YSC4230: Programming Language Design and Implementation

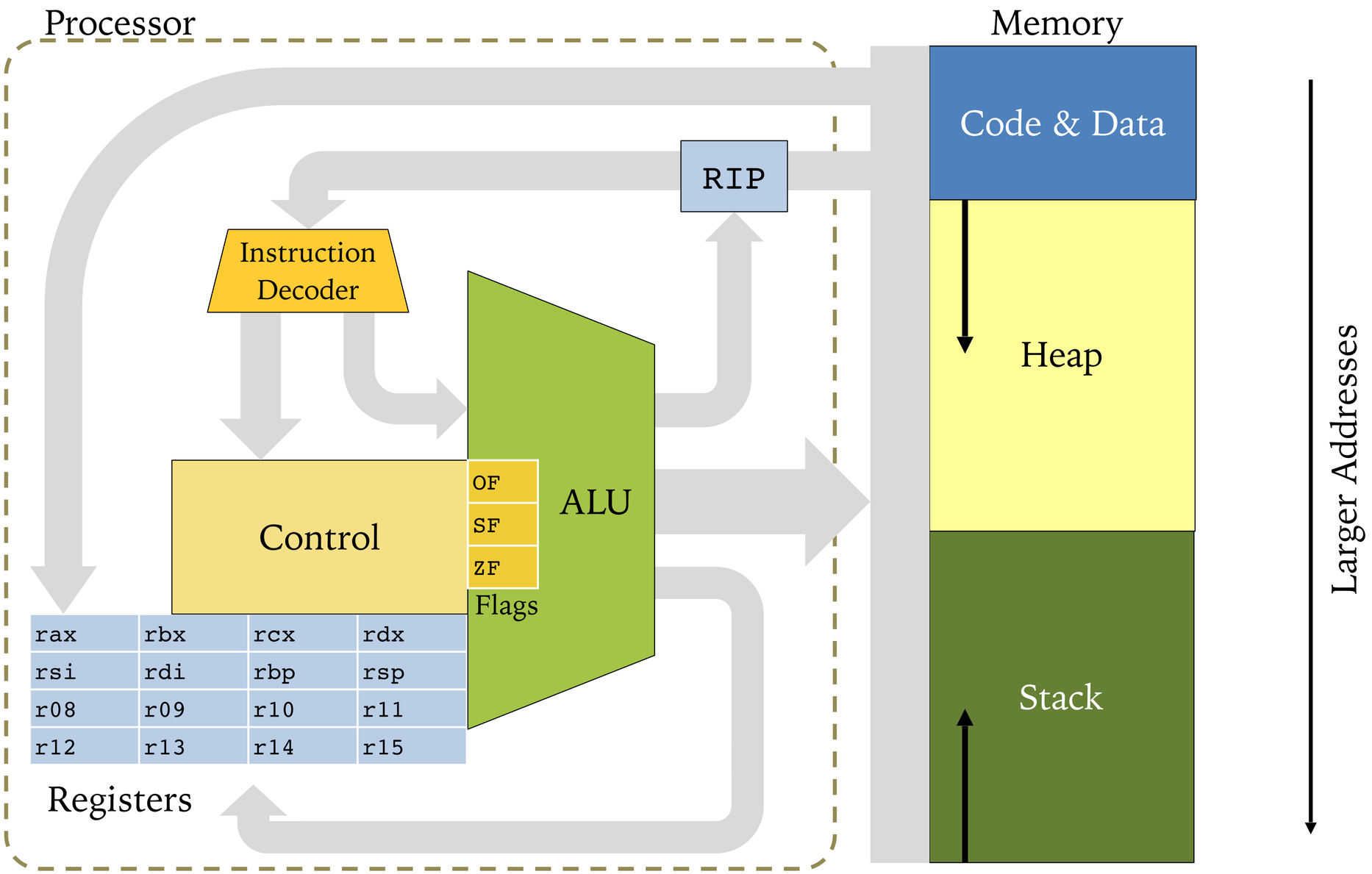
Week 3: Compiling Function Calls to x86; Intermediate Representations

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Implementing Functions & C Calling Conventions

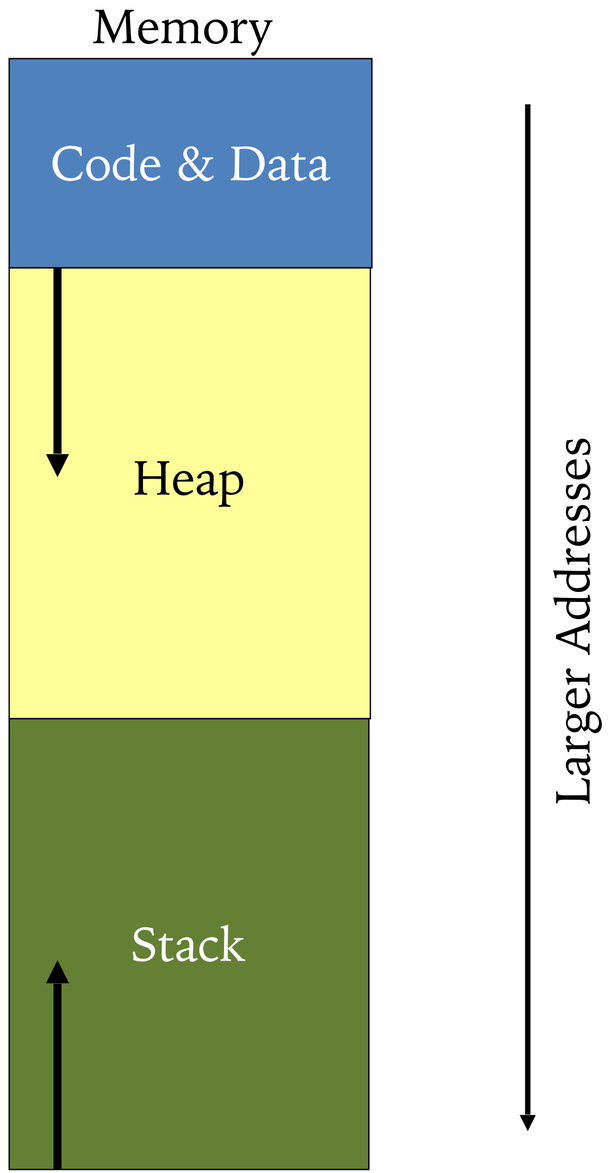
X86 Schematic



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

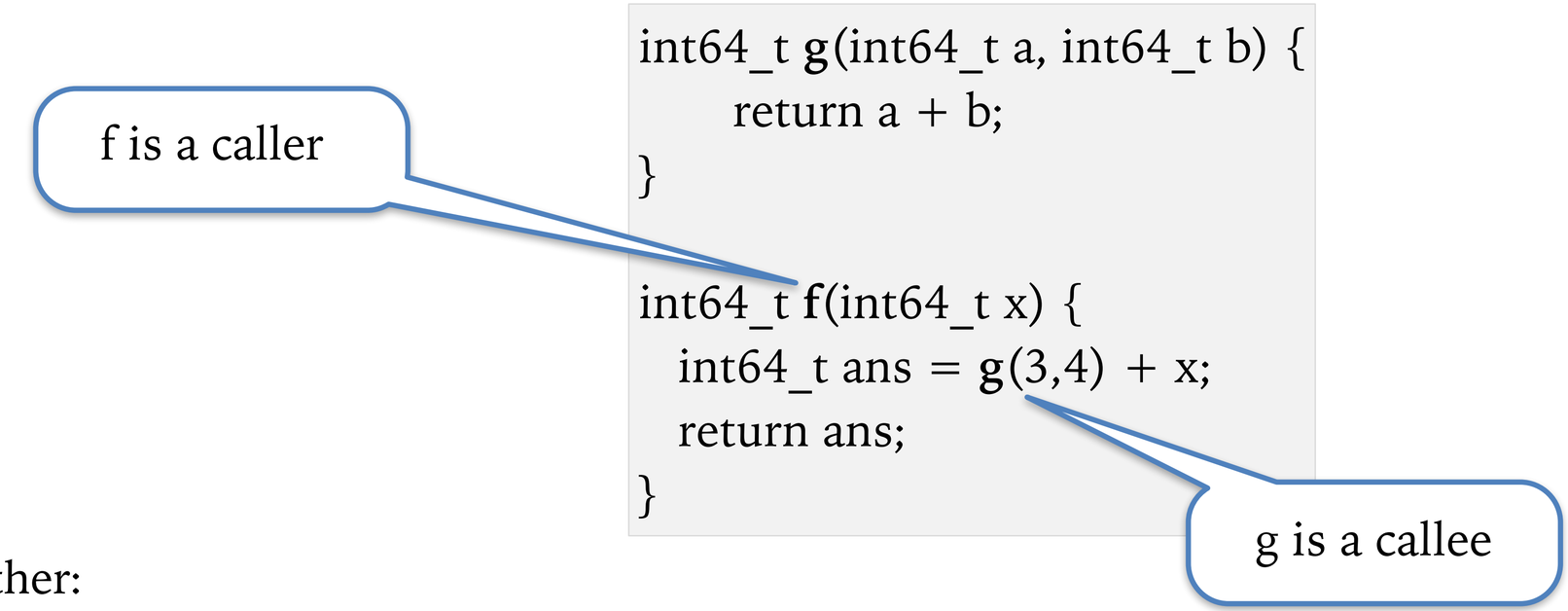
- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- 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

