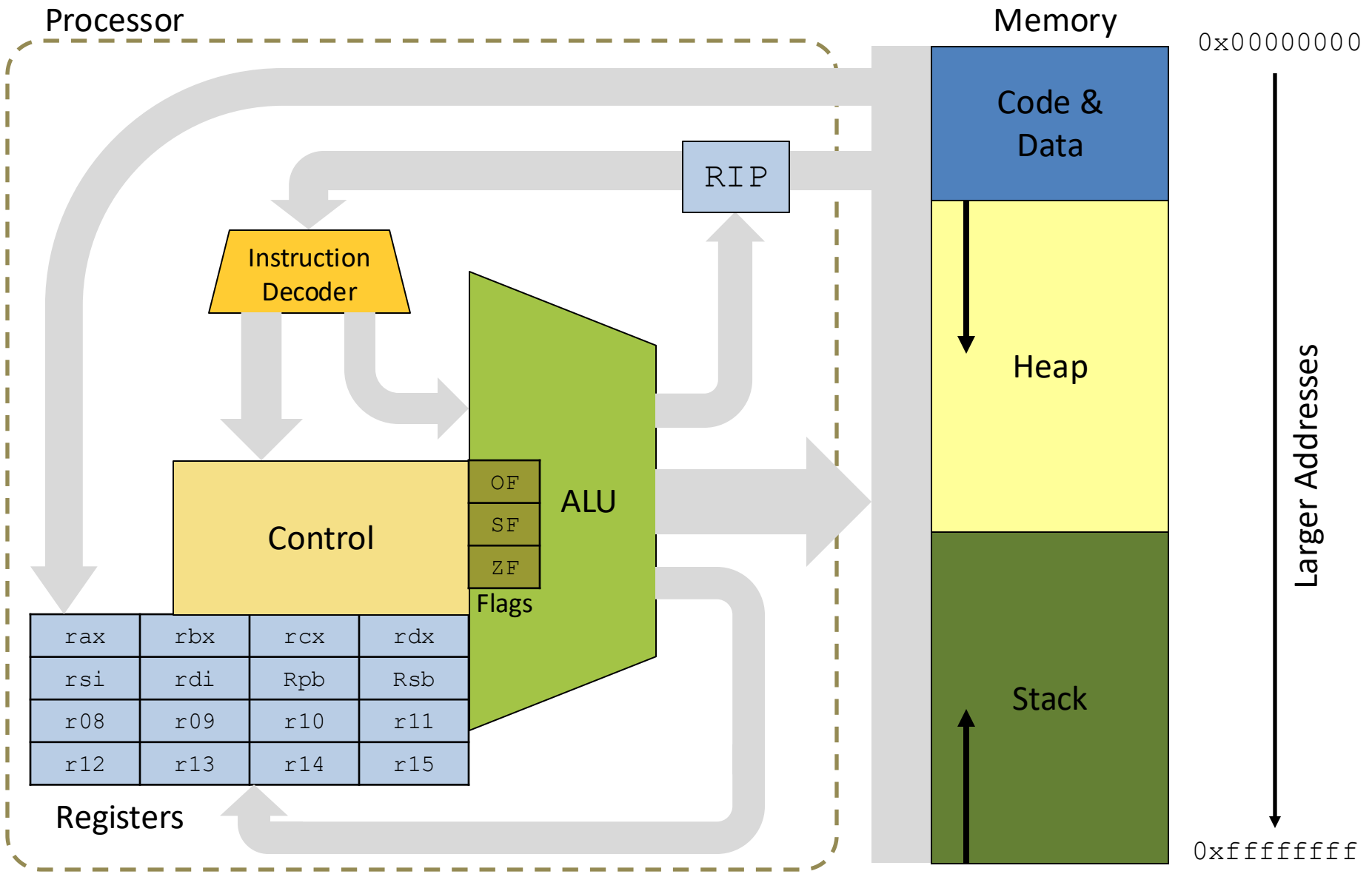
Lecture 4

CIS 3410/7000: COMPILERS

Announcements

- HW2: X86lite
  - Available on the course web pages soon. (look for announcement on Blackboard / Piazza).
  - Due: October 21st.
  - Pair-programming project
  - NOTE: much more difficult than hw1, so please start early!

X86 Schematic



X86lite State: Condition Flags & Codes

- X86 instructions set flags as a side effect
- X86lite has only 3 flags:
  - `OF`: "overflow" set when the result is too big/small to fit in 64-bit reg.
  - `SF`: "sign" set to the sign or the result (0=positive, 1 = negative)
  - `ZF`: "zero" set when the result is 0

- From these flags, we can define *Condition Codes*
  - To compare SRC1 and SRC2, compute SRC1 – SRC2 to set the flags
  - `eq` equality holds when `ZF` is set
  - `ne` inequality holds when (not `ZF`)
  - `gt` greater than holds when (not le) holds,
    - i.e. (SF = OF) && not(ZF)
  - `lt` less than holds when `SF` <> `OF`
    - Equivalently: ((`SF` && not `OF`) || (not `SF` && `OF`))
  - `ge` greater or equal holds when (not lt) holds, i.e. (SF = OF)
  - `le` than or equal holds when `SF` <> `OF` or `ZF`

Code Blocks & Labels

- X86 assembly code is organized into *labeled blocks*:

```
label1:
        <instruction>
        <instruction>
        …
        <instruction>

label2:
        <instruction>
        <instruction>
        …
        <instruction>
```

- Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls).
- Labels are translated away by the linker and loader – instructions live in the heap in the "code segment"
- An X86 program begins executing at a designated code label (usually "main").

Conditional Instructions

- `cmpq` SRC1, SRC2 — Compute SRC2 – SRC1, set condition flags
- `setb`CC DEST — DEST's lower byte ← if CC then 1 else 0
- `j`CC SRC — `rip` ← if CC then SRC else fallthrough
- Example:

```
cmpq %rcx, %rax
je __truelbl
```

Compare `rax` to `ecx`
If `rax` = `rcx` then jump to `__truelbl`

Jumps, Call and Return

- `jmp` SRC `rip` ← SRC Jump to location in SRC

- `callq` SRC Push `rip`; `rip` ← SRC
  - Call a procedure: Push the program counter to the stack (decrementing `rsp`) and then jump to the machine instruction at the address given by SRC.

- `retq` Pop into `rip`
  - Return from a procedure: Pop the current top of the stack into `rip` (incrementing `rsp`).
  - This instruction effectively jumps to the address at the top of the stack

See: runtime.c and x86.ml in lec04.zip

DEMO: HANDCODING X86LITE

Compiling, Linking, Running

- To use hand-coded X86:
  1. Compile main.ml (or something like it) to either native or bytecode
  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 -o test runtime.c test.s
  4. You should be able to run the resulting exectuable:
     ./test

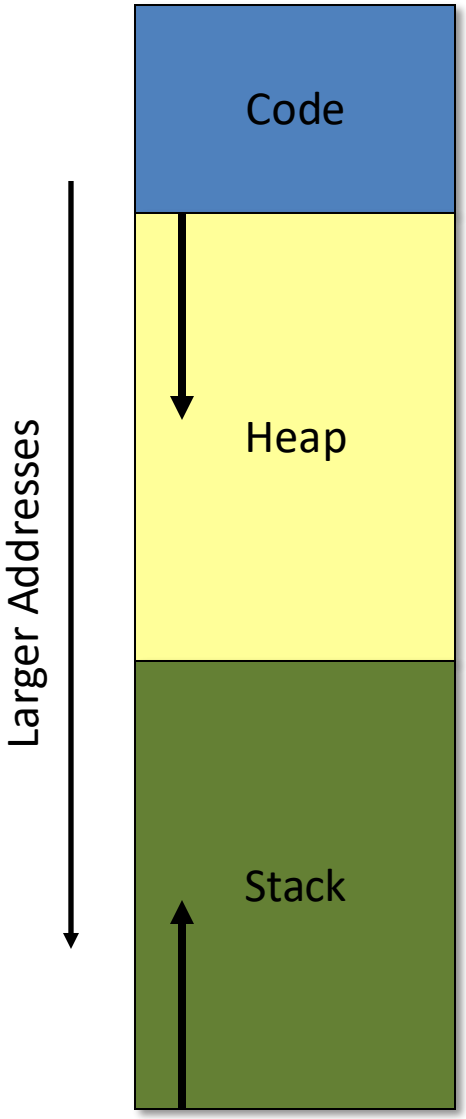
- If you want to debug in gdb:
  - Call gcc with the –g flag too

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"
  - managed by 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

