

Lecture 4

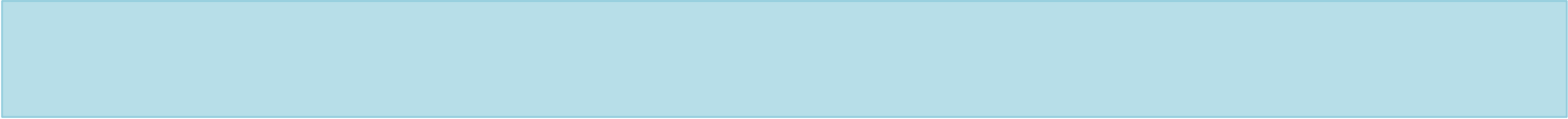
CIS 4521/5521: COMPILERS

CIS 4521/5521 Announcements

- HW1: Hellocaml
 - Due tomorrow night at 10:00pm
- HW2: X86lite
 - Available on the course web pages Thursday
 - Due: Weds. Feb. 12th at 10:00pm
 - Pair-programming project
 - NOTE: much more difficult than HW1, so please start early!

Plan for Today

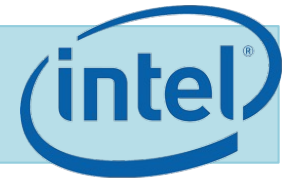
- Wrap up `translate.ml`
- X86 architecture



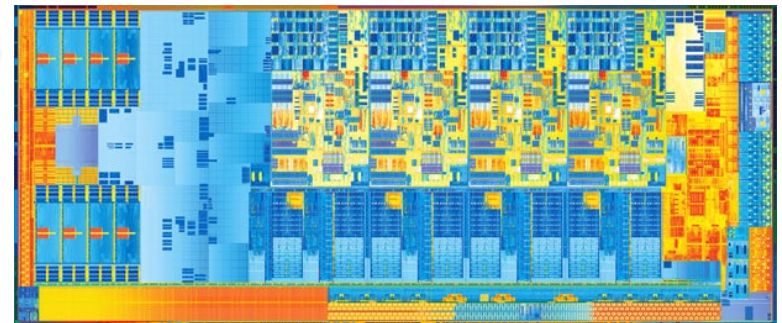
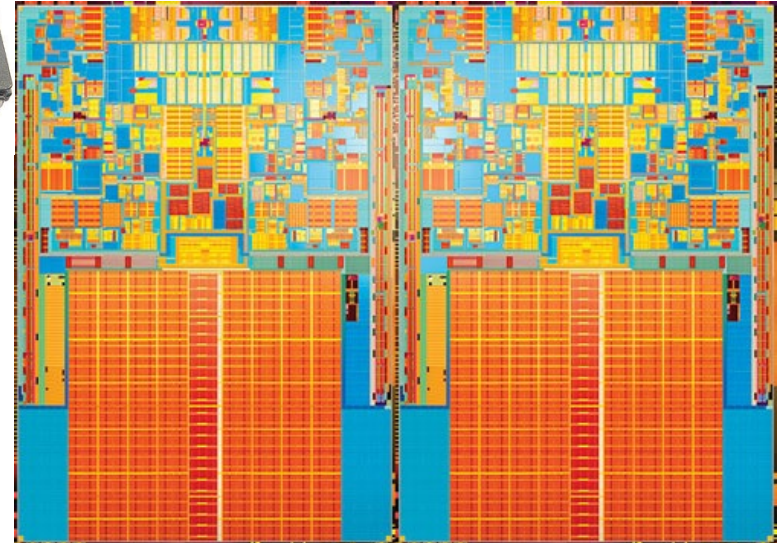
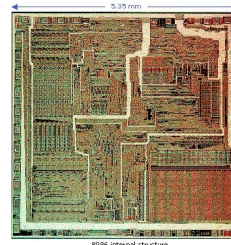
The target architecture for CIS4521/5521

X86LITE

Intel's X86 Architecture



- 1978: Intel introduces 8086
- 1982: 80186, 80286
- 1985: 80386
- 1989: 80486 (100MHz, 1 μ m)
- 1993: Pentium
- 1995: Pentium Pro
- 1997: Pentium II/III
- 2000: Pentium 4
- 2003: Pentium M, Intel Core
- 2006: Intel Core 2
- 2008: Intel Core i3/i5/i7
- 2011: SandyBridge / IvyBridge
- 2013: Haswell
- 2014: Broadwell
- 2015: Skylake (core i3/i5/i7/i9) (2.4GHz, 14nm)
- 2016: Xeon Phi
- 2017-2021: Intel Sunny Cove (Lake-U and Y), Cypress Cove (Rocket Lake)
- 2020: Willow Cove
- 2022: Raptor Cove
- 2024: Lunar Lake, Core Ultra



My first Intel was an 80386 purchased in 1992.

