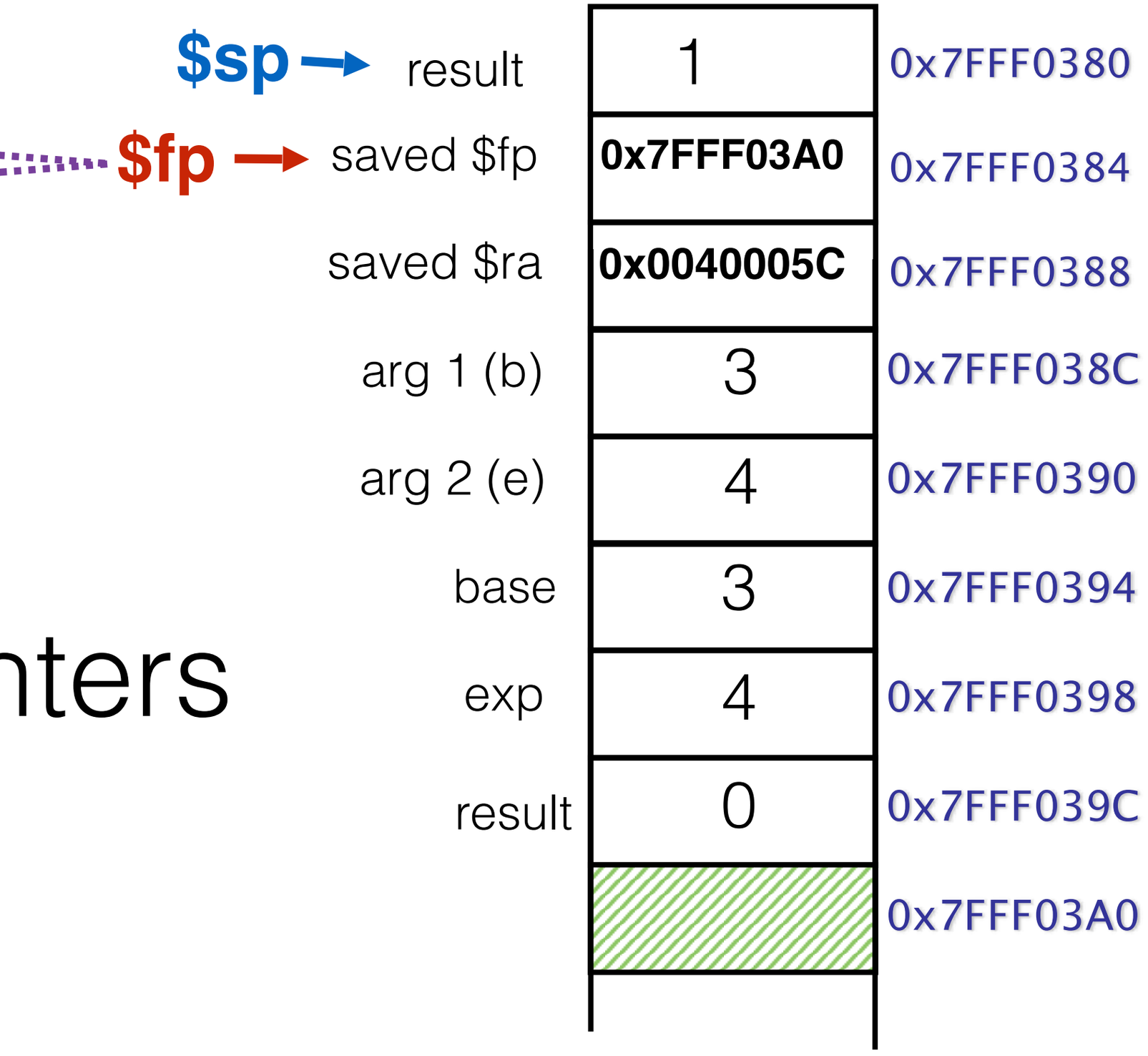
Step 7:
allocate local variables

in this function,
one local variable **result**



Frame Pointers

$fp always points to an old **saved copy of $fp**

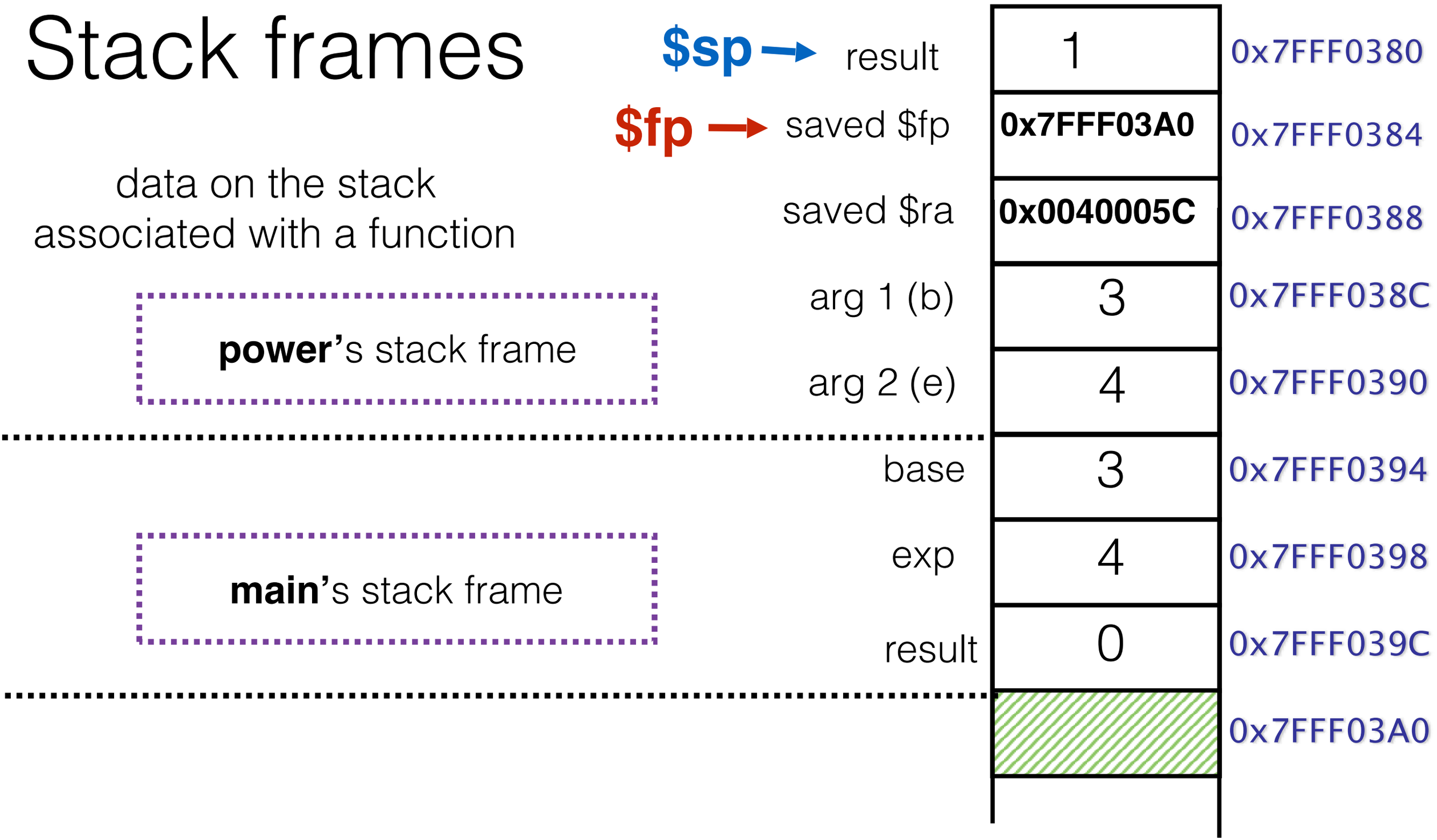


Stack frames

data on the stack
associated with a function

power's stack frame

main's stack frame

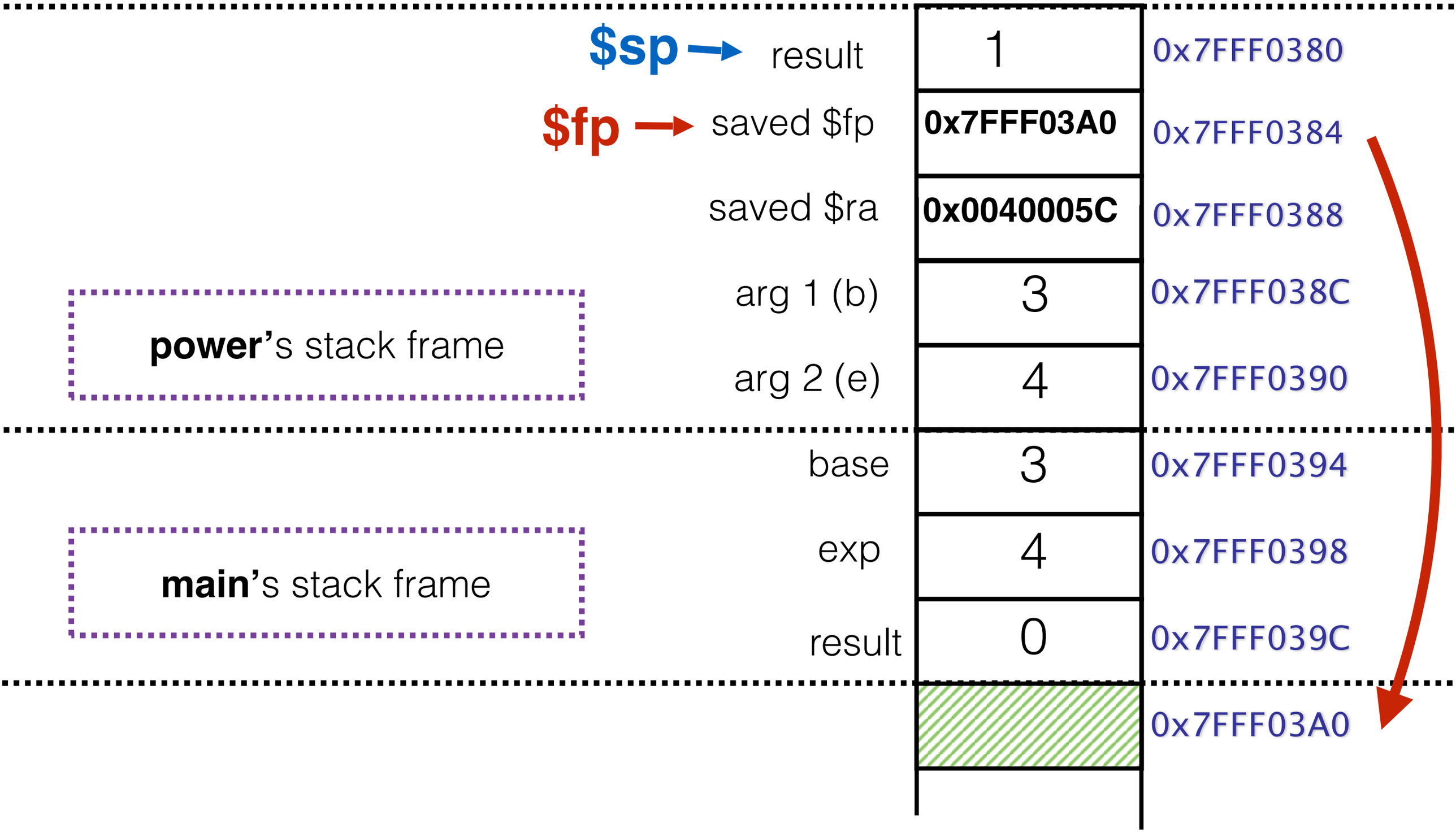


Function returning

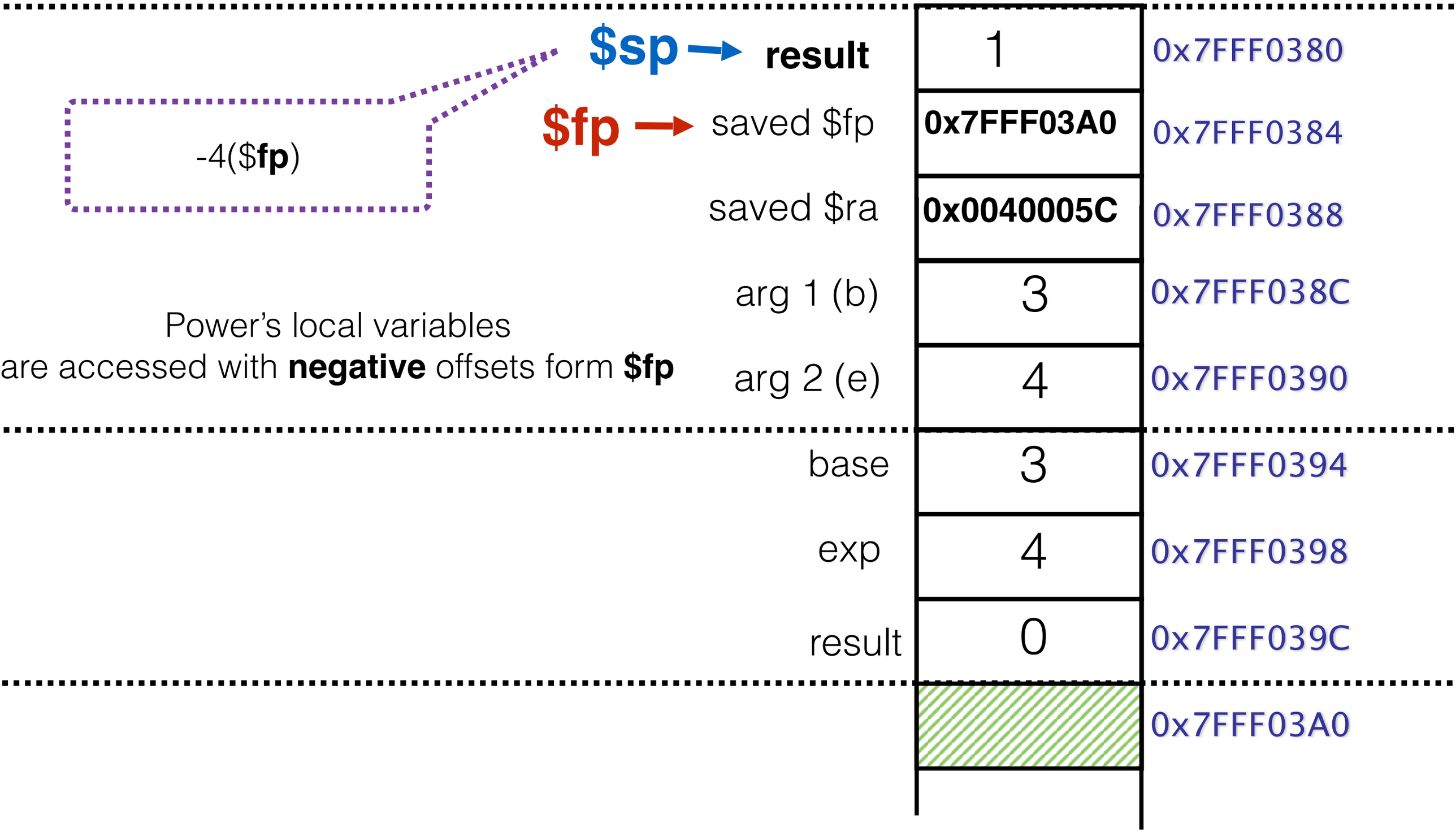
power.py

```
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result

main()
```

Stack frames