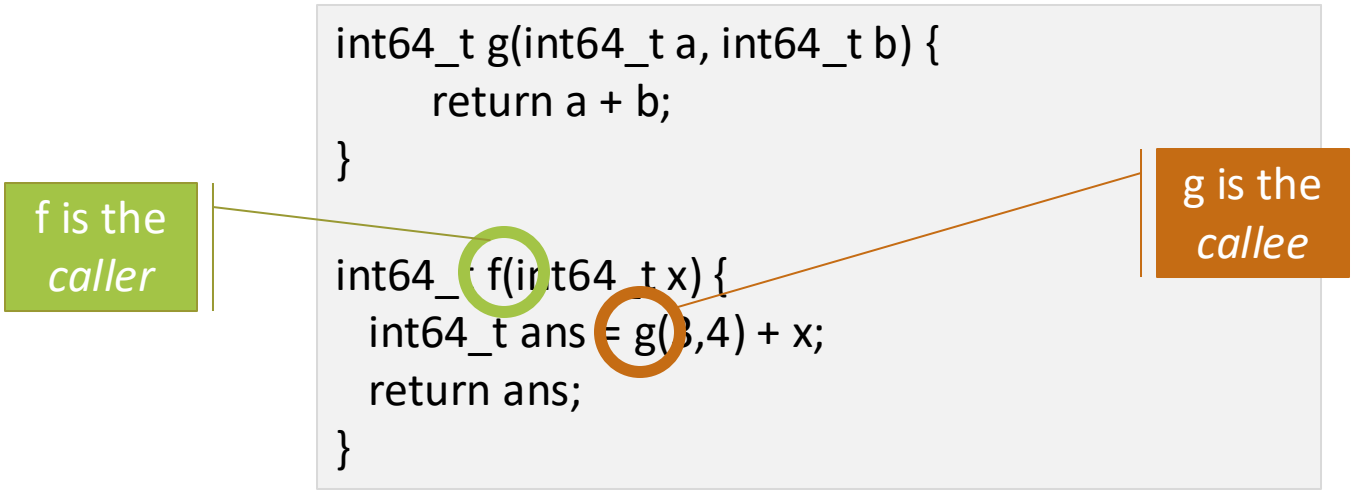
- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- 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*

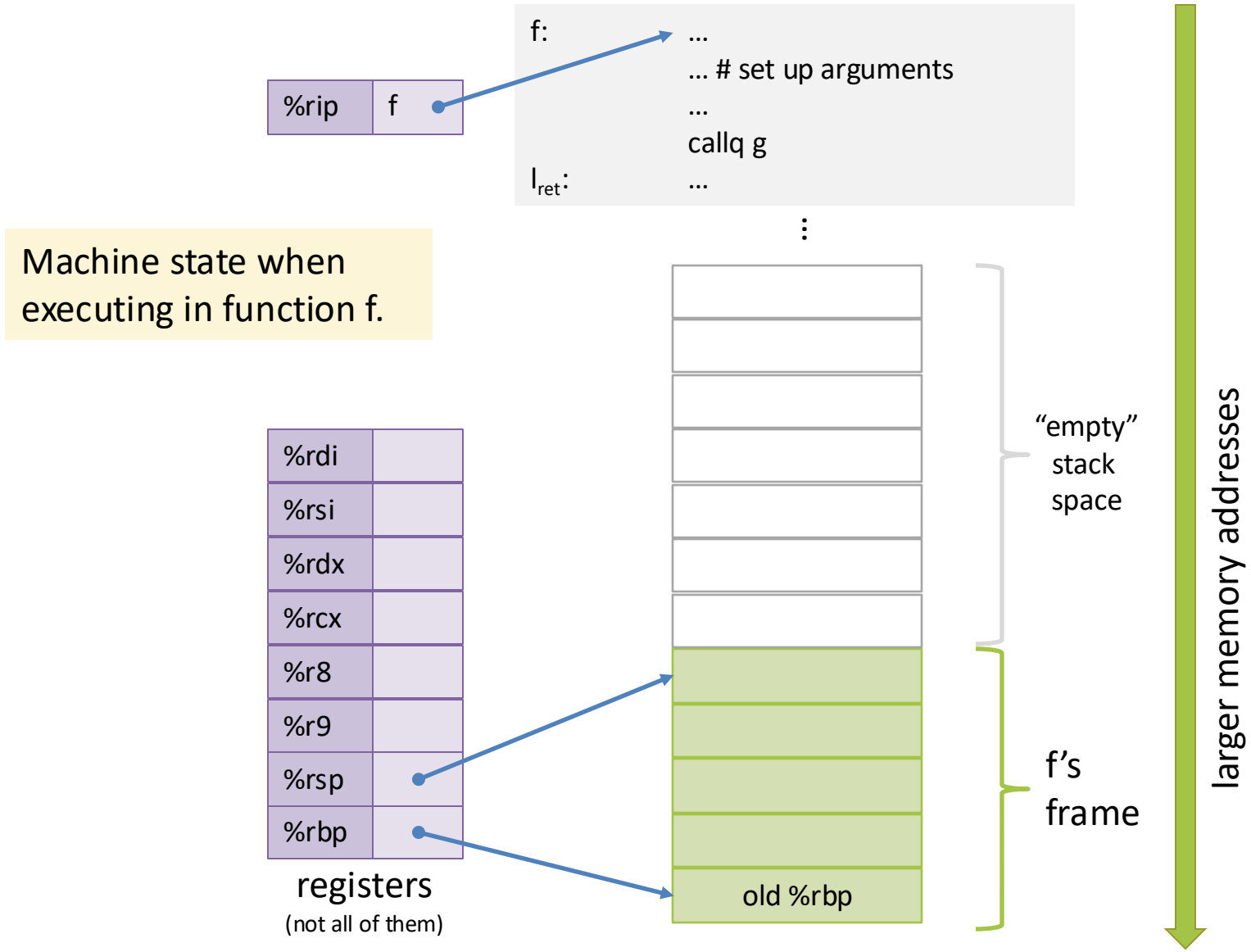
Calling Conventions

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



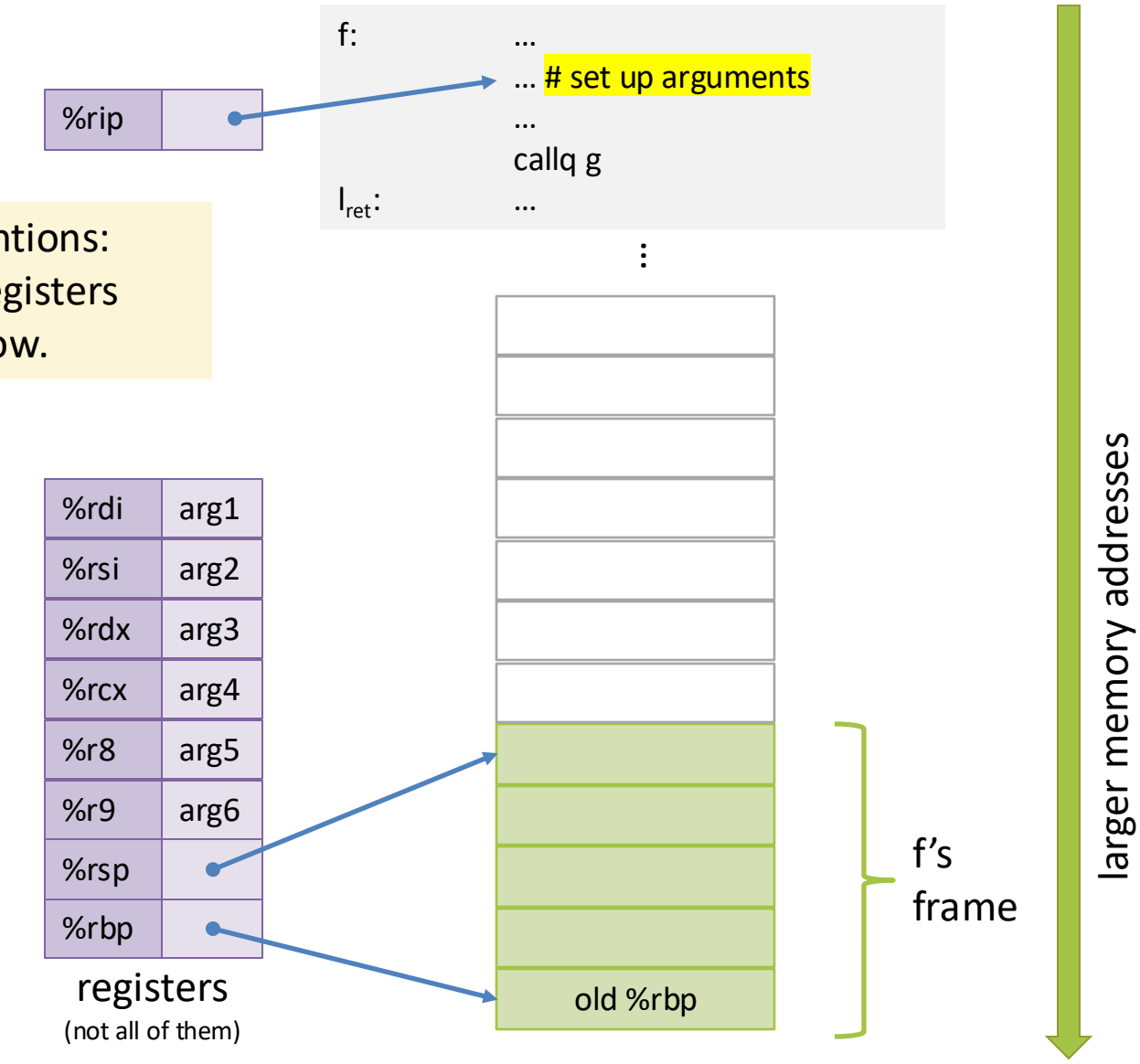
- Designate registers either:
  - Caller Save – *e.g.*, freely usable by the called code
  - Callee Save – *e.g.*, must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes supporting variable number of arguments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

