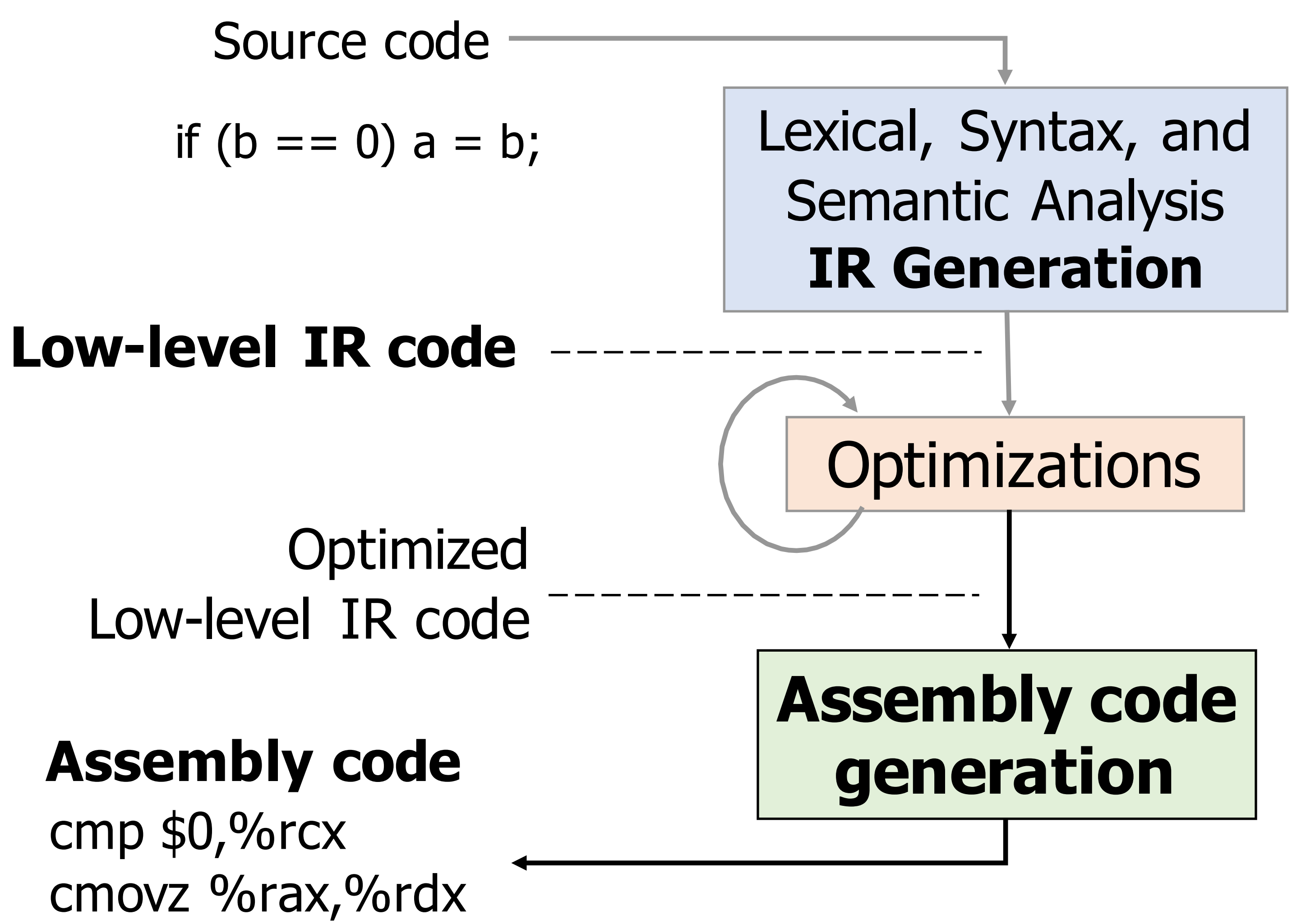
Where We Are



Source code
if (b == 0) a = b;
Lexical, Syntax, and
Semantic Analysis
IR Generation
Low-level IR code
Optimizations
Optimized
Low-level IR code
Assembly code
generation
Assembly code
cmp $0,%rcx
cmovz %rax,%rdx

Low IR to Assembly Translation

```
t3 = this.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
k = t4
```

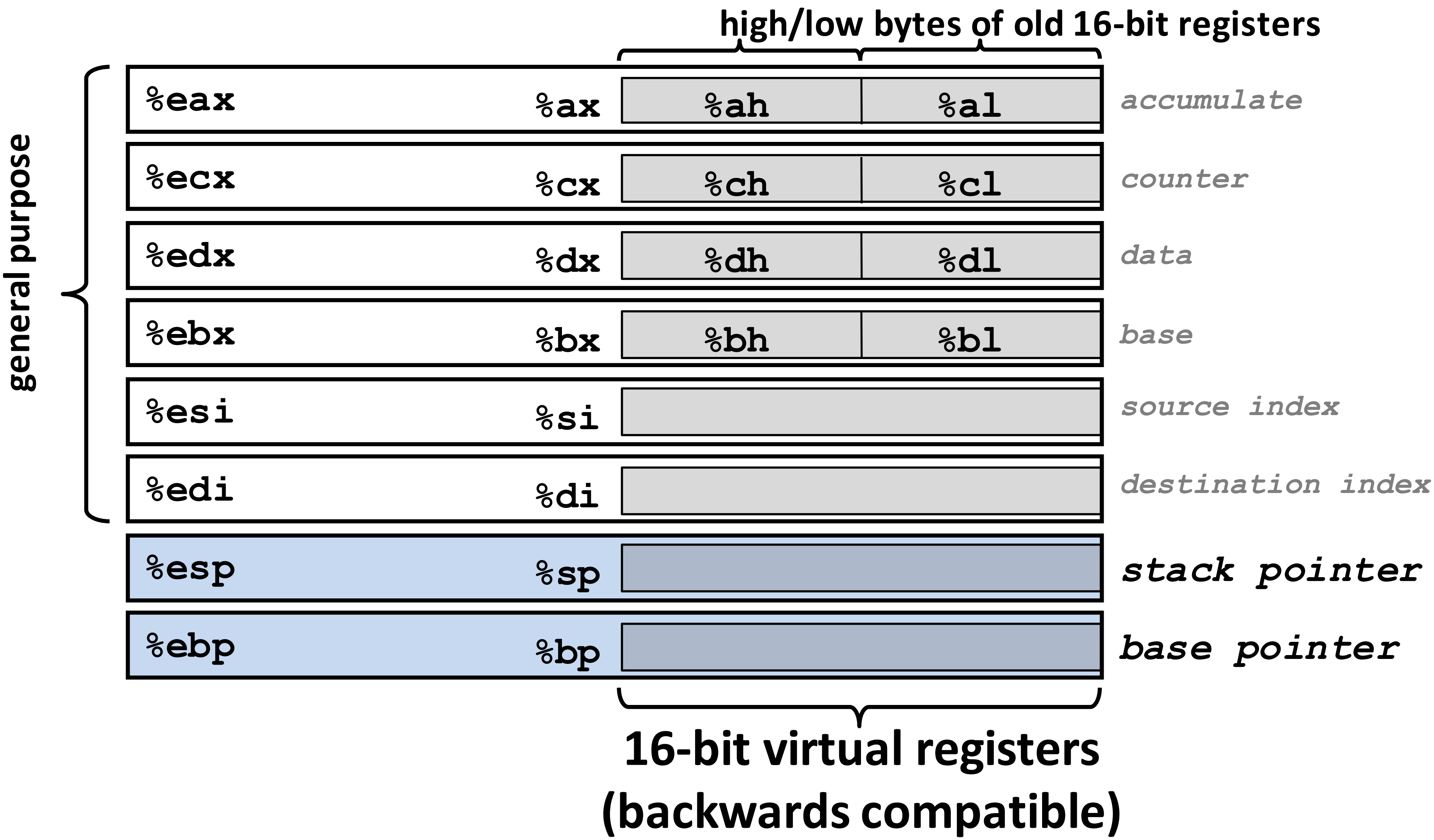
- Low IR code (TAC):
  - Variables (and temporaries)
  - No run-time stack
  - No calling sequences
  - Some abstract set of instructions
- Translation
  - Calling sequences:
    - Translate function calls and returns
    - Manage run-time stack
  - Variables:
    - globals, locals, arguments, etc. assigned memory location
  - Instruction selection:
    - map sets of low level IR instructions to instructions in the target machine