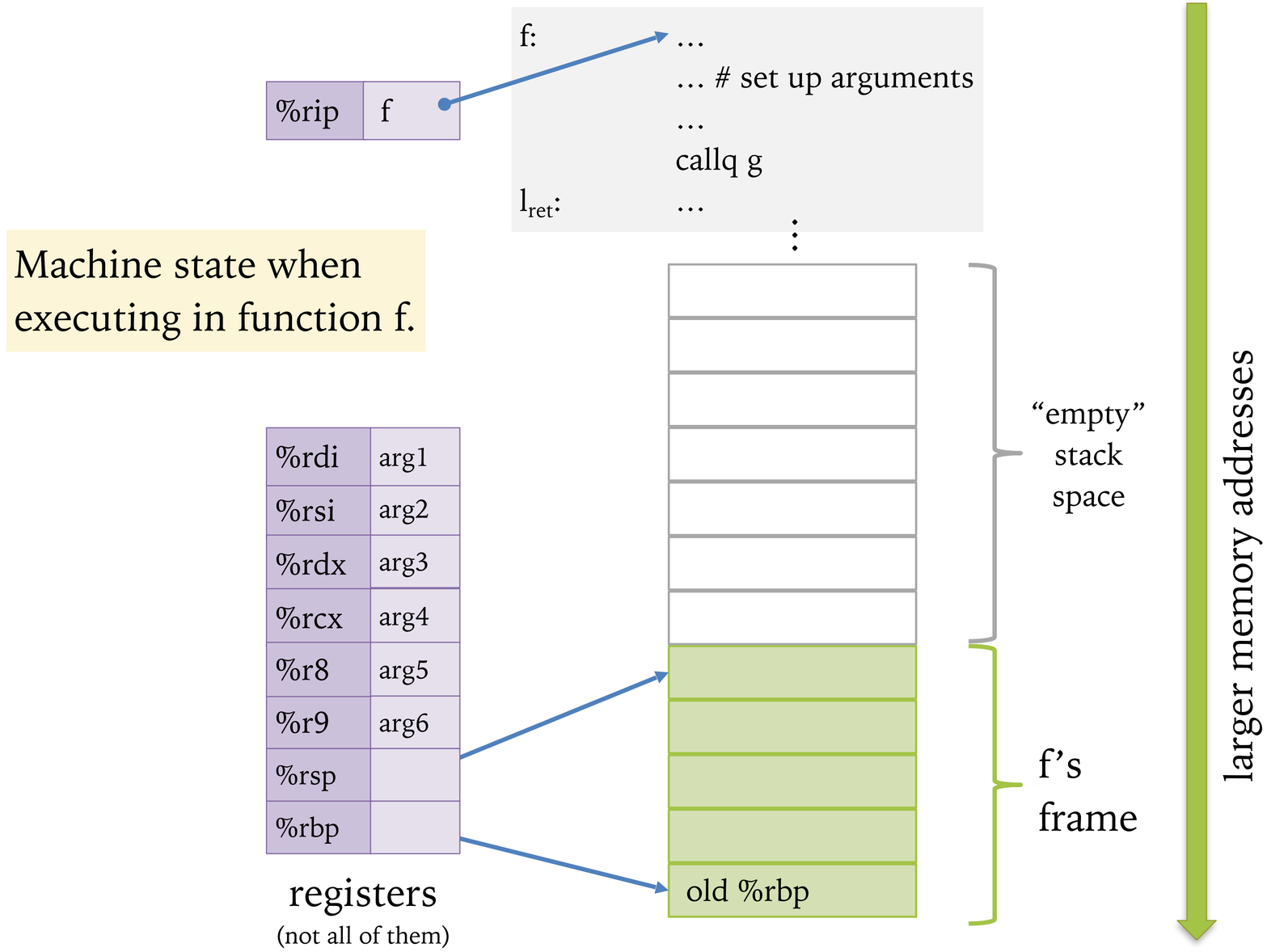
Calling Conventions

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



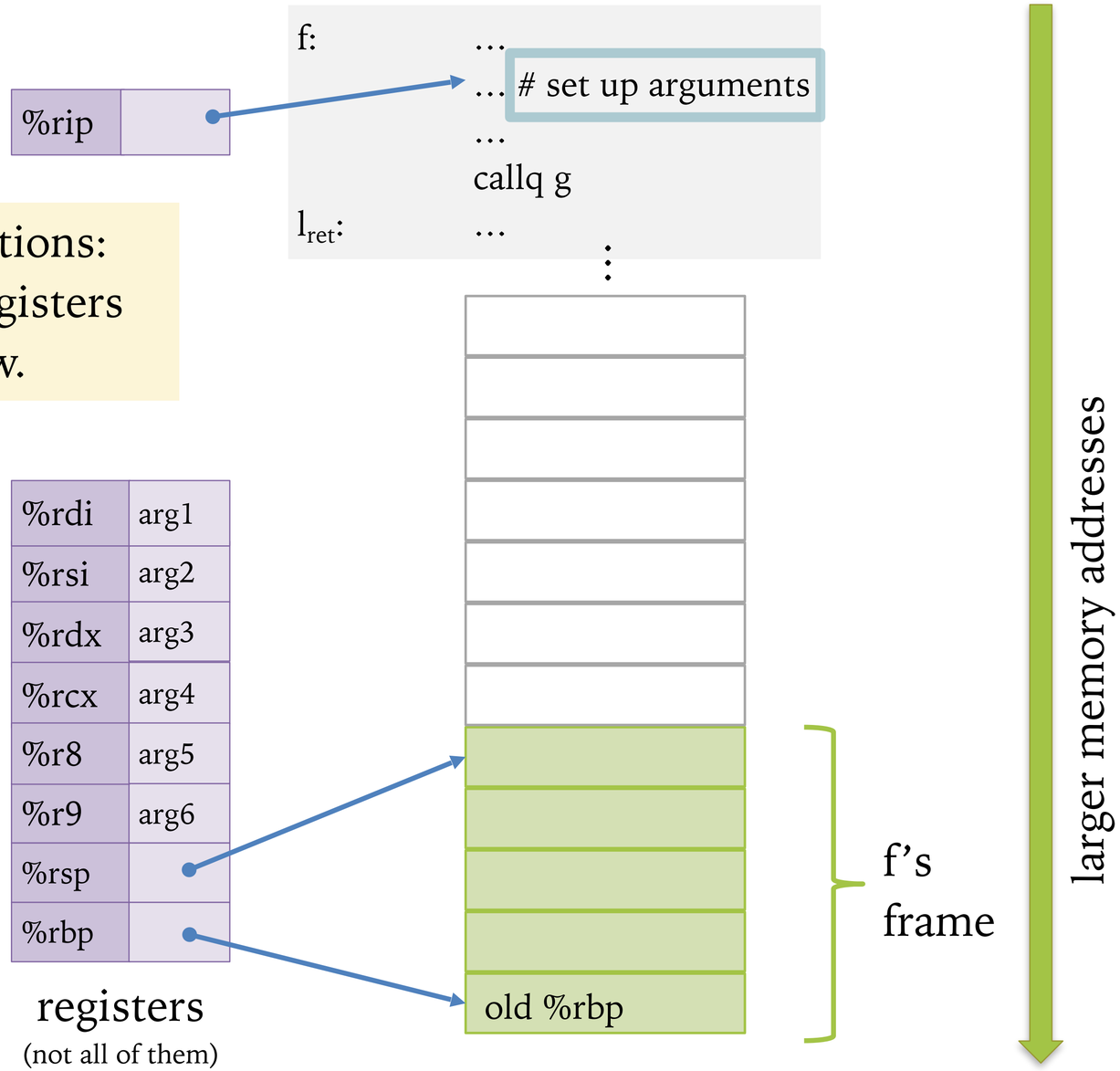
- Designate registers either:
  - Caller Save – e.g., freely usable by the called code
  - Callee Save – e.g., must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes variable number arguments harder — the callee doesn't know how many are those)