x64 Calling Conventions: Caller Protocol

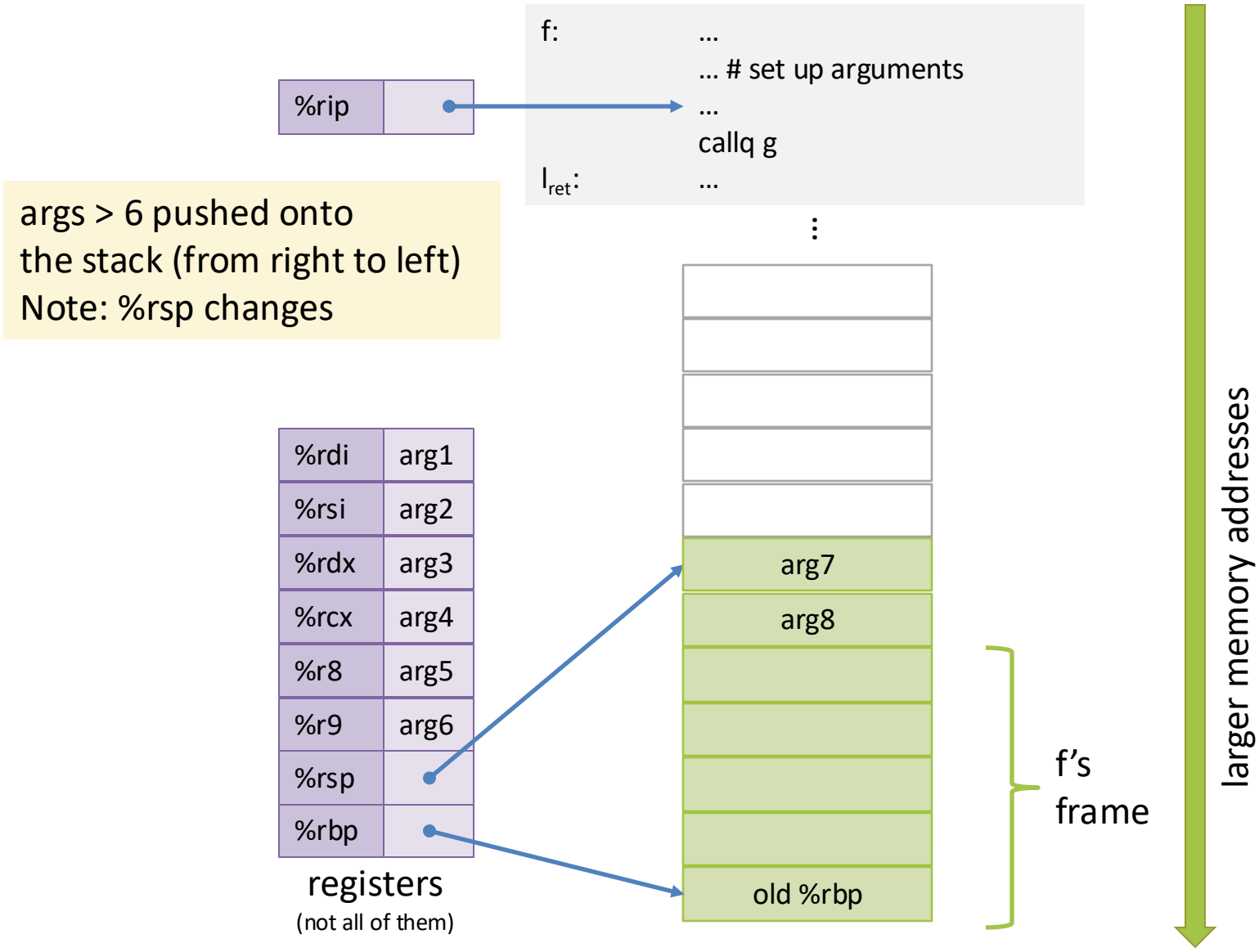


x64 Calling Conventions: Caller Protocol

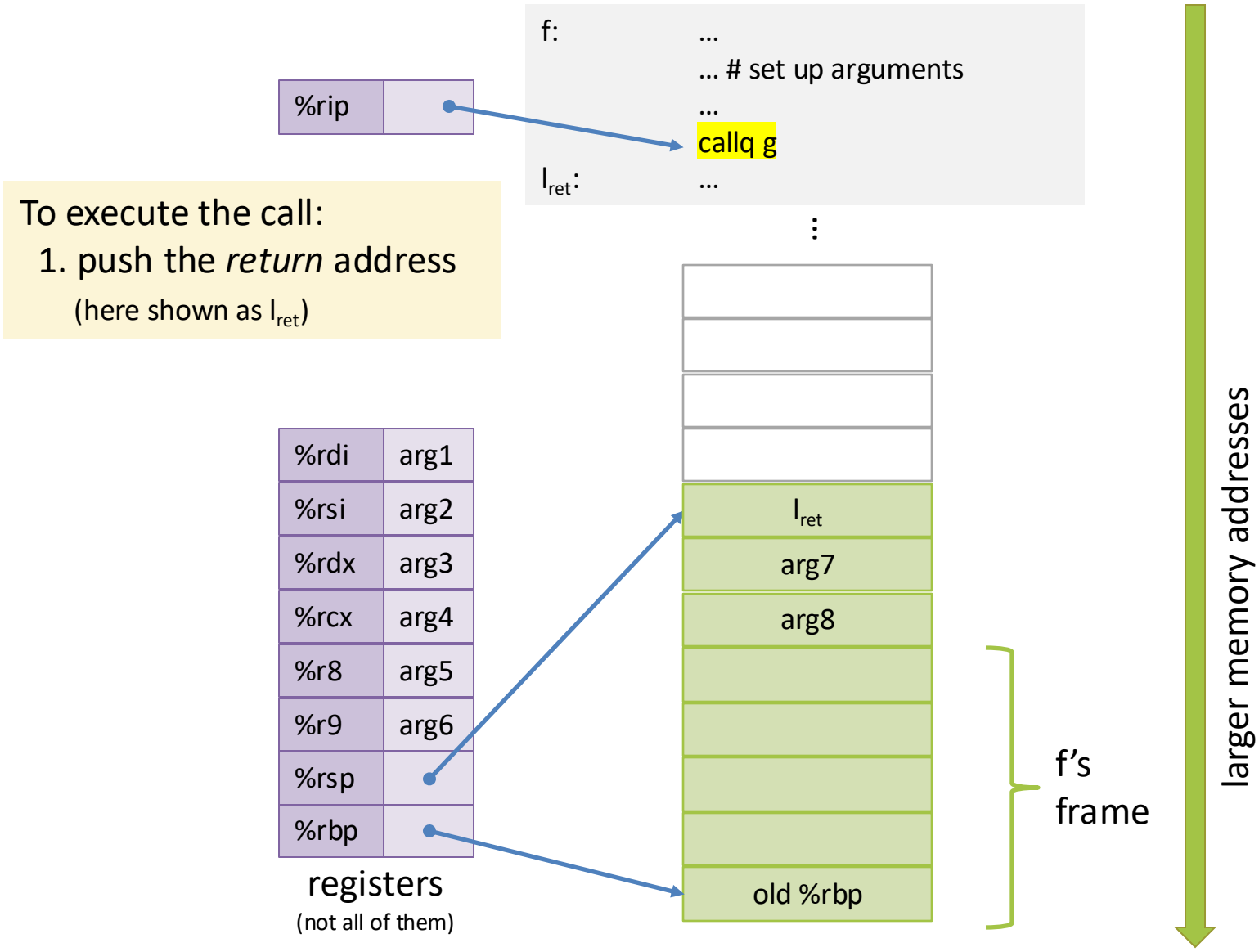
Calling conventions:
args 1…6 in registers
as shown below.