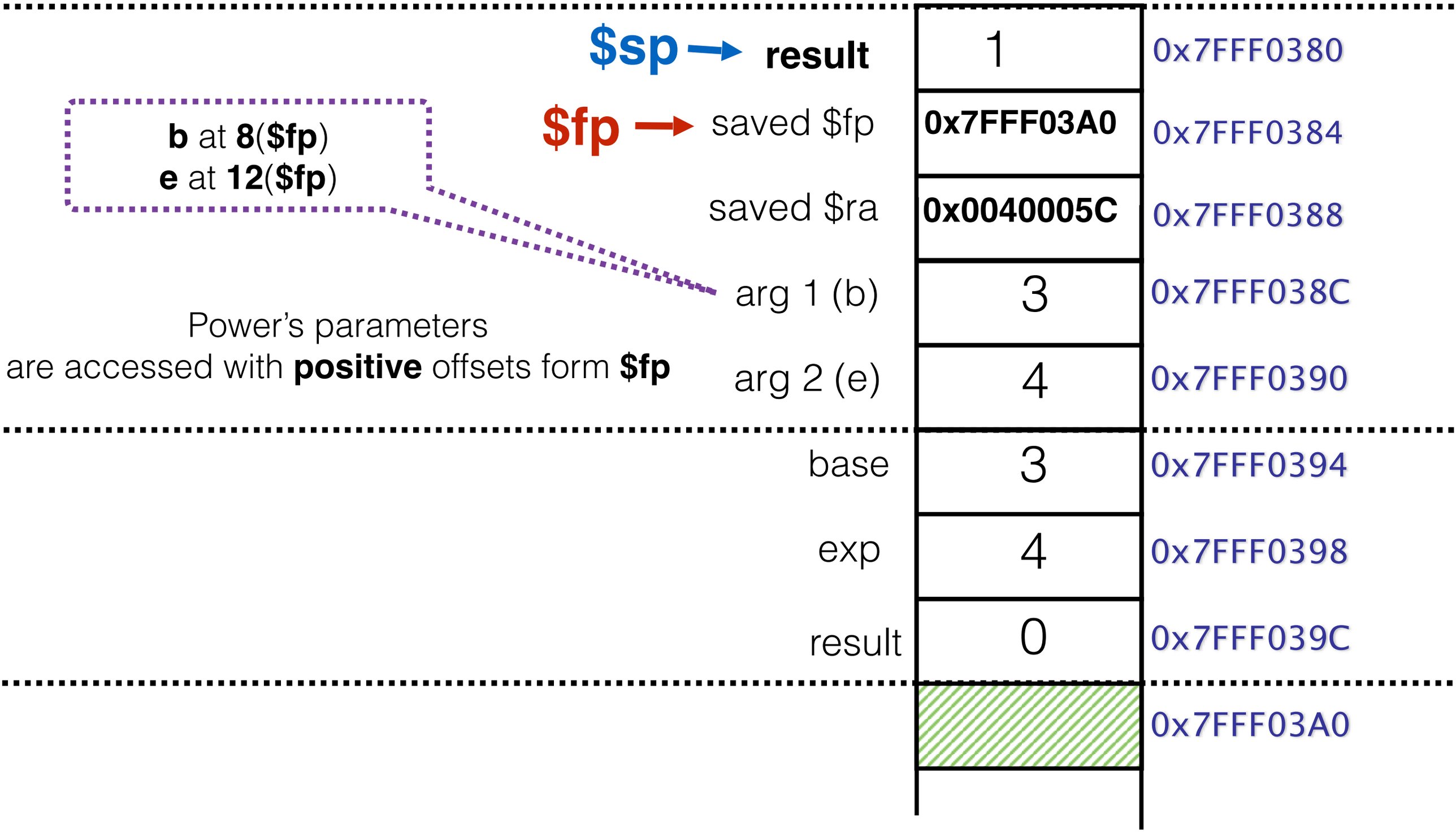


Local variables



Power's local variables
are accessed with **negative** offsets form **$fp**

Function parameters



b at **8**(**$fp**)
e at **12**(**$fp**)

Power's parameters
are accessed with **positive** offsets form **$fp**

```python
def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result
```

```
power:  # Save $ra and $fp
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $fp, 0($sp)


# Allocate local variables
# 1 * 4 = 4 bytes.
addi $sp, $sp, -4

# Initialize locals.
addi $t0, $0, 1
sw $t0, -4($fp)  # result
```