x64 Calling Conventions: Caller Protocol

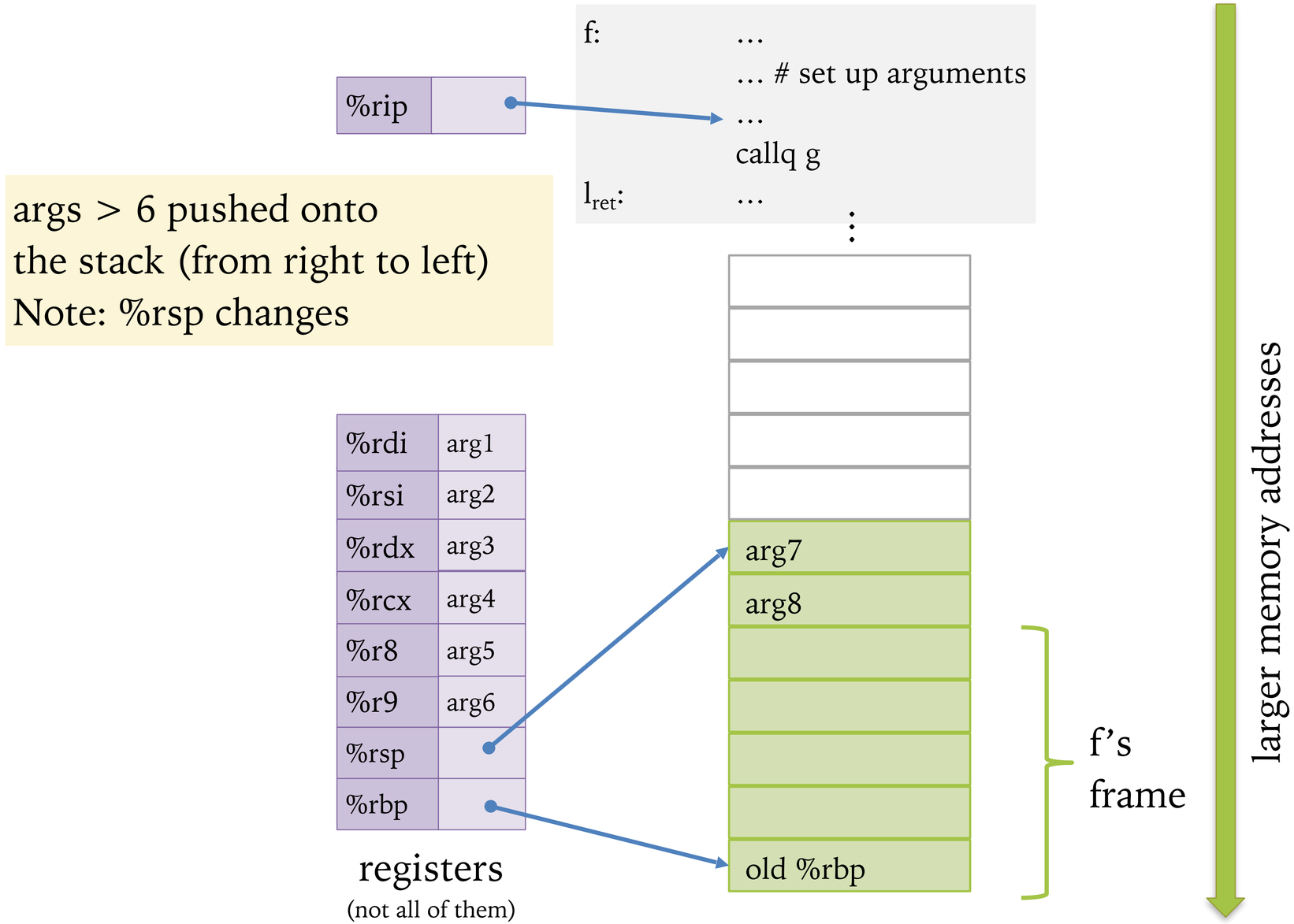


x64 Calling Conventions: Caller Protocol

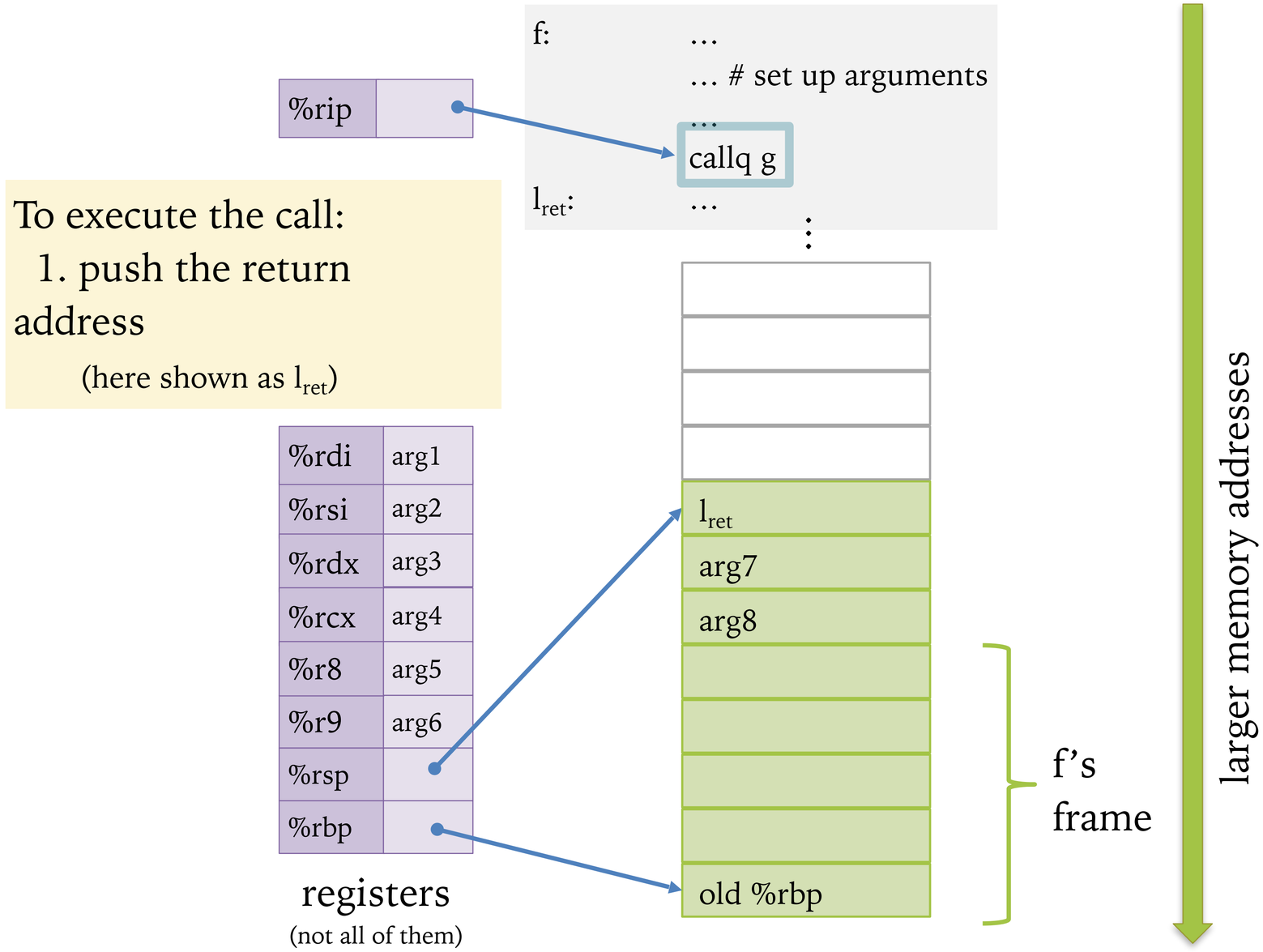
Calling conventions:
args 1…6 in registers
as shown below.



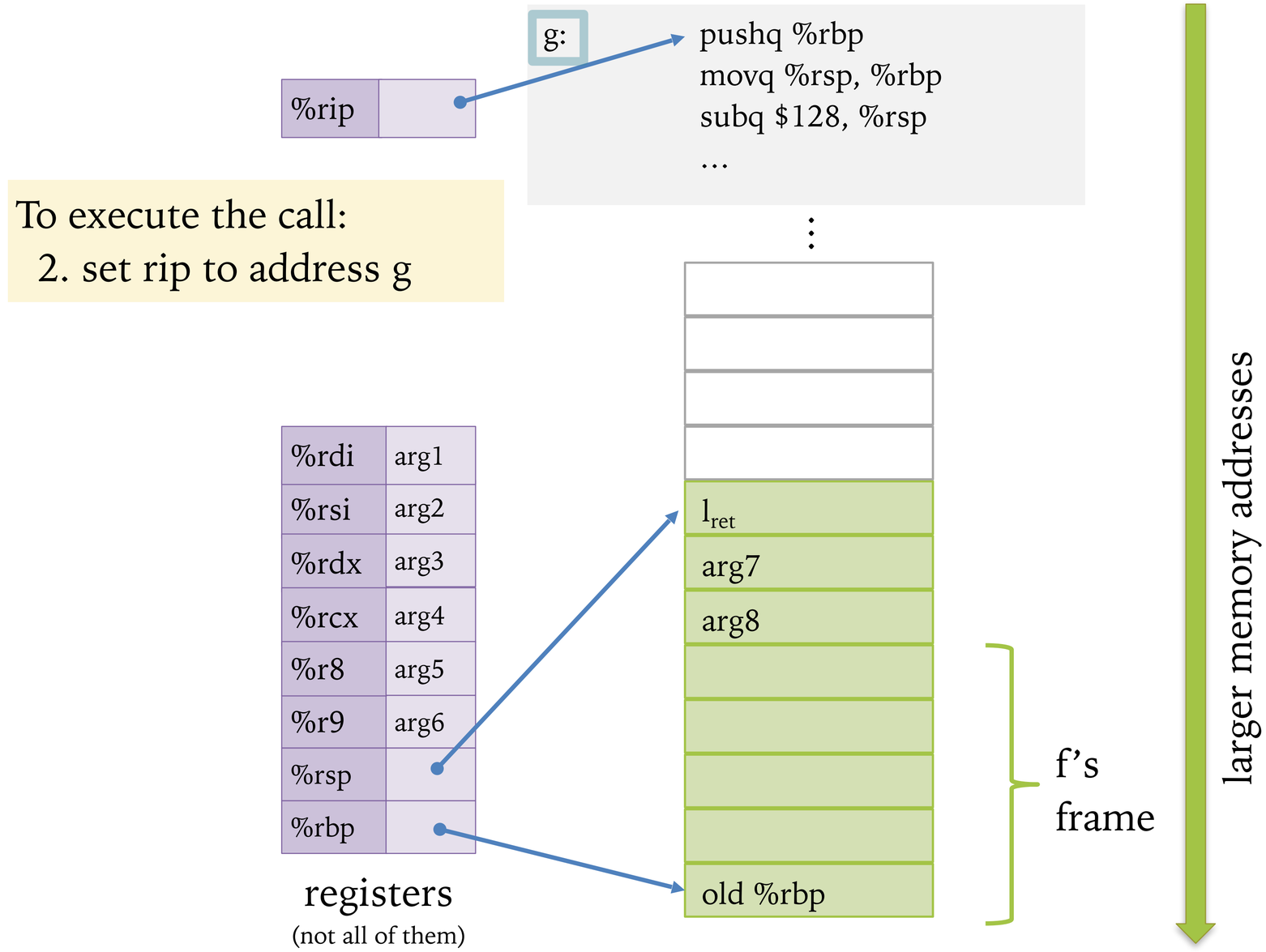
x64 Calling Conventions: Caller Protocol