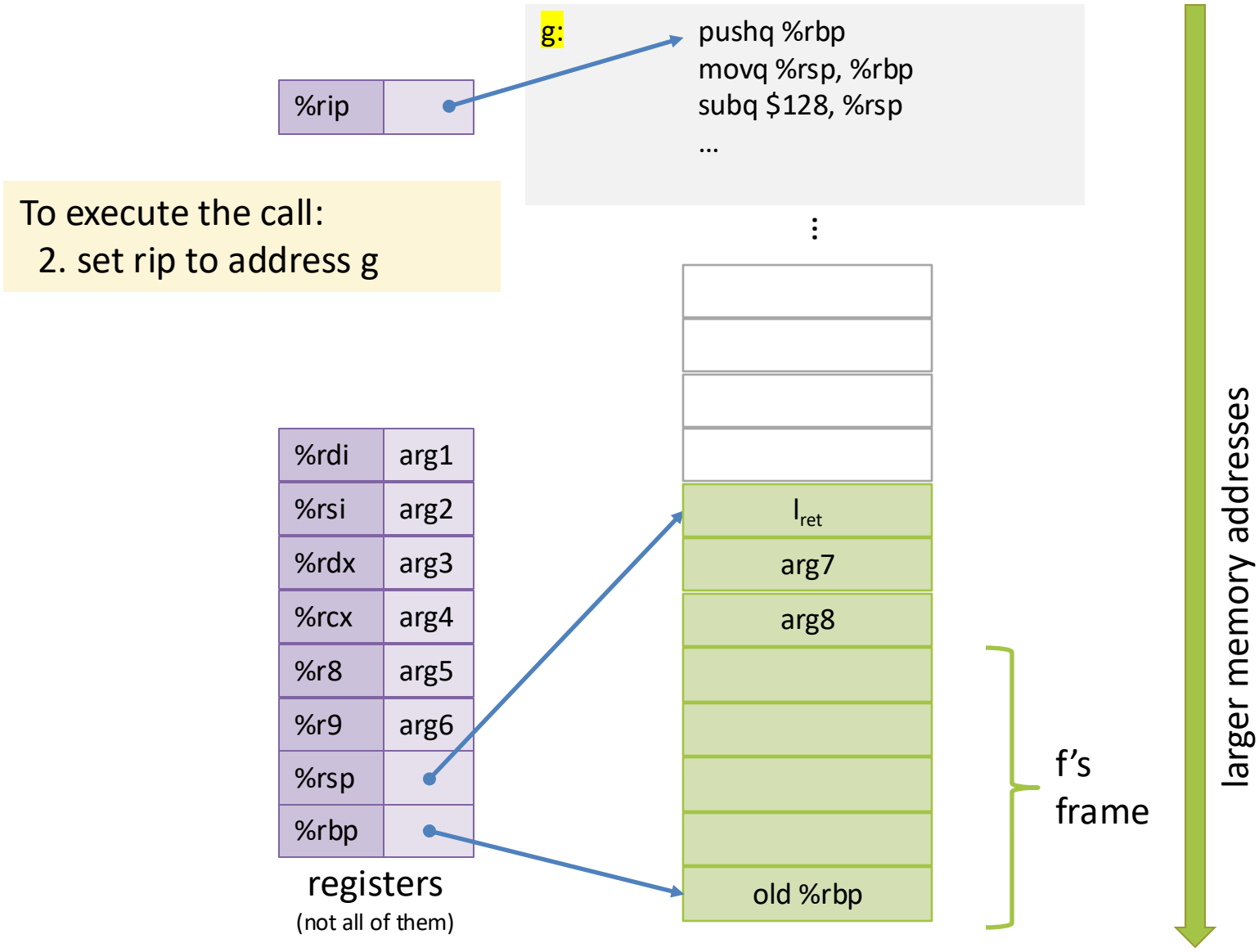
x64 Calling Conventions: Caller Protocol