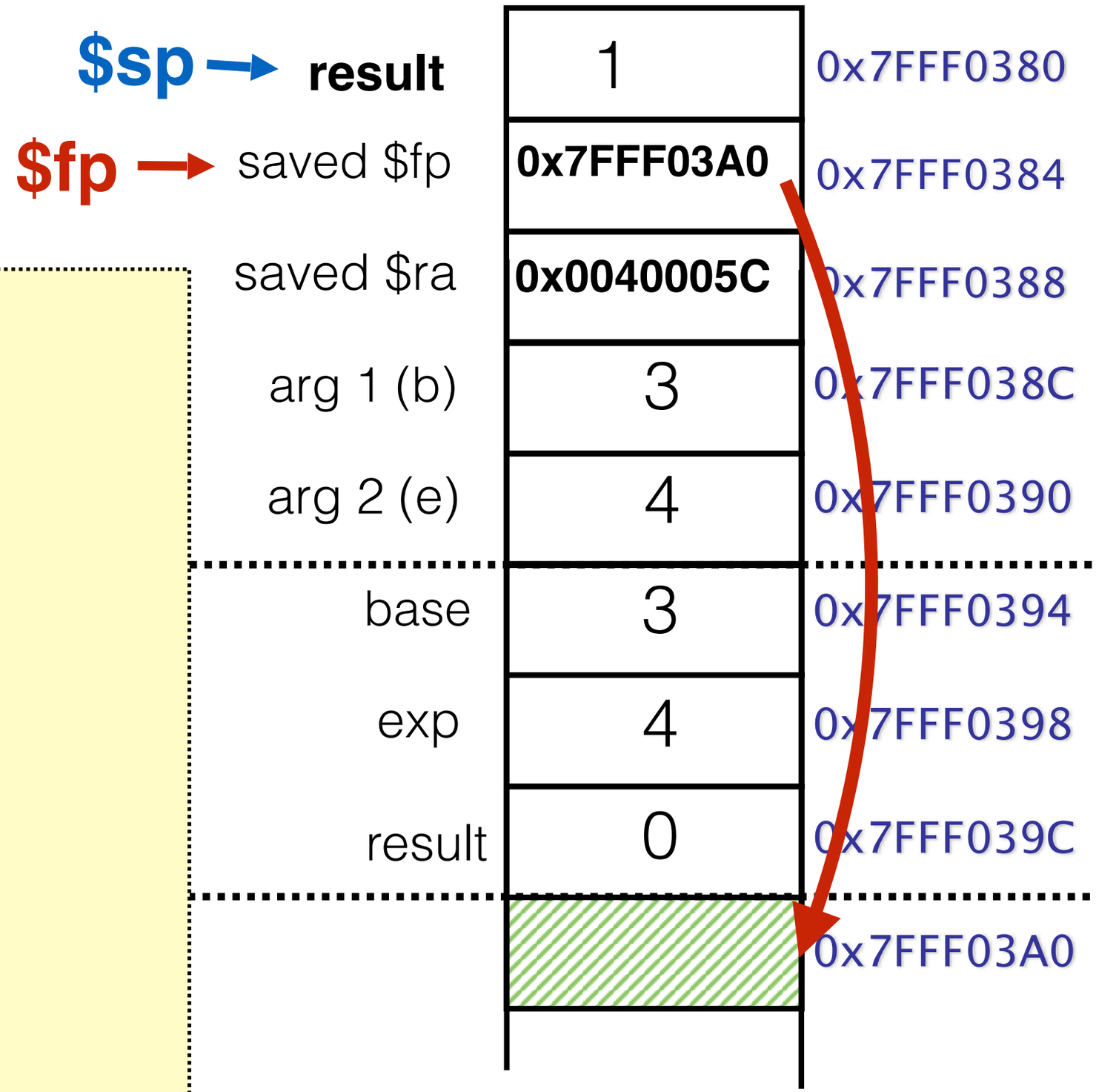
What next?

Example: callee

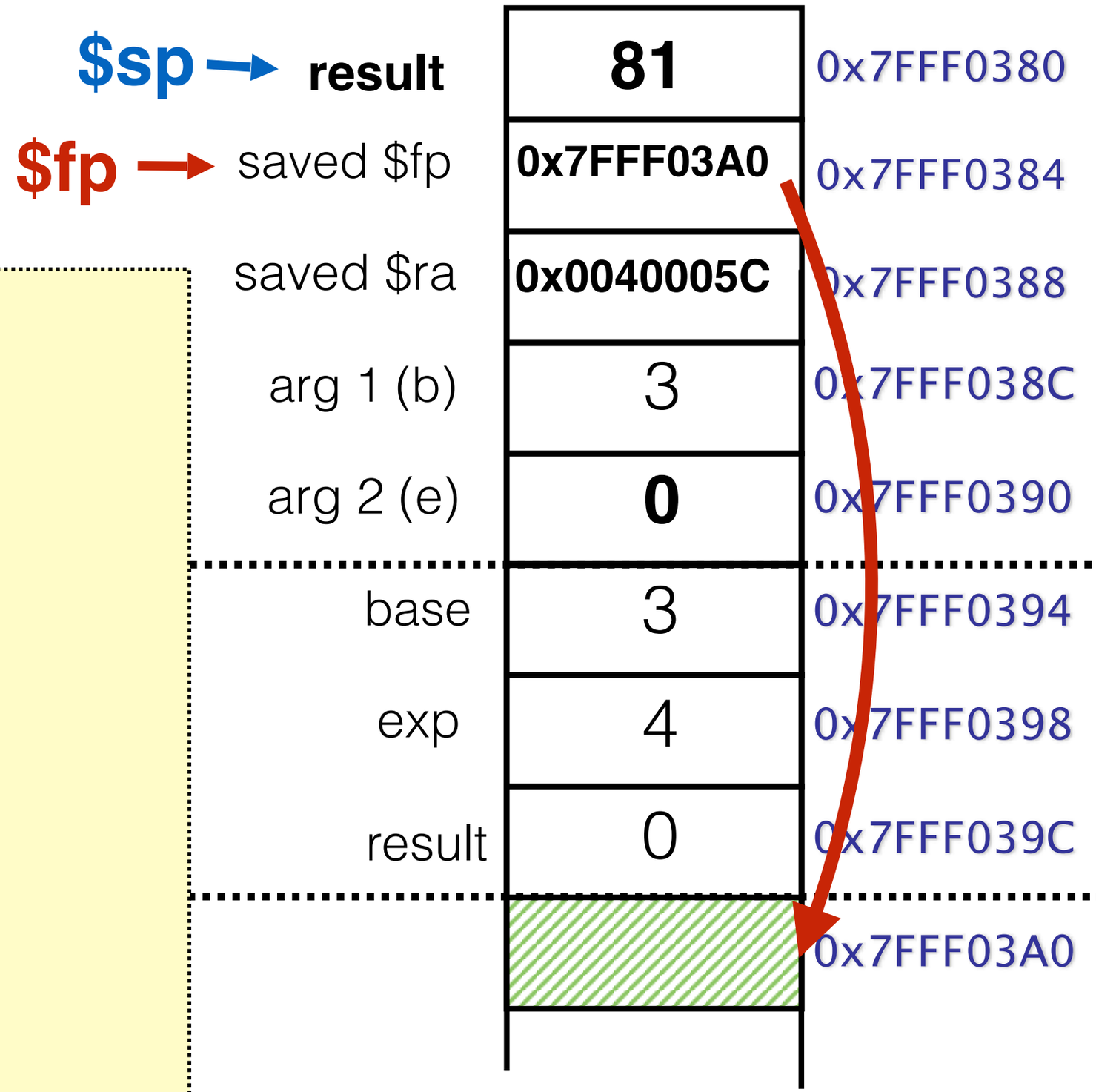
```
def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result
```

```
# Loop
loop:   # Stop if !(e > 0)
        lw $t0, 12($fp)  # e
        slt $t0, $0, $t0
        beq $t0, 0, end

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

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

        # Repeat loop.
        j loop

end:    # Now ready to return.
# Continued ...
```

```
# Loop
loop:   # Stop if !(e > 0)
        lw $t0, 12($fp)  # e
        slt $t0, $0, $t0
        beq $t0, 0, end

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

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

        # Repeat loop.
        j loop

end:    # Now ready to return.
# Continued ...
```



```
# Loop
loop:   # Stop if !(e > 0)
        lw $t0, 12($fp)  # e
        slt $t0, $0, $t0
        beq $t0, 0, end

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

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

        # Repeat loop.
        j loop

end:    # Now ready to return.
# Continued ...
```