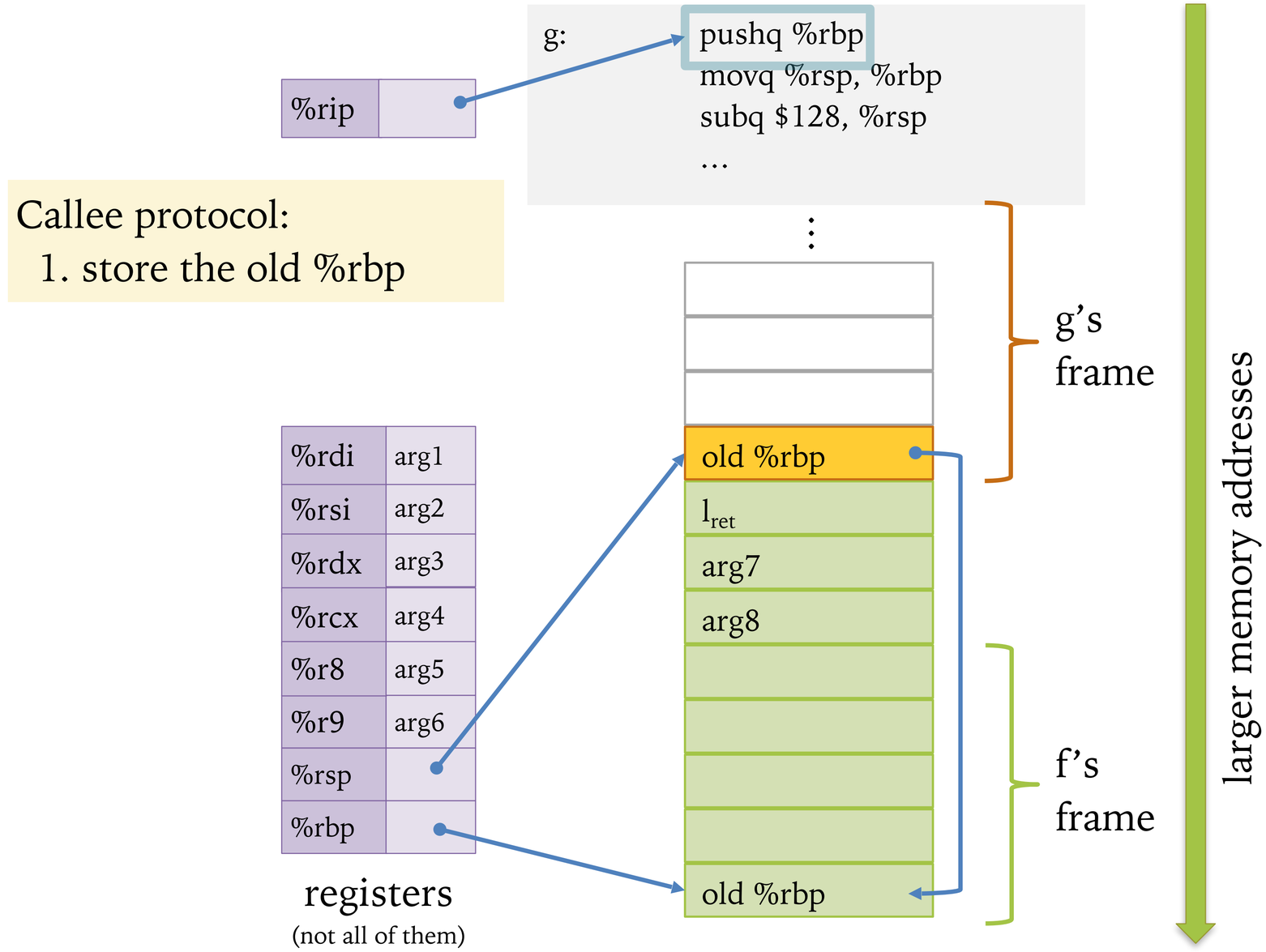
Call Instruction



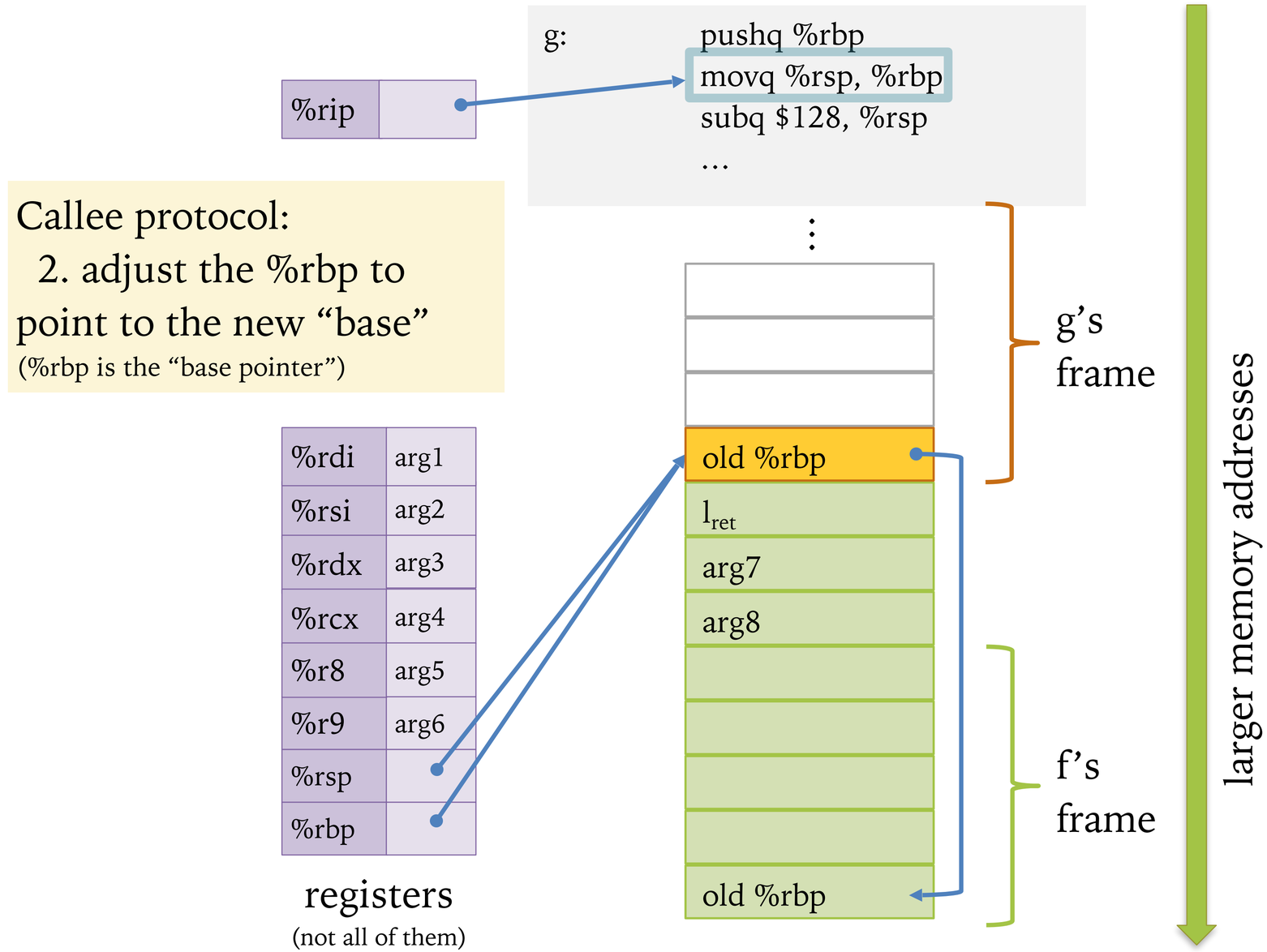
Call Instruction



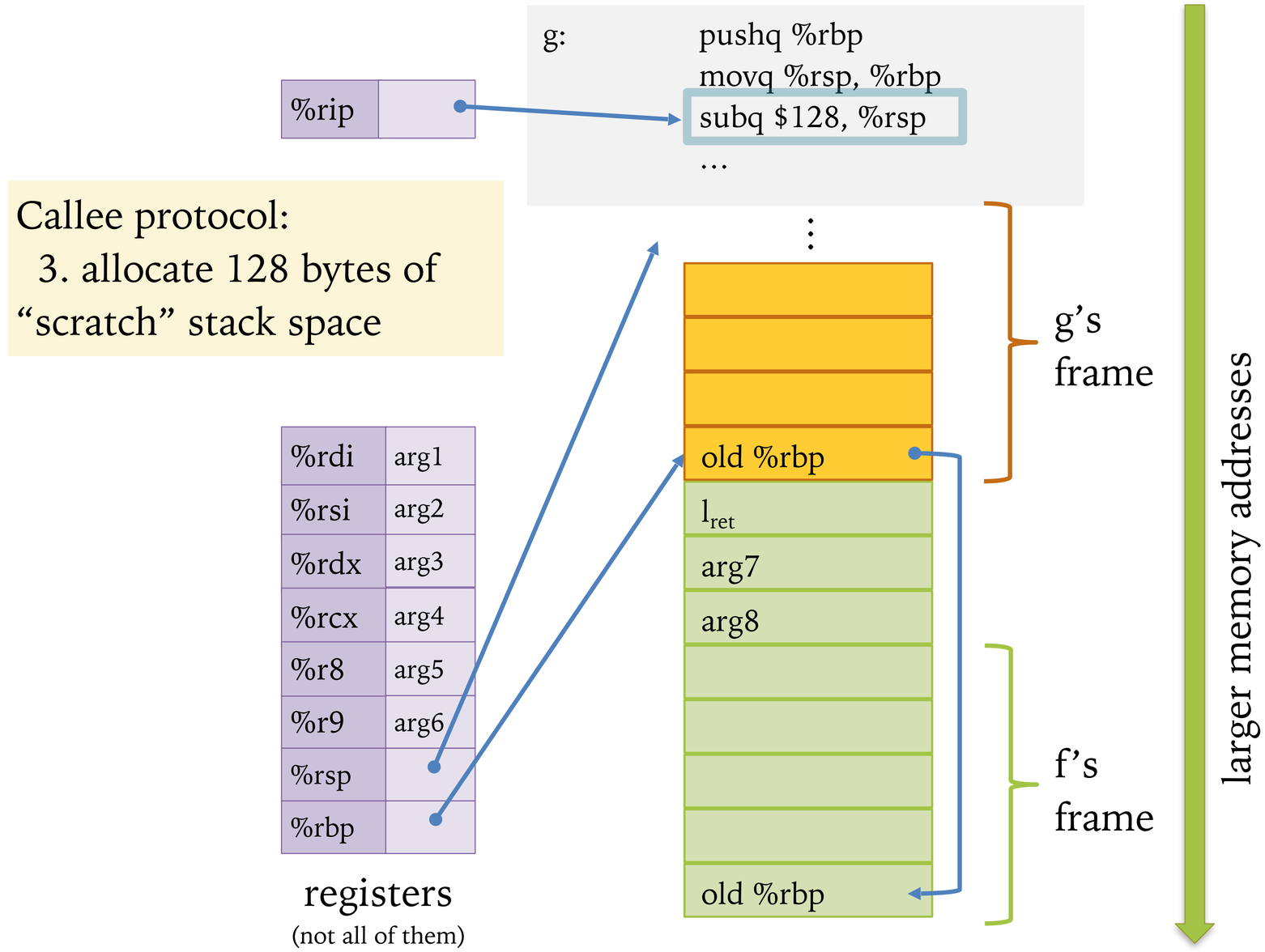
Callee Function Prologue