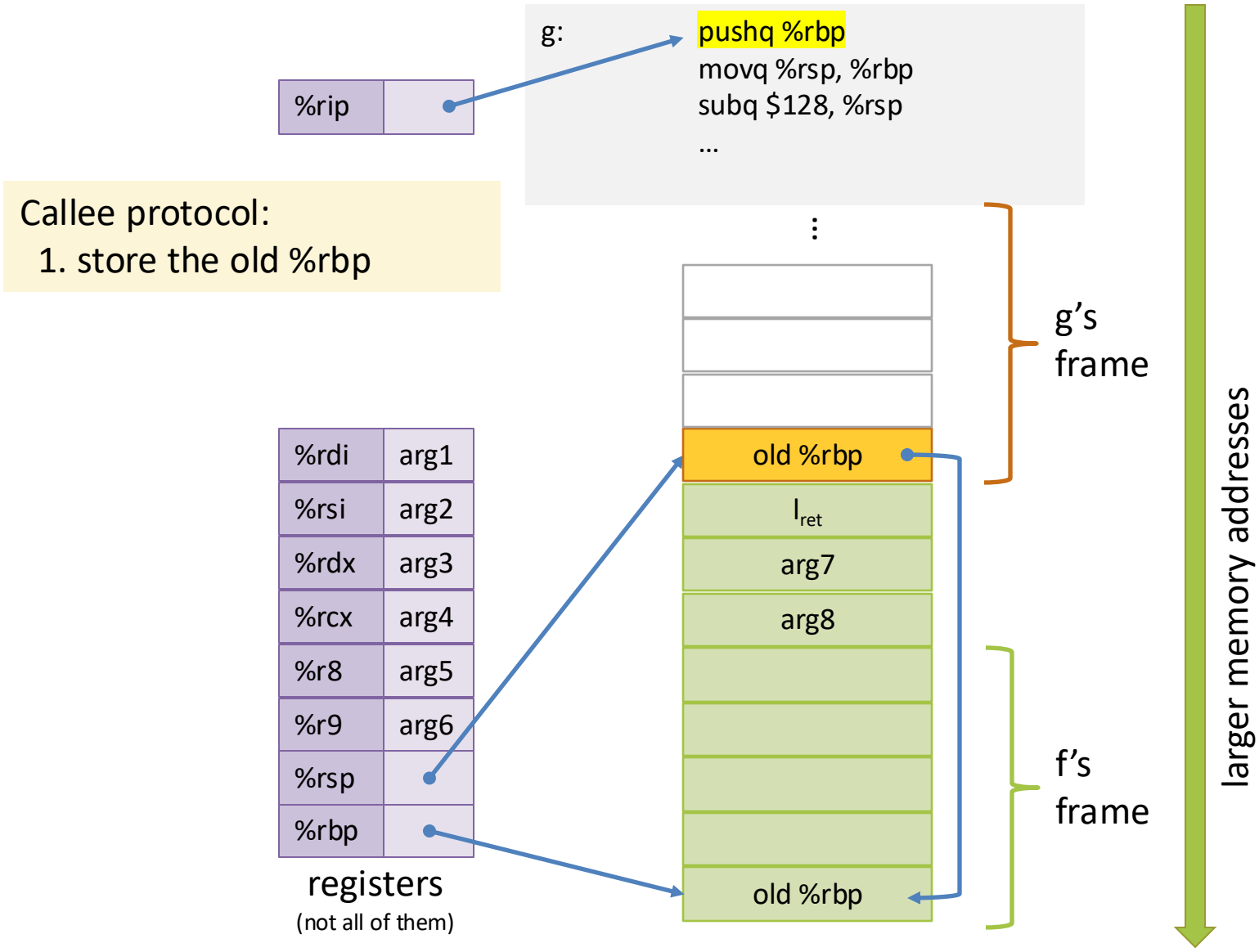
call instruction



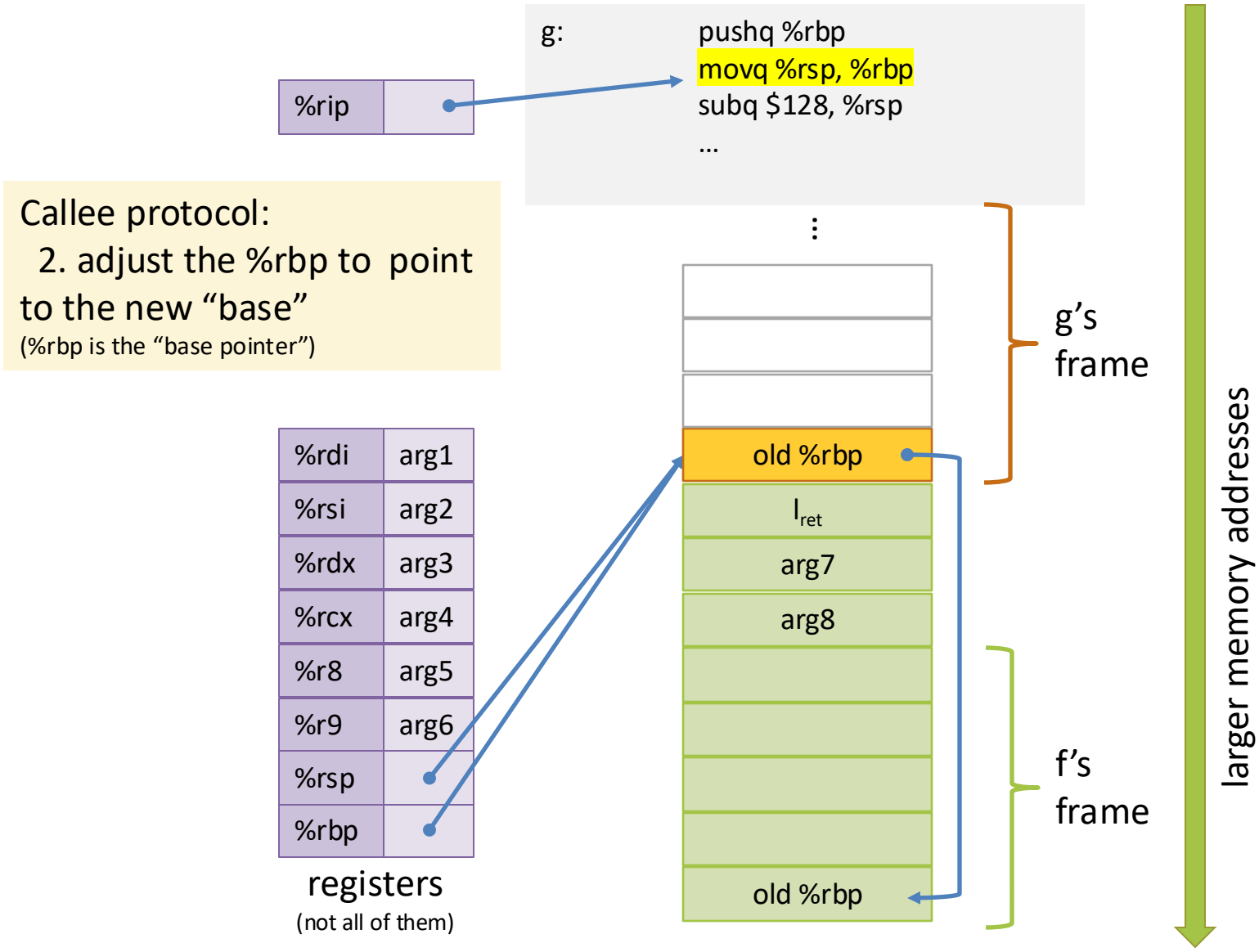
call instruction



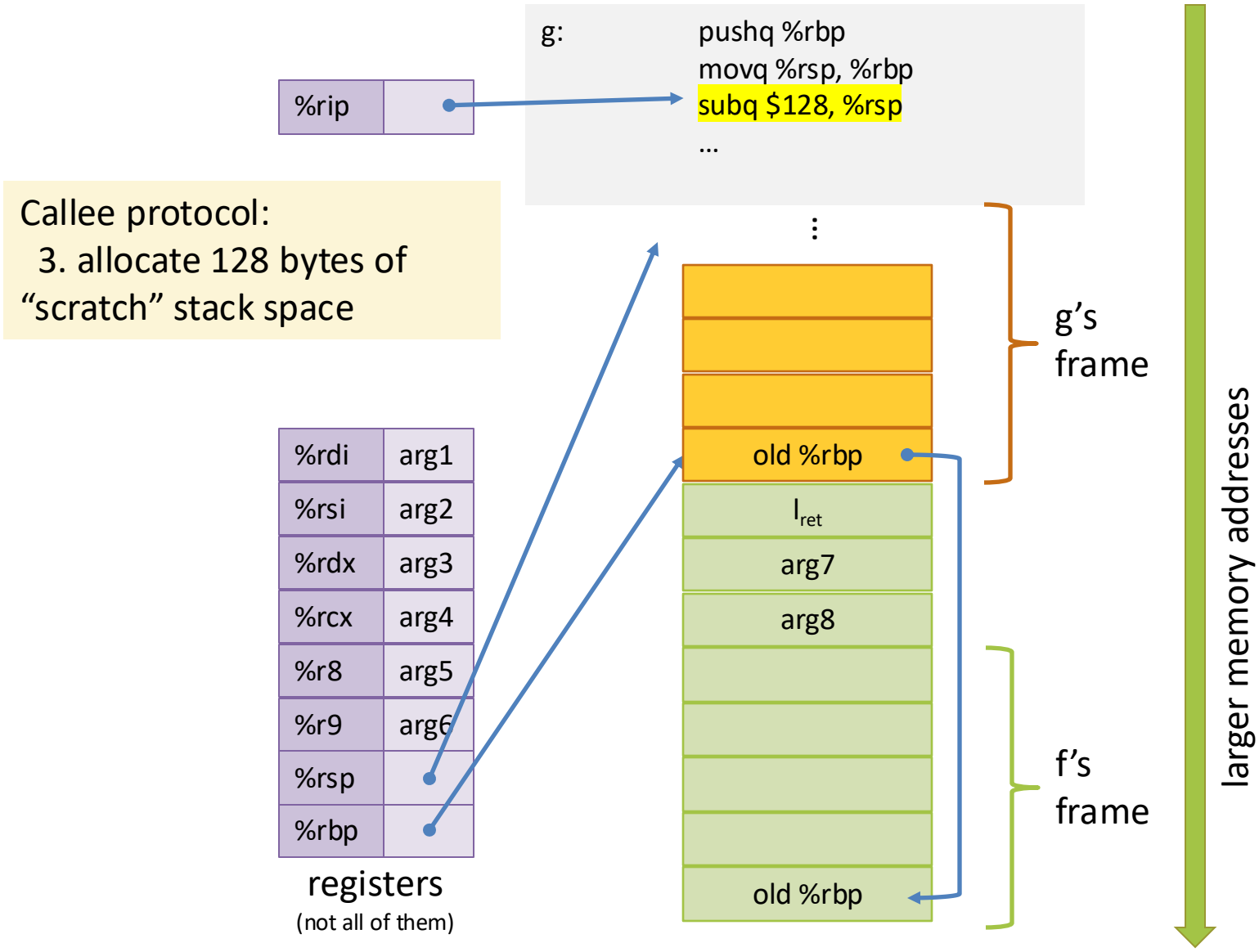
callee function prologue



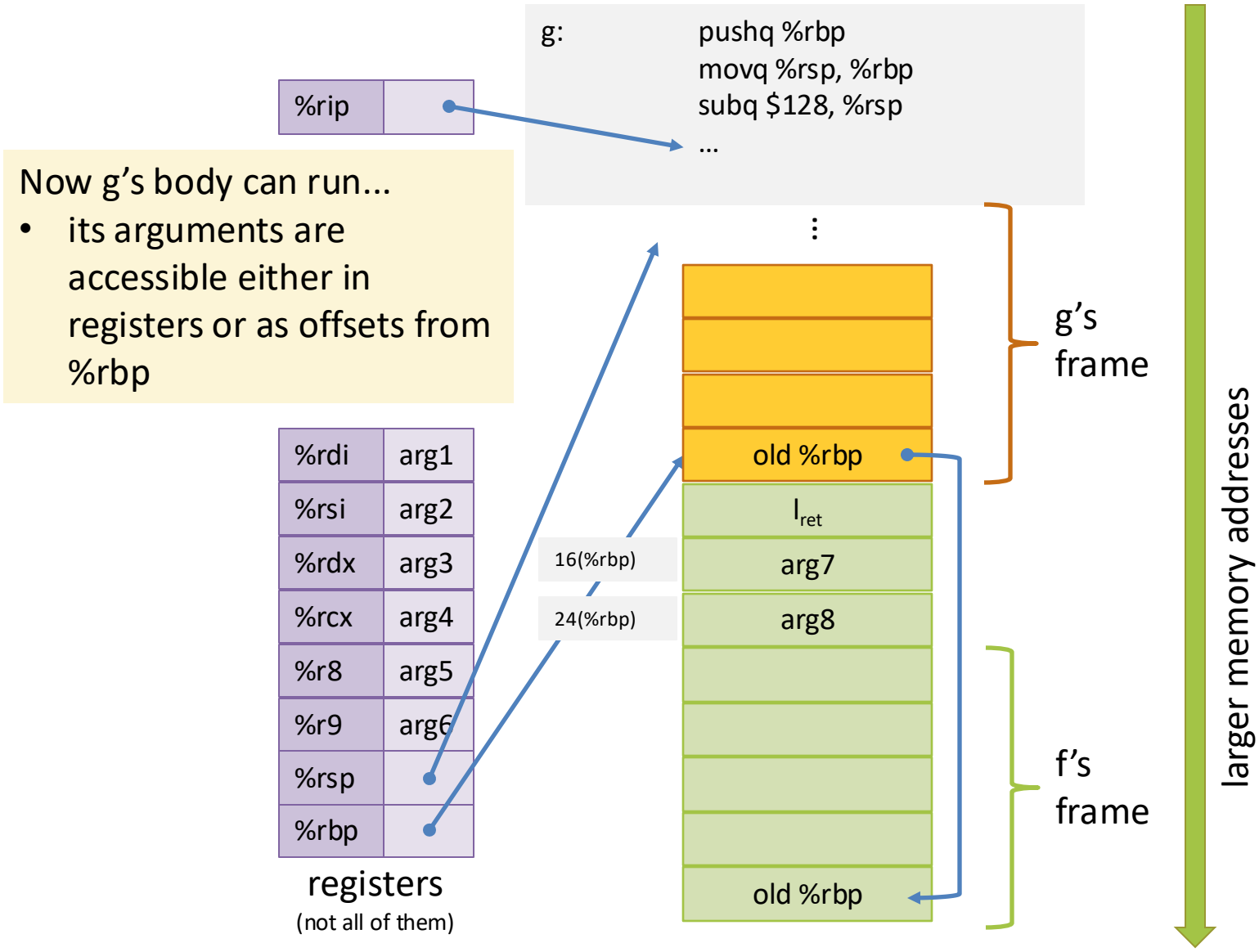
