Lecture 4

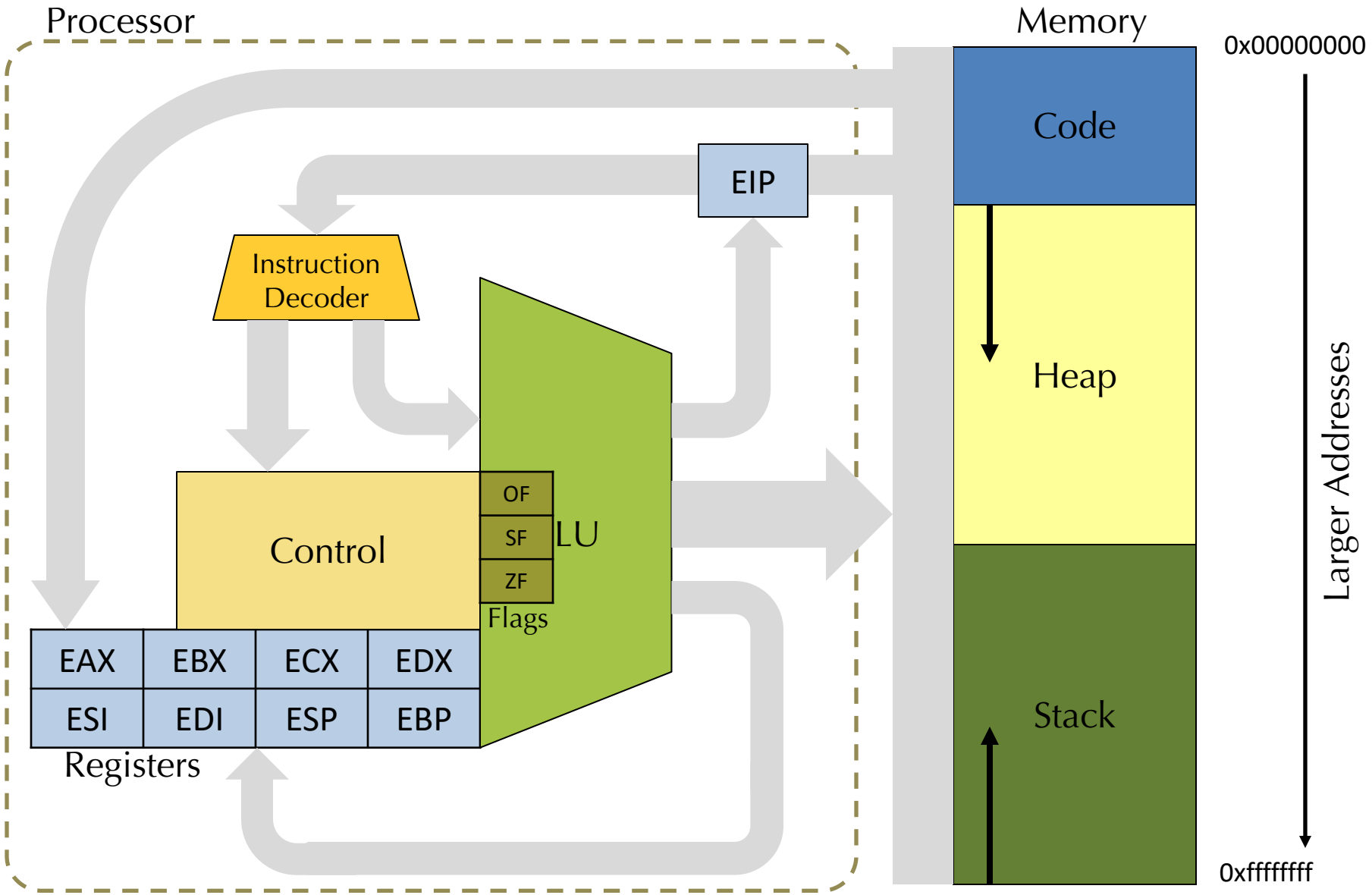
EECS 483:
COMPILER CONSTRUCTION

Announcements

- HW2: X86lite
  - Available on the course web pages now.
  - Due: Tues. Feb. 6th at 11:59pm
  - Partners are allowed, encouraged

  - 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

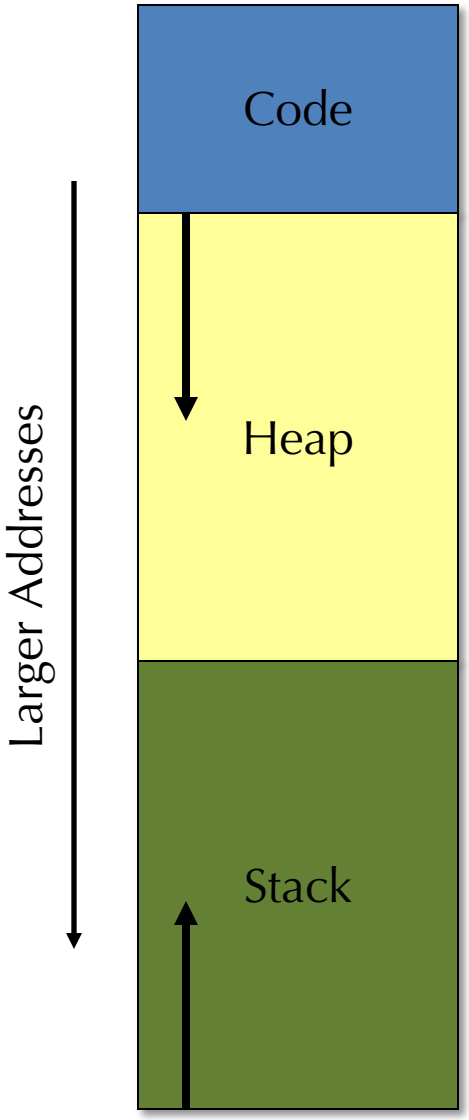
- Some compilers / architectures need "program" rather than "_program" for the entry label.
- If you want to debug in gdb:
  - Call gcc with the –g flag too
  - On Mac, use lldb: lldb ./test

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

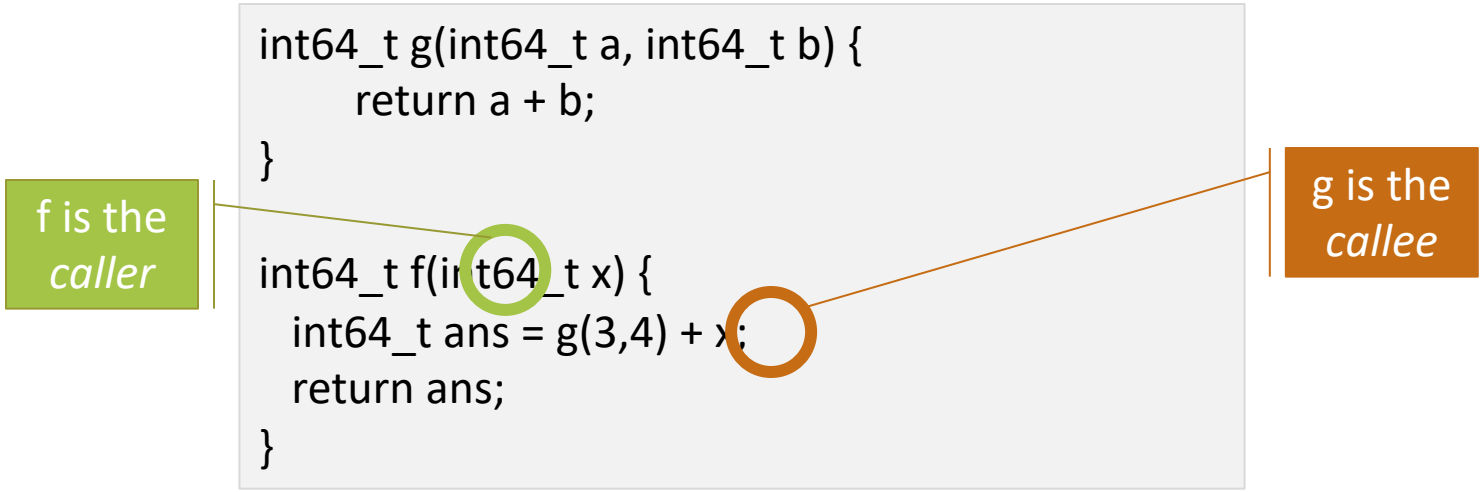
- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- 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*