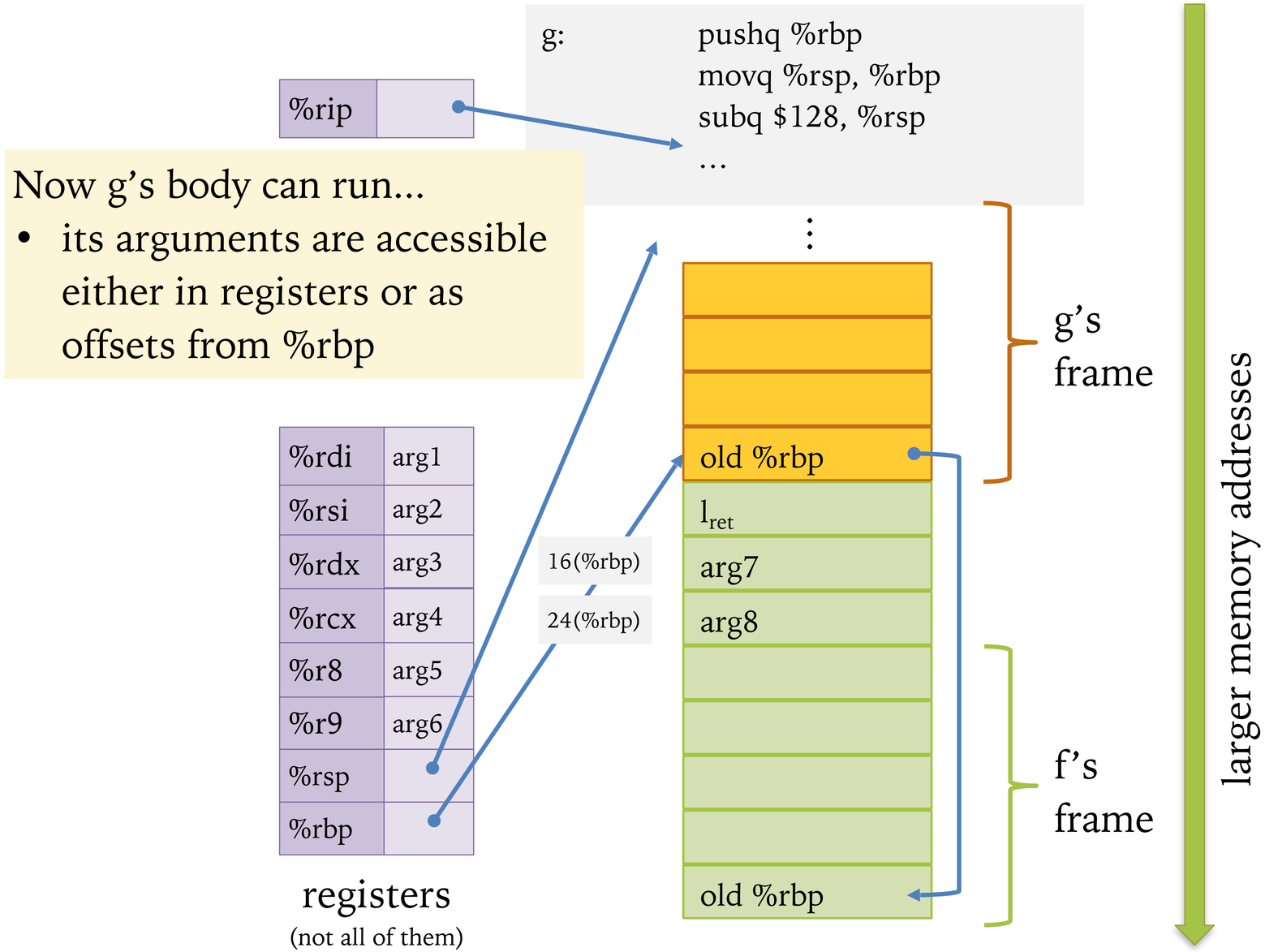
Callee Function Prologue



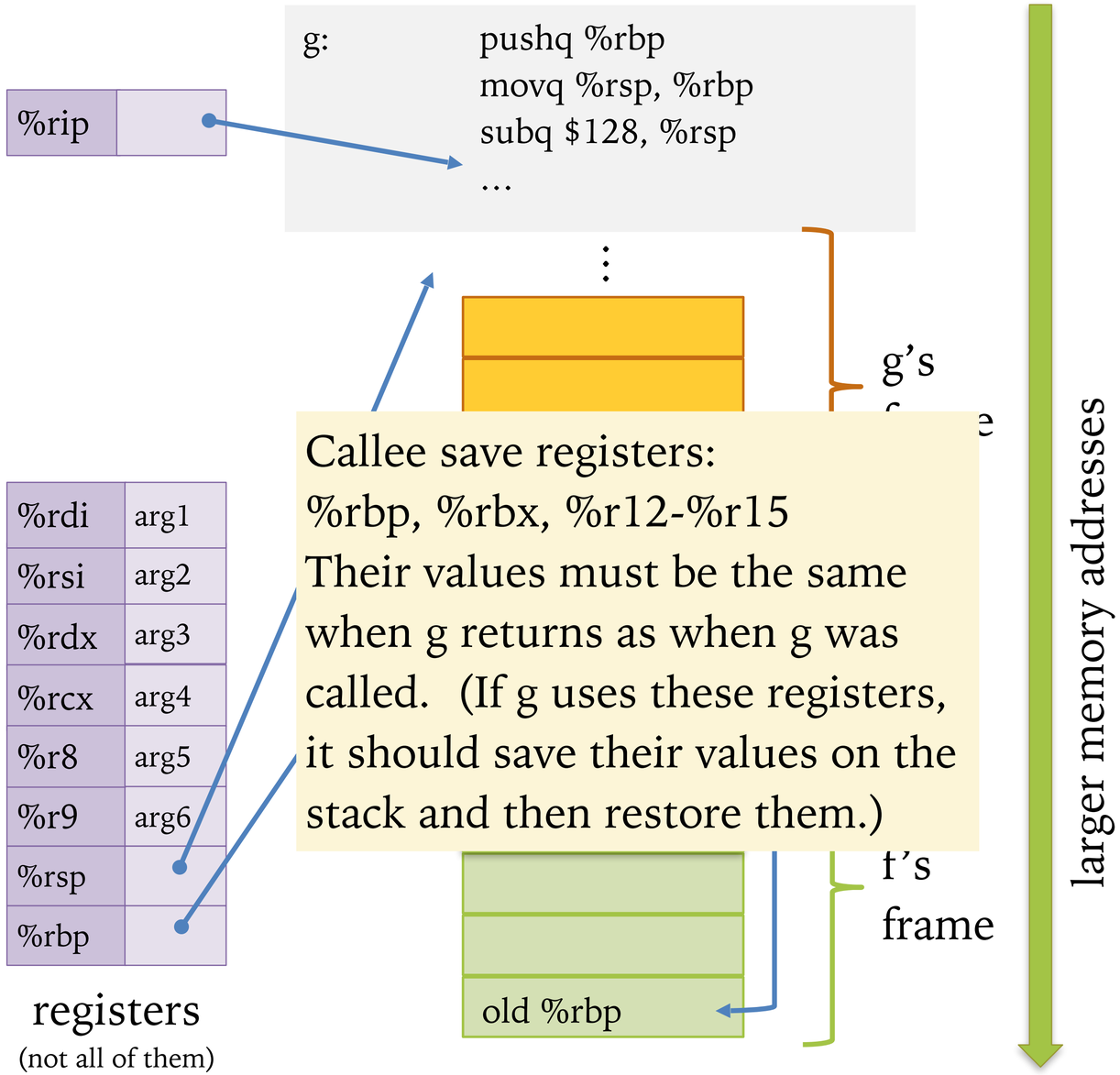
Callee Function Prologue



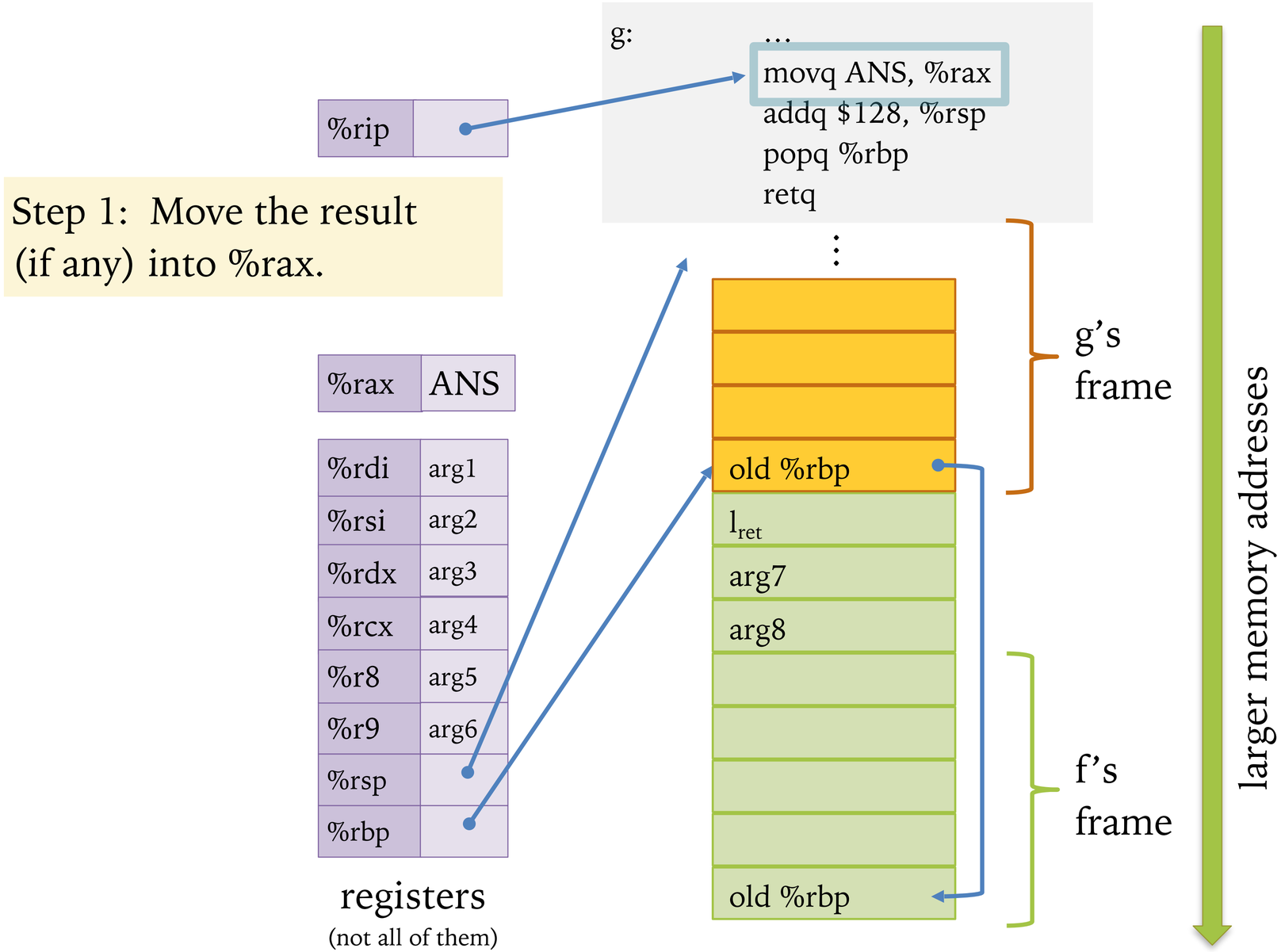