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**

- **Return first**, in **$v0** (if necessary)… then reverse convention

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

Callee

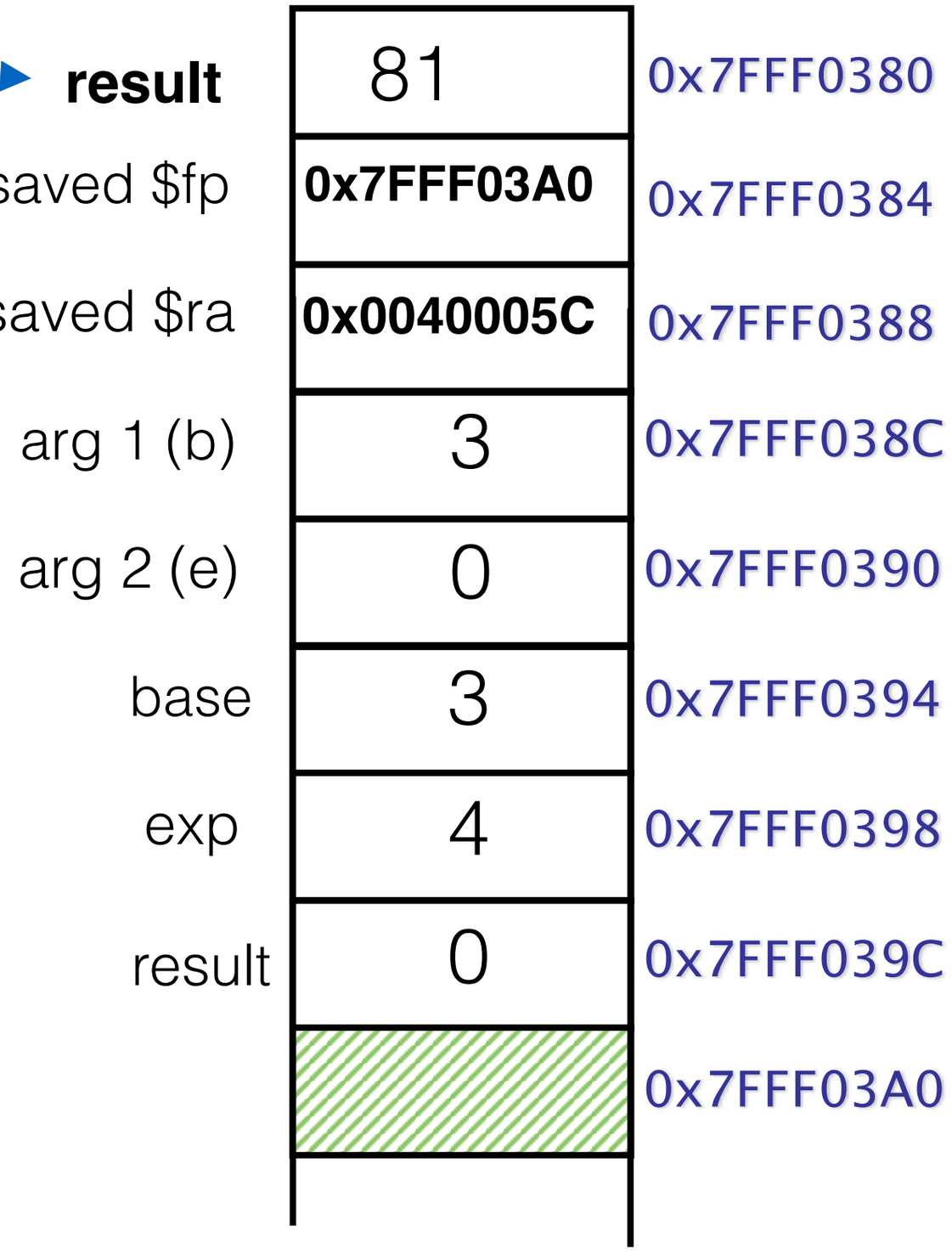
Returning

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

$v0 = 81

Step 1: Put return value in register **$v0**

(no visible effect on the stack)



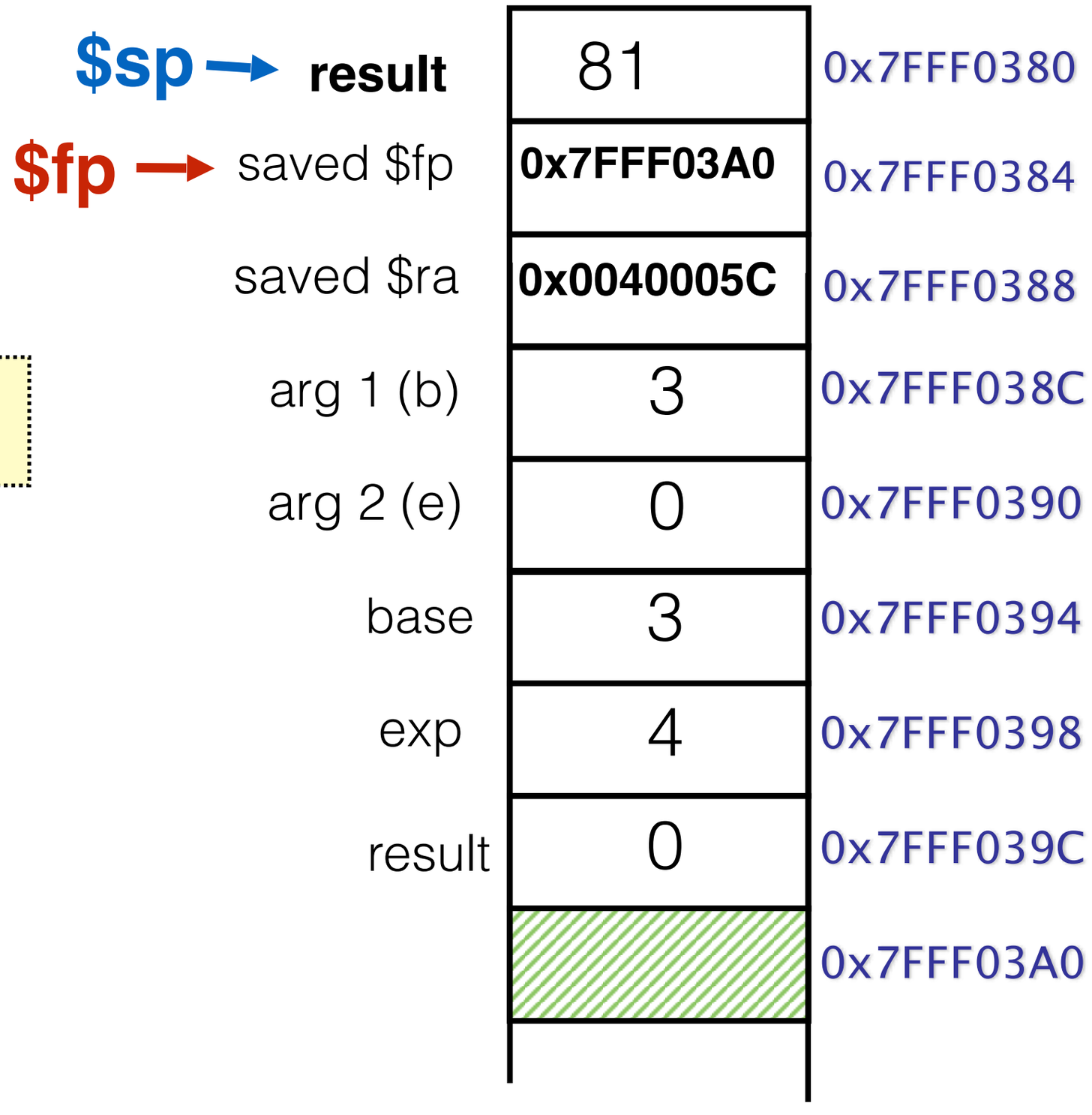
Callee

Returning

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

Step 2: Deallocate local variables by popping allocated space off stack

(one local variable to be deleted)



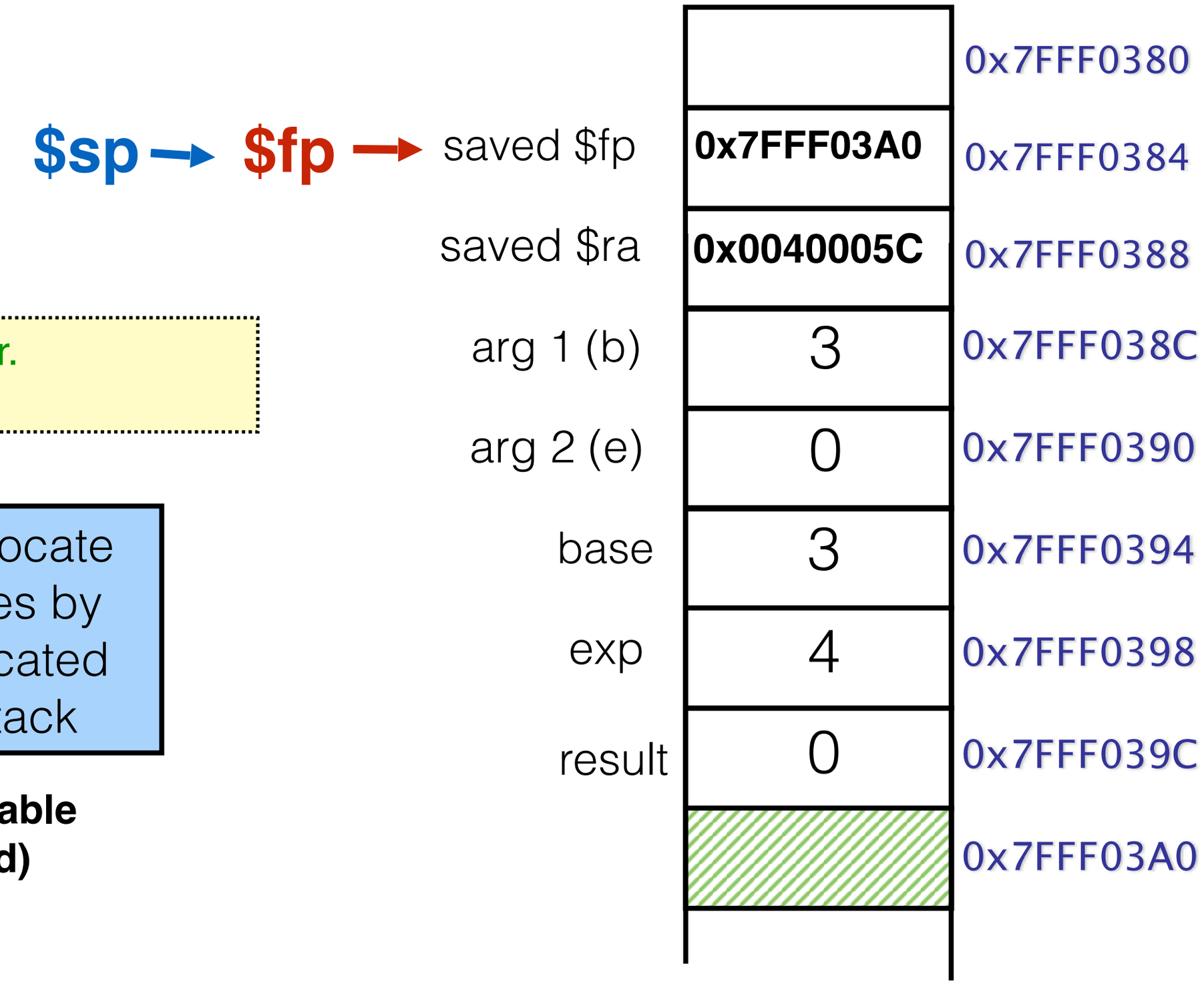
Callee

Returning

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

Step 2: Deallocate local variables by popping allocated space off stack

(one local variable to be deleted)