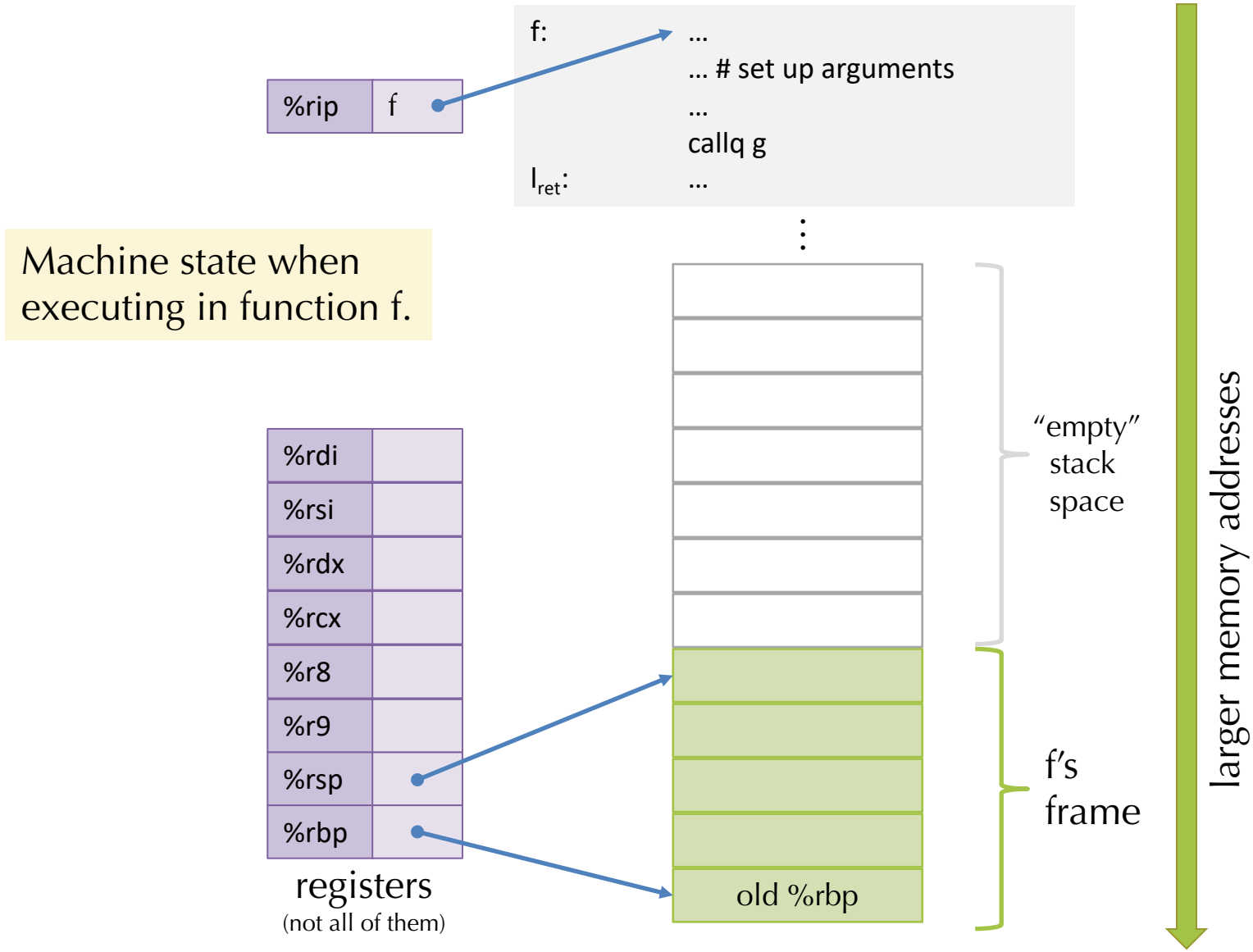
Calling Conventions

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

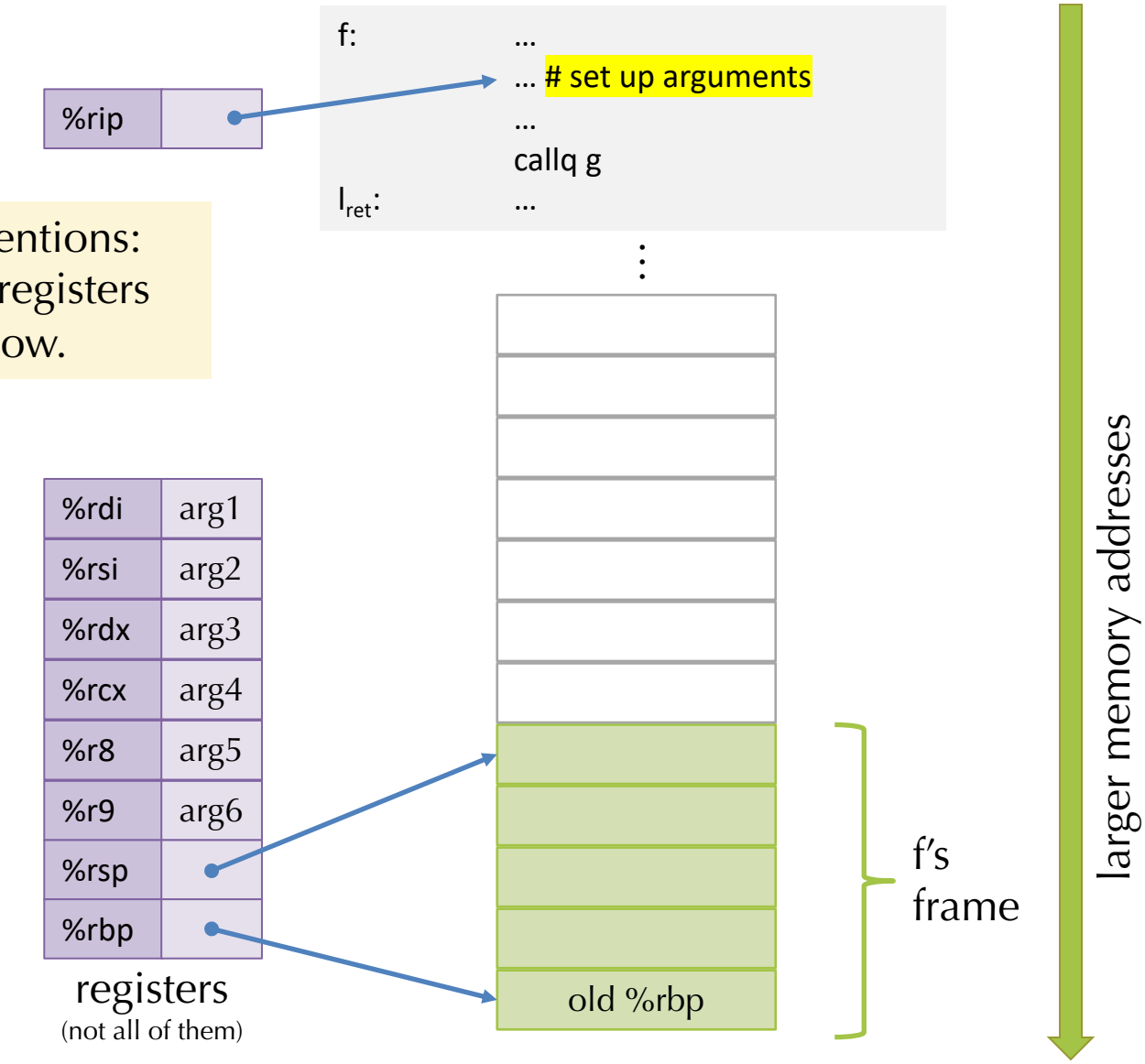


- Designate registers either:
  - Caller Save – *e.g.*, freely usable by the called code
  - Callee Save – *e.g.*, must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes variable arguments harder)

