Lecture 5

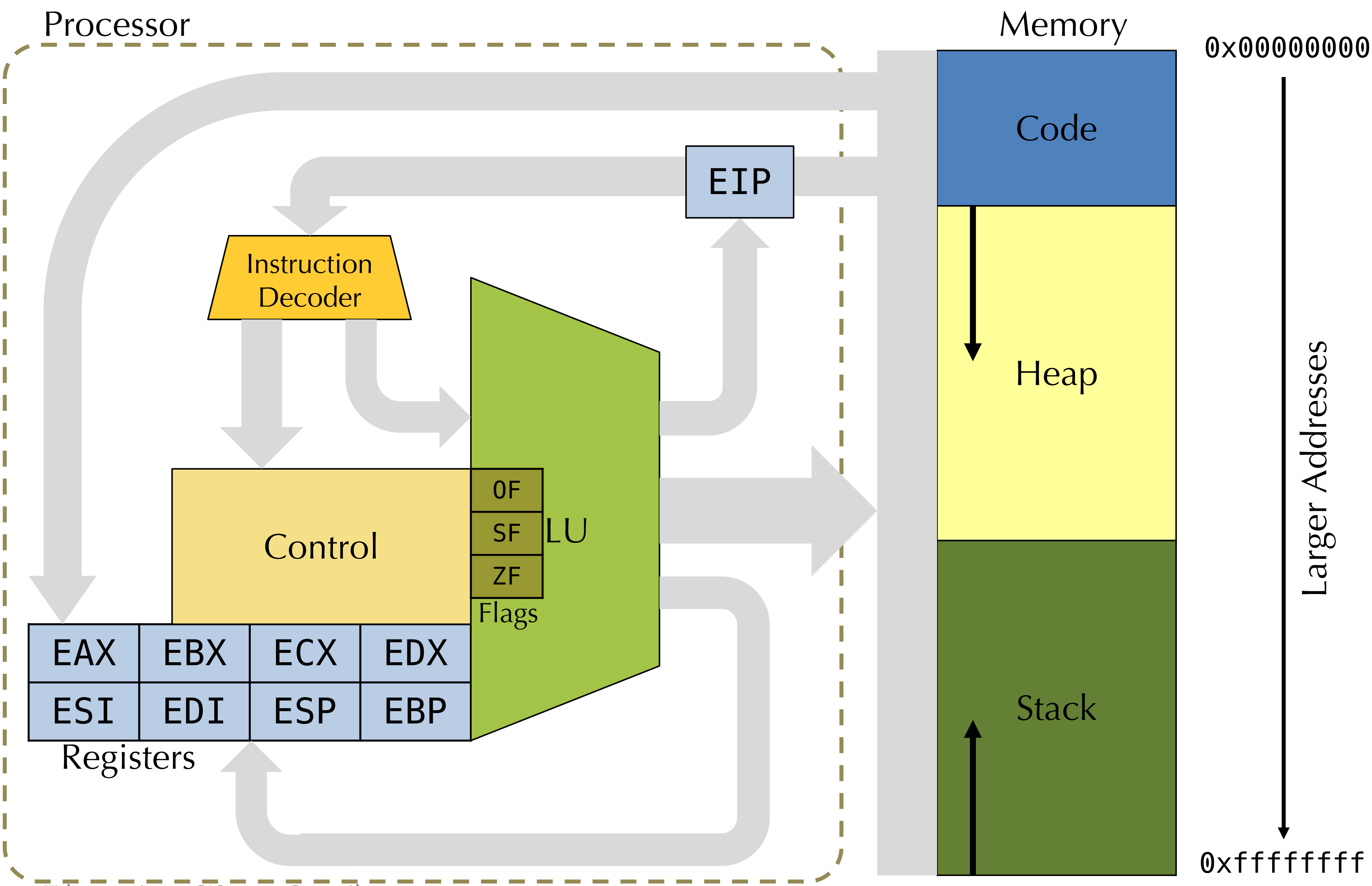
CIS 341: COMPILERS

CIS 341 Announcements

- HW2: X86lite
  - Available on the course web pages soon.
    (look for Piazza announcement).
  - Due: Weds. Feb. 9th at midnight
  - Pair-programming project

  - NOTE: much more difficult than hw1, so please start early!

X86 Schematic



See: runtime.c

DEMO: HANDCODING X86LITE

Compiling, Linking, Running

- See the code in lec04.zip
- To use hand-coded X86:
  1. Compile main.ml (or something like it) to either native or bytecode
     `dune build`
  2. Run it, redirecting the output to some .s file, e.g.:
     `./main.exe > test.s`
  3. Use gcc to compile & link with runtime.c:
     `gcc -arch x86_64 -o test runtime.c test.s`
  4. You should be able to run the resulting exectuable:
     `./test`

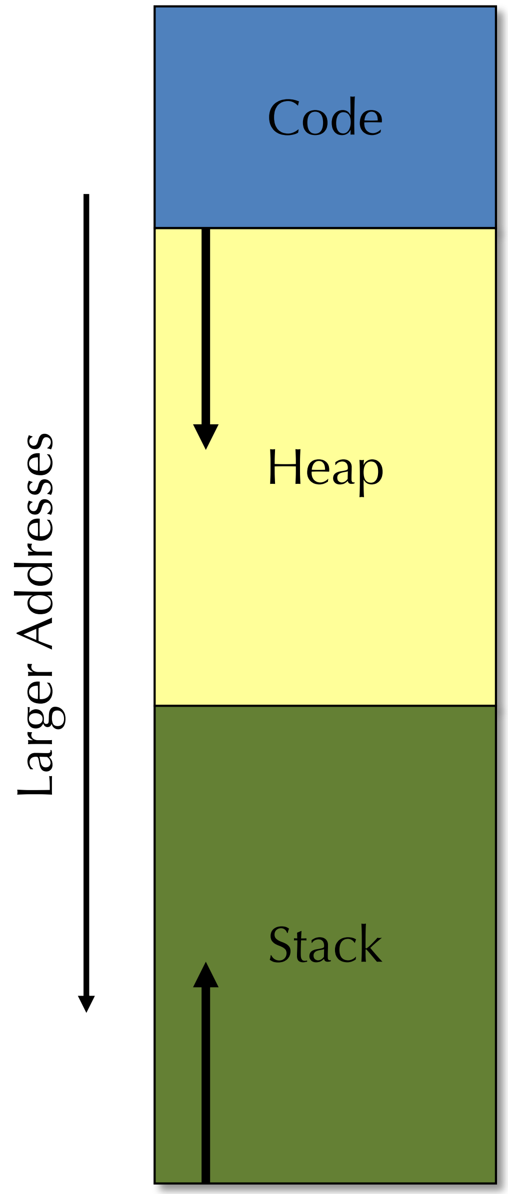
- Some compilers / architectures need "`program`" rather than "`_program`" for the entry label.
- If you want to debug in gdb:
  - Call gcc with the –g flag too

PROGRAMMING IN X86LITE

3 parts of the C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function

- In practice, most languages use this model.



Local/Temporary Variable Storage

- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- Processors provide two options
  - Registers: fast, small size (64 bits), very limited number
  - Memory: slow, very large amount of space (2 GB)
    - caching important

- In practice on X86:
  - Registers are limited (and have restrictions)
  - Divide memory into regions including the *stack* and the *heap*

Calling Conventions

- Specify the locations (*e.g.*, register or stack) of arguments passed to a function and returned by the function

```
int64_t g(int64_t a, int64_t b) {
   return a + b;
}

int64_t f(int64_t x) {
   int64_t ans = g(3,4) + x;
   return ans;
}
```