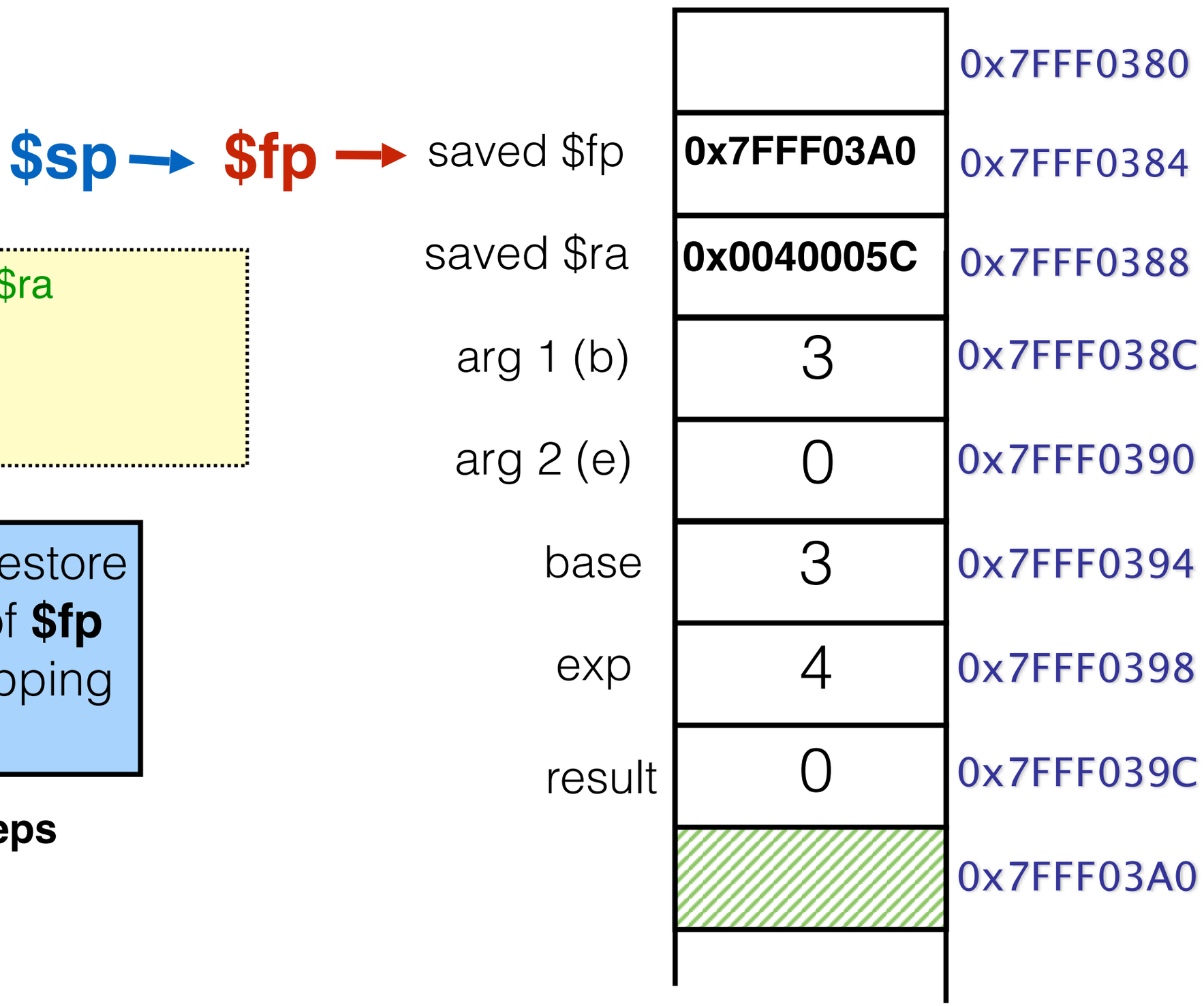
Callee

Returning

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

Step 3 and 4: Restore saved values of **$fp** and **$ra** by popping of stack

(can do both steps at once)