x64 Calling Conventions: Caller Protocol

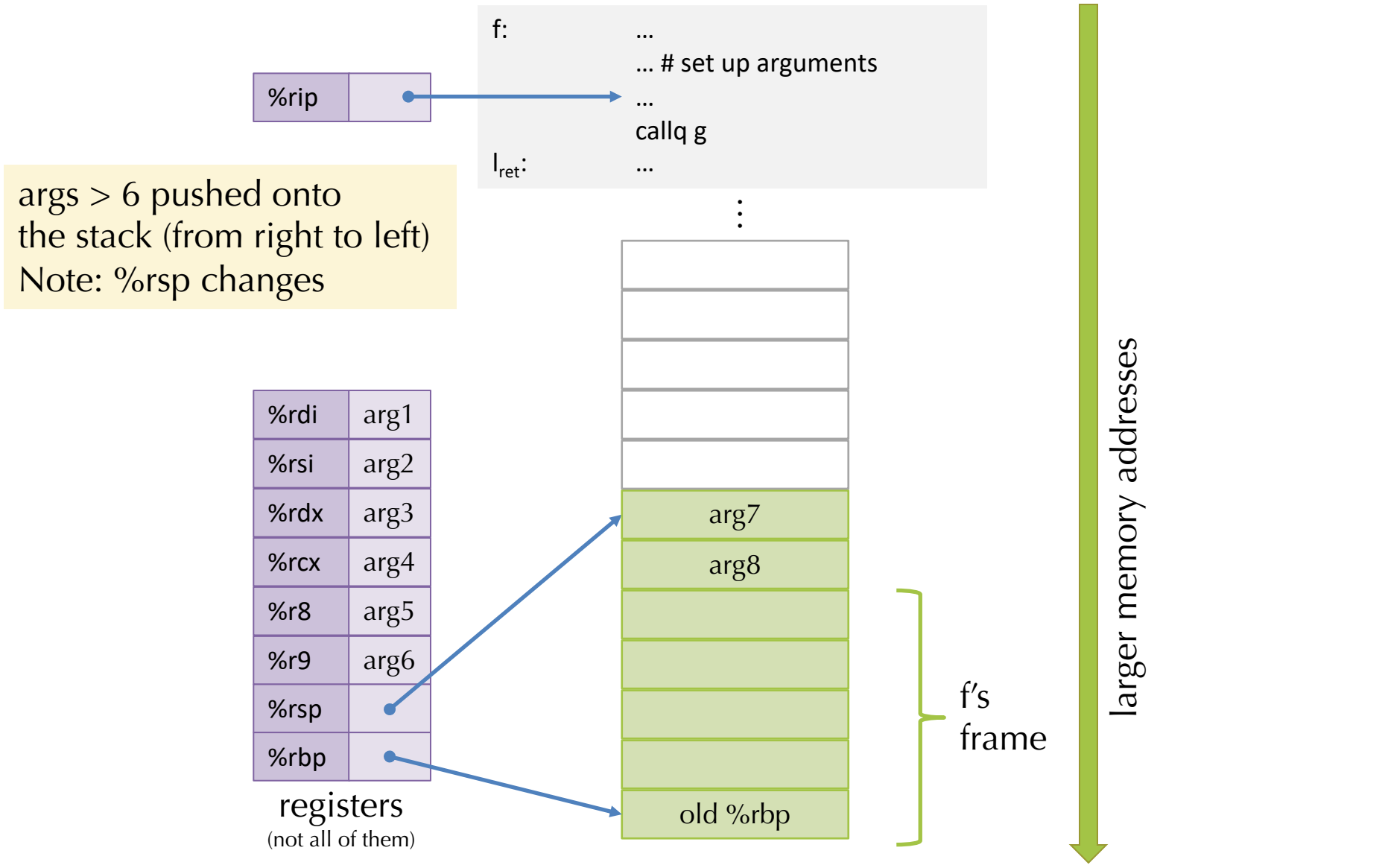


x64 Calling Conventions: Caller Protocol

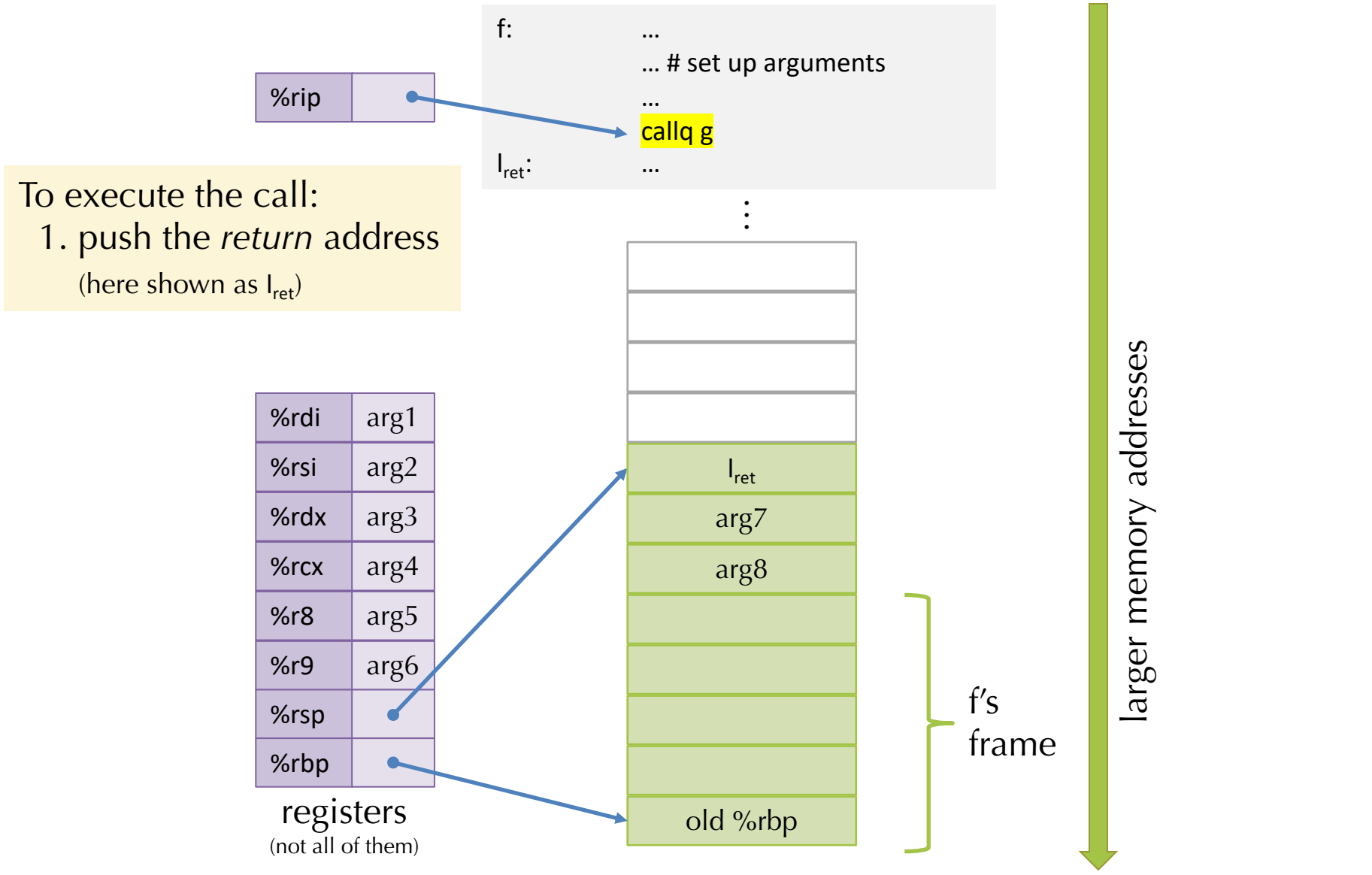


Calling conventions: args 1…6 in registers as shown below.

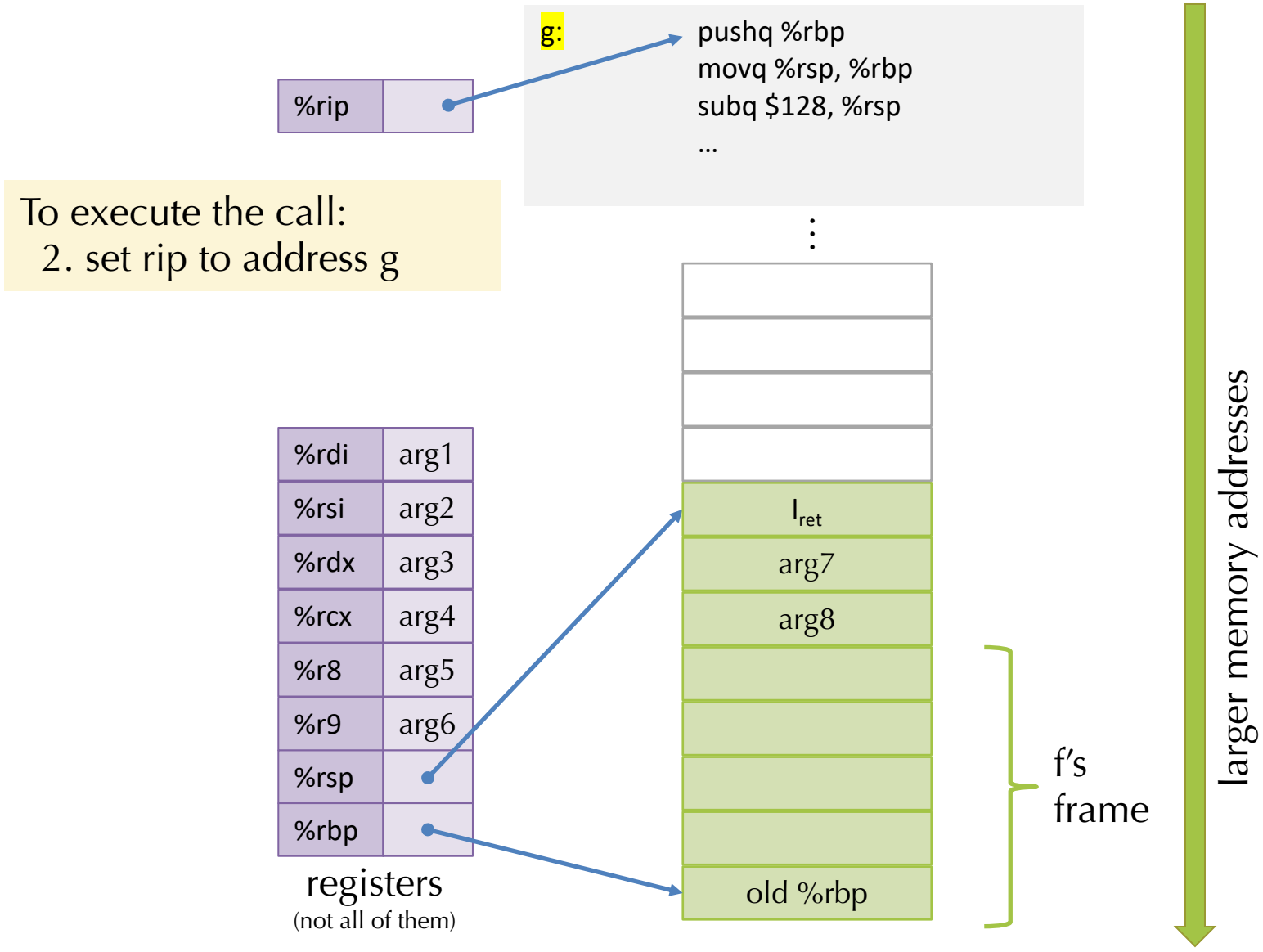
x64 Calling Conventions: Caller Protocol