Callee Invariants: Function Arguments



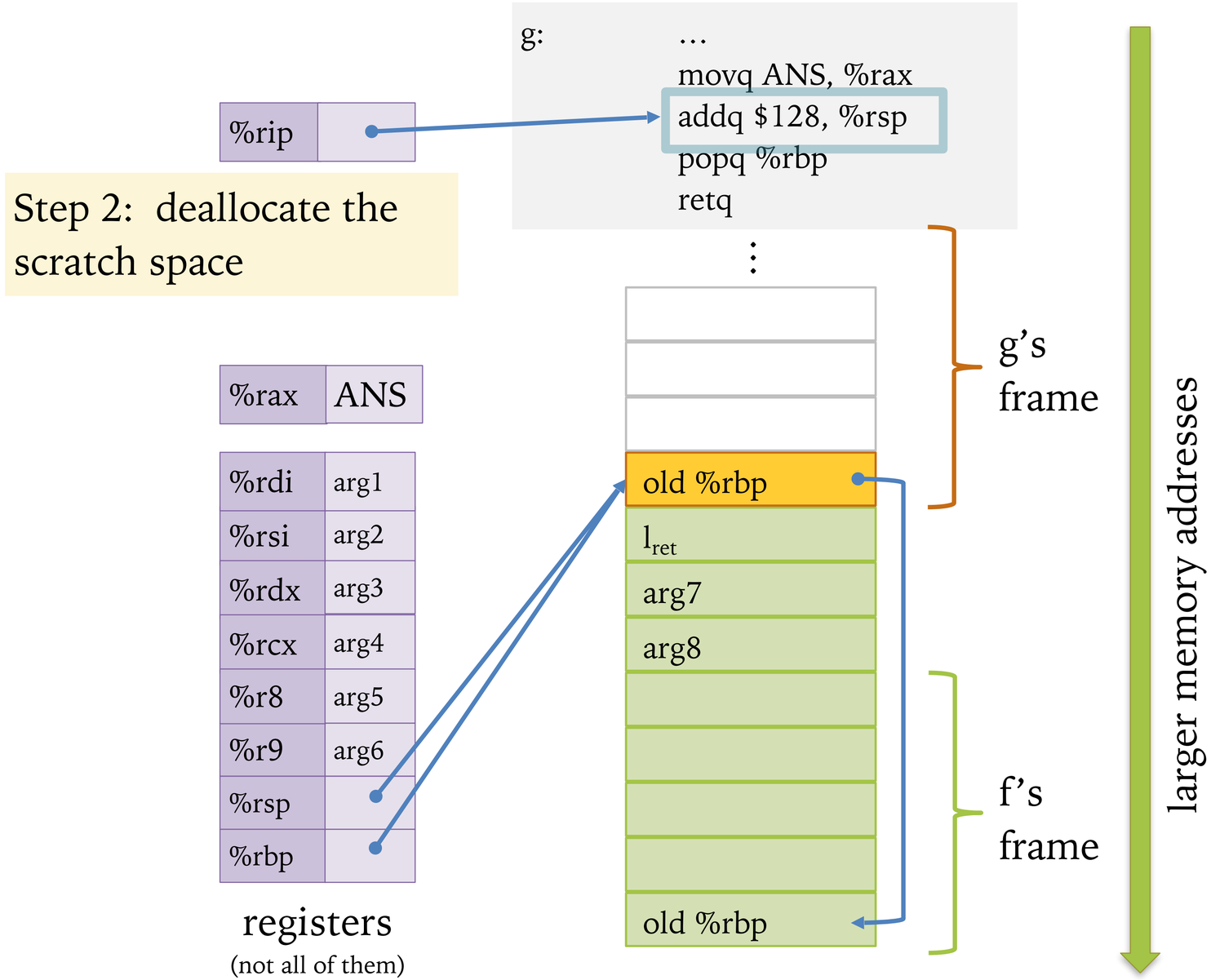
Callee Invariants: Callee Same Registers



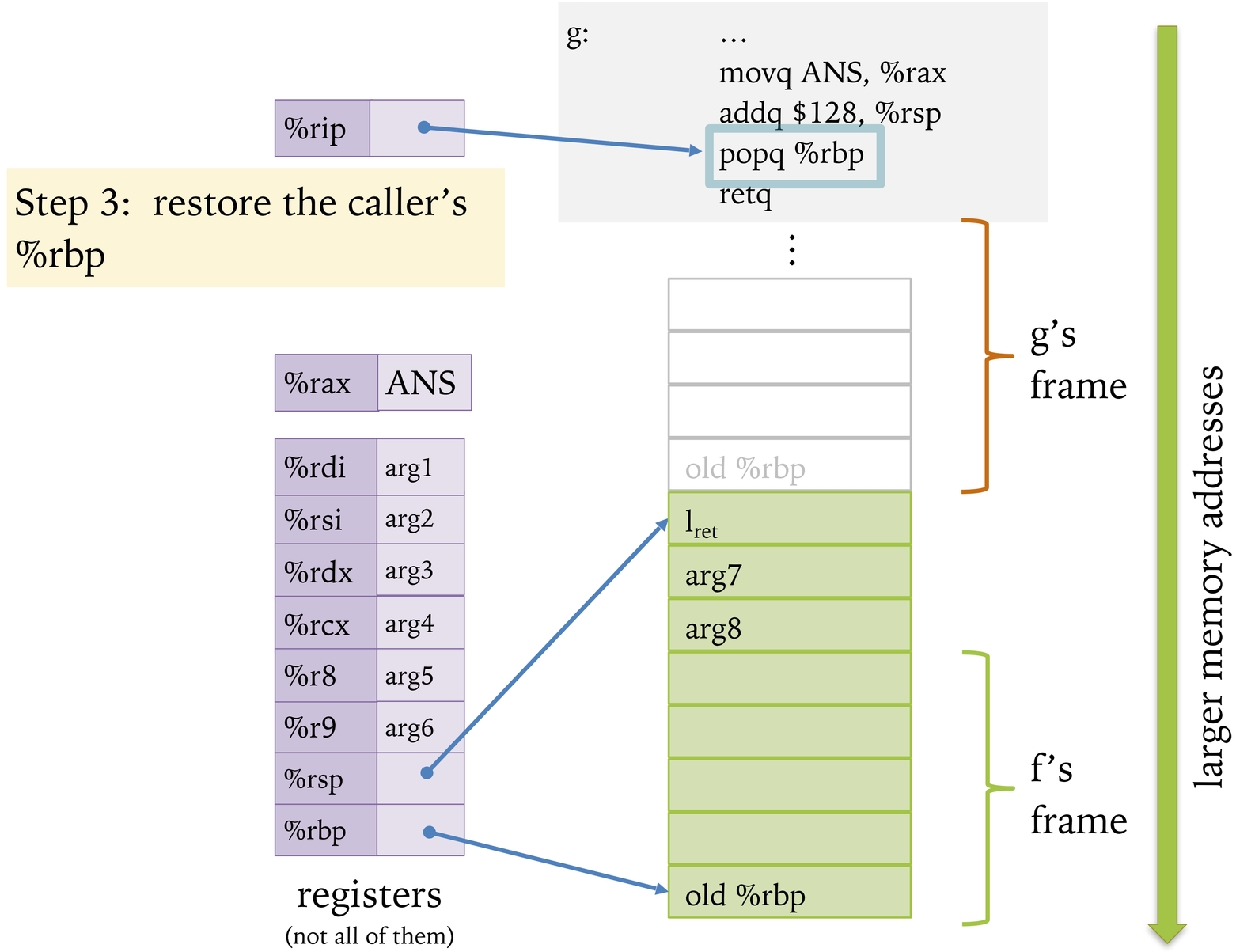
Callee Epilogue (Return Protocol)



