CS4212: Compiler Design

Week 3: Compiling Function Calls to x86; Intermediate Representations

Ilya Sergey

ilya@nus.edu.sg

ilyasergey.net/CS4212/

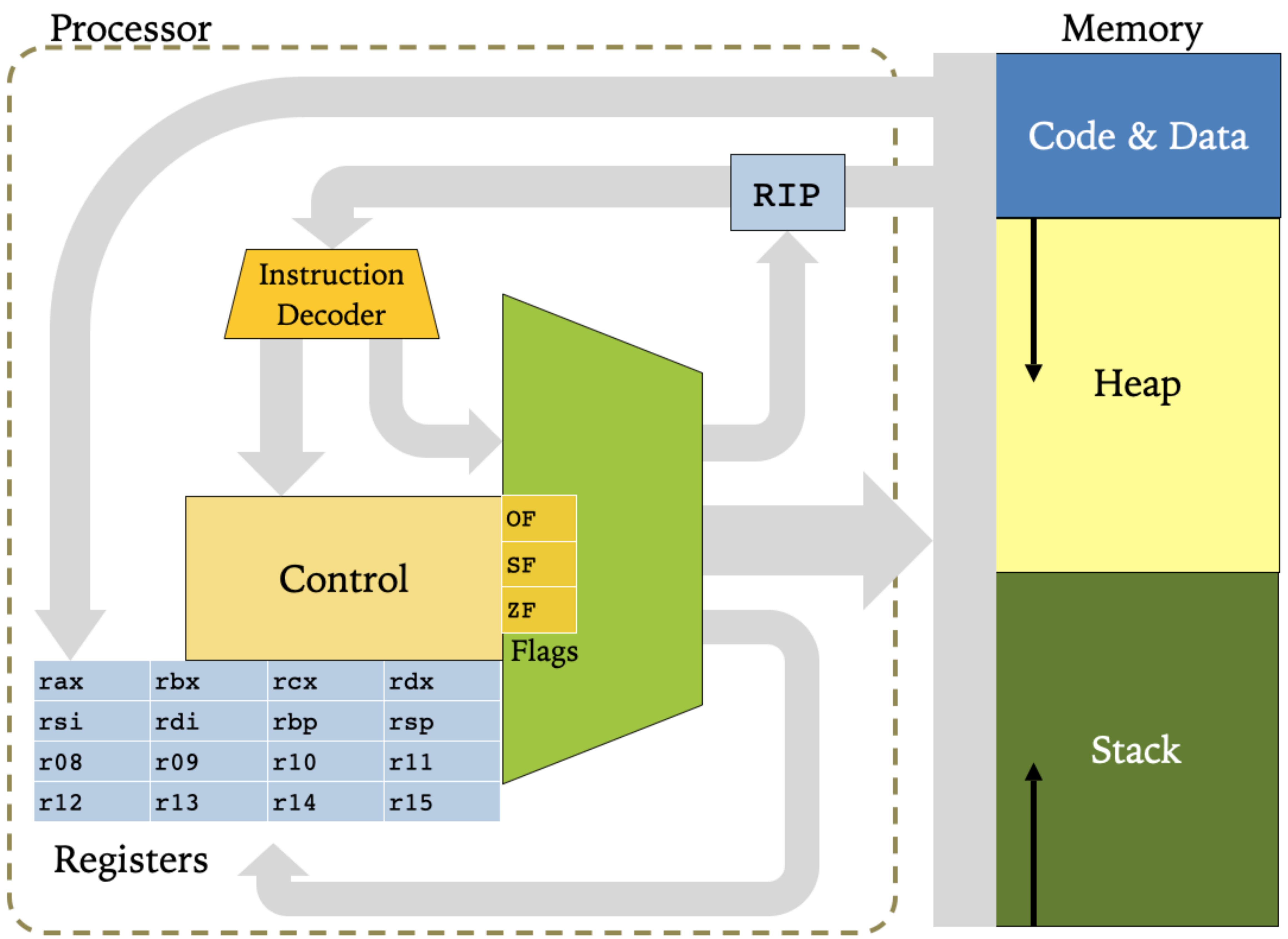
Comparisons and Conditioning

X86lite State: Flags & Condition 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*
  - You can think of Cond. Codes as of additional registers, whose value changes depending on the current flags

- E.g., `cmpq` SRC1, SRC2 computes SRC1 – SRC2 to set the flags
- Now we can check conditional codes:
  - `eq` equality holds when `ZF` is set
  - `neq` inequality holds when (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`
  - `gt` greater than holds when (not le) holds,
    - i.e. (SF = OF) && not(ZF)



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 do nothing

• Example:

`cmpq %rcx, %rax`	Compare `rax` to `ecx`
`jeq __truelbl`	If `rax` = `rcx` then jump to `__truelbl`

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).

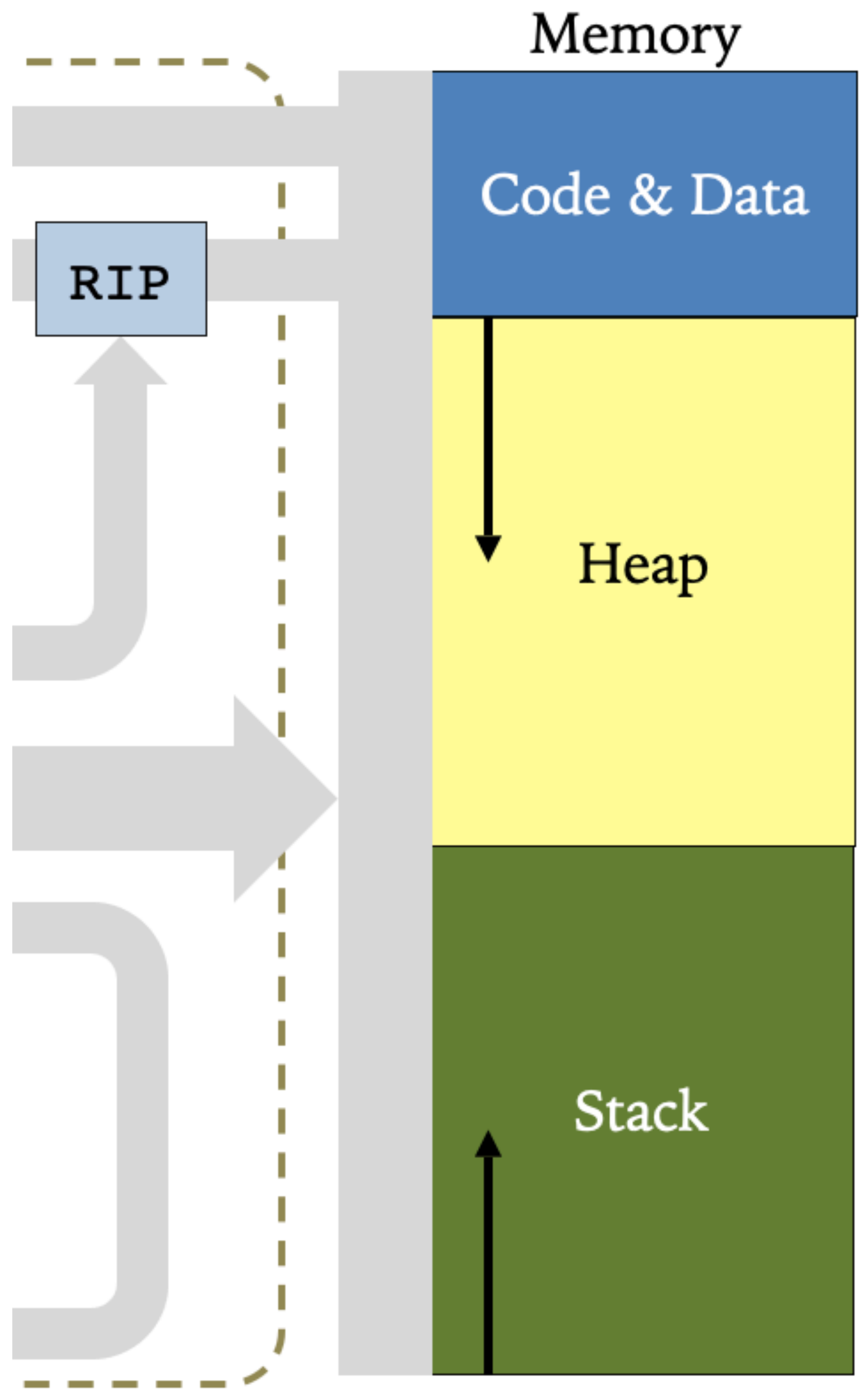
- Labels are translated away by the linker and loader – instructions live in the heap in the "code segment"

- An X86 program begins executing at a designated code label (usually "main").