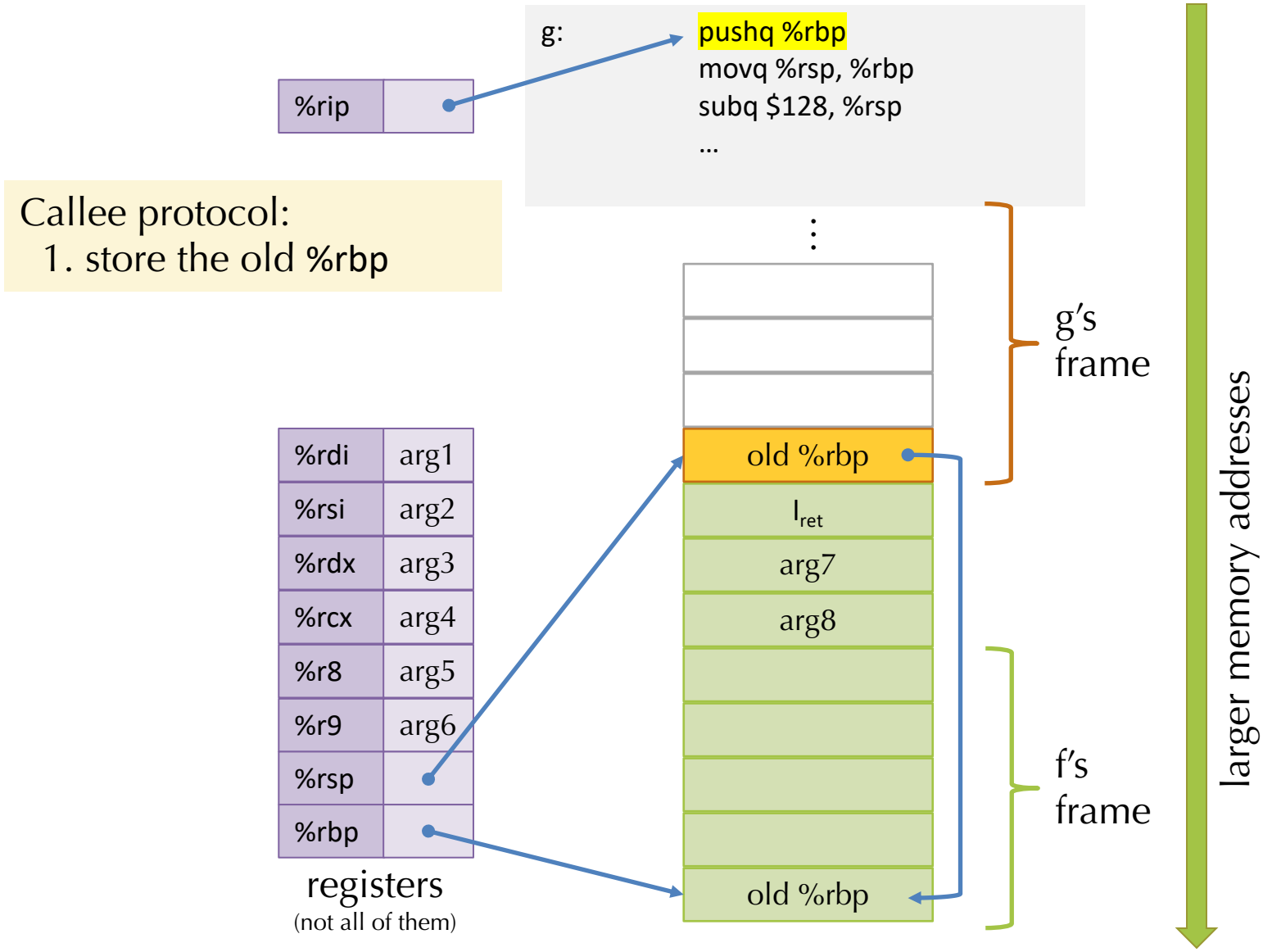
call instruction



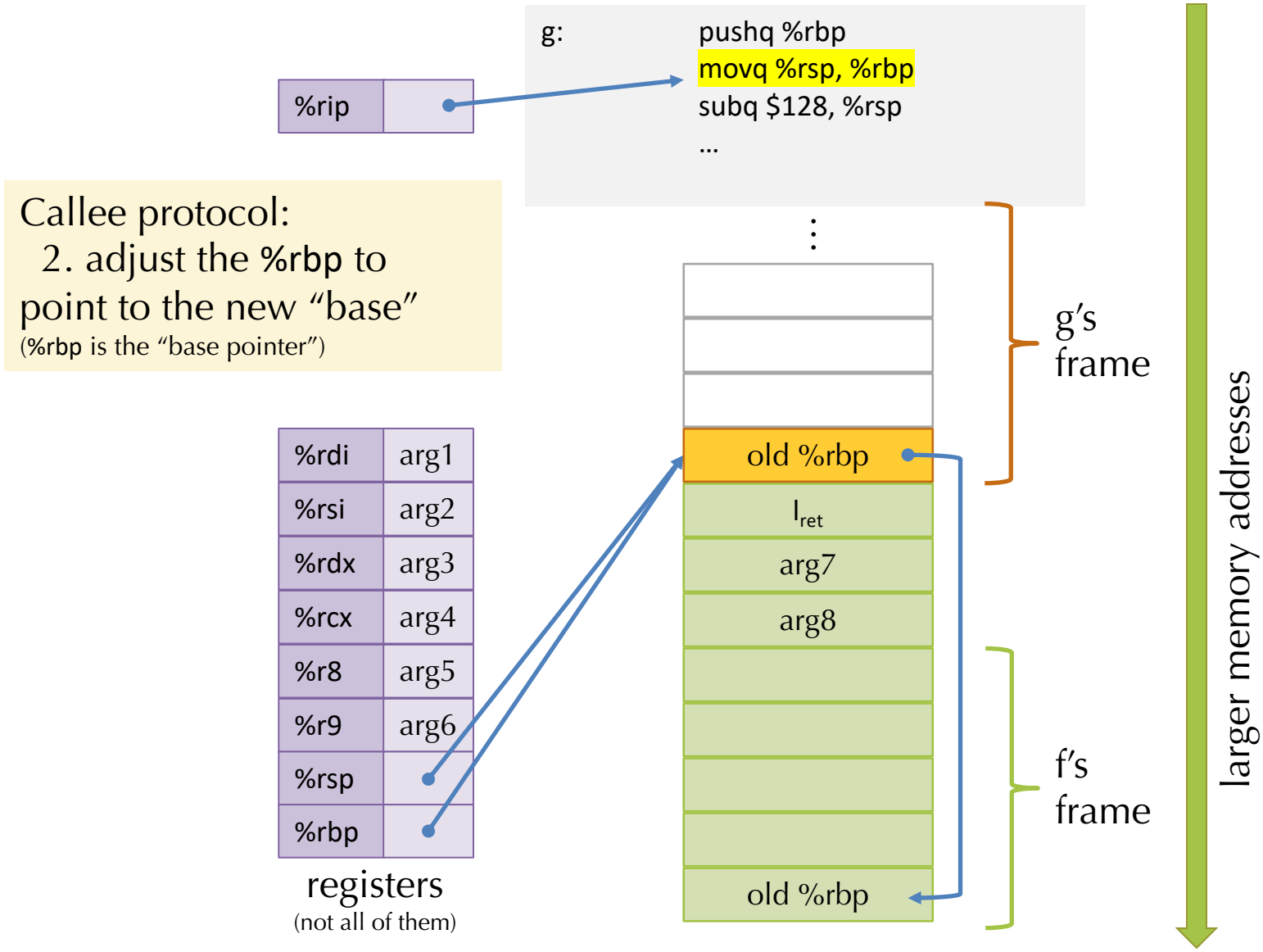
call instruction



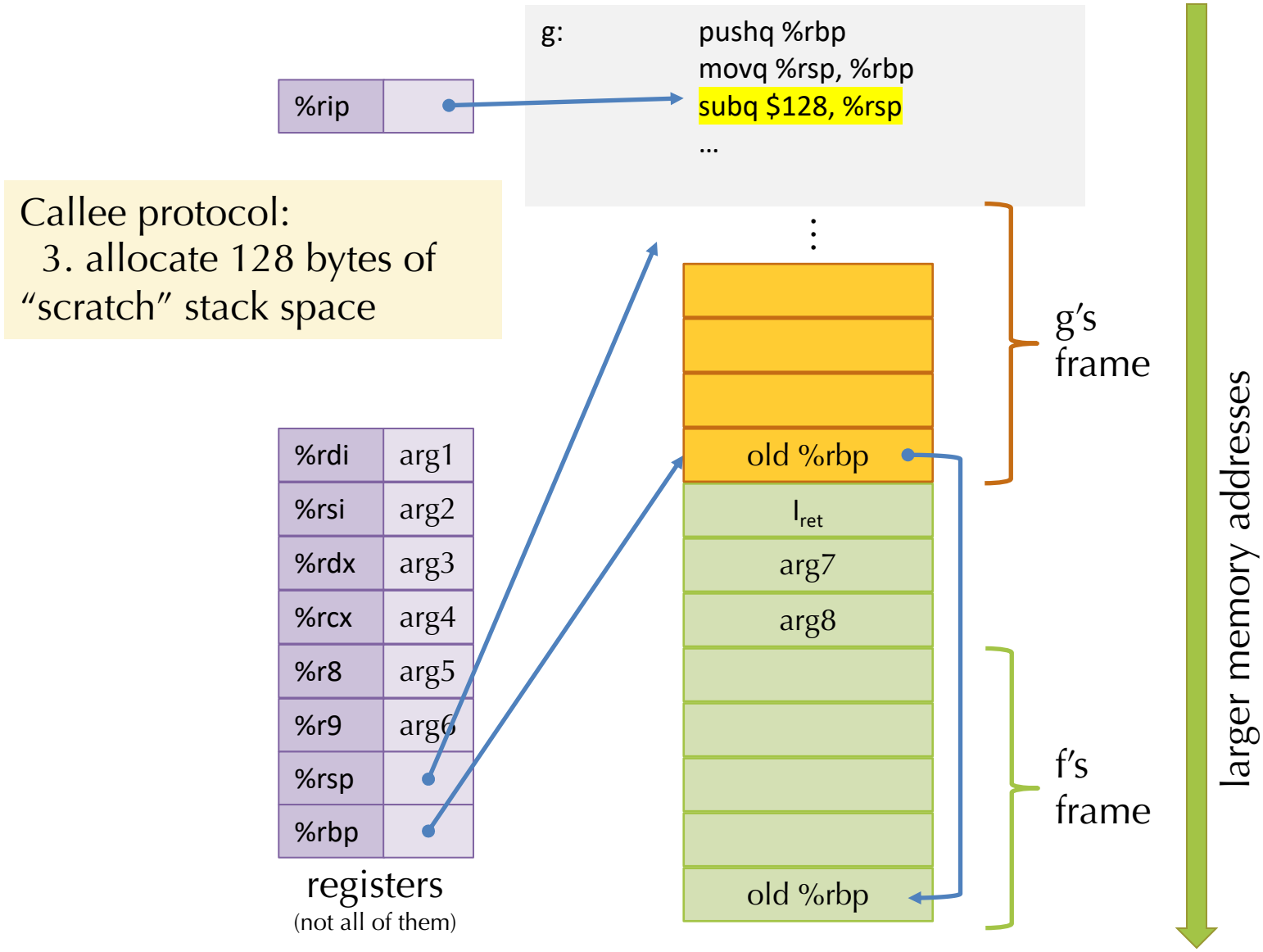