callee function prologue



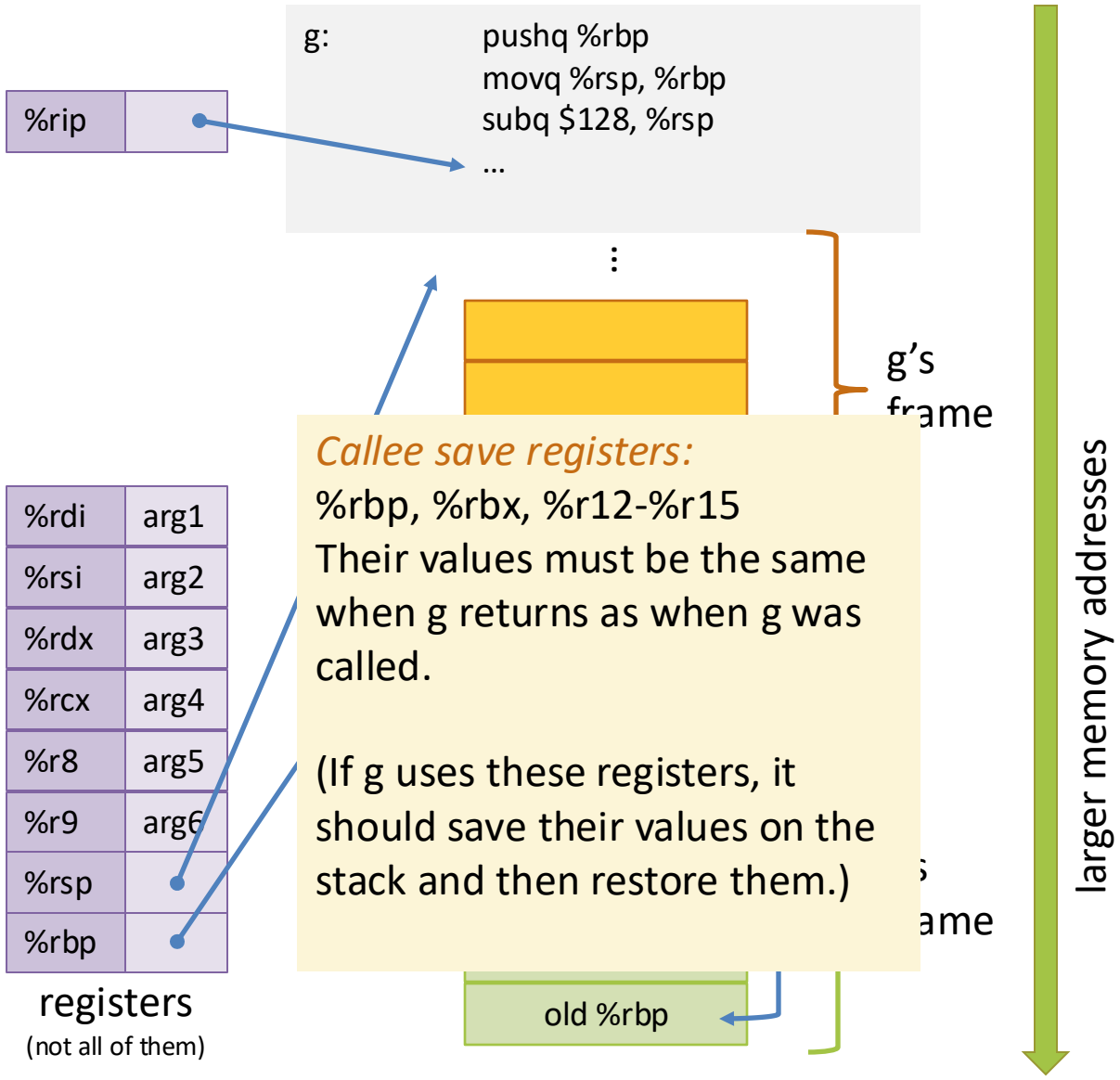
callee function prologue



callee invariants: function arguments



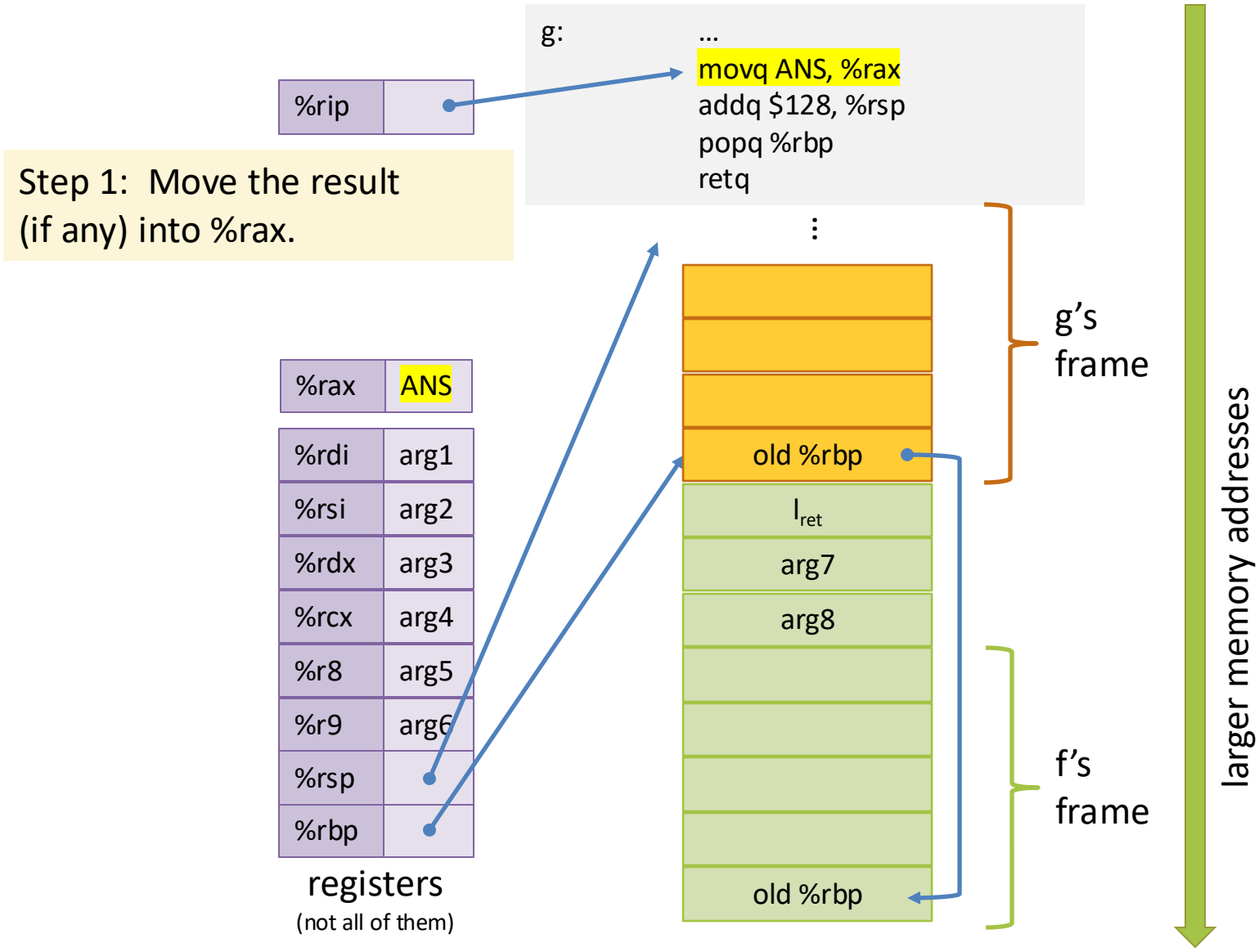
callee invariants: callee save registers