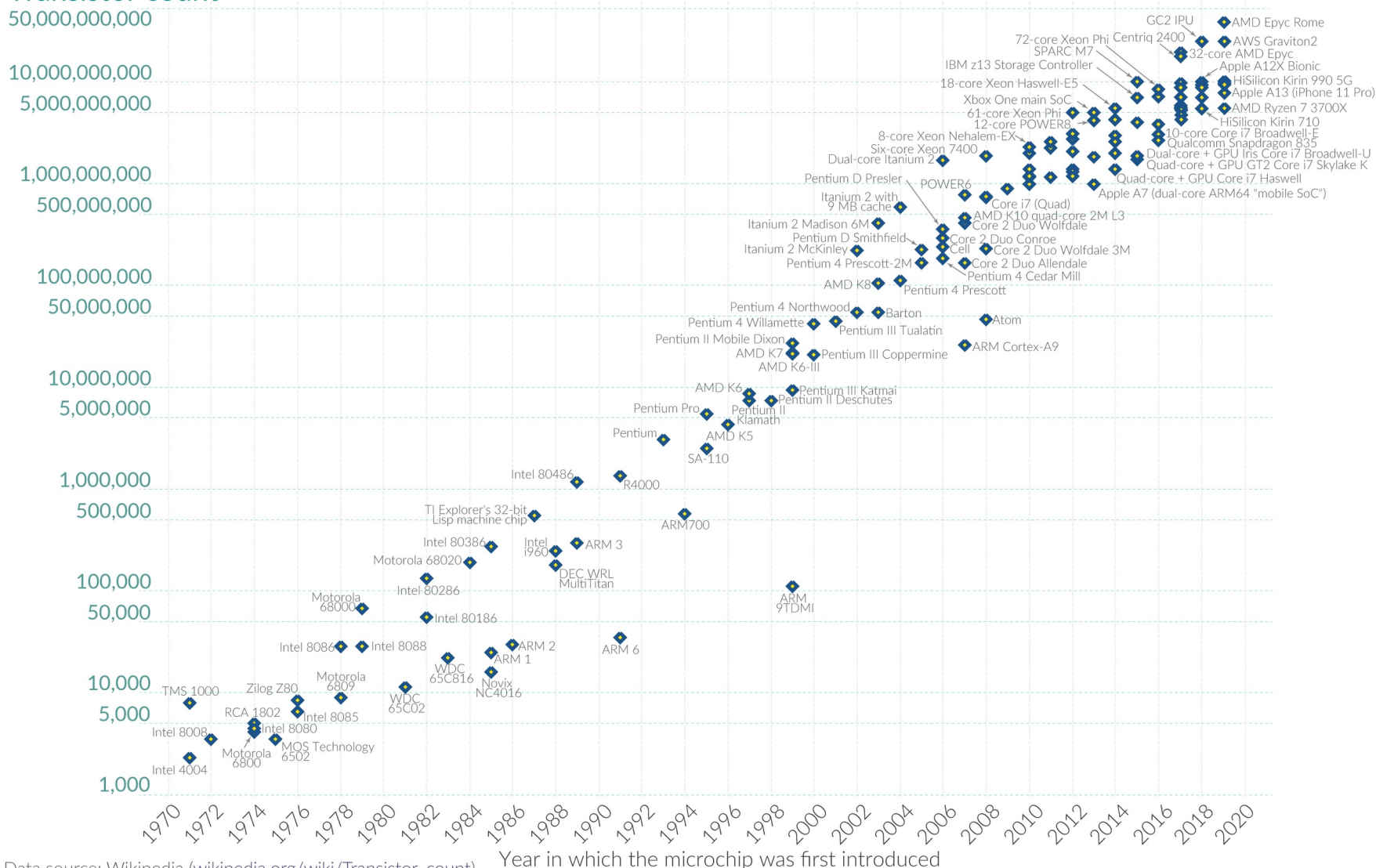
X86 Evolution & Moore's Law

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World
in Data

Transistor count



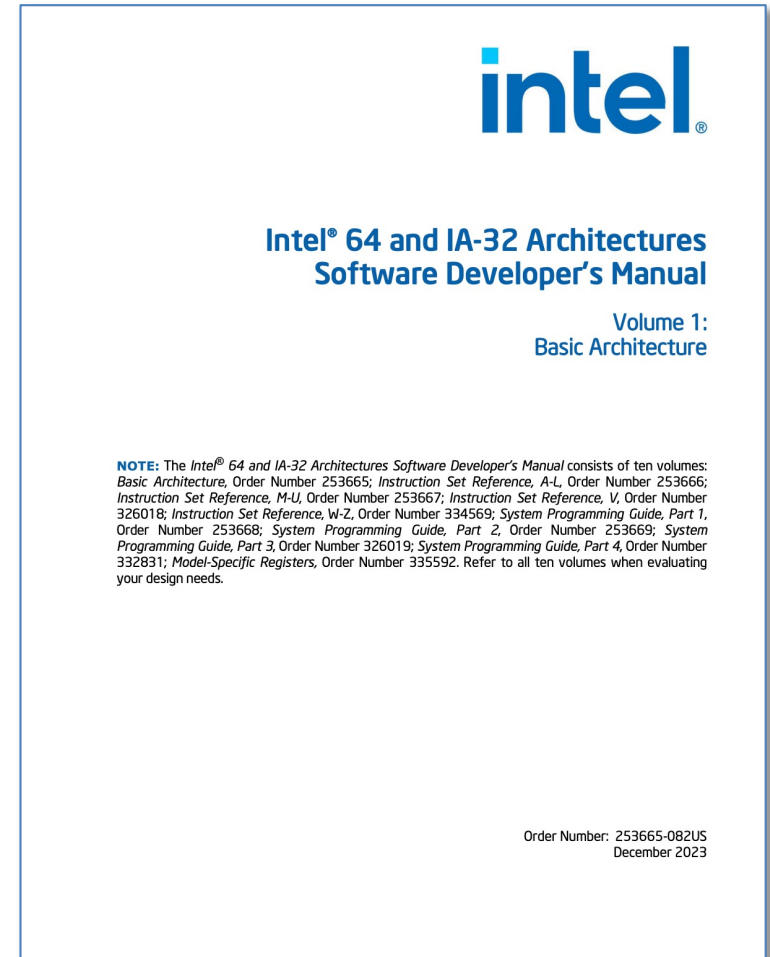
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under [CC-BY](#) by the authors Hannah Ritchie and Max Roser.

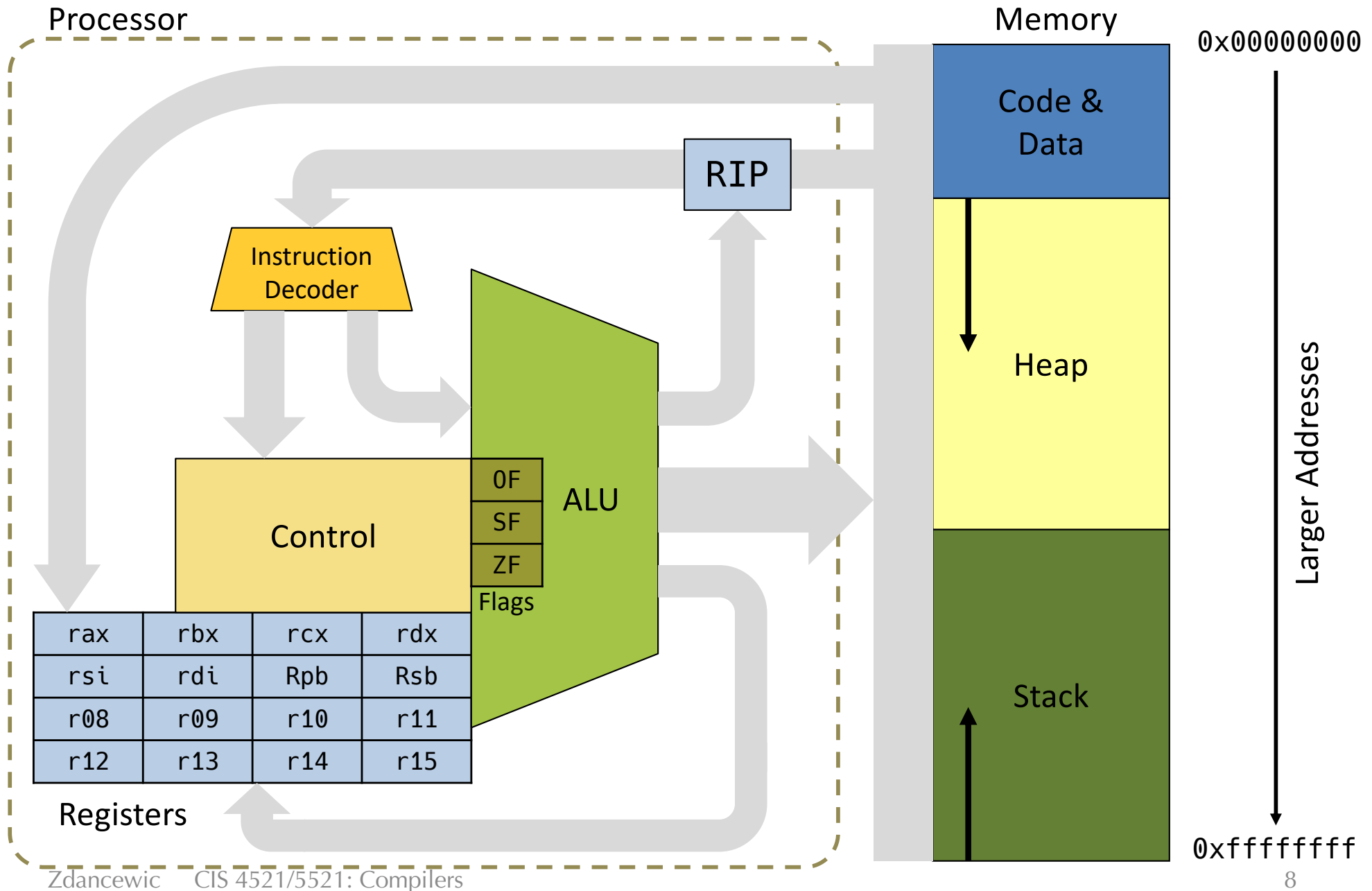
X86 vs. X86lite

- X86 assembly is *very* complicated:
 - 8-, 16-, 32-, 64-bit values + floating points, etc.
 - Intel 64 and IA 32 architectures have a *huge* number of functions
 - "CISC" - complex instructions
 - Machine code: instructions range in size from 1 byte to 17 bytes
 - Lots of hold-over design decisions for backwards compatibility
 - Hard to understand
- X86lite is a *very* simple subset of X86:
 - Only 64-bit signed integers (no floating point, no 16bit, no ...)
 - Only about 20 instructions
 - Sufficient as a target language for general-purpose computing



~500 Pages for
Volume 1
"Basic Architecture"

X86 Schematic



X86lite Machine State: Registers

- Register File: 16 64-bit registers
 - `rax` general purpose accumulator
 - `rbx` base register, pointer to data
 - `rcx` counter register for strings & loops
 - `rdx` data register for I/O
 - `rsi` pointer register, string source register
 - `rdi` pointer register, string destination register
 - `rbp` base pointer, points to the stack frame
 - `rsp` stack pointer, points to the top of the stack
 - `r08-r15` general purpose registers
- `rip` a “virtual” register, points to the current instruction
 - `rip` is modified only indirectly via jumps and return, but it can be mentioned as a register elsewhere

Simplest instruction: mov

- `movq SRC, DEST` copy SRC into DEST
- Here, DEST and SRC are *operands*
- DEST is treated as a *location*
 - A location can be a register or a memory address
- SRC is treated as a *value*
 - A value is the *contents* of a register or memory address
 - A value can also be an *immediate* (constant) or a label
- `movq $4, %rax` // move the 64-bit immediate value 4 into rax
- `movq %rbx, %rax` // move the contents of rbx into rax

A Note About Instruction Syntax

- X86 presented in *two* common syntax formats

- AT&T notation: source *before* destination

- Prevalent in the Unix/Mac ecosystems
- Immediate values prefixed with '\$'
- Registers prefixed with '%'
- Mnemonic suffixes: **movq** vs. **mov**
 - q = quadword (4 words)
 - l = long (2 words)
 - w = word
 - b = byte

```
movq $5, %rax
```

```
movl $5, %eax
```

src dest

Note: X86lite uses the AT&T notation and the 64-bit only version of the instructions and registers.