callee function prologue



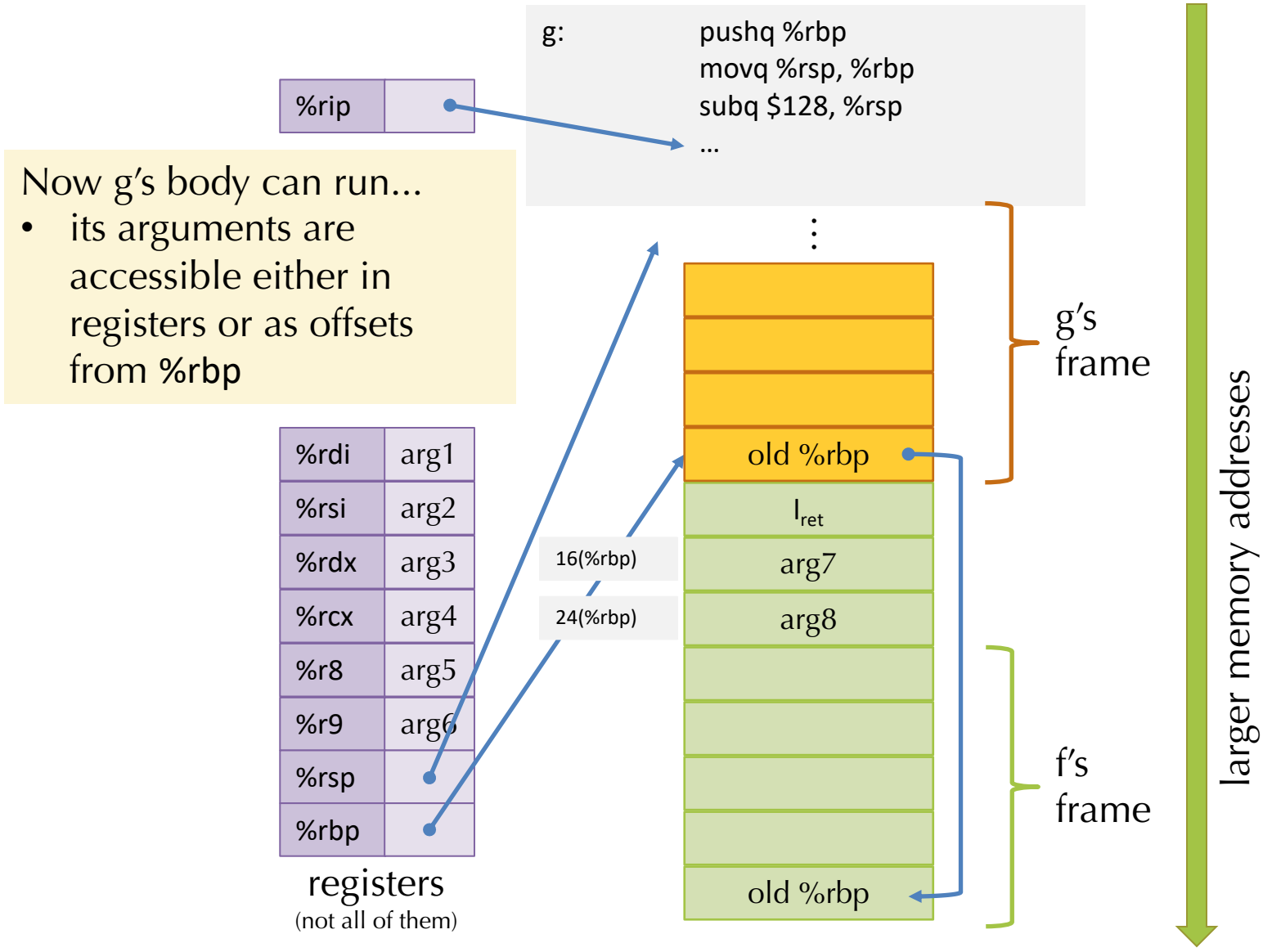
callee function prologue



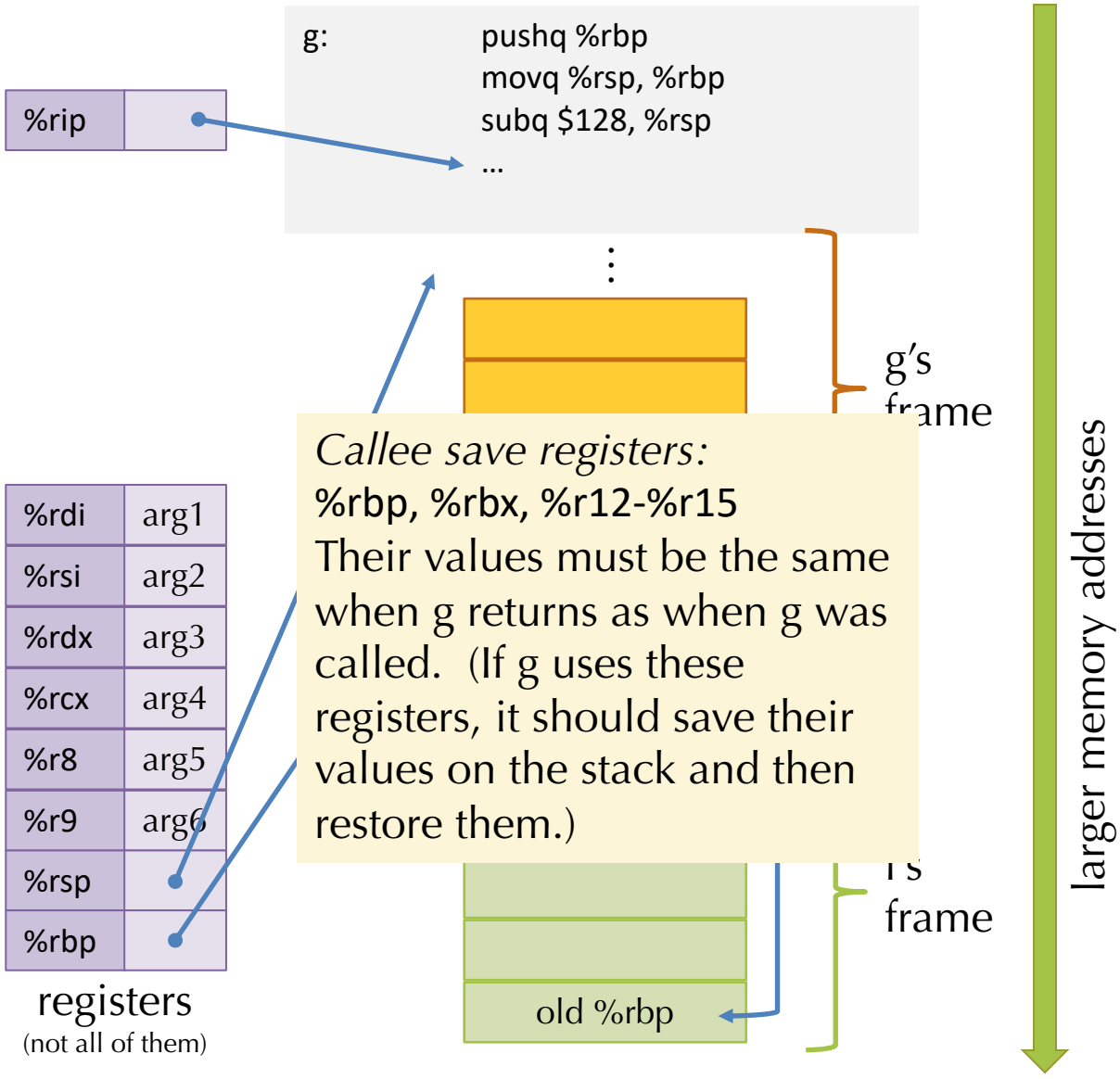
callee function prologue