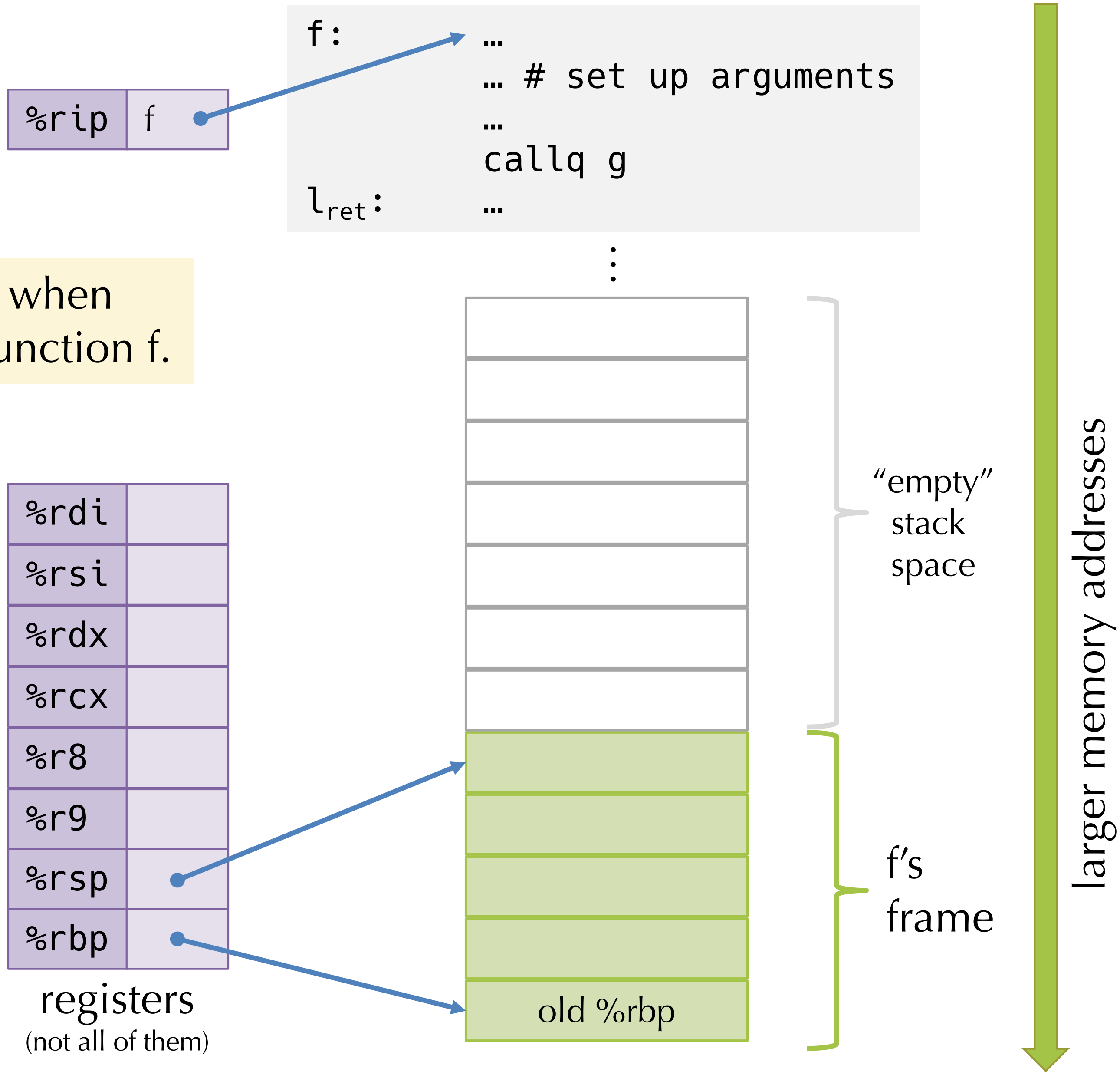
f is the *caller*

g is the *callee*

- Designate registers either:
  - Caller Save – *e.g.*, freely usable by the called code
  - Callee Save – *e.g.*, must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes variable arguments harder)

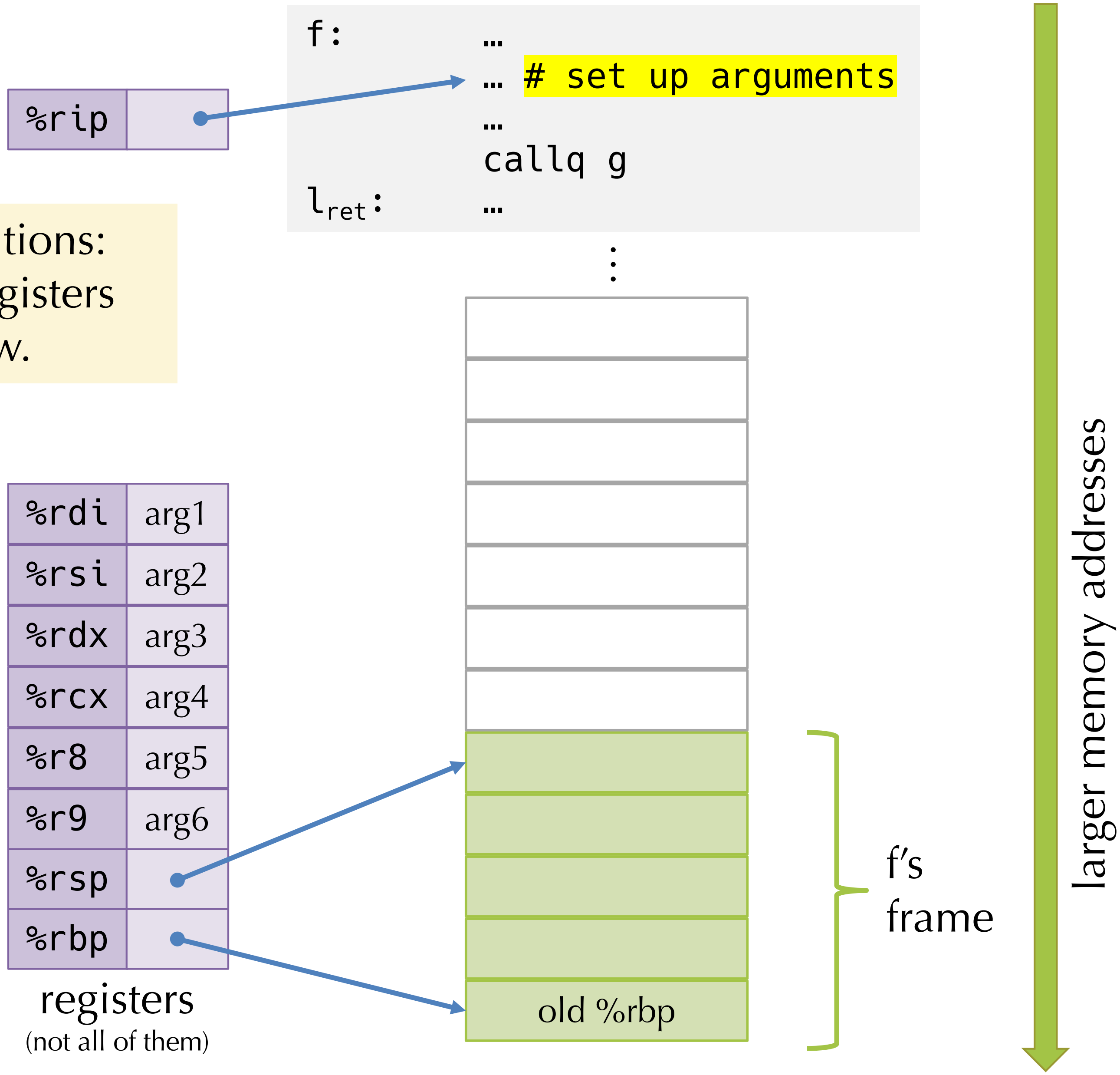
x64 Calling Conventions: Caller Protocol

Machine state when executing in function f.



x64 Calling Conventions: Caller Protocol

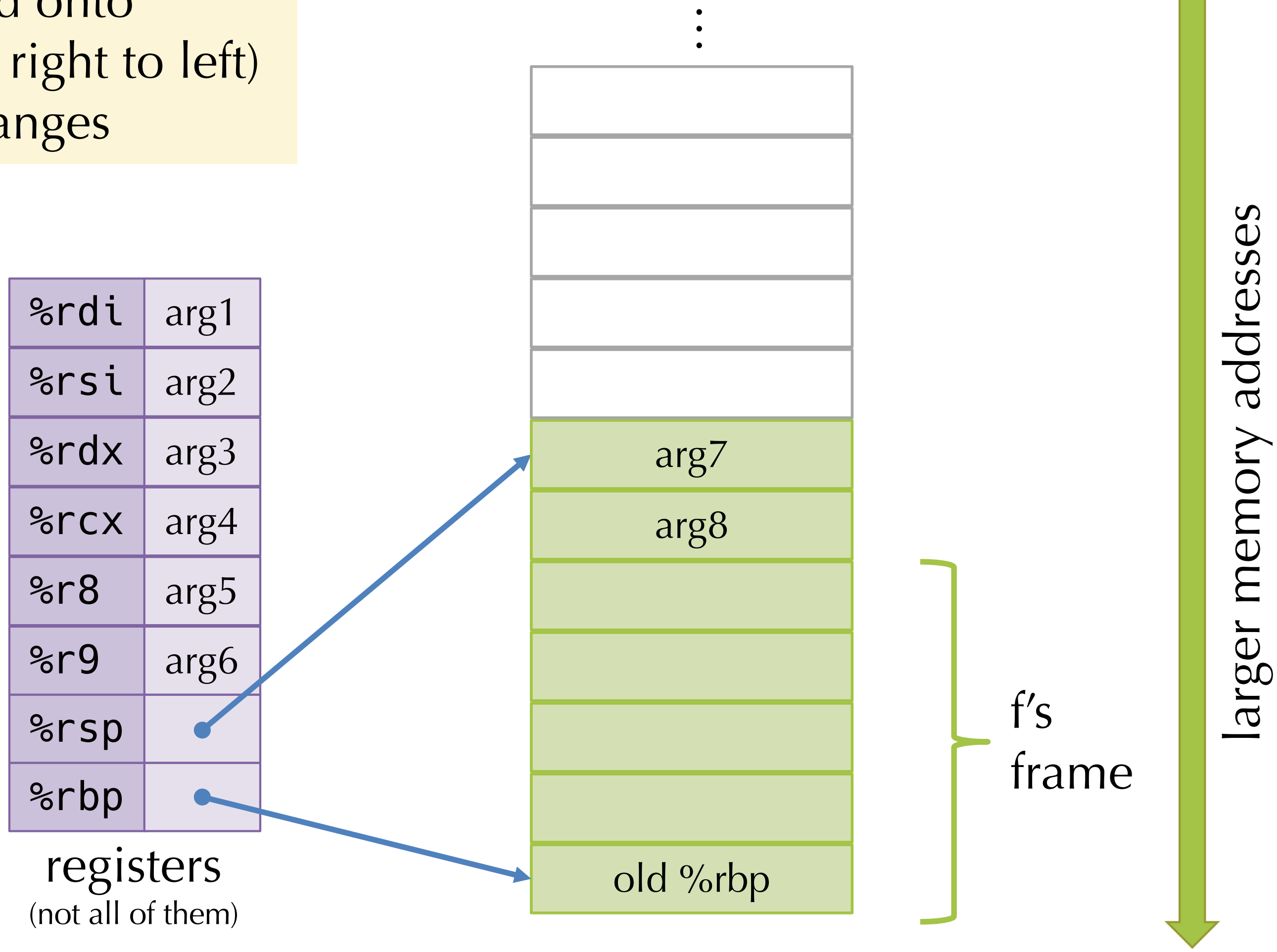
Calling conventions:
args 1…6 in registers
as shown below.