- Intel notation: destination *before* source
 - Used in the Intel specification / manuals
 - Prevalent in the Windows ecosystem
 - Instruction variant determined by register name

```
mov rax, 5
```

```
mov eax, 5
```

dest src

X86lite Arithmetic instructions

- `negq DEST` two's complement negation
- `addq SRC, DEST` $DEST \leftarrow DEST + SRC$
- `subq SRC, DEST` $DEST \leftarrow DEST - SRC$
- `imulq SRC, Reg` $Reg \leftarrow Reg * SRC$ (truncated 128-bit mult.)

Examples as written in:

```
addq %rbx, %rax    // rax ← rax + rbx
subq $4, rsp       // rsp ← rsp - 4
```

- Note: `Reg` (in `imulq`) must be a register, not a memory address

X86lite Logic/Bit manipulation Operations

- `notq DEST` logical negation
- `andq SRC, DEST` $DEST \leftarrow DEST \ \&\& \ SRC$
- `orq SRC, DEST` $DEST \leftarrow DEST \ || \ SRC$
- `xorq SRC, DEST` $DEST \leftarrow DEST \ xor \ SRC$

- `sarq Amt, DEST` $DEST \leftarrow DEST \ >> \ amt$ (arithmetic shift right)
- `shlq Amt, DEST` $DEST \leftarrow DEST \ << \ amt$ (arithmetic shift left)
- `shrq Amt, DEST` $DEST \leftarrow DEST \ >>> \ amt$ (bitwise shift right)

X86 Operands

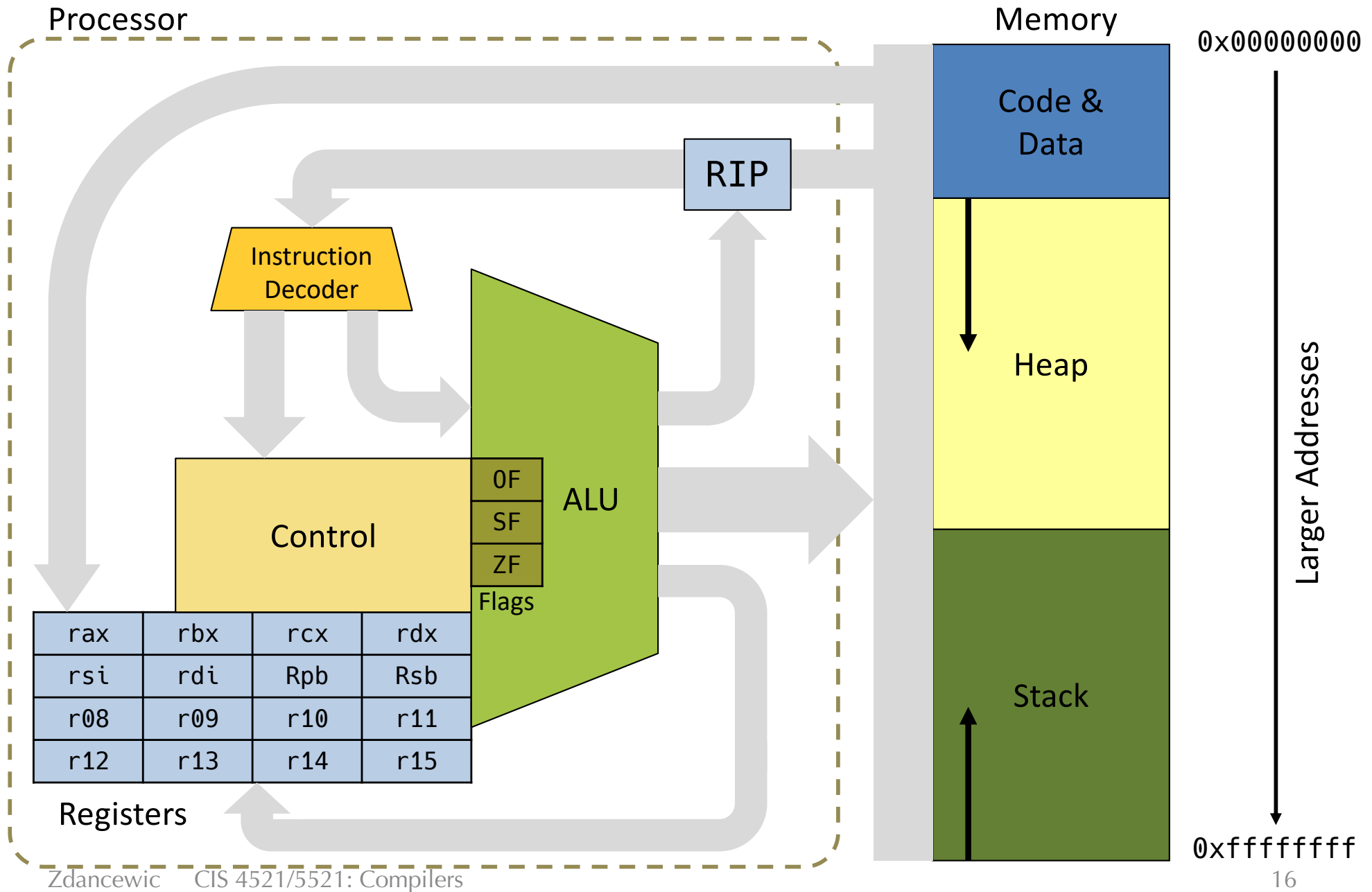
- Operands are the values operated on by the assembly instructions
- Imm 64-bit literal signed integer “immediate”
- Lbl a “label” representing a machine address
the assembler/linker/loader resolve labels
- Reg One of the 16 registers, the value of a register is
its contents
- Ind [base:Reg][index:Reg,scale:int32][disp]
machine address (see next slide)

X86 Addressing

- In general, there are three components of an indirect address
 - **Base:** a machine address stored in a register
 - **Index * scale:** a variable offset from the base
 - **Disp:** a constant offset (displacement) from the base
- $\text{addr(ind)} = \text{Base} + [\text{Index} \times \text{scale}] + \text{Disp}$
 - When used as a *location*, ind denotes the address addr(ind)
 - When used as a *value*, ind denotes $\text{Mem}[\text{addr(ind)}]$, the contents of the memory address
- Example: $-4(\%rsp)$ denotes address: $rsp - 4$
- Example: $(\%rax, \%rcx, 4)$ denotes address: $rax + 4 * rcx$
- Example: $12(\%rax, \%rcx, 4)$ denotes address: $rax + 4 * rcx + 12$
- Note: Index cannot be `rsp`

Note: X86lite does not need this full generality. It does not use $\text{index} \times \text{scale}$.