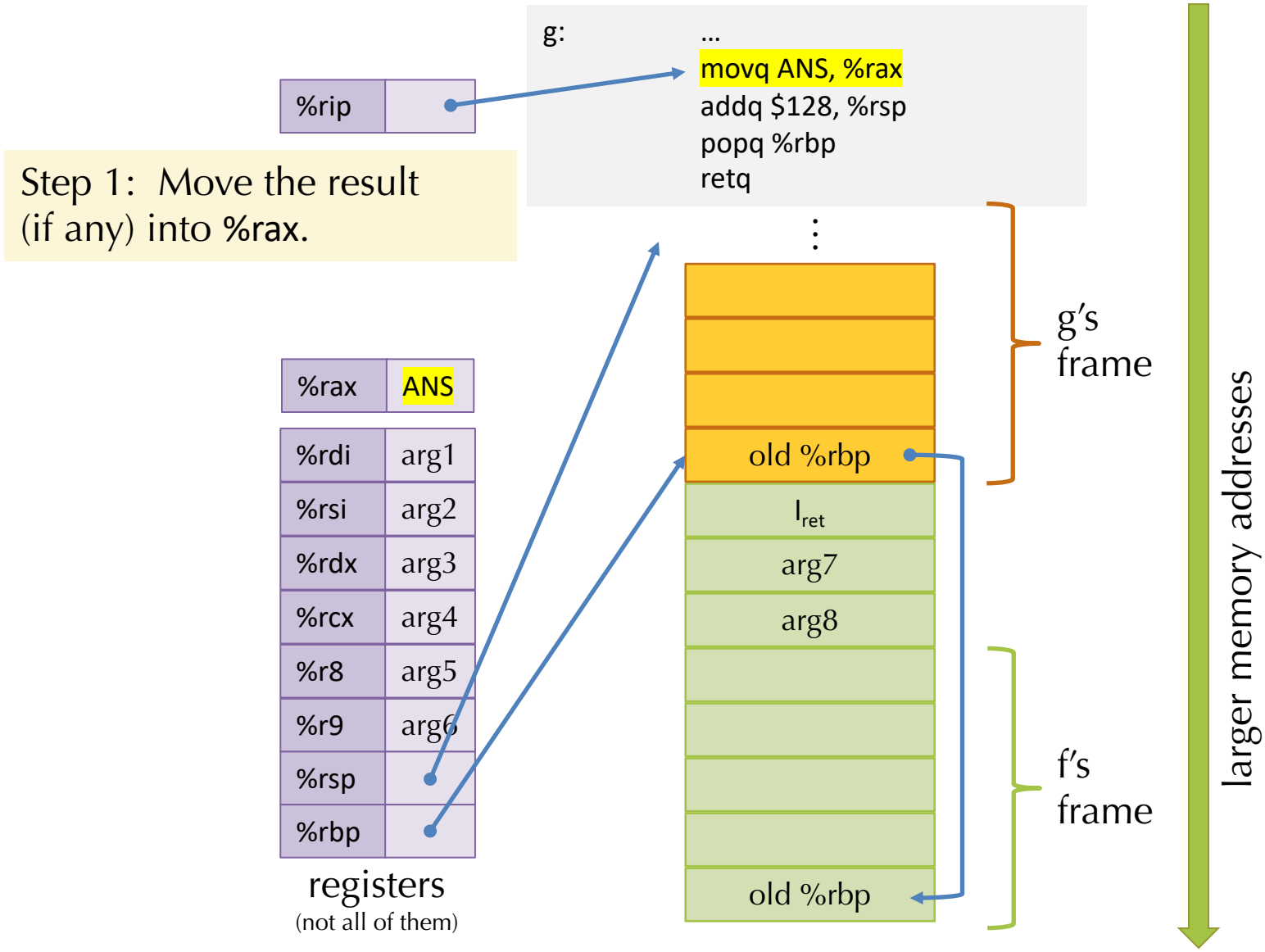
callee invariants: function arguments



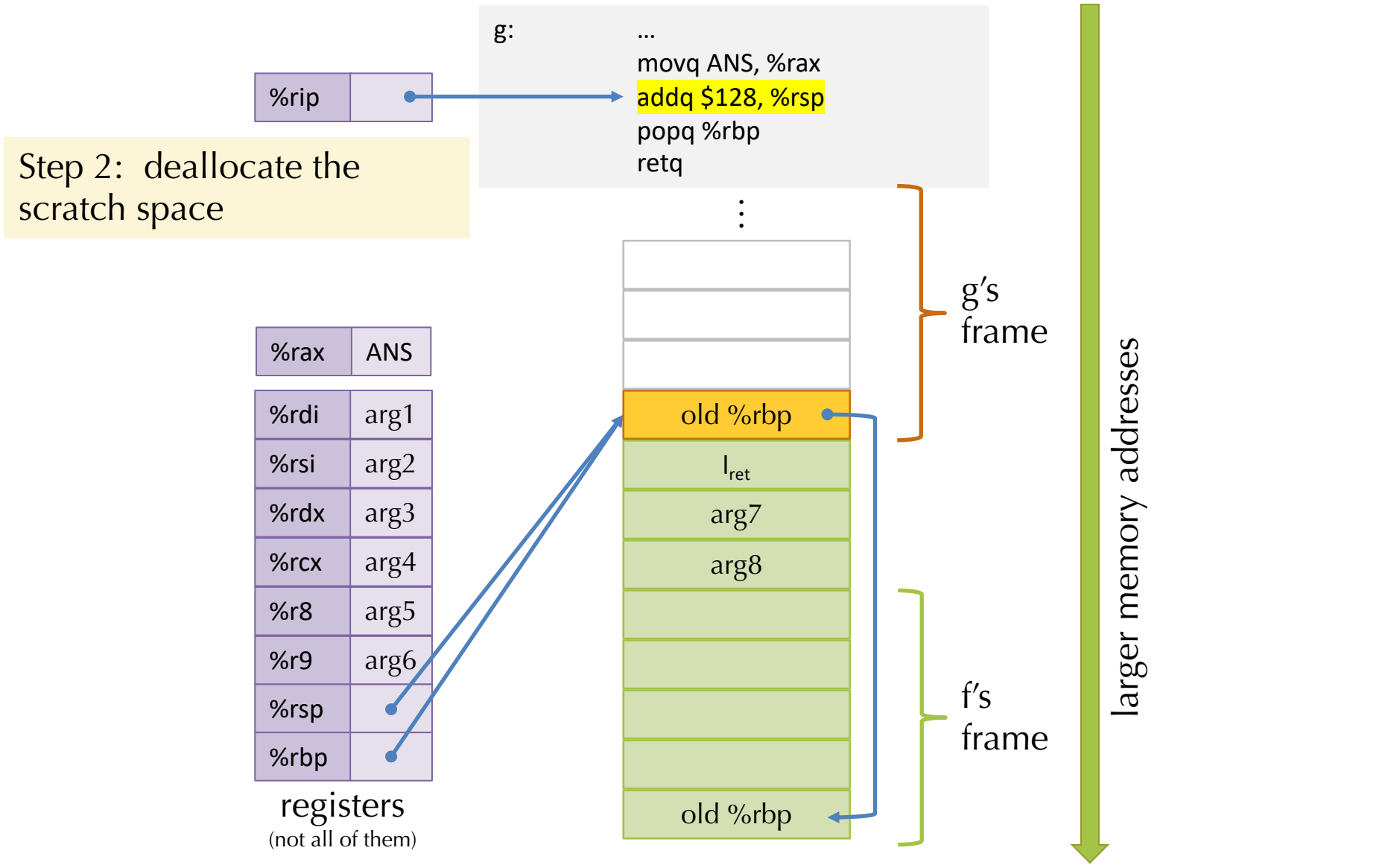
callee invariants: callee save registers