Basic Control Flow

Jumps, Calls, 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



Loop-based Factorial in Assembly

```
.globl  _program
_program:
    movq    $1, %rax
    movq    $6, %rdi
loop:
    cmpq    $0, %rdi
    je  exit
    imulq   %rdi, %rax
    decq    %rdi
    jmp loop
exit:
    retq
```

```
int program() {
  int acc = 1;
  int n   = 6;
  while (0 < n) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

Demo: Hand-Coded x86Lite

- https://github.com/cs4212/week-02-x86lite
- Basic definitions: `x86.ml`
- Linking with assembly: `test.c`
- Example program, simple output, factorial

Compiling, Linking, Running

- To use hand-coded X86:
  1. Compile OCaml program `main1.ml` to the executable by running make

  2. Run it, redirecting the output to some .s file, e.g.:
     ```
     ./main1.native >> prog.s
     ```

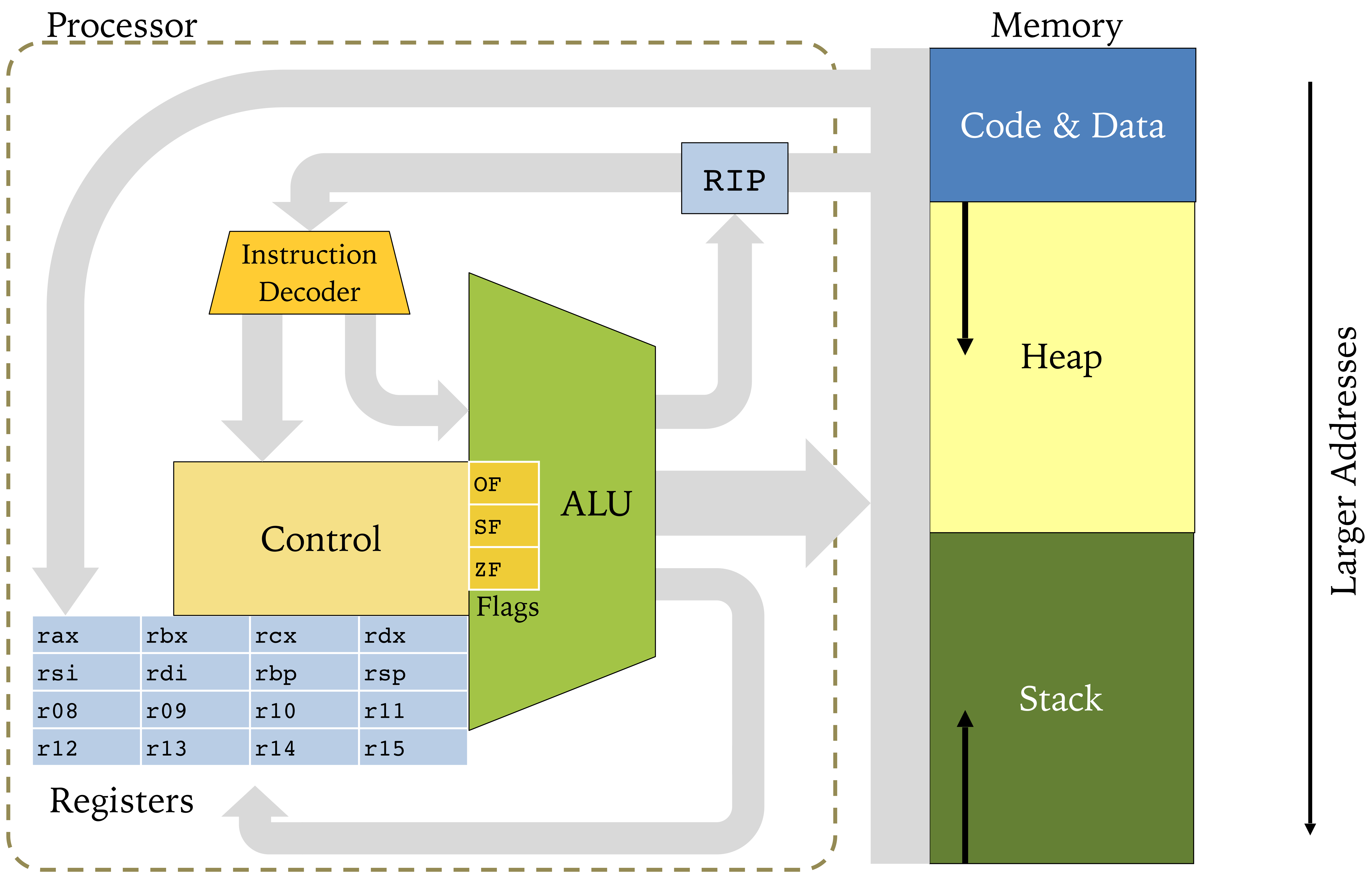
  3. Use clang (or gcc) to compile & link with `test.c:`
     ```
     clang -o test test.c prog.s
     ```

     One M1/M2 (Apple Silicon) Mac, use the following flags:
     ```
     clang -arch x86_64 -o test prog.s test.c
     ```

  4. You should be able to run the resulting executable:
     ```
     ./test
     ```

Implementing Functions & C Calling Conventions

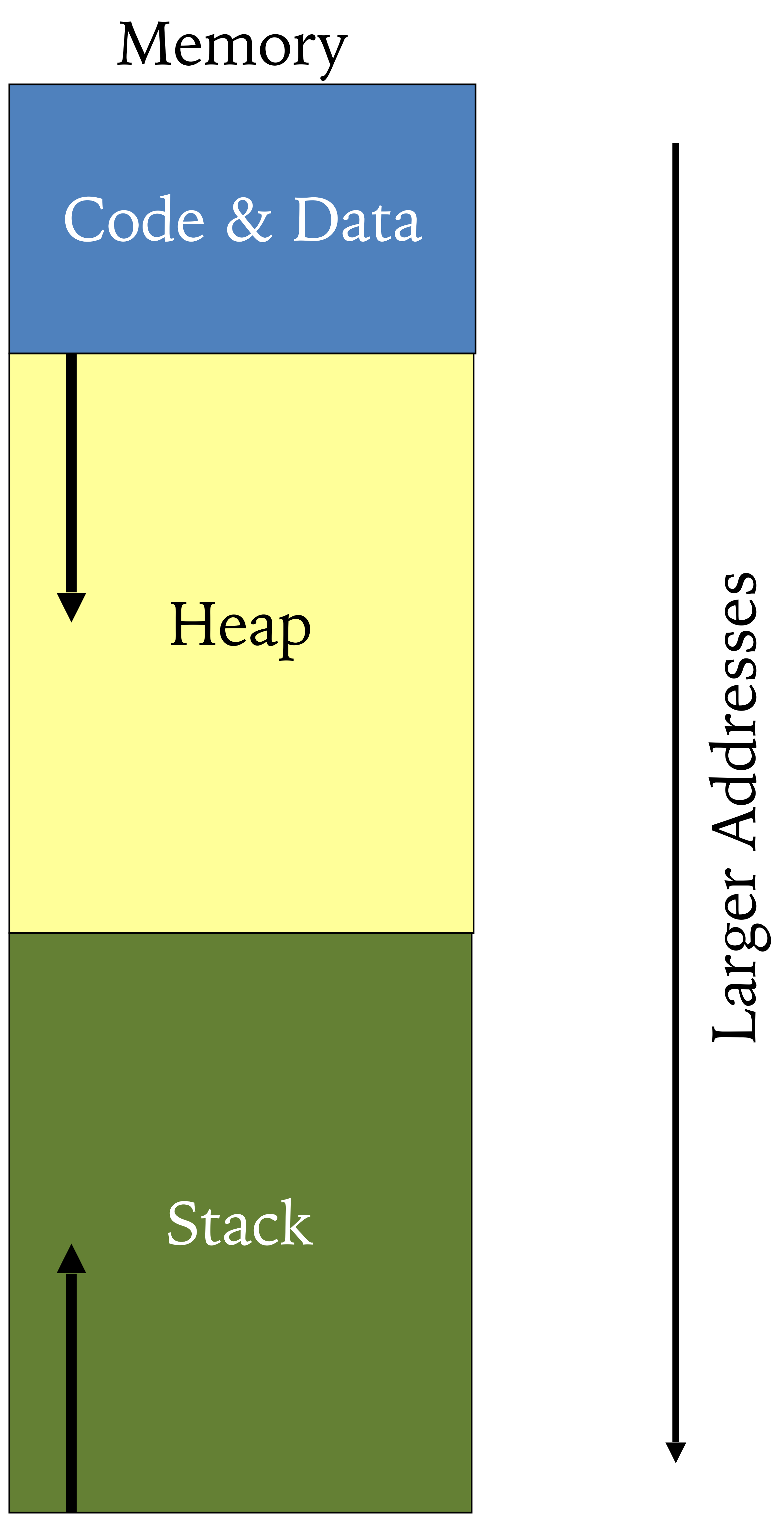
X86 Schematic



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 (e.g., only 16 in x86Lite)
  - Memory: slow, very large amount of space (2GB or more)
    - 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;
}
```

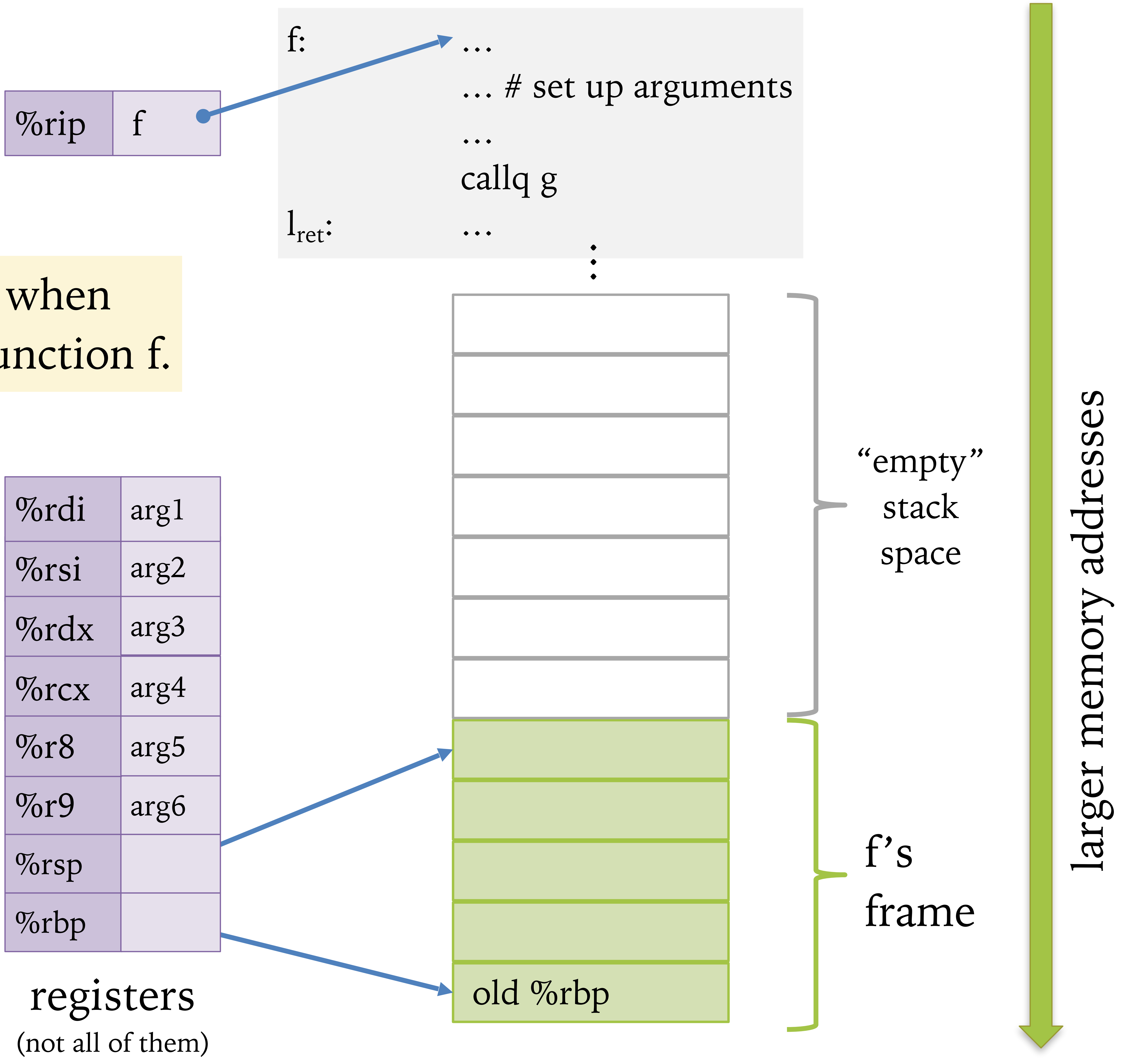
f is a caller

g is a 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 number arguments harder — the callee doesn't know how many are those)