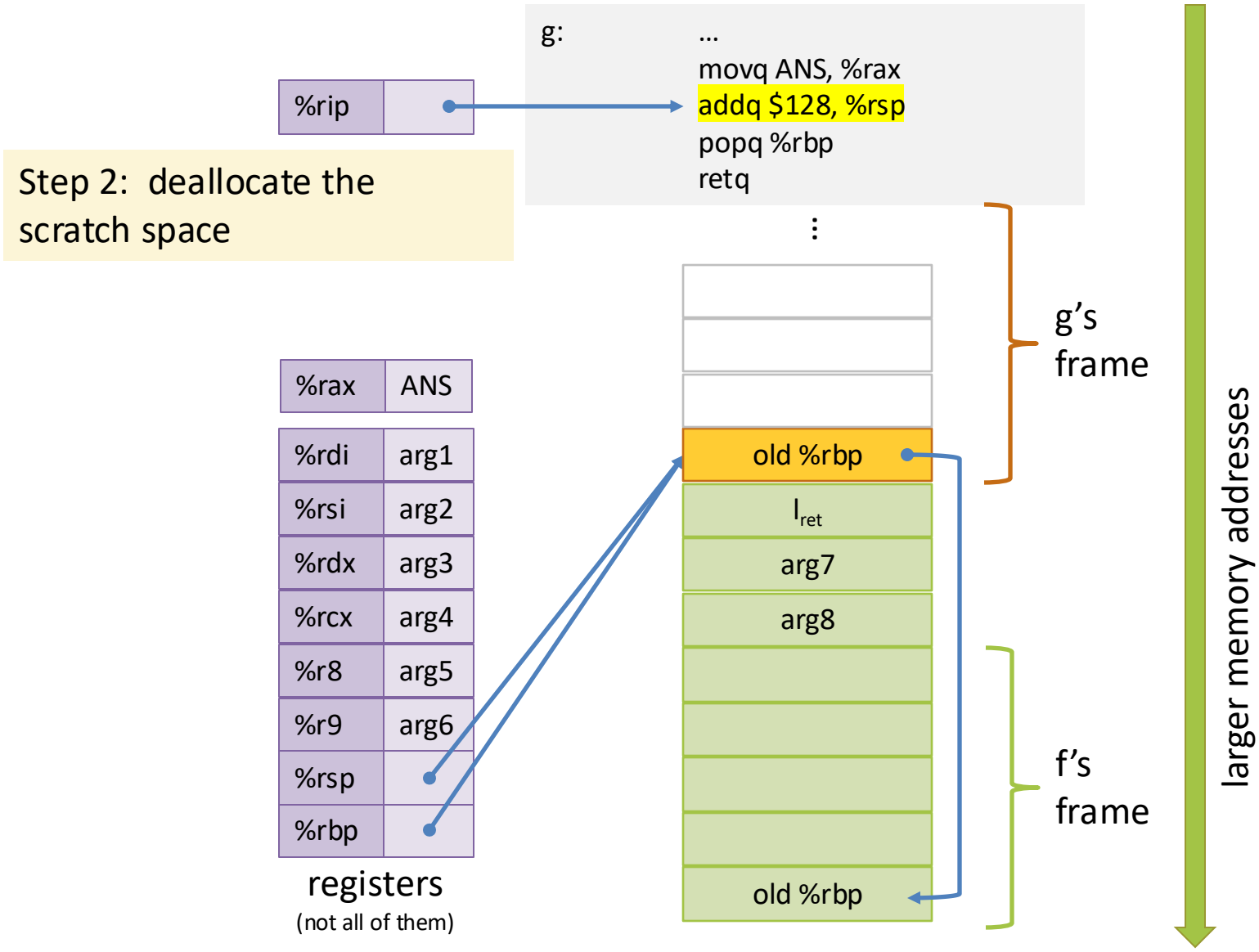
Callee save registers:
%rbp, %rbx, %r12-%r15
Their values must be the same when g returns as when g was called.

(If g uses these registers, it should save their values on the stack and then restore them.)

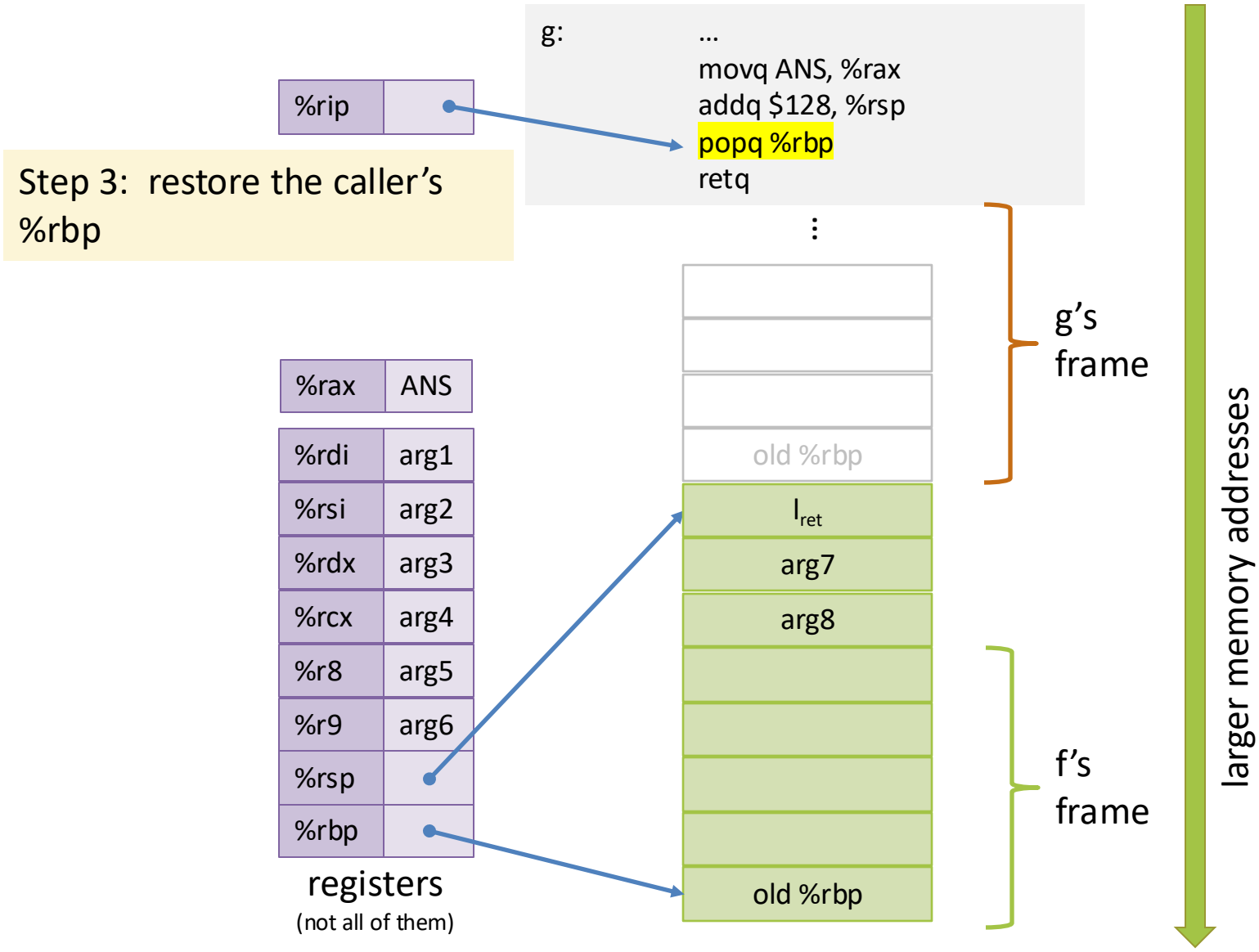
callee epilogue (return protocol)



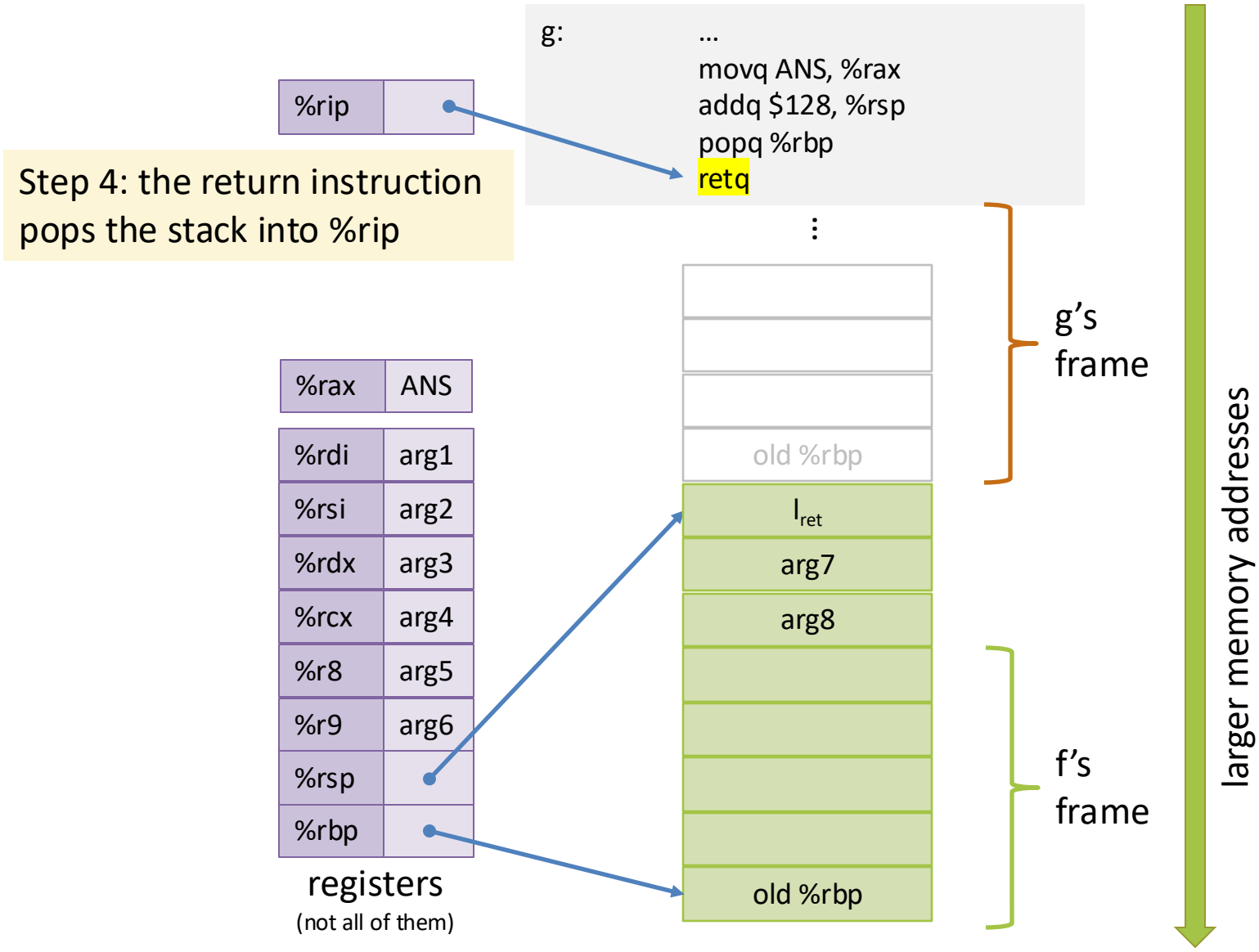
callee epilogue (return protocol)