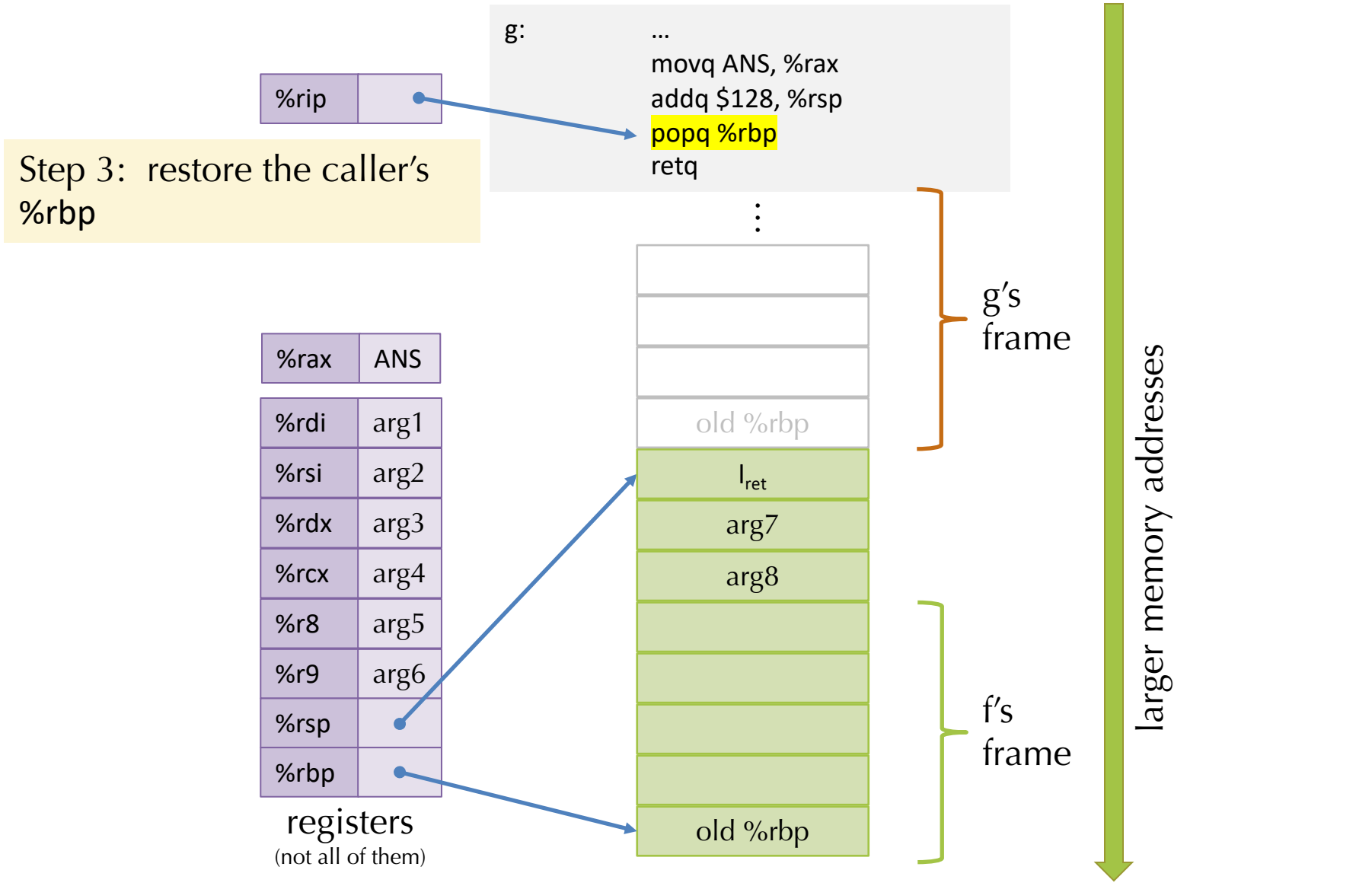
callee epilogue (return protocol)



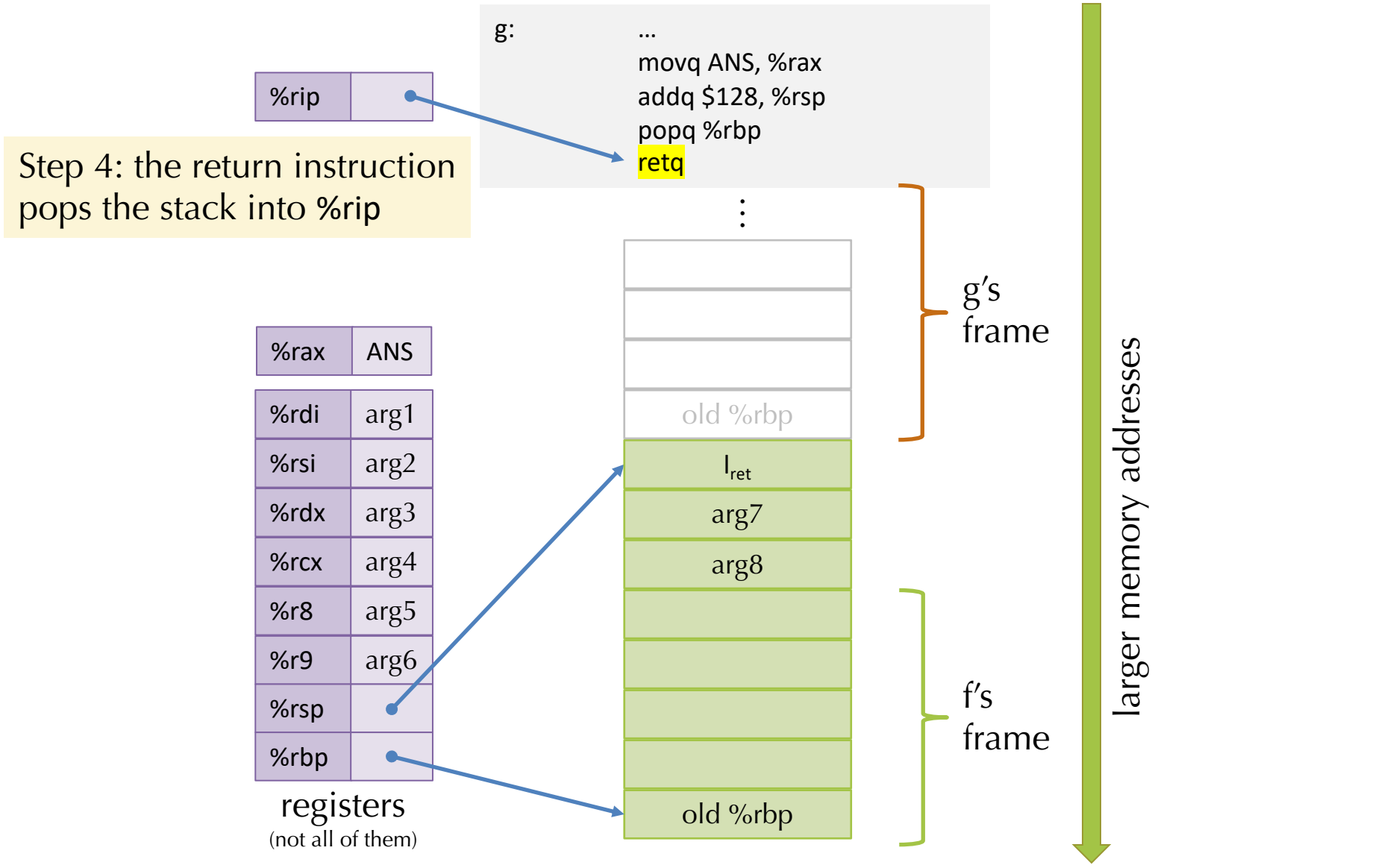
callee epilogue (return protocol)