X86 Schematic



X86lite Memory Model

- The X86lite memory consists of 2^{64} bytes numbered `0x00000000` through `0xffffffff`.
- X86lite treats the memory as consisting of 64-bit (8-byte) quadwords.
- Therefore: legal X86lite memory addresses consist of 64-bit, quadword-aligned pointers.
 - All memory addresses are evenly divisible by 8
- `leaq Ind, DEST` $\text{DEST} \leftarrow \text{addr}(\text{Ind})$ loads a pointer into DEST
- By convention, there is a stack that grows from high addresses to low addresses
- The register `rsp` points to the top of the stack
 - `pushq SRC` $\text{rsp} \leftarrow \text{rsp} - 8; \text{Mem}[\text{rsp}] \leftarrow \text{SRC}$
 - `popq DEST` $\text{DEST} \leftarrow \text{Mem}[\text{rsp}]; \text{rsp} \leftarrow \text{rsp} + 8$

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 of 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. $(SF = OF) \ \&\& \ \text{not}(ZF)$
 - **lt** less than holds when $SF \neq OF$
 - Equivalently: $((SF \ \&\& \ \text{not } OF) \ || \ (\text{not } SF \ \&\& \ OF))$
 - **ge** greater or equal holds when (not **lt**) holds, i.e. $(SF = OF)$
 - **le** less than or equal holds when $SF \neq OF$ 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

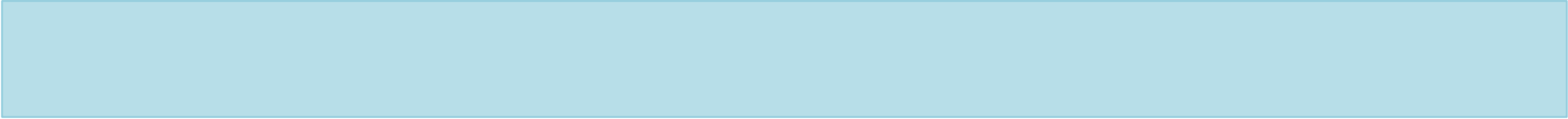
- `cmpq SRC1, SRC2` Compute $SRC2 - SRC1$, set condition flags
- `setbCC DEST` $DEST's\ lower\ byte \leftarrow$ if CC then 1 else 0
- `jCC SRC` $rip \leftarrow$ if CC then SRC else fallthrough
- Example:

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

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

Jumps, Call and Return

- `jmp SRC` `rip` $\leftarrow$ SRC Jump to location in SRC
- `callq SRC` Push `rip`; `rip` $\leftarrow$ 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

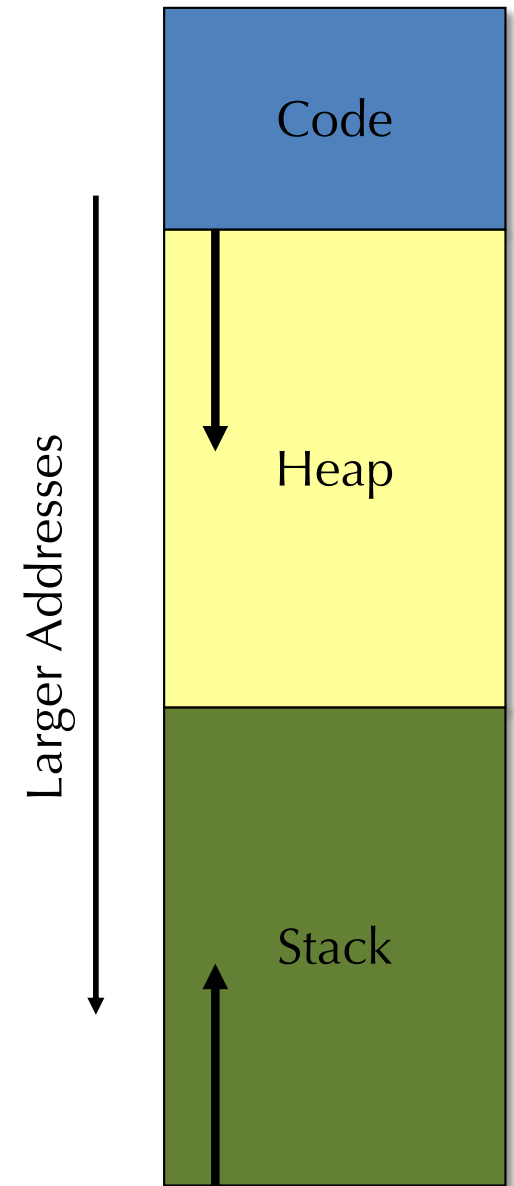
Compiling, Linking, Running

- 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 executable:
`./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.