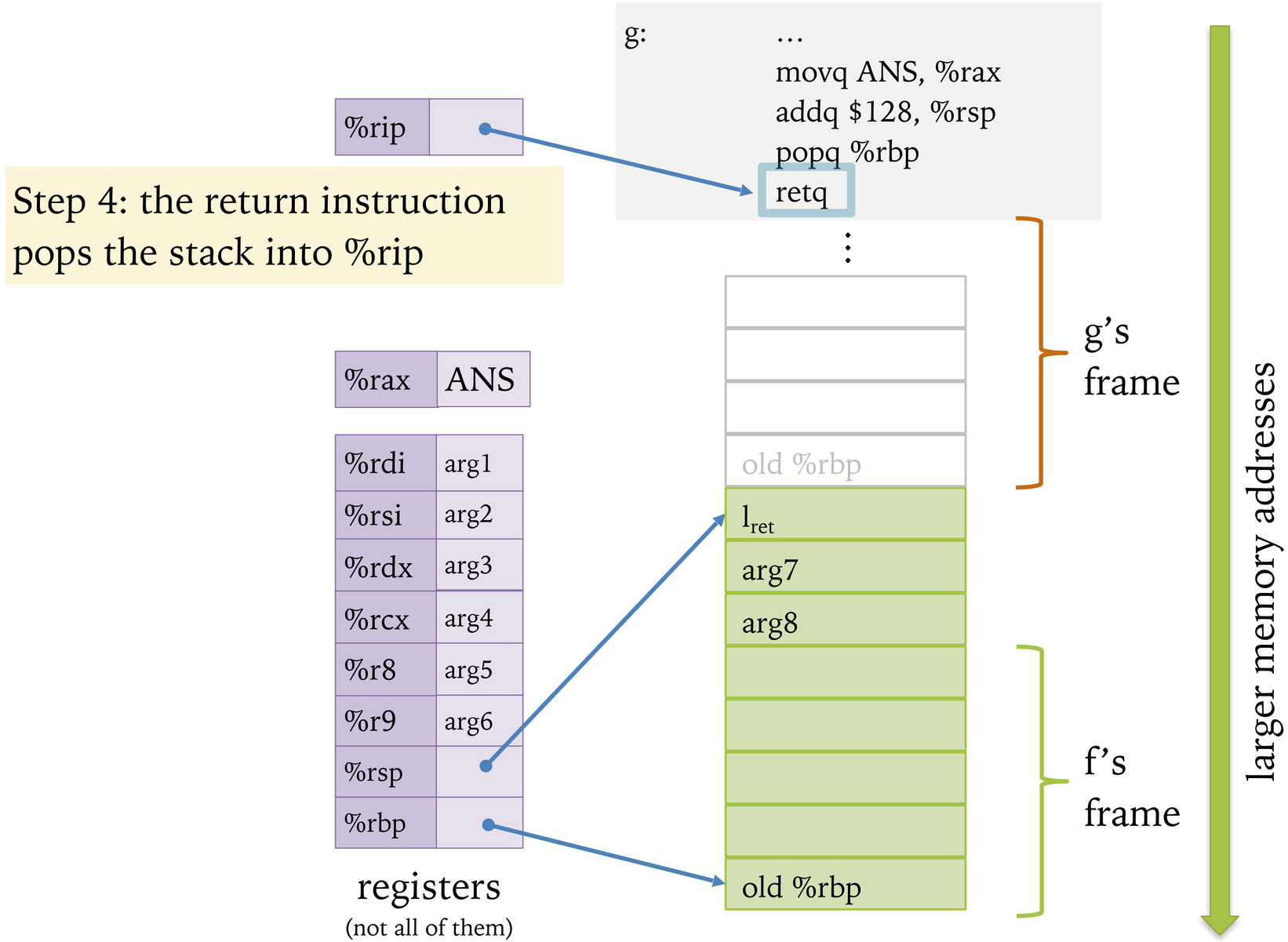
Callee Epilogue (Return Protocol)



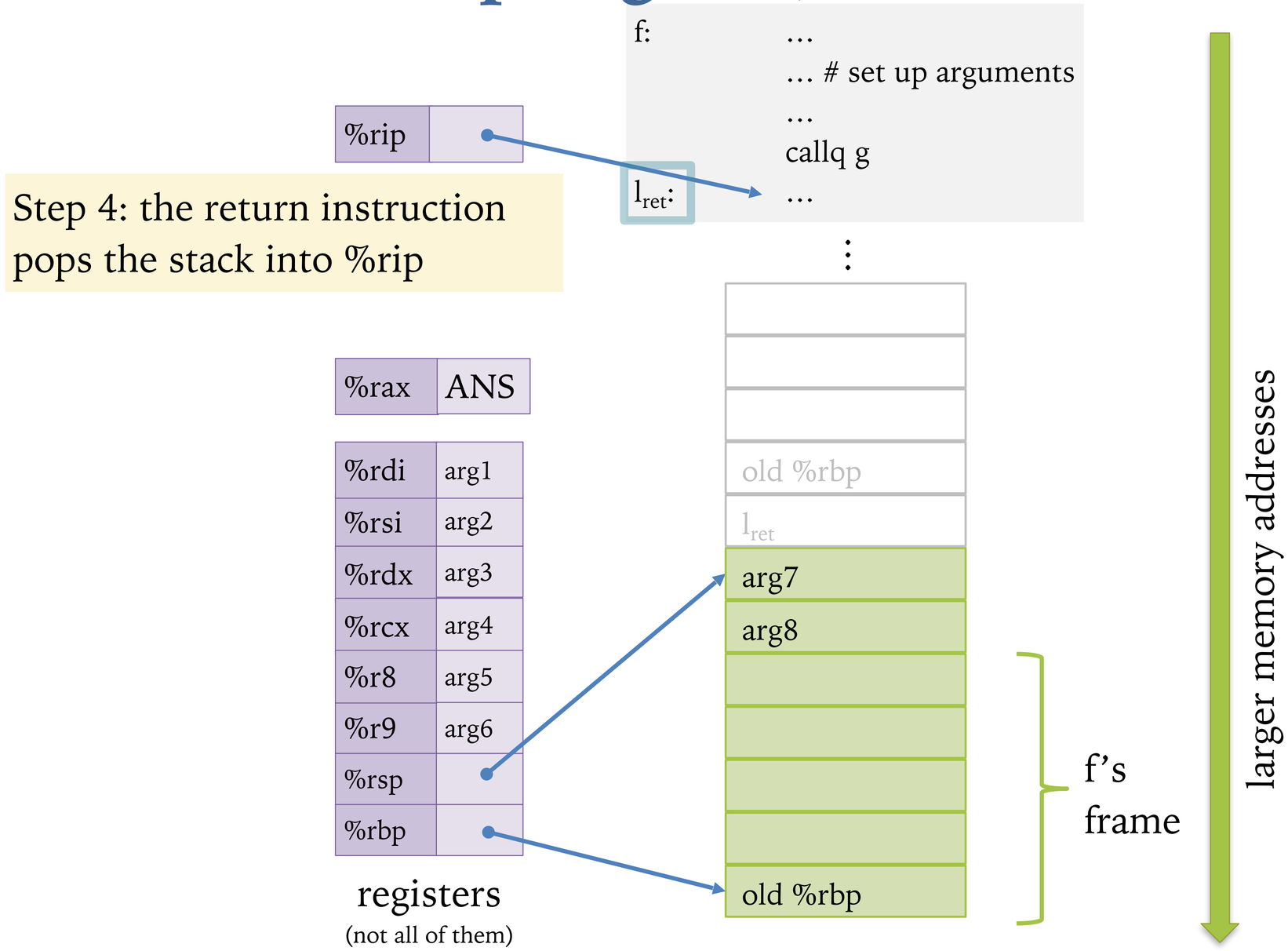
Callee Epilogue (Return Protocol)



Callee Epilogue (Return Protocol)