x64 Calling Conventions: Caller Protocol

```
f:        …
          … # set up arguments
          …
          callq g
l_ret:    …
```

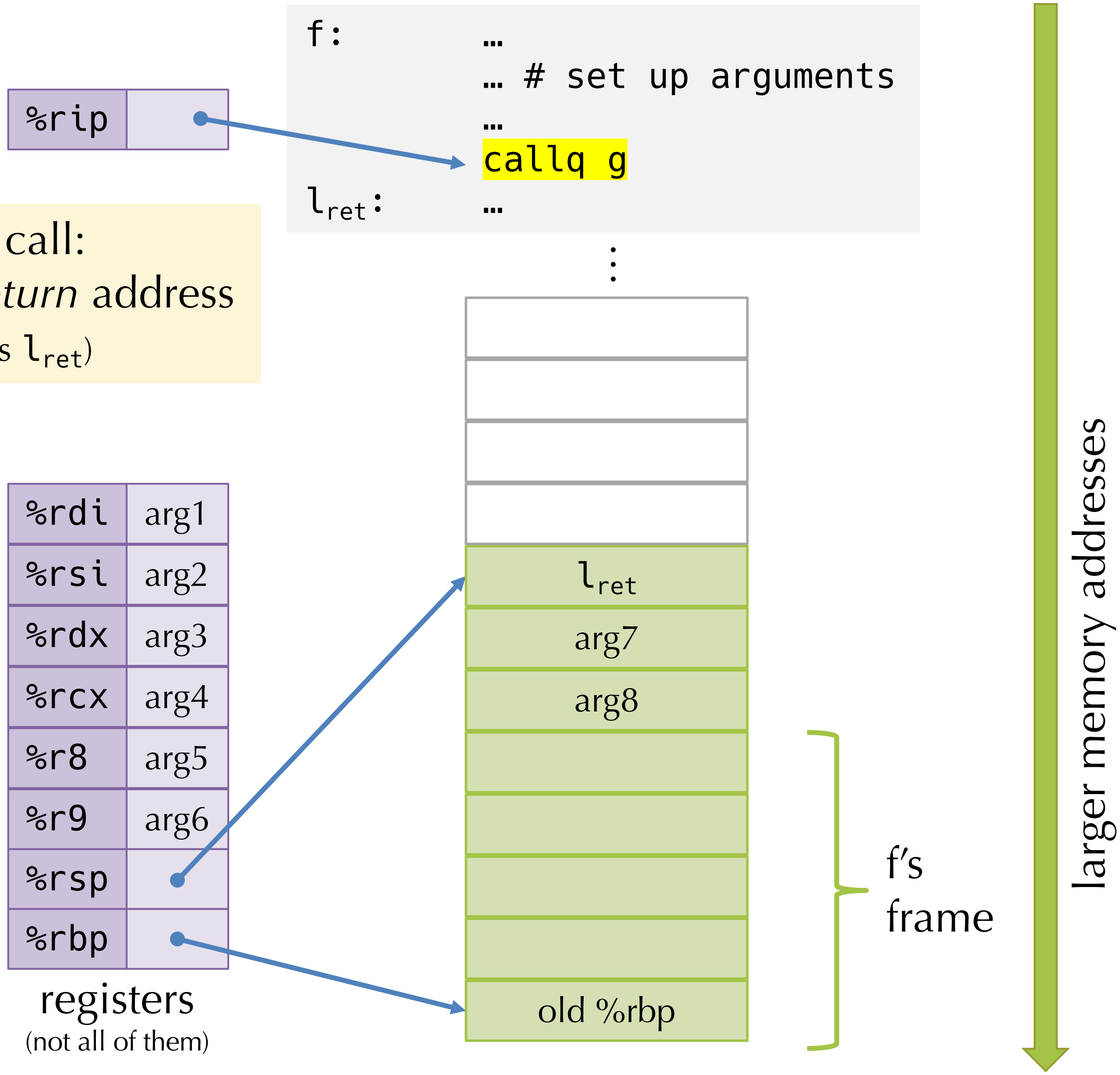
args > 6 pushed onto
the stack (from right to left)
Note: %rsp changes



call instruction

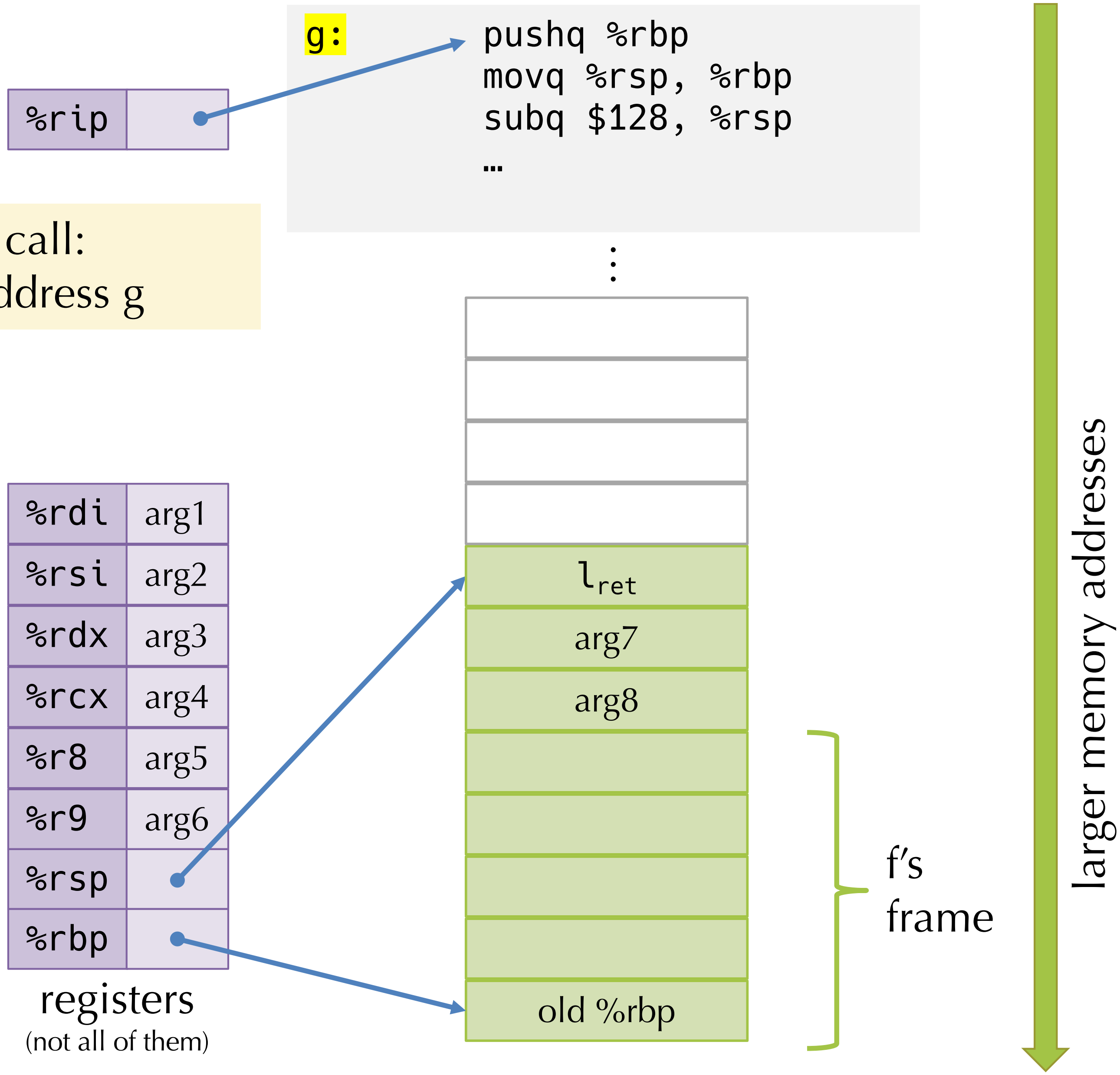
To execute the call:

1. push the *return* address (here shown as $\mathtt{l}_{ret}$)



call instruction

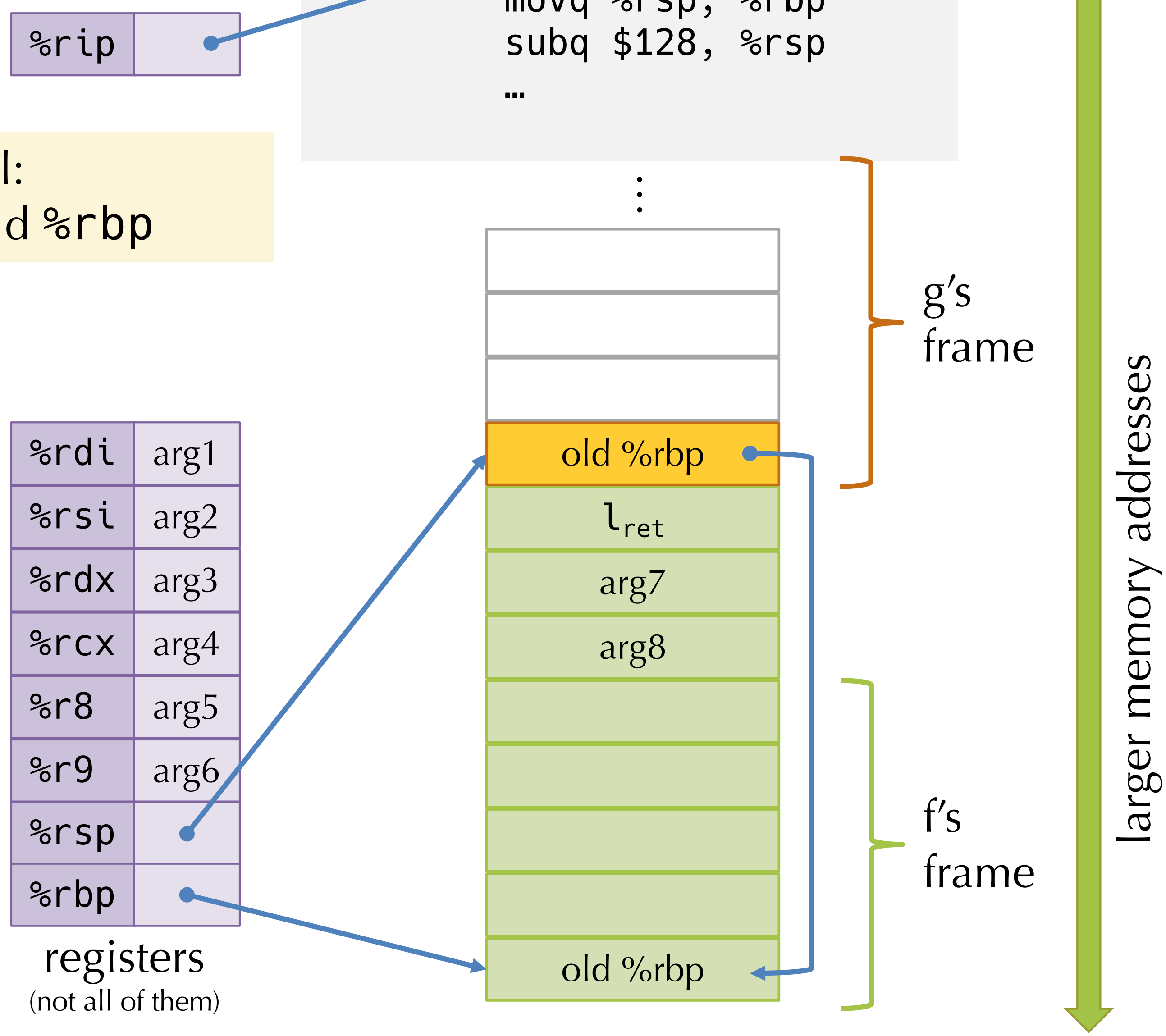
To execute the call:
2. set rip to address g



callee function prologue

```
g:    pushq %rbp
      movq %rsp, %rbp
      subq $128, %rsp
      …
```

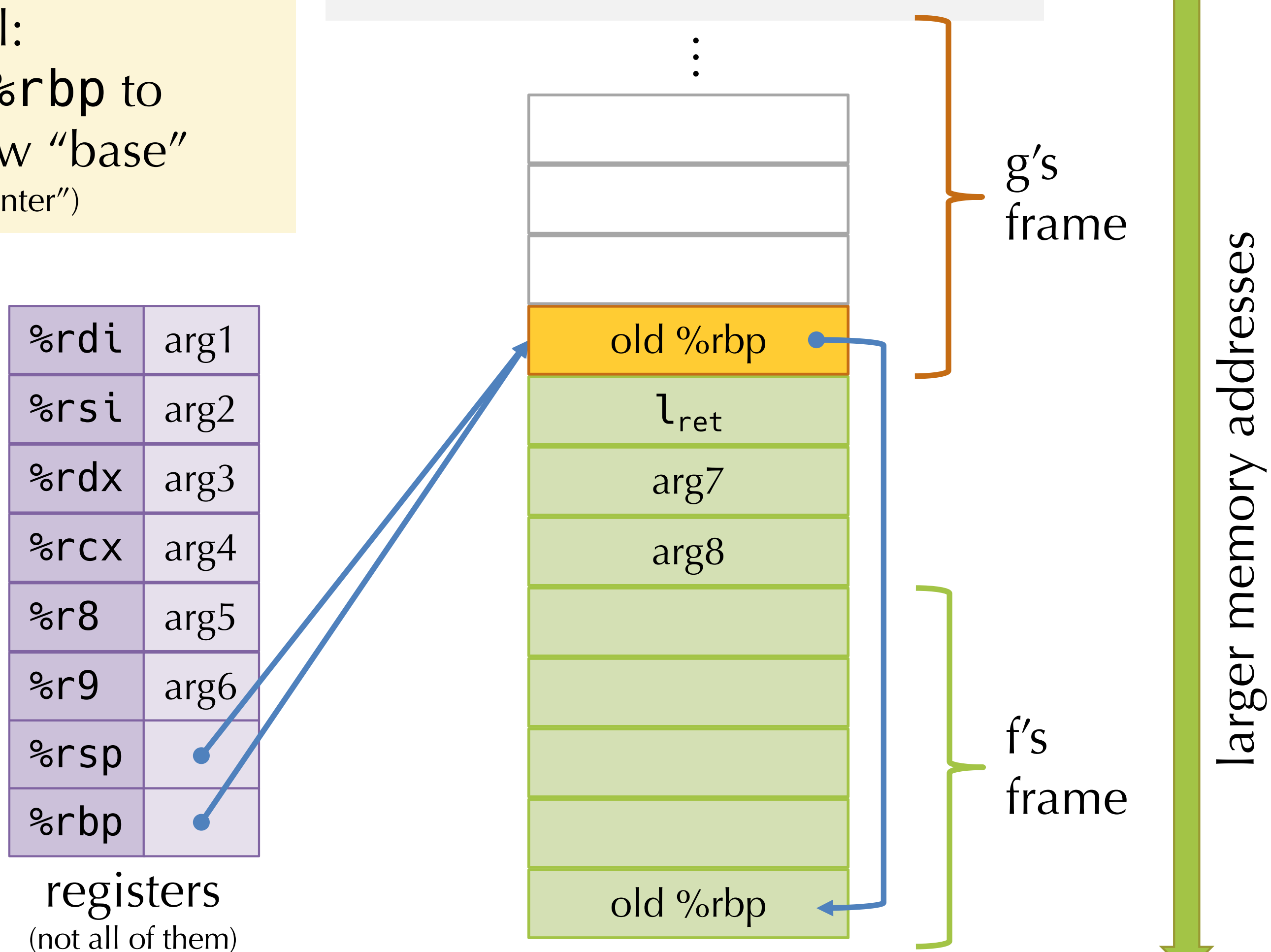
Callee protocol:
1. store the old `%rbp`



callee function prologue

```
g:        pushq %rbp
          movq %rsp, %rbp
          subq $128, %rsp
          …
```