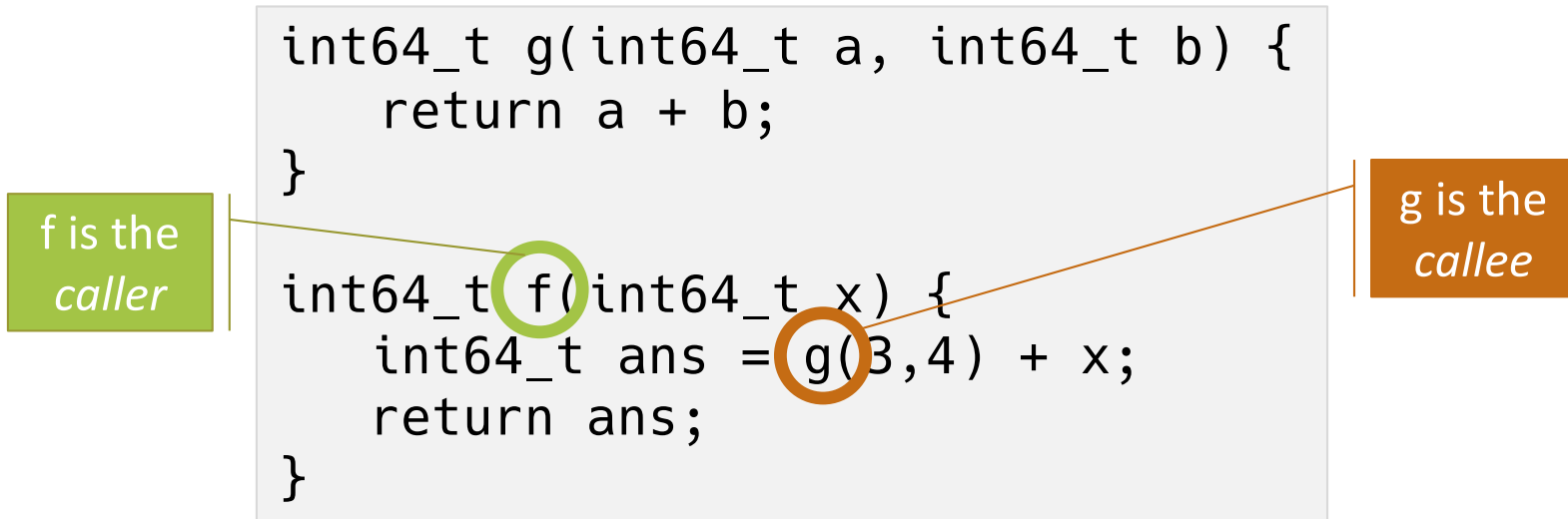


Local/Temporary Variable Storage

- 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*

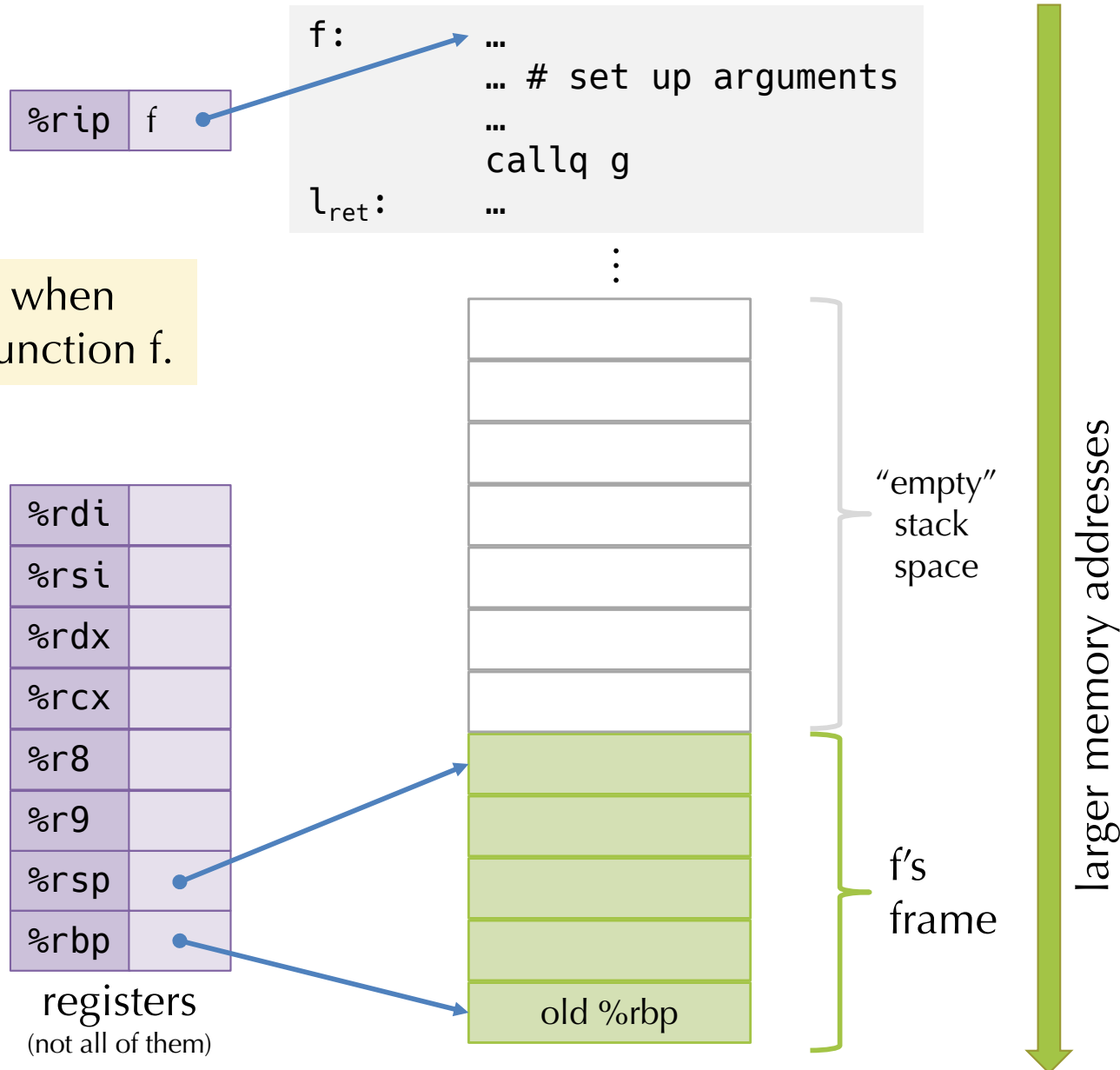
Calling Conventions

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



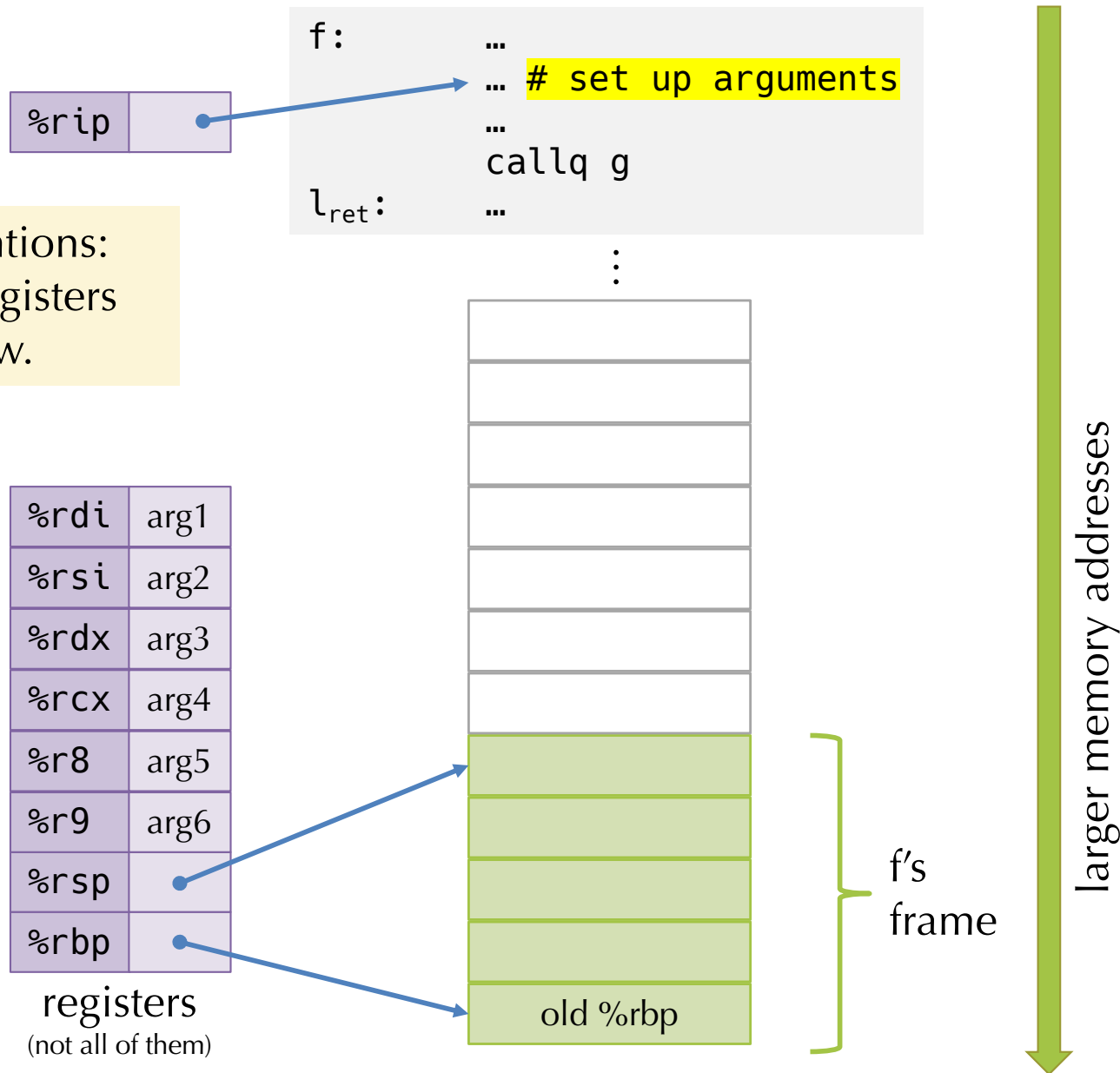
- Designate registers either:
 - **Caller Save** – e.g., freely usable by the called code
 - **Callee Save** – e.g., must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
 - Caller cleans up
 - Callee cleans up (makes supporting variable number of arguments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

x64 Calling Conventions: Caller Protocol



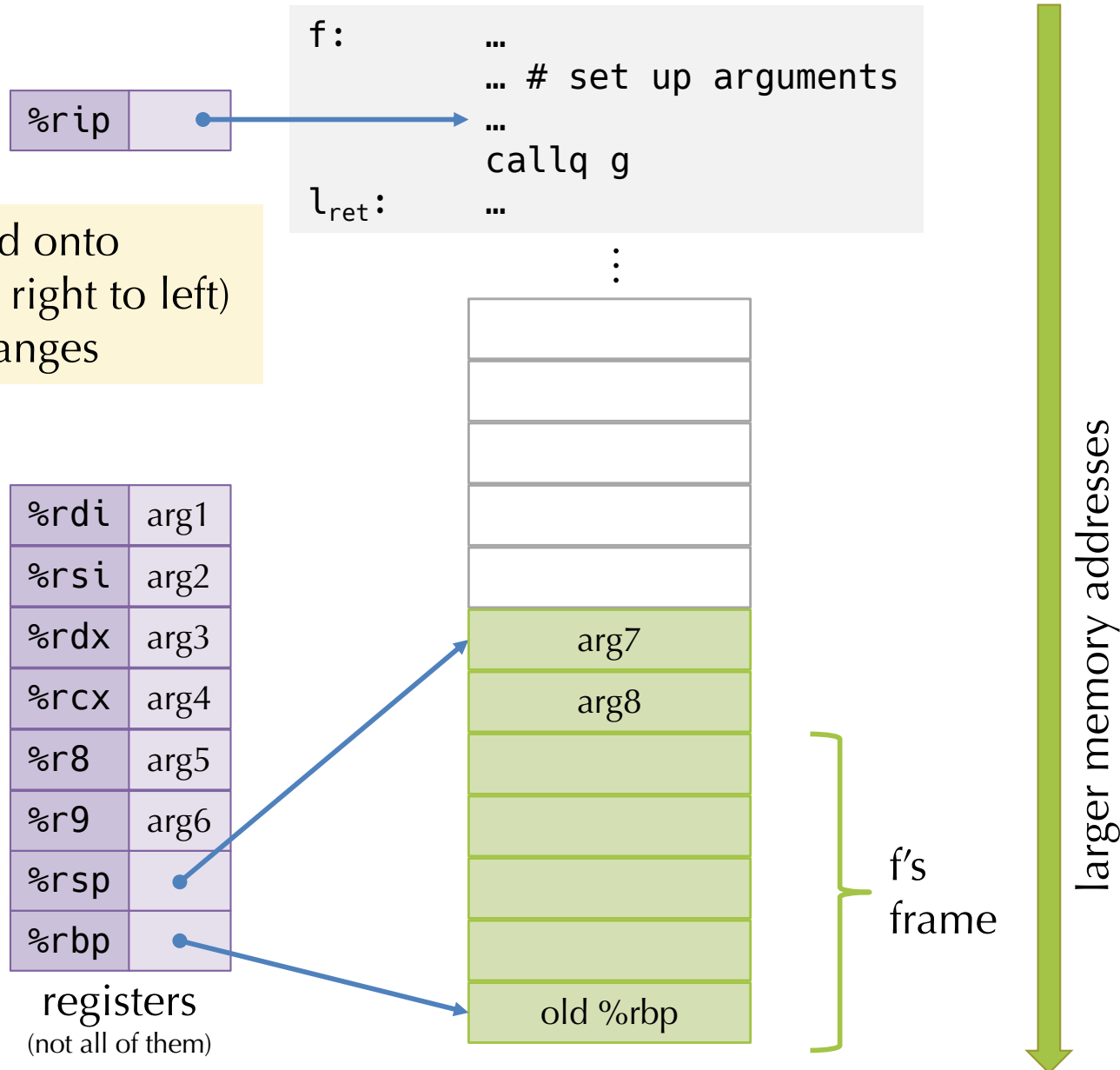
x64 Calling Conventions: Caller Protocol