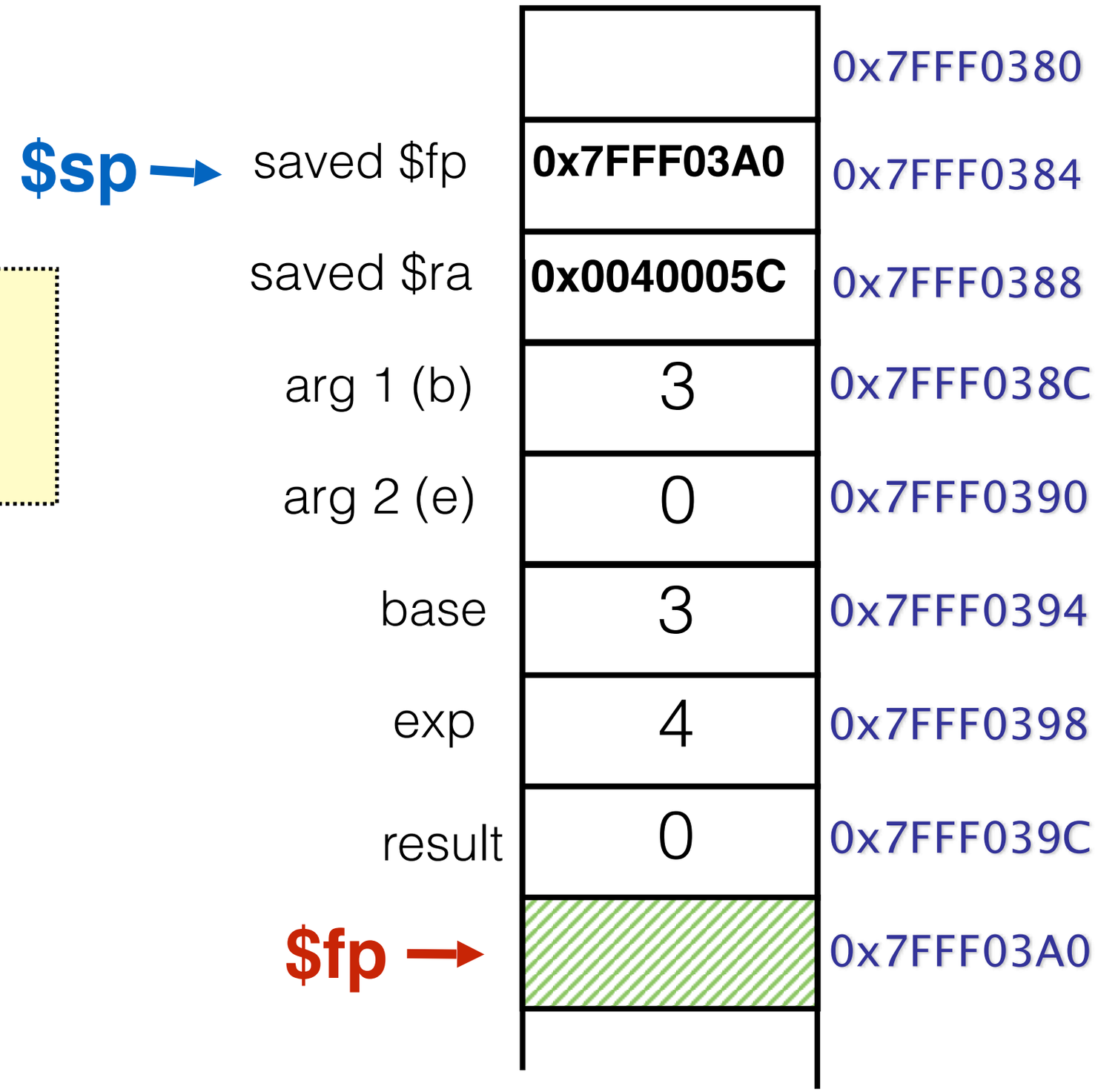
Callee

Returning

$fp = 0x7FFF03A0

$ra = 0x0040005C

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



Callee

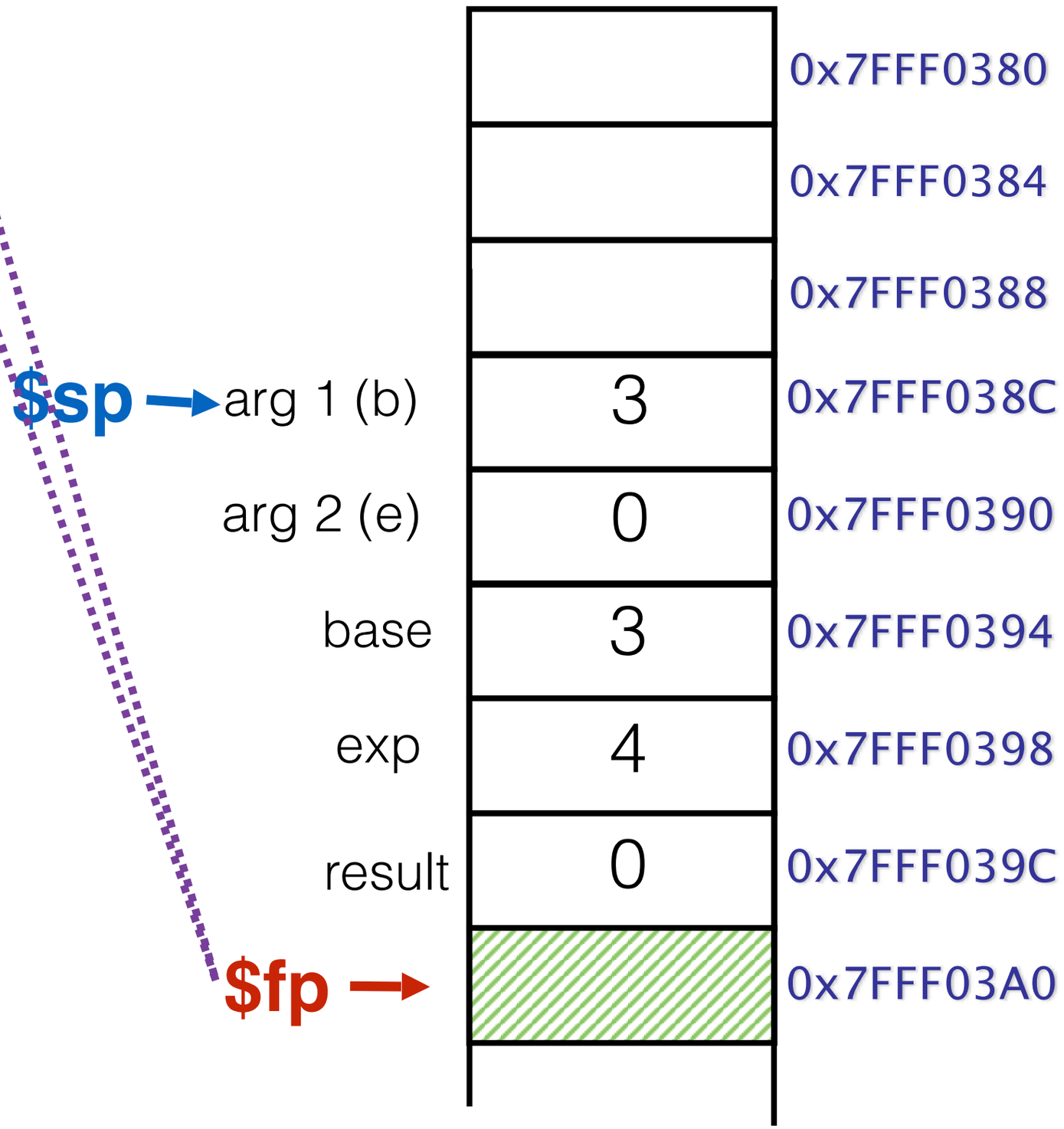
Returning

$fp points back to main's stack frame

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

Step 3 and 4: Restore saved values of **$fp** and **$ra** by popping of stack

(can do both steps at once)



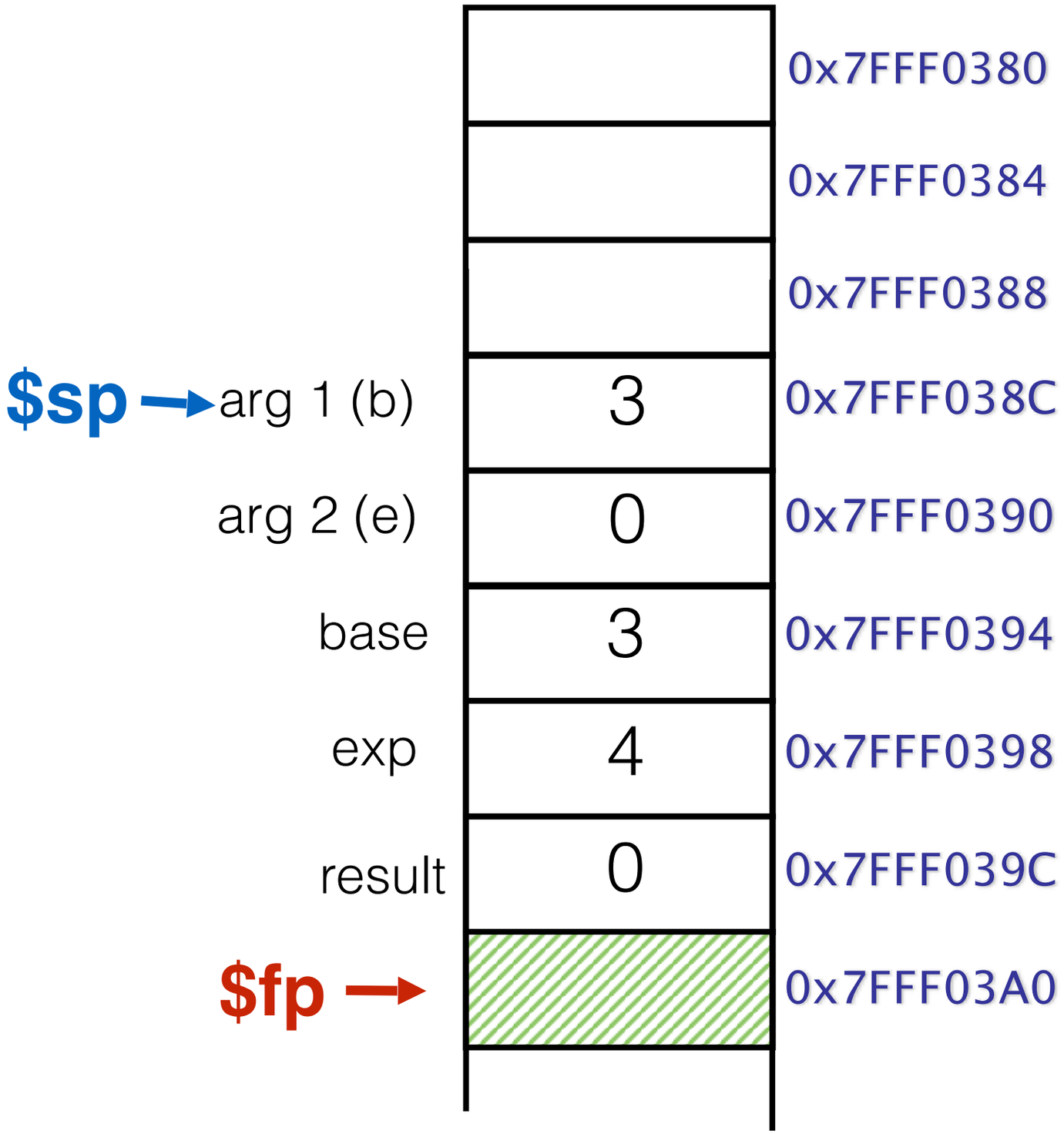
Callee

Returning

```
# Return to caller.
jr $ra
```

Step 5: Return by executing **jr $ra**

(nothing visible on the stack)



power.py

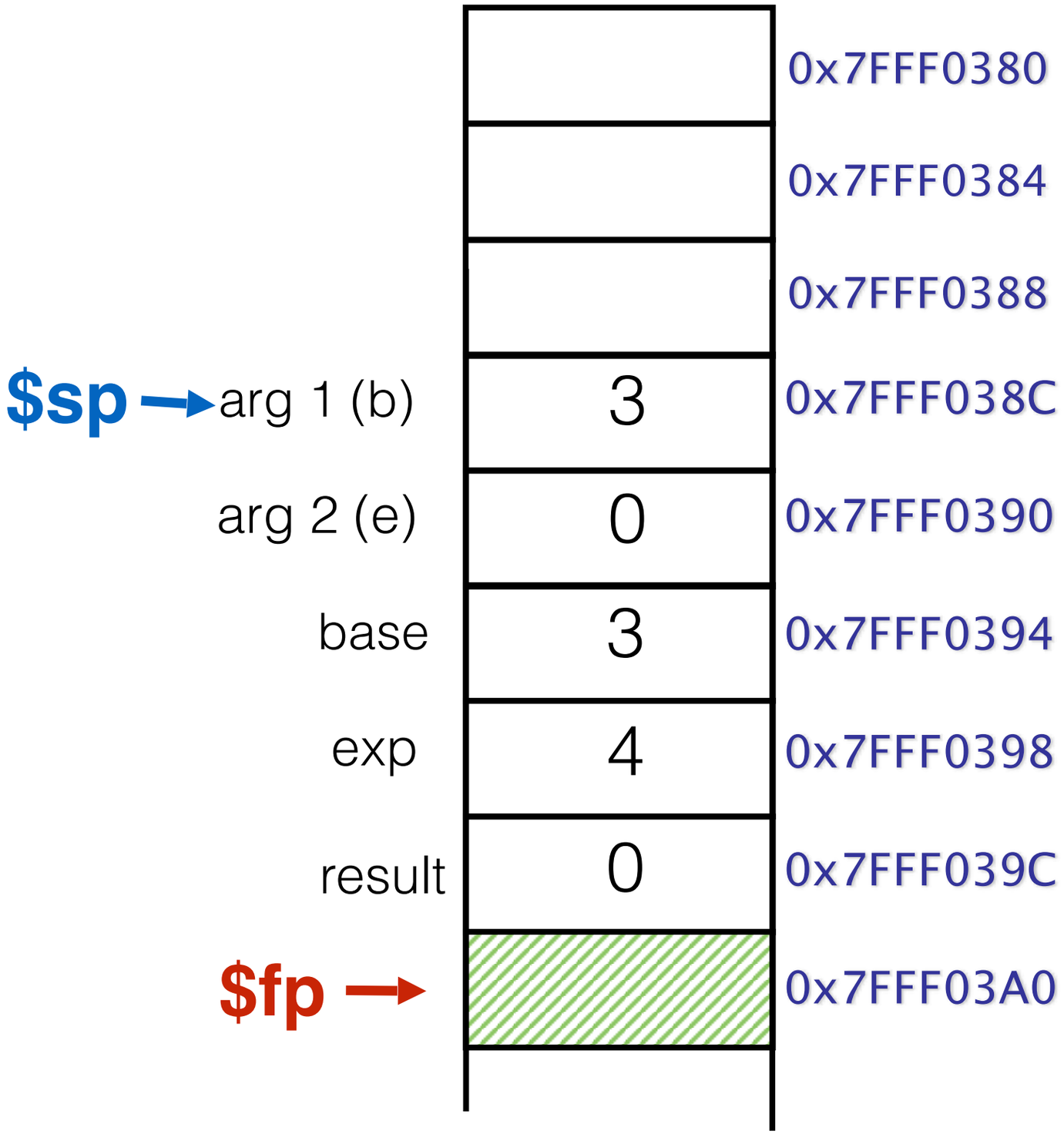
```
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())
    result = power(base, exp)
    print(result)


def power(b, e):
    result = 1

    while e > 0:
        result *= b
        e -= 1
    return result

main()
```

Caller

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

Step 6: Clear function arguments by popping them off the stack

(nothing visible on the stack)