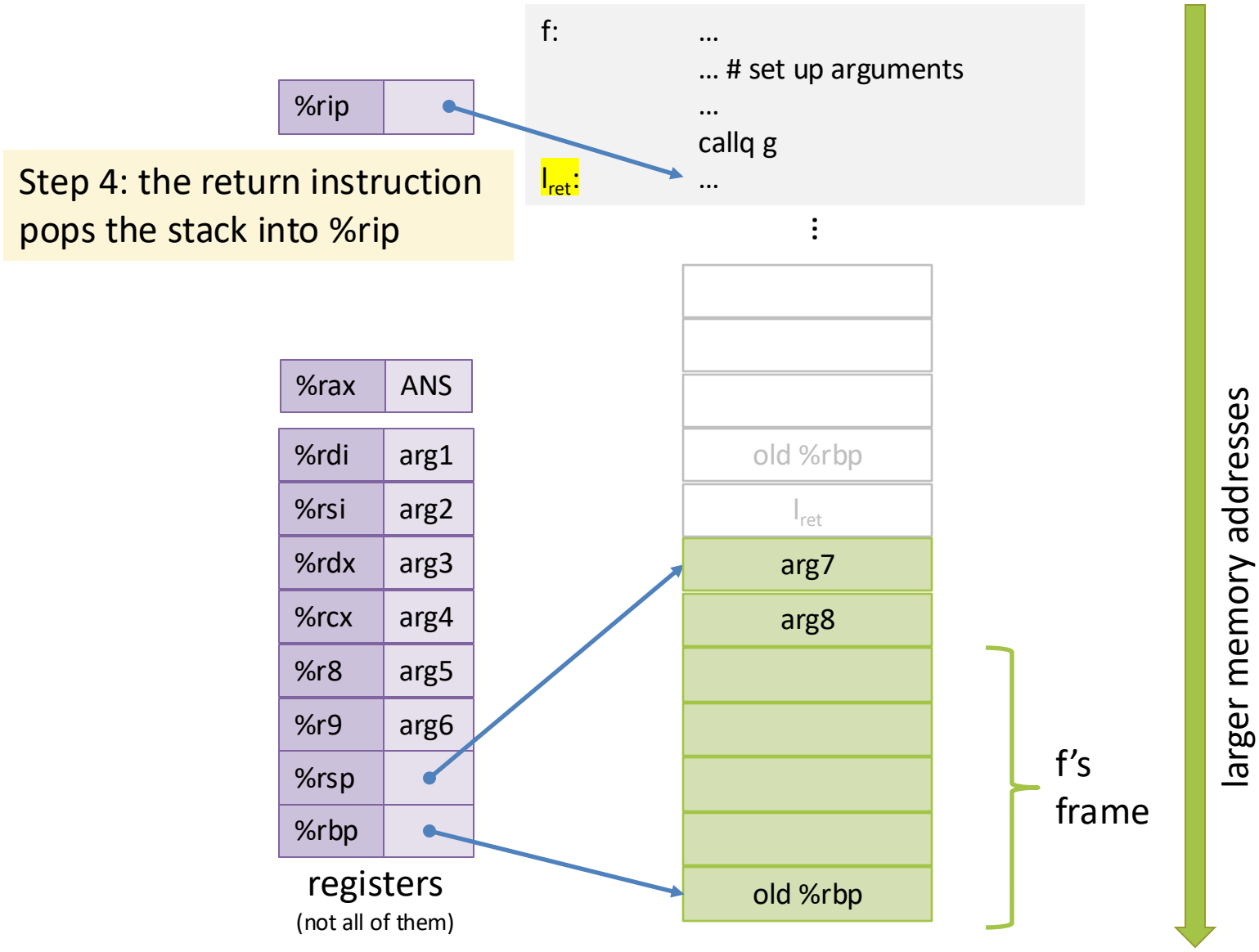
callee epilogue (return protocol)



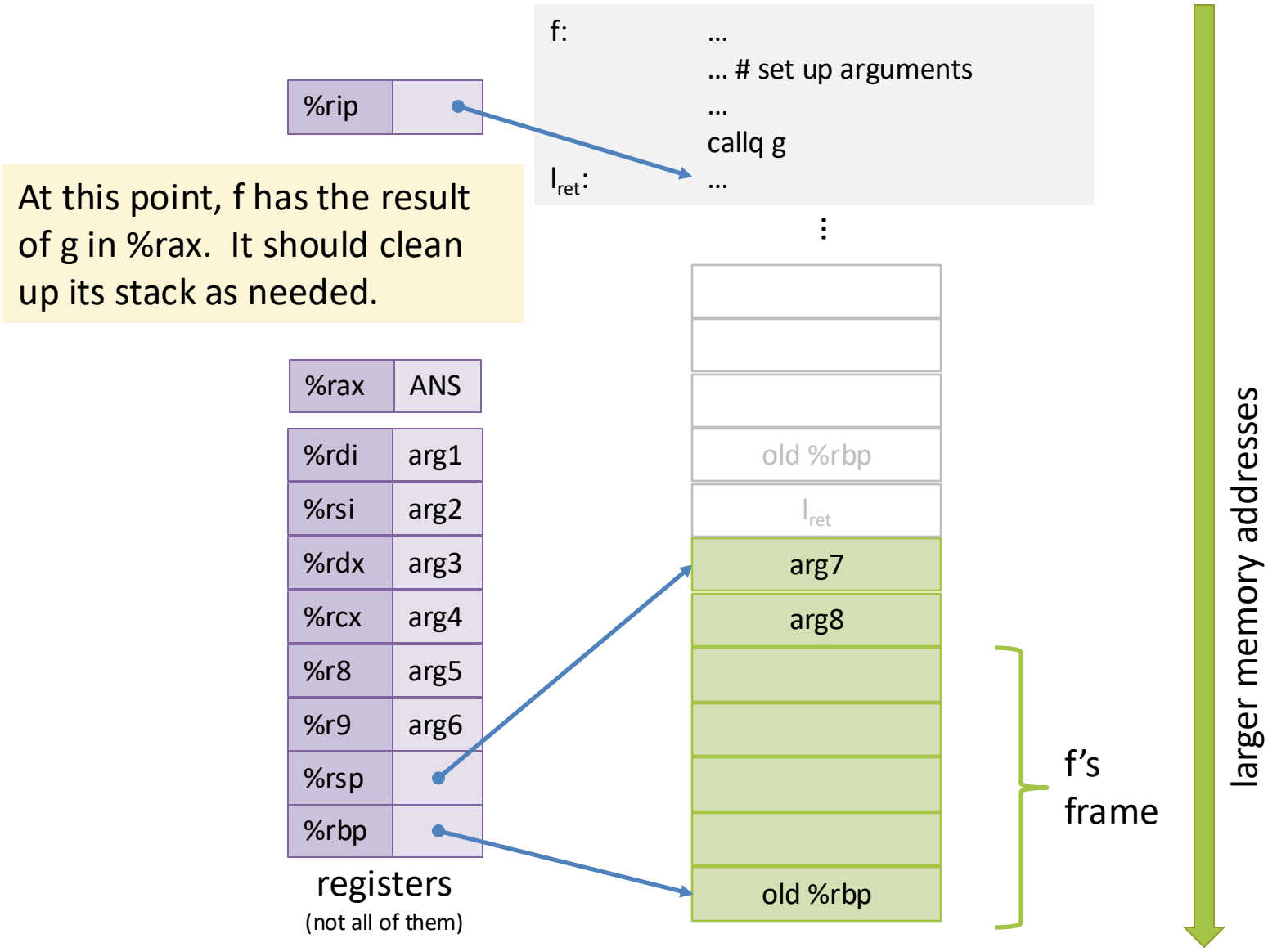
callee epilogue (return protocol)



callee epilogue (return protocol)



back in f


f:
...
... # set up arguments
...
callq g
l_{ret}:
...
%rip
At this point, f has the result of g in %rax. It should clean up its stack as needed.
%rax
ANS
%rdi
arg1
%rsi
arg2
%rdx
arg3
%rcx
arg4
%r8
arg5
%r9
arg6
%rsp
%rbp
registers
(not all of them)
old %rbp
l_{ret}
arg7
arg8
old %rbp
f's frame
larger memory addresses

X86-64 SYSTEM V AMD 64 ABI

- 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)