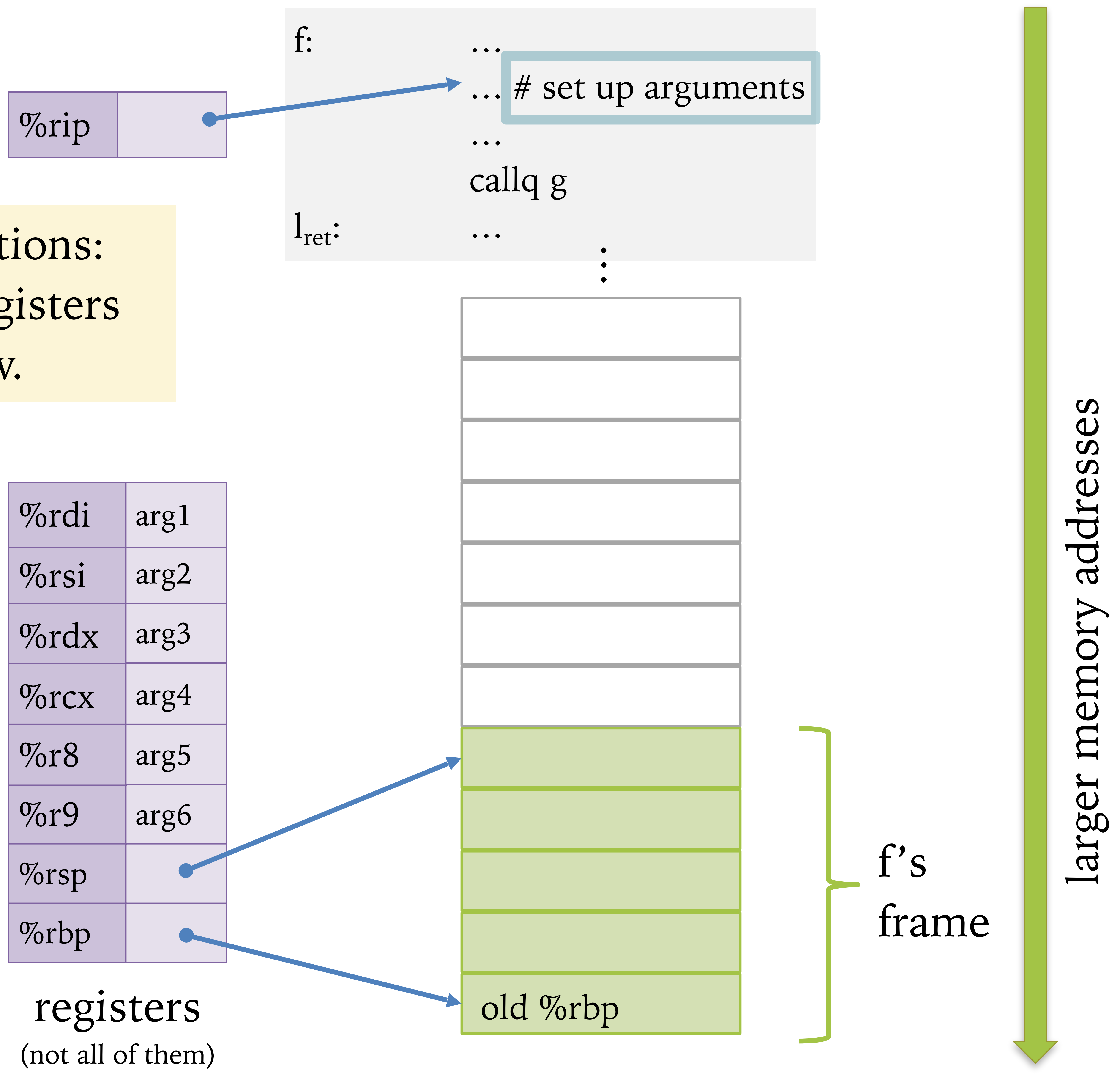
x64 Calling Conventions: Caller Protocol

Machine state when executing in function f.



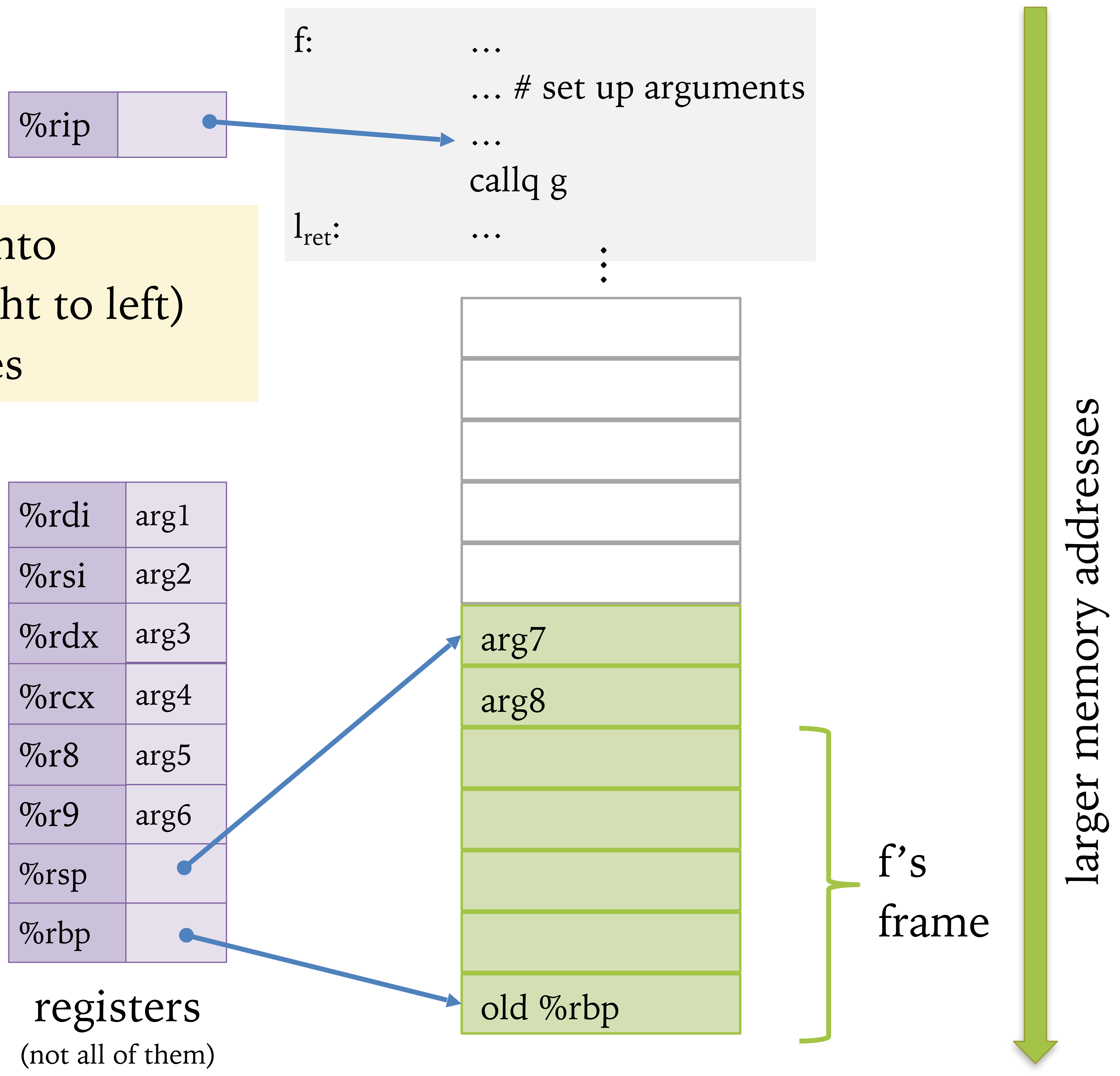
x64 Calling Conventions: Caller Protocol

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

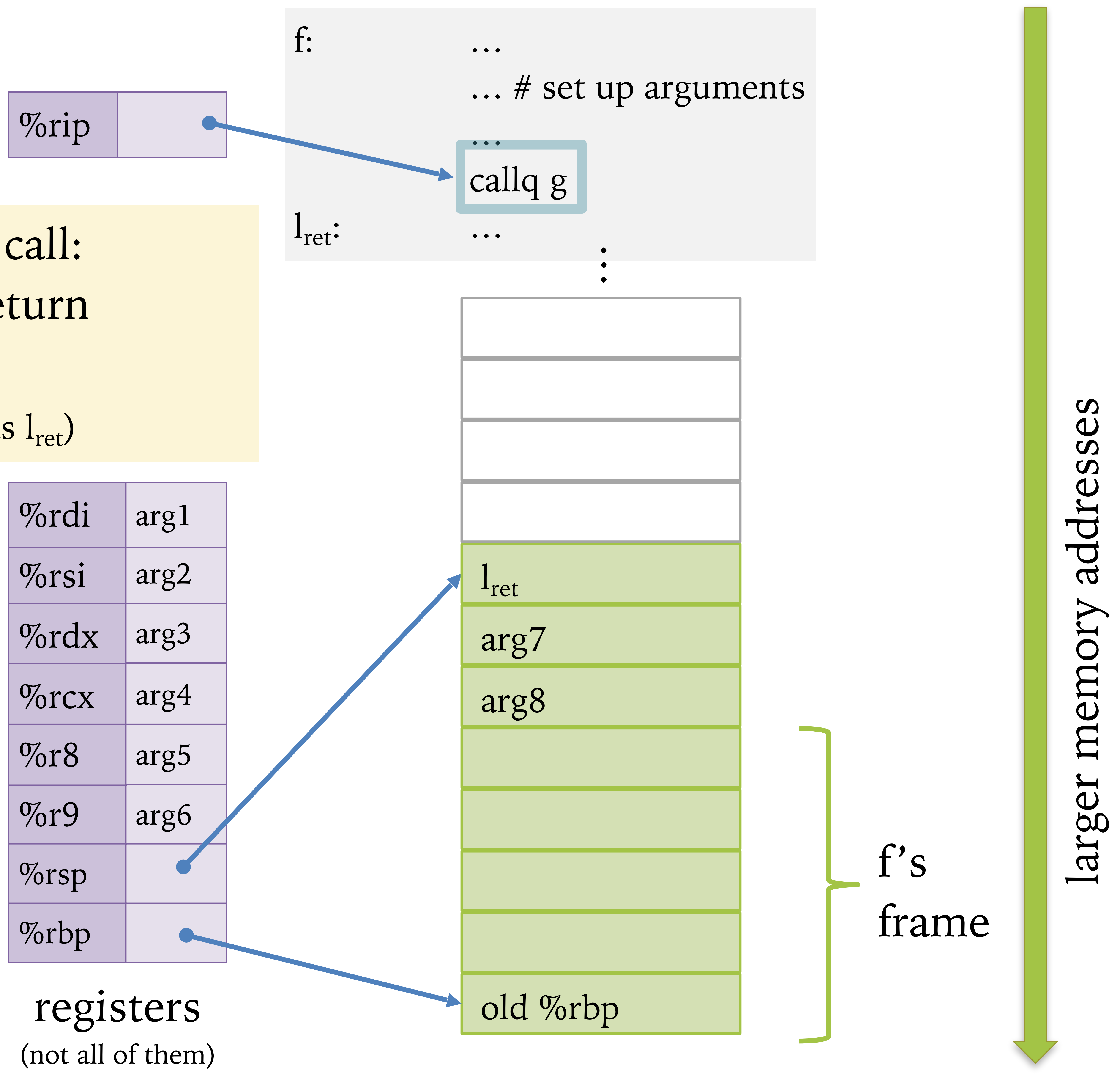


x64 Calling Conventions: Caller Protocol

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



Call Instruction



To execute the call:

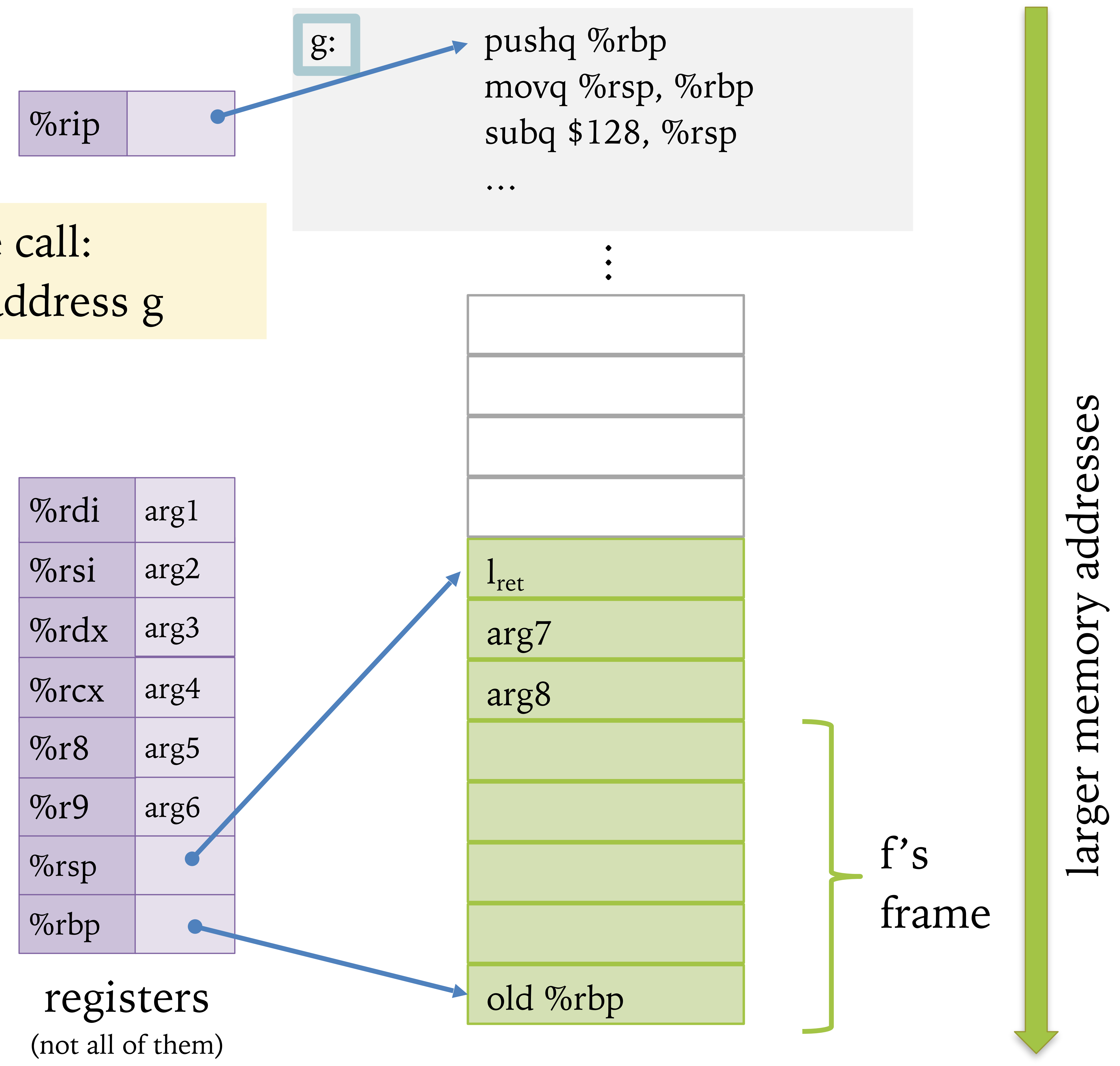
1. push the return address

(here shown as l_{ret})

Call Instruction

To execute the call:

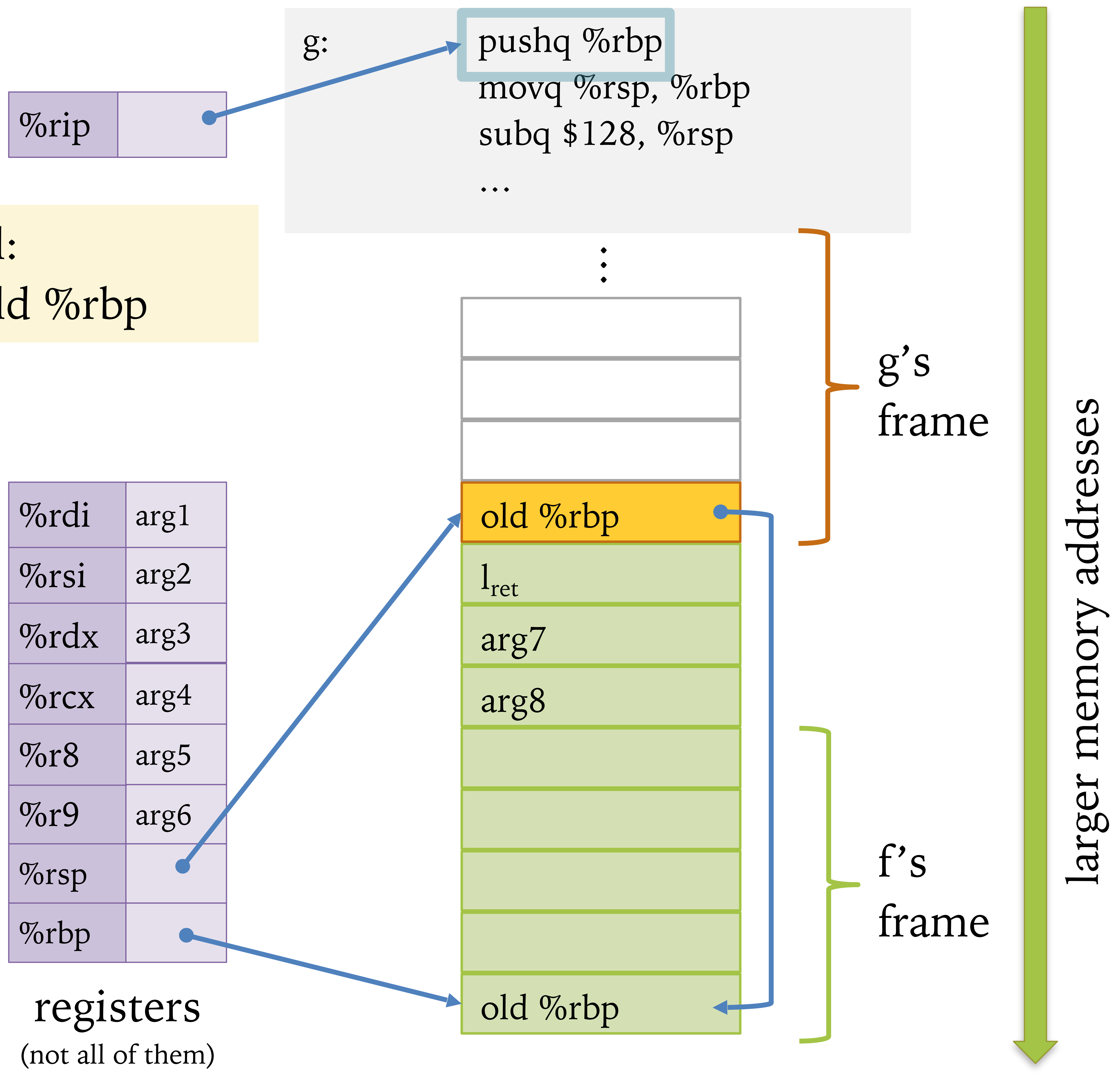
2. set rip to address g



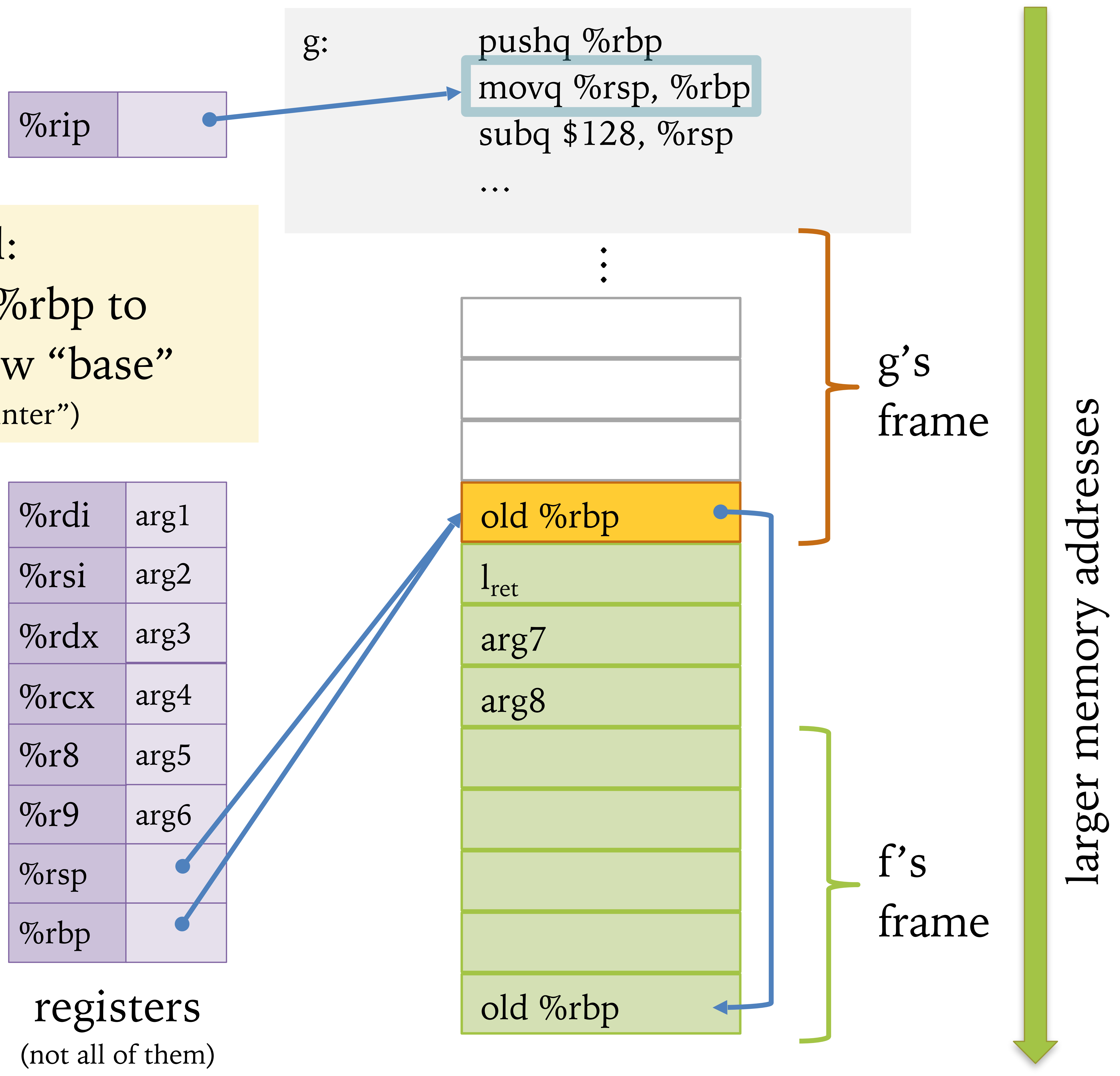
Callee Function Prologue

Callee protocol:

1. store the old %rbp



Callee Function Prologue



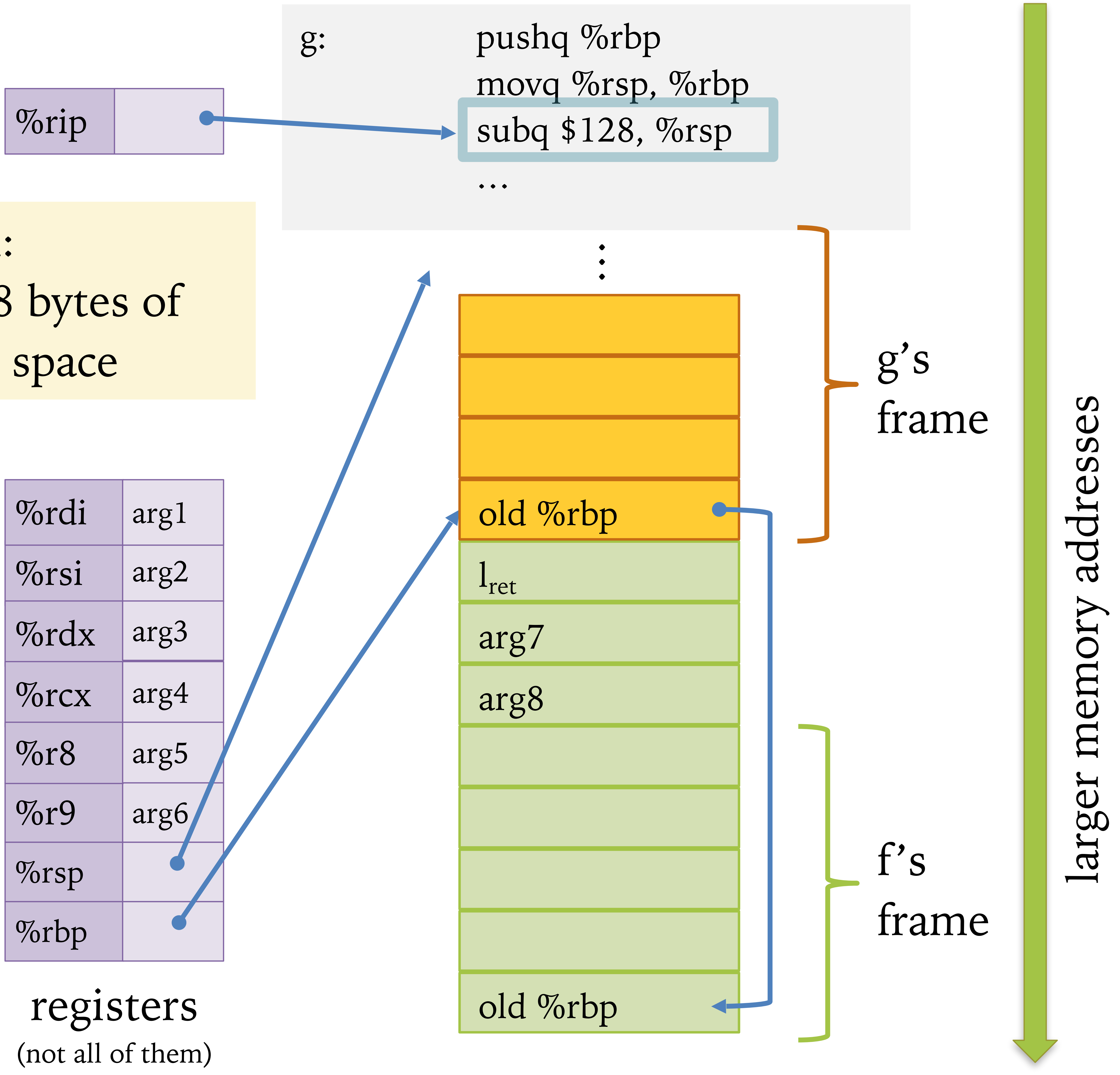
```
g:          pushq %rbp
            movq %rsp, %rbp
            subq $128, %rsp
            …
```

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

Callee Function Prologue

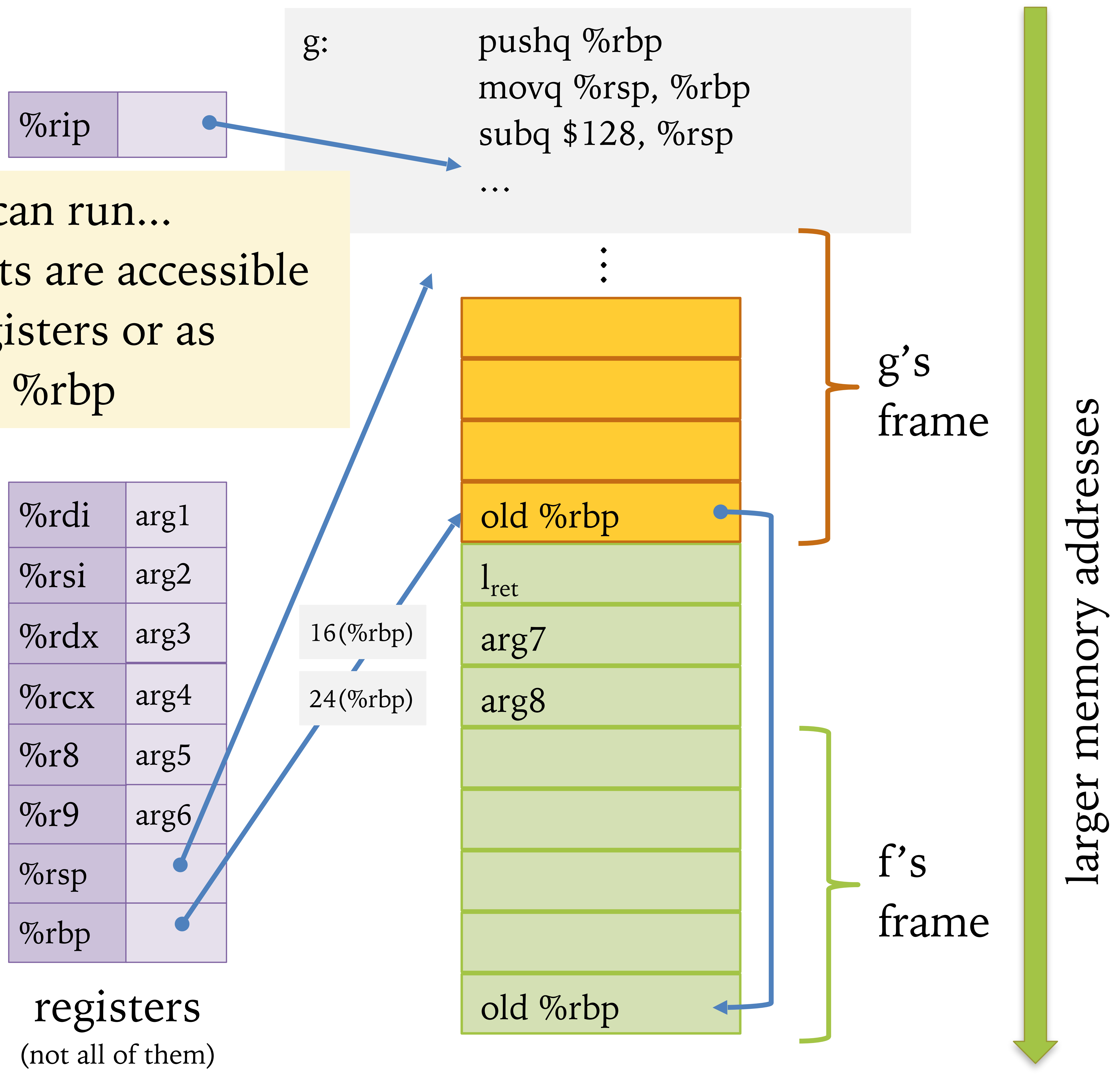
```
g:          pushq %rbp
            movq %rsp, %rbp
            subq $128, %rsp
            …
```