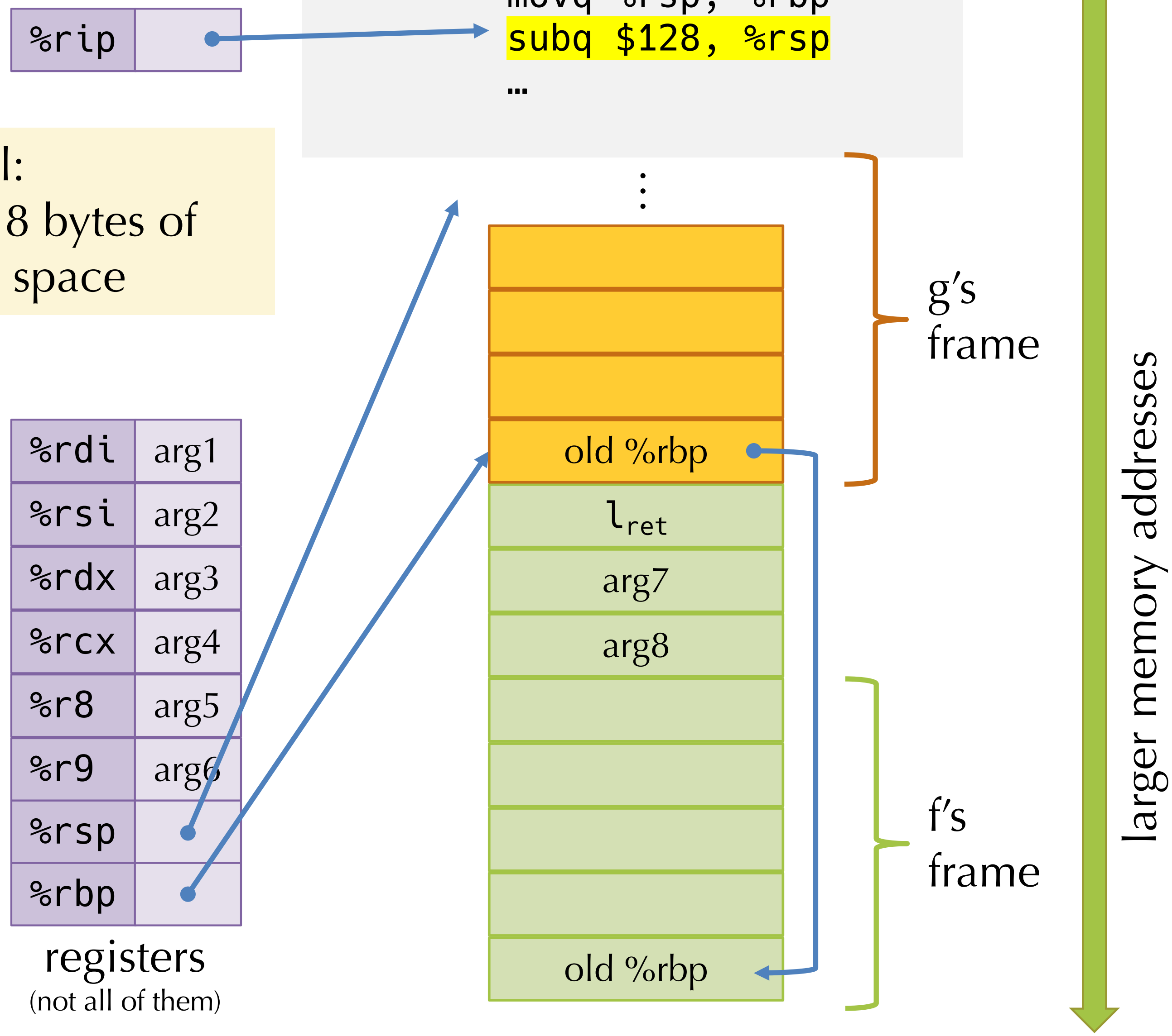
Callee protocol:
2. adjust the `%rbp` to point to the new "base"
(`%rbp` is the "base pointer")



callee function prologue

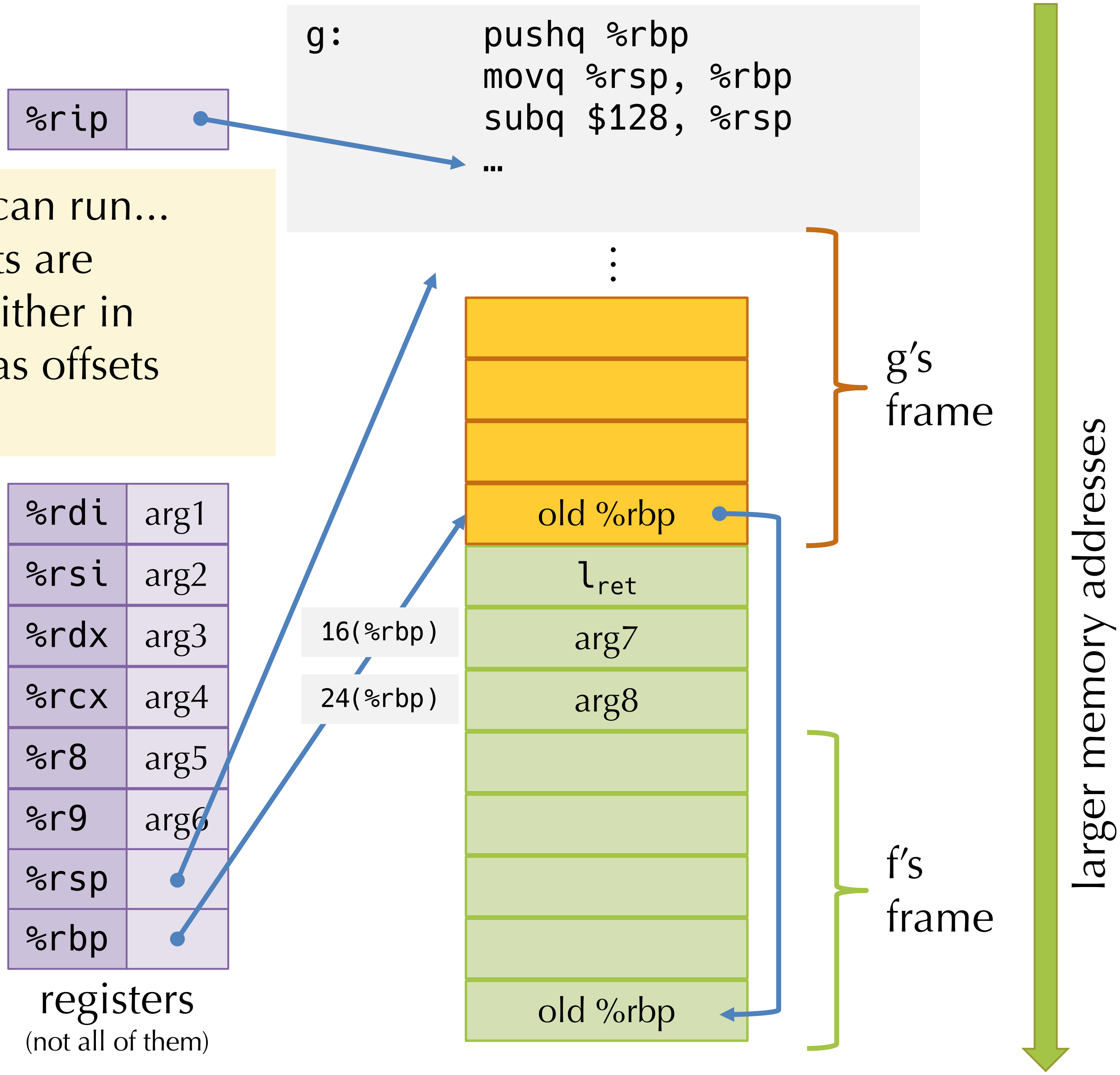
```
g:        pushq %rbp
          movq %rsp, %rbp
          subq $128, %rsp
          …
```

Callee protocol:
3. allocate 128 bytes of "scratch" stack space

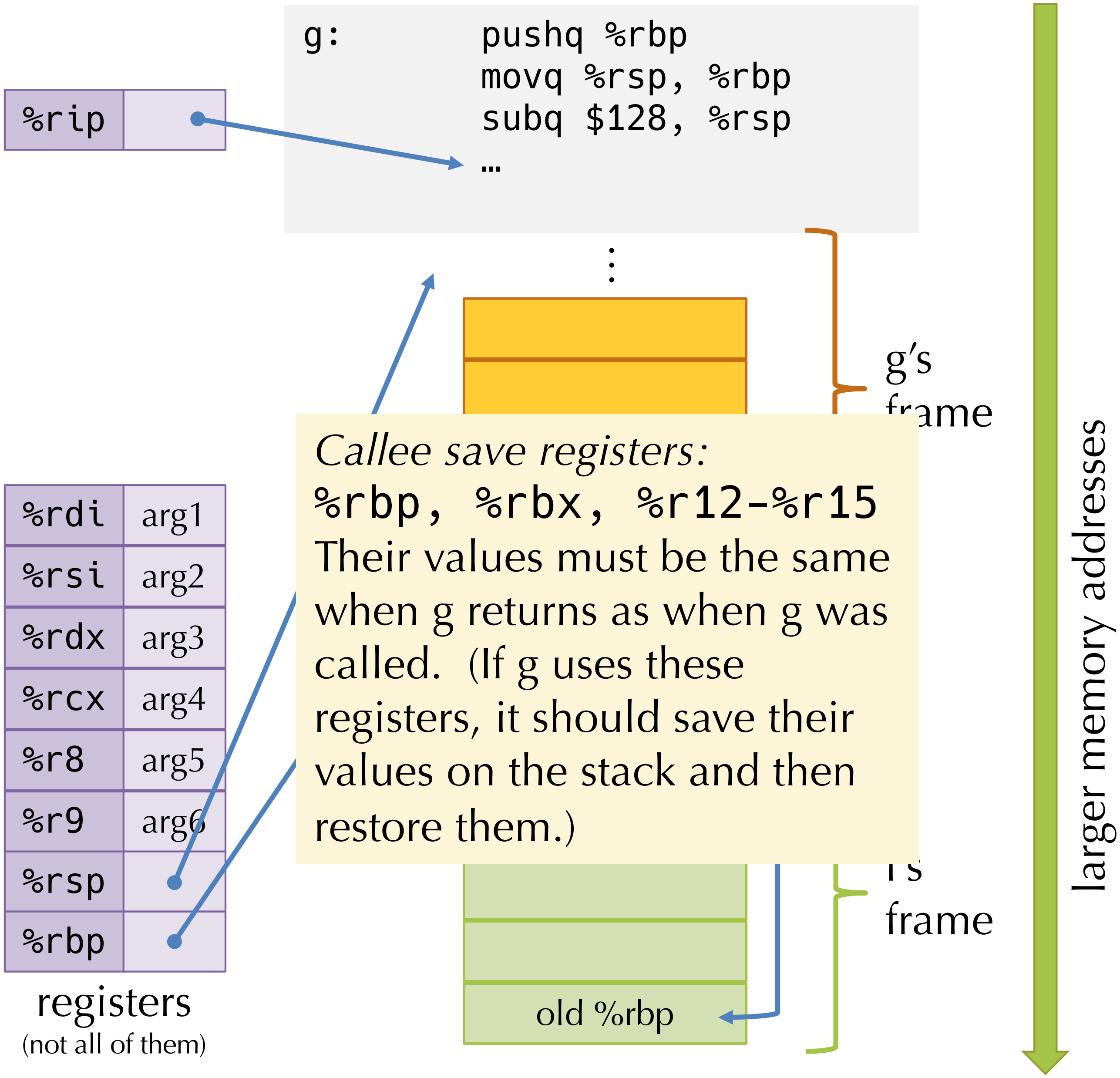


callee invariants: function arguments

Now g's body can run…
- its arguments are accessible either in registers or as offsets from `%rbp`