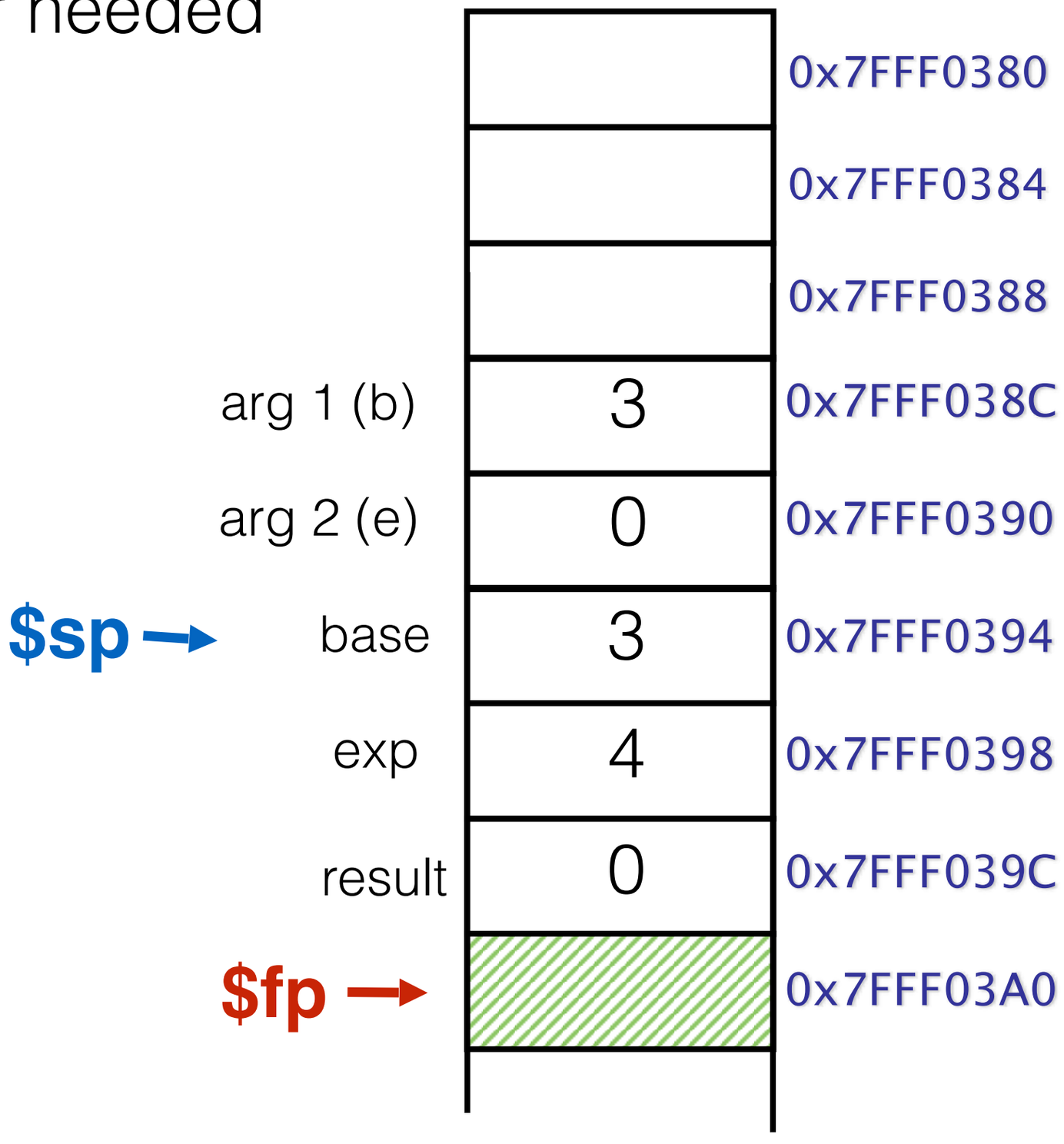
Caller

b and **e** are no longer needed

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

Step 6: Clear function arguments by popping them off the stack

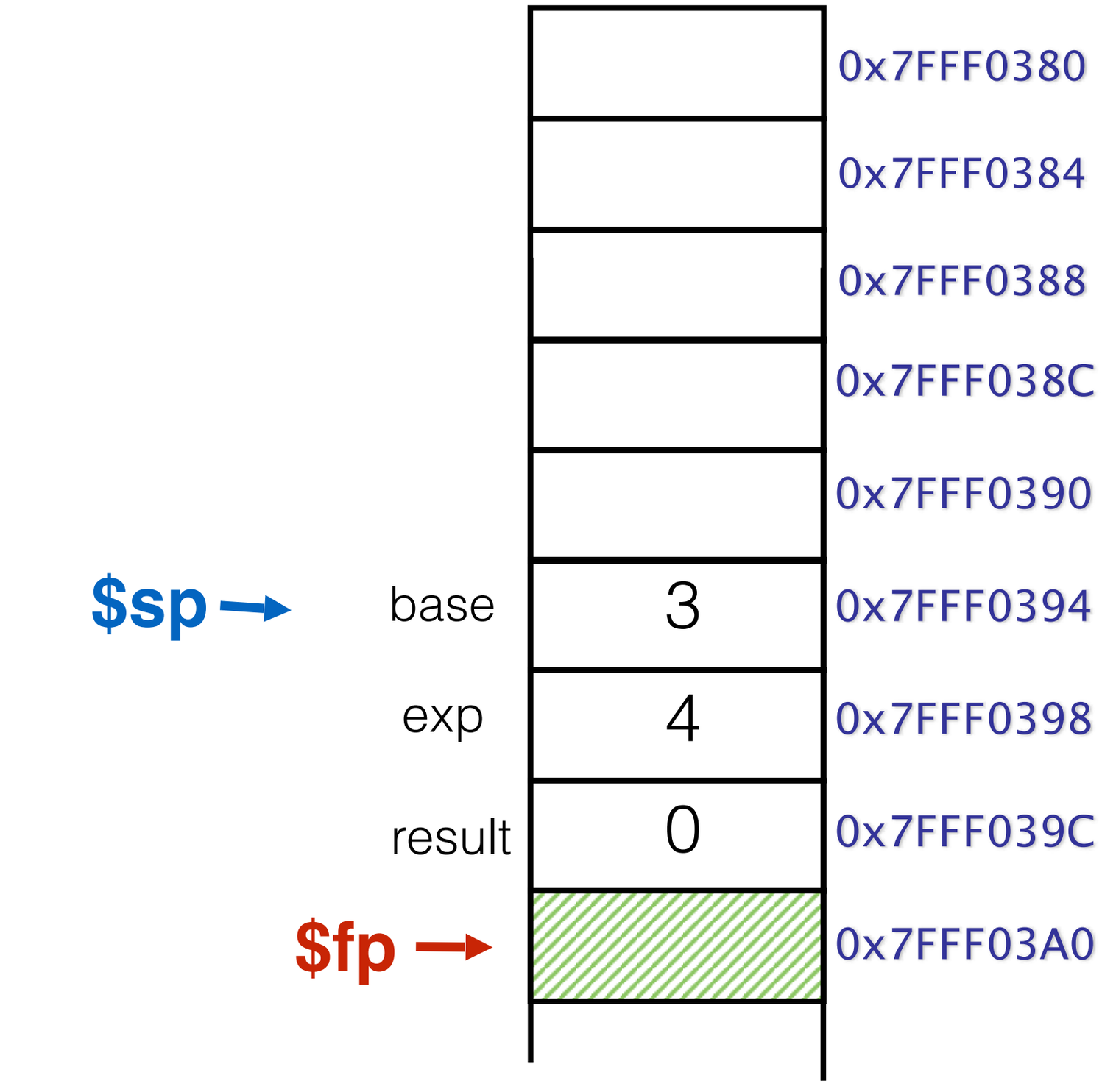
(nothing visible on the stack)



Caller

Step 7: Restore any saved temporary registers by popping value of the stack

(we did not have any)



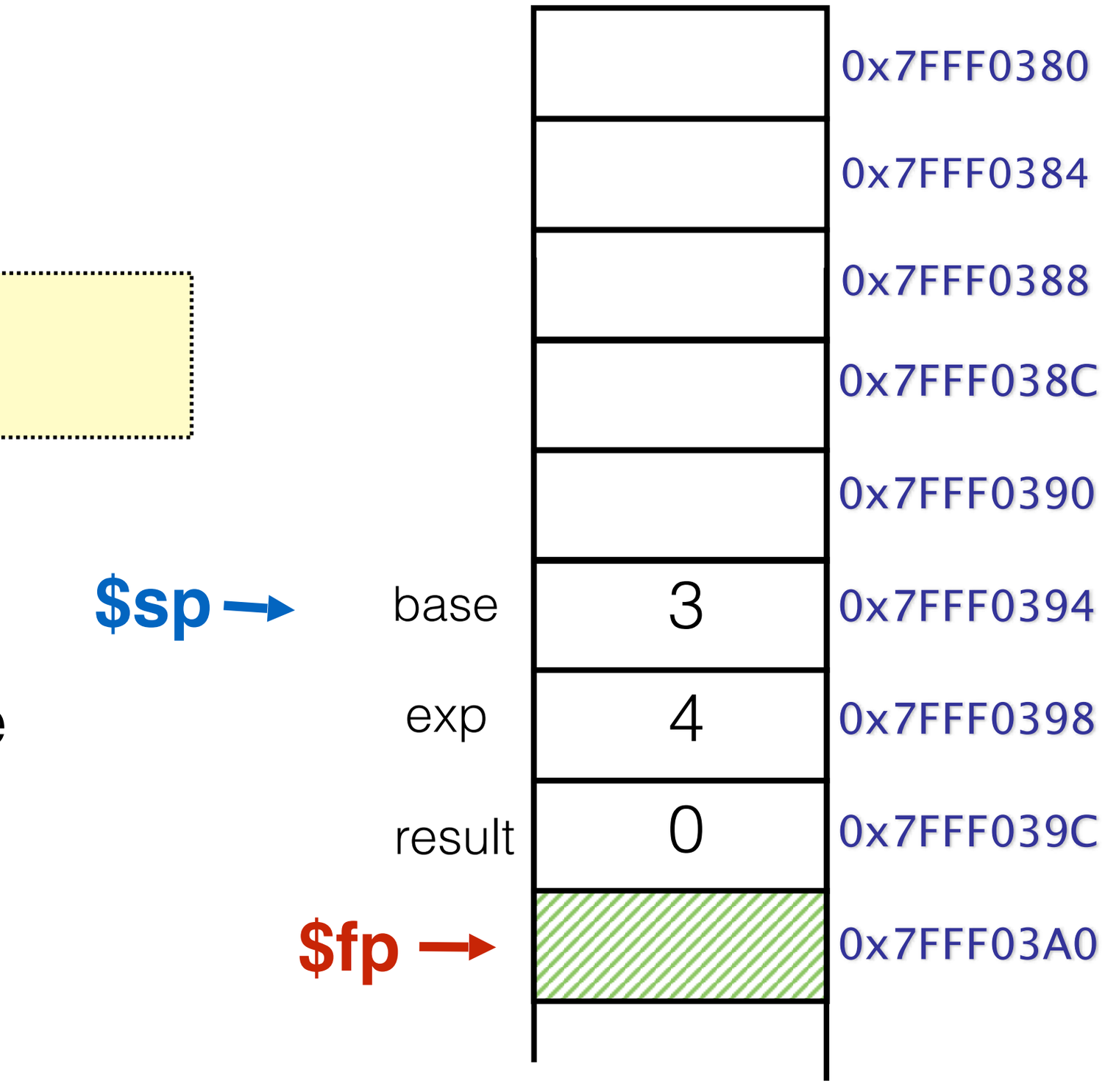
Caller

```
# Store return value
# in result.
sw $v0, -4($fp)  # result
```