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

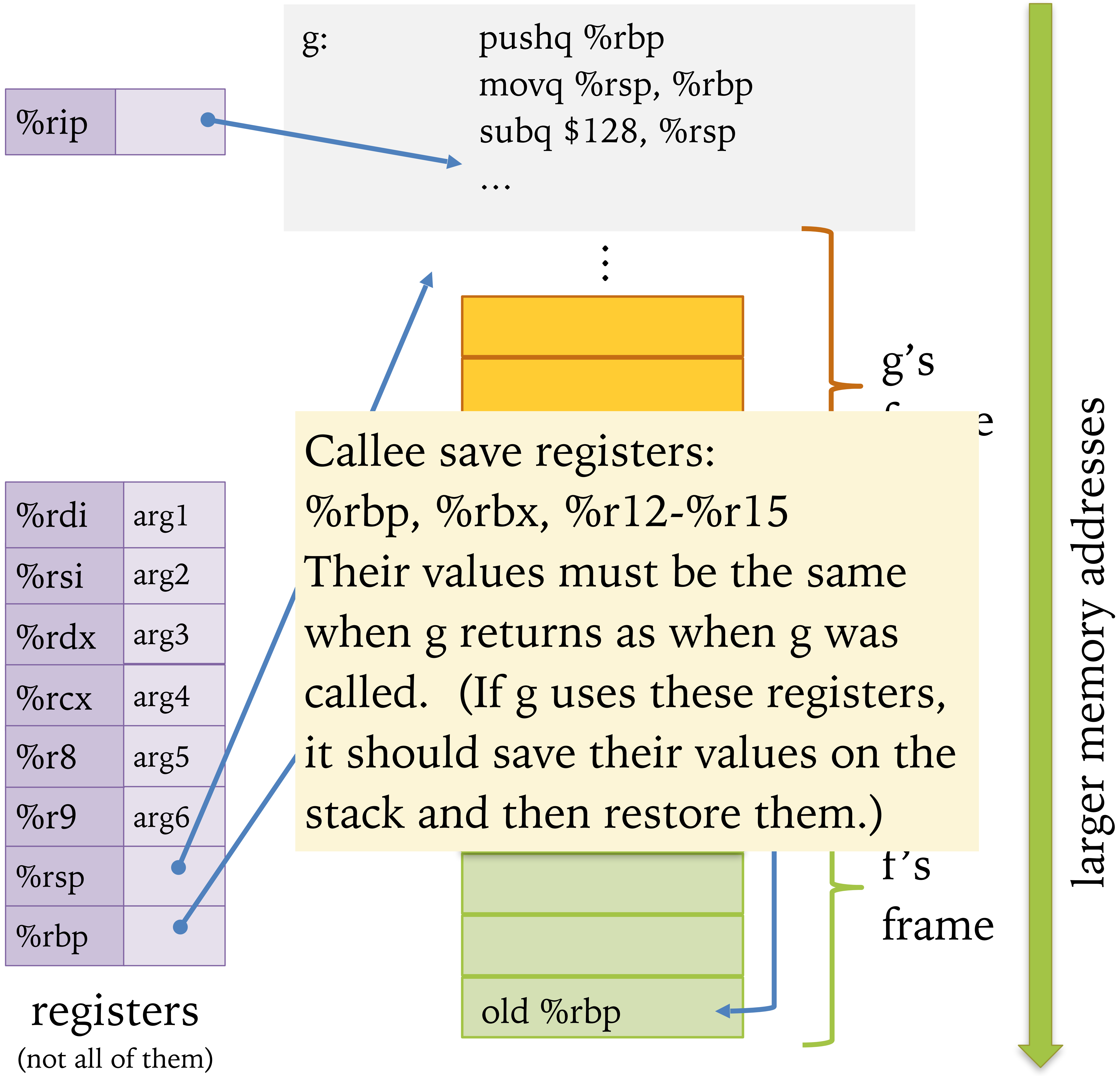


Callee Invariants: Function Arguments

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



Callee Invariants: Callee Same Registers

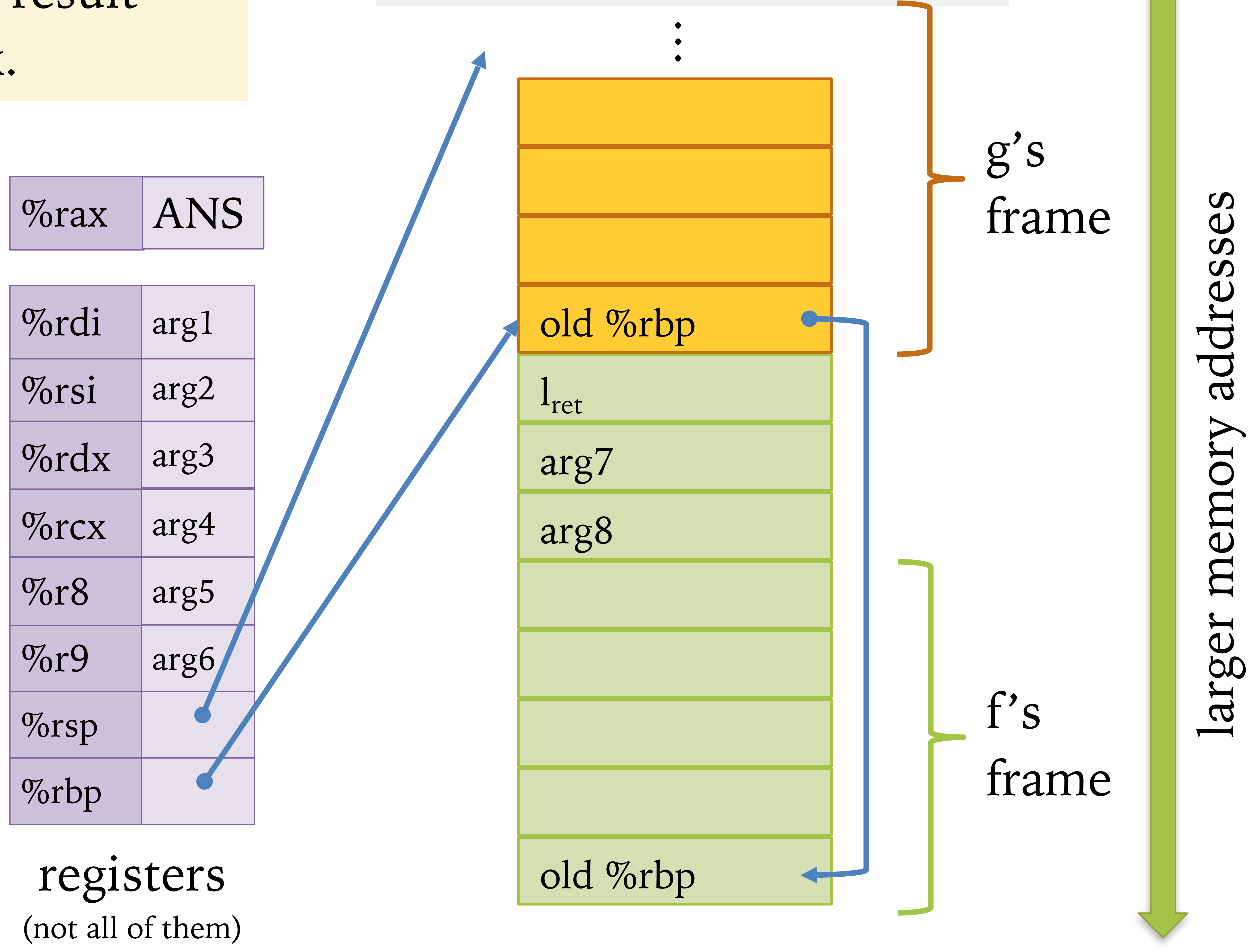


Callee save registers:
%rbp, %rbx, %r12-%r15
Their values must be the same when g returns as when g was called. (If g uses these registers, it should save their values on the stack and then restore them.)

Callee Epilogue (Return Protocol)

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

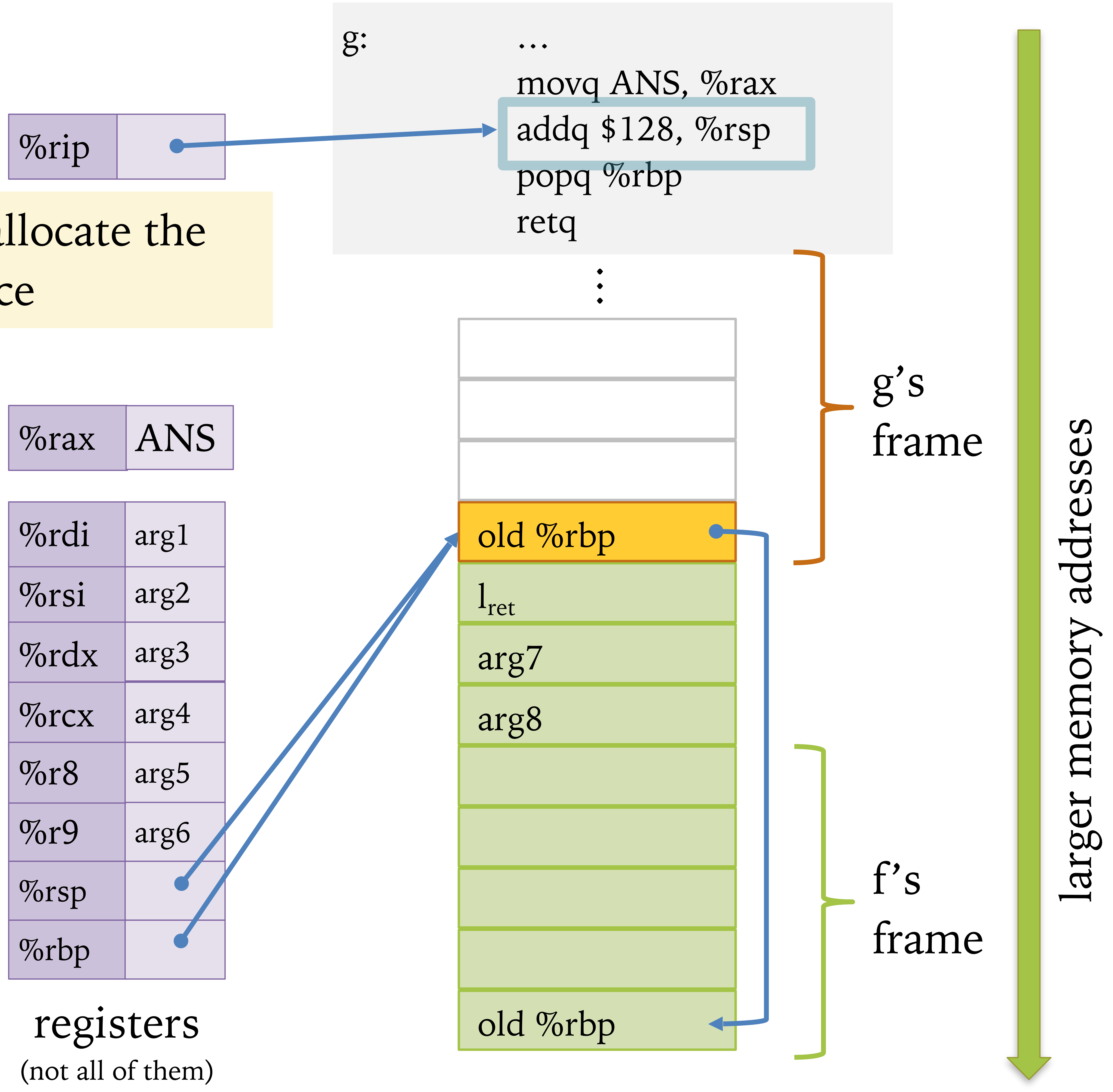
```
g:        …
          movq ANS, %rax
          addq $128, %rsp
          popq %rbp
          retq
```



Callee Epilogue (Return Protocol)

Step 2: deallocate the scratch space

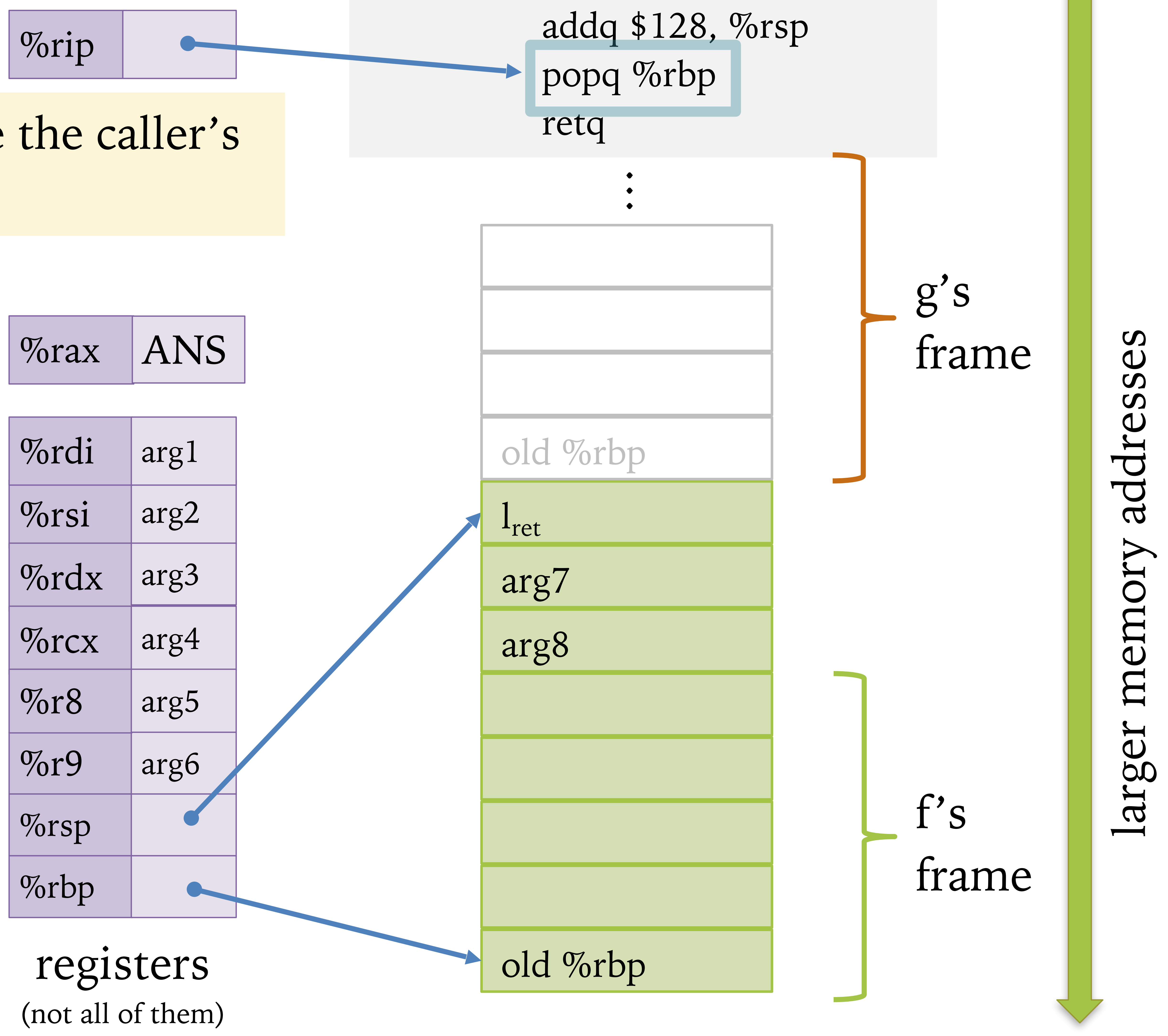
```
g:          …
            movq ANS, %rax
            addq $128, %rsp
            popq %rbp
            retq
```



Callee Epilogue (Return Protocol)