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



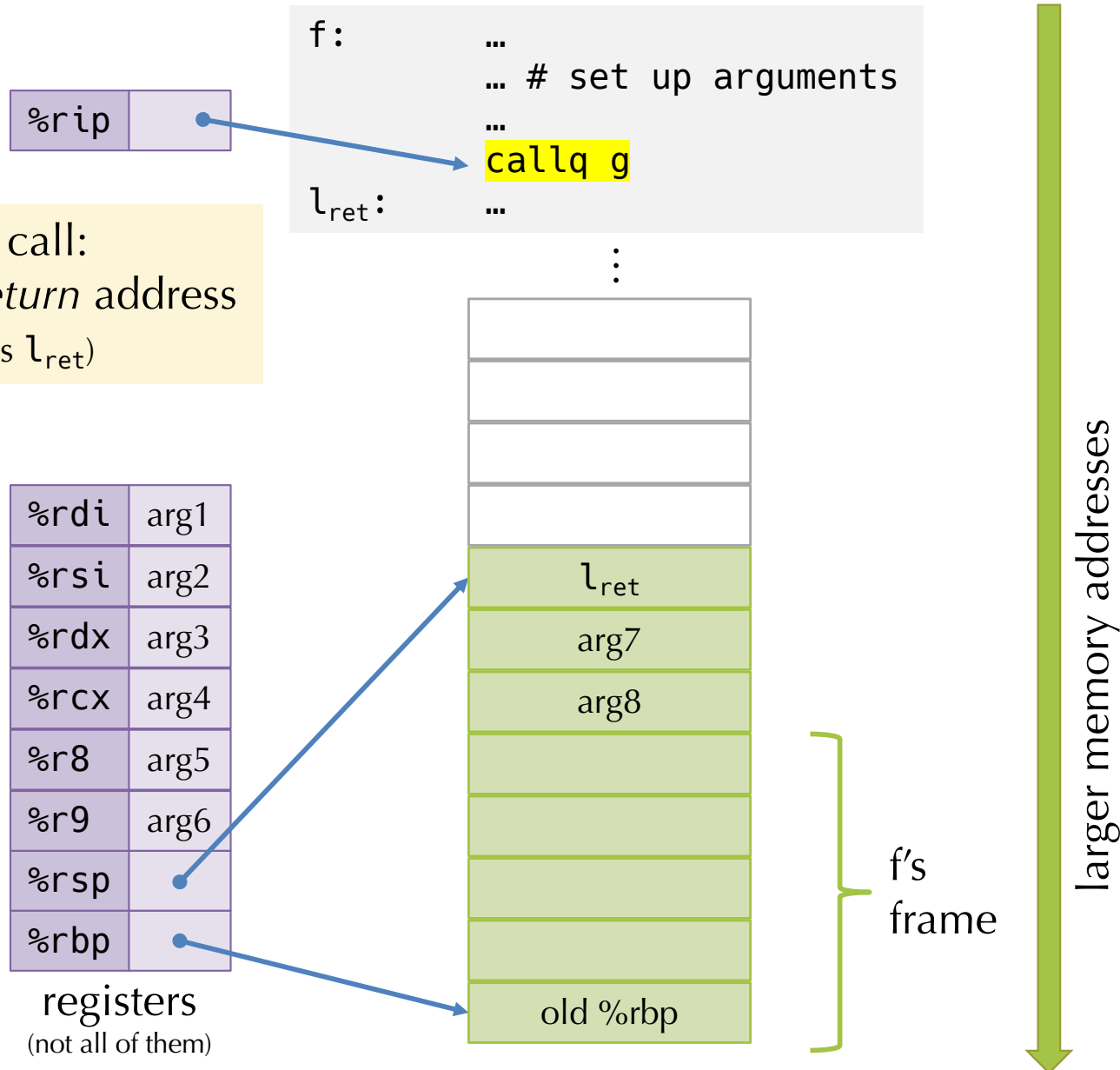
x64 Calling Conventions: Caller Protocol

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

%rip

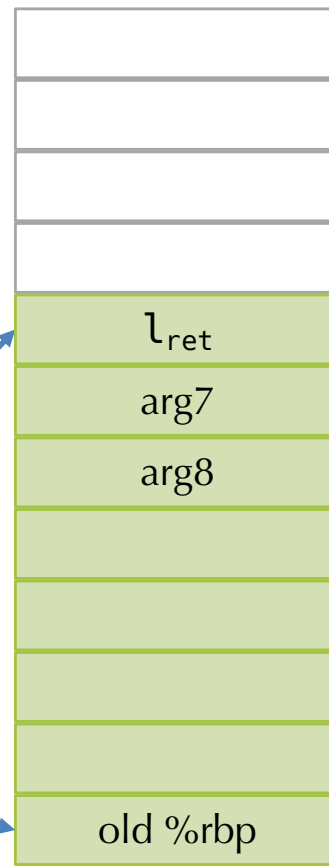
g:

```
pushq %rbp
movq %rsp, %rbp
subq $128, %rsp
...
```

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

%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



f's
frame

larger memory addresses

callee function prologue

g:
pushq %rbp
movq %rsp, %rbp
subq \$128, %rsp
...

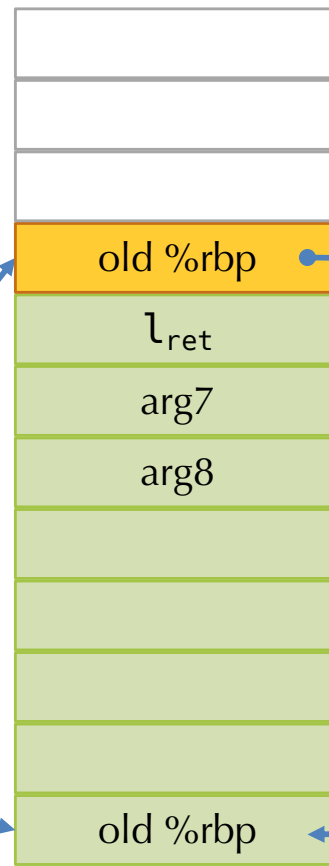
%rip

Callee protocol:

1. store the old %rbp

%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



g's
frame

f's
frame

larger memory addresses

callee function prologue

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

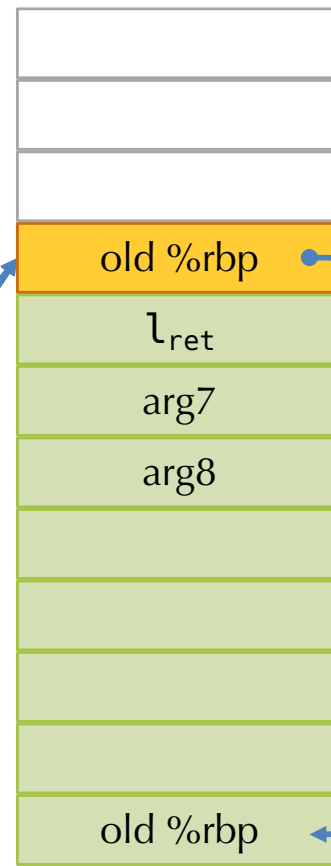
%rip

Callee protocol:

2. adjust the %rbp to point to the new “base”
(%rbp is the “base pointer”)

%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



g's
frame

f's
frame

larger memory addresses

callee function prologue

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

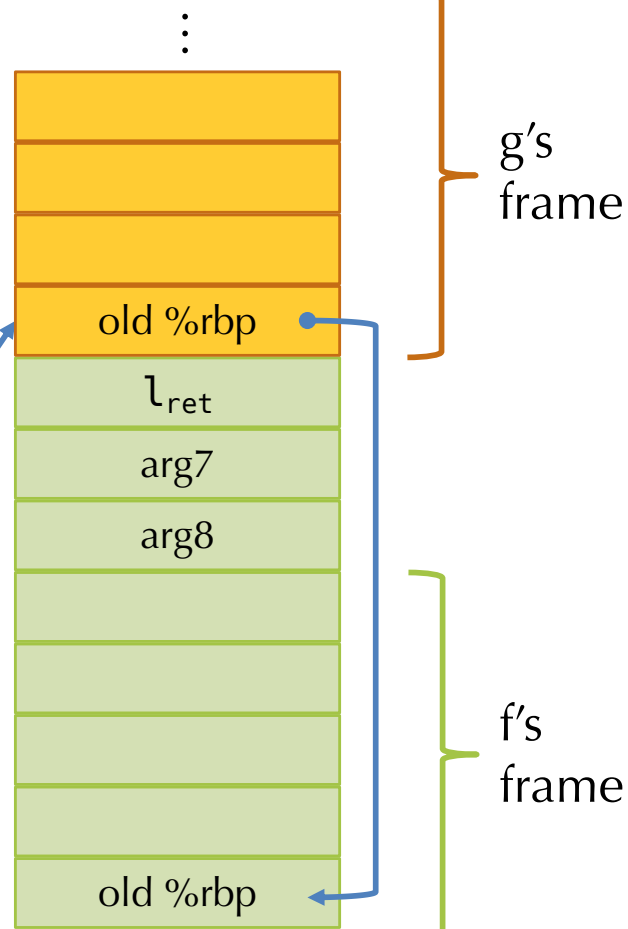
%rip

Callee protocol:

3. allocate 128 bytes of
"scratch" stack space

%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



larger memory addresses

callee invariants: function arguments

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

%rip

Now g's body can run...

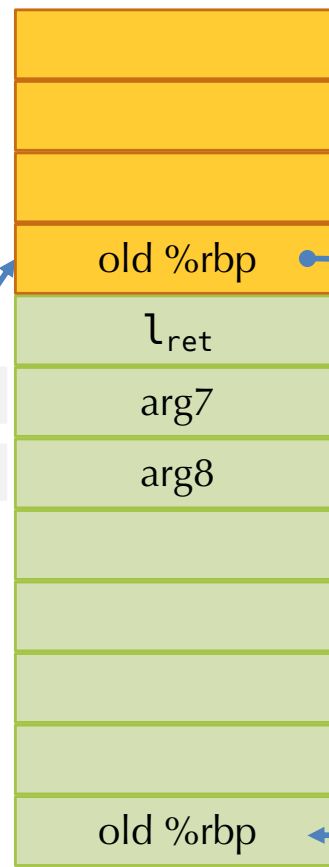
- its arguments are accessible either in registers or as offsets from %rbp

%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)

16(%rbp)

24(%rbp)