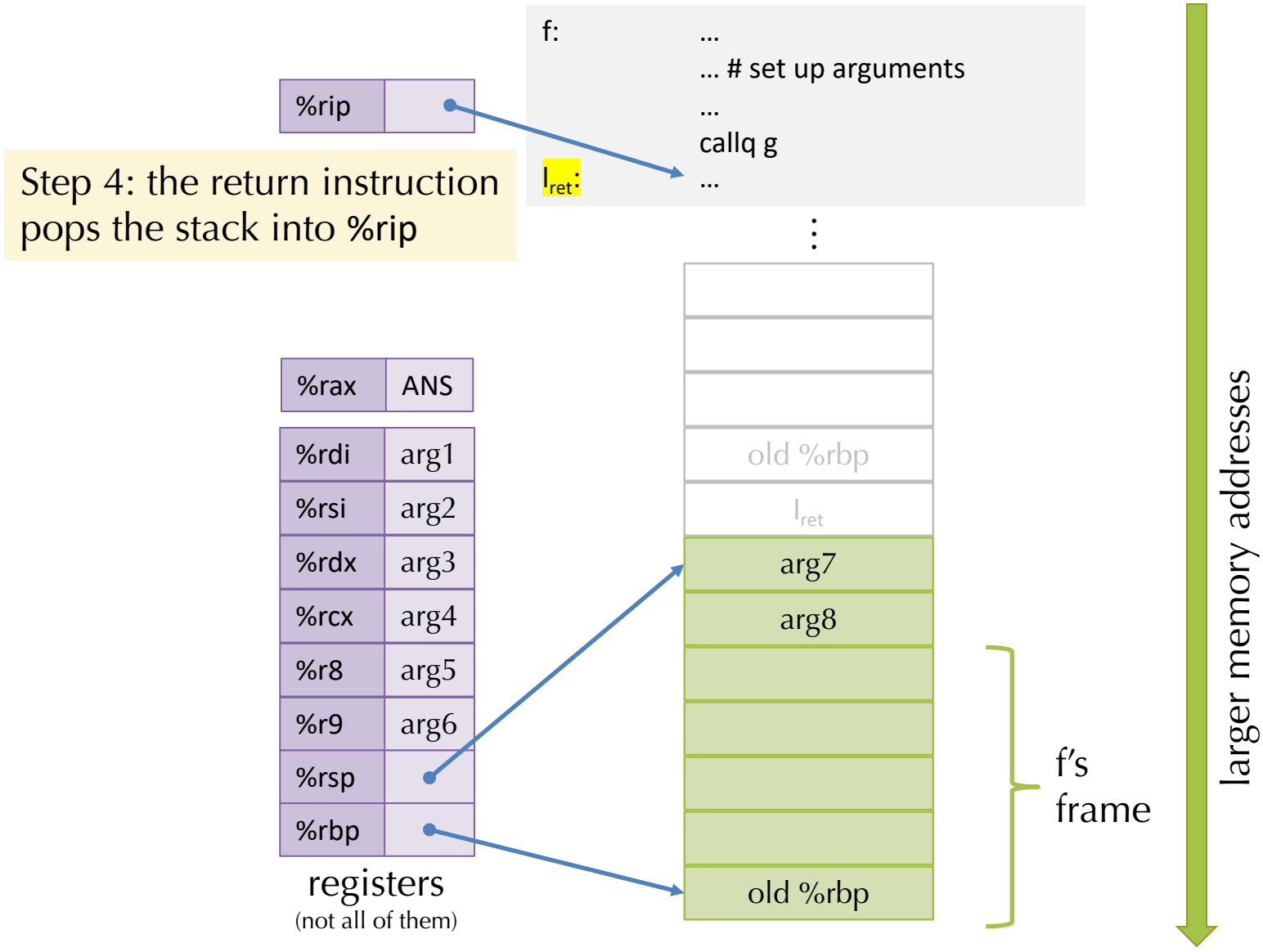
callee epilogue (return protocol)



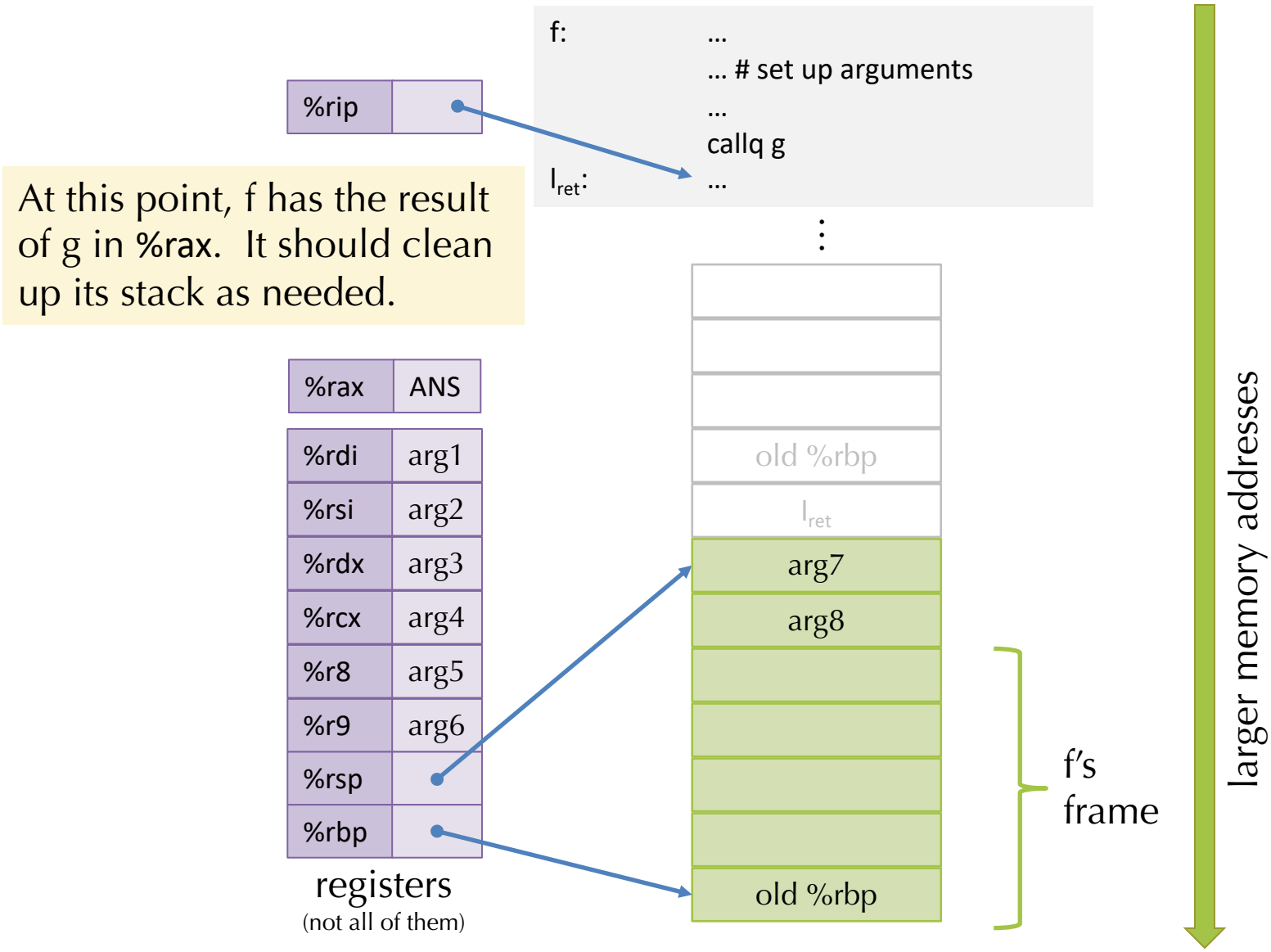
callee epilogue (return protocol)



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

- 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)