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

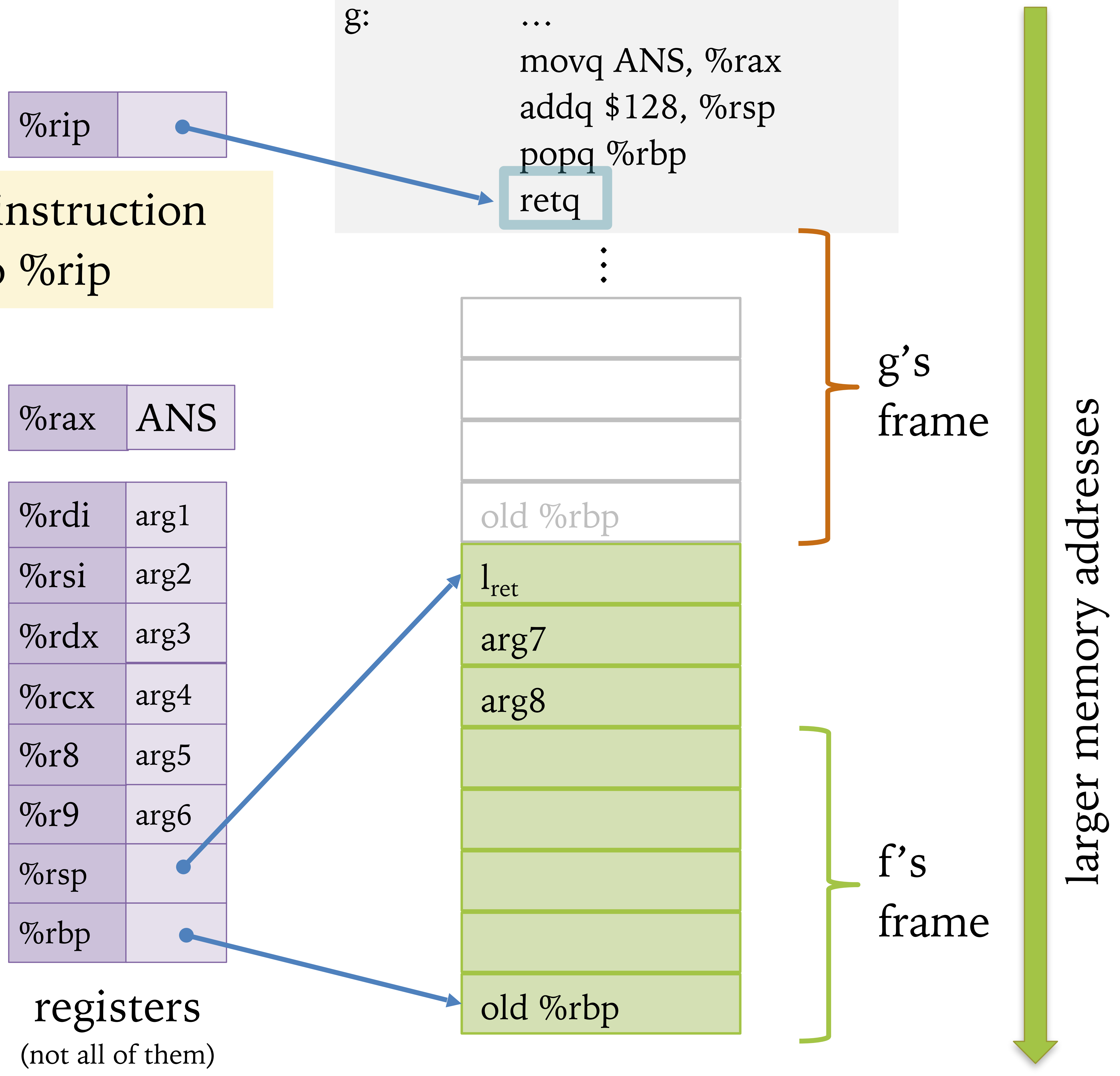
Step 3: restore the caller's %rbp



Callee Epilogue (Return Protocol)

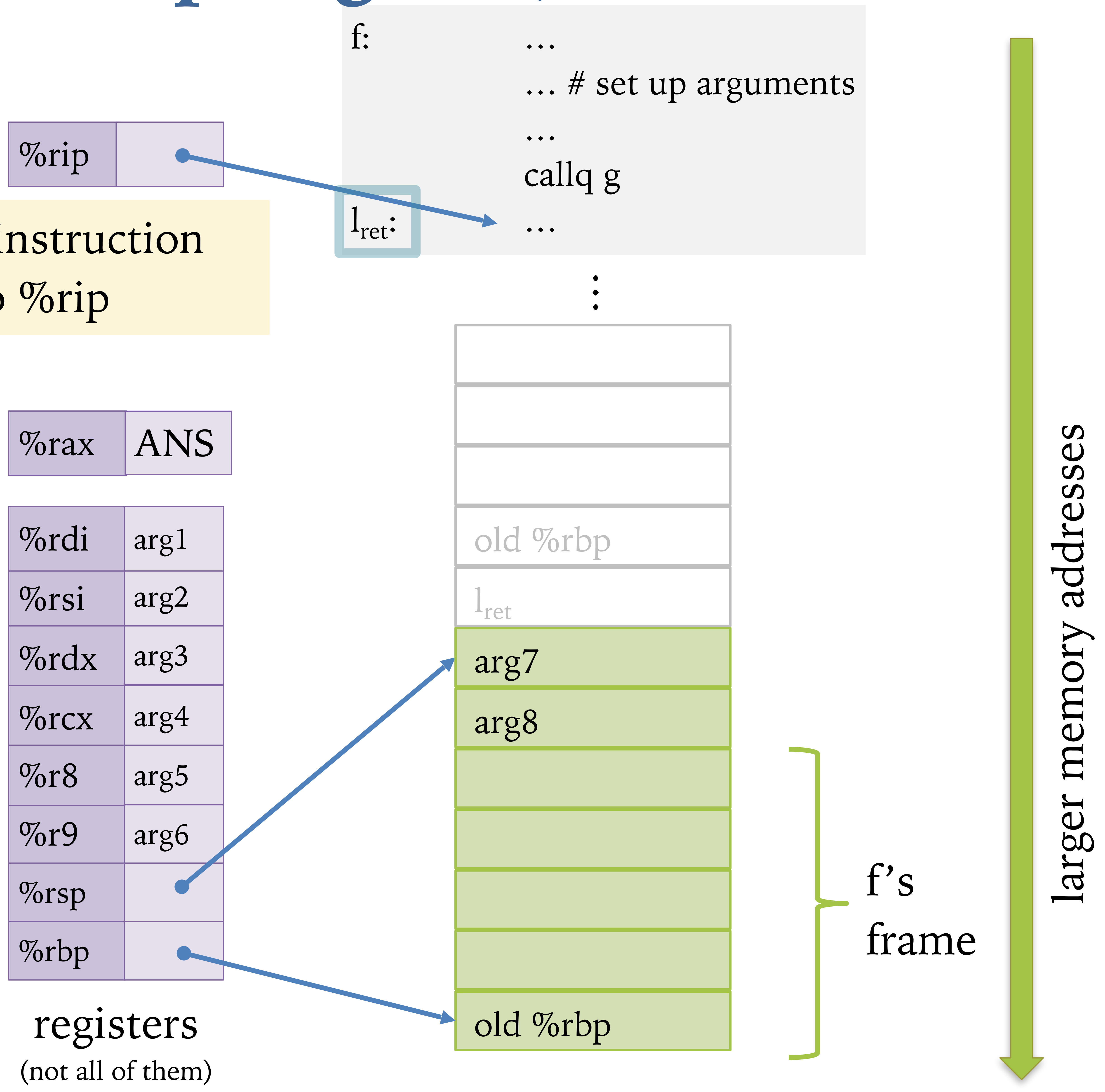
Step 4: the return instruction pops the stack into %rip

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



Callee Epilogue (Return Protocol)

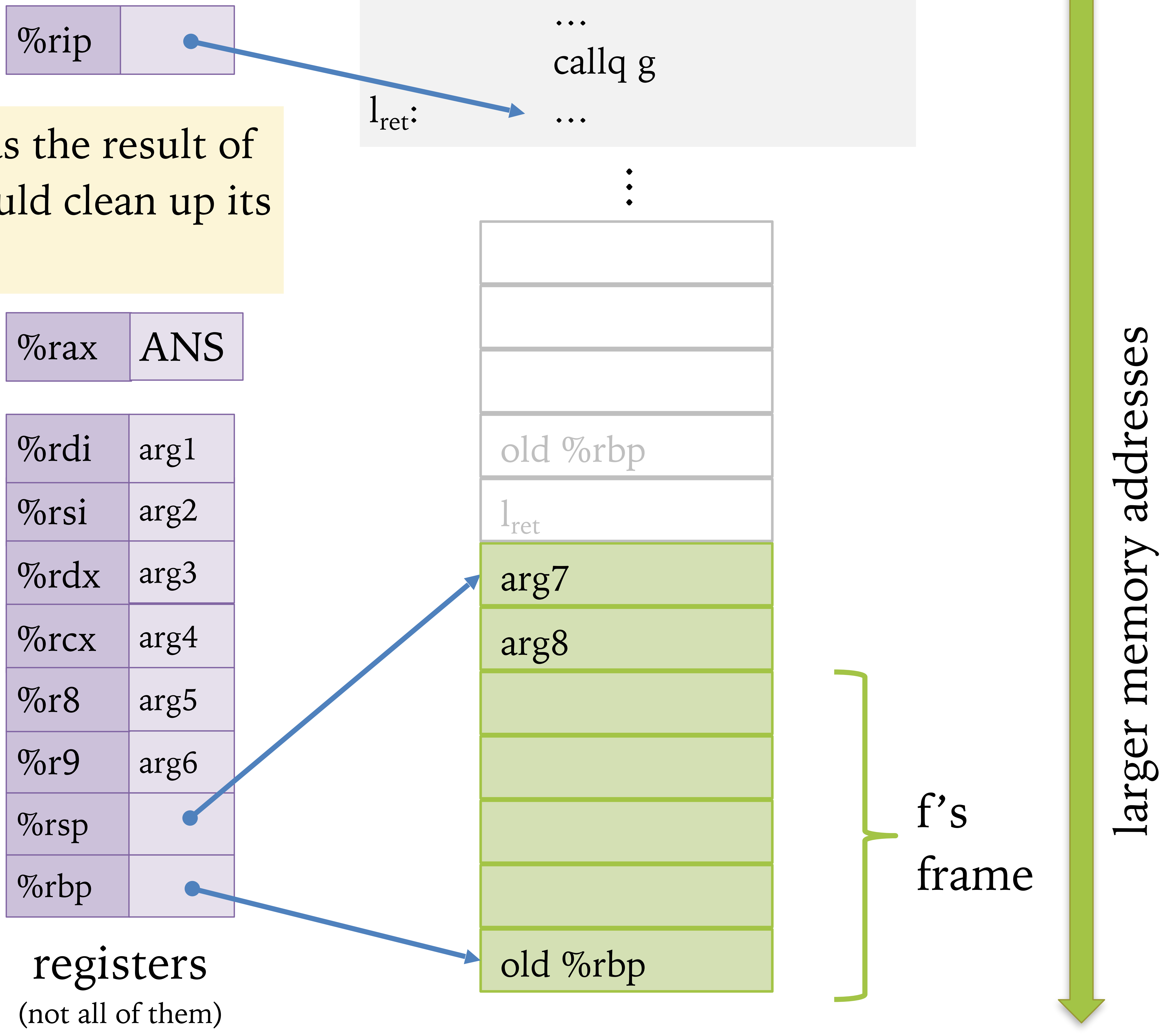
Step 4: the return instruction pops the stack into %rip



Back in f

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

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 nth 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 optimised away

Announcements

- HW2: X86lite
  - Due: Sunday, September 11 at 23:59

- Pair Programming:
  - Use GitHub Classroom link to create a new team for the project or join an existing one
  - Choose a funny group name!
  - Submission by any group member done on Canvas counts for the group

Demo: Directly Compiling Expressions to X86lite

- https://github.com/cs4212/week-02-x86lite
- Definition of compilation: `compile.ml`
- Example programs: `main2.ml`
- Linking with assembly: `calculator.c`

Directly Translating AST to Assembly

- For simple languages, no need for intermediate representation.
  - e.g. the arithmetic expression language from
- Main Idea: Maintain invariants
  - e.g. Code emitted for a given expression *always* computes the answer into %rax
- Key Challenges:
  - storing intermediate values needed to compute complex expressions
  - some instructions use specific registers (e.g. shift)

One Simple Strategy

- Compilation is the process of “emitting” instructions into an instruction stream.
- To compile an expression, we recursively compile sub expressions and then process the results.