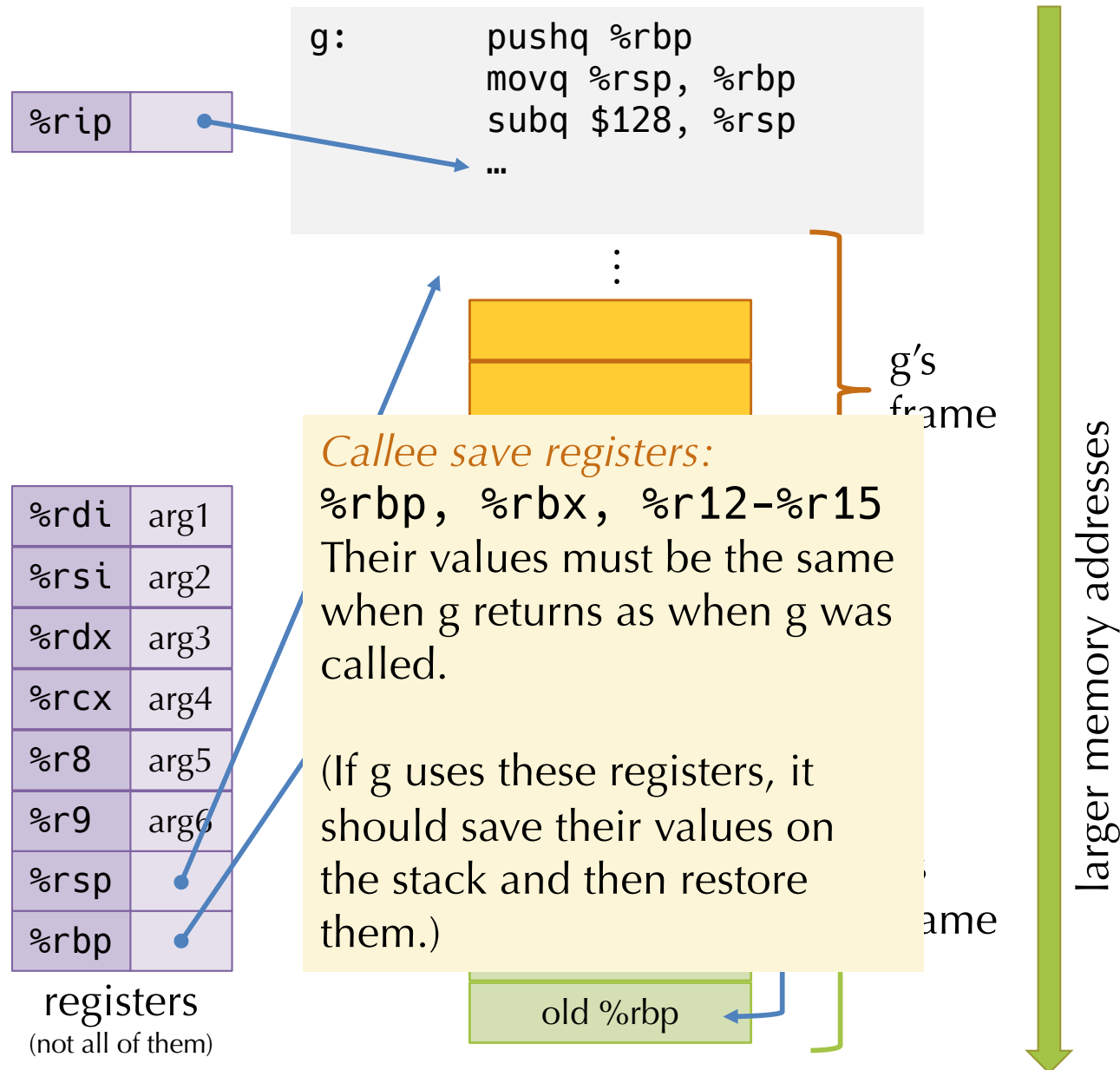


g's
frame

f's
frame

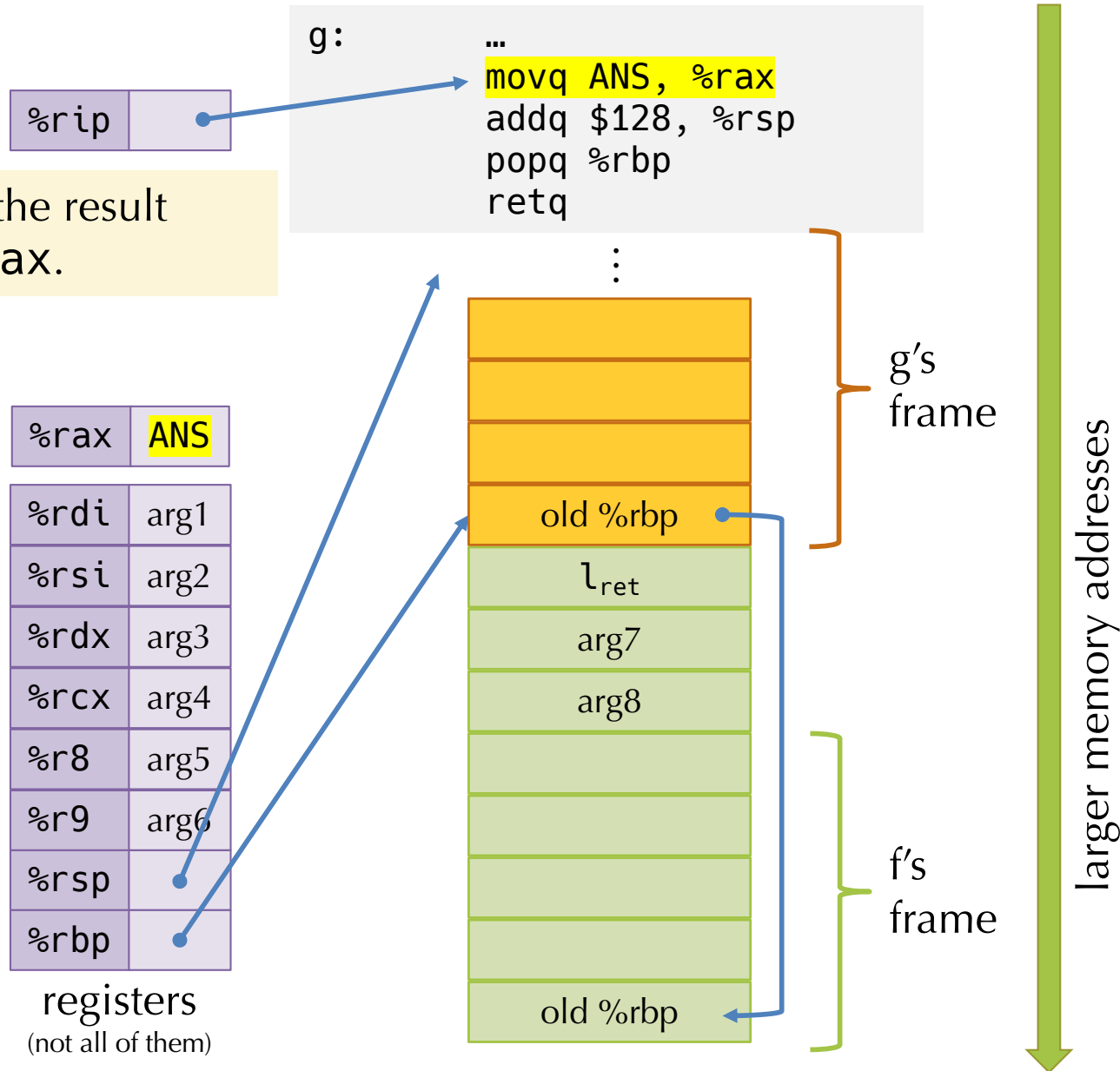
larger memory addresses

callee invariants: callee save registers



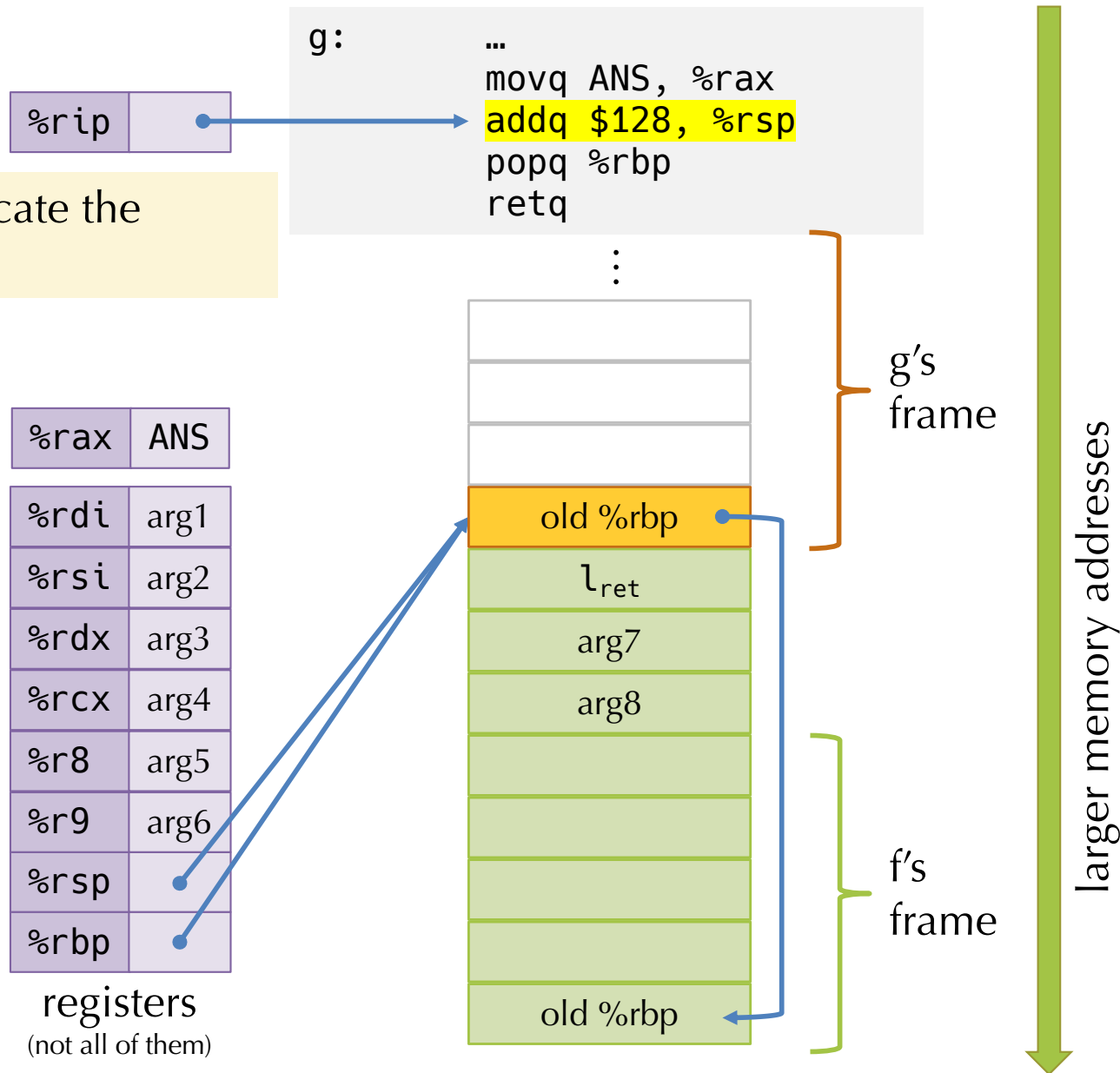
callee epilogue (return protocol)

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



callee epilogue (return protocol)

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

%rax	ANS
%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



g's
frame

f's
frame

larger memory addresses

callee epilogue (return protocol)

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

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

%rax	ANS
%rdi	arg1
%rsi	arg2
%rdx	arg3
%rcx	arg4
%r8	arg5
%r9	arg6
%rsp	
%rbp	

registers
(not all of them)



g's
frame

f's
frame

larger memory addresses

callee epilogue (return protocol)

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

```
f:      ...
      ... # set up arguments
      ...
      callq g
      ...
```

l_{ret}:

 $\mathcal{L}_{\text{ret}}:$

•
•
•

%rax	ANS
------	-----

%rdi	arg1
------	------

%rsi	arg2
------	------

%rdx	arg3
------	------

%rcx	arg4
------	------

%r8	arg5
-----	------

%r9	arg6
-----	------

%rsp

%rbp

registers
(not all of them)

old %rbp

 τ_{ret}

arg7

arg8

old %rbp

f's
frame

larger memory addresses

back in f

%rip

```
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.

%rax	ANS
------	-----

%rdi	arg1
------	------

%rsi	arg2
------	------

%rdx	arg3
------	------

%rcx	arg4
------	------

%r8	arg5
-----	------

%r9	arg6
-----	------

%rsp	
------	---

%rbp

registers

(not all of them)

compilers

old %rbp
l_{ret}
arg7
arg8
old %rbp

old %rbp
l_{ret}
arg7
arg8
old %rbp

old %rbp
l_{ret}
arg7
arg8
old %rbp

old %rbp
l_{ret}
arg7
arg8
old %rbp

old %rbp
l_{ret}
arg7
arg8
old %rbp

f's
frame



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)