callee invariants: callee save registers

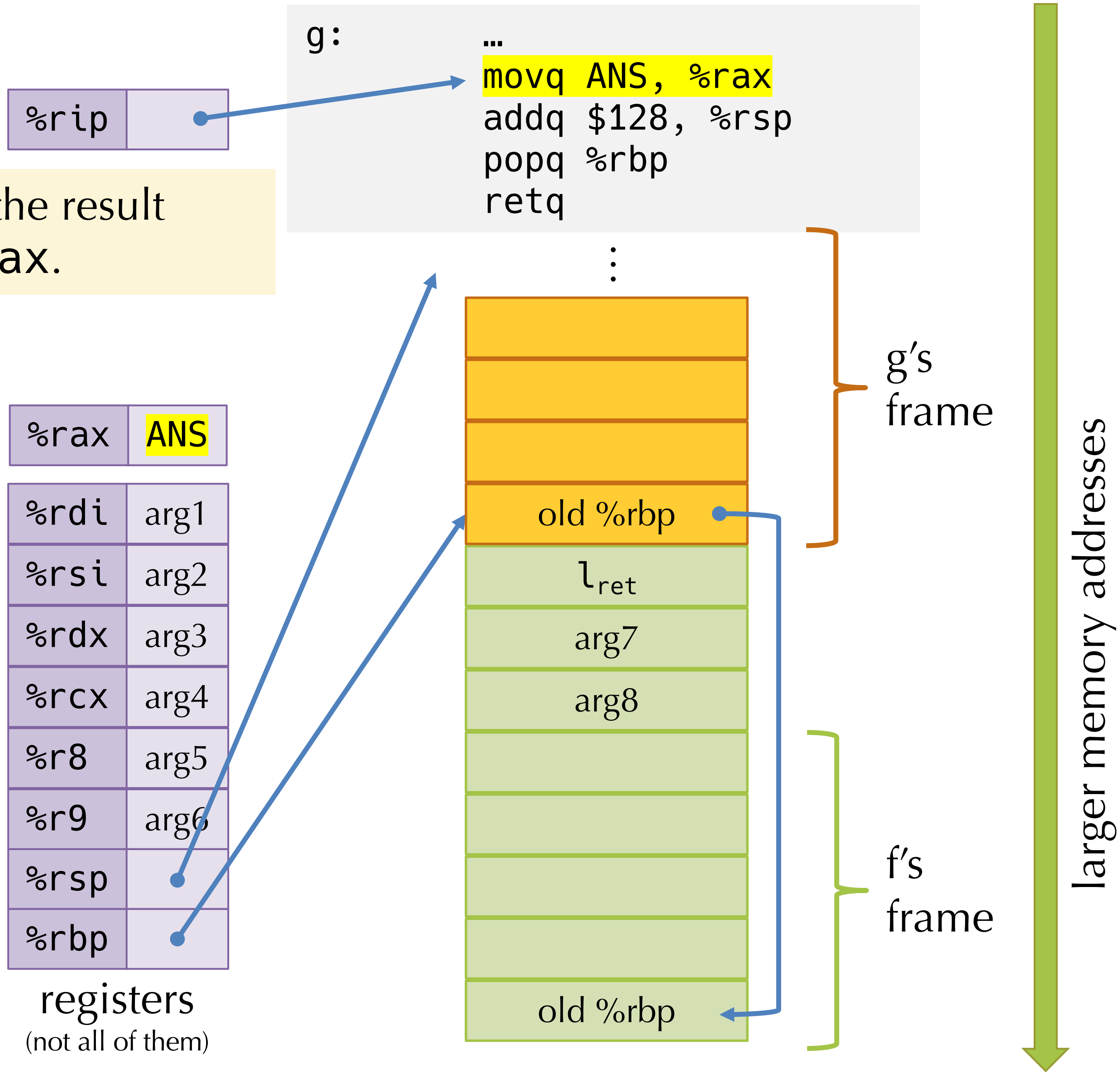


Callee save registers:
`%rbp, %rbx, %r12-%r15`
Their values must be the same when g returns as when g was called. (If g uses these registers, it should save their values on the stack and then restore them.)

callee epilogue (return protocol)

Step 1: Move the result (if any) into `%rax`.

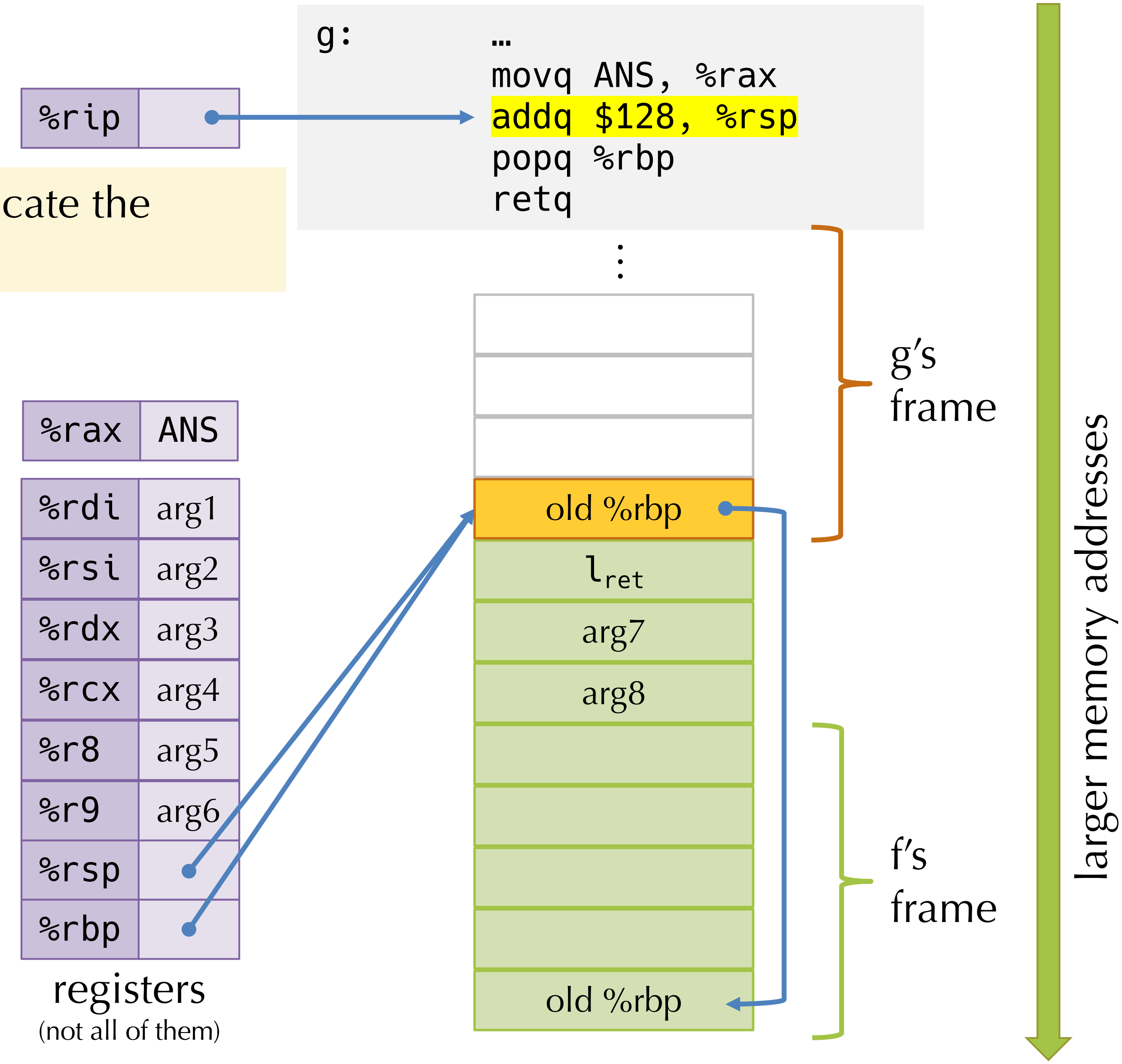
```
g:      …
        movq ANS, %rax
        addq $128, %rsp
        popq %rbp
        retq
```



callee epilogue (return protocol)

```
g:        …
          movq ANS, %rax
          addq $128, %rsp
          popq %rbp
          retq
```