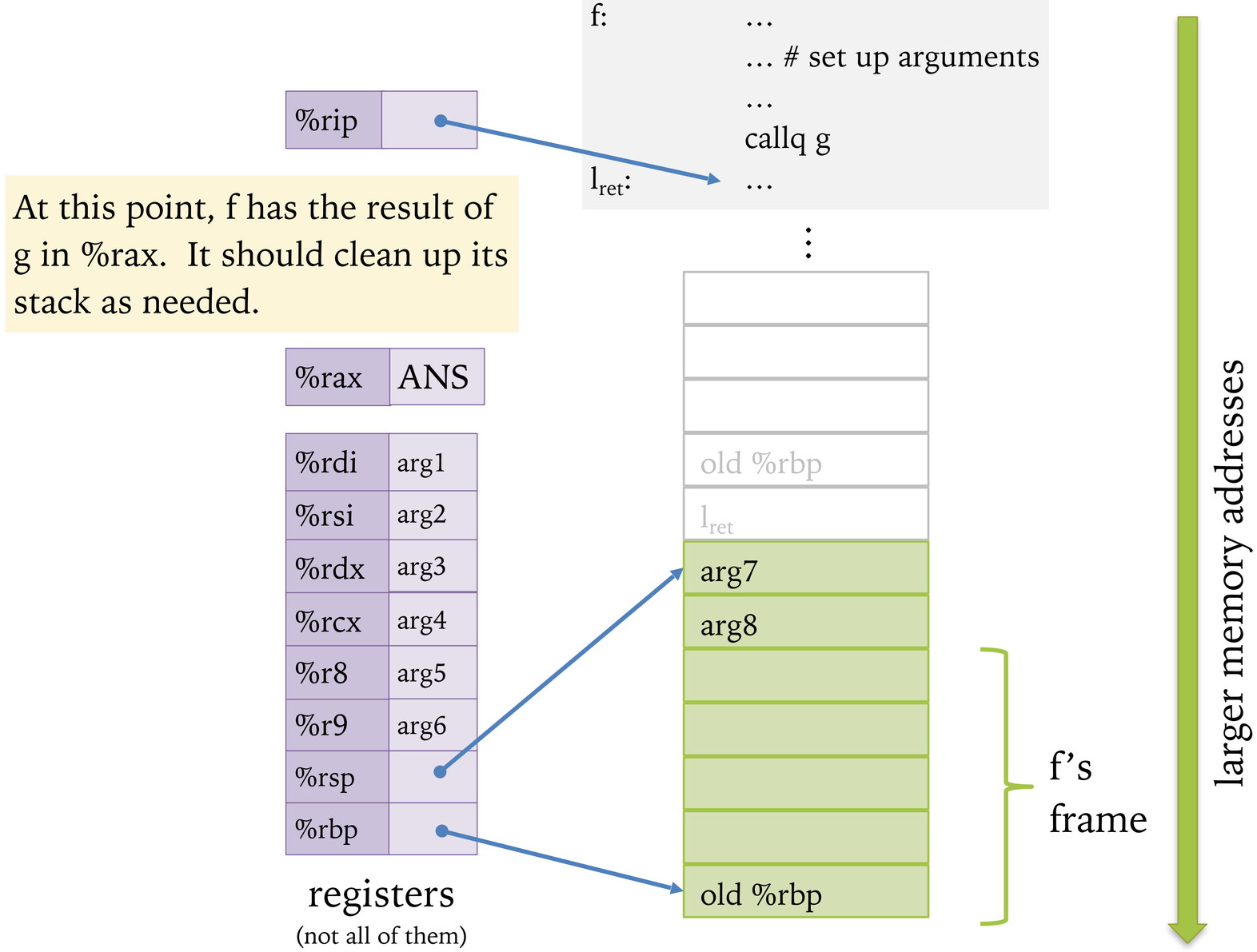


Callee Epilogue (Return Protocol)



Back in f

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



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 optimised away

Announcements

- HW2: X86lite
  - Due: Tuesday, September 10th at 23:59

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

Demo: Directly Compiling Expressions to X86lite

- https://github.com/ysc4230/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?

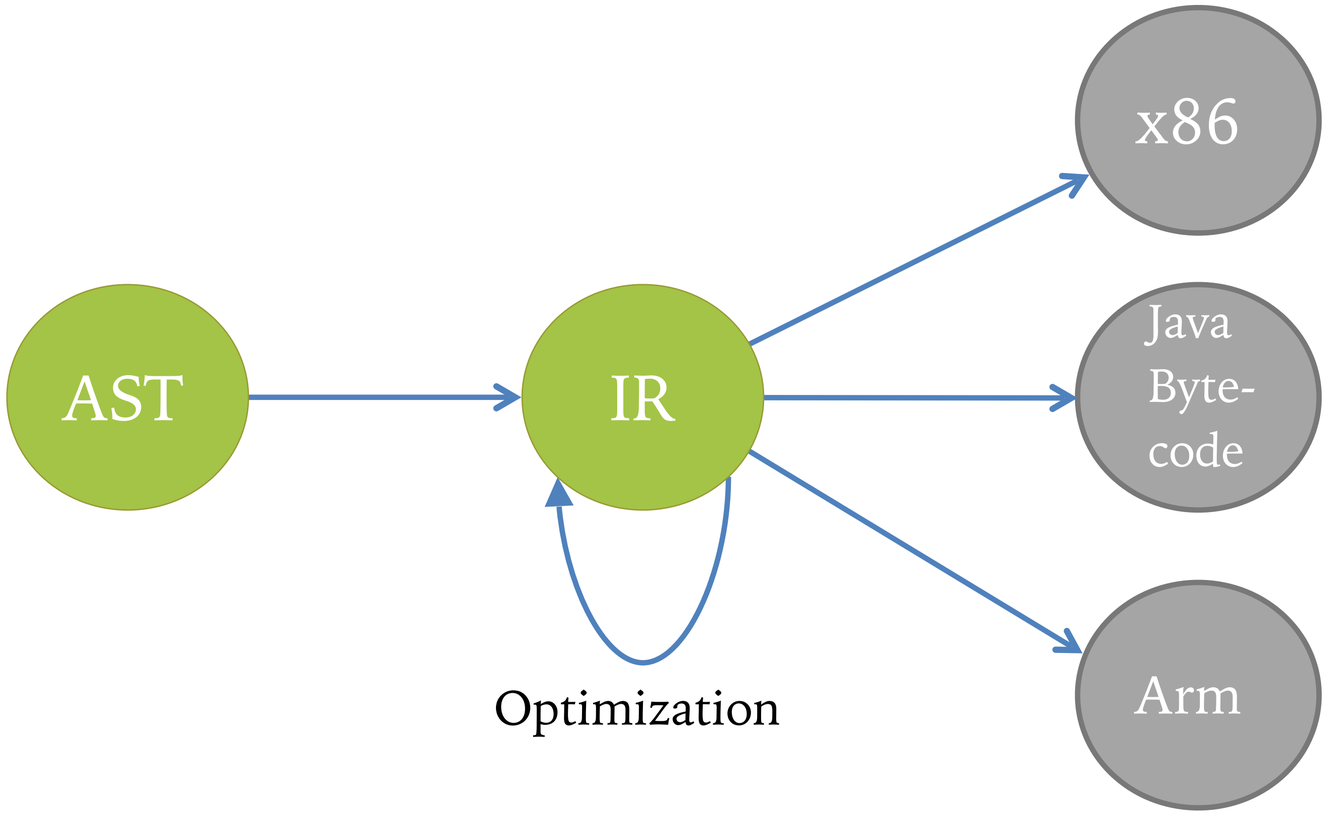
- We have seen a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

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)

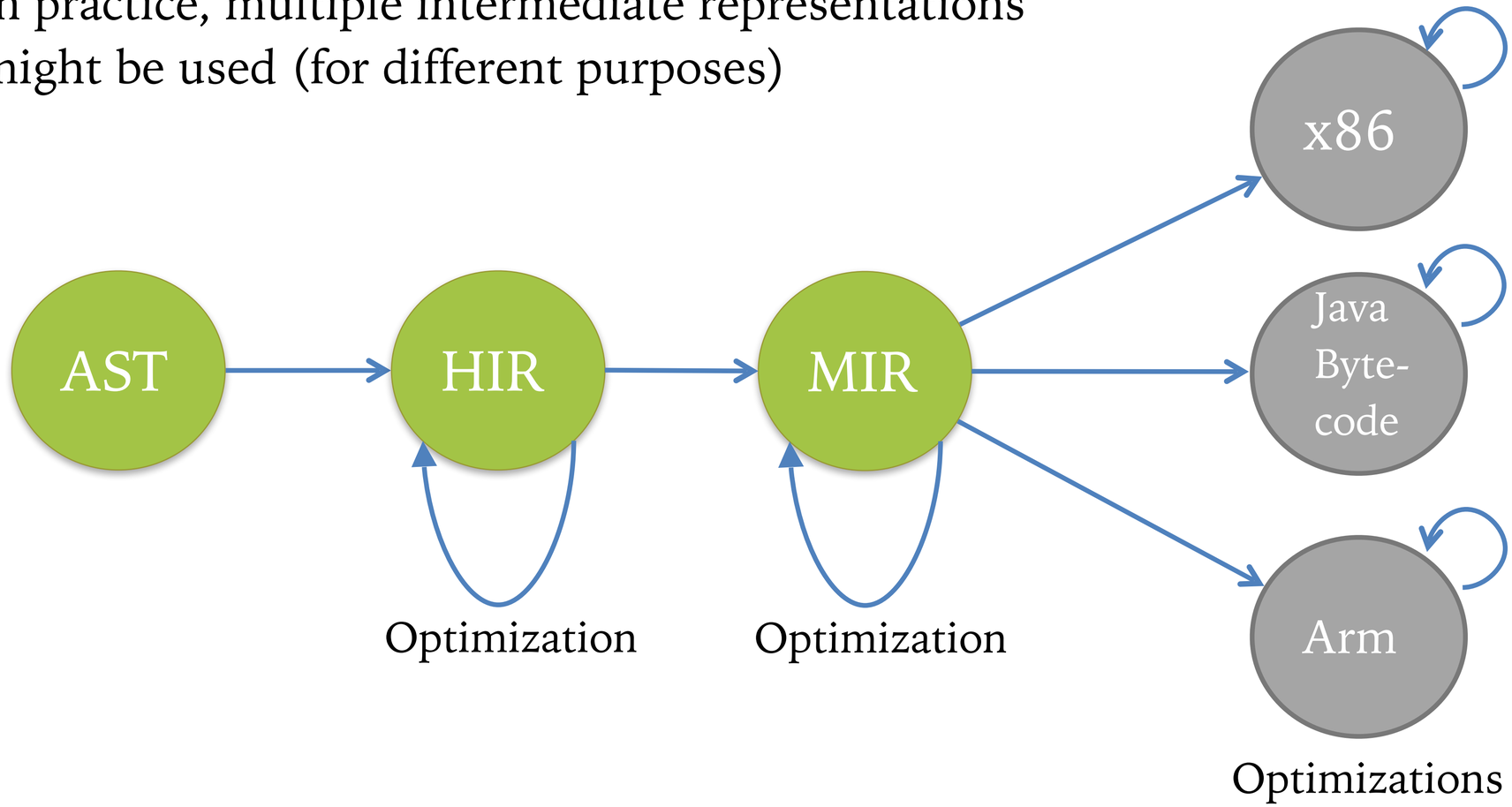
- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization.



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.

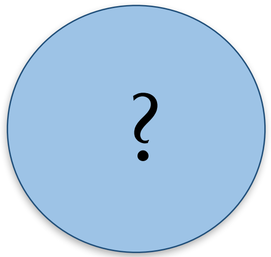
Source

```
((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/ysc4230/week-03-intermediate-2021
- 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