x86-64 crash course

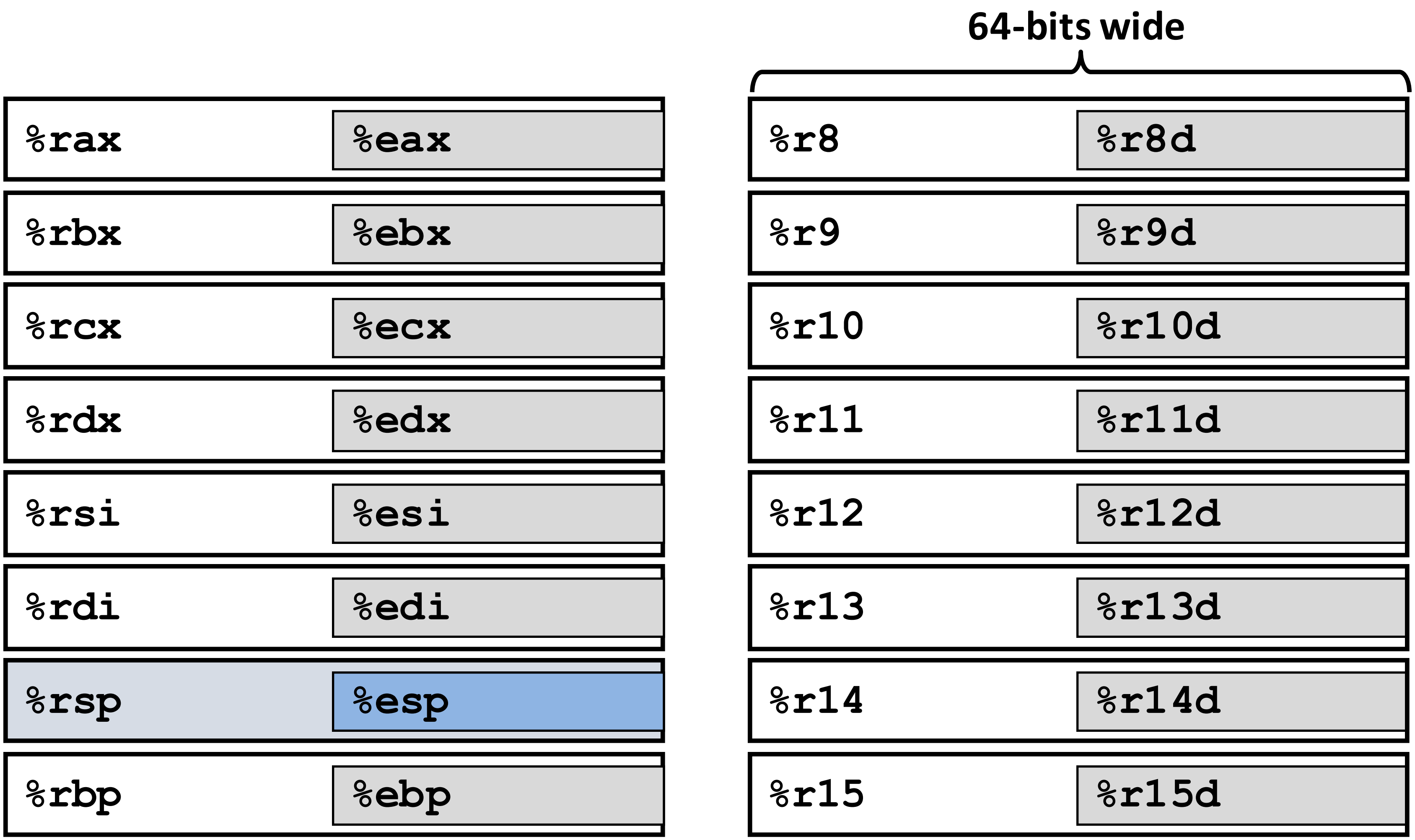
- a.k.a. CS 240 review, upgrade to 64 bits
- Focus on specific recurring details we need to get right.

- Calling Conventions
- Memory addressing
  - Field access
  - Array indexing
- Wacky instructions
  - Division
  - Store absolute address
  - setCC and movzbq

x86 IA-32: registers



x86-64: more registers



Only %rsp is special-purpose.

Most 2-operand instructions

`movq` *Source*, *Dest*:

- Get argument(s) from *Source* (and *Dest* if, *e.g.*, arithmetic)
- Store result in *Dest*.
- Operand Types:
  - ***Immediate:*** Literal integer data, starts with $
    - Examples: **`$0x400`** **or** **`$-533`** **or** **`$foo`**
  - ***Register:*** One of 16 integer registers
    - Examples: **`%rax`** **or** **`%rsi`**
  - ***Memory:*** 8 consecutive bytes in memory, at address held by register
    - Simplest example: **`(%rax)`**
    - Various other “address modes”

Memory Addressing Modes

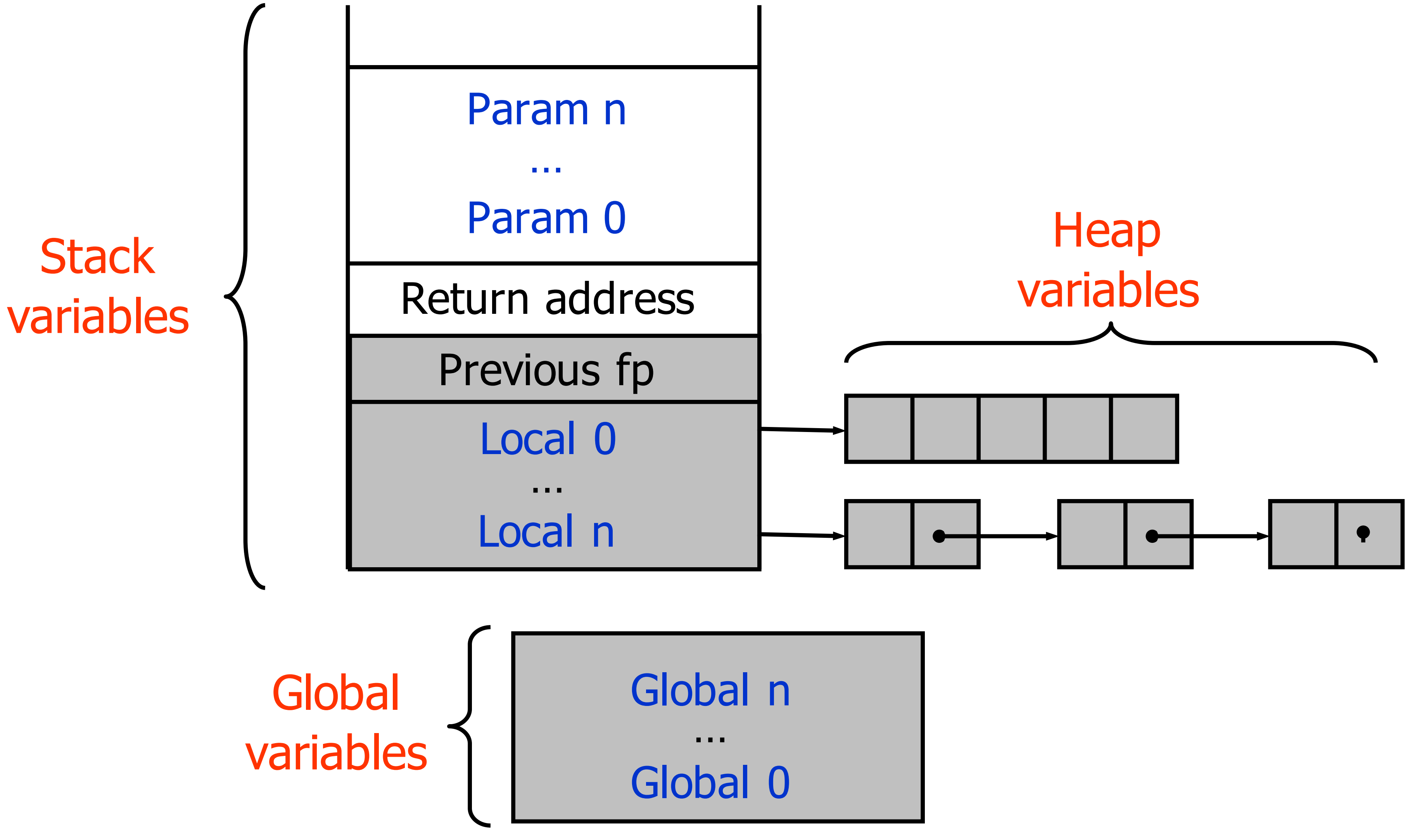
- General Form: **D(Rb,Ri,S)** Mem[Reg[Rb] + S*Reg[Ri] + D]
  - **D**: Displacement (offset): literal value represented in 1, 2, 4, or 8 bytes
  - **Rb**: Base register: Any register
  - **Ri**: Index register: Any register except `%rsp`
  - **S**: Scale: literal 1, 2, 4, or 8

- Special Cases: use any combination of D, Rb, Ri and S

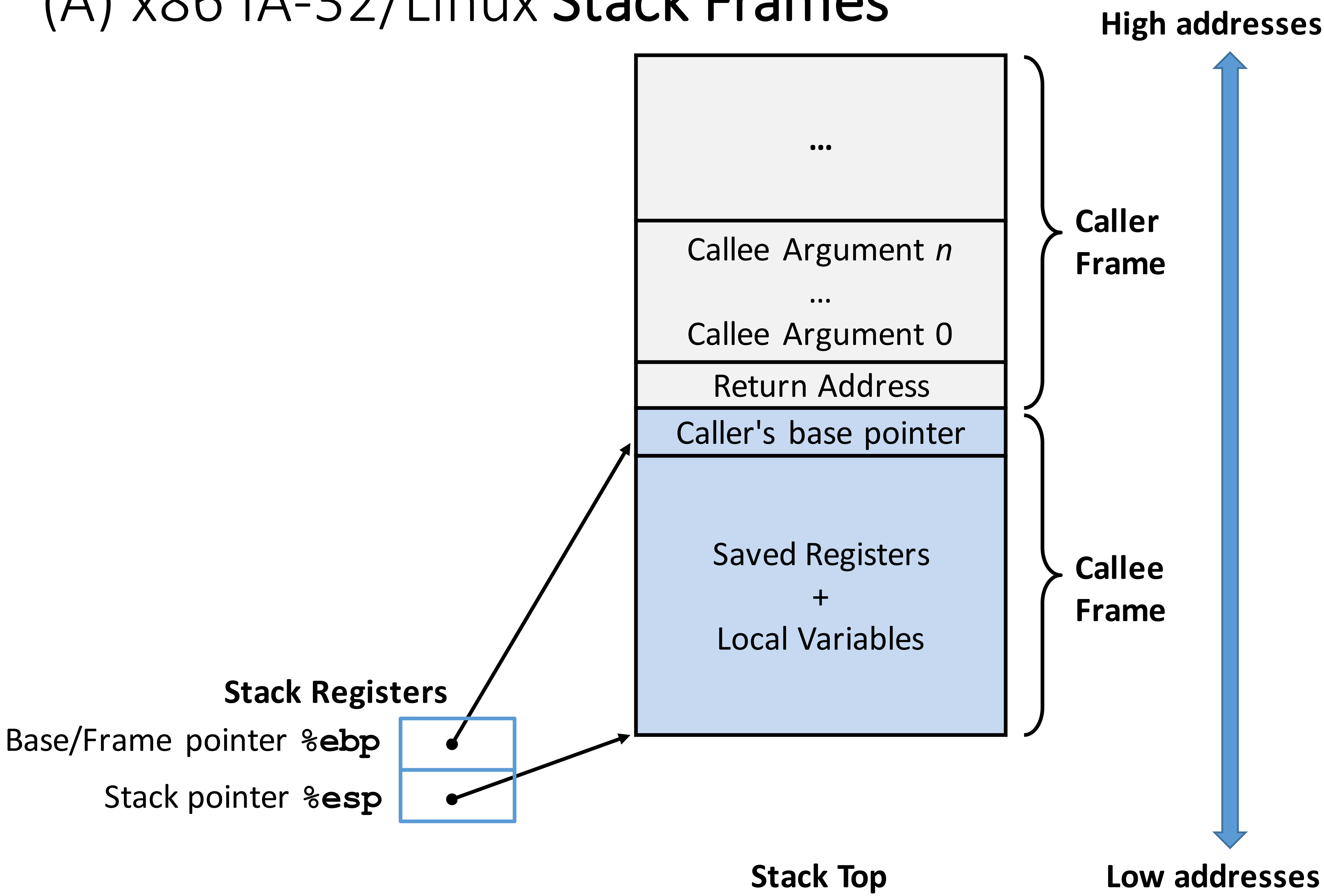
(Rb)	Mem[Reg[Rb]]	(Ri=0,S=1,D=0)
D(Rb)	Mem[Reg[Rb] + D]	(Ri=0,S=1)
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]	(D=0)
D(,Ri,S)	Mem[S*Reg[Ri]+D]	(Rb=0)

…

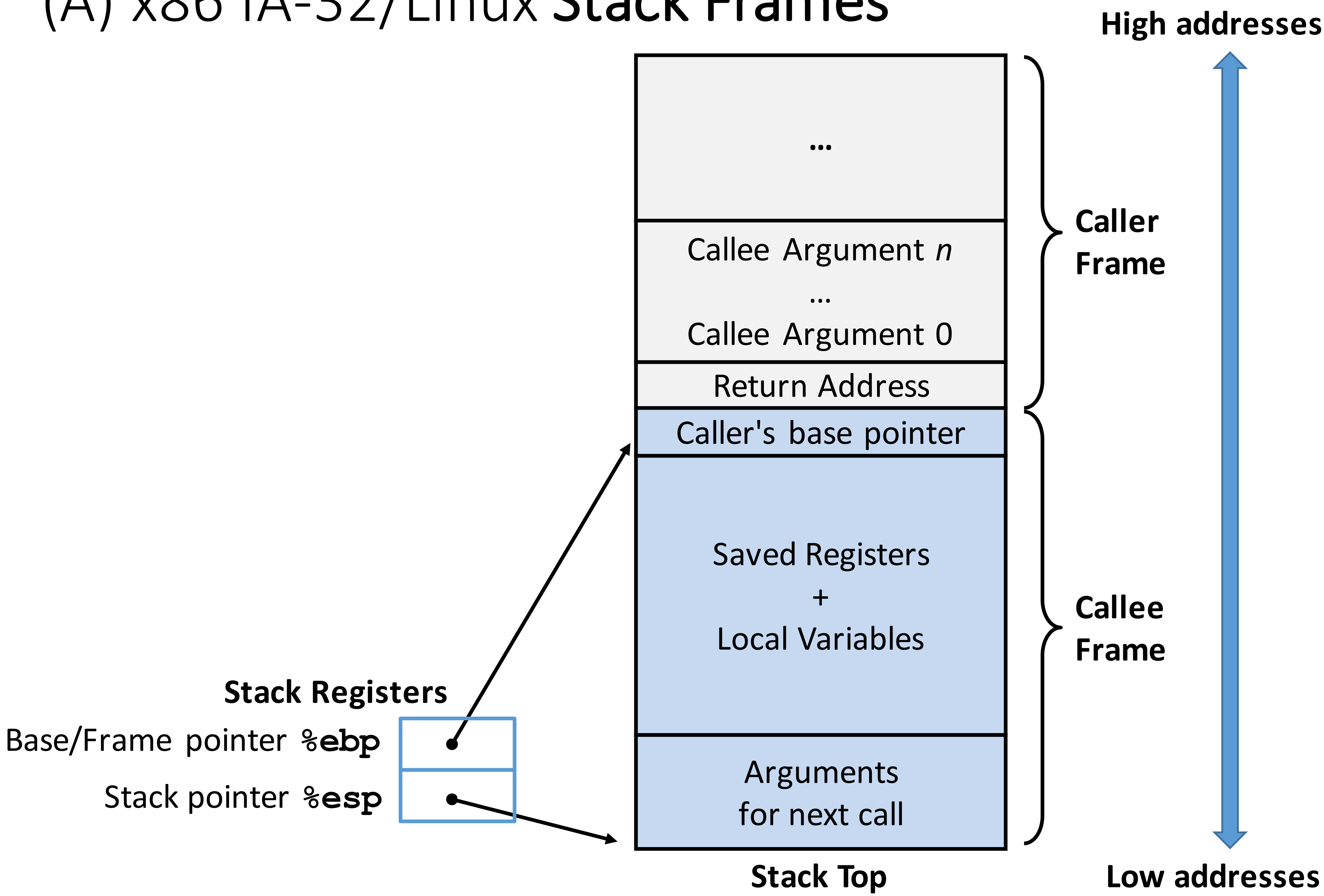
Big Picture: Memory Layout



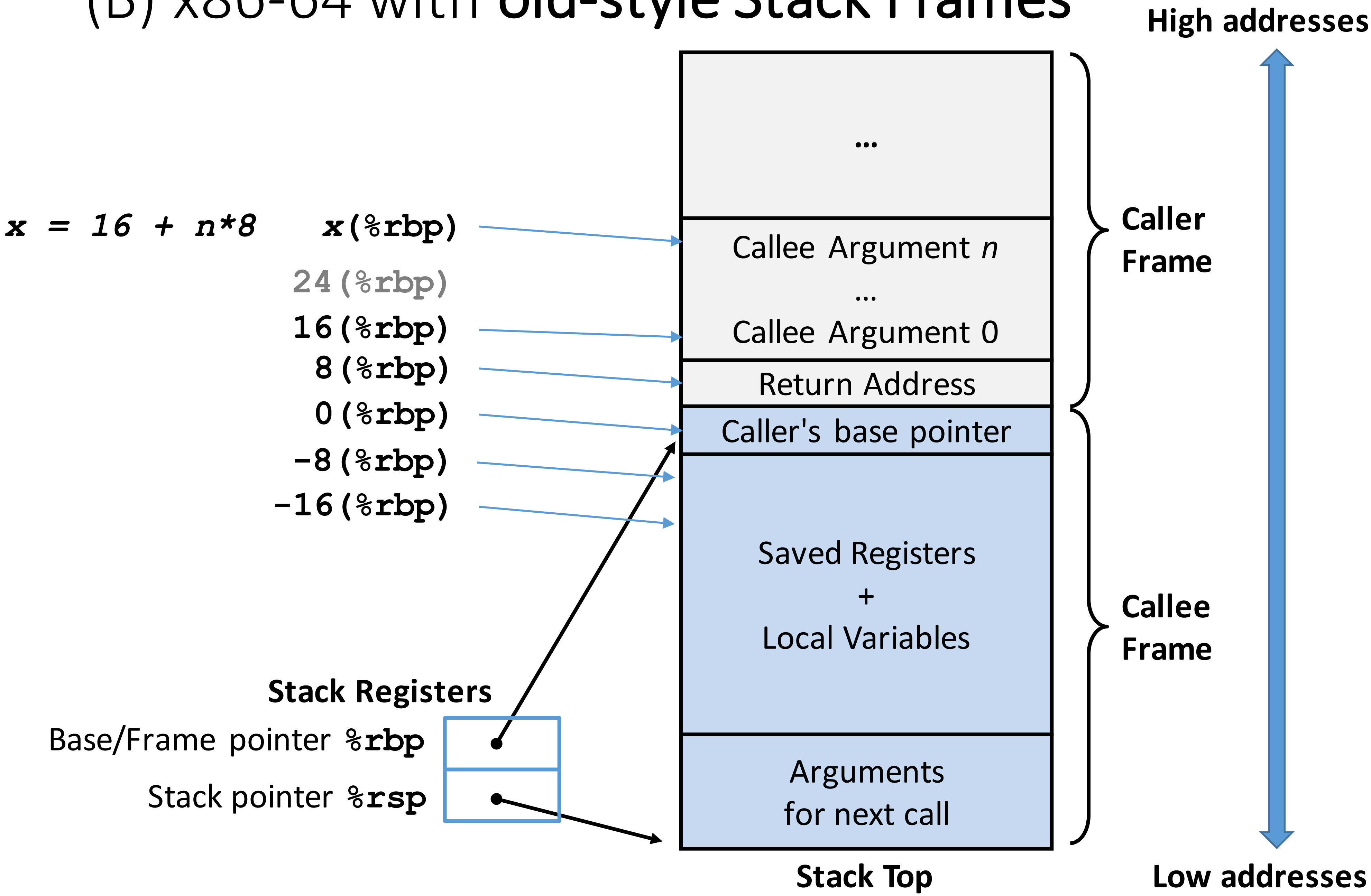
(A) x86 IA-32/Linux **Stack Frames**



(A) x86 IA-32/Linux **Stack Frames**



(B) x86-64 with **old-style Stack Frames**



(C) x86-64 with **new-style Stack Frames**

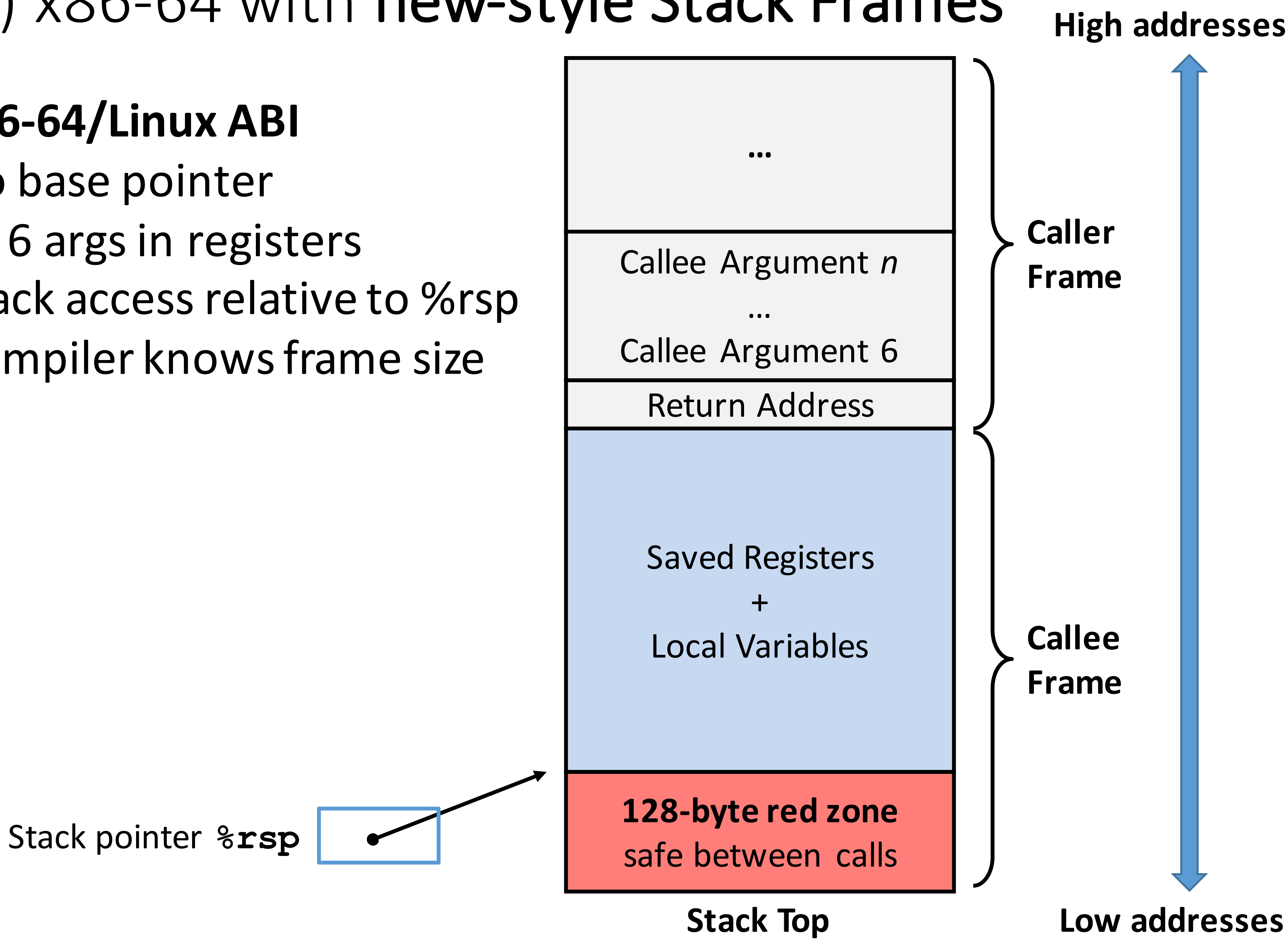
x86-64/Linux ABI

No base pointer

1st 6 args in registers

Stack access relative to %rsp

Compiler knows frame size



(C) Typical x86-64 **new-style Stack**

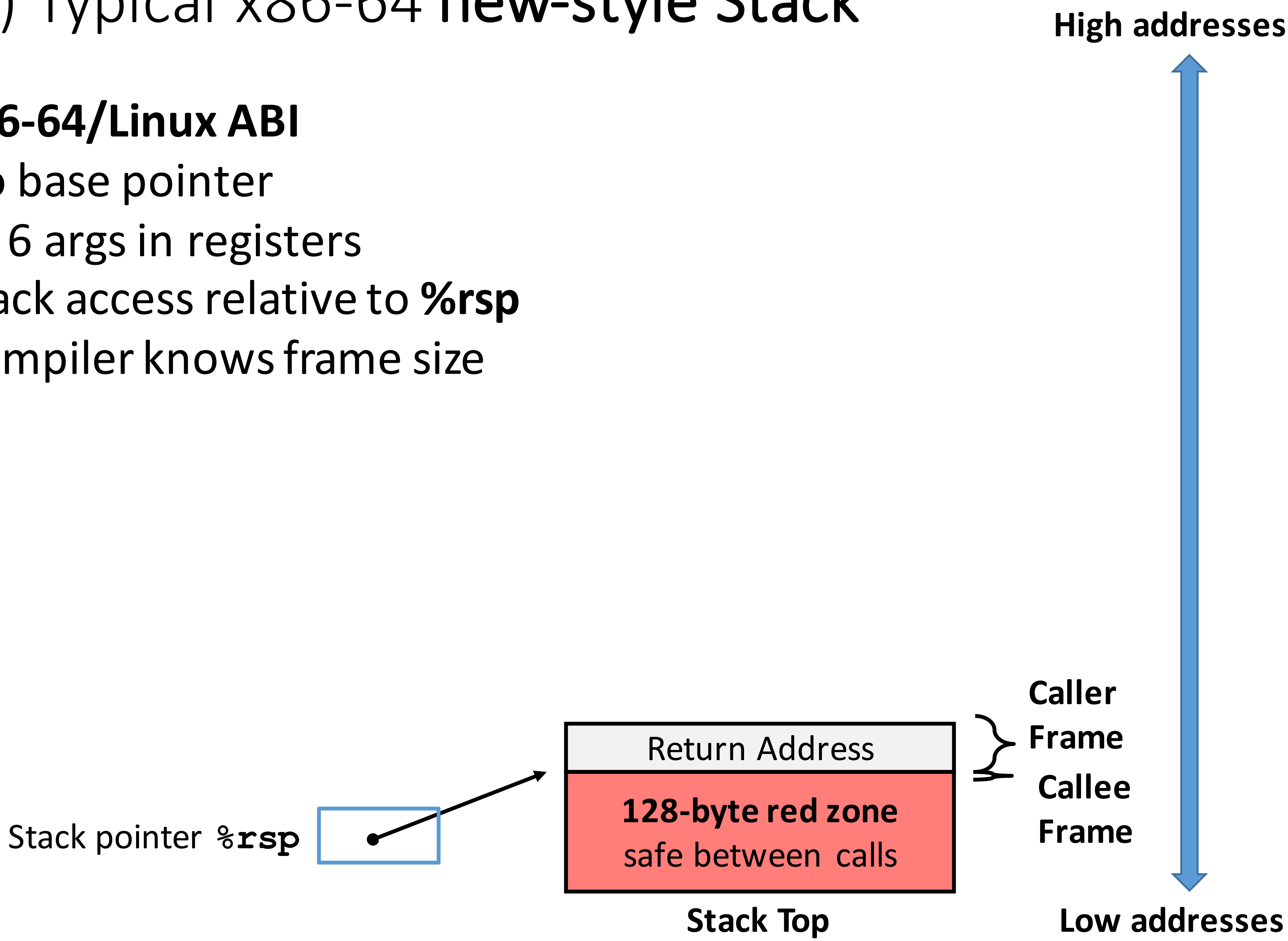
x86-64/Linux ABI

No base pointer

1st 6 args in registers

Stack access relative to **%rsp**

Compiler knows frame size



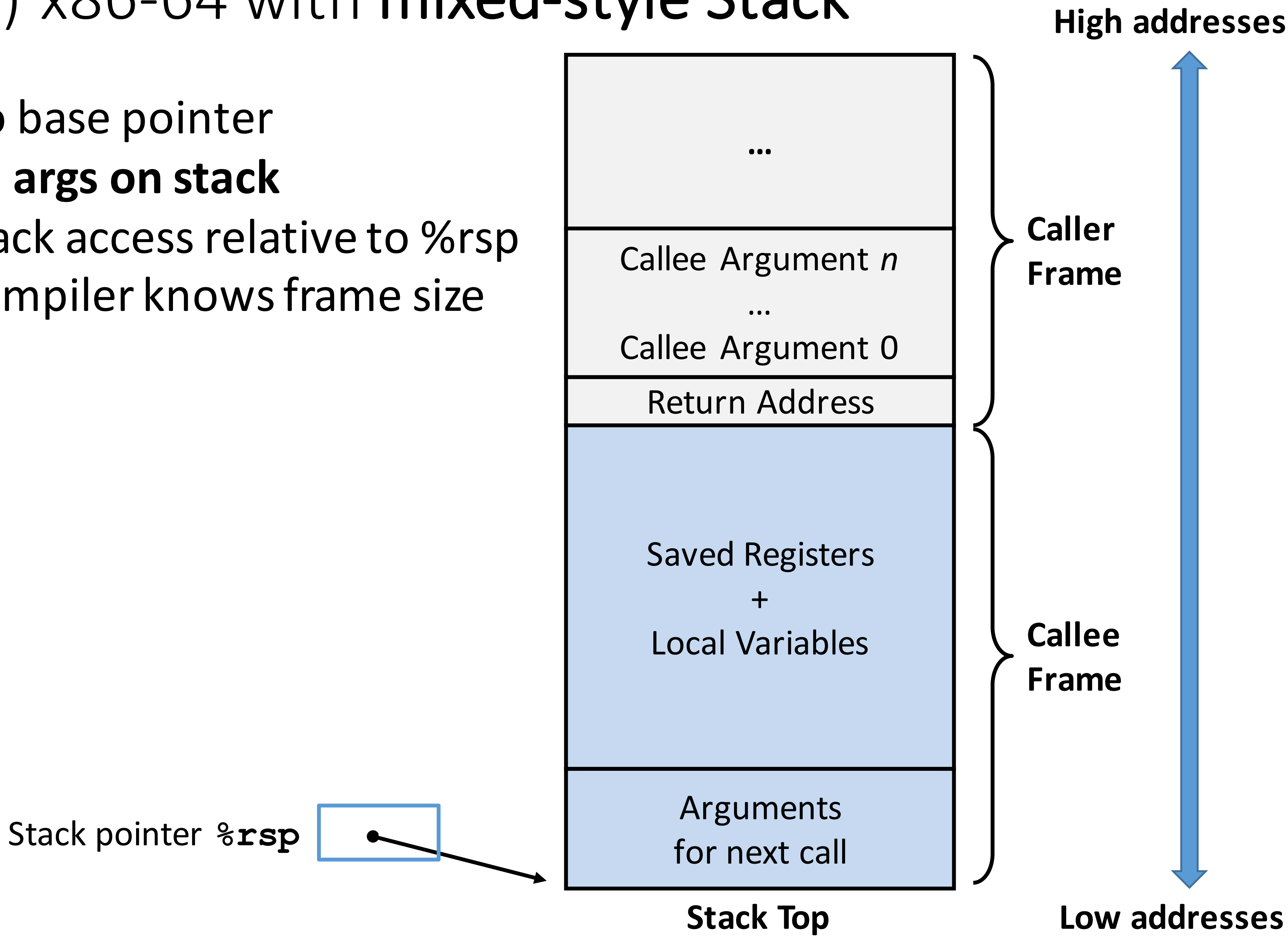
(D) x86-64 with **mixed-style Stack**

No base pointer

All args on stack

Stack access relative to %rsp

Compiler knows frame size



Saving Registers During Function Calls

- Problem: execution of callee may overwrite necessary values in registers

- Possibilities:
  - Callee saves and restores registers
  - Caller saves and restores registers
  - … or both

x86-64/Linux ABI: register conventions

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

Only %rsp is special-purpose.

ICC Calling Convention

- Always follow **x86-64/Linux register save convention**.

- To interface with **external code** (LIB), use:
  - **(C)** x86-64/Linux calling convention.

- To interface with other **ICC-generated code**, use one of:
  - **(B)** use frame pointer and stack pointer, all args on stack
    - Easiest, more work to convert if you convert to (C) later.
  - **(D)** use only stack pointer, all args on stack
    - Moderately easy, easier to convert to (C) later.
  - **(C)** x86-64/Linux calling convention
    - Harder, requires more register allocation work, more efficient, **only use this later if you have time.**

Example (*B*)

- Consider call **`foo(3, 5)`**:
  - **`%rcx`** caller-saved
  - **`%rbx`** callee-saved
  - result passed back in **`%rax`**

Save only the caller-save registers that are used after the call.

- Code before call instruction:

```
push %rcx        # push caller saved registers
push $5          # push second parameter
push $3          # push first parameter
call _foo        # push return address & jump to callee
```

- Prologue at start of function:

```
push %rbp        # push old fp
mov %rsp, %rbp   # compute new fp
sub $24, %rsp    # push 3 integer local variables
push %rbx        # push callee saved registers
```

Save only the callee-save registers that are overwritten in function

Example (*B*)

- Epilogue and end of function:

```
pop %rbx            # restore callee-saved registers
mov %rbp,%rsp       # pop callee frame, including locals
pop %rbp            # restore old fp
ret                 # pop return address and jump
```

- Code after call instruction:

```
add $16,%rsp        # pop parameters
pop %rcx            # restore caller-saved registers
                    # %rax contains return result
```

You are not likely to need to save/restore registers with the most basic code generation techniques.

Simple Code Generation (*D*)

- Three-address code makes it easy to generate assembly

e.g. `a = p+q` →

```
movq 16(%rsp), %rax
addq 8(%rsp), %rax
movq %rax, 24(%rsp)
```

- Need to consider many language constructs:
  - Operations: arithmetic, logic, comparisons
  - Accesses to local variables, global variables
  - Array accesses, field accesses
  - Control flow: conditional and unconditional jumps
  - Method calls, dynamic dispatch
  - Dynamic allocation (new)
  - Run-time checks

Division

```
movq …, %rcx # divisor, any reg. but %rax,%rdx
movq …, %rax # dividend
cqto         # sign-extend %rax into %rdx:%rax
idivq %rcx   # divide %rdx:%rax by %rcx
             # quotient in %rax
             # remainder in %rdx
```

String Literals, using calling convention (D)

```
.rodata
        ...
        .align 8
        .quad 13
strlit3:
        .ascii "Hello, World!"
        ...
.text
        ...
      # t4 = "Hello, World!"
      # Works on both LLVM/Mac OS X and GCC/Linux:
        leaq strlit3(%rip), %rax     # GCC only: movq $strlit3, %rax
        movq %rax, 8(%rsp)
      # Library.println(t4);
        movq 8(%rsp), %rax
        movq %rax, -8(%rsp)
        subq 8, %rsp
        callq __LIB_println
```

Method vectors/vtables and vtable pointer initialization will be similar.

`cmpq` and **`testq`**

`cmpq %rcx,%rax` computes **`%rax - %rcx`**, sets CF, OF SF, ZF, discards result

`testq %rax,%rcx` computes **`%rax & %rcx`**, sets SF, ZF, discards result

Flags/condition codes:

- CF: carry flag, 1 iff carry out
- OF: overflow flag, 1 iff signed overflow
- SF: sign flag, 1 iff result's MSB=1
- ZF: zero flag, 1 iff result=0

Common pattern to test for 0 or <0: **`testq %rax, %rax`**

`jmp` and `j`*CC*

Always jump: `jmp`

Jump iff *condition*: all other rows

jCC	Condition	Jump iff ...
`jmp`	`1`	**Unconditional**
`je, jz`	`ZF`	**Equal / Zero**
`jne, jnz`	`~ZF`	**Not Equal / Not Zero**
`jg`	`~(SF^OF)&~ZF`	**Greater** (Signed)
`jge`	`~(SF^OF)`	**Greater or Equal** (Signed)
`jl`	`(SF^OF)`	**Less** (Signed)
`jle`	`(SF^OF)|ZF`	**Less or Equal** (Signed)
`js`	**`SF`**	**Negative**
`jns`	**`~SF`**	**Nonnegative**
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

set*CC* and **movzbq**

```
# t7 = t4 <= t9
movq 72(%rsp), %rdx      # %rdx = t9
cmpq 32(%rsp), %rdx      # set flags: t9 - t4
setle %al              # set byte to 0x00 or 0x01
                       # based on condition le: <=
                       # as in %rdx <= %rcx
movzbq %al, %rax       # move, zero-extend byte to quad
                       # (Extend to 64 bits.)
movq %rax, 56(%rsp)      # t7 = result
```

Set has all the same flavors as conditional jump.

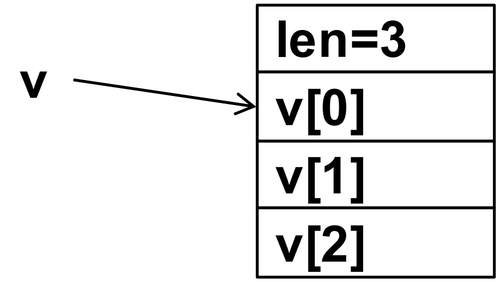
Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Allocation function returns address of allocated heap data
  - Access heap data through that reference

- Array accesses in Java
  - access **`a[i]`** requires:
    - computing address of element: **`a + i * size`**
    - accessing memory at that address
  - Indexed memory accesses do it all
  - Example: assume size of array elements is 8 bytes, and local variables a, i (offsets –8, -16)

```
a[i] = 1   ➡   mov -8(%rbp), %rbx          (load a)
               mov -16(%rbp), %rcx         (load i)
               mov $1, (%rbx,%rcx,8)       (store into the heap)
```

Run-time Checks



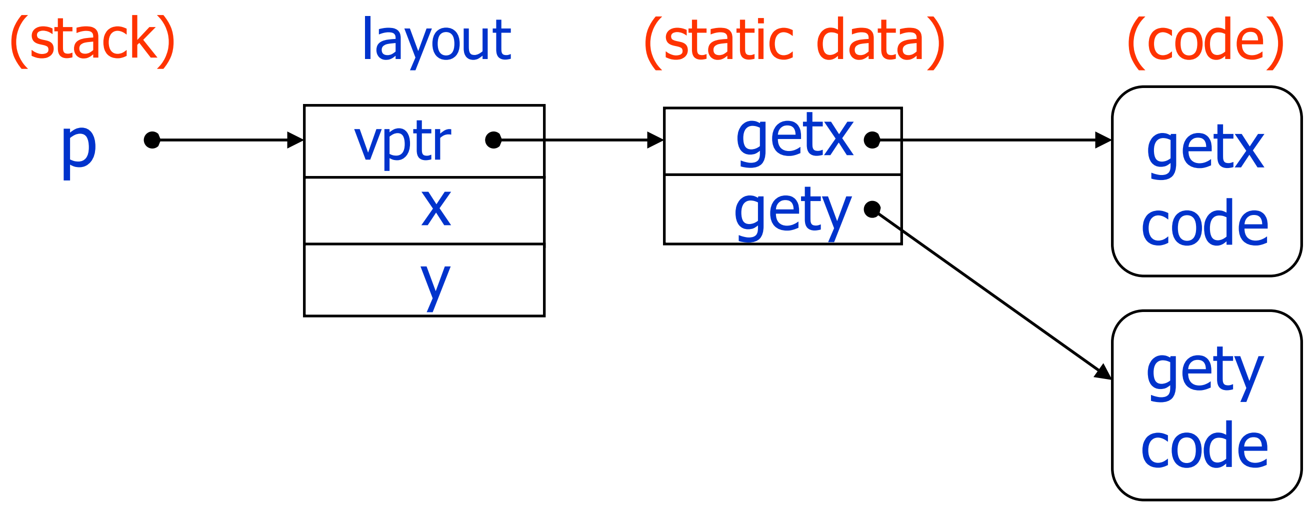
- Run-time checks:
  - Check if array/object references are non-null
  - Check if array index is within bounds

- Example: array bounds checks:
  - if v holds the address of an array, insert array bounds checking code for v before each load (…=v[i]) or store (v[i] = …)
  - Array length is stored just before array elements:

```
cmp $0, -24(%rbp)            (compare i to 0)
jl ArrayBoundsError          (test lower bound)
mov -16(%rbp), %rcx          (load v into %ecx)
mov -8(%rcx), %rcx           (load array length into %ecx)
cmp -24(%rbp), %rcx          (compare i to array length)
jle ArrayBoundsError         (test upper bound)
 ...
```

Object Layout

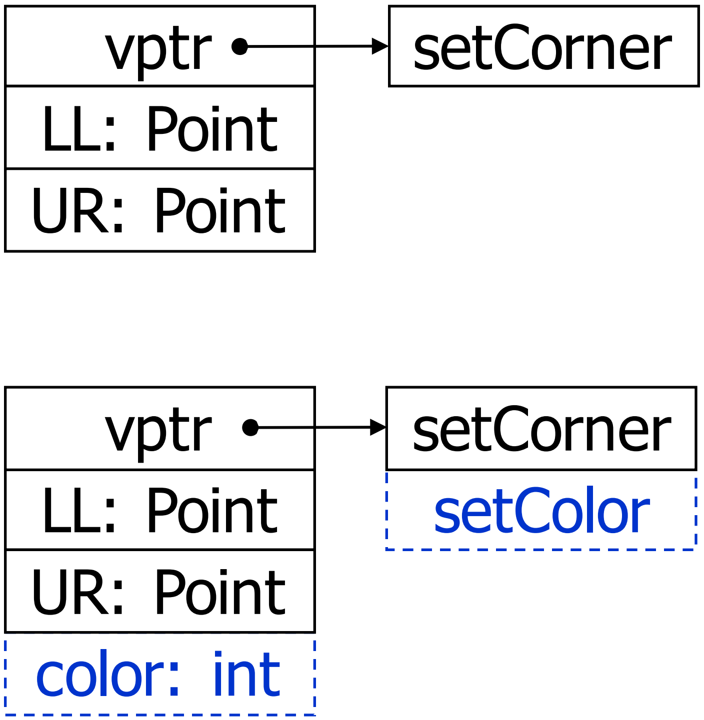
- Object consists of:
  - Methods
  - Fields
- Layout:
  - Pointer to VT, which contains pointers to methods
  - Fields.



Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses

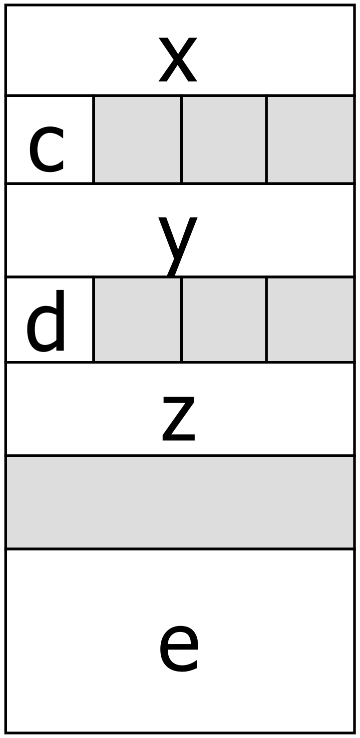
```
class Shape {
   Point LL /* 8 */ , UR; /* 16 */
   void setCorner(Point p);
}

class ColoredRect extends Shape {
   Color c; /* 24 */
   void setColor(Color c);
}
```



Field Alignment

- In many processors, a 32-bit load must be to an address divisible by 4, address of 64-bit load must be divisible by 8
- x86: unaligned access typically permitted, but slower
- Fields should be aligned

```
struct {
   int x;
   char c;
   int y;
   char d;
   int z; double e;
}
```



VTable Lookup

C <: B <: A

A f

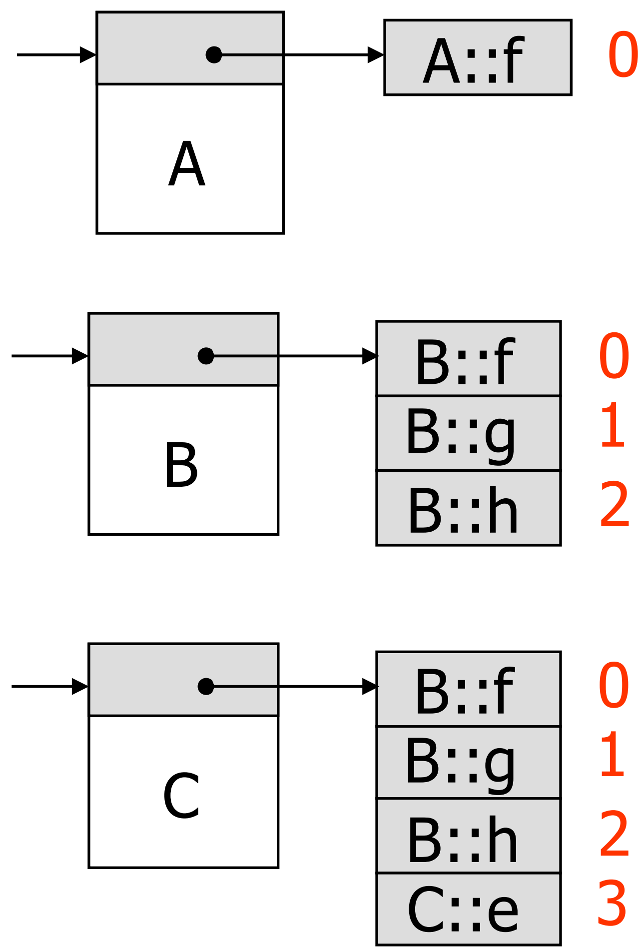
B f,g,h

C f,g,h,e

```
class A {
      void f() {...}   0
}
class B extends A {
      void f() {…}     0
      void g() {…}     1
      void h() {…}     2
}
class C extends B {
      void e() {…}     3
}
```

VTable Layouts

- Index of f is the same in any object of type T <: A
- To execute a method m:
    - Lookup entry m in vector
    - Execute code pointed to by entry value



Code Generation: Virtual Tables

- Statically allocate one vtable per class

```
.data
ListVT: .quad _List_first
        .quad _List_rest
        .quad _List_length
```

Method Arguments

- Receiver object is (implicit) argument to method

```
class A {
  int f(int x,
        int y)
       { … }
}
```

compile as

```
int f(A this,
      int x,
      int y)
      { … }
```

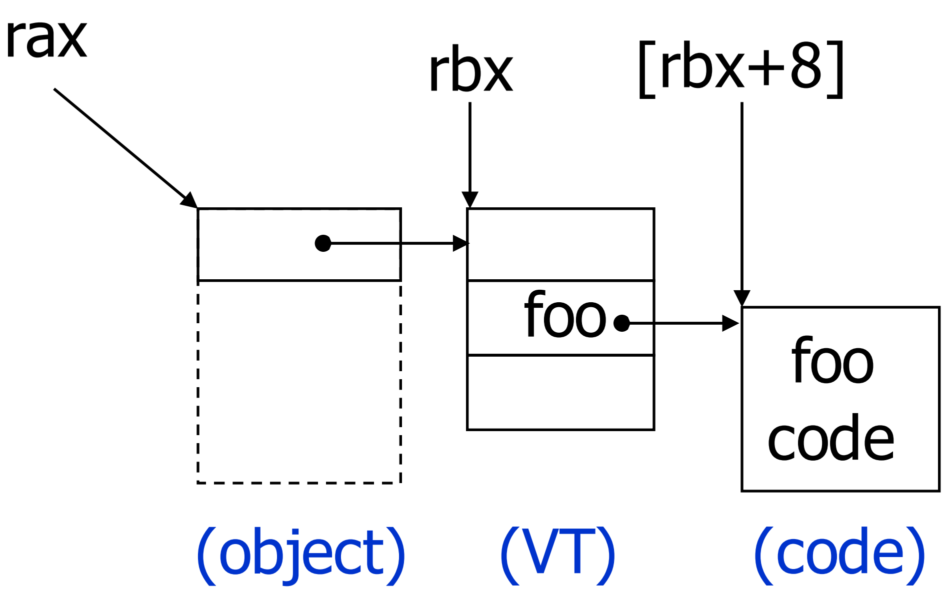
Code Generation: Method Calls

- Pre-function-call code:
  - Save registers
  - Push parameters
  - call function by its label

- Pre-method call:
  - Save registers
  - Push parameters
  - *Push receiver object reference*
  - *Lookup method in vtable*

Example

```
o.foo(2,3);
```




```
push $3
push $2
push %rax
mov (%rax), %rbx

call *8(%rbx)

add $24, %rsp
```

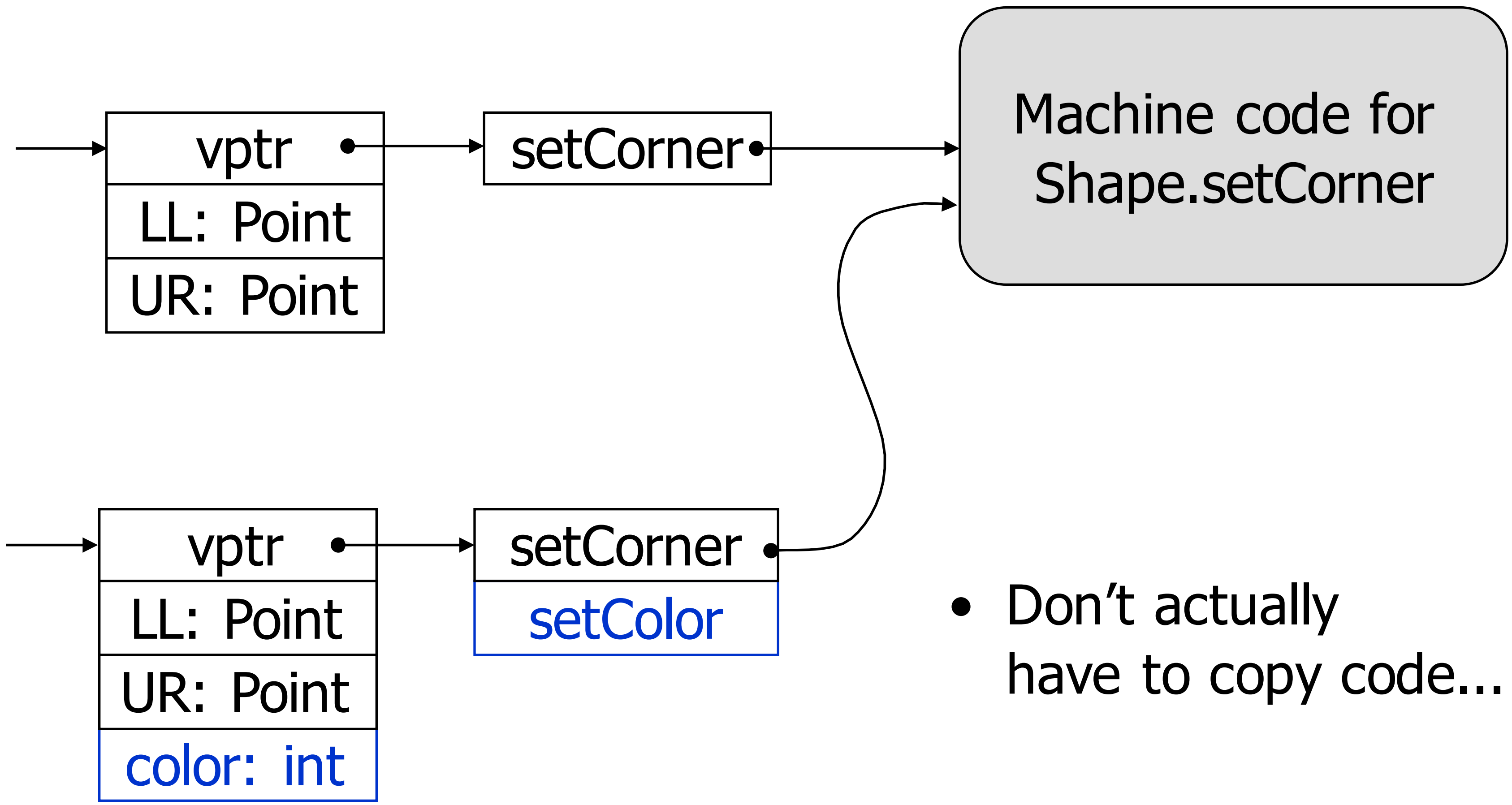
compiler knows offset of foo in table

Interfaces, Abstract Classes

- Interfaces
  - no implementation
  - no dispatch vector info
  - (slow lookup a la SmallTalk)

- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class
  - Can construct vtable- just leave abstract entries "blank"

Code Sharing



- Don't actually have to copy code...

Code Generation: Library Calls

- Pass params in registers
  - %rdi for first param
  - %rsi for second param
- Return result is in %rax
- Warning: library functions may modify caller save registers

```
movq $100, %rdi
call __LIB_printi

…

movq $20, %rdi
call __LIB_random
movq %rax, -32(%rbp)
```

Code Generation: Allocation

- Heap allocation: o = new C()
  - Allocate heap space for object
  - Store pointer to vtable into newly allocated memory

```
movq $32, %rdi # 3 fields + vptr
call __LIB_allocObject
leaq _C_VT(%rip), %rdi
movq %rdi, (%rax)
```