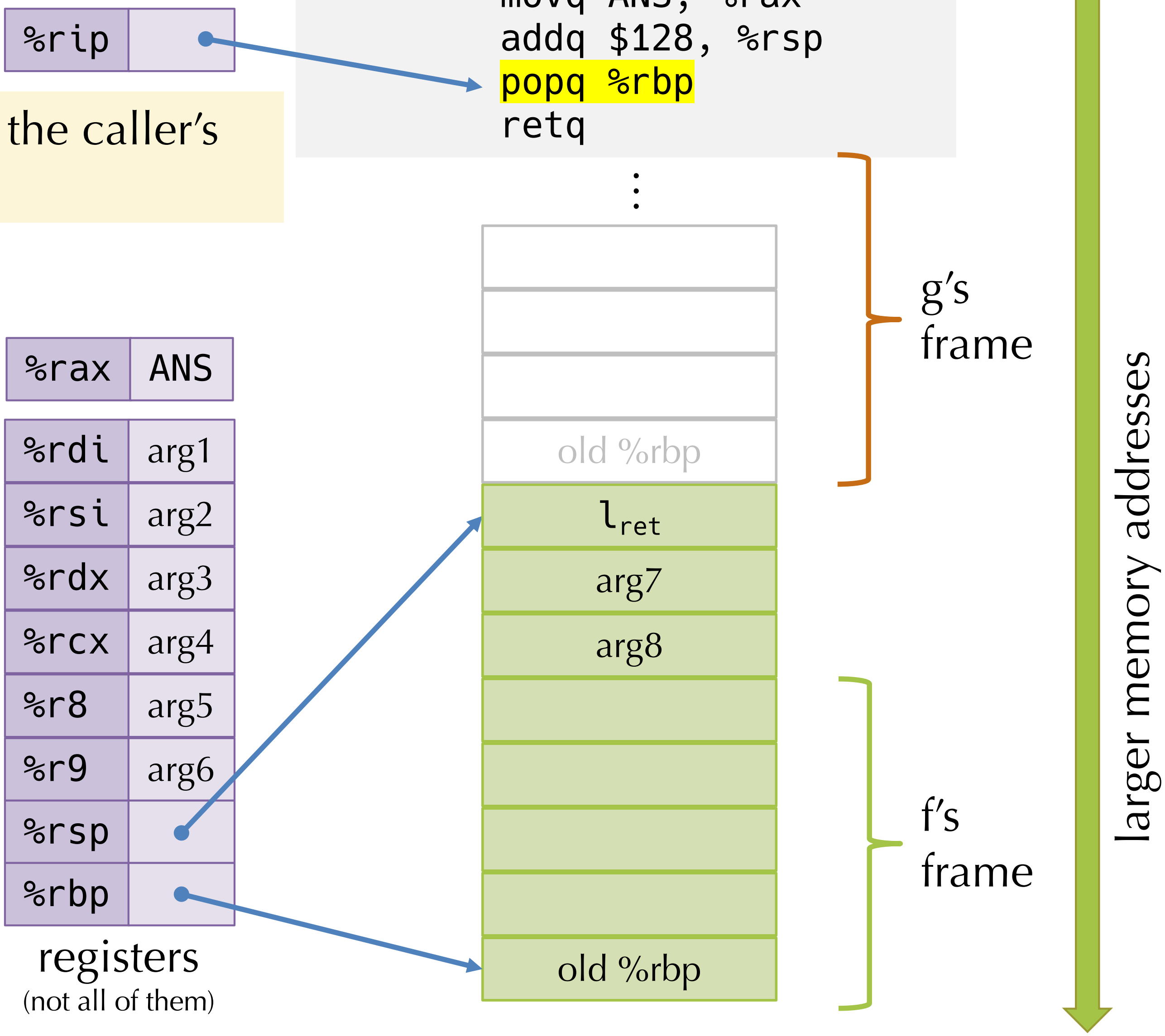
Step 2: deallocate the scratch space



callee epilogue (return protocol)

```
g:        …
          movq ANS, %rax
          addq $128, %rsp
          popq %rbp
          retq
```

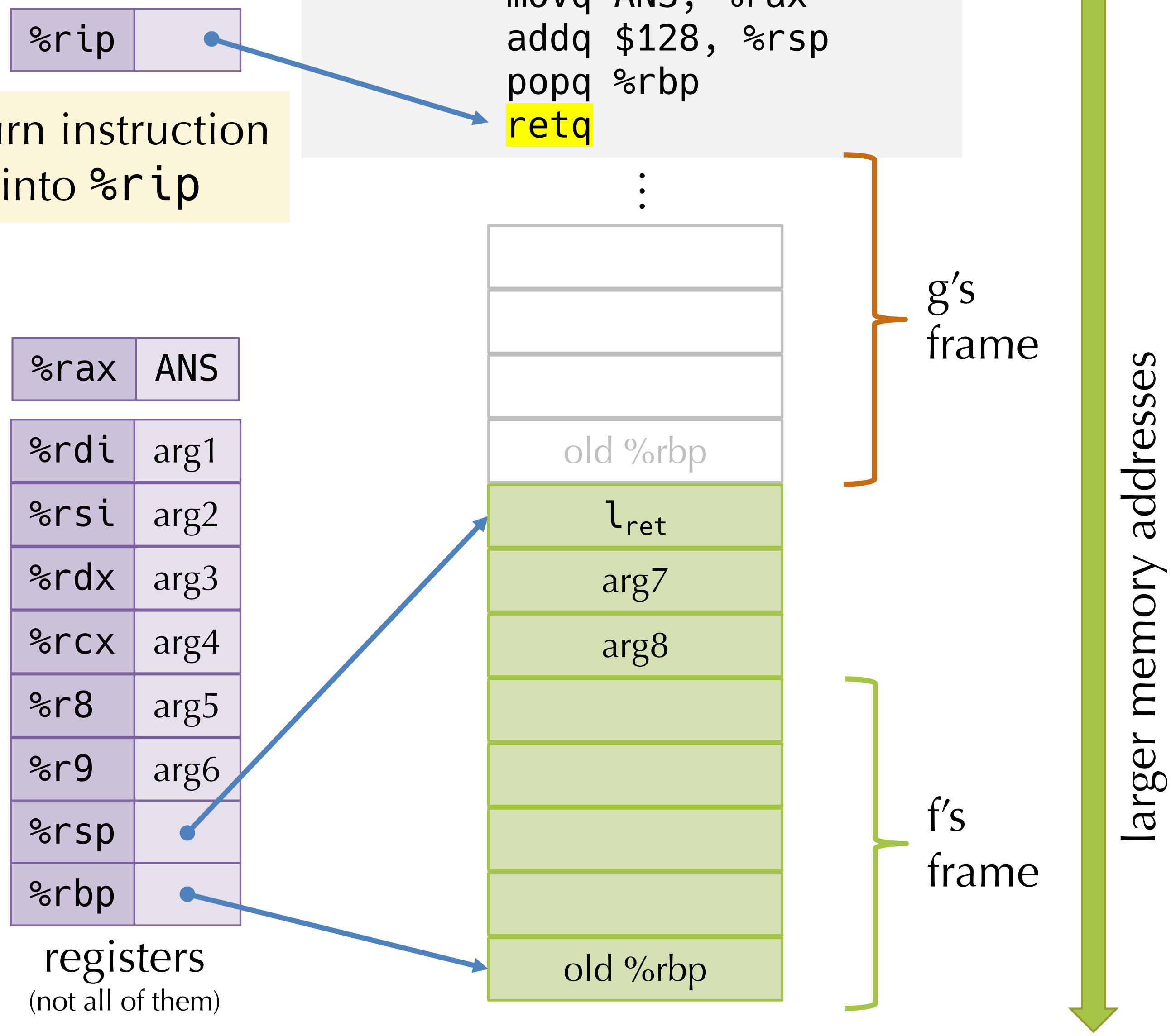
Step 3: restore the caller's `%rbp`



callee epilogue (return protocol)

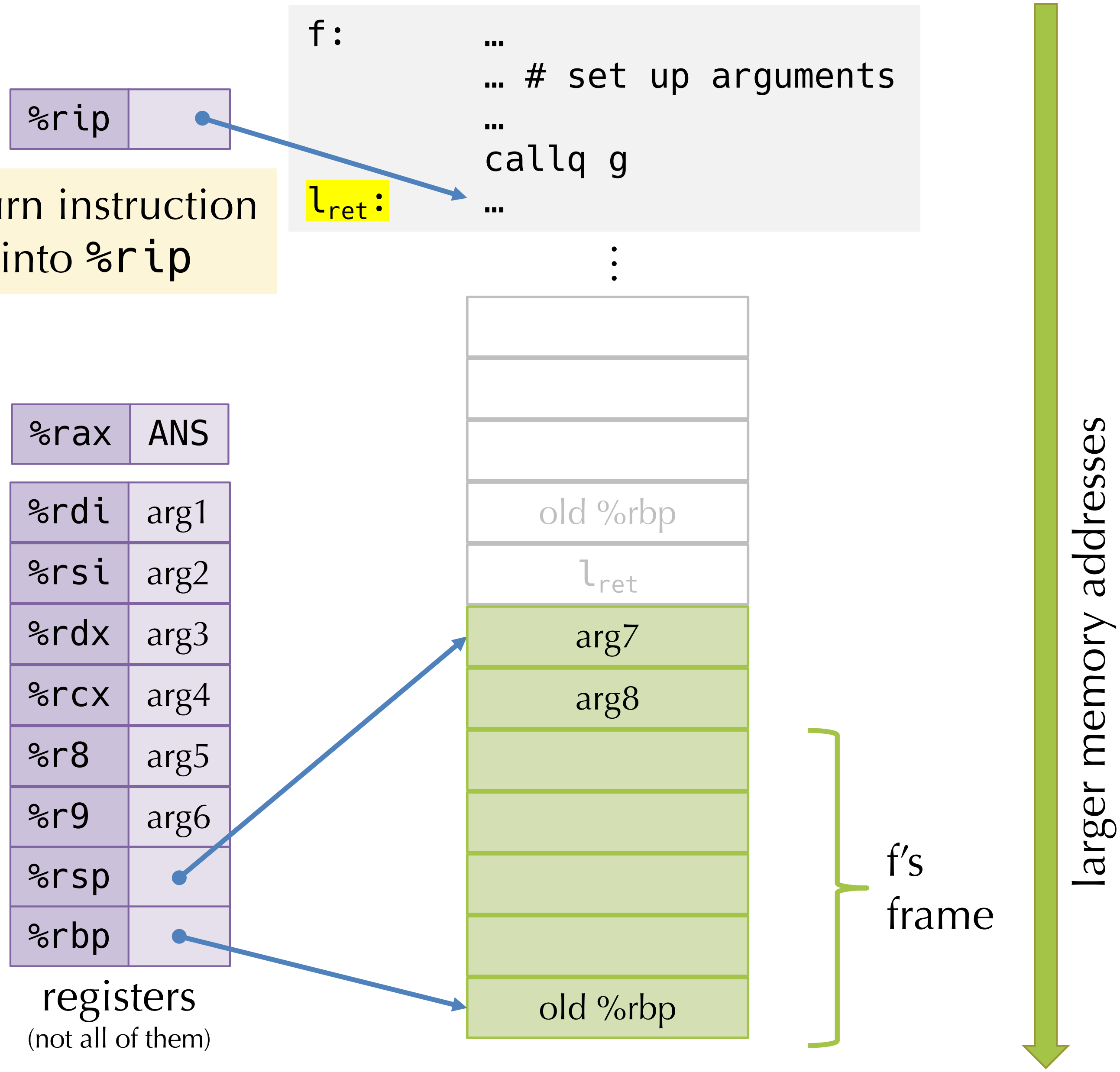
Step 4: the return instruction pops the stack into `%rip`

```
g:        …
          movq ANS, %rax
          addq $128, %rsp
          popq %rbp
          retq
```



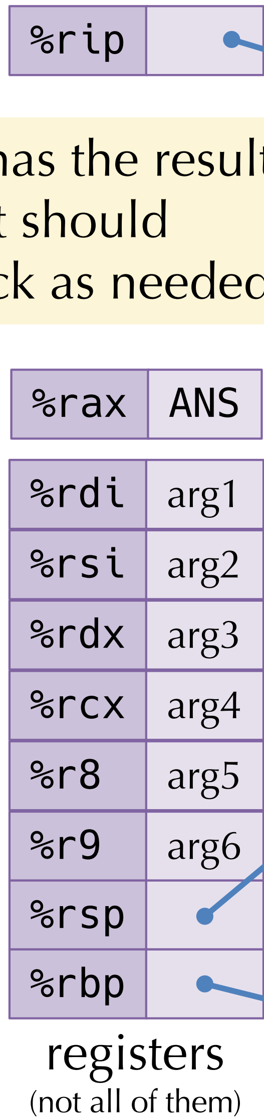
callee epilogue (return protocol)

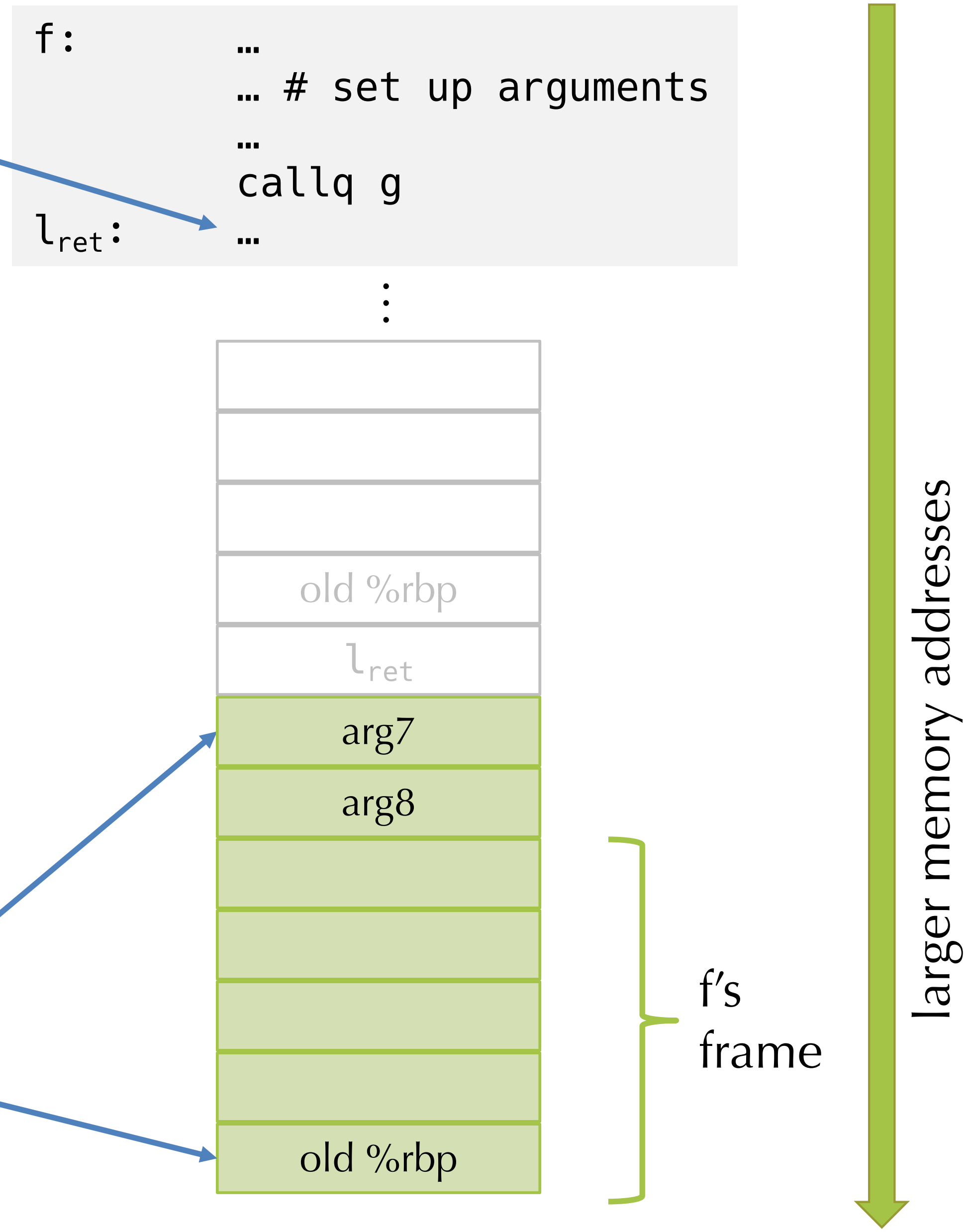
Step 4: the return instruction pops the stack into `%rip`



back in f

At this point, f has the result of g in `%rax`. It should clean up its stack as needed.





X86-64 SYSTEM V AMD 64 ABI

- More modern variant of C calling conventions
  - used on Linux, Solaris, BSD, OS X

- Callee save: `%rbp, %rbx, %r12-%r15`
- Caller save: all others

- Parameters 1 .. 6 go in: `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Parameters 7+ go on the stack (in right-to-left order)
  - so: for $n > 6$, the n^{th} argument is located at (((n-7)+2)*8)(`%rbp`)
  - e.g.: argument 7 is at `16(%rbp)` and argument 8 is at `24(%rbp)`

- Return value: in `%rax`

- 128 byte "red zone" – scratch pad for the callee's data
  - typical of C compilers, not required
  - can be optimized away

32-bit cdecl calling conventions

- Still "Standard" on X86 for many C-based operating systems
  - Still some wrinkles about return values
    (*e.g.*, some compilers use `EAX` and `EDX` to return small values)
  - 64 bit allows for packing multiple values in one register
- All arguments are passed on the stack in right-to-left order
- Return value is passed in `EAX`
- Registers `EAX`, `ECX`, `EDX` are caller save
- Other registers are callee save
  - Ignoring these conventions will cause havoc (bus errors or seg faults)