- Invariants:
  - Compilation of an expression yields its result in %rax
  - Argument (Xi) is stored in a dedicated operand register
  - Intermediate values are pushed onto the stack
  - Stack slot is popped after use (so the space is reclaimed)

- Resulting code is wrapped (e.g., with retq) to comply with cdecl calling conventions

- Alternative strategy: see the compile2 in compile.ml

Intermediate Representations

Why do something else?

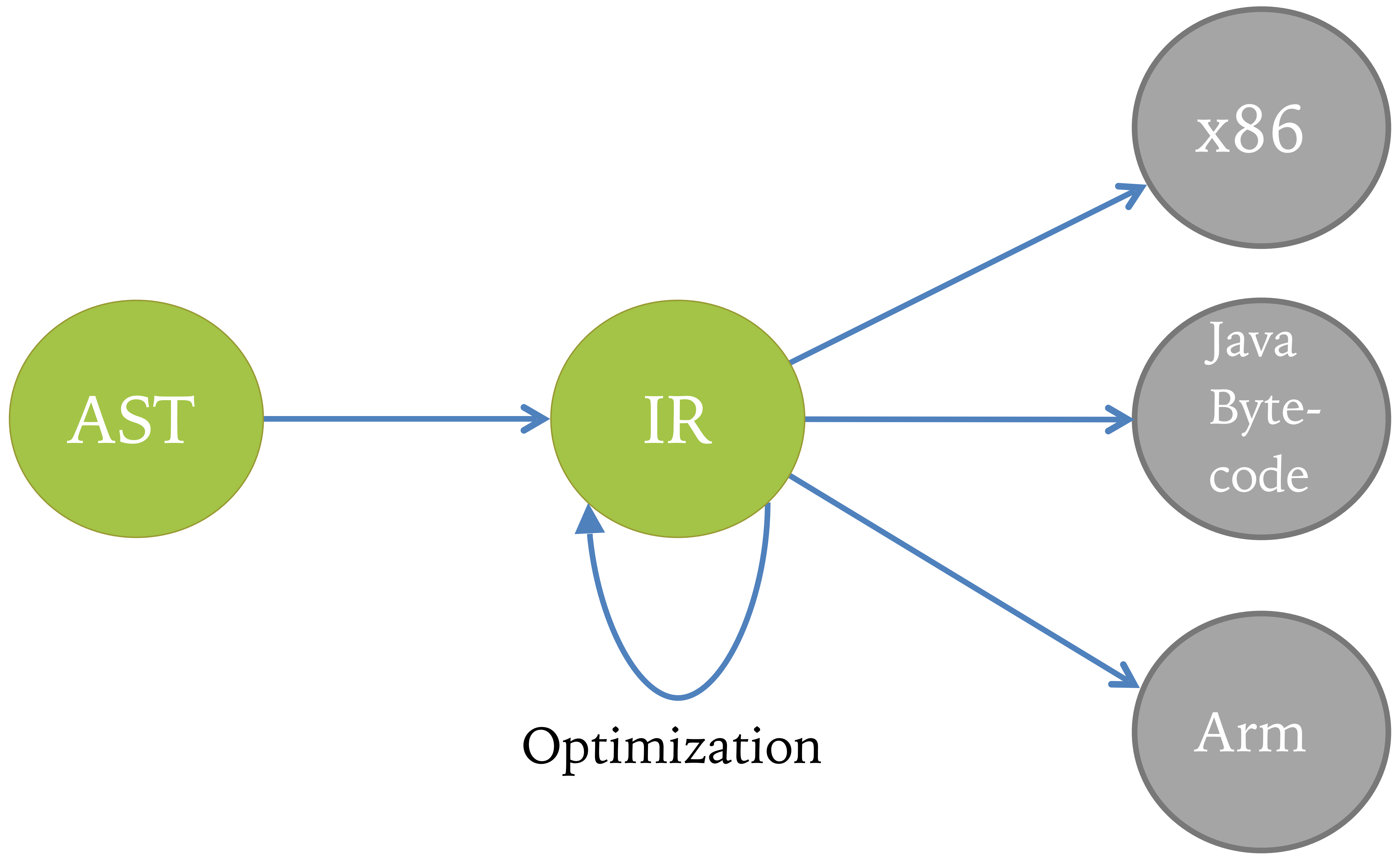
- We have seen a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located
- Control-flow is not structured:
  - Arbitrary jumps from one code block to another
  - Implicit fall-through makes sequences of code non-modular (i.e. you can't rearrange sequences of code easily)
- Retargeting the compiler to a new architecture is hard.
  - Target assembly code is hard-wired into the translation

Intermediate Representations (IR's)

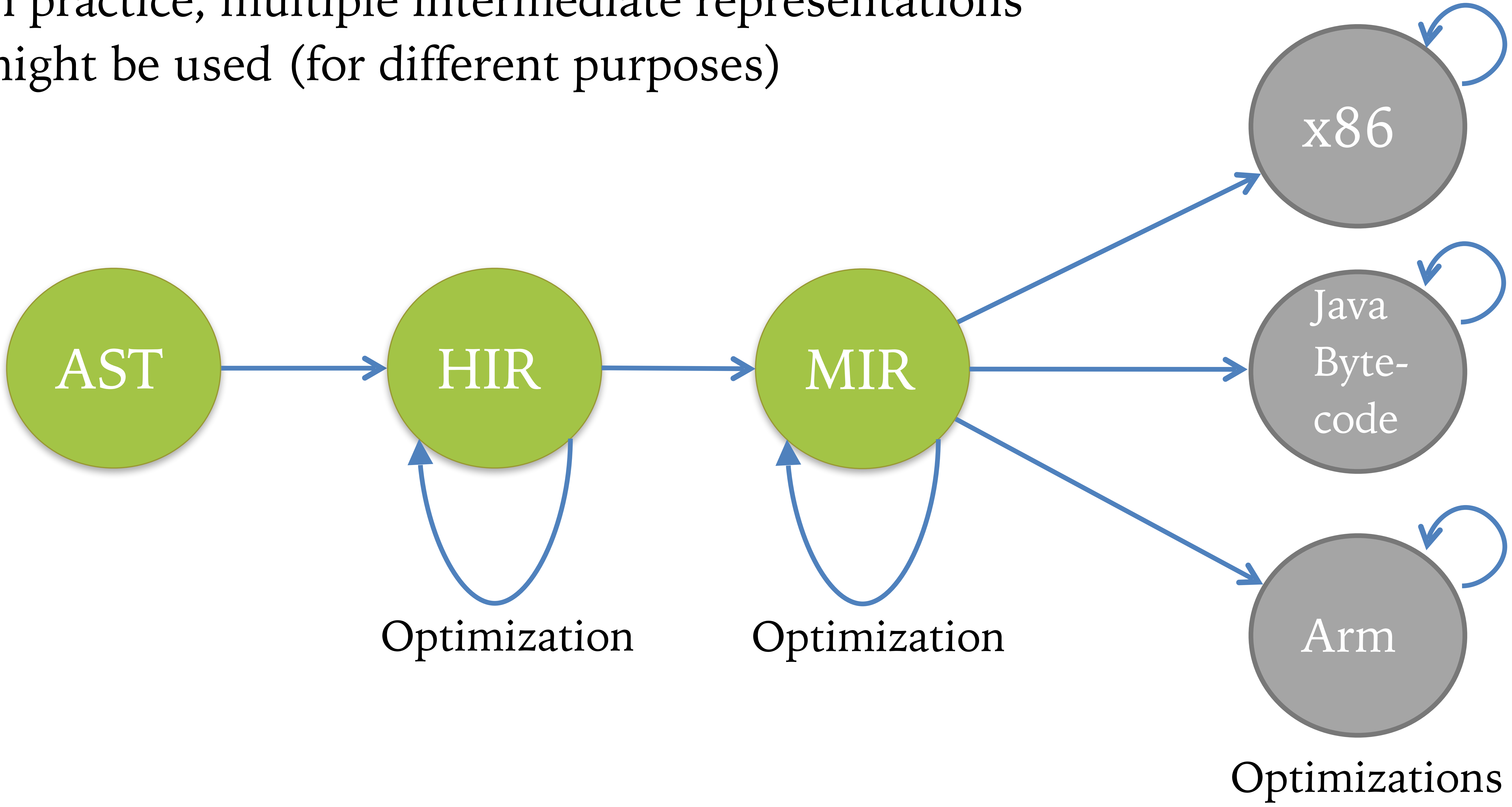
- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization.



Multiple IR's

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations

- In practice, multiple intermediate representations might be used (for different purposes)



What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations

- Example: Source language might have "while", "for", and "foreach" loops (and maybe more variants)
  - IR might have only "while" loops and sequencing
  - Translation eliminates "for" and "foreach"

```
⟦for(pre; cond; post) {body}⟧
 =
  ⟦pre; while(cond) {body;post}⟧
```

  - Here the notation ⟦cmd⟧ denotes the "translation" or "compilation" of the command cmd.

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g. convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g. inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

- Low-level IR's
  - Machine dependent assembly code + extra pseudo-instructions
    - e.g. a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    - e.g. (on x86) a `imulq` instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is straightforward
  - Allows low-level optimizations based on target architecture
    - e.g. register allocation, instruction selection, memory layout, etc.

- What's in between?

Mid-level IR's: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers, or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values are named to facilitate optimizations that attempt to minimize stack/register usage
- Many examples:
  - Triples: OP a b
    - Useful for instruction selection on X86 via "graph tiling" (a way to better utilise registers)
  - Quadruples: a = b OP c (RISC-like "three address form")
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM: industrial-strength IR, based on SSA
  - Stack-based:
    - Easy to generate
    - e.g. Java Bytecode, UCODE

Growing an IR

- Develop an IR in detail… starting from the very basic.

- Start: a (very) simple intermediate representation for the *arithmetic language*
  - Very high level
  - No control flow

- Goal: A simple subset of the LLVM IR
  - LLVM = “Low-level Virtual Machine”
  - Used in HW3+

- Add features needed to compile rich source languages

Simple let-based IR

Eliminating Nested Expressions

- Fundamental problem:
  - Compiling complex & nested expression forms to simple operations.

Source

```
((1 + X4) + (3 + (X1 * 5)))
```

AST

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

IR

?

- Idea: *name* intermediate values, make order of evaluation explicit.
  - No nested operations.

Translation to SLL

- Given this:

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

- Translate to this desired SLL form:

```
let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
tmp3
```

- Translation makes the order of evaluation explicit.
- Names intermediate values
- Note: introduced temporaries are never modified

Demo

- https://github.com/cs4212/week-03-ir-2024
- Using IRs: `ir_by_hand.ml`
- Definitions: `ir<X>.ml`

Intermediate Representations

- IR1: Expressions
  - simple arithmetic expressions, immutable global variables
- IR2: Commands
  - global *mutable* variables
  - commands for update and sequencing
- IR3: Local control flow
  - conditional commands & while loops
  - *basic blocks*
- IR4: Procedures (top-level functions)
  - local state
  - call stack
- IR5: ”almost” LLVM IR

IR3: Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the "link"
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

Next Lecture

- LLVM