$v0 = 81

Use returned value found in **$v0**

store returned value in a variable



Caller

```
def main():
    base = 0
    exp = 0
    result = 0

    base = int(input())
    exp = int(input())

    result = power(base, exp)
    print(result)
```

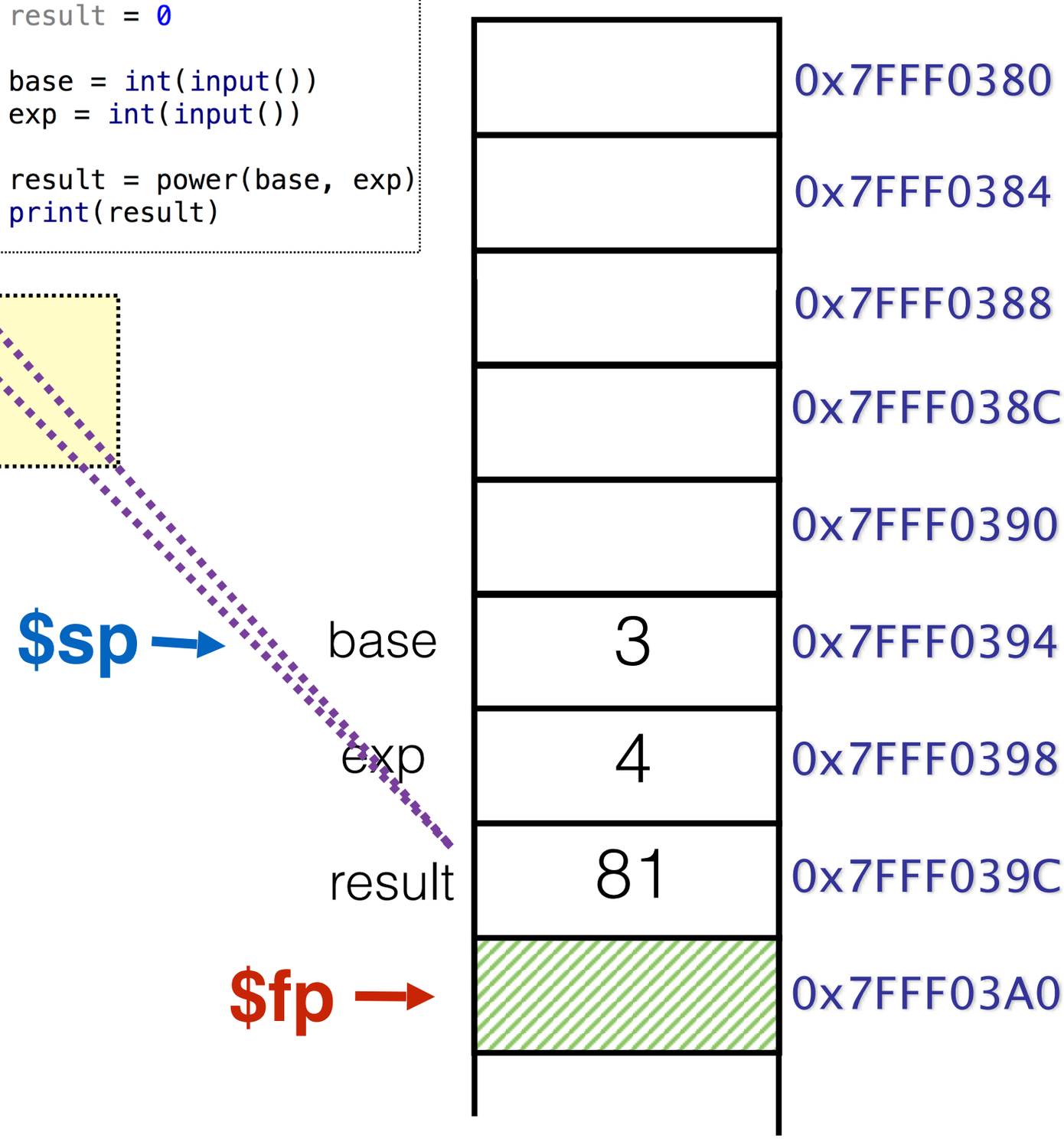
main stores return value
into local variable return

```
# Store return value
# in result.
sw $v0, -4($fp)  # result
```

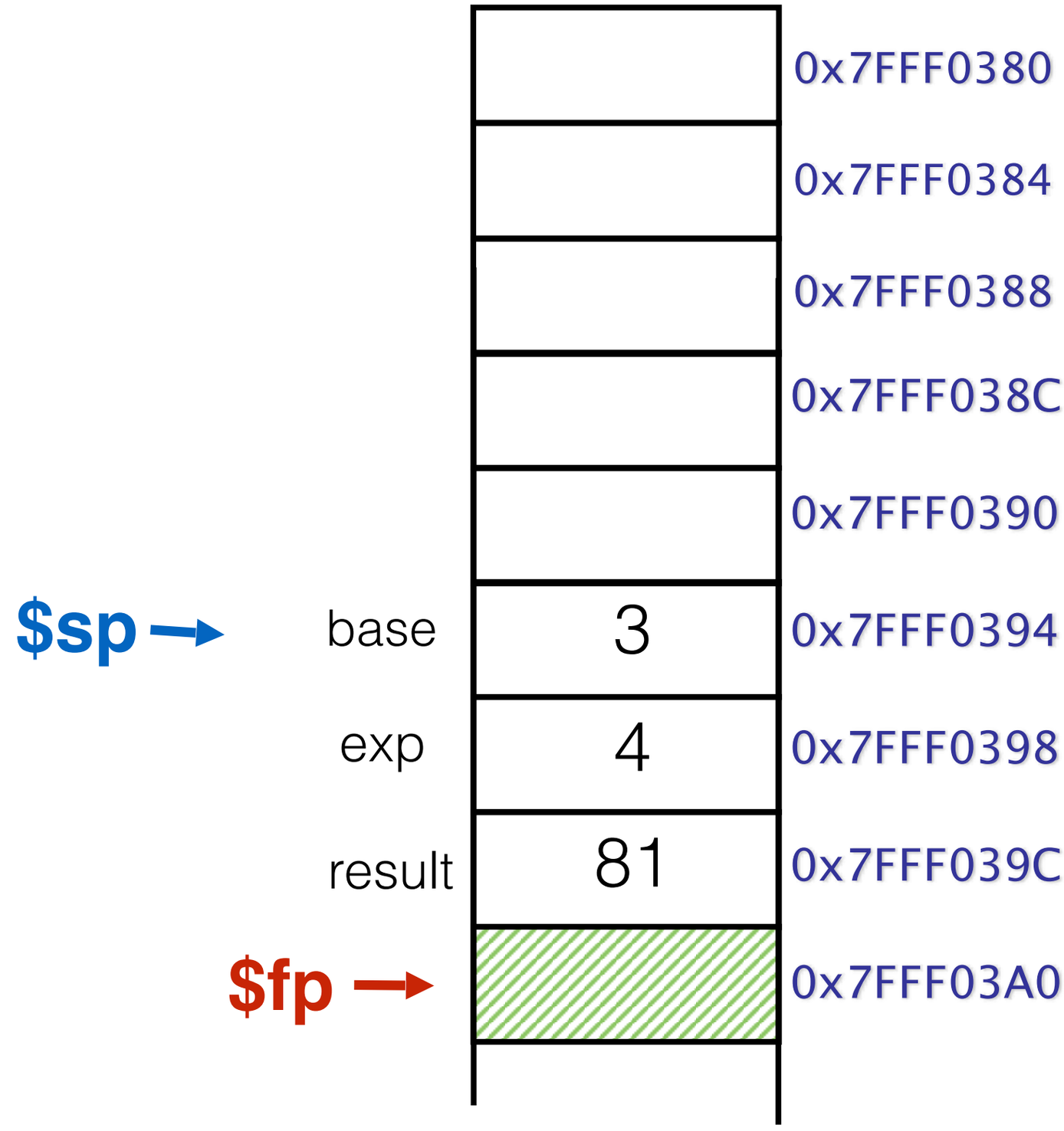
$v0 = 81

Use returned value
found in **$v0**

**store returned value
in a variable**



After call stack is in its original state



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**

Going further

Official MIPS stack frame convention

- Doesn’t use $fp at all!
- Slightly more efficient than FIT1008 convention
- Can be generated by compilers
- Hard for humans to write/understand

Summary

- Accessing function parameters
- Returning from functions
- Function